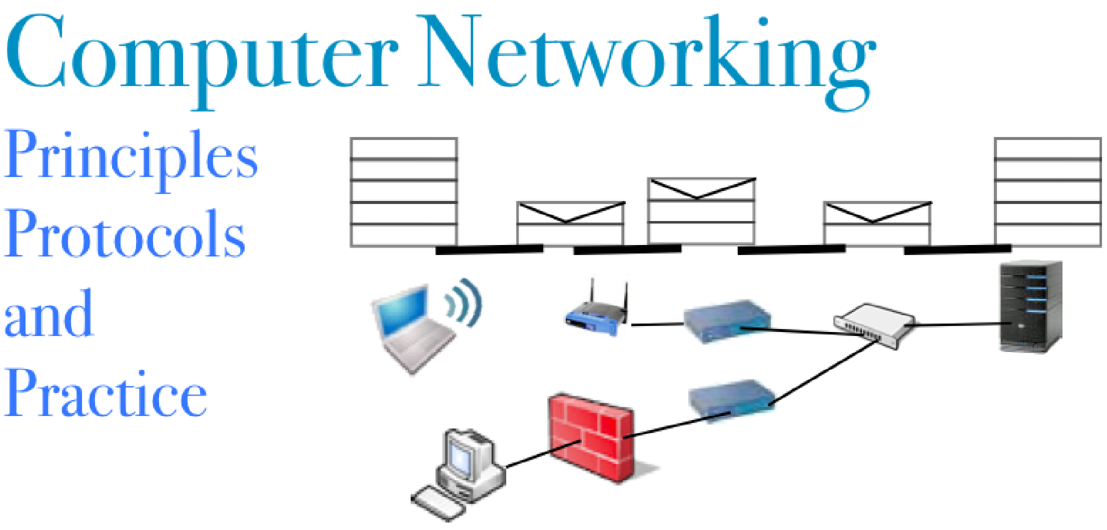


Computer Networking : Principles, Protocols and Practice

Release 2021

Olivier Bonaventure

Dec 21, 2021

CONTENTS

CHAPTER

ONE

PREFACE

This textbook came from a frustration of first main author. Many authors choose to write a textbook because there are no textbooks in their field or because they are not satisfied with the existing textbooks. This frustration has produced several excellent textbooks in the networking community. At a time when networking textbooks were mainly theoretical, Douglas Comer chose to write a textbook entirely focused on the TCP/IP protocol suite [Comer1988], a difficult choice at that time. He later extended his textbook by describing a complete TCP/IP implementation, adding practical considerations to the theoretical descriptions in [Comer1988]. Richard Stevens approached the Internet like an explorer and explained the operation of protocols by looking at all the packets that were exchanged on the wire [Stevens1994]. Jim Kurose and Keith Ross reinvented the networking textbooks by starting from the applications that the students use and later explained the Internet protocols by removing one layer after the other [KuroseRoss09].

The frustrations that motivated this book are different. When I started to teach networking in the late 1990s, students were already Internet users, but their usage was limited. Students were still using reference textbooks and spent time in the library. Today's students are completely different. They are avid and experimented web users who find lots of information on the web. This is a positive attitude since they are probably more curious than their predecessors. Thanks to the information that is available on the Internet, they can check or obtain additional information about the topics explained by their teachers. This abundant information creates several challenges for a teacher. Until the end of the nineteenth century, a teacher was by definition more knowledgeable than his students and it was very difficult for the students to verify the lessons given by their teachers. Today, given the amount of information available at the fingertips of each student through the Internet, verifying a lesson or getting more information about a given topic is sometimes only a few clicks away. Websites such as wikipedia provide lots of information on various topics and students often consult them. Unfortunately, the organisation of the information on these websites is not well suited to allow students to learn from them. Furthermore, there are huge differences in the quality and depth of the information that is available for different topics.

The second reason is that the computer networking community is a strong participant in the open-source movement. Today, there are high-quality and widely used open-source implementations for most networking protocols. This includes the TCP/IP implementations that are part of linux, freebsd or the uIP stack running on 8bits controllers, but also servers such as bind, unbound, apache or sendmail and implementations of routing protocols such as xorp or quagga . Furthermore, the documents that define almost all of the Internet protocols have been developed within the Internet Engineering Task Force (IETF) using an open process. The IETF publishes its protocol specifications in the publicly available RFC and new proposals are described in Internet drafts.

This open textbook aims to fill the gap between the open-source implementations and the open-source network specifications by providing a detailed but pedagogical description of the key principles that guide the operation of the Internet. The book is released under a creative commons licence. Such an open-source license is motivated by two reasons. The first is that we hope that this will allow many students to use the book to learn computer networks. The second is that I hope that other teachers will reuse, adapt and improve it. Time will tell if it is possible to build a community of contributors to improve and develop the book further. As a starting point, the first release contains all the material for a one-semester first upper undergraduate or a graduate networking course.

The first edition of this ebook has been written by Olivier Bonaventure. Laurent Vanbever, Virginie Van den Schriek, Damien Saucez and Mickael Hoerdt have contributed to exercises. Pierre Reinbold designed the icons used to represent switches and Nipaul Long has redrawn many figures in the SVG format. Stephane Bortzmeyer sent many suggestions and corrections to the text.

Over the years, students and colleagues contributed to parts of the text, including:

- Virginie Van den Schriek contributed to various exercises
- Laurent Vanbever contributed to various exercises
- Damien Saucez contributed to various exercises
- Mickael Hoerdt contributed to various exercises
- Pierre Reinbold designed the icons used to represent routers, switches, … and provided all the sysadmin support to host the book
- Nipaul Long converted most of the figures to SVG format
- Daire O'Doherty helped to improve the writing throughout the book
- Quentin De Coninck improved the text and exercises

The first and second versions of the e-book were developed on github. A lot of text for the third edition was part of the two previous editions. Here is the list of contributors to these two first editions:

- Alexis Nootens
- Antoine Paris
- Benoît Legat
- Daire O'Doherty
- David Lebrun
- Diego Havenstein
- Eduardo Grosclaude
- Florian Knop
- Mathieu Jadin
- Juan Antonio Cordero
- Joris Van Hecke
- Léonard Julement
- Laurent Lantsogh
- Laurent Vanbever
- Marcel Waldvogel
- Matthieu Baerts
- Melanie Sedda
- Mickael Hoerdt
- motateko
- Nicolas Pettiaux
- Nipaul Long
- Olivier Tilmans
- Pablo Gonzalez

- Raphael Bauduin
- Robin Descamps
- Hélène Verhaeghe
- Virginie Vandenschriek

The main contributors to the third edition were Olivier Bonaventure and Quentin De Coninck. Other contributions to this edition include:

- Adrien Defer
- Anthony Gégo
- François Michel
- Benjamin Caudron
- Mohamed Elshawaf
- Amadéo David
- Fabien Duchene
- Florent Dardenne
- Nicolas Rosar
- Gauthier de Moffarts
- Marcin Wilk

The entire source code for the ebook is available on https://github.com/CNP3/ebook If you spot any error, typo or want to improve the ebook, please add issues or suggest pull requests.

The HTML version of the ebook is available from https://www.computer-networking.info It includes various online exercises hosted on the https://www.inginious.org platform.

The ebook covers only a small subset of the *Computer Networking* domain. To encourage the readers to explore other aspects of this field, we regularly post pointers to relevant information on the *Networking Notes blog* at https://blog.computer-networking.info You can also follow us on twitter via @cnp3_ebook.

Note: *Computer Networking : Principles, Protocols and Practice*, (c) 2011-2021, Olivier Bonaventure, Universite catholique de Louvain (Belgium) and the collaborators listed above, used under a Creative Commons Attribution (CC BY) license made possible by funding from The Saylor Foundation's Open Textbook Challenge in order to be incorporated into Saylor.org' collection of open courses available at http://www.saylor.org. Full license terms may be viewed at : http://creativecommons.org/licenses/by/3.0/

CHAPTER

TWO

PART 1: PRINCIPLES

2.1 Connecting two hosts

The first step when building a network, even a worldwide network such as the Internet, is to connect two hosts together. This is illustrated in the figure below.

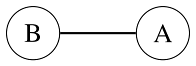


Fig. 1: Connecting two hosts together

To enable the two hosts to exchange information, they need to be linked together by some kind of physical media. Computer networks have used various types of physical media to exchange information, notably :

- *electrical cable*. Information can be transmitted over different types of electrical cables. The most common ones are the twisted pairs (that are used in the telephone network, but also in enterprise networks) and the coaxial cables (that are still used in cable TV networks, but are no longer used in enterprise networks). Some networking technologies operate over the classical electrical cable.
- *optical fiber*. Optical fibers are frequently used in public and enterprise networks when the distance between the communication devices is larger than one kilometer. There are two main types of optical fibers : multi-mode and single-mode. Multi-mode is much cheaper than single-mode fiber because a LED can be used to send a signal over a multi-mode fiber while a single-mode fiber must be driven by a laser. Due to the different modes of propagation of light, multi-mode fibers are limited to distances of a few kilometers while single-mode fibers can be used over distances greater than several tens of kilometers. In both cases, repeaters can be used to regenerate the optical signal at one endpoint of a fiber to send it over another fiber.
- *wireless*. In this case, a radio signal is used to encode the information exchanged between the communicating devices. Many types of modulation techniques are used to send information over a wireless channel and there is lot of innovation in this field with new techniques appearing every year. While most wireless networks rely on radio signals, some use a laser that sends light pulses to a remote detector. These optical techniques allow to create point-to-point links while radio-based techniques can be used to build networks containing devices spread over a small geographical area.

2.1.1 The physical layer

These physical media can be used to exchange information once this information has been converted into a suitable electrical signal. Entire telecommunication courses and textbooks are devoted to the problem of converting analog or digital information into an electrical signal so that it can be transmitted over a given physical *link*. In this book, we only consider two very simple schemes that allow to transmit information over an electrical cable. This enables us to highlight the key problems when transmitting information over a physical link. We are only interested in techniques that allow transmitting digital information through the wire. Here, we will focus on the transmission of bits, i.e. either *0* or *1*.

Note: Bit rate

In computer networks, the bit rate of the physical layer is always expressed in bits per second. One Mbps is one million bits per second and one Gbps is one billion bits per second. This is in contrast with memory specifications that are usually expressed in bytes (8 bits), KiloBytes (1024 bytes) or MegaBytes (1048576 bytes). Transferring one MByte through a 1 Mbps link lasts 8.39 seconds.

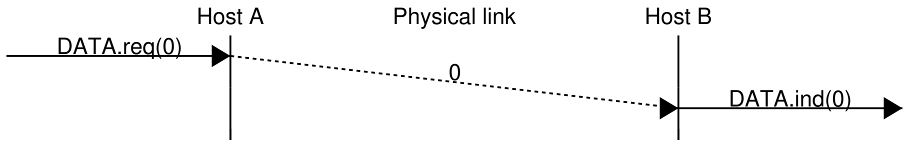
Bit rate	Bits per second
1 Kbps	10^3
1 Mbps	10^6
1 Gbps	10^9
1 Tbps	10^{12}

To understand some of the principles behind the physical transmission of information, let us consider the simple case of an electrical wire that is used to transmit bits. Assume that the two communicating hosts want to transmit one thousand bits per second. To transmit these bits, the two hosts can agree on the following rules :

- **On the sender side :**
  - set the voltage on the electrical wire at `+5V` during one millisecond to transmit a bit set to *1*
  - set the voltage on the electrical wire at `-5V` during one millisecond to transmit a bit set to *0*
- **On the receiver side :**
  - every millisecond, record the voltage applied on the electrical wire. If the voltage is set to `+5V`, record the reception of bit *1*. Otherwise, record the reception of bit *0*

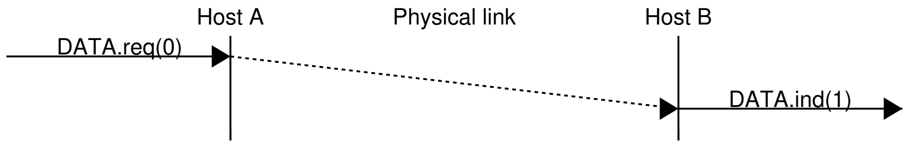
This transmission scheme has been used in some early networks. We use it as a basis to understand how hosts communicate. From a Computer Science viewpoint, dealing with voltages is unusual. Computer scientists frequently rely on models that enable them to reason about the issues that they face without having to consider all implementation details. The physical transmission scheme described above can be represented by using a *time-sequence diagram*.

A *time-sequence diagram* describes the interactions between communicating hosts. By convention, the communicating hosts are represented in the left and right parts of the diagram while the electrical link occupies the middle of the diagram. In such a time-sequence diagram, time flows from the top to the bottom of the diagram. The transmission of one bit of information is represented by three arrows. Starting from the left, the first horizontal arrow represents the request to transmit one bit of information. This request is represented by a *primitive* which can be considered as a kind of procedure call. This primitive has one parameter (the bit being transmitted) and a name (*DATA.request* in this example). By convention, all primitives that are named *something.request* correspond to a request to transmit some information. The dashed arrow indicates the transmission of the corresponding electrical signal on the wire. Electrical and optical signals do not travel instantaneously. The diagonal dashed arrow indicates that it takes some time for the electrical signal to be transmitted from *Host A* to *Host B*. Upon reception of the electrical signal, the electronics on *Host B*'s network interface detects the voltage and converts it into a bit. This bit is delivered as a *DATA.indication* primitive. All primitives that are named *something.indication* correspond to the reception of some information. The dashed lines also represents the relationship between two (or more) primitives. Such a time-sequence diagram provides information about the ordering of the different primitives, but the distance between two primitives does not represent a precise amount of time.

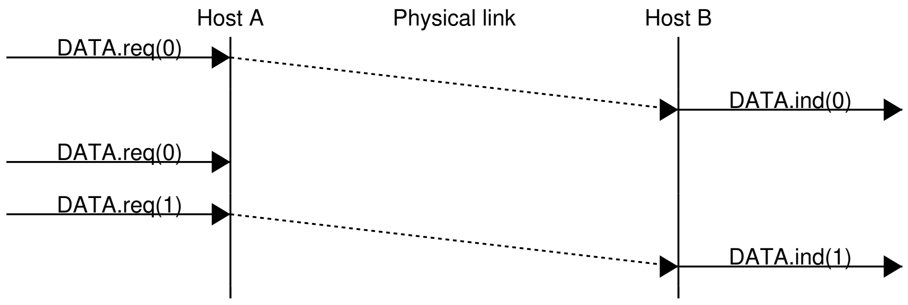


Time-sequence diagrams are useful when trying to understand the characteristics of a given communication scheme. When considering the above transmission scheme, it is useful to evaluate whether this scheme allows the two communicating hosts to reliably exchange information. A digital transmission is considered as reliable when a sequence of bits that is transmitted by a host is received correctly at the other end of the wire. In practice, achieving perfect reliability when transmitting information using the above scheme is difficult. Several problems can occur with such a transmission scheme.

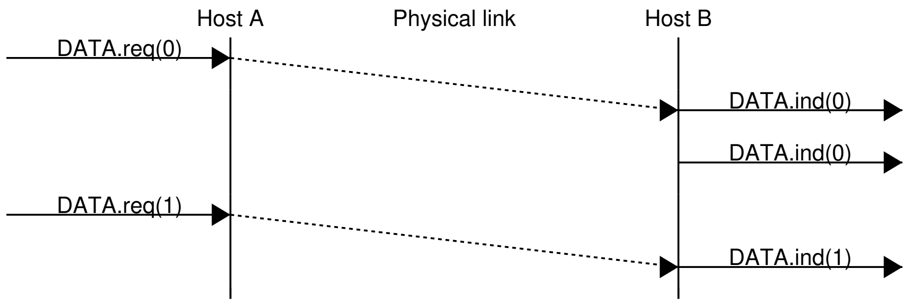
The first problem is that electrical transmission can be affected by electromagnetic interference. Interference can have various sources including natural phenomenons (like thunderstorms, variations of the magnetic field,. . .) but also other electrical signals (such as interference from neighboring cables, interference from neighboring antennas,. . .). Due to these various types of interference, there is unfortunately no guarantee that when a host transmit one bit on a wire, the same bit is received at the other end. This is illustrated in the figure below where a *DATA.request(0)* on the left host leads to a *Data.indication(1)* on the right host.



With the above transmission scheme, a bit is transmitted by setting the voltage on the electrical cable to a specific value during some period of time. We have seen that due to electromagnetic interference, the voltage measured by the receiver can differ from the voltage set by the transmitter. This is the main cause of transmission errors. However, this is not the only type of problem that can occur. Besides defining the voltages for bits *0* and *1*, the above transmission scheme also specifies the duration of each bit. If one million bits are sent every second, then each bit lasts 1 microsecond. On each host, the transmission (resp. the reception) of each bit is triggered by a local clock having a 1 MHz frequency. These clocks are the second source of problems when transmitting bits over a wire. Although the two clocks have the same specification, they run on different hosts, possibly at a different temperature and with a different source of energy. In practice, it is possible that the two clocks do not operate at exactly the same frequency. Assume that the clock of the transmitting host operates at exactly 1000000 Hz while the receiving clock operates at 999999 Hz. This is a very small difference between the two clocks. However, when using the clock to transmit bits, this difference is important. With its 1000000 Hz clock, the transmitting host will generate one million bits during a period of one second. During the same period, the receiving host will sense the wire 999999 times and thus will receive one bit less than the bits originally transmitted. This small difference in clock frequencies implies that bits can "disappear" during their transmission on an electrical cable. This is illustrated in the figure below.



A similar reasoning applies when the clock of the sending host is slower than the clock of the receiving host. In this case, the receiver will sense more bits than the bits that have been transmitted by the sender. This is illustrated in the figure below where the second bit received on the right was not transmitted by the left host.



From a Computer Science viewpoint, the physical transmission of information through a wire is often considered as a black box that allows transmitting bits. This black box is commonly referred to as the *physical layer service* and is represented by using the *DATA.request* and *DATA.indication* primitives introduced earlier. This physical layer service facilitates the sending and receiving of bits, by abstracting the technological details that are involved in the actual transmission of the bits as an electromagnetic signal. However, it is important to remember that the *physical layer service* is imperfect and has the following characteristics :

- the *Physical layer service* may change, e.g. due to electromagnetic interference, the value of a bit being transmitted
- the *Physical layer service* may deliver *more* bits to the receiver than the bits sent by the sender
- the *Physical layer service* may deliver *fewer* bits to the receiver than the bits sent by the sender

Many other types of encodings have been defined to transmit information over an electrical cable. All physical layers are able to send and receive physical symbols that represent values *0* and *1*. However, for various reasons that are outside the scope of this chapter, several physical layers exchange other physical symbols as well. For example, the Manchester encoding used in several physical layers can send four different symbols. The Manchester encoding is a differential encoding scheme in which time is divided into fixed-length periods. Each period is divided in two halves and two different voltage levels can be applied. To send a symbol, the sender must set one of these two voltage levels during each half period. To send a *1* (resp. *0*), the sender must set a high (resp. low) voltage during the first half of the period and a low (resp. high) voltage during the second half. This encoding ensures that there will be a transition at the middle of each period and allows the receiver to synchronize its clock to the sender's clock. Apart from the encodings for *0* and *1*, the Manchester encoding also supports two additional symbols : *InvH* and *InvB* where the same voltage level is used for the two half periods. By definition, these two symbols cannot appear inside a frame which is only composed of *0* and *1*. Some technologies use these special symbols as markers for the beginning or end of frames.

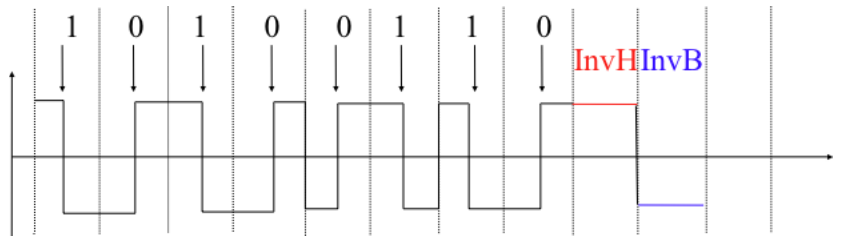


Fig. 2: Manchester encoding

All the functions related to the physical transmission or information through a wire (or a wireless link) are usually known as the *physical layer*. The physical layer allows thus two or more entities that are directly attached to the same transmission medium to exchange bits. Being able to exchange bits is important as virtually any information can be encoded as a sequence of bits. Electrical engineers are used to processing streams of bits, but computer scientists usually prefer to deal with higher level concepts. A similar issue arises with file storage. Storage devices such as hard-disks also store streams of bits. There are hardware devices that process the bit stream produced by a hard-disk,

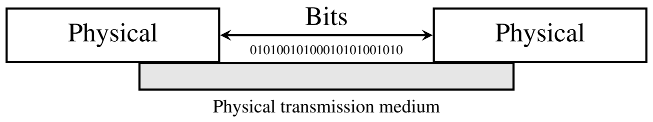


Fig. 3: The Physical layer

but computer scientists have designed filesystems to allow applications to easily access such storage devices. These filesystems are typically divided into several layers as well. Hard-disks store sectors of 512 bytes or more. Unix filesystems group sectors in larger blocks that can contain data or *inodes* representing the structure of the filesystem. Finally, applications manipulate files and directories that are translated in blocks, sectors and eventually bits by the operating system.

Computer networks use a similar approach. Each layer provides a service that is built above the underlying layer and is closer to the needs of the applications. The datalink layer builds upon the service provided by the physical layer. We will see that it also contains several functions.

2.1.2 The datalink layer

Computer scientists are usually not interested in exchanging bits between two hosts. They prefer to write software that deals with larger blocks of data in order to transmit messages or complete files. Thanks to the physical layer service, it is possible to send a continuous stream of bits between two hosts. This stream of bits can include logical blocks of data, but we need to be able to extract each block of data from the bit stream despite the imperfections of the physical layer. In many networks, the basic unit of information exchanged between two directly connected hosts is often called a *frame*. A *frame* can be defined as a sequence of bits that has a particular syntax or structure. We will see examples of such frames later in this chapter.

To enable the transmission/reception of frames, the first problem to be solved is how to encode a frame as a sequence of bits, so that the receiver can easily recover the received frame despite the limitations of the physical layer.

If the physical layer were perfect, the problem would be very simple. We would simply need to define how to encode each frame as a sequence of consecutive bits. The receiver would then easily be able to extract the frames from the received bits. Unfortunately, the imperfections of the physical layer make this framing problem slightly more complex. Several solutions have been proposed and are used in practice in different network technologies.

Framing

The *framing* problem can be defined as : "*How does a sender encode frames so that the receiver can efficiently extract them from the stream of bits that it receives from the physical layer*".

A first solution to this problem is to require the physical layer to remain idle for some time after the transmission of each frame. These idle periods can be detected by the receiver and serve as a marker to delineate frame boundaries. Unfortunately, this solution is not acceptable for two reasons. First, some physical layers cannot remain idle and always need to transmit bits. Second, inserting an idle period between frames decreases the maximum bit rate that can be achieved.

Note: Bit rate and bandwidth

Bit rate and bandwidth are often used to characterize the transmission capacity of the physical service. The original definition of bandwidth, as listed in the Webster dictionary is *a range of radio frequencies which is occupied by a modulated carrier wave, which is assigned to a service, or over which a device can operate*. This definition corresponds to the characteristics of a given transmission medium or receiver. For example, the human ear is able to decode sounds in roughly the 0-20 KHz frequency range. By extension, bandwidth is also used to represent the capacity of a communication system in bits per second. For example, a Gigabit Ethernet link is theoretically capable of transporting one billion bits per second.

Given that multi-symbol encodings cannot be used by all physical layers, a generic solution which can be used with any physical layer that is able to transmit and receive only bits *0* and *1* is required. This generic solution is called *stuffing* and two variants exist : *bit stuffing* and *character stuffing*. To enable a receiver to easily delineate the frame boundaries, these two techniques reserve special bit strings as frame boundary markers and encode the frames so that these special bit strings do not appear inside the frames.

Bit stuffing reserves the *01111110* bit string as the frame boundary marker and ensures that there will never be six consecutive *1* symbols transmitted by the physical layer inside a frame. With bit stuffing, a frame is sent as follows. First, the sender transmits the marker, i.e. *01111110*. Then, it sends all the bits of the frame and inserts an additional bit set to *0* after each sequence of five consecutive *1* bits. This ensures that the sent frame never contains a sequence of six consecutive bits set to *1*. As a consequence, the marker pattern cannot appear inside the frame sent. The marker is also sent to mark the end of the frame. The receiver performs the opposite to decode a received frame. It first detects the beginning of the frame thanks to the *01111110* marker. Then, it processes the received bits and counts the number of consecutive bits set to *1*. If a *0* follows five consecutive bits set to *1*, this bit is removed since it was inserted by the sender. If a *1* follows five consecutive bits sets to *1*, it indicates a marker if it is followed by a bit set to *0*. The table below illustrates the application of bit stuffing to some frames.

Original frame	Transmitted frame
0001001001001001001000011	01111110000100100100100100100001101111110
0110111111111111111110010	01111110011011111011111011111011001001111110
0111110	011111100111110001111110
01111110	0111111001111101001111110

For example, consider the transmission of *0110111111111111111110010*. The sender will first send the *01111110* marker followed by *011011111*. After these five consecutive bits set to *1*, it inserts a bit set to *0* followed by *11111*. A new *0* is inserted, followed by *11111*. A new *0* is inserted followed by the end of the frame *110010* and the *01111110* marker.

Bit stuffing increases the number of bits required to transmit each frame. The worst case for bit stuffing is of course a long sequence of bits set to *1* inside the frame. If transmission errors occur, stuffed bits or markers can be in error. In these cases, the frame affected by the error and possibly the next frame will not be correctly decoded by the receiver, but it will be able to resynchronize itself at the next valid marker.

Bit stuffing can be easily implemented in hardware. However, implementing it in software is difficult given the complexity of performing bit manipulations in software. Software implementations prefer to process characters than bits, software-based datalink layers usually use *character stuffing*. This technique operates on frames that contain an integer number of characters. In computer networks, characters are usually encoded by relying on the *ASCII* table. This table defines the encoding of various alphanumeric characters as a sequence of bits. **RFC 20** provides the ASCII table that is used by many protocols on the Internet. For example, the table defines the following binary representations :

- *A* : *1000011* b
- *0* : *0110000* b
- *z* : *1111010* b
- @ : *1000000* b
- *space* : *0100000* b

In addition, the *ASCII* table also defines several non-printable or control characters. These characters were designed to allow an application to control a printer or a terminal. These control characters include *CR* and *LF*, that are used to terminate a line, and the *BEL* character which causes the terminal to emit a sound.

- *NUL*: *0000000* b

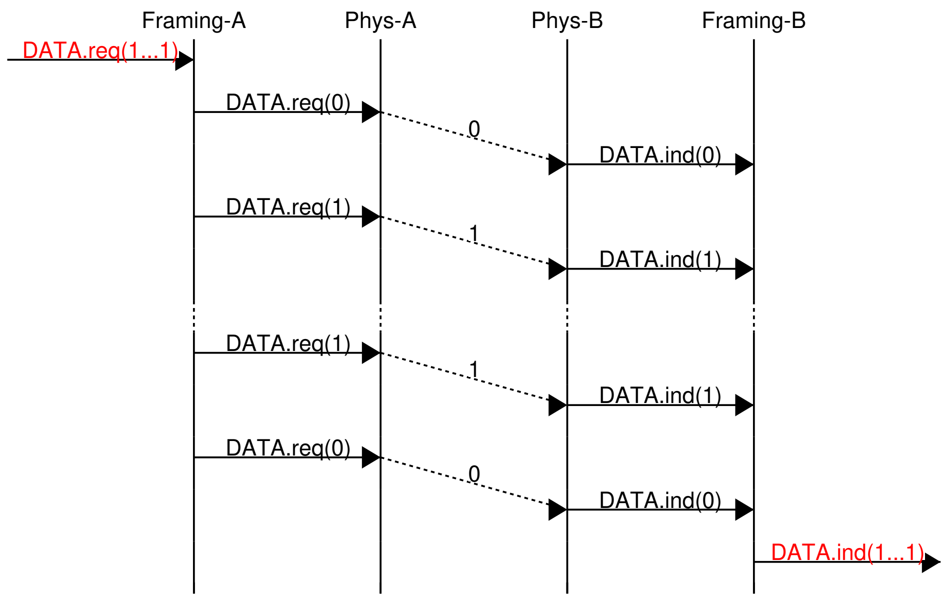
- *BEL*: *0000111* b
- *CR* : *0001101* b
- *LF* : *0001010* b
- *DLE*: *0010000* b
- *STX*: *0000010* b
- *ETX*: *0000011* b

Some characters are used as markers to delineate the frame boundaries. Many *character stuffing* techniques use the *DLE*, *STX* and *ETX* characters of the ASCII character set. *DLE STX* (resp. *DLE ETX*) is used to mark the beginning (end) of a frame. When transmitting a frame, the sender adds a *DLE* character after each transmitted *DLE* character. This ensures that none of the markers can appear inside the transmitted frame. The receiver detects the frame boundaries and removes the second *DLE* when it receives two consecutive *DLE* characters. For example, to transmit frame *1 2 3 DLE STX 4*, a sender will first send *DLE STX* as a marker, followed by *1 2 3 DLE*. Then, the sender transmits an additional *DLE* character followed by *STX 4* and the *DLE ETX* marker.

Original frame	Transmitted frame
1 2 3 4	*DLE STX* **1 2 3 4** *DLE ETX*
1 2 3 DLE STX 4	*DLE STX* **1 2 3 DLE** *DLE* **STX** *4 DLE ETX*
DLE STX DLE ETX	*DLE STX* **DLE** *DLE* **STX DLE** *DLE* ETX** *DLE ETX*

Character stuffing, like bit stuffing, increases the length of the transmitted frames. For *character stuffing*, the worst frame is a frame containing many *DLE* characters. When transmission errors occur, the receiver may incorrectly decode one or two frames (e.g. if the errors occur in the markers). However, it will be able to resynchronize itself with the next correctly received markers.

Bit stuffing and character stuffing allow recovering frames from a stream of bits or bytes. This framing mechanism provides a richer service than the physical layer. Through the framing service, one can send and receive complete frames. This framing service can also be represented by using the *DATA.request* and *DATA.indication* primitives. This is illustrated in the figure below, assuming hypothetical frames containing four useful bits and one bit of framing for graphical reasons.



We can now build upon the framing mechanism to allow the hosts to exchange frames containing an integer number of bits or bytes. Once the framing problem has been solved, we can focus on designing a technique that allows reliably exchanging frames.

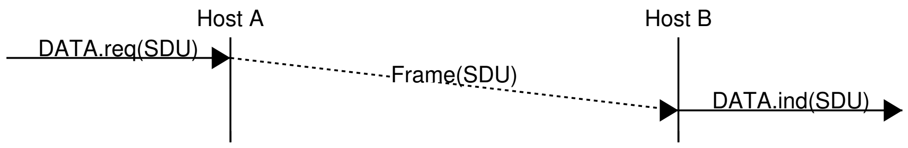
Recovering from transmission errors

In this section, we develop a reliable datalink protocol running above the physical layer service. To design this protocol, we first assume that the physical layer provides a perfect service. We will then develop solutions to recover from the transmission errors.

The datalink layer is designed to send and receive frames on behalf of a user. We model these interactions by using the *DATA.req* and *DATA.ind* primitives. However, to simplify the presentation and to avoid confusion between a *DATA.req* primitive issued by the user of the datalink layer entity, and a *DATA.req* issued by the datalink layer entity itself, we will use the following terminology :

- the interactions between the user and the datalink layer entity are represented by using the classical *DATA.req* and the *DATA.ind* primitives
- the interactions between the datalink layer entity and the framing sub-layer are represented by using *send* instead of *DATA.req* and *recvd* instead of *DATA.ind*

When running on top of a perfect framing sub-layer, a datalink entity can simply issue a *send(SDU)* upon arrival of a *DATA.req(SDU)*[1]. Similarly, the receiver issues a *DATA.ind(SDU)* upon receipt of a *recvd(SDU)*. Such a simple protocol is sufficient when a single SDU is sent. This is illustrated in the figure below.



Unfortunately, this is not always sufficient to ensure a reliable delivery of the SDUs. Consider the case where a client sends tens of SDUs to a server. If the server is faster than the client, it will be able to receive and process all the frames sent by the client and deliver their content to its user. However, if the server is slower than the client, problems may arise. The datalink entity contains buffers to store SDUs that have been received as a *Data.request* but have not yet been sent. If the application is faster than the physical link, the buffer may become full. At this point, the operating system suspends the application to let the datalink entity empty its transmission queue. The datalink entity also uses a buffer to store the received frames that have not yet been processed by the application. If the application is slow to process the data, this buffer may overflow and the datalink entity will not able to accept any additional frame. The buffers of the datalink entity have a limited size and if they overflow, the arriving frames will be discarded, even if they are correct.

To solve this problem, a reliable protocol must include a feedback mechanism that allows the receiver to inform the sender that it has processed a frame and that another one can be sent. This feedback is required even though there are no transmission errors. To include such a feedback, our reliable protocol must process two types of frames :

- data frames carrying a SDU
- control frames carrying an acknowledgment indicating that the previous frames was correctly processed

These two types of frames can be distinguished by dividing the frame in two parts :

- the *header* that contains one bit set to *0* in data frames and set to *1* in control frames
- the payload that contains the SDU supplied by the application

The datalink entity can then be modeled as a finite state machine, containing two states for the receiver and two states for the sender. The figure below provides a graphical representation of this state machine with the sender above and the receiver below.

[1] SDU is the acronym of Service Data Unit. We use it as a generic term to represent the data that is transported by a protocol.

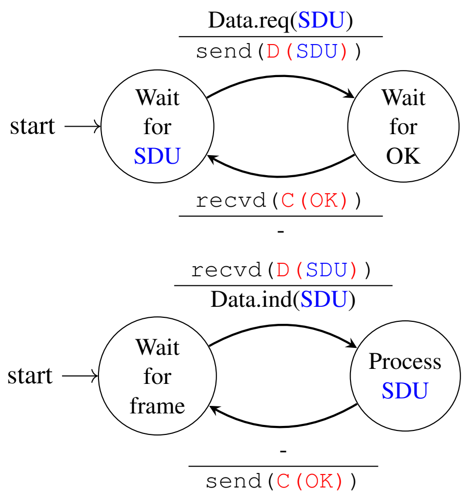
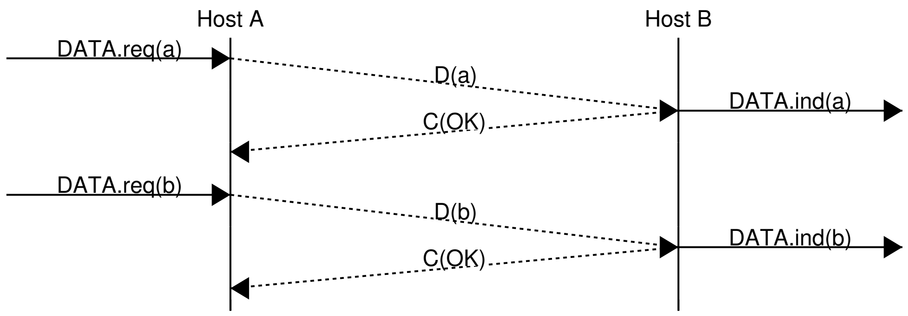


Fig. 4: Finite state machines of the simplest reliable protocol (sender above, receiver below)

The above FSM shows that the sender has to wait for an acknowledgment from the receiver before being able to transmit the next SDU. The figure below illustrates the exchange of a few frames between two hosts.



Note: Services and protocols

An important aspect to understand before studying computer networks is the difference between a *service* and a *protocol*. For this, it is useful to start with real world examples. The traditional Post provides a service where a postman delivers letters to recipients. The Post precisely defines which types of letters (size, weight, etc) can be delivered by using the Standard Mail service. Furthermore, the format of the envelope is specified (position of the sender and recipient addresses, position of the stamp). Someone who wants to send a letter must either place the letter at a Post Office or inside one of the dedicated mailboxes. The letter will then be collected and delivered to its final recipient. Note that for the regular service the Post usually does not guarantee the delivery of each particular letter. Some letters may be lost, and some letters are delivered to the wrong mailbox. If a letter is important, then the sender can use the registered service to ensure that the letter will be delivered to its recipient. Some Post services also provide an acknowledged service or an express mail service that is faster than the regular service.

Reliable data transfer on top of an imperfect link

The datalink layer must deal with the transmission errors. In practice, we mainly have to deal with two types of errors in the datalink layer :

- Frames can be corrupted by transmission errors
- Frames can be lost or unexpected frames can appear

A first glance, loosing frames might seem strange on a single link. However, if we take framing into account, transmission errors can affect the frame delineation mechanism and make the frame unreadable. For the same reason, a receiver could receive two (likely invalid) frames after a sender has transmitted a single frame.

To deal with these types of imperfections, reliable protocols rely on different types of mechanisms. The first problem is transmission errors. Data transmission on a physical link can be affected by the following errors :

- random isolated errors where the value of a single bit has been modified due to a transmission error
- random burst errors where the values of *n* consecutive bits have been changed due to transmission errors
- random bit creations and random bit removals where bits have been added or removed due to transmission errors

The only solution to protect against transmission errors is to add redundancy to the frames that are sent. *Information Theory* defines two mechanisms that can be used to transmit information over a transmission channel affected by random errors. These two mechanisms add redundancy to the transmitted information, to allow the receiver to detect or sometimes even correct transmission errors. A detailed discussion of these mechanisms is outside the scope of this chapter, but it is useful to consider a simple mechanism to understand its operation and its limitations.

Information theory defines *coding schemes*. There are different types of coding schemes, but let us focus on coding schemes that operate on binary strings. A coding scheme is a function that maps information encoded as a string of *m* bits into a string of *n* bits. The simplest coding scheme is the (even) parity coding. This coding scheme takes an *m* bits source string and produces an *m+1* bits coded string where the first *m* bits of the coded string are the bits of the source string and the last bit of the coded string is chosen such that the coded string will always contain an even number of bits set to *1*. For example :

- *1001* is encoded as *10010*
- *1101* is encoded as *11011*

This parity scheme has been used in some RAMs as well as to encode characters sent over a serial line. It is easy to show that this coding scheme allows the receiver to detect a single transmission error, but it cannot correct it. However, if two or more bits are in error, the receiver may not always be able to detect the error.

Some coding schemes allow the receiver to correct some transmission errors. For example, consider the coding scheme that encodes each source bit as follows :

- *1* is encoded as *111*
- *0* is encoded as *000*

For example, consider a sender that sends *111*. If there is one bit in error, the receiver could receive *011* or *101* or *110*. In these three cases, the receiver will decode the received bit pattern as a *1* since it contains a majority of bits set to *1*. If there are two bits in error, the receiver will not be able anymore to recover from the transmission error.

This simple coding scheme forces the sender to transmit three bits for each source bit. However, it allows the receiver to correct single bit errors. More advanced coding systems that allow recovering from errors are used in several types of physical layers.

Besides framing, datalink layers also include mechanisms to detect and sometimes even recover from transmission errors. To allow a receiver to notice transmission errors, a sender must add some redundant information as an *error detection* code to the frame sent. This *error detection* code is computed by the sender on the frame that it transmits. When the receiver receives a frame with an error detection code, it recomputes it and verifies whether the received *error detection code* matches the computed *error detection code*. If they match, the frame is considered to be valid.

Many error detection schemes exist and entire books have been written on the subject. A detailed discussion of these techniques is outside the scope of this book, and we will only discuss some examples to illustrate the key principles.

To understand *error detection codes*, let us consider two devices that exchange bit strings containing *N* bits. To allow the receiver to detect a transmission error, the sender converts each string of *N* bits into a string of *N*+*r* bits. Usually, the *r* redundant bits are added at the beginning or the end of the transmitted bit string, but some techniques interleave redundant bits with the original bits. An *error detection code* can be defined as a function that computes the *r* redundant bits corresponding to each string of *N* bits. The simplest error detection code is the parity bit. There are two types of parity schemes : even and odd parity. With the *even* (resp. *odd*) parity scheme, the redundant bit is chosen so that an even (resp. odd) number of bits are set to *1* in the transmitted bit string of *N*+*r* bits. The receiver can easily recompute the parity of each received bit string and discard the strings with an invalid parity. The parity scheme is often used when 7-bit characters are exchanged. In this case, the eighth bit is often a parity bit. The table below shows the parity bits that are computed for bit strings containing three bits.

3 bits string	Odd parity	Even parity
000	1	0
001	0	1
010	0	1
100	0	1
111	0	1
110	1	0
101	1	0
011	1	0

The parity bit allows a receiver to detect transmission errors that have affected a single bit among the transmitted *N*+*r* bits. If there are two or more bits in error, the receiver may not necessarily be able to detect the transmission error. More powerful error detection schemes have been defined. The Cyclical Redundancy Checks (CRC) are widely used in datalink layer protocols. An N-bits CRC can detect all transmission errors affecting a burst of less than N bits in the transmitted frame and all transmission errors that affect an odd number of bits. Additional details about CRCs may be found in [Williams1993].

It is also possible to design a code that allows the receiver to correct transmission errors. The simplest *error correction code* is the triple modular redundancy (TMR). To transmit a bit set to *1* (resp. *0*), the sender transmits *111* (resp. *000*). When there are no transmission errors, the receiver can decode *111* as *1*. If transmission errors have affected a single bit, the receiver performs majority voting as shown in the table below. This scheme allows the receiver to correct all transmission errors that affect a single bit.

Received bits	Decoded bit
000	0
001	0
010	0
100	0
111	1
110	1
101	1
011	1

Other more powerful error correction codes have been proposed and are used in some applications. The Hamming Code is a clever combination of parity bits that provides error detection and correction capabilities.

Reliable protocols use error detection schemes, but none of the widely used reliable protocols rely on error correction schemes. To detect errors, a frame is usually divided into two parts :

- a *header* that contains the fields used by the reliable protocol to ensure reliable delivery. The header contains a checksum or Cyclical Redundancy Check (CRC) [Williams1993] that is used to detect transmission errors

- a *payload* that contains the user data

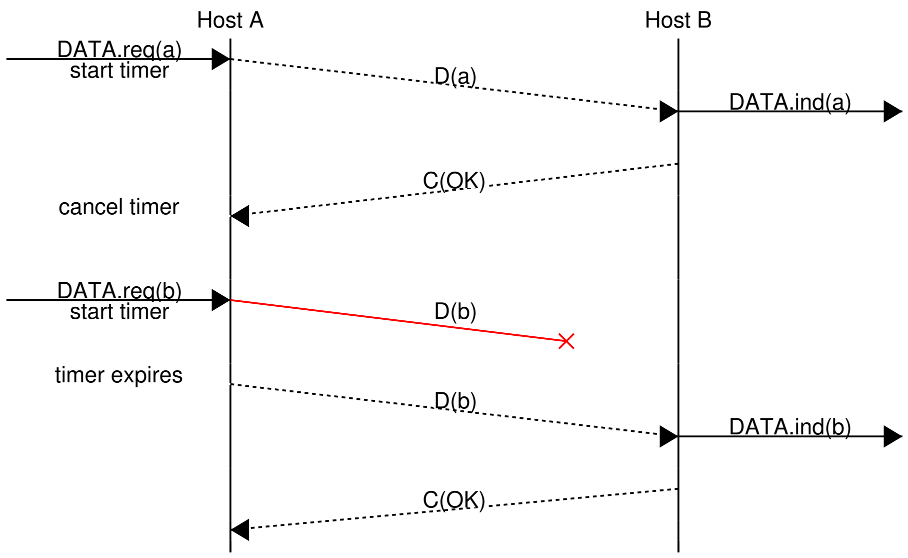
Some headers also include a *length* field, which indicates the total length of the frame or the length of the payload.

The simplest error detection scheme is the checksum. A checksum is basically an arithmetic sum of all the bytes that a frame is composed of. There are different types of checksums. For example, an eight bit checksum can be computed as the arithmetic sum of all the bytes of (both the header and trailer of) the frame. The checksum is computed by the sender before sending the frame and the receiver verifies the checksum upon frame reception. The receiver discards frames received with an invalid checksum. Checksums can be easily implemented in software, but their error detection capabilities are limited. Cyclical Redundancy Checks (CRC) have better error detection capabilities [SGP98], but require more CPU when implemented in software.

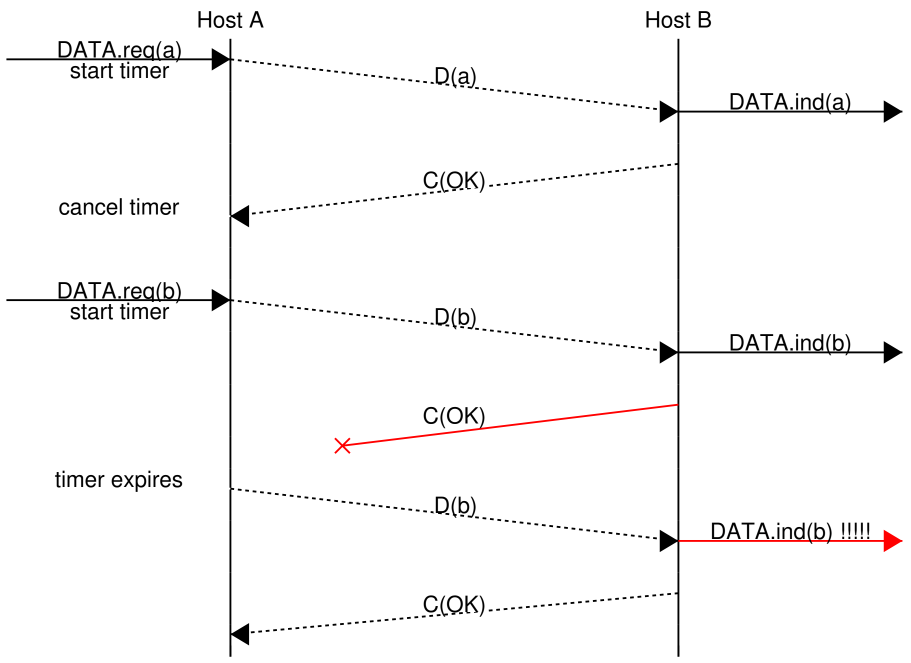
Note: Checksums, CRCs,…

Most of the protocols in the TCP/IP protocol suite rely on the simple Internet checksum in order to verify that a received packet has not been affected by transmission errors. Despite its popularity and ease of implementation, the Internet checksum is not the only available checksum mechanism. Cyclical Redundancy Checks (CRC) are very powerful error detection schemes that are used notably on disks, by many datalink layer protocols and file formats such as `zip` or `png`. They can easily be implemented efficiently in hardware and have better error-detection capabilities than the Internet checksum [SGP98] . However, CRCs are sometimes considered to be too CPU-intensive for software implementations and other checksum mechanisms are preferred. The TCP/IP community chose the Internet checksum, the OSI community chose the Fletcher checksum [Sklower89]. Nowadays there are efficient techniques to quickly compute CRCs in software [Feldmeier95].

Since the receiver sends an acknowledgment after having received each data frame, the simplest solution to deal with losses is to use a retransmission timer. When the sender sends a frame, it starts a retransmission timer. The value of this retransmission timer should be larger than the *round-trip-time*, i.e. the delay between the transmission of a data frame and the reception of the corresponding acknowledgment. When the retransmission timer expires, the sender assumes that the data frame has been lost and retransmits it. This is illustrated in the figure below.



Unfortunately, retransmission timers alone are not sufficient to recover from losses. Let us consider, as an example, the situation depicted below where an acknowledgment is lost. In this case, the sender retransmits the data frame that has not been acknowledged. However, as illustrated in the figure below, the receiver considers the retransmission as a new frame whose payload must be delivered to its user.



To solve this problem, datalink protocols associate a *sequence number* to each data frame. This *sequence number* is one of the fields found in the header of data frames. We use the notation *D(x,...)* to indicate a data frame whose sequence number field is set to value *x*. The acknowledgments also contain a sequence number indicating the data frames that it is acknowledging. We use *OKx* to indicate an acknowledgment frame that confirms the reception of *D(x,...)*. The sequence number is encoded as a bit string of fixed length. The simplest reliable protocol is the Alternating Bit Protocol (ABP).

The Alternating Bit Protocol uses a single bit to encode the sequence number. It can be implemented easily. The sender (resp. the receiver) only require a four-state (resp. three-state) Finite State Machine.

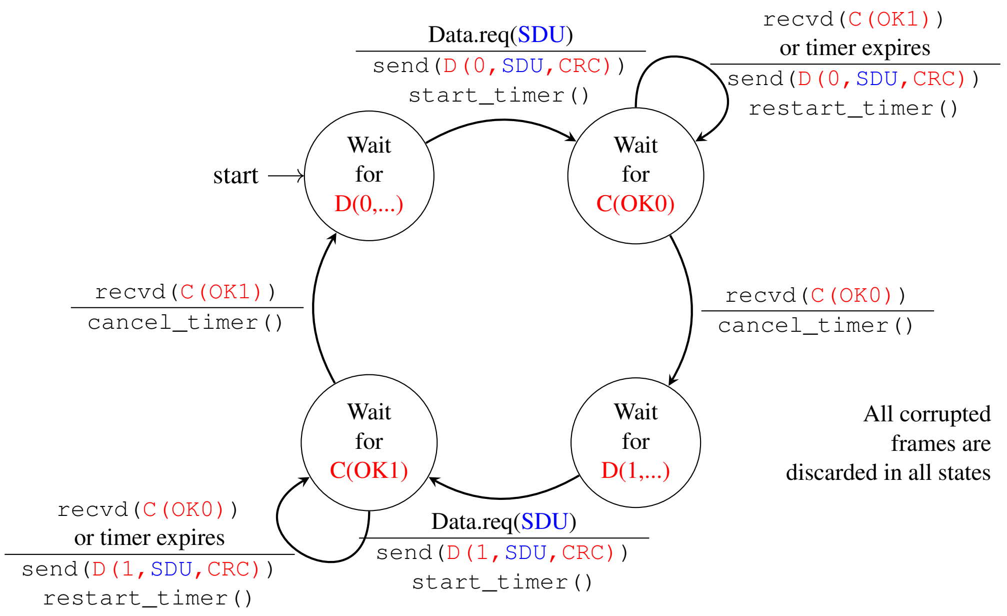


Fig. 5: Alternating bit protocol: Sender FSM

The initial state of the sender is *Wait for D(0,. . .)*. In this state, the sender waits for a *Data.request*. The first data frame that it sends uses sequence number *0*. After having sent this frame, the sender waits for an *OK0* acknowledgment. A frame is retransmitted upon expiration of the retransmission timer or if an acknowledgment with an incorrect sequence number has been received.

The receiver first waits for *D(0,. . .)*. If the frame contains a correct *CRC*, it passes the SDU to its user and sends *OK0*. If the frame contains an invalid CRC, it is immediately discarded. Then, the receiver waits for *D(1,. . .)*. In this state, it may receive a duplicate *D(0,. . .)* or a data frame with an invalid CRC. In both cases, it returns an *OK0* frame to allow the sender to recover from the possible loss of the previous *OK0* frame.

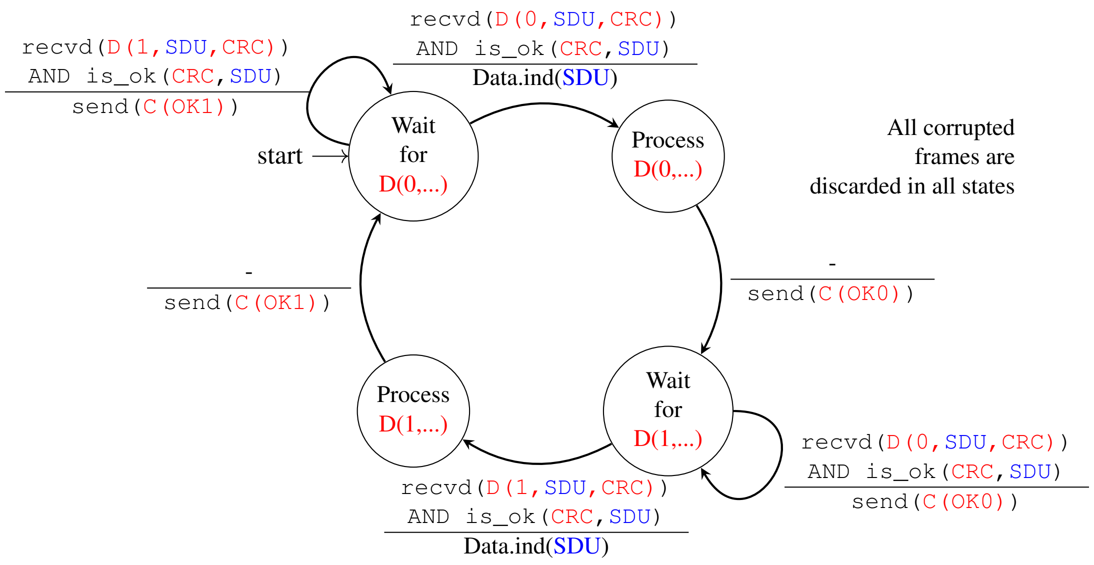
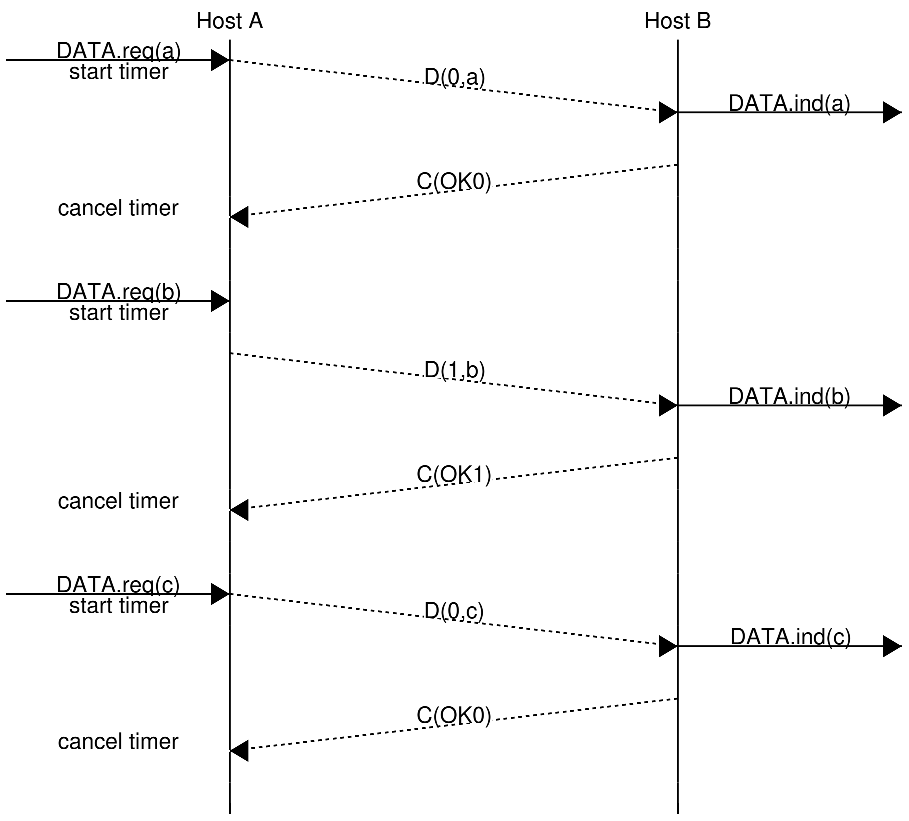

recvd(D(1,SDU,CRC)) AND is_ok(CRC,SDU)
send(C(OK1))
recvd(D(0,SDU,CRC)) AND is_ok(CRC,SDU)
Data.ind(SDU)
start
Wait for D(0,...)
Process D(0,...)
All corrupted frames are discarded in all states
-
send(C(OK1))
-
send(C(OK0))
Process D(1,...)
Wait for D(1,...)
recvd(D(0,SDU,CRC)) AND is_ok(CRC,SDU)
send(C(OK0))
recvd(D(1,SDU,CRC)) AND is_ok(CRC,SDU)
Data.ind(SDU)

Fig. 6: Alternating bit protocol: Receiver FSM

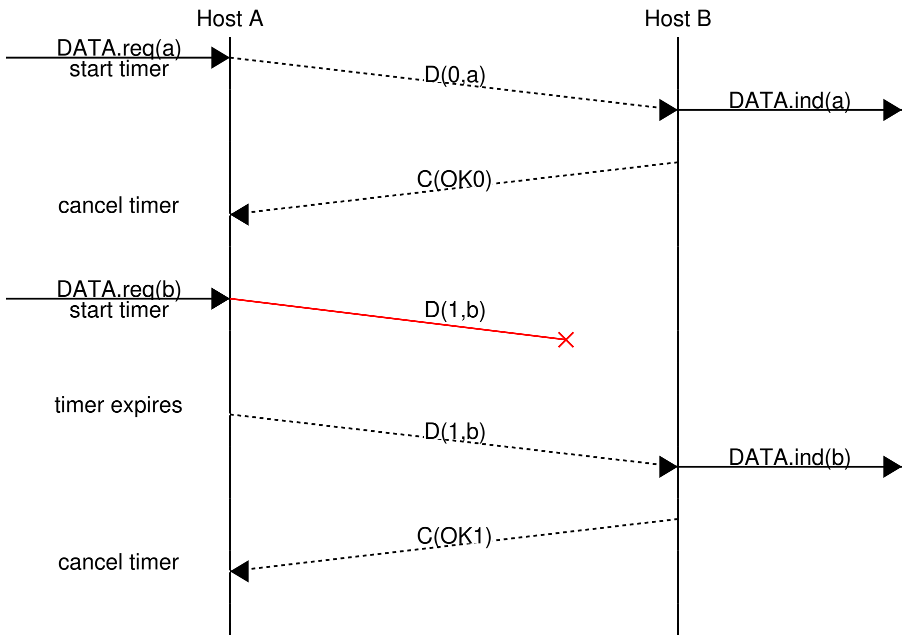
Note: Dealing with corrupted frames

The receiver FSM of the Alternating bit protocol discards all frames that contain an invalid CRC. This is the safest approach since the received frame can be completely different from the frame sent by the remote host. A receiver should not attempt at extracting information from a corrupted frame because it cannot know which portion of the frame has been affected by the error.

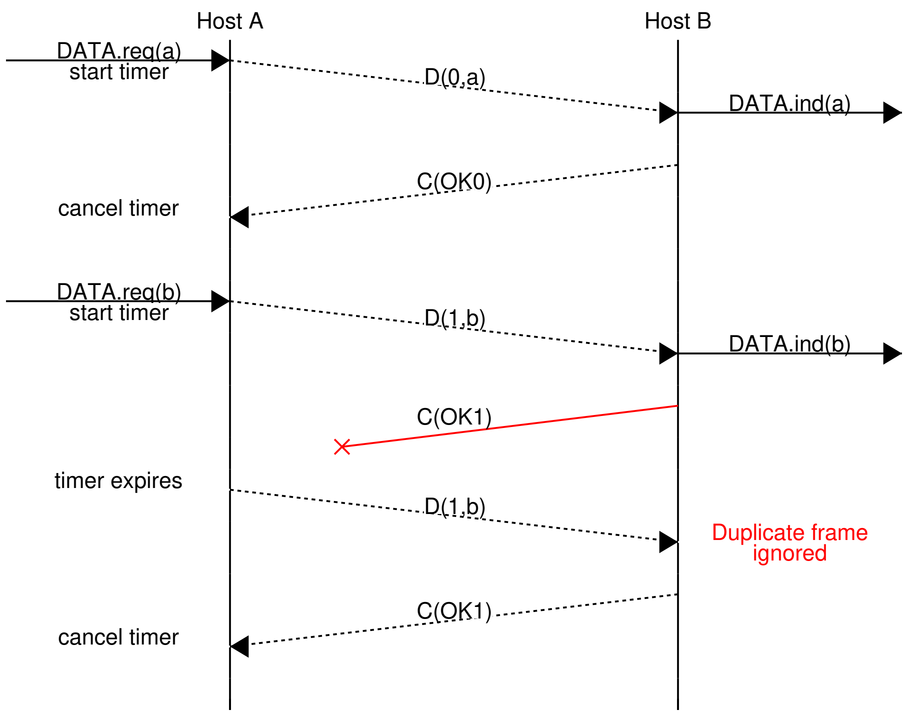
The figure below illustrates the operation of the alternating bit protocol.



The Alternating Bit Protocol can recover from the losses of data or control frames. This is illustrated in the two figures below. The first figure shows the loss of one data frame.



The second figure illustrates how the hosts handle the loss of one control frame.



The Alternating Bit Protocol can recover from transmission errors and frame losses. However, it has one important drawback. Consider two hosts that are directly connected by a 50 Kbits/sec satellite link that has a 250 milliseconds propagation delay. If these hosts send 1000 bits frames, then the maximum throughput that can be achieved by the alternating bit protocol is one frame every $20 + 250 + 250 = 520$ milliseconds if we ignore the transmission time of the acknowledgment. This is less than 2 Kbits/sec !

Go-back-n and selective repeat

To overcome the performance limitations of the alternating bit protocol, reliable protocols rely on *pipelining*. This technique allows a sender to transmit several consecutive frames without being forced to wait for an acknowledgment after each frame. Each data frame contains a sequence number encoded as an *n* bits field.

Pipelining allows the sender to transmit frames at a higher rate. However this higher transmission rate may overload the receiver. In this case, the frames sent by the sender will not be correctly received by their final destination. The reliable protocols that rely on pipelining allow the sender to transmit *W* unacknowledged frames before being forced to wait for an acknowledgment from the receiving entity.

This is implemented by using a *sliding window*. The sliding window is the set of consecutive sequence numbers that the sender can use when transmitting frames without being forced to wait for an acknowledgment. The figure below shows a sliding window containing five frames (*6,7,8,9* and *10*). Two of these sequence numbers (*6* and *7*) have been used to send frames and only three sequence numbers (*8*, *9* and *10*) remain in the sliding window. The sliding window is said to be closed once all sequence numbers contained in the sliding window have been used.

The figure below illustrates the operation of the sliding window. It uses a sliding window of three frames. The sender can thus transmit three frames before being forced to wait for an acknowledgment. The sliding window moves to the higher sequence numbers upon the reception of each acknowledgment. When the first acknowledgment (*OK0*) is received, it enables the sender to move its sliding window to the right and sequence number *3* becomes available. This sequence number is used later to transmit the frame containing *d*.

In practice, as the frame header includes an *n* bits field to encode the sequence number, only the sequence numbers between 0 and $2^n - 1$ can be used. This implies that, during a long transfer, the same sequence number will be used for different frames and the sliding window will wrap. This is illustrated in the figure below assuming that *2* bits are used

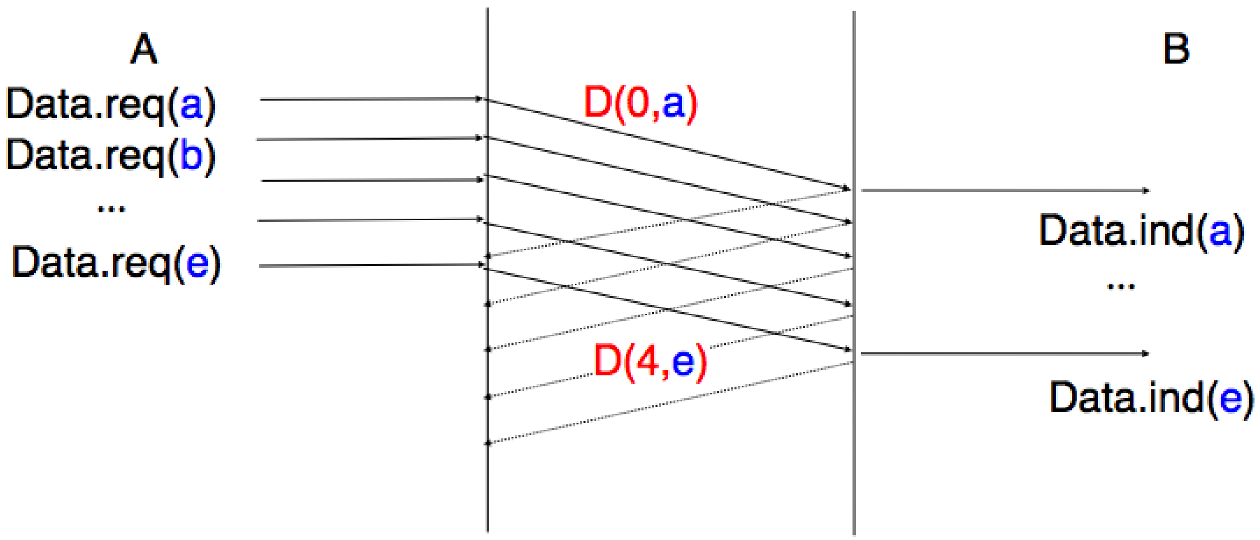


Fig. 7: Pipelining improves the performance of reliable protocols

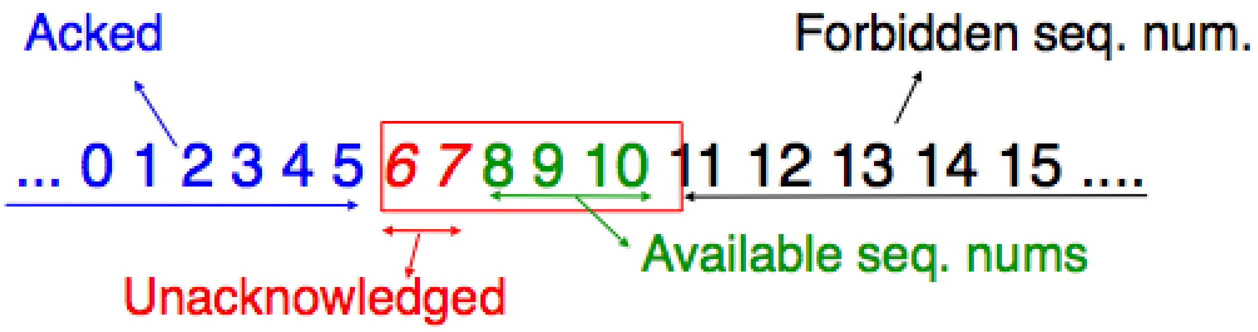


Fig. 8: The sliding window

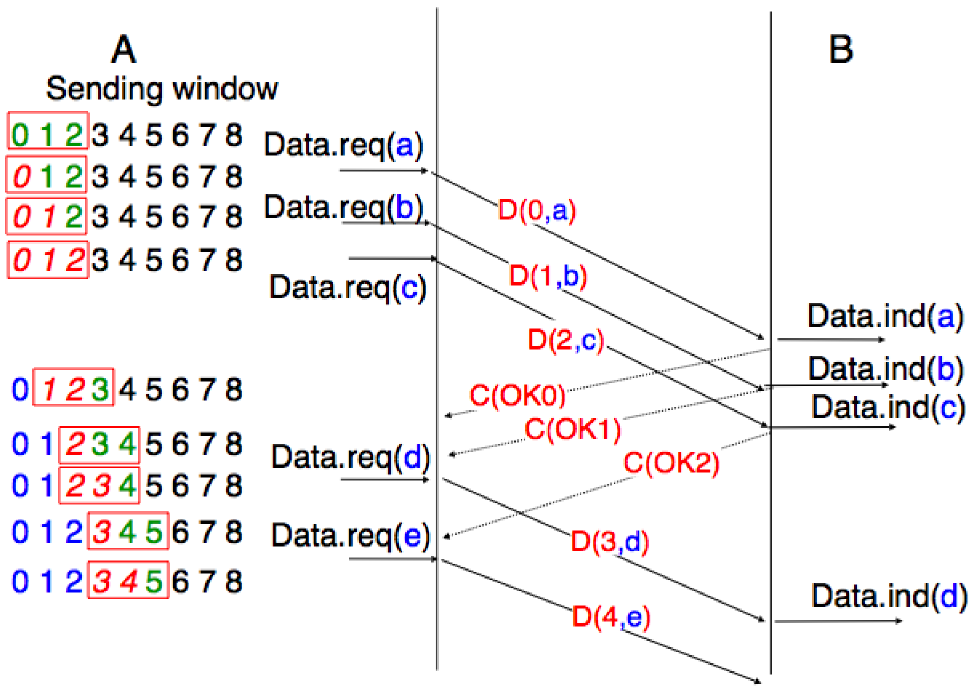


Fig. 9: Sliding window example

to encode the sequence number in the frame header. Note that upon reception of *OK1*, the sender slides its window and can use sequence number *0* again.

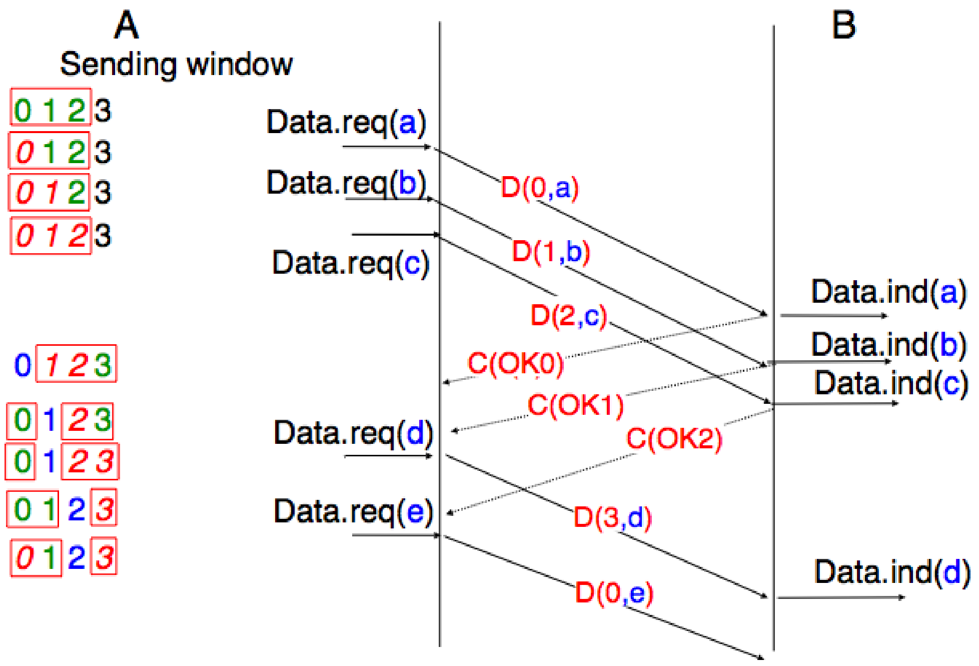


Fig. 10: Utilisation of the sliding window with modulo arithmetic

Unfortunately, frame losses do not disappear because a reliable protocol uses a sliding window. To recover from losses, a sliding window protocol must define :

- a heuristic to detect frame losses
- a *retransmission strategy* to retransmit the lost frames

The simplest sliding window protocol uses the *go-back-n* recovery. Intuitively, *go-back-n* operates as follows. A *go-back-n* receiver is as simple as possible. It only accepts the frames that arrive in-sequence. A *go-back-n* receiver discards any out-of-sequence frame that it receives. When *go-back-n* receives a data frame, it always returns an acknowledgment containing the sequence number of the last in-sequence frame that it has received. This acknowledgment is said to be *cumulative*. When a *go-back-n* receiver sends an acknowledgment for sequence number *x*, it implicitly acknowledges the reception of all frames whose sequence number is earlier than *x*. A key advantage of these cumulative acknowledgments is that it is easy to recover from the loss of an acknowledgment. Consider for example a *go-back-n* receiver that received frames *1*, *2* and *3*. It sent *OK1*, *OK2* and *OK3*. Unfortunately, *OK1* and *OK2* were lost. Thanks to the cumulative acknowledgments, when the sender receives *OK3*, it knows that all three frames have been correctly received.

The figure below shows the FSM of a simple *go-back-n* receiver. This receiver uses two variables : *lastack* and *next*. *next* is the next expected sequence number and *lastack* the sequence number of the last data frame that has been acknowledged. The receiver only accepts the frame that are received in sequence. *maxseq* is the number of different sequence numbers (2^n).

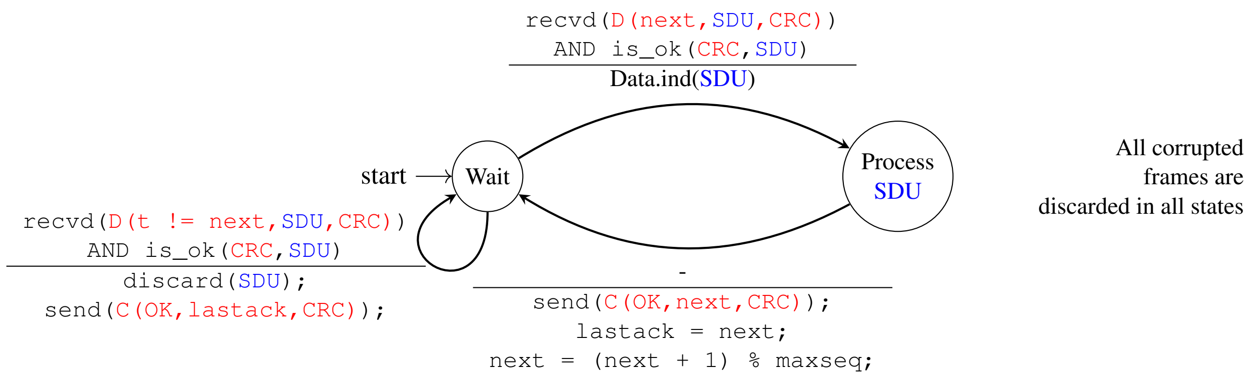


Fig. 11: Go-back-n: receiver FSM

A *go-back-n* sender is also very simple. It uses a sending buffer that can store an entire sliding window of frames[2]. The frames are sent with increasing sequence numbers (modulo *maxseq*). The sender must wait for an acknowledgment once its sending buffer is full. When a *go-back-n* sender receives an acknowledgment, it removes from the sending buffer all the acknowledged frames and uses a retransmission timer to detect frame losses. A simple *go-back-n* sender maintains one retransmission timer per connection. This timer is started when the first frame is sent. When the *go-back-n sender* receives an acknowledgment, it restarts the retransmission timer only if there are still unacknowledged frames in its sending buffer. When the retransmission timer expires, the *go-back-n* sender assumes that all the unacknowledged frames currently stored in its sending buffer have been lost. It thus retransmits all the unacknowledged frames in the buffer and restarts its retransmission timer.

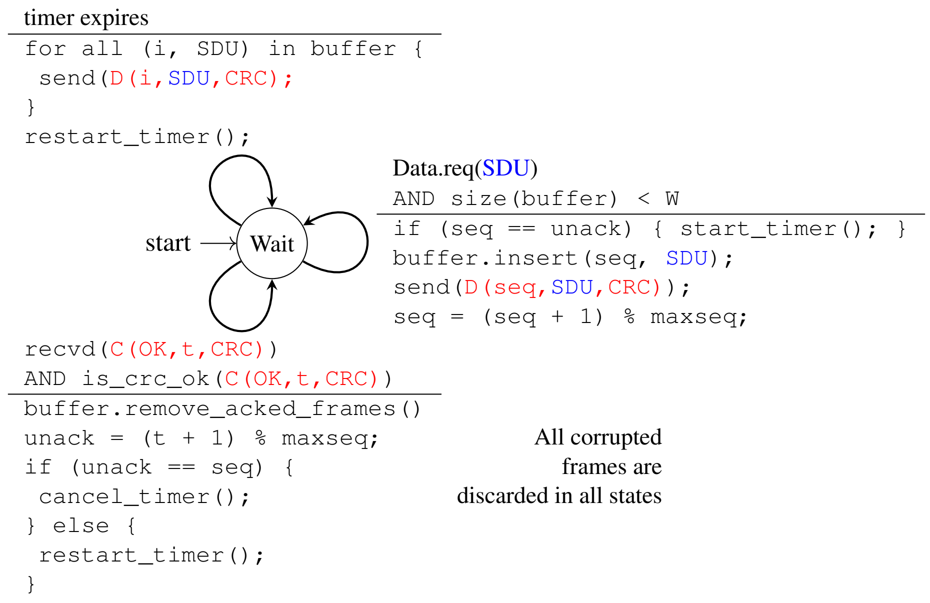

timer expires
for all (i, SDU) in buffer {
send(D(i,SDU,CRC);
}
restart_timer();
start
Wait
Data.req(SDU)
AND size(buffer) < W
if (seq == unack) { start_timer(); }
buffer.insert(seq, SDU);
send(D(seq,SDU,CRC));
seq = (seq + 1) % maxseq;
recvd(C(OK,t,CRC))
AND is_crc_ok(C(OK,t,CRC))
buffer.remove_acked_frames()
unack = (t + 1) % maxseq;
if (unack == seq) {
cancel_timer();
} else {
restart_timer();
}
All corrupted frames are discarded in all states

Fig. 12: Go-back-n: sender FSM

[2] The size of the sliding window can be either fixed for a given protocol or negotiated during the connection establishment phase. Some protocols allow to change the maximum window size during the data transfer. We will explain these techniques with real protocols later.

The operation of *go-back-n* is illustrated in the figure below. In this figure, note that upon reception of the out-of-sequence frame *D(2,c)*, the receiver returns a cumulative acknowledgment *C(OK,0)* that acknowledges all the frames that have been received in sequence. The lost frame is retransmitted upon the expiration of the retransmission timer.

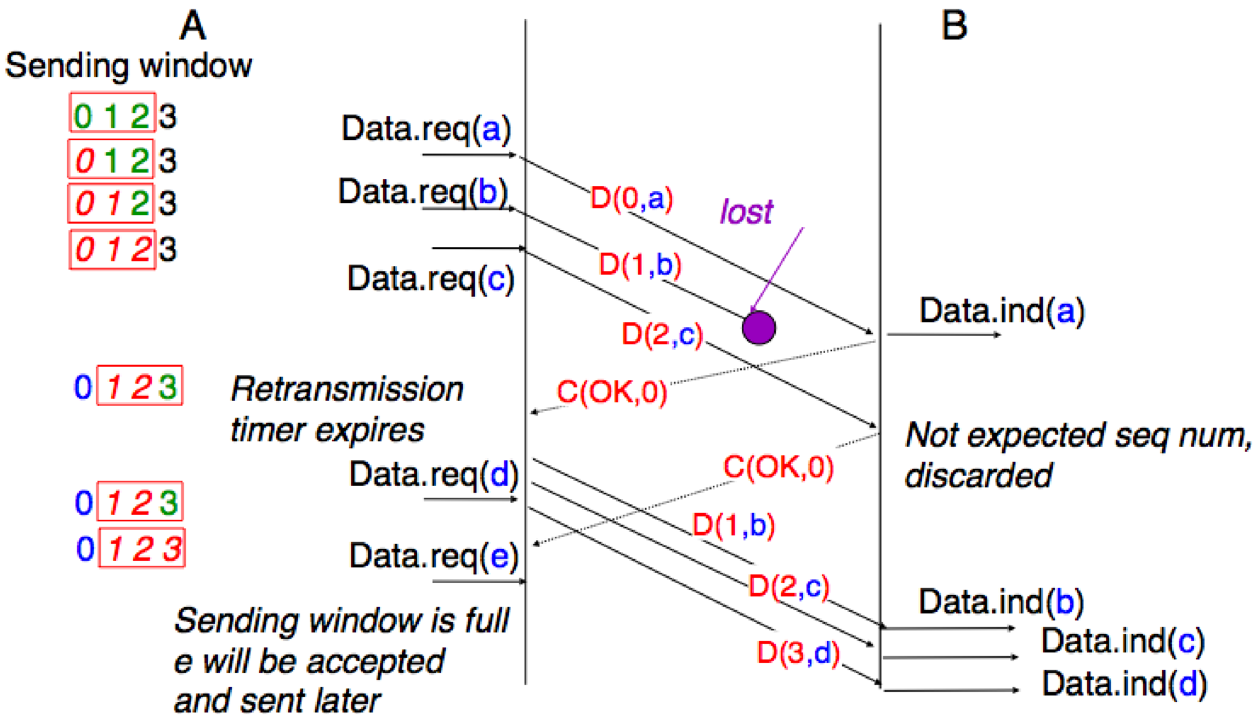


Fig. 13: Go-back-n : example

The main advantage of *go-back-n* is that it can be easily implemented, and it can also provide good performance when only a few frames are lost. However, when there are many losses, the performance of *go-back-n* quickly drops for two reasons :

- the *go-back-n* receiver does not accept out-of-sequence frames
- the *go-back-n* sender retransmits all unacknowledged frames once it has detected a loss

Selective repeat is a better strategy to recover from losses. Intuitively, *selective repeat* allows the receiver to accept out-of-sequence frames. Furthermore, when a *selective repeat* sender detects losses, it only retransmits the frames that have been lost and not the frames that have already been correctly received.

A *selective repeat* receiver maintains a sliding window of *W* frames and stores in a buffer the out-of-sequence frames that it receives. The figure below shows a five-frame receive window on a receiver that has already received frames *7* and *9*.

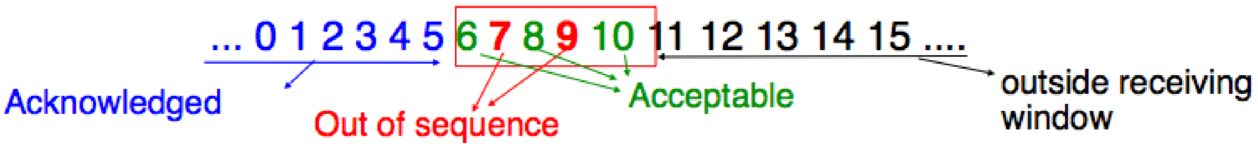


Fig. 14: The receiving window with selective repeat

A *selective repeat* receiver discards all frames having an invalid CRC, and maintains the variable *lastack* as the sequence number of the last in-sequence frame that it has received. The receiver always includes the value of *lastack* in the acknowledgments that it sends. Some protocols also allow the *selective repeat* receiver to acknowledge the out-of-sequence frames that it has received. This can be done for example by placing the list of the correctly received, but out-of-sequence frames in the acknowledgments together with the *lastack* value.

When a *selective repeat* receiver receives a data frame, it first verifies whether the frame is inside its receiving window. If yes, the frame is placed in the receive buffer. If not, the received frame is discarded and an acknowledgment containing *lastack* is sent to the sender. The receiver then removes all consecutive frames starting at *lastack* (if any) from the receive buffer. The payloads of these frames are delivered to the user, *lastack* and the receiving window are updated, and an acknowledgment acknowledging the last frame received in sequence is sent.

The *selective repeat* sender maintains a sending buffer that can store up to *W* unacknowledged frames. These frames are sent as long as the sending buffer is not full. Several implementations of a *selective repeat* sender are possible. A simple implementation associates one retransmission timer to each frame. The timer is started when the frame is sent and canceled upon reception of an acknowledgment that covers this frame. When a retransmission timer expires, the corresponding frame is retransmitted and this retransmission timer is restarted. When an acknowledgment is received, all the frames that are covered by this acknowledgment are removed from the sending buffer and the sliding window is updated.

The figure below illustrates the operation of *selective repeat* when frames are lost. In this figure, *C(OK,x)* is used to indicate that all frames, up to and including sequence number *x* have been received correctly.

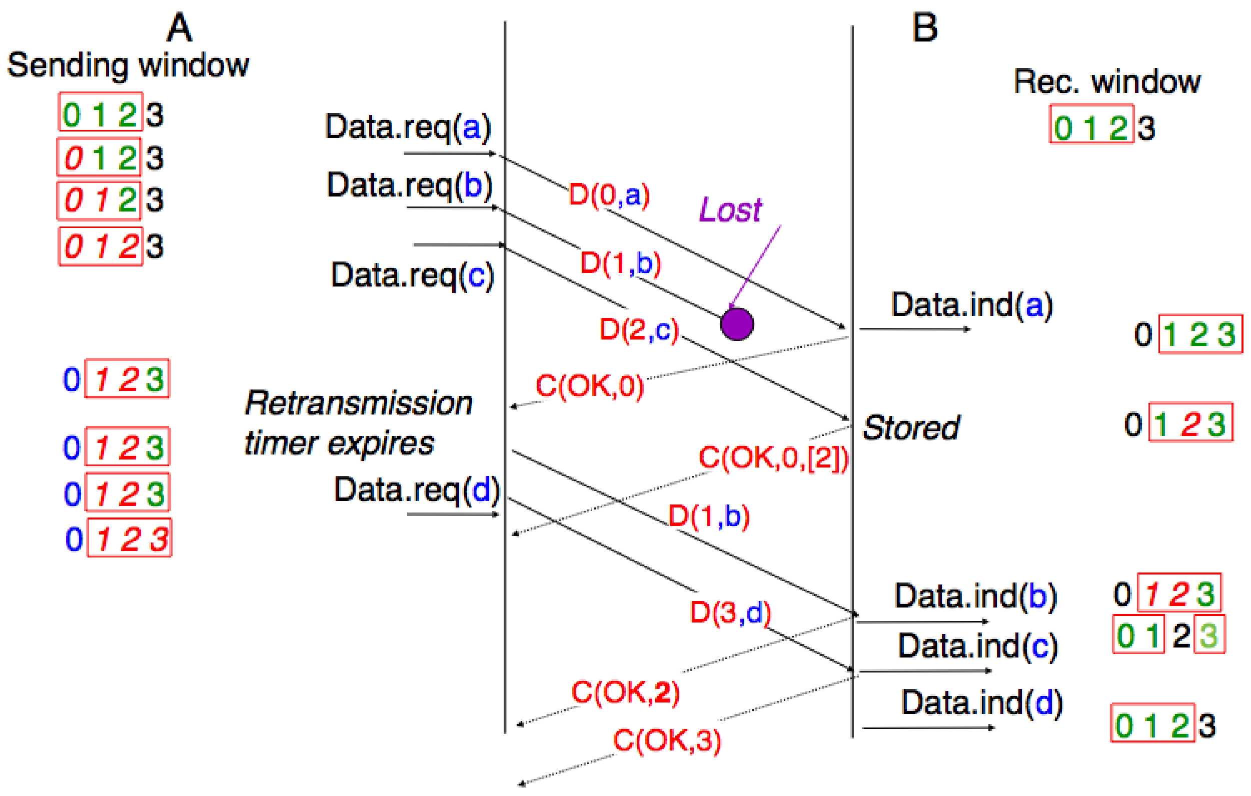


Fig. 15: Selective repeat : example

Pure cumulative acknowledgments work well with the *go-back-n* strategy. However, with only cumulative acknowledgments a *selective repeat* sender cannot easily determine which frames have been correctly received after a data frame has been lost. For example, in the figure above, the second *C(OK,0)* does not inform explicitly the sender of the reception of *D(2,c)* and the sender could retransmit this frame although it has already been received. A possible solution to improve the performance of *selective repeat* is to provide additional information about the received frames in the acknowledgments that are returned by the receiver. For example, the receiver could add in the returned acknowledgment the list of the sequence numbers of all frames that have already been received. Such acknowledgments are sometimes called *selective acknowledgments*. This is illustrated in the figure above.

In the figure above, when the sender receives *C(OK,0,[2])*, it knows that all frames up to and including *D(0,. . .)* have been correctly received. It also knows that frame *D(2,. . .)* has been received and can cancel the retransmission timer associated to this frame. However, this frame should not be removed from the sending buffer before the reception of a cumulative acknowledgment (*C(OK,2)* in the figure above) that covers this frame.

Note: Maximum window size with *go-back-n* and *selective repeat*

A reliable protocol that uses *n* bits to encode its sequence number can send up to 2^n successive frames. However, to ensure a reliable delivery of the frames, *go-back-n* and *selective repeat* cannot use a sending window of 2^n frames. Consider first *go-back-n* and assume that a sender sends 2^n frames. These frames are received in-sequence by the destination, but all the returned acknowledgments are lost. The sender will retransmit all frames. These frames will all be accepted by the receiver and delivered a second time to the user. It is easy to see that this problem can be avoided if the maximum size of the sending window is $2^n - 1$ frames. A similar problem occurs with *selective repeat*. However, as the receiver accepts out-of-sequence frames, a sending window of $2^n - 1$ frames is not sufficient to ensure a reliable delivery. It can be easily shown that to avoid this problem, a *selective repeat* sender cannot use a window that is larger than $\frac{2^n}{2}$ frames.

Reliable protocols often need to send data in both directions. To reduce the overhead caused by the acknowledgments, most reliable protocols use *piggybacking*. Thanks to this technique, a datalink entity can place the acknowledgments and the receive window that it advertises for the opposite direction of the data flow inside the header of the data frames that it sends. The main advantage of piggybacking is that it reduces the overhead as it is not necessary to send a complete frame to carry an acknowledgment. This is illustrated in the figure below where the acknowledgment number is underlined in the data frames. Piggybacking is only used when data flows in both directions. A receiver will generate a pure acknowledgment when it does not send data in the opposite direction as shown in the bottom of the figure.

2.2 Building a network

In the previous section, we have explained how reliable protocols allow hosts to exchange data reliably even if the underlying physical layer is imperfect and thus unreliable. Connecting two hosts together through a wire is the first step to build a network. However, this is not sufficient. Hosts usually need to interact with other hosts that are not directly connected through a direct physical layer link. This can be achieved by adding one layer above the datalink layer: the *network* layer.

The main objective of the network layer is to allow hosts, connected to different networks, to exchange information through intermediate systems called *router*. The unit of information in the network layer is called a *packet*.

Before explaining the operation of the network layer, it is useful to remember the characteristics of the service provided by the *datalink* layer. There are many variants of the datalink layer. Some provide a reliable service while others do not provide any guarantee of delivery. The reliable datalink layer services are popular in environments such as wireless networks where transmission errors are frequent. On the other hand, unreliable services are usually used when the physical layer provides an almost reliable service (i.e. only a negligible fraction of the frames are affected by transmission errors). Such *almost reliable* services are frequently used in wired and optical networks. In this chapter, we will assume that the datalink layer service provides an *almost reliable* service since this is both the most general one and also the most widely deployed one.

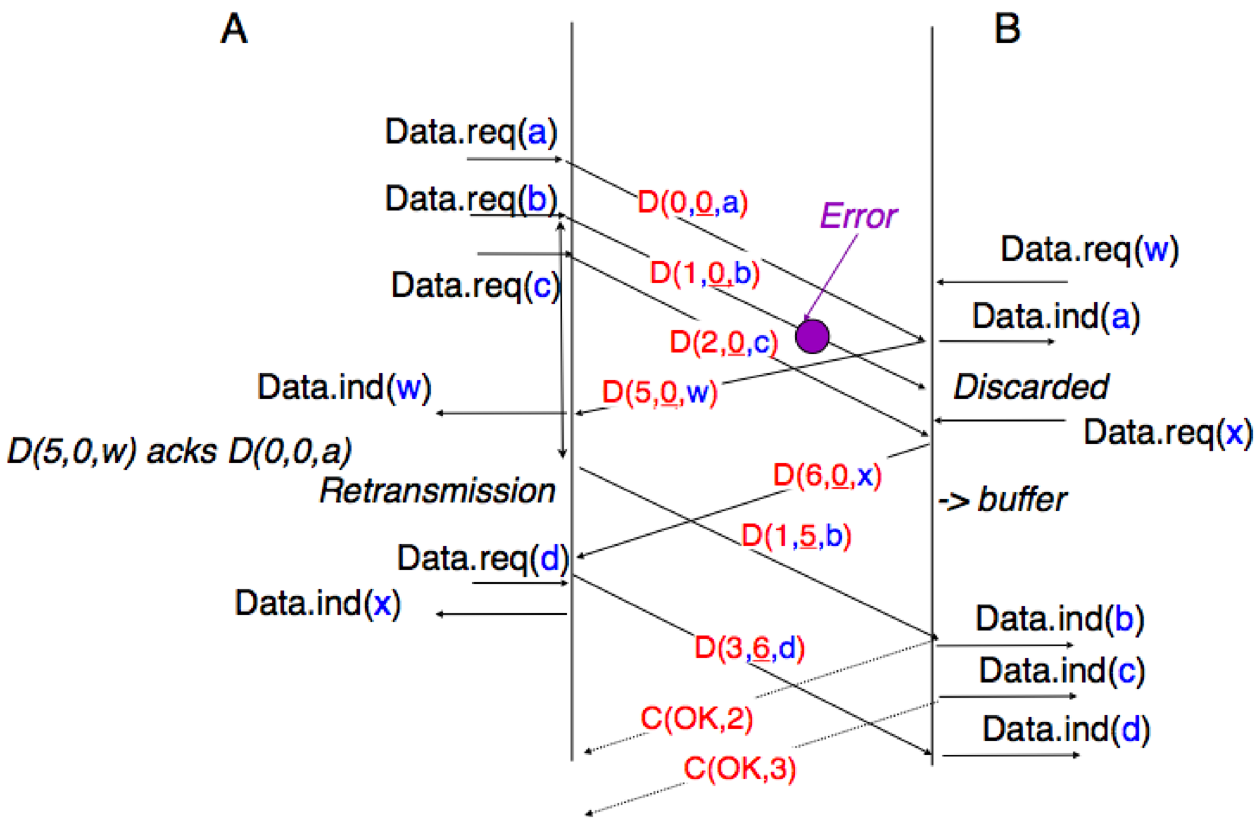


Fig. 16: Piggybacking example

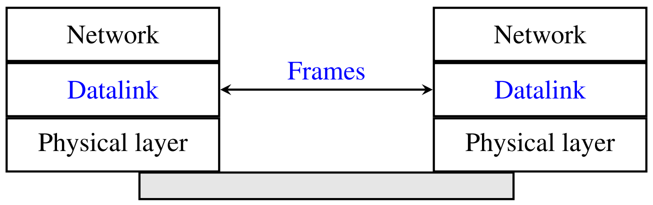


Fig. 17: The point-to-point datalink layer

There are two main types of datalink layers. The simplest datalink layer is when there are only two communicating systems that are directly connected through the physical layer. Such a datalink layer is used when there is a point-to-point link between the two communicating systems. These two systems can be hosts or routers. PPP (Point-to-Point Protocol), defined in **RFC 1661**, is an example of such a point-to-point datalink layer. Datalink layer entities exchange *frames*. A datalink *frame* sent by a datalink layer entity on the left is transmitted through the physical layer, so that it can reach the datalink layer entity on the right. Point-to-point datalink layers can either provide an unreliable service (frames can be corrupted or lost) or a reliable service (in this case, the datalink layer includes retransmission mechanisms).

The second type of datalink layer is the one used in Local Area Networks (LAN). Conceptually, a LAN is a set of communicating devices such that any two devices can directly exchange frames through the datalink layer. Both hosts and routers can be connected to a LAN. Some LANs only connect a few devices, but there are LANs that can connect hundreds or even thousands of devices. In this chapter, we focus on the utilization of point-to-point datalink layers. We describe later the organization and the operation of Local Area Networks and their impact on the network layer.

Even if we only consider the point-to-point datalink layers, there is an important characteristic of these layers that we cannot ignore. No datalink layer is able to send frames of unlimited size. Each datalink layer is characterized by a maximum frame size. There are more than dozen different datalink layers and unfortunately most of them use a different maximum frame size. This heterogeneity in the maximum frame sizes will cause problems when we will need to exchange data between hosts attached to different types of datalink layers.

As a first step, let us assume that we only need to exchange a small amount of data. In this case, there is no issue with the maximum length of the frames. However, there are other more interesting problems that we need to tackle. To understand these problems, let us consider the network represented in the figure below.

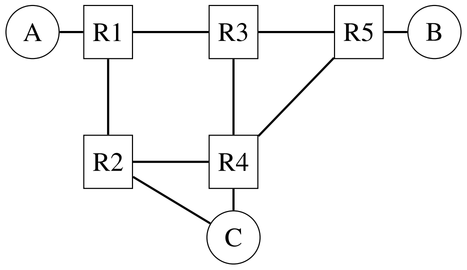


Fig. 18: A simple network containing three hosts and five routers

This network contains two types of devices. The hosts, represented with circles and the routers, represented as boxes. A host is a device which is able to send and receive data for its own usage in contrast with routers that most of the time simply forward data towards their final destination. Routers have multiple links to neighboring routers or hosts. Hosts are usually attached via a single link to the network. Nowadays, with the growth of wireless networks, more and more hosts are equipped with several physical interfaces. These hosts are often called *multihomed*. Still, using several interfaces at the same time often leads to practical issues that are beyond the scope of this document. For this reason, we only consider *single-homed* hosts in this e-book.

To understand the key principles behind the operation of a network, let us analyze all the operations that need to be performed to allow host *A* in the above network to send one byte to host *B*. Thanks to the datalink layer used above the *A-R1* link, host *A* can easily send a byte to router *R1* inside a frame. However, upon reception of this frame, router *R1* needs to understand that this byte is destined to host *B* and not to itself. This is the objective of the network layer.

The network layer enables the transmission of information between hosts that are not directly connected through intermediate routers. This transmission is carried out by putting the information to be transmitted inside a data structure which is called a *packet*. As a frame that contains useful data and control information, a packet also contains both data supplied by the user and control information. An important issue in the network layer is the ability to identify a node (host or router) inside the network. This identification is performed by associating an address to each node. An *address* is usually represented as a sequence of bits. Most networks use fixed-length addresses. At this stage, let us simply assume that each of the nodes in the above network has an address which corresponds to the binary representation of its name on the figure.

To send one byte of information to host *B*, host *A* needs to place this information inside a *packet*. In addition to the data being transmitted, the packet also contains either the addresses of the source and the destination nodes or information that indicates the path that needs to be followed to reach the destination.

There are two possible organizations for the network layer :

- *datagram*
- *virtual circuits*

2.2.1 The datagram organization

The first and most popular organization of the network layer is the datagram organization. This organization is inspired by the organization of the postal service. Each host is identified by a *network layer address*. To send information to a remote host, a host creates a packet that contains:

- the network layer address of the destination host
- its own network layer address
- the information to be sent

To understand the datagram organization, let us consider the figure below. A network layer address, represented by a letter, has been assigned to each host and router. To send some information to host *J*, host *A* creates a packet containing its own address, the destination address and the information to be exchanged.

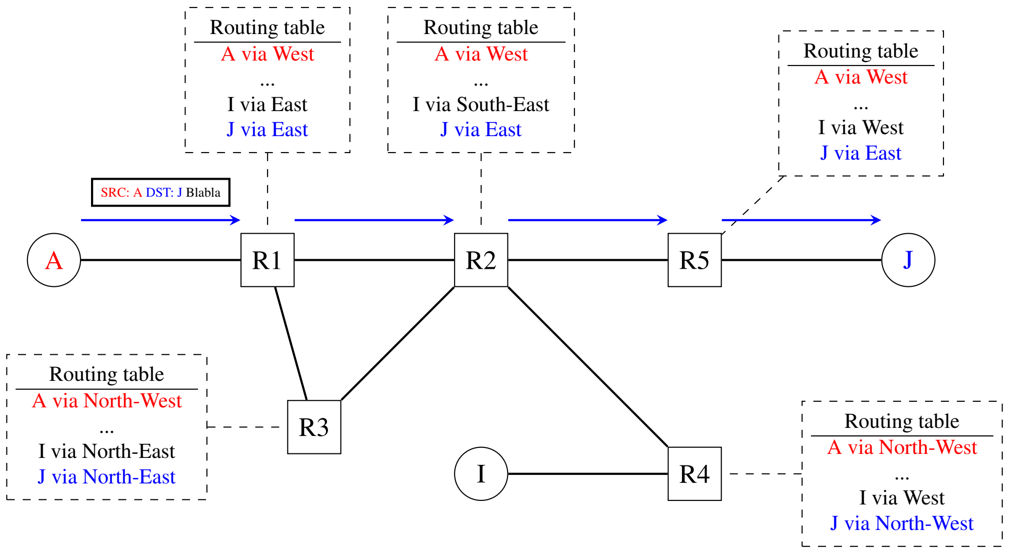


Fig. 19: A simple internetwork

With the datagram organization, routers use *hop-by-hop forwarding*. This means that when a router receives a packet that is not destined to itself, it looks up the destination address of the packet in its *forwarding table*. A *forwarding table* is a data structure that maps each destination address (or set of destination addresses) to the outgoing interface over which a packet destined to this address must be forwarded to reach its final destination. The router consults its forwarding table to forward each packet that it handles.

The figure illustrates some possible forwarding tables in this network. By inspecting the forwarding tables of the different routers, one can find the path followed by packets sent from a source to a particular destination. In the example above, host *A* sends its packet to router *R1*. *R1* consults its forwarding table and forwards the packet towards *R2*. Based on its own table, *R2* decides to forward the packet to *R5* that can deliver it to its destination. Thus, the path from *A* to *J* is *A -> R1 -> R2 -> R5 -> J*.

The computation of the forwarding tables of all the routers inside a network is a key element for the correct operation of the network. This computation can be carried out by using either distributed or centralized algorithms. These algorithms provide different performance, may lead to different types of paths, but their composition must lead to valid paths.

In a network, a path can be defined as the list of all intermediate routers for a given source destination pair. For a given source/destination pair, the path can be derived by first consulting the forwarding table of the router attached to the source to determine the next router on the path towards the chosen destination. Then, the forwarding table of this router is queried for the same destination… The queries continue until the destination is reached. In a network that has valid forwarding tables, all the paths between all source/destination pairs contain a finite number of intermediate routers. However, if forwarding tables have not been correctly computed, two types of invalid paths can occur.

A path may lead to a *black hole*. In a network, a black hole is a router that receives packets for at least one given source/destination pair but does not have an entry inside its forwarding table for this destination. Since it does not know how to reach the destination, the router cannot forward the received packets and must discard them. Any centralized or distributed algorithm that computes forwarding tables must ensure that there are not black holes inside the network.

A second type of problem may exist in networks using the datagram organization. Consider a path that contains a cycle. For example, router *R1* sends all packets towards destination *D* via router *R2*. Router *R2* forwards these packets to router *R3* and finally router *R3*'s forwarding table uses router *R1* as its nexthop to reach destination *D*. In this case, if a packet destined to *D* is received by router *R1*, it will loop on the *R1 -> R2 -> R3 -> R1* cycle and will never reach its final destination. As in the black hole case, the destination is not reachable from all sources in the network. In practice the loop problem is more annoying than the black hole problem because when a packet is caught in a forwarding loop, it unnecessarily consumes bandwidth. In the black hole case, the problematic packet is quickly discarded. We will see later that network layer protocols include techniques to minimize the impact of such forwarding loops.

Any solution which is used to compute the forwarding tables of a network must ensure that all destinations are reachable from any source. This implies that it must guarantee the absence of black holes and forwarding loops.

The *forwarding tables* and the precise format of the packets that are exchanged inside the network are part of the *data plane* of the network. This *data plane* contains all the protocols and algorithms that are used by hosts and routers to create and process the packets that contain user data. On high-end routers, the data plane is often implemented in hardware for performance reasons.

Besides the *data plane*, a network is also characterized by its *control plane*. The control plane includes all the protocols and algorithms (often distributed) that compute the forwarding tables that are installed on all routers inside the network. While there is only one possible *data plane* for a given networking technology, different networks using the same technology may use different control planes.

The simplest *control plane* for a network is to manually compute the forwarding tables of all routers inside the network. This simple control plane is sufficient when the network is (very) small, usually up to a few routers.

In most networks, manual forwarding tables are not a solution for two reasons. First, most networks are too large to enable a manual computation of the forwarding tables. Second, with manually computed forwarding tables, it is very difficult to deal with link and router failures. Networks need to operate 24h a day, 365 days per year. Many events can affect the routers and links that compose a network. Link failures are regular events in deployed networks. Links can fail for various reasons, including electromagnetic interference, fiber cuts, hardware or software problems on the terminating routers,… Some links also need to be either added to or removed from the network because their utilization is too low or their cost is too high.

Similarly, routers also fail. There are two types of failures that affect routers. A router may stop forwarding packets due to hardware or software problems (e.g., due to a crash of its operating system). A router may also need to be halted from time to time (e.g., to upgrade its operating system or to install new interface cards). These planned and unplanned events affect the set of links and routers that can be used to forward packets in the network. Still, most network users expect that their network will continue to correctly forward packets despite all these events. With manually computed forwarding tables, it is usually impossible to pre-compute the forwarding tables while taking into account all possible failure scenarios.

An alternative to manually computed forwarding tables is to use a network management platform that tracks the network status and can push new forwarding tables on the routers when it detects any modification to the network topology. This solution gives some flexibility to the network managers in computing the paths inside their network. However, this solution only works if the network management platform is always capable of reaching all routers even when the network topology changes. This may require a dedicated network that allows the management platform to push information on the forwarding tables. Openflow is a modern example of such solutions [MAB2008]. In a nutshell, Openflow is a protocol that enables a network controller to install specific entries in the forwarding tables of remote routers and much more.

Another interesting point that is worth being discussed is when the forwarding tables are computed. A widely used solution is to compute the entries of the forwarding tables for all destinations on all routers. This ensures that each router has a valid route towards each destination. These entries can be updated when an event occurs and the network topology changes. A drawback of this approach is that the forwarding tables can become large in large networks since each router must always maintain one entry for each destination inside its forwarding table.

Some networks use the arrival of packets as the trigger to compute the corresponding entries in the forwarding tables. Several technologies have been built upon this principle. When a packet arrives, the router consults its forwarding table to find a path towards the destination. If the destination is present in the forwarding table, the packet is forwarded. Otherwise, the router needs to find a way to forward the packet and update its forwarding table.

Computing forwarding tables

Networks deployed several techniques to update the forwarding tables upon the arrival of a packet. In this section, we briefly present the principles that underlie three of these techniques.

The first technique assumes that the underlying network topology is a tree. A tree is the simplest network to be considered when forwarding packets. The main advantage of using a tree is that there is only one path between any pair of nodes inside the network. Since a tree does not contain any cycle, it is impossible to have forwarding loops in a tree-shaped network.

In a tree-shaped network, it is relatively simple for each node to automatically compute its forwarding table by inspecting the packets that it receives. For this, each node uses the source and destination addresses present inside each packet. Thanks to the source address, a node can learn the location of the different sources inside the network. Each source has a unique address. When a node receives a packet over a given interface, it learns that the source (address) of this packet is reachable via this interface. The node maintains a data structure that maps each known source address to an incoming interface. This data structure is often called the *port-address table* since it indicates the interface (or port) to reach a given address.

Learning the location of the sources is not sufficient, nodes also need to forward packets towards their destination. When a node receives a packet whose destination address is already present inside its port-address table, it simply forwards the packet on the interface listed in the port-address table. In this case, the packet will follow the port-address table entries in the downstream nodes and will reach the destination. If the destination address is not included in the port-address table, the node simply forwards the packet on all its interfaces, except the interface from which the packet was received. Forwarding a packet over all interfaces is usually called *broadcasting* in the terminology of computer networks. Sending the packet over all interfaces except one is a costly operation since the packet is sent over links that do not reach the destination. Given the tree-shape of the network, the packet will explore all downstream branches of the tree and will finally reach its destination. In practice, the *broadcasting* operation does not occur too often and its performance impact remains limited.

To understand the operation of the port-address table, let us consider the example network shown in the figure below. This network contains three hosts: *A*, *B* and *C* and five routers, *R1* to *R5*. When the network boots, all the forwarding tables of the nodes are empty.

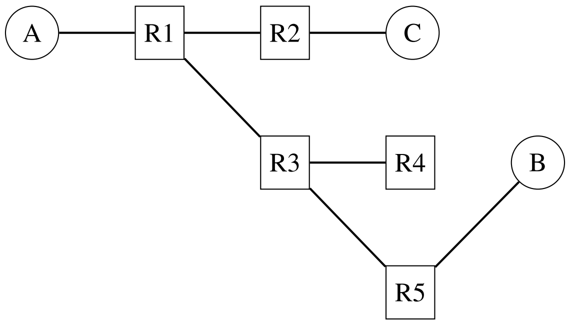


Fig. 20: A simple tree-shaped network

Host *A* sends a packet towards *B*. When receiving this packet, *R1* learns that *A* is reachable via its *West* interface. Since it does not have an entry for destination *B* in its port-address table, it forwards the packet to both *R2* and *R3*. When *R2* receives the packet, it updates its own forwarding table and forward the packet to *C*. Since *C* is not the intended recipient, it simply discards the received packet. Router *R3* also receives the packet. It learns that *A* is reachable via its *North-West* interface and broadcasts the packet to *R4* and *R5*. *R5* also updates its forwarding table and finally forwards it to destination *B*. Let us now consider what happens when *B* sends a reply to *A*. *R5* first learns that *B* is attached to its *North-East* port. It then consults its port-address table and finds that *A* is reachable via its *North-West* interface. The packet is then forwarded hop-by-hop to *A* without any broadcasting. Later on, if *C* sends a packet to *B*, this packet will reach *R1* that contains a valid forwarding entry in its forwarding table.

By inspecting the source and destination addresses of packets, network nodes can automatically derive their forwarding tables. As we will discuss later, this technique is used in *Ethernet* networks. Despite being widely used, it has two important drawbacks. First, packets sent to unknown destinations are broadcasted in the network even if the destination is not attached to the network. Consider the transmission of ten packets destined to *Z* in the network above. When a node receives a packet towards this destination, it can only broadcast that packet. Since *Z* is not attached to the network, no node will ever receive a packet whose source is *Z* to update its forwarding table. The second and more important problem is that few networks have a tree-shaped topology. It is interesting to analyze what happens when a port-address table is used in a network that contains a cycle. Consider the simple network shown below with a single host.

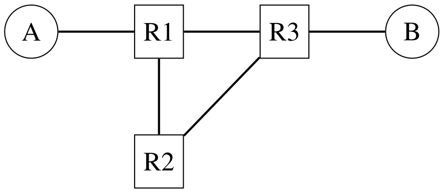


Fig. 21: A simple and redundant network

Assume that the network has started and all port-address and forwarding tables are empty. Host *A* sends a packet towards *B*. Upon reception of this packet, *R1* updates its port-address table. Since *B* is not present in the port-address table, the packet is broadcasted. Both *R2* and *R3* receive a copy of the packet sent by *A*. They both update their port-address table. Unfortunately, they also both broadcast the received packet. *B* receives a first copy of the packet, but *R3* and *R2* receive it again. *R3* will then broadcast this copy of the packet to *B* and *R1* while *R2* will broadcast its copy to *R1*. Although *B* has already received two copies of the packet, it is still inside the network and continues to loop. Due to the presence of the cycle, a single packet towards an unknown destination generates many copies of this packet that loop and will eventually saturate the network. Network operators who are using port-address tables to automatically compute the forwarding tables also use distributed algorithms to ensure that the network topology is always a tree.

Another technique called *source routing* can be used to automatically compute forwarding tables. It has been used in interconnecting Token Ring networks and in some wireless networks. Intuitively, *source routing* enables a destination to automatically discover the paths from a given source towards itself. This technique requires nodes to encode information inside some packets. For simplicity, let us assume that the *data plane* supports two types of packets :

- the *data packets*
- the *control packets*

Data packets are used to exchange data while *control packets* are used to discover the paths between hosts. With *source routing*, routers can be kept as simple as possible and all the complexity is placed on the hosts. This is in contrast with the previous technique where the nodes had to maintain a port-address and a forwarding table while the hosts simply sent and received packets. Each node is configured with one unique address and there is one identifier per outgoing link. For simplicity and to avoid cluttering the figures with those identifiers, we assume that each node uses as link identifiers north, west, south,… In practice, a node would associate one integer to each outgoing link.

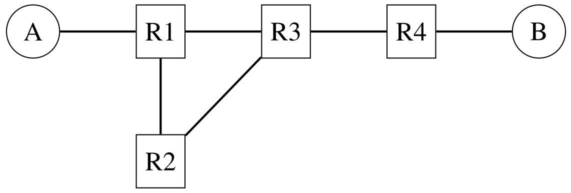


Fig. 22: A simple network with two hosts and four routers

In the network above, router *R2* is attached to two outgoing links. *R2* is connected to both *R1* and *R3*. *R2* can easily determine that it is connected to these two nodes by exchanging packets with them or observing the packets that it receives over each interface. Assume for example that when a node (either host or router) starts, it sends a special control packet over each of its interfaces to advertise its own address to its neighbors. When a node receives such a packet, it automatically replies with its own address. This exchange can also be used to verify whether a neighbor, either router or host, is still alive. With *source routing*, the data plane packets include a list of identifiers. This list is called a *source route*. It indicates the path to be followed by the packet as a sequence of link identifiers. When a node receives such a *data plane* packet, it first checks whether the packet's destination is a direct neighbor. In this case, the packet is forwarded to this neighbor. Otherwise, the node extracts the next address from the list and forwards it to the neighbor. This allows the source to specify the explicit path to be followed for each packet. For example, in the figure above there are two possible paths between *A* and *B*. To use the path via *R2*, *A* would send a packet that contains *R1,R2,R3* as source route. To avoid going via *R2*, *A* would place *R1,R3* as the source route in its transmitted packet. If *A* knows the complete network topology and all link identifiers, it can easily compute the source route towards each destination. It can even use different paths, e.g. for redundancy, to reach a given destination. However, in a real network hosts do not usually have a map of the entire network topology.

In networks that rely on source routing, hosts use control packets to automatically discover the best path(s). In addition to the source and destination addresses, *control packets* contain a list that records the intermediate nodes. This list is often called the *record route* because it allows recording the route followed by a given packet. When a node receives such a *control packet*, it first checks whether its address is included in the record route. If yes, the packet has already been forwarded by this node and it is silently discarded. Otherwise, it adds its own address to the *record route* and forwards the packet to all its interfaces, except the interface over which the packet has been received. Thanks to this, the *control packet* can explore all paths between a source and a given destination.

For example, consider again the network topology above. *A* sends a control packet towards *B*. The initial *record route* is empty. When *R1* receives the packet, it adds its own address to the *record route* and forwards a copy to *R2* and another to *R3*. *R2* receives the packet, adds itself to the *record route* and forwards it to *R3*. *R3* receives two copies of the packet. The first contains the *[R1,R2] record route* and the second *[R1]*. In the end, *B* will receive two control packets containing *[R1,R2,R3,R4]* and *[R1,R3,R4]* as *record routes*. *B* can keep these two paths or select the best one and discard the second. A popular heuristic is to select the *record route* of the first received packet as being the best one since this likely corresponds to the shortest delay path.

With the received *record route*, *B* can send a *data packet* to *A*. For this, it simply reverses the chosen *record route*. However, we still need to communicate the chosen path to *A*. This can be done by putting the *record route* inside a control packet which is sent back to *A* over the reverse path. An alternative is to simply send a *data packet* back to *A*. This packet will travel back to *A*. To allow *A* to inspect the entire path followed by the *data packet*, its *source route* must contain all intermediate routers when it is received by *A*. This can be achieved by encoding the *source route* using a data structure that contains an index and the ordered list of node addresses. The index always points to the next address in the *source route*. It is initialized at *0* when a packet is created and incremented by each intermediate node.

The third technique to compute forwarding tables is to rely on a control plane using a distributed algorithm. Routers exchange control messages to discover the network topology and build their forwarding table based on them. We dedicate a more detailed description of such distributed algorithms later in this section.

Flat or hierarchical addresses

The last, but important, point to discuss about the *data plane* of the networks that rely on the datagram mode is their addressing scheme. In the examples above, we have used letters to represent the addresses of the hosts and network nodes. In practice, all addresses are encoded as a bit string. Most network technologies use a fixed size bit string to represent source and destination address. These addresses can be organized in two different ways.

The first organization, which is the one that we have implicitly assumed until now, is the *flat addressing* scheme. Under this scheme, each host and network node has a unique address. The unicity of the addresses is important for the operation of the network. If two hosts have the same address, it can become difficult for the network to forward packets towards this destination. *Flat addresses* are typically used in situations where network nodes and hosts need to be able to communicate immediately with unique addresses. These *flat addresses* are often embedded inside the network interface cards. The network card manufacturer creates one unique address for each interface and this address is stored in the read-only memory of the interface. An advantage of this addressing scheme is that it easily supports unstructured and mobile networks. When a host moves, it can attach to another network and remain confident that its address is unique and enables it to communicate inside the new network.

With *flat addressing* the lookup operation in the forwarding table can be implemented as an exact match. The *forwarding table* contains the (sorted) list of all known destination addresses. When a packet arrives, a network node only needs to check whether this address is included in the forwarding table or not. In software, this is an *O(log(n))* operation if the list is sorted. In hardware, Content Addressable Memories can efficiently perform this lookup operation, but their size is usually limited.

A drawback of the *flat addressing scheme* is that the forwarding tables linearly grow with the number of hosts and routers in the network. With this addressing scheme, each forwarding table must contain an entry that points to every address reachable inside the network. Since large networks can contain tens of millions of hosts or more, this is a major problem on routers that need to be able to quickly forward packets. As an illustration, it is interesting to consider the case of an interface running at 10 Gbps. Such interfaces are found on high-end servers and in various routers today. Assuming a packet size of 1000 bits, a pretty large and conservative number, such interface must forward ten million packets every second. This implies that a router that receives packets over such a link must forward one 1000 bits packet every 100 nanoseconds. This is the same order of magnitude as the memory access times of old DRAMs.

A widely used alternative to the *flat addressing scheme* is the *hierarchical addressing scheme*. This addressing scheme builds upon the fact that networks usually contain much more hosts than routers. In this case, a first solution to reduce the size of the forwarding tables is to create a hierarchy of addresses. This is the solution chosen by the post office since postal addresses contain a country, sometimes a state or province, a city, a street and finally a street number. When an envelope is forwarded by a post office in a remote country, it only looks at the destination country, while a post office in the same province will look at the city information. Only the post office responsible for a given city will look at the street name and only the postman will use the street number. *Hierarchical addresses* provide a similar solution for network addresses. For example, the address of an Internet host attached to a campus network could contain in the high-order bits an identification of the Internet Service Provider (ISP) that serves the campus network. Then, a subsequent block of bits identifies the campus network which is one of the customers of the ISP. Finally, the low order bits of the address identify the host in the campus network.

This hierarchical allocation of addresses can be applied in any type of network. In practice, the allocation of the addresses must follow the network topology. Usually, this is achieved by dividing the addressing space in consecutive blocks and then allocating these blocks to different parts of the network. In a small network, the simplest solution is to allocate one block of addresses to each network node and assign the host addresses from the attached node.

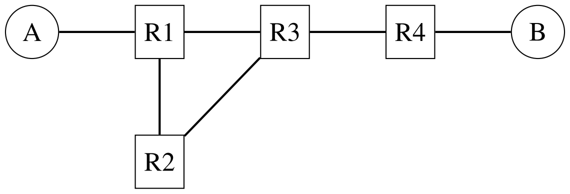


Fig. 23: A simple network with two hosts and four routers

In the above figure, assume that the network uses 16 bits addresses and that the prefix *01001010* has been assigned to the entire network. Since the network contains four routers, the network operator could assign one block of sixty-four addresses to each router. *R1* would use address *0100101000000000* while *A* could use address *0100101000000001*. *R2* could be assigned all addresses from *0100101001000000* to *0100101001111111*. *R4* could then use *0100101011000000* and assign *0100101011000001* to *B*. Other allocation schemes are possible. For example, *R3* could be allocated a larger block of addresses than *R2* and *R4* could use a sub-block from *R3* ‘s address block.

The main advantage of hierarchical addresses is that it is possible to significantly reduce the size of the forwarding tables. In many networks, the number of routers can be several orders of magnitude smaller than the number of hosts. A campus network may contain a dozen routers and thousands of hosts. The largest Internet Services Providers typically contain no more than a few tens of thousands of routers but still serve tens or hundreds of millions of hosts.

Despite their popularity, *hierarchical addresses* have some drawbacks. Their first drawback is that a lookup in the forwarding table is more complex than when using *flat addresses*. For example, on the Internet, network nodes have to perform a longest-match to forward each packet. This is partially compensated by the reduction in the size of the forwarding tables, but the additional complexity of the lookup operation has been a difficulty to implement hardware support for packet forwarding. A second drawback of the utilization of hierarchical addresses is that when a host connects for the first time to a network, it must contact one router to determine its own address. This requires some packet exchanges between the host and some routers. Furthermore, if a host moves and is attached to another routers, its network address will change. This can be an issue with some mobile hosts.

Dealing with heterogeneous datalink layers

Sometimes, the network layer needs to deal with heterogeneous datalink layers. For example, two hosts connected to different datalink layers exchange packets via routers that are using other types of datalink layers. Thanks to the network layer, this exchange of packets is possible provided that each packet can be placed inside a datalink layer frame before being transmitted. If all datalink layers support the same frame size, this is simple. When a node receives a frame, it decapsulates the packet that it contains, checks the header and forwards it, encapsulated inside another frame, to the outgoing interface. Unfortunately, the encapsulation operation is not always possible. Each datalink layer is characterized by the maximum frame size that it supports. Datalink layers typically support frames containing up to a few hundreds or a few thousands of bytes. The maximum frame size that a given datalink layer supports depends on its underlying technology. Unfortunately, most datalink layers support a different maximum frame size. This implies that when a host sends a large packet inside a frame to its nexthop router, there is a risk that this packet will have to traverse a link that is not capable of forwarding the packet inside a single frame. In principle, there are three possibilities to solve this problem. To discuss them, we consider a simple scenario with two hosts connected to a router as shown in the figure below.

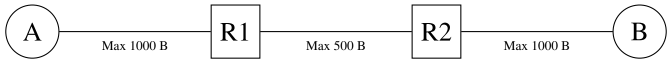


Fig. 24: A simple heterogeneous network

Consider in the network above that host *A* wants to send a 900 bytes packet (870 bytes of payload and 30 bytes of header) to host *B* via router *R1*. Host *A* encapsulates this packet inside a single frame. The frame is received by router *R1* which extracts the packet. Router *R1* has three possible options to process this packet.

1. The packet is too large and router *R1* cannot forward it to router *R2*. It rejects the packet and sends a control packet back to the source (host *A*) to indicate that it cannot forward packets longer than 500 bytes (minus the packet header). The source could react to this control packet by retransmitting the information in smaller packets.
2. The network layer is able to fragment a packet. In our example, the router could fragment the packet in two parts. The first part contains the beginning of the payload and the second the end. There are two possible ways to perform this fragmentation.
3. Router *R1* fragments the packet into two fragments before transmitting them to router *R2*. Router *R2* reassembles the two packet fragments in a larger packet before transmitting them on the link towards host *B*.
4. Each of the packet fragments is a valid packet that contains a header with the source (host *A*) and destination (host *B*) addresses. When router *R2* receives a packet fragment, it treats this packet as a regular packet and forwards it to its final destination (host *B*). Host *B* reassembles the received fragments.

These three solutions have advantages and drawbacks. With the first solution, routers remain simple and do not need to perform any fragmentation operation. This is important when routers are implemented mainly in hardware. However, hosts must be complex since they need to store the packets that they produce if they need to pass through a link that does not support large packets. This increases the buffering required on the hosts.

Furthermore, a single large packet may potentially need to be retransmitted several times. Consider for example a network similar to the one shown above but with four routers. Assume that the link *R1->R2* supports 1000 bytes packets, link *R2->R3* 800 bytes packets and link *R3->R4* 600 bytes packets. A host attached to *R1* that sends large packet will have to first try 1000 bytes, then 800 bytes and finally 600 bytes. Fortunately, this scenario does not occur very often in practice and this is the reason why this solution is used in real networks.

Fragmenting packets on a per-link basis, as presented for the second solution, can minimize the transmission overhead since a packet is only fragmented on the links where fragmentation is required. Large packets can continue to be used downstream of a link that only accepts small packets. However, this reduction of the overhead comes with two drawbacks. First, fragmenting packets, potentially on all links, increases the processing time and the buffer requirements on the routers. Second, this solution leads to a longer end-to-end delay since the downstream router has to reassemble all the packet fragments before forwarding the packet.

The last solution is a compromise between the two others. Routers need to perform fragmentation but they do not need to reassemble packet fragments. Only the hosts need to have buffers to reassemble the received fragments. This solution has a lower end-to-end delay and requires fewer processing time and memory on the routers.

The first solution to the fragmentation problem presented above suggests the utilization of control packets to inform the source about the reception of a too long packet. This is only one of the functions that are performed by the control protocol in the network layer. Other functions include :

- sending a control packet back to the source if a packet is received by a router that does not have a valid entry in its forwarding table
- sending a control packet back to the source if a router detects that a packet is looping inside the network
- verifying that packets can reach a given destination

We will discuss these functions in more details when we will describe the protocols that are used in the network layer of the TCP/IP protocol suite.

2.2.2 Virtual circuit organization

The second organization of the network layer, called *virtual circuits*, has been inspired by the organization of telephone networks. Telephone networks have been designed to carry phone calls that usually last a few minutes. Each phone is identified by a telephone number and is attached to a telephone switch. To initiate a phone call, a telephone first needs to send the destination's phone number to its local switch. The switch cooperates with the other switches in the network to create a bi-directional channel between the two telephones through the network. This channel will be used by the two telephones during the lifetime of the call and will be released at the end of the call. Until the 1960s, most of these channels were created manually, by telephone operators, upon request of the caller. Today's telephone networks use automated switches and allow several channels to be carried over the same physical link, but the principles roughly remain the same.

In a network using virtual circuits, all hosts are also identified with a network layer address. However, packet forwarding is not performed by looking at the destination address of each packet. With the *virtual circuit* organization, each data packet contains one label[1]. A label is an integer which is part of the packet header. Routers implement *label switching* to forward *labelled data packet*. Upon reception of a packet, a router consults its *label forwarding table* to find the outgoing interface for this packet. In contrast with the datagram mode, this lookup is very simple. The *label forwarding table* is an array stored in memory and the label of the incoming packet is the index to access this array. This implies that the lookup operation has an *O(1)* complexity in contrast with other packet forwarding techniques. To ensure that on each node the packet label is an index in the *label forwarding table*, each router that forwards a packet replaces the label of the forwarded packet with the label found in the *label forwarding table*. Each entry of the *label forwarding table* contains two pieces of information :

- the outgoing interface for the packet
- the label for the outgoing packet

For example, consider the *label forwarding table* of a network node below.

index	outgoing interface	label
0	South	7
1	none	none
2	West	2
3	East	2

If this node receives a packet with *label=2*, it forwards the packet on its *West* interface and sets the *label* of the outgoing packet to *2*. If the received packet's *label* is set to *3*, then the packet is forwarded over the *East* interface and the *label* of the outgoing packet is set to *2*. If a packet is received with a label field set to *1*, the packet is discarded since the corresponding *label forwarding table* entry is invalid.

Label switching enables a full control over the path followed by packets inside the network. Consider the network below and assume that we want to use two virtual circuits : *R1->R3->R4->R2->R5* and *R2->R1->R3->R4->R5*.

To create these virtual circuits, we need to configure the *label forwarding tables* of all routers. For simplicity, assume that a label forwarding table only contains two entries. Assume that *R5* wants to receive the packets from the virtual circuit created by *R1* (resp. *R2*) with *label=1* (*label=0*). *R4* could use the following *label forwarding table*:

[1] We will see later a more detailed description of Multiprotocol Label Switching, a networking technology that is capable of using one or more labels.

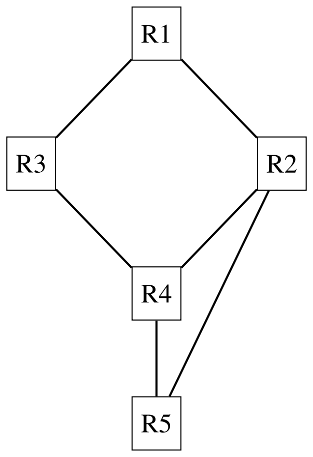


Fig. 25: An example of network where label switching can be applied to tune its paths

R4's label forwarding table		
index	outgoing interface	label
0	->R2	1
1	->R5	0

Since a packet received with *label=1* must be forwarded to *R5* with *label=1*, *R2*'s *label forwarding table* could contain :

R2's label forwarding table		
index	outgoing interface	label
0	none	none
1	->R5	1

Two virtual circuits pass through *R3*. They both need to be forwarded to *R4*, but *R4* expects *label=1* for packets belonging to the virtual circuit originated by *R2* and *label=0* for packets belonging to the other virtual circuit. *R3* could choose to leave the labels unchanged.

R3's label forwarding table		
index	outgoing interface	label
0	->R4	0
1	->R4	1

With the above *label forwarding table*, *R1* needs to originate the packets that belong to the *R1->R3->R4->R2->R5* circuit with *label=0*. The packets received from *R2* and belonging to the *R2->R1->R3->R4->R5* circuit would then use *label=1* on the *R1-R3* link. *R1* 's label forwarding table could be built as follows :

R1's label forwarding table		
index	outgoing interface	label
0	->R3	0
1	->R3	1

The figure below shows the path followed by the packets on the *R1->R3->R4->R2->R5* path in red with on each arrow the label used in the packets.

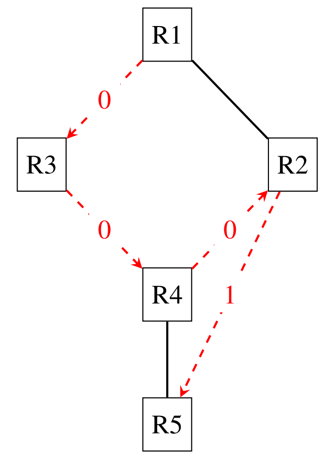


Fig. 26: The path followed by packets for a specific circuit

MultiProtocol Label Switching (MPLS) is the example of a deployed networking technology that relies on label switching. MPLS is more complex than the above description because it has been designed to be easily integrated with datagram technologies. However, the principles remain. *Asynchronous Transfer Mode* (ATM) and Frame Relay are other examples of technologies that rely on *label switching*.

Nowadays, most deployed networks rely on distributed algorithms, called routing protocols, to compute the forwarding tables that are installed on the routers. These distributed algorithms are part of the *control plane*. They are usually implemented in software and are executed on the main CPU of the routers. There are two main families of routing protocols : distance vector routing and link state routing. Both are capable of discovering autonomously the network and react dynamically to topology changes.

2.2.3 The control plane

One of the objectives of the *control plane* in the network layer is to maintain the routing tables that are used on all routers. As indicated earlier, a routing table is a data structure that contains, for each destination address (or block of addresses) known by the router, the outgoing interface over which the router must forward a packet destined to this address. The routing table may also contain additional information such as the address of the next router on the path towards the destination or an estimation of the cost of this path.

In this section, we discuss the main techniques that can be used to maintain the forwarding tables in a network.

Distance vector routing

Distance vector routing is a simple distributed routing protocol. Distance vector routing allows routers to automatically discover the destinations reachable inside the network as well as the shortest path to reach each of these destinations. The shortest path is computed based on *metrics* or *costs* that are associated to each link. We use *l.cost* to represent the metric that has been configured for link *l* on a router.

Each router maintains a routing table. The routing table *R* can be modeled as a data structure that stores, for each known destination address *d*, the following attributes :

- *R[d].link* is the outgoing link that the router uses to forward packets towards destination *d*
- *R[d].cost* is the sum of the metrics of the links that compose the shortest path to reach destination *d*
- *R[d].time* is the timestamp of the last distance vector containing destination *d*

A router that uses distance vector routing regularly sends its distance vector over all its interfaces. This distance vector is a summary of the router's routing table that indicates the distance towards each known destination. This distance vector can be computed from the routing table by using the pseudo-code below.

```
Every N seconds:
    v = Vector()
    for d in R[]:
        # add destination d to vector
        v.add(Pair(d, R[d].cost))
    for i in interfaces
        # send vector v on this interface
        send(v, i)
```

When a router boots, it does not know any destination in the network and its routing table only contains its local address(es). It thus sends to all its neighbors a distance vector that contains only its address at a distance of *0*. When a router receives a distance vector on link *l*, it processes it as follows.

```
# V : received Vector
# l : link over which vector is received
def received(V, l):
    # received vector from link l
    for d in V[]
        if not (d in R[]):
            # new route
            R[d].cost = V[d].cost + l.cost
            R[d].link = l
            R[d].time = now
        else:
            # existing route, is the new better ?
            if ((V[d].cost + l.cost) < R[d].cost) or (R[d].link == l):
                # Better route or change to current route
                R[d].cost = V[d].cost + l.cost
                R[d].link = l
                R[d].time = now
```

The router iterates over all addresses included in the distance vector. If the distance vector contains a destination address that the router does not know, it inserts it inside its routing table via link *l* and at a distance which is the sum between the distance indicated in the distance vector and the cost associated to link *l*. If the destination was already known by the router, it only updates the corresponding entry in its routing table if either :

- the cost of the new route is smaller than the cost of the already known route *((V[d].cost + l.cost) < R[d].cost)*
- the new route was learned over the same link as the current best route towards this destination *(R[d].link == l)*

The first condition ensures that the router discovers the shortest path towards each destination. The second condition is used to take into account the changes of routes that may occur after a link failure or a change of the metric associated to a link.

To understand the operation of a distance vector protocol, let us consider the network of five routers shown below.

Assume that router *A* is the first to send its distance vector *[A=0]*.

- *B* and *D* process the received distance vector and update their routing table with a route towards *A*.
- *D* sends its distance vector *[D=0,A=1]* to *A* and *E*. *E* can now reach *A* and *D*.
- *C* sends its distance vector *[C=0]* to *B* and *E*
- *E* sends its distance vector *[E=0,D=1,A=2,C=1]* to *D*, *B* and *C*. *B* can now reach *A*, *C*, *D* and *E*

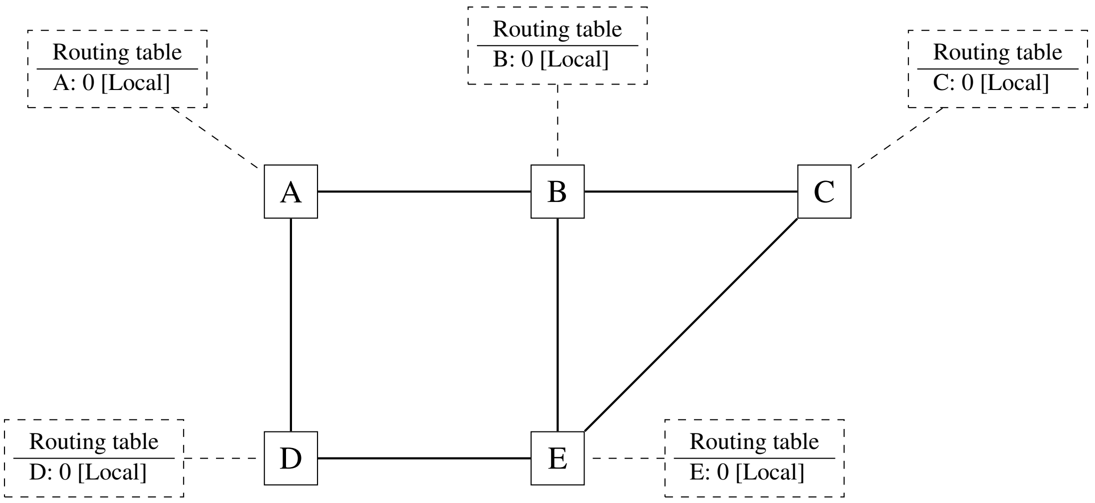


Fig. 27: Operation of distance vector routing in a simple network

- *B* sends its distance vector *[B=0,A=1,C=1,D=2,E=1]* to *A*, *C* and *E*. *A*, *B*, *C* and *E* can now reach all five routers of this network.
- *A* sends its distance vector *[A=0,B=1,C=2,D=1,E=2]* to *B* and *D*.

At this point, all routers can reach all other routers in the network thanks to the routing tables shown in the figure below.

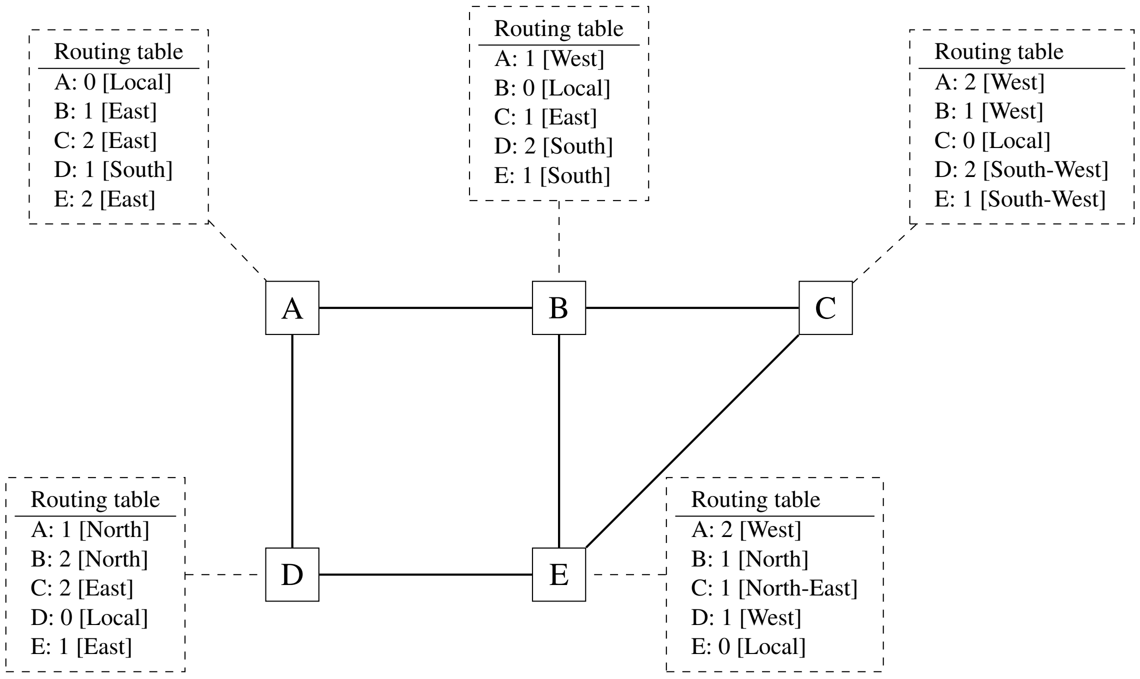


Fig. 28: Routing tables computed by distance vector in a simple network

To deal with link and router failures, routers use the timestamp stored in their routing table. As all routers send their distance vector every *N* seconds, the timestamp of each route should be regularly refreshed. Thus no route should have a timestamp older than *N* seconds, unless the route is not reachable anymore. In practice, to cope with the possible loss of a distance vector due to transmission errors, routers check the timestamp of the routes stored in their routing table every N seconds and remove the routes that are older than $3 \times N$ seconds.

When a router notices that a route towards a destination has expired, it must first associate an ∞ cost to this route and send its distance vector to its neighbors to inform them. The route can then be removed from the routing table after some time (e.g. $3 \times N$ seconds), to ensure that the neighboring routers have received the bad news, even if some distance vectors do not reach them due to transmission errors.

Consider the example above and assume that the link between routers *A* and *B* fails. Before the failure, *A* used *B* to reach destinations *B*, *C* and *E* while *B* only used the *A*-*B* link to reach *A*. The two routers detect the failure by the timeouts in the affected entries in their routing tables. Both routers *A* and *B* send their distance vector.

- *A* sends its distance vector $[A = 0, B = \infty, C = \infty, D = 1, E = \infty]$. *D* knows that it cannot reach *B* anymore via *A*
- *D* sends its distance vector $[D = 0, B = \infty, A = 1, C = 2, E = 1]$ to *A* and *E*. *A* recovers routes towards *C* and *E* via *D*.
- *B* sends its distance vector $[B = 0, A = \infty, C = 1, D = 2, E = 1]$ to *E* and *C*. *C* learns that there is no route anymore to reach *A* via *B*.
- *E* sends its distance vector $[E = 0, A = 2, C = 1, D = 1, B = 1]$ to *D*, *B* and *C*. *D* learns a route towards *B*. *C* and *B* learn a route towards *A*.

At this point, all routers have a routing table allowing them to reach all other routers, except router *A*, which cannot yet reach router *B*. *A* recovers the route towards *B* once router *D* sends its updated distance vector $[A = 1, B = 2, C = 2, D = 0, E = 1]$. This last step is illustrated in figure below, which shows the routing tables on all routers.

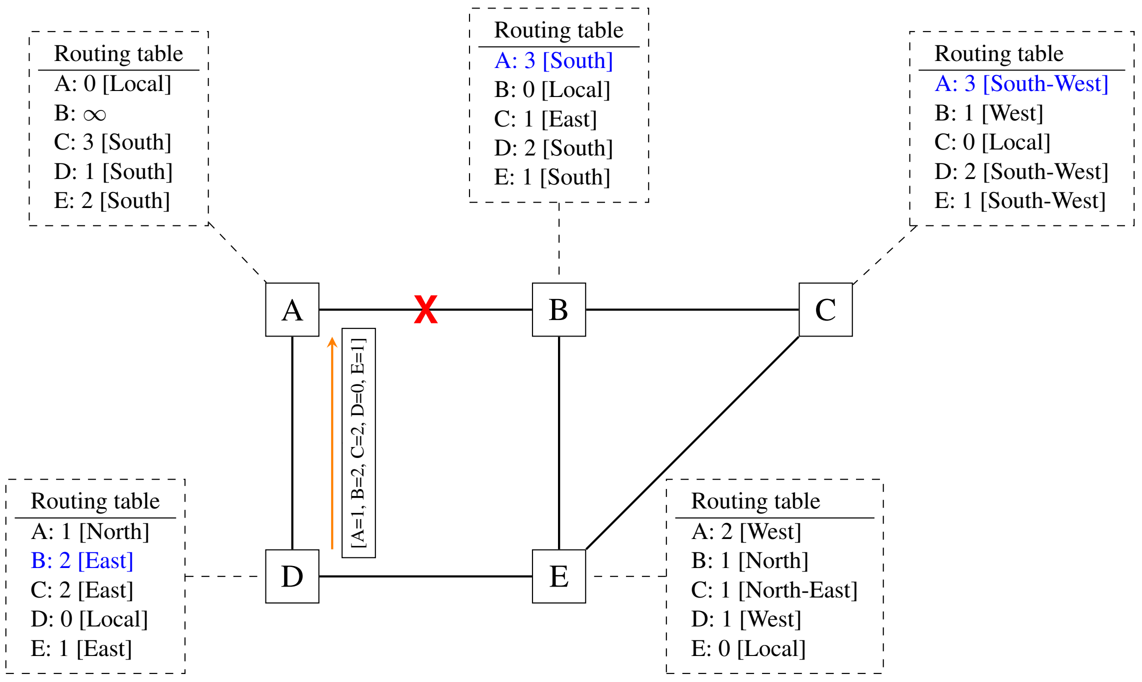


Fig. 29: Routing tables computed by distance vector after a failure

Consider now that the link between *D* and *E* fails. The network is now partitioned into two disjoint parts: (*A* , *D*) and (*B*, *E*, *C*). The routes towards *B*, *C* and *E* expire first on router *D*. At this time, router *D* updates its routing table.

If *D* sends $[D = 0, A = 1, B = \infty, C = \infty, E = \infty]$, *A* learns that *B*, *C* and *E* are unreachable and updates its routing table.

Unfortunately, if the distance vector sent to A is lost or if A sends its own distance vector ($[A = 0, D = 1, B = 3, C = 3, E = 2]$) at the same time as *D* sends its distance vector, *D* updates its routing table to use the shorter routes advertised by A towards *B*, *C* and *E*. After some time *D* sends a new distance vector : $[D = 0, A = 1, E = 3, C = 4, B = 4]$. A updates its routing table and after some time sends its own distance vector $[A = 0, D = 1, B = 5, C = 5, E = 4]$, etc. This problem is known as the *count to infinity problem* in the networking literature.

Routers A and *D* exchange distance vectors with increasing costs until these costs reach ∞. This problem may occur in other scenarios than the one depicted in the above figure. In fact, distance vector routing may suffer from count to infinity problems as soon as there is a cycle in the network. Unfortunately, cycles are widely used in networks since they provide the required redundancy to deal with link and router failures. To mitigate the impact of counting to infinity, some distance vector protocols consider that $16 = \infty$. Unfortunately, this limits the metrics that network operators can use and the diameter of the networks using distance vectors.

This count to infinity problem occurs because router A advertises to router *D* a route that it has learned via router *D*. A possible solution to avoid this problem could be to change how a router creates its distance vector. Instead of computing one distance vector and sending it to all its neighbors, a router could create a distance vector that is specific to each neighbor and only contains the routes that have not been learned via this neighbor. This could be implemented by the following pseudocode.

```
# split horizon
Every N seconds:
    # one vector for each interface
    for l in interfaces:
        v = Vector()
        for d in R[]:
            if (R[d].link != l):
                v = v + Pair(d, R[d].cost)
        send(v, l)
        # end for d in R[]
    # end for l in interfaces
```

This technique is called *split-horizon*. With this technique, the count to infinity problem would not have happened in the above scenario, as router A would have advertised $[A = 0]$ after the failure, since it learned all its other routes via router *D*. Another variant called *split-horizon with poison reverse* is also possible. Routers using this variant advertise a cost of ∞ for the destinations that they reach via the router to which they send the distance vector. This can be implemented by using the pseudo-code below.

```
# split horizon with poison reverse
Every N seconds:
    for l in interfaces:
        # one vector for each interface
        v = Vector()
        for d in R[]:
            if (R[d].link != l):
                v = v + Pair(d, R[d].cost)
            else:
                v = v + Pair(d, infinity)
        send(v, l)
        # end for d in R[]
    # end for l in interfaces
```

Unfortunately, split-horizon is not sufficient to avoid all count to infinity problems with distance vector routing. Consider the failure of link *A-B* in the four routers network shown below.

After having detected the failure, router *B* sends its distance vectors:

- $[A = \infty, B = 0, C = \infty, E = 1]$ to router *C*

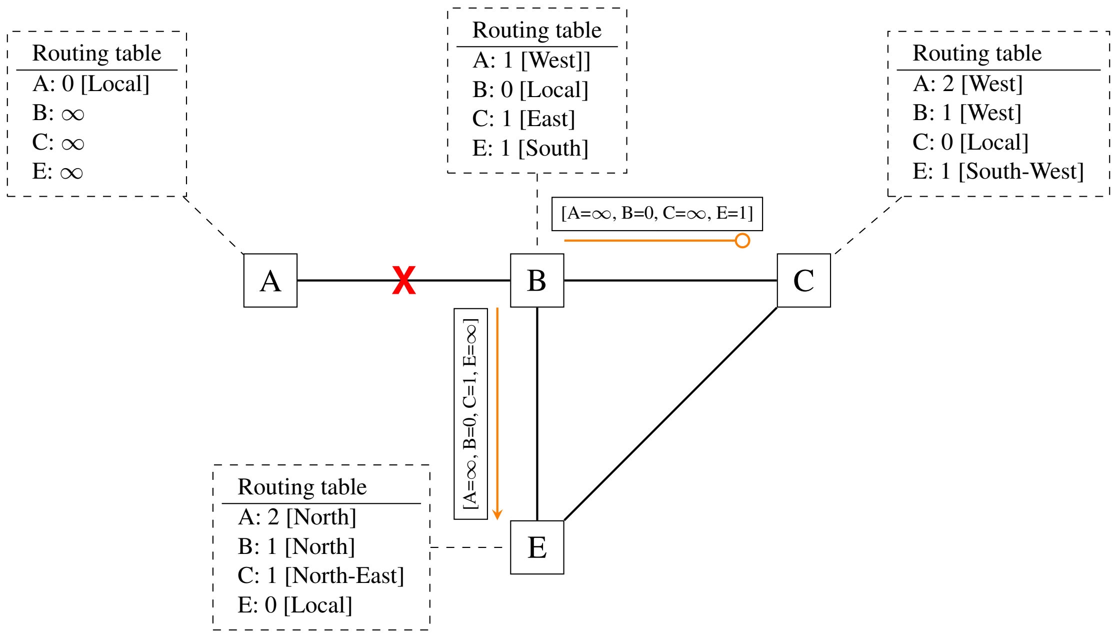


Fig. 30: Count to infinity problem

- $[A = \infty, B = 0, C = 1, E = \infty]$ to router *E*

If, unfortunately, the distance vector sent to router *C* is lost due to a transmission error or because router *C* is overloaded, a new count to infinity problem can occur. If router *C* sends its distance vector $[A = 2, B = 1, C = 0, E = \infty]$ to router *E*, this router installs a route of distance *3* to reach *A* via *C*. Router *E* sends its distance vectors $[A = 3, B = \infty, C = 1, E = 1]$ to router *B* and $[A = \infty, B = 1, C = \infty, E = 0]$ to router *C*. This distance vector allows *B* to recover a route of distance *4* to reach *A*.

Note: Forwarding tables versus routing tables

Routers usually maintain at least two data structures that contain information about the reachable destinations. The first data structure is the *routing table*. The *routing table* is a data structure that associates a destination to an outgoing interface or a nexthop router and a set of additional attributes. Different routing protocols can associate different attributes for each destination. Distance vector routing protocols will store the cost to reach the destination along the shortest path. Other routing protocols may store information about the number of hops of the best path, its lifetime or the number of sub paths. A *routing table* may store different paths towards a given destination and flag one of them as the best one.

The *routing table* is a software data structure which is updated by (one or more) routing protocols. The *routing table* is usually not directly used when forwarding packets. Packet forwarding relies on a more compact data structure which is the *forwarding table*. On high-end routers, the *forwarding table* is implemented directly in hardware while lower performance routers will use a software implementation. A *forwarding table* contains a subset of the information found in the *routing table*. It only contains the nexthops towards each destination that are used to forward packets and no attributes. A *forwarding table* will typically associate each destination to one or more outgoing interface or nexthop router.

Link state routing

Link state routing is the second family of routing protocols. While distance vector routers use a distributed algorithm to compute their routing tables, link-state routers exchange messages to allow each router to learn the entire network topology. Based on this learned topology, each router is then able to compute its routing table by using a shortest path computation such as Dijkstra's algorithm [Dijkstra1959]. A detailed description of this shortest path algorithm may be found in [Wikipedia:Dijkstra].

For link-state routing, a network is modeled as a *directed weighted graph*. Each router is a node, and the links between routers are the edges in the graph. A positive weight is associated to each directed edge and routers use the shortest path to reach each destination. In practice, different types of weights can be associated to each directed edge :

- unit weight. If all links have a unit weight, shortest path routing prefers the paths with the least number of intermediate routers.
- weight proportional to the propagation delay on the link. If all link weights are configured this way, shortest path routing uses the paths with the smallest propagation delay.
- $weight = \frac{C}{bandwidth}$ where C is a constant larger than the highest link bandwidth in the network. If all link weights are configured this way, shortest path routing prefers higher bandwidth paths over lower bandwidth paths.

Usually, the same weight is associated to the two directed edges that correspond to a physical link (i.e. $R1 \rightarrow R2$ and $R2 \rightarrow R1$). However, nothing in the link state protocols requires this. For example, if the weight is set in function of the link bandwidth, then an asymmetric ADSL link could have a different weight for the upstream and downstream directions. Other variants are possible. Some networks use optimization algorithms to find the best set of weights to minimize congestion inside the network for a given traffic demand [FRT2002].

When a link-state router boots, it first needs to discover to which routers it is directly connected. For this, each router sends a HELLO message every N seconds on all its interfaces. This message contains the router's address. Each router has a unique address. As its neighboring routers also send HELLO messages, the router automatically discovers to which neighbors it is connected. These HELLO messages are only sent to neighbors that are directly connected to a router, and a router never forwards the HELLO messages that it receives. HELLO messages are also used to detect link and router failures. A link is considered to have failed if no HELLO message has been received from a neighboring router for a period of $k \times N$ seconds.

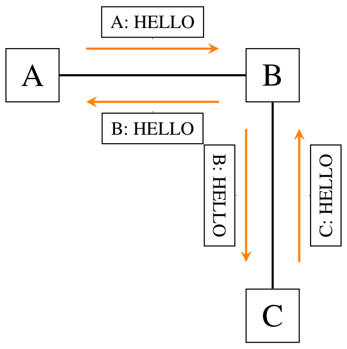


Fig. 31: The exchange of HELLO messages

Once a router has discovered its neighbors, it must reliably distribute all its outgoing edges to all routers in the network to allow them to compute their local view of the network topology. For this, each router builds a *link-state packet* (LSP) containing the following information:

- LSP.Router: identification (address) of the sender of the LSP
- LSP.age: age or remaining lifetime of the LSP

- LSP.seq: sequence number of the LSP
- LSP.Links[]: links advertised in the LSP. Each directed link is represented with the following information:
  - LSP.Links[i].Id: identification of the neighbor
  - LSP.Links[i].cost: cost of the link

These LSPs must be reliably distributed inside the network without using the router's routing table since these tables can only be computed once the LSPs have been received. The *Flooding* algorithm is used to efficiently distribute the LSPs of all routers. Each router that implements *flooding* maintains a *link state database* (LSDB) containing the most recent LSP sent by each router. When a router receives a LSP, it first verifies whether this LSP is already stored inside its LSDB. If so, the router has already distributed the LSP earlier and it does not need to forward it. Otherwise, the router forwards the LSP on all its links except the link over which the LSP was received. Flooding can be implemented by using the following pseudo-code.

```
# links is the set of all links on the router
# Router R's LSP arrival on link l
if newer(LSP, LSDB(LSP.Router)) :
    LSDB.add(LSP)  # implicitly removes older LSP from same router
    for i in links:
        if i!=l:
           send(LSP,i)
# else, LSP has already been flooded
```

In this pseudo-code, *LSDB(r)* returns the most recent *LSP* originating from router *r* that is stored in the *LSDB*. *newer(lsp1, lsp2)* returns true if *lsp1* is more recent than *lsp2*. See the note below for a discussion on how *newer* can be implemented.

Note: Which is the most recent LSP ?

A router that implements flooding must be able to detect whether a received LSP is newer than the stored LSP. This requires a comparison between the sequence number of the received LSP and the sequence number of the LSP stored in the link state database. The ARPANET routing protocol [MRR1979] used a 6 bits sequence number and implemented the comparison as follows **RFC 789**

```
def newer( lsp1, lsp2 ):
    return ( ((lsp1.seq > lsp2.seq) and ((lsp1.seq - lsp2.seq) <= 32)) or
           ( (lsp1.seq < lsp2.seq) and ((lsp2.seq - lsp1.seq) > 32)) )
```

This comparison takes into account the modulo 2^6 arithmetic used to increment the sequence numbers. Intuitively, the comparison divides the circle of all sequence numbers into two halves. Usually, the sequence number of the received LSP is equal to the sequence number of the stored LSP incremented by one, but sometimes the sequence numbers of two successive LSPs may differ, e.g. if one router has been disconnected for some time. The comparison above worked well until October 27, 1980. On this day, the ARPANET crashed completely. The crash was complex and involved several routers. At one point, LSP *40* and LSP *44* from one of the routers were stored in the LSDB of some routers in the ARPANET. As LSP *44* was the newest, it should have replaced LSP *40* on all routers. Unfortunately, one of the ARPANET routers suffered from a memory problem and sequence number *40* (*101000* in binary) was replaced by *8* (*001000* in binary) in the buggy router and flooded. Three LSPs were present in the network and *44* was newer than *40* which is newer than *8*, but unfortunately *8* was considered to be newer than *44*… All routers started to exchange these three link state packets forever and the only solution to recover from this problem was to shutdown the entire network **RFC 789**.

Current link state routing protocols usually use 32 bits sequence numbers and include a special mechanism in the unlikely case that a sequence number reaches the maximum value (with a 32 bits sequence number space, it takes 136 years to cycle the sequence numbers if a link state packet is generated every second).

To deal with the memory corruption problem, link state packets contain a checksum or CRC. This checksum is computed by the router that generates the LSP. Each router must verify the checksum when it receives or floods an LSP. Furthermore, each router must periodically verify the checksums of the LSPs stored in its LSDB. This enables them to cope with memory errors that could corrupt the LSDB as the one that occurred in the ARPANET.

Flooding is illustrated in the figure below. By exchanging HELLO messages, each router learns its direct neighbors. For example, router *E* learns that it is directly connected to routers *D*, *B* and *C*. Its first LSP has sequence number *0* and contains the directed links *E->D*, *E->B* and *E->C*. Router *E* sends its LSP on all its links and routers *D*, *B* and *C* insert the LSP in their LSDB and forward it over their other links.

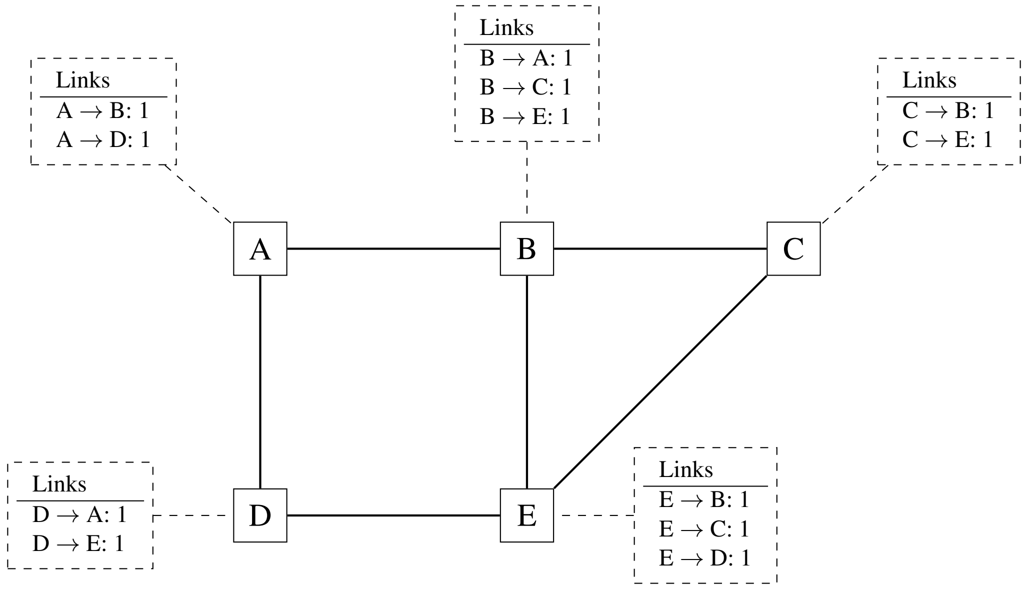


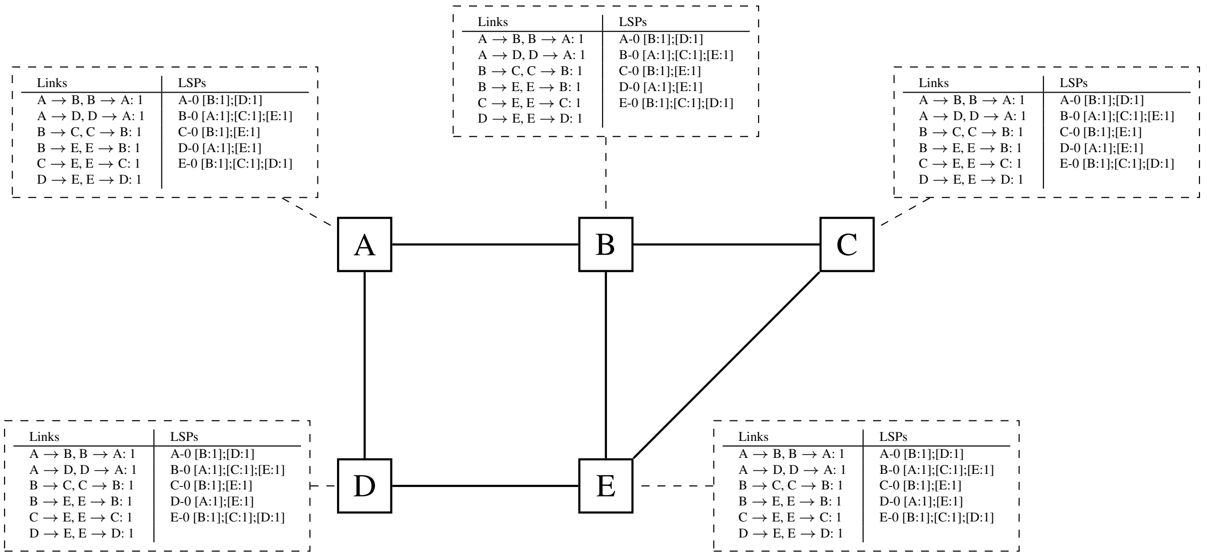
Fig. 32: Flooding: example

Flooding allows LSPs to be distributed to all routers inside the network without relying on routing tables. In the example above, the LSP sent by router *E* is likely to be sent twice on some links in the network. For example, routers *B* and *C* receive *E*'s LSP at almost the same time and forward it over the *B-C* link. To avoid sending the same LSP twice on each link, a possible solution is to slightly change the pseudo-code above so that a router waits for some random time before forwarding a LSP on each link. The drawback of this solution is that the delay to flood an LSP to all routers in the network increases. In practice, routers immediately flood the LSPs that contain new information (e.g. addition or removal of a link) and delay the flooding of refresh LSPs (i.e. LSPs that contain exactly the same information as the previous LSP originating from this router) [FFEB2005].

To ensure that all routers receive all LSPs, even when there are transmissions errors, link state routing protocols use *reliable flooding*. With *reliable flooding*, routers use acknowledgments and if necessary retransmissions to ensure that all link state packets are successfully transferred to each neighboring router. Thanks to reliable flooding, all routers store in their LSDB the most recent LSP sent by each router in the network. By combining the received LSPs with its own LSP, each router can build a graph that represents the entire network topology.

Note: Static or dynamic link metrics ?

As link state packets are flooded regularly, routers are able to measure the quality (e.g. delay or load) of their links and adjust the metric of each link according to its current quality. Such dynamic adjustments were included in the ARPANET routing protocol [MRR1979] . However, experience showed that it was difficult to tune the dynamic



Links	LSPs
A → B, B → A: 1	A-0 [B:1];[D:1]
A → D, D → A: 1	B-0 [A:1];[C:1];[E:1]
B → C, C → B: 1	C-0 [B:1];[E:1]
B → E, E → B: 1	D-0 [A:1];[E:1]
C → E, E → C: 1	E-0 [B:1];[C:1];[D:1]
D → E, E → D: 1	

Links	LSPs
A → B, B → A: 1	A-0 [B:1];[D:1]
A → D, D → A: 1	B-0 [A:1];[C:1];[E:1]
B → C, C → B: 1	C-0 [B:1];[E:1]
B → E, E → B: 1	D-0 [A:1];[E:1]
C → E, E → C: 1	E-0 [B:1];[C:1];[D:1]
D → E, E → D: 1	

Links	LSPs
A → B, B → A: 1	A-0 [B:1];[D:1]
A → D, D → A: 1	B-0 [A:1];[C:1];[E:1]
B → C, C → B: 1	C-0 [B:1];[E:1]
B → E, E → B: 1	D-0 [A:1];[E:1]
C → E, E → C: 1	E-0 [B:1];[C:1];[D:1]
D → E, E → D: 1	

Links	LSPs
A → B, B → A: 1	A-0 [B:1];[D:1]
A → D, D → A: 1	B-0 [A:1];[C:1];[E:1]
B → C, C → B: 1	C-0 [B:1];[E:1]
B → E, E → B: 1	D-0 [A:1];[E:1]
C → E, E → C: 1	E-0 [B:1];[C:1];[D:1]
D → E, E → D: 1	

Links	LSPs
A → B, B → A: 1	A-0 [B:1];[D:1]
A → D, D → A: 1	B-0 [A:1];[C:1];[E:1]
B → C, C → B: 1	C-0 [B:1];[E:1]
B → E, E → B: 1	D-0 [A:1];[E:1]
C → E, E → C: 1	E-0 [B:1];[C:1];[D:1]
D → E, E → D: 1	

Fig. 33: Link state databases received by all routers

adjustments and ensure that no forwarding loops occur in the network [KZ1989]. Today's link state routing protocols use metrics that are manually configured on the routers and are only changed by the network operators or network management tools [FRT2002].

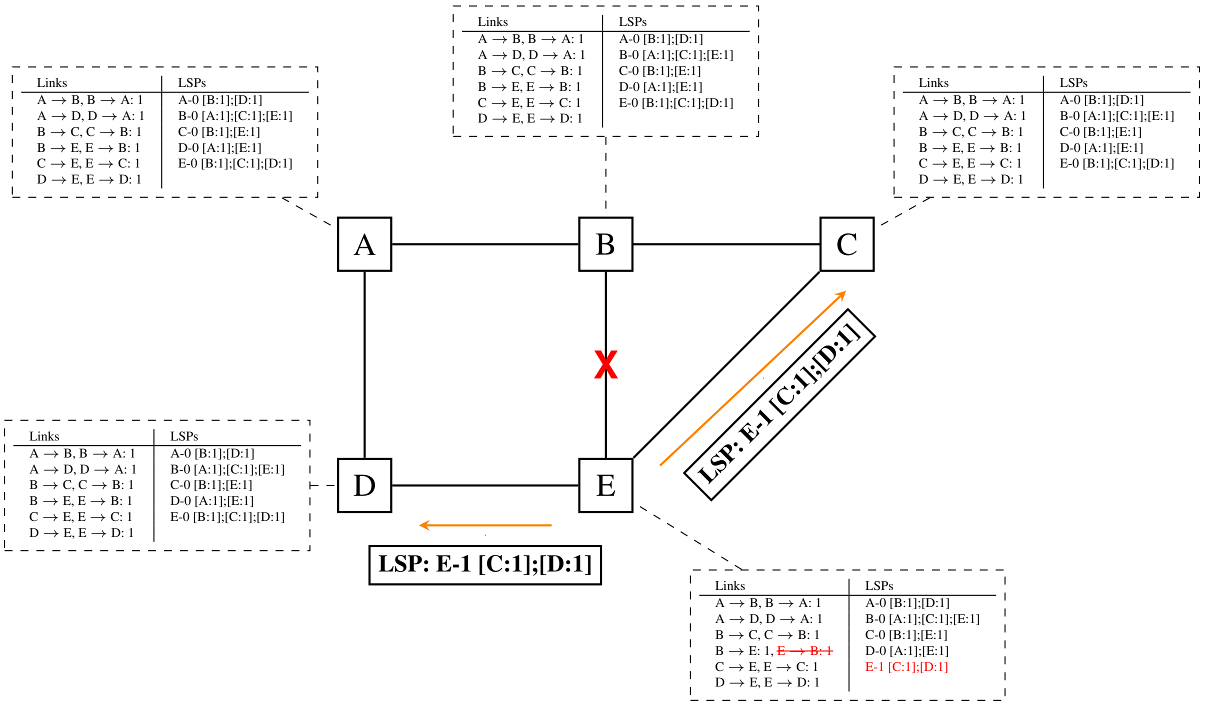
When a link fails, the two routers attached to the link detect the failure by the absence of HELLO messages received during the last $k \times N$ seconds. Once a router has detected the failure of one of its local links, it generates and floods a new LSP that no longer contains the failed link. This new LSP replaces the previous LSP in the network. In practice, the two routers attached to a link do not detect this failure exactly at the same time. During this period, some links may be announced in only one direction. This is illustrated in the figure below. Router *E* has detected the failure of link *E-B* and flooded a new LSP, but router *B* has not yet detected this failure.

When a link is reported in the LSP of only one of the attached routers, routers consider the link as having failed and they remove it from the directed graph that they compute from their LSDB. This is called the *two-way connectivity check*. This check allows link failures to be quickly flooded as a single LSP is sufficient to announce such bad news. However, when a link comes up, it can only be used once the two attached routers have sent their LSPs. The *two-way connectivity check* also allows for dealing with router failures. When a router fails, all its links fail by definition. These failures are reported in the LSPs sent by the neighbors of the failed router. The failed router does not, of course, send a new LSP to announce its failure. However, in the graph that represents the network, this failed router appears as a node that only has outgoing edges. Thanks to the *two-way connectivity check*, this failed router cannot be considered as a transit router to reach any destination since no outgoing edge is attached to it.

When a router has failed, its LSP must be removed from the LSDB of all routers[2]. This can be done by using the *age* field that is included in each LSP. The *age* field is used to bound the maximum lifetime of a link state packet in the network. When a router generates a LSP, it sets its lifetime (usually measured in seconds) in the *age* field. All routers regularly decrement the *age* of the LSPs in their LSDB and a LSP is discarded once its *age* reaches *0*. Thanks to the *age* field, the LSP from a failed router does not remain in the LSDBs forever.

To compute its forwarding table, each router computes the spanning tree rooted at itself by using Dijkstra's shortest path algorithm [Dijkstra1959]. The forwarding table can be derived automatically from the spanning as shown in the

[2] It should be noted that link state routing assumes that all routers in the network have enough memory to store the entire LSDB. The routers that do not have enough memory to store the entire LSDB cannot participate in link state routing. Some link state routing protocols allow routers to report that they do not have enough memory and must be removed from the graph by the other routers in the network, but this is outside the scope of this e-book.



Links	LSPs
A → B, B → A: 1	A-0 [B:1];[D:1]
A → D, D → A: 1	B-0 [A:1];[C:1];[E:1]
B → C, C → B: 1	C-0 [B:1];[E:1]
B → E, E → B: 1	D-0 [A:1];[E:1]
C → E, E → C: 1	E-0 [B:1];[C:1];[D:1]
D → E, E → D: 1	

Links	LSPs
A → B, B → A: 1	A-0 [B:1];[D:1]
A → D, D → A: 1	B-0 [A:1];[C:1];[E:1]
B → C, C → B: 1	C-0 [B:1];[E:1]
B → E, E → B: 1	D-0 [A:1];[E:1]
C → E, E → C: 1	E-0 [B:1];[C:1];[D:1]
D → E, E → D: 1	

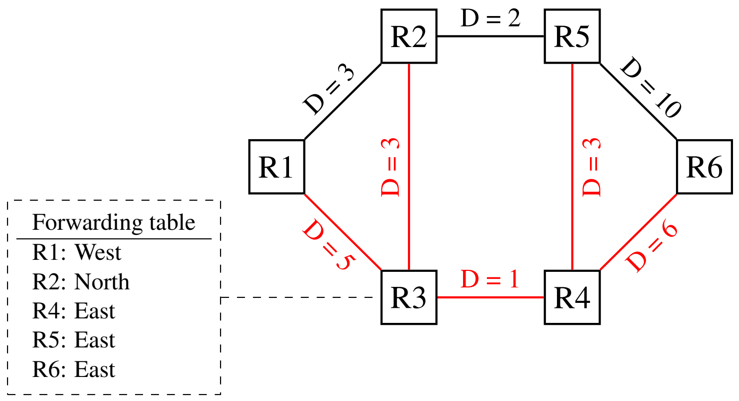
Links	LSPs
A → B, B → A: 1	A-0 [B:1];[D:1]
A → D, D → A: 1	B-0 [A:1];[C:1];[E:1]
B → C, C → B: 1	C-0 [B:1];[E:1]
B → E, E → B: 1	D-0 [A:1];[E:1]
C → E, E → C: 1	E-0 [B:1];[C:1];[D:1]
D → E, E → D: 1	

Links	LSPs
A → B, B → A: 1	A-0 [B:1];[D:1]
A → D, D → A: 1	B-0 [A:1];[C:1];[E:1]
B → C, C → B: 1	C-0 [B:1];[E:1]
B → E, E → B: 1	D-0 [A:1];[E:1]
C → E, E → C: 1	E-0 [B:1];[C:1];[D:1]
D → E, E → D: 1	

Links	LSPs
A → B, B → A: 1	A-0 [B:1];[D:1]
A → D, D → A: 1	B-0 [A:1];[C:1];[E:1]
B → C, C → B: 1	C-0 [B:1];[E:1]
B → E: 1, ~~E → B: 1~~	D-0 [A:1];[E:1]
C → E, E → C: 1	E-1 [C:1];[D:1]
D → E, E → D: 1	

Fig. 34: The two-way connectivity check

figure below.



Forwarding table
R1: West
R2: North
R4: East
R5: East
R6: East

Fig. 35: Computation of the forwarding table, the paths used by packets sent by R3 are shown in red

2.3 Applications

There are two important models used to organize a networked application. The first and oldest model is the client-server model. In this model, a server provides services to clients that exchange information with it. This model is highly asymmetrical: clients send requests and servers perform actions and return responses. It is illustrated in the figure below.

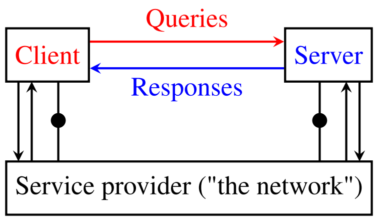


Fig. 36: The client-server model

The client-server model was the first model to be used to develop networked applications. This model comes naturally from the mainframes and minicomputers that were the only networked computers used until the 1980s. A minicomputer is a multi-user system that is used by tens or more users at the same time. Each user interacts with the minicomputer by using a terminal. Such a terminal was mainly a screen, a keyboard and a cable directly connected to the minicomputer.

There are various types of servers as well as various types of clients. A web server provides information in response to the query sent by its clients. A print server prints documents sent as queries by the client. An email server forwards towards their recipient the email messages sent as queries while a music server delivers the music requested by the client. From the viewpoint of the application developer, the client and the server applications directly exchange messages (the horizontal arrows labeled *Queries* and *Responses* in the above figure), but in practice these messages are exchanged thanks to the underlying layers (the vertical arrows in the above figure). In this chapter, we focus on these horizontal exchanges of messages.

Networked applications do not exchange random messages. In order to ensure that the server is able to understand the queries sent by a client, and also that the client is able to understand the responses sent by the server, they must both agree on a set of syntactical and semantic rules. These rules define the format of the messages exchanged as well as their ordering. This set of rules is called an application-level *protocol*.

An *application-level protocol* is similar to a structured conversation between humans. Assume that Alice wants to know the current time but does not have a watch. If Bob passes close by, the following conversation could take place :

- Alice : *Hello*
- Bob : *Hello*
- Alice : *What time is it ?*
- Bob : *11:55*
- Alice : *Thank you*
- Bob : *You're welcome*

Such a conversation succeeds if both Alice and Bob speak the same language. If Alice meets Tchang who only speaks Chinese, she won't be able to ask him the current time. A conversation between humans can be more complex. For example, assume that Bob is a security guard whose duty is to only allow trusted secret agents to enter a meeting room. If all agents know a secret password, the conversation between Bob and Trudy could be as follows :

- Bob : *What is the secret password ?*

- Trudy : *1234*
- Bob : *This is the correct password, you're welcome*

If Alice wants to enter the meeting room but does not know the password, her conversation could be as follows :

- Bob : *What is the secret password ?*
- Alice : *3.1415*
- Bob : *This is not the correct password.*

Human conversations can be very formal, e.g. when soldiers communicate with their hierarchy, or informal such as when friends discuss. Computers that communicate are more akin to soldiers and require well-defined rules to ensure a successful exchange of information. There are two types of rules that define how information can be exchanged between computers :

- Syntactical rules that precisely define the format of the messages that are exchanged. As computers only process bits, the syntactical rules specify how information is encoded as bit strings.
- Organization of the information flow. For many applications, the flow of information must be structured and there are precedence relationships between the different types of information. In the time example above, Alice must greet Bob before asking for the current time. Alice would not ask for the current time first and greet Bob afterwards. Such precedence relationships exist in networked applications as well. For example, a server must receive a username and a valid password before accepting more complex commands from its clients.

Let us first discuss the syntactical rules. We will later explain how the information flow can be organized by analyzing real networked applications.

Application-layer protocols exchange two types of messages. Some protocols such as those used to support electronic mail exchange messages expressed as strings or lines of characters. As the transport layer allows hosts to exchange bytes, they need to agree on a common representation of the characters. The first and simplest method to encode characters is to use the *ASCII* table. **RFC 20** provides the ASCII table that is used by many protocols on the Internet. For example, the table defines the following binary representations :

- *A* : *1000011b*
- *0* : *0110000b*
- *z* : *1111010b*
- *@* : *1000000b*
- *space* : *0100000b*

In addition, the *ASCII* table also defines several non-printable or control characters. These characters were designed to allow an application to control a printer or a terminal. These control characters include *CR* and *LF*, that are used to terminate a line, and the *Bell* character which causes the terminal to emit a sound.

- *carriage return* (*CR*) : *0001101b*
- *line feed* (*LF*) : *0001010b*
- *Bell*: *0000111b*

The *ASCII* characters are encoded as a seven bits field, but transmitted as an eight-bits byte whose high order bit is usually set to *0*. Bytes are always transmitted starting from the high order or most significant bit.

Most applications exchange strings that are composed of fixed or variable numbers of characters. A common solution to define the character strings that are acceptable is to define them as a grammar using a Backus-Naur Form (*BNF*) such as the Augmented BNF defined in **RFC 5234**. A BNF is a set of production rules that generate all valid character strings. For example, consider a networked application that uses two commands, where the user can supply a username and a password. The BNF for this application could be defined as shown in the figure below.

```
command         = usercommand / passwordcommand
usercommand     = "user" SP username CRLF
passwordcommand = "pass" SP password CRLF
username        = 1*8ALPHA
password        = (ALPHA) *(ALPHA/DIGIT)
ALPHA           = %x41-5A / %x61-7A
CR              = %x0D
CRLF            = CR LF
DIGIT           = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"
LF              = %x0A
SP              = %x20 / %x09
```

Fig. 37: A simple BNF specification

The example above defines several terminals and two commands : *usercommand* and *passwordcommand*. The *ALPHA* terminal contains all letters in upper and lower case. In the *ALPHA* rule, *%x41* corresponds to ASCII character code 41 in hexadecimal, i.e. capital *A*. The *CR* and *LF* terminals correspond to the carriage return and linefeed control characters. The *CRLF* rule concatenates these two terminals to match the standard end of line termination. The *DIGIT* terminal contains all digits. The *SP* terminal corresponds to the white space characters. The *usercommand* is composed of two strings separated by white space. In the ABNF rules that define the messages used by Internet applications, the commands are case-insensitive. The rule *"user"* corresponds to all possible cases of the letters that compose the word between brackets, e.g. *user*, *uSeR*, *USER*, *usER*, ... A *username* contains at least one letter and up to 8 letters. User names are case-sensitive as they are not defined as a string between brackets. The *password* rule indicates that a password starts with a letter and can contain any number of letters or digits. The white space and the control characters cannot appear in a *password* defined by the above rule.

Besides character strings, some applications also need to exchange 16 bits and 32 bits fields such as integers. A naive solution would have been to send the 16- or 32-bits field as it is encoded in the host's memory. Unfortunately, there are different methods to store 16- or 32-bits fields in memory. Some CPUs store the most significant byte of a 16-bits field in the first address of the field while others store the least significant byte at this location. When networked applications running on different CPUs exchange 16 bits fields, there are two possibilities to transfer them over the transport service :

- send the most significant byte followed by the least significant byte
- send the least significant byte followed by the most significant byte

The first possibility was named *big-endian* in a note written by Cohen [Cohen1980] while the second was named *little-endian*. Vendors of CPUs that used *big-endian* in memory insisted on using *big-endian* encoding in networked applications while vendors of CPUs that used *little-endian* recommended the opposite. Several studies were written on the relative merits of each type of encoding, but the discussion became almost a religious issue [Cohen1980]. Eventually, the Internet chose the *big-endian* encoding, i.e. multi-byte fields are always transmitted by sending the most significant byte first, **RFC 791** refers to this encoding as the *network-byte order*. Most libraries[1] used to write networked applications contain functions to convert multi-byte fields from memory to the network byte order and the reverse.

Besides 16 and 32 bit words, some applications need to exchange data structures containing bit fields of various lengths. For example, a message may be composed of a 16 bits field followed by eight, one bit flags, a 24 bits field and two 8 bits bytes. Internet protocol specifications will define such a message by using a representation such as the one below. In this representation, each line corresponds to 32 bits and the vertical lines are used to delineate fields. The numbers above the lines indicate the bit positions in the 32-bits word, with the high order bit at position *0*.

The message mentioned above will be transmitted starting from the upper 32-bits word in network byte order. The first field is encoded in 16 bits. It is followed by eight one bit flags (*A-H*), a 24 bits field whose high order byte is shown in the first line and the two low order bytes appear in the second line followed by two one byte fields. This ASCII representation is frequently used when defining binary protocols. We will use it for all the binary protocols that are discussed in this book.

[1] For example, the `htonl(3)` (resp. `ntohl(3)`) function the standard C library converts a 32-bits unsigned integer from the byte order used by the CPU to the network byte order (resp. from the network byte order to the CPU byte order). Similar functions exist in other programming languages.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     First field (16 bits)     |A|B|C|D|E|F|G|H|    Second     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        field (24 bits)        |  First Byte   |  Second Byte  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 38: Message format

The peer-to-peer model emerged during the last ten years as another possible architecture for networked applications. In the traditional client-server model, hosts act either as servers or as clients and a server serves a large number of clients. In the peer-to-peer model, all hosts act as both servers and clients and they play both roles. The peer-to-peer model has been used to develop various networked applications, ranging from Internet telephony to file sharing or Internet-wide filesystems. A detailed description of peer-to-peer applications may be found in [BYL2008]. Surveys of peer-to-peer protocols and applications may be found in [AS2004] and [LCP2005].

2.4 The transport layer

A network is always designed and built to enable applications running on hosts to exchange information. In a previous chapter, we have explained the principles of the *network layer* that enables hosts connected to different types of datalink layers to exchange information through routers. These routers act as relays in the network layer and ensure the delivery of packets between any pair of hosts attached to the network.

The network layer ensures the delivery of packets on a hop-by-hop basis through intermediate nodes. As such, it provides a service to the upper layer. In practice, this layer is usually the *transport layer* that improves the service provided by the *network layer* to make it usable by applications.

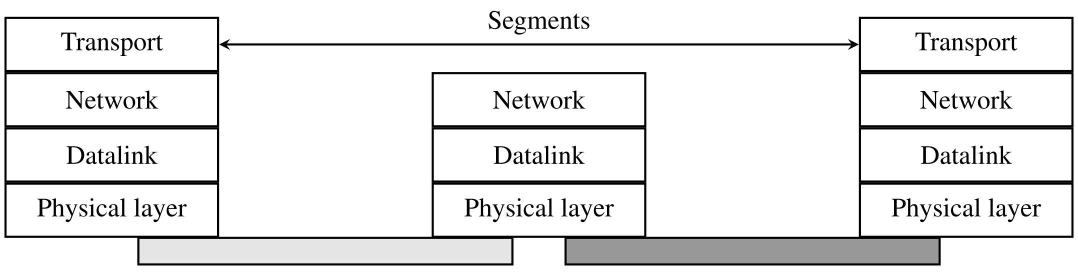
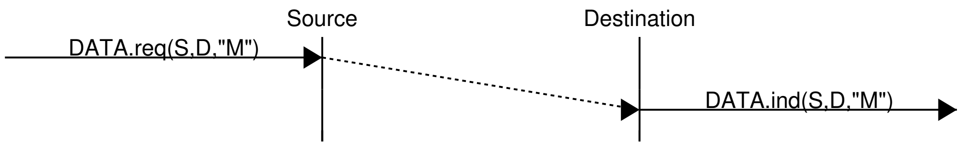


Fig. 39: The transport layer

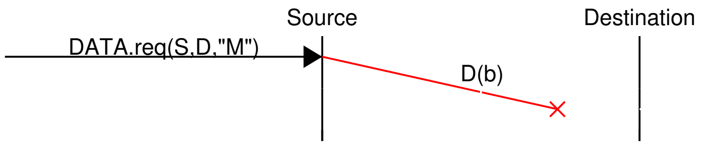
Most networks use a datagram organization and provide a simple service which is called the *connectionless service*.

The figure below provides a representation of the connectionless service as a *time-sequence diagram*. The user on the left, having address *S*, issues a *Data.request* primitive containing Service Data Unit (SDU) *M* that must be delivered by the service provider to destination *D*. The dashed line between the two primitives indicates that the *Data.indication* primitive that is delivered to the user on the right corresponds to the *Data.request* primitive sent by the user on the left.



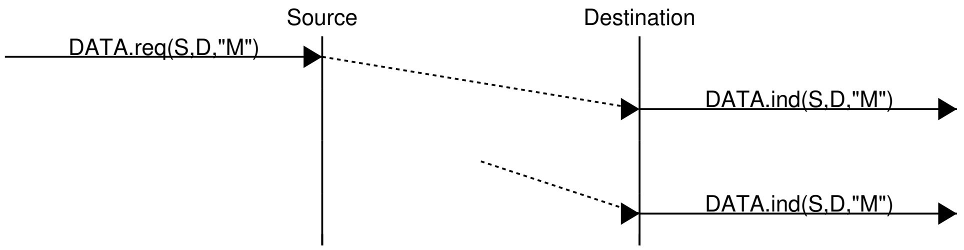
There are several possible implementations of the connectionless service. Before studying these realizations, it is useful to discuss the possible characteristics of the connectionless service. A *reliable connectionless service* is a service where the service provider guarantees that all SDUs submitted in *Data.requests* by a user will eventually be delivered to their destination. Such a service would be very useful for users, but guaranteeing perfect delivery is difficult in practice. For this reason, network layers usually support an *unreliable connectionless service*.

An *unreliable connectionless* service may suffer from various types of problems compared to a *reliable connectionless service*. First of all, an *unreliable connectionless service* does not guarantee the delivery of all SDUs. This can be expressed graphically by using the time-sequence diagram below.

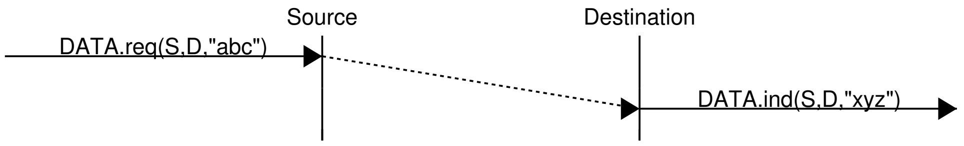


In practice, an *unreliable connectionless service* will usually deliver a large fraction of the SDUs. However, since the delivery of SDUs is not guaranteed, the user must be able to recover from the loss of any SDU.

A second imperfection that may affect an *unreliable connectionless service* is that it may duplicate SDUs. Some packets may be duplicated in a network and be delivered twice to their destination. This is illustrated by the time-sequence diagram below.



Finally, some unreliable connectionless service providers may deliver to a destination a different SDU than the one that was supplied in the *Data.request*. This is illustrated in the figure below.



As the transport layer is built on top of the network layer, it is important to know the key features of the network layer service. In this book, we only consider the *connectionless network layer service* which is the most widespread. Its main characteristics are :

- the *connectionless network layer service* can only transfer SDUs of *limited size*
- the *connectionless network layer service* may discard SDUs
- the *connectionless network layer service* may corrupt SDUs
- the *connectionless network layer service* may delay, reorder or even duplicate SDUs

These imperfections of the *connectionless network layer service* are caused by the operations of the *network layer*. This *layer* is able to deliver packets to their intended destination, but it cannot guarantee their delivery. The main cause of packet losses and errors are the buffers used on the network nodes. If the buffers of one of these nodes becomes full, all arriving packets must be discarded. This situation frequently happens in practice[2]. Transmission errors can also affect packet transmissions on links where reliable transmission techniques are not enabled or because of errors in the buffers of the network nodes.

[2] In the application layer, most servers are implemented as processes. The network and transport layer on the other hand are usually implemented inside the operating system and the amount of memory that they can use is limited by the amount of memory allocated to the entire kernel.

2.4.1 Transport layer services

When two applications need to communicate, they need to structure their exchange of information. Structuring this exchange of information requires solving two different problems. The first problem is how to represent the information being exchanged knowing that the two applications may be running on hosts that use different operating systems, different processors and have different conventions to store information. This requires a common syntax to transfer the information between the two applications. For this chapter, let us assume that this syntax exists and that the two applications simply need to exchange bytes. We will discuss later how more complex data can be encoded as sequences of bytes to be exchanged. The second problem is how to organize the interactions between the application and the underlying network. From the application's viewpoint, the *network* will appear as the *transport layer* service. This *transport layer* can provide three types of services to the applications :

- the *connectionless service*
- the *connection oriented service*
- the *request-response service*

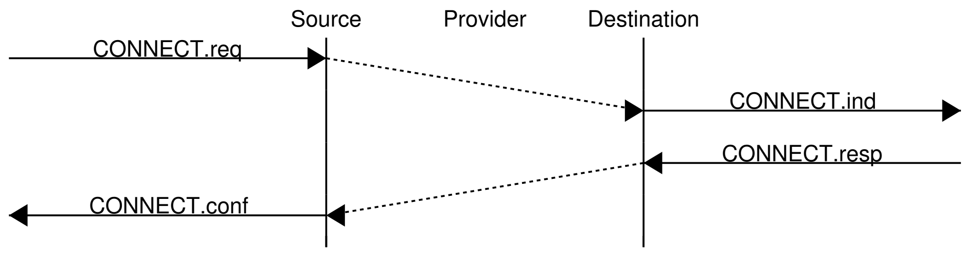
The connectionless service

The *connectionless service* that we have described earlier is frequently used by users who need to exchange small SDUs. It can be easily built on top of the connectionless network layer service that we have described earlier. Users needing to either send or receive several different and potentially large SDUs, or who need structured exchanges often prefer the *connection-oriented service*.

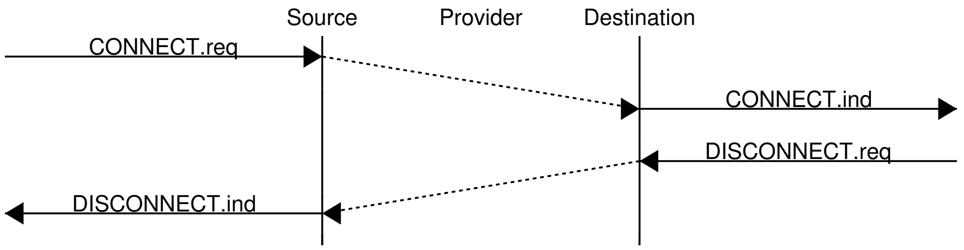
The connection-oriented service

An invocation of the *connection-oriented service* is divided into three phases. The first phase is the establishment of a *connection*. A *connection* is a temporary association between two users through a service provider. Several connections may exist at the same time between any pair of users. Once established, the connection is used to transfer SDUs. *Connections* usually provide one bidirectional stream supporting the exchange of SDUs between the two users that are associated through the *connection*. This stream is used to transfer data during the second phase of the connection called the *data transfer* phase. The third phase is the termination of the connection. Once the users have finished exchanging SDUs, they request the service provider to terminate the connection. As we will see later, there are also some cases where the service provider may need to terminate a connection itself.

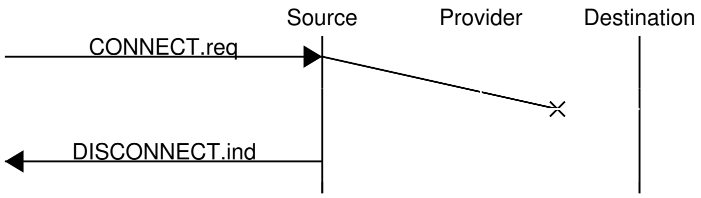
The establishment of a connection can be modeled by using four primitives : *Connect.request*, *Connect.indication*, *Connect.response* and *Connect.confirm*. The *Connect.request* primitive is used to request the establishment of a connection. The main parameter of this primitive is the *address* of the destination user. The service provider delivers a *Connect.indication* primitive to inform the destination user of the connection attempt. If it accepts to establish a connection, it responds with a *Connect.response* primitive. At this point, the connection is considered to be established and the destination user can start sending SDUs over the connection. The service provider processes the *Connect.response* and will deliver a *Connect.confirm* to the user who initiated the connection. The delivery of this primitive terminates the connection establishment phase. At this point, the connection is considered to be open and both users can send SDUs. A successful connection establishment is illustrated below.



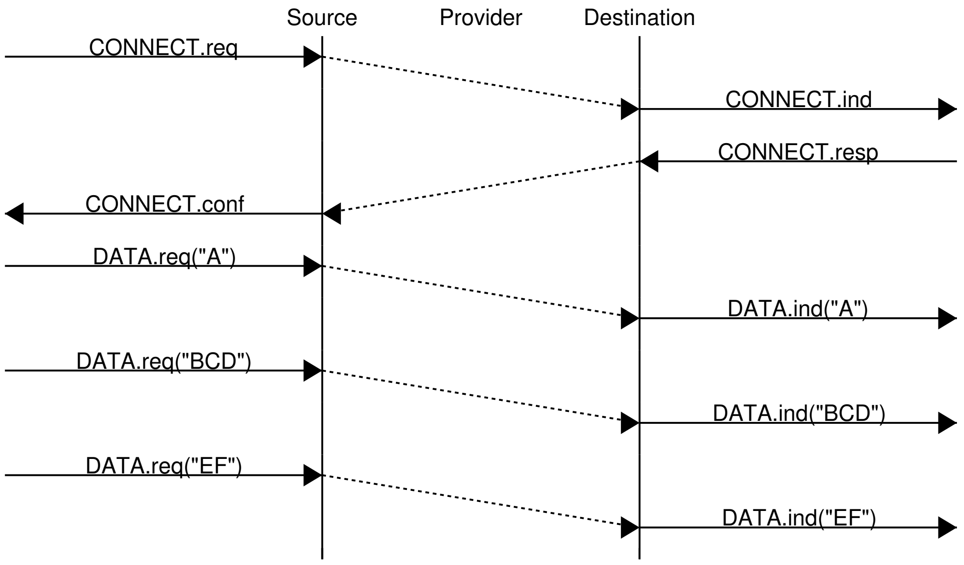
The example above shows a successful connection establishment. However, in practice not all connections are successfully established. One reason is that the destination user may not agree, for policy or performance reasons, to establish a connection with the initiating user at this time. In this case, the destination user responds to the *Connect.indication* primitive by a *Disconnect.request* primitive that contains a parameter to indicate why the connection has been refused. The service provider will then deliver a *Disconnect.indication* primitive to inform the initiating user.



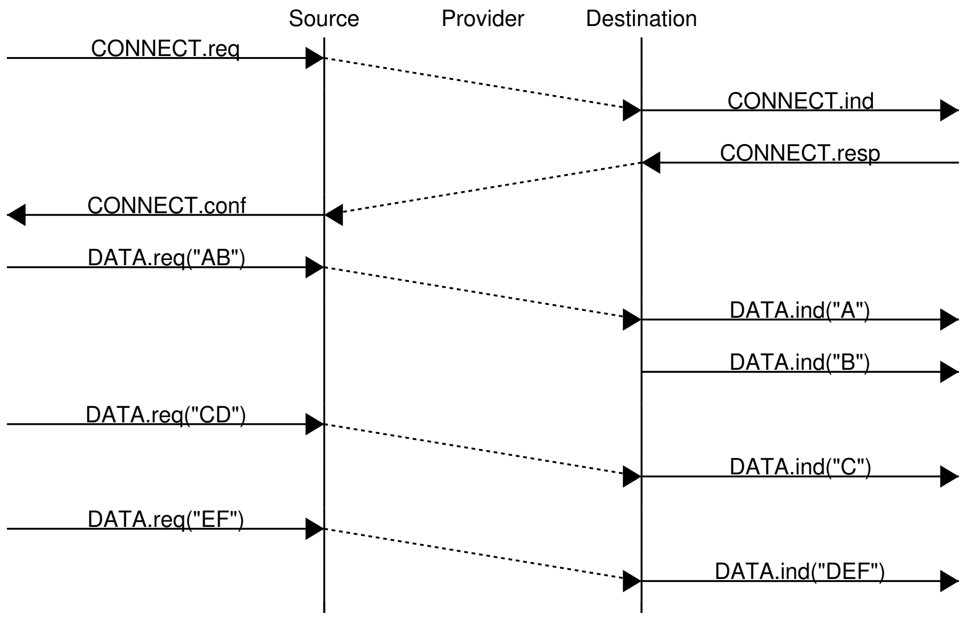
A second reason is when the service provider is unable to reach the destination user. This might happen because the destination user is not currently attached to the network or due to congestion. In these cases, the service provider responds to the *Connect.request* with a *Disconnect.indication* primitive whose *reason* parameter contains additional information about the failure of the connection.



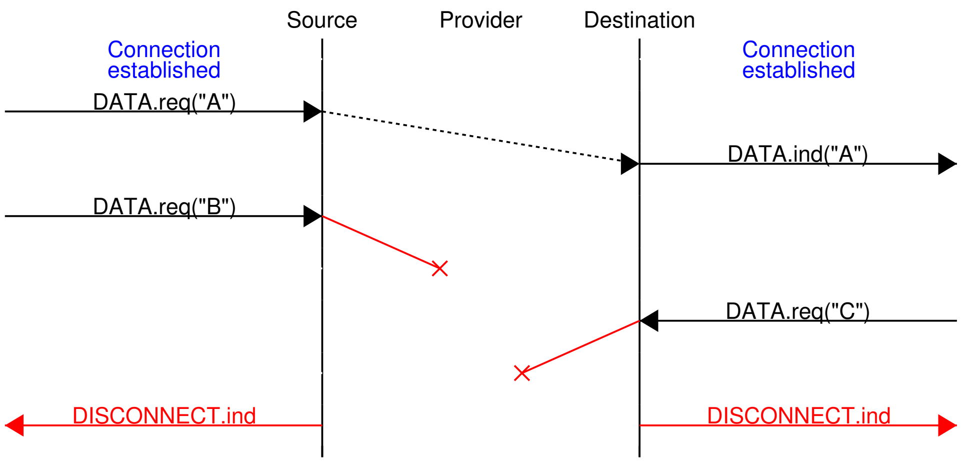
Once the connection has been established, the service provider supplies two data streams to the communicating users. The first data stream can be used by the initiating user to send SDUs. The second data stream allows the responding user to send SDUs to the initiating user. The data streams can be organized in different ways. A first organization is the *message-mode* transfer. With the *message-mode* transfer, the service provider guarantees that one and only one *Data.indication* will be delivered to the endpoint of the data stream for each *Data.request* primitive issued by the other endpoint. The *message-mode* transfer is illustrated in the figure below. The main advantage of the *message-transfer* mode is that the recipient receives exactly the SDUs that were sent by the other user. If each SDU contains a command, the receiving user can process each command as soon as it receives a SDU.



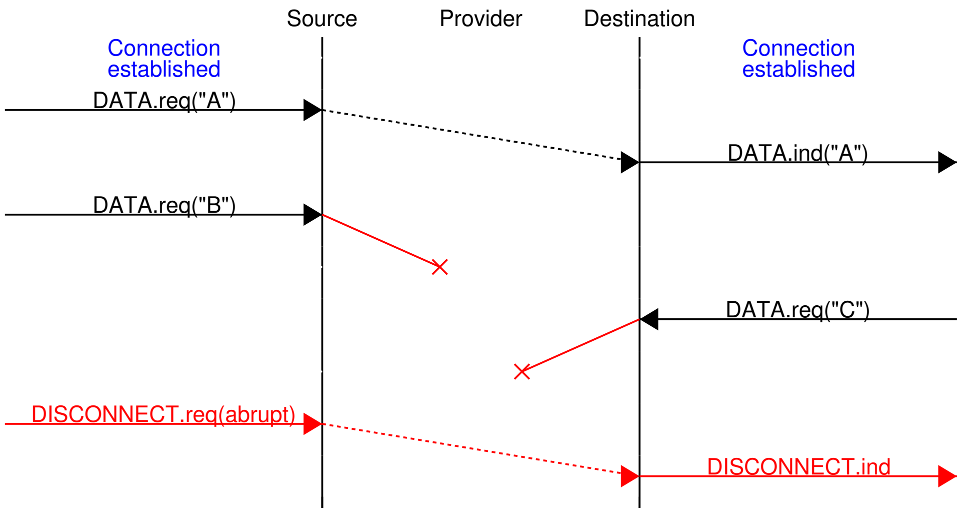
Unfortunately, the *message-mode* transfer is not widely used on the Internet. On the Internet, the most popular connection-oriented service transfers SDUs in *stream-mode*. With the *stream-mode*, the service provider supplies a byte stream that links the two communicating users. The sending user sends bytes by using *Data.request* primitives that contain sequences of bytes as SDUs. The service provider delivers SDUs containing consecutive bytes to the receiving user by using *Data.indication* primitives. The service provider ensures that all the bytes sent at one end of the stream are delivered correctly in the same order at the other endpoint. However, the service provider does not attempt to preserve the boundaries of the SDUs. There is no relation enforced by the service provider between the number of *Data.request* and the number of *Data.indication* primitives. The *stream-mode* is illustrated in the figure below. In practice, a consequence of the utilization of the *stream-mode* is that if the users want to exchange structured SDUs, they will need to provide the mechanisms that allow the receiving user to separate successive SDUs in the byte stream that it receives. Application layer protocols often use specific delimiters such as the end of line character to delineate SDUs in a bytestream.



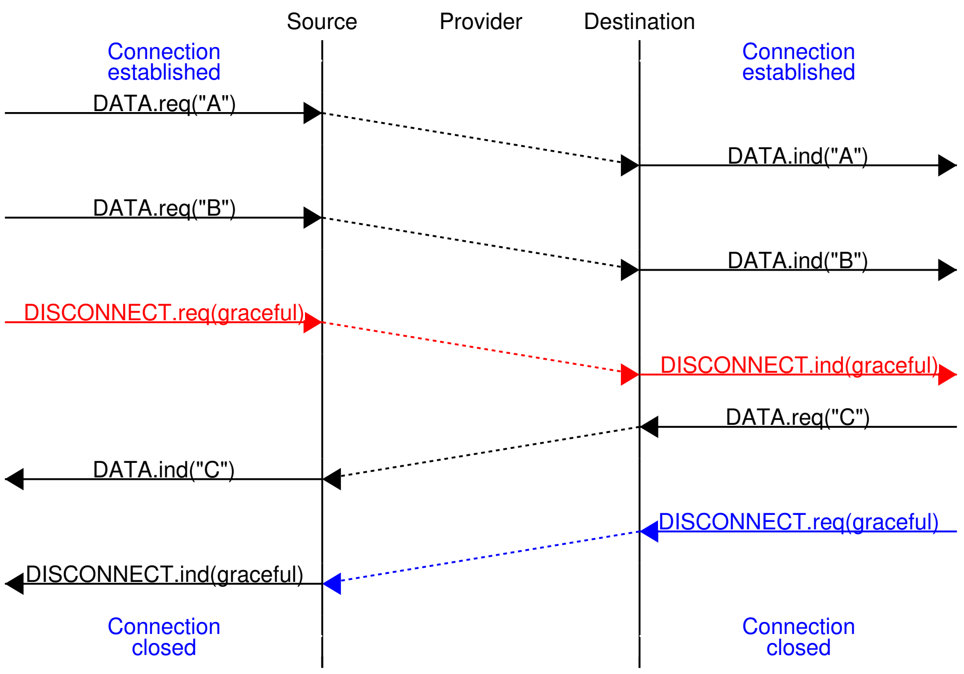
The third phase of a connection is its release. As a connection involves three parties (two users and one service provider), any of them can request the termination of the connection. Usually, connections are terminated upon request of one user once the data transfer is finished. However, sometimes the service provider may be forced to terminate a connection. This can be due to lack of resources inside the service provider or because one of the users is not reachable anymore through the network. In this case, the service provider will issue *Disconnect.indication* primitives to both users. These primitives will contain, as parameter, some information about the reason for the termination of the connection. Unfortunately, as illustrated in the figure below, when a service provider is forced to terminate a connection it cannot guarantee that all SDUs sent by each user have been delivered to the other user. This connection release is said to be abrupt as it can cause losses of data.



An abrupt connection release can also be triggered by one of the users. If a user needs, for any reason, to terminate a connection quickly, it can issue a *Disconnect.request* primitive and to request an abrupt release. The service provider will process the request, stop the two data streams and deliver the *Disconnect.indication* primitive to the remote user as soon as possible. As illustrated in the figure below, this abrupt connection release may cause losses of SDUs.



To ensure a reliable delivery of the SDUs sent by each user over a connection, we need to consider the two streams that compose a connection as independent. A user should be able to release the stream that it uses to send SDUs once it has sent all the SDUs that it planned to send over this connection, but still continue to receive SDUs over the opposite stream. This *graceful* connection release is usually performed as shown in the figure below. One user issues a *Disconnect.request* primitive to its provider once it has issued all its *Data.request* primitives. The service provider will wait until all *Data.indication* primitives have been delivered to the receiving user before issuing the *Disconnnect.indication* primitive. This primitive informs the receiving user that it will no longer receive SDUs over this connection, but it is still able to issue *Data.request* primitives on the stream in the opposite direction. Once the user has issued all of its *Data.request* primitives, it issues a *Disconnnect.request* primitive to request the termination of the remaining stream. The service provider will process the request and deliver the corresponding *Disconnect.indication* to the other user once it has delivered all the pending *Data.indication* primitives. At this point, all data has been delivered, the two streams have been released successfully and the connection is completely closed.



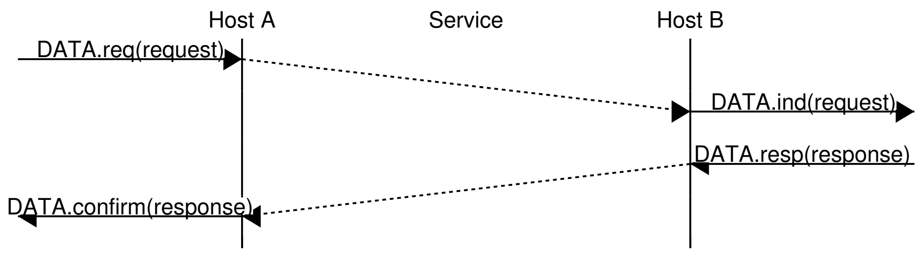
Note: Reliability of the connection-oriented service

An important point to note about the connection-oriented service is its reliability. A *connection-oriented* service can only guarantee the correct delivery of all SDUs provided that the connection has been released gracefully. This implies that while the connection is active, there is no guarantee for the actual delivery of the SDUs exchanged as the connection may need to be abruptly released at any time.

The request-response service

The *request-response service* is a compromise between the *connectionless service* and the *connection-oriented service*. Many applications need to send a small amount of data and receive a small amount of information back. This is similar to procedure calls in programming languages. A call to a procedure takes a few arguments and returns a simple answer. In a network, it is sometimes useful to execute a procedure on a different host and receive the result of the computation. Executing a procedure on another host is often called Remote Procedure Call. It is possible to use the *connectionless service* for this application. However, since this service is usually unreliable, this would force the application to deal with any type of error that could occur. Using the *connection oriented service* is another alternative. This service ensures the reliable delivery of the data, but a connection must be created before the beginning of the data transfer. This overhead can be important for applications that only exchange a small amount of data.

The *request-response service* allows to efficiently exchange small amounts of information in a request and associate it with the corresponding response. This service can be depicted by using the time-sequence diagram below.



Note: Services and layers

In the previous sections, we have described services that are provided by the transport layer. However, it is important to note that the notion of service is more general than in the transport layer. As explained earlier, the network layer also provides a service, which in most networks is an unreliable connectionless service. There are network layers that provide a connection-oriented service. Similarly, the datalink layer also provides services. Some datalink layers will provide a connectionless service. This will be the case in Local Area Networks for examples. Other datalink layers, e.g. in public networks, provide a connection oriented service.

2.4.2 The transport layer

The transport layer entity interacts with both a user in the application layer and the network layer. It improves the network layer service to make it usable by applications. From the application's viewpoint, the main limitations of the network layer service come from its unreliable service:

- the network layer may corrupt data;
- the network layer may loose data;
- the network layer may not deliver data in-order;
- the network layer has an upper bound on maximum length of the data;
- the network layer may duplicate data.

To deal with these issues, the transport layer includes several mechanisms that depend on the service that it provides. It interacts with both the applications and the underlying network layer.

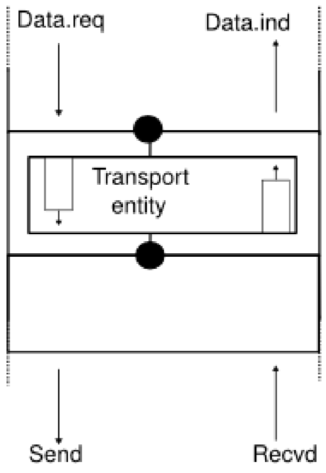


Fig. 40: Interactions between the transport layer, its user, and its network layer provider

We have already described in the datalink layers mechanisms to deal with data losses and transmission errors. These techniques are also used in the transport layer.

Connectionless transport

The simplest service that can be provided in the transport layer is the connectionless transport service. Compared to the connectionless network layer service, this transport service includes two additional features :

- an *error detection* mechanism that allows detecting corrupted data
- a *multiplexing technique* that enables several applications running on one host to exchange information with another host

To exchange data, the transport protocol encapsulates the SDU produced by its user inside a *segment*. The *segment* is the unit of transfer of information in the transport layer. Transport layer entities always exchange segments. When a transport layer entity creates a segment, this segment is encapsulated by the network layer into a packet which contains the segment as its payload and a network header. The packet is then encapsulated in a frame to be transmitted in the datalink layer.

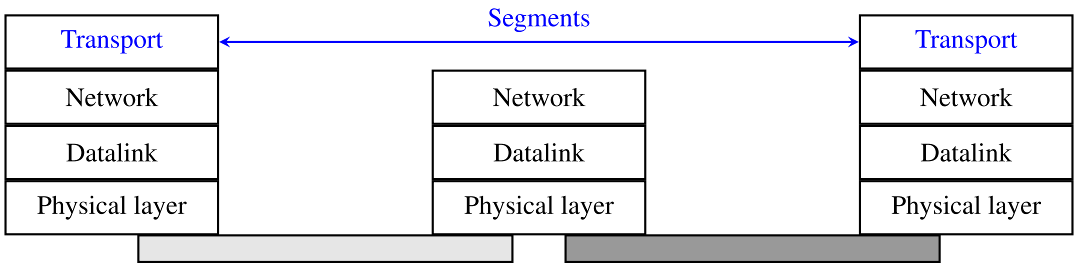


Fig. 41: Segments are the unit of transfer at transport layer

A *segment* also contains control information, usually stored inside a *header* and the payload that comes from the application. To detect transmission errors, transport protocols rely on checksums or CRCs like the datalink layer protocols.

Compared to the connectionless network layer service, the transport layer service allows several applications running on a host to exchange SDUs with several other applications running on remote hosts. Let us consider two hosts, e.g. a client and a server. The network layer service allows the client to send information to the server, but if an application running on the client wants to contact a particular application running on the server, then an additional addressing mechanism is required. The network layer address identifies a host, but it is not sufficient to differentiate the applications running on a host. *Port numbers* provides this additional addressing. When a server application is launched on a host, it registers a *port number*. This *port number* will be used by the clients to contact the server process.

The figure below shows a typical usage of port numbers. The client process uses port number *1234* while the server process uses port number *5678*. When the client sends a request, it is identified as originating from port number *1234* on the client host and destined to port number *5678* on the server host. When the server process replies to this request, the server's transport layer returns the reply as originating from port *5678* on the server host and destined to port *1234* on the client host.

To support the connection-oriented service, the transport layer needs to include several mechanisms to enrich the connectionless network-layer service. We discuss these mechanisms in the following sections.

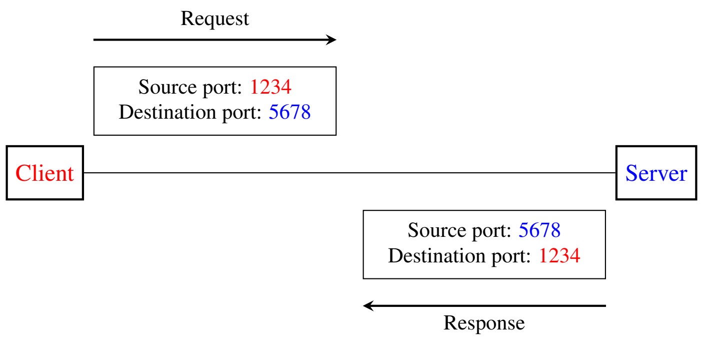


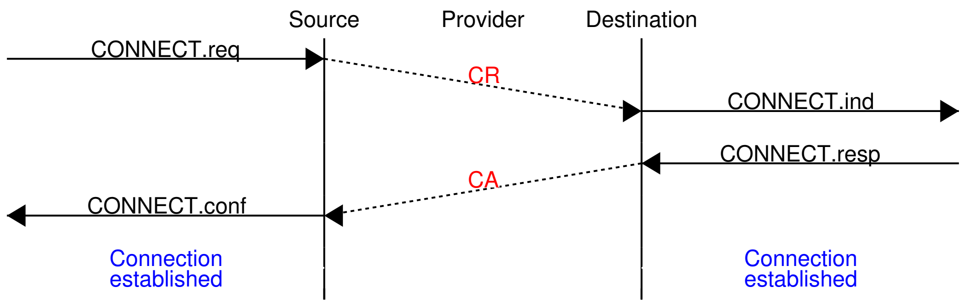
Fig. 42: Utilization of port numbers

Connection establishment

Like the connectionless service, the connection-oriented service allows several applications running on a given host to exchange data with other hosts. The port numbers described above for the connectionless service are also used by the connection-oriented service to multiplex several applications. Similarly, connection-oriented protocols use checksums/CRCs to detect transmission errors and discard segments containing an invalid checksum/CRC.

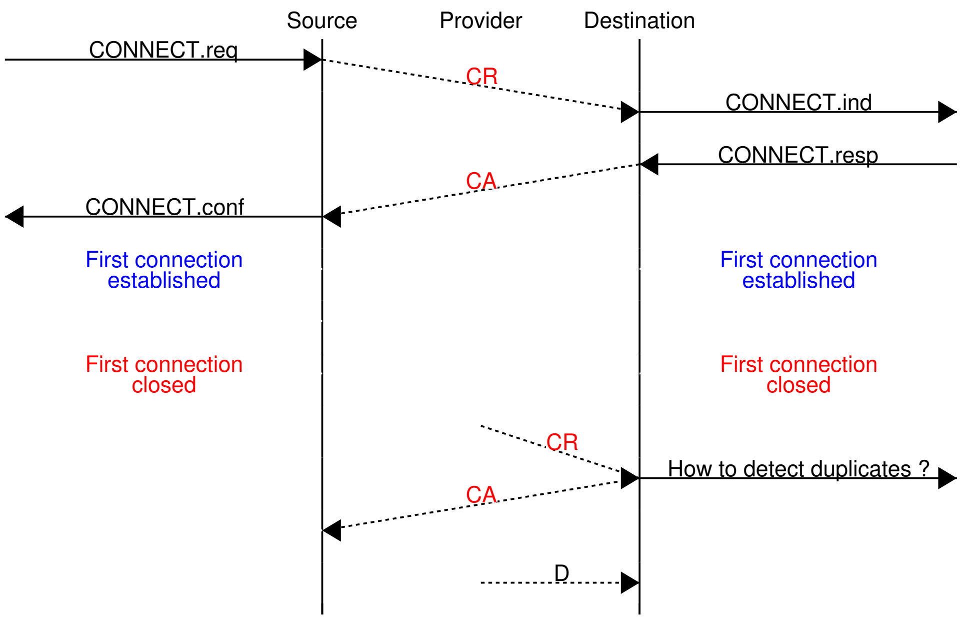
An important difference between the connectionless service and the connection-oriented one is that the transport entities in the latter maintain some state during lifetime of the connection. This state is created when a connection is established and is removed when it is released.

The simplest approach to establish a transport connection would be to define two special control segments : *CR* (Connection Request) and *CA* (Connection Acknowledgment). The *CR* segment is sent by the transport entity that wishes to initiate a connection. If the remote entity wishes to accept the connection, it replies by sending a *CA* segment. The *CR* and *CA* segments contain *port numbers* that allow identifying the communicating applications. The transport connection is considered to be established once the *CA* segment has been received. At that point, data segments can be sent in both directions.



Unfortunately, this scheme is not sufficient given the unreliable network layer. Since the network layer is imperfect, the *CR* or *CA* segments can be lost, delayed, or suffer from transmission errors. To deal with these problems, the control segments must be protected by a CRC or a checksum to detect transmission errors. Furthermore, since the *CA* segment acknowledges the reception of the *CR* segment, the *CR* segment can be protected using a retransmission timer.

Unfortunately, this scheme is not sufficient to ensure the reliability of the transport service. Consider for example a short-lived transport connection where a single, but important transfer (e.g. money transfer from a bank account) is sent. Such a short-lived connection starts with a *CR* segment acknowledged by a *CA* segment, then the data segment is sent, acknowledged and the connection terminates. Unfortunately, as the network layer service is unreliable, delays combined to retransmissions may lead to the situation depicted in the figure below, where a delayed *CR* and data segments from a former connection are accepted by the receiving entity as valid segments, and the corresponding data is delivered to the user. Duplicating SDUs is not acceptable, and the transport protocol must solve this problem.



To avoid these duplicates, transport protocols require the network layer to bound the *Maximum Segment Lifetime (MSL)*. The organization of the network must guarantee that no segment remains in the network for longer than *MSL* seconds. For example, on today's Internet, *MSL* is expected to be 2 minutes. To avoid duplicate transport connections, transport protocol entities must be able to safely distinguish between a duplicate *CR* segment and a new *CR* segment, without forcing each transport entity to remember all the transport connections that it has established in the past.

A classical solution to avoid remembering the previous transport connections to detect duplicates is to use a clock inside each transport entity. This *transport clock* has the following characteristics :

- the *transport clock* is implemented as a *k* bits counter and its clock cycle is such that $2^k \times cycle >> MSL$. Furthermore, the *transport clock* counter is incremented every clock cycle and after each connection establishment. This clock is illustrated in the figure below.
- the *transport clock* must continue to be incremented even if the transport entity stops or reboots

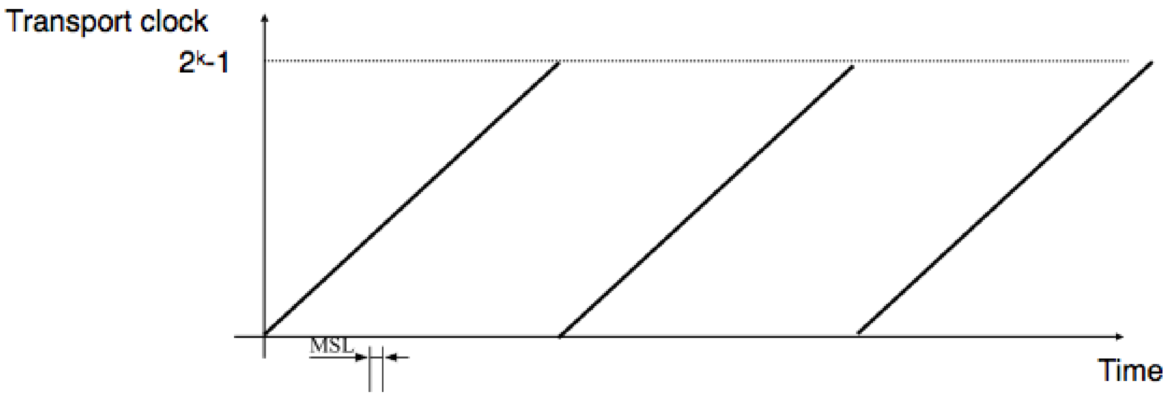


Fig. 43: Transport clock

It should be noted that *transport clocks* do not need and usually are not synchronized to the real-time clock. Precisely synchronizing real-time clocks is an interesting problem, but it is outside the scope of this document. See [Mills2006] for a detailed discussion on synchronizing the real-time clock.

This *transport clock* can now be combined with an exchange of three segments, called the *three way handshake*, to detect duplicates. This *three way handshake* occurs as follows :

1. The initiating transport entity sends a *CR* segment. This segment requests the establishment of a transport connection. It contains a port number (not shown in the figure) and a sequence number (*seq=x* in the figure below) whose value is extracted from the *transport clock*. The transmission of the *CR* segment is protected by a retransmission timer.
2. The remote transport entity processes the *CR* segment and creates state for the connection attempt. At this stage, the remote entity does not yet know whether this is a new connection attempt or a duplicate segment. It returns a *CA* segment that contains an acknowledgment number to confirm the reception of the *CR* segment (*ack=x* in the figure below) and a sequence number (*seq=y* in the figure below) whose value is extracted from its transport clock. At this stage, the connection is not yet established.
3. The initiating entity receives the *CA* segment. The acknowledgment number of this segment confirms that the remote entity has correctly received the *CR* segment. The transport connection is considered to be established by the initiating entity and the numbering of the data segments starts at sequence number *x*. Before sending data segments, the initiating entity must acknowledge the received *CA* segments by sending another *CA* segment.
4. The remote entity considers the transport connection to be established after having received the segment that acknowledges its *CA* segment. The numbering of the data segments sent by the remote entity starts at sequence number *y*.

The three way handshake is illustrated in the figure below.

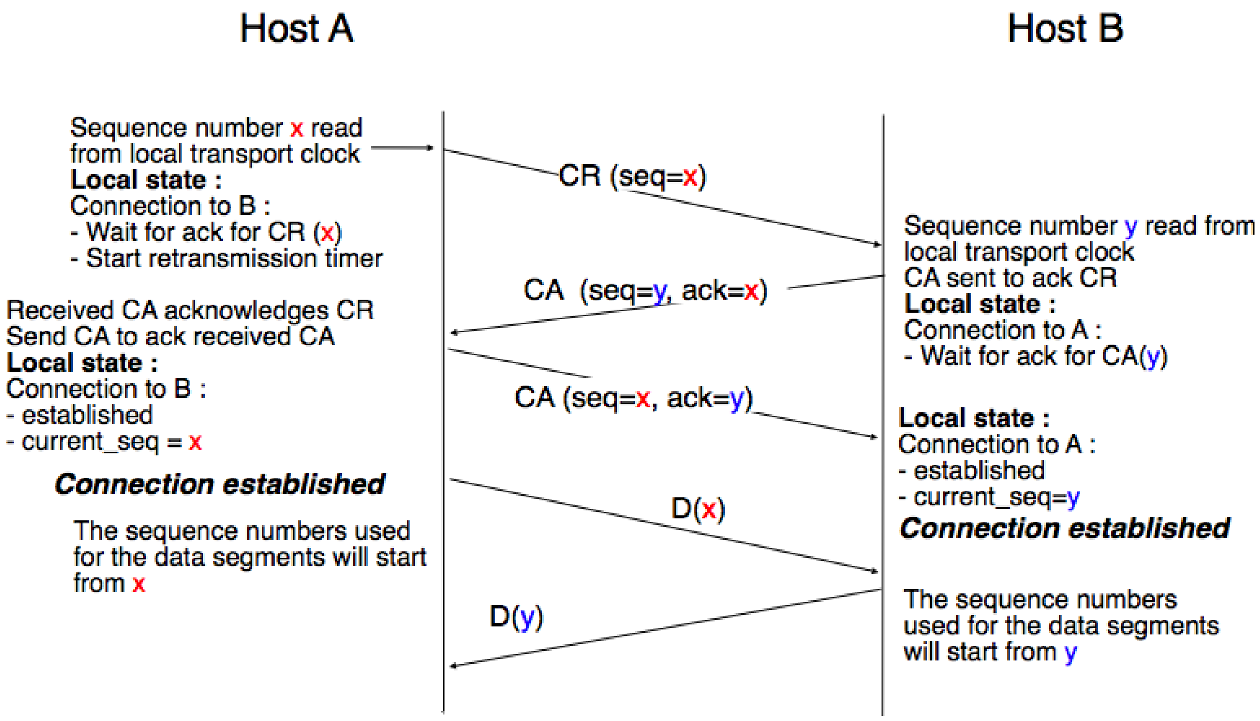

Host A
Host B
Sequence number x read from local transport clock
Local state :
Connection to B :
- Wait for ack for CR (x)
- Start retransmission timer
CR (seq=x)
Sequence number y read from local transport clock
CA sent to ack CR
Local state :
Connection to A :
- Wait for ack for CA(y)
CA (seq=y, ack=x)
Received CA acknowledges CR
Send CA to ack received CA
Local state :
Connection to B :
- established
- current_seq = x
CA (seq=x, ack=y)
Local state :
Connection to A :
- established
- current_seq=y
Connection established
Connection established
D(x)
The sequence numbers used for the data segments will start from x
D(y)
The sequence numbers used for the data segments will start from y

Fig. 44: Three-way handshake

Thanks to the three way handshake, transport entities avoid duplicate transport connections. This is illustrated by considering the three scenarios below.

The first scenario is when the remote entity receives an old *CR* segment. It considers this *CR* segment as a connection establishment attempt and replies by sending a *CA* segment. However, the initiating host cannot match the received *CA* segment with a previous connection attempt. It sends a control segment (*REJECT* in the figure below) to cancel the spurious connection attempt. The remote entity cancels the connection attempt upon reception of this control segment.

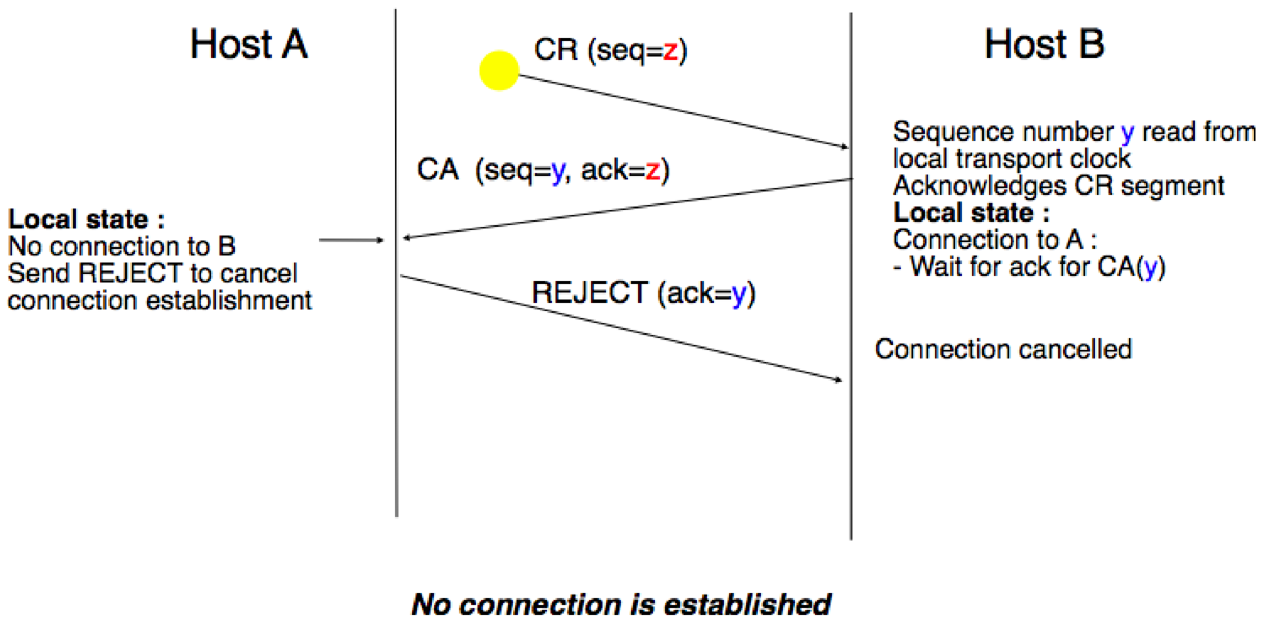


Fig. 45: Three-way handshake : recovery from a duplicate *CR*

A second scenario is when the initiating entity sends a *CR* segment that does not reach the remote entity and receives a duplicate *CA* segment from a previous connection attempt. This duplicate *CA* segment cannot contain a valid acknowledgment for the *CR* segment as the sequence number of the *CR* segment was extracted from the transport clock of the initiating entity. The *CA* segment is thus rejected and the *CR* segment is retransmitted upon expiration of the retransmission timer.

The last scenario is less likely, but it is important to consider it as well. The remote entity receives an old *CR* segment. It notes the connection attempt and acknowledges it by sending a *CA* segment. The initiating entity does not have a matching connection attempt and replies by sending a *REJECT*. Unfortunately, this segment never reaches the remote entity. Instead, the remote entity receives a retransmission of an older *CA* segment that contains the same sequence number as the first *CR* segment. This *CA* segment cannot be accepted by the remote entity as a confirmation of the transport connection as its acknowledgment number cannot have the same value as the sequence number of the first *CA* segment.

Data transfer

Now that the transport connection has been established, it can be used to transfer data. To ensure a reliable delivery of the data, the transport protocol will include sliding windows, retransmission timers and *go-back-n* or *selective repeat*. However, we cannot simply reuse the techniques from the datalink because a transport protocol needs to deal with more types of errors than a reliable protocol in datalink layer. The first difference between the two layers is the transport layer must face with more variable delays. In the datalink layer, when two hosts are connected by a link, the transmission delay or the round-trip-time over the link is almost fixed. In a network that can span the globe, the delays and the round-trip-times can vary significantly on a per packet basis. This variability can be caused by two factors. First, packets sent through a network do not necessarily follow the same path to reach their destination. Second, some packets may be queued in the buffers of routers when the load is high and these queuing delays can lead to increased end-to-end delays. A second difference between the datalink layer and the transport layer is that a network does not always deliver packets in sequence. This implies that packets may be reordered by the network. Furthermore, the network may sometimes duplicate packets. The last issue that needs to be dealt with in the transport layer is the transmission of large SDUs. In the datalink layer, reliable protocols transmit small frames. Applications could generate SDUs that are much larger than the maximum size of a packet in the network layer. The transport layer needs to include mechanisms to fragment and reassemble these large SDUs.

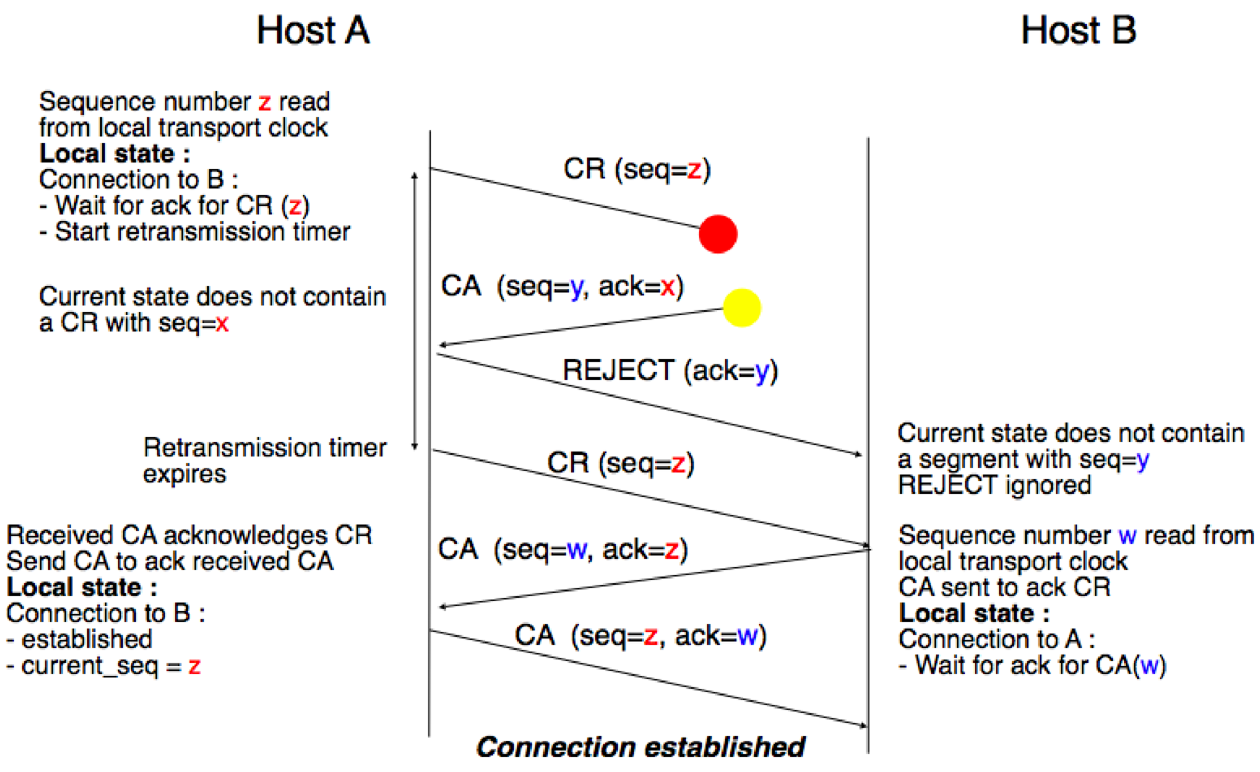

Host A
Host B
Sequence number z read from local transport clock
Local state :
Connection to B :
- Wait for ack for CR (z)
- Start retransmission timer
CR (seq=z)
CA (seq=y, ack=x)
Current state does not contain a CR with seq=x
REJECT (ack=y)
Current state does not contain a segment with seq=y
REJECT ignored
Retransmission timer expires
CR (seq=z)
Sequence number w read from local transport clock
CA sent to ack CR
Local state :
Connection to A :
- Wait for ack for CA(w)
CA (seq=w, ack=z)
Received CA acknowledges CR
Send CA to ack received CA
Local state :
Connection to B :
- established
- current_seq = z
CA (seq=z, ack=w)
Connection established

Fig. 46: Three-way handshake : recovery from a duplicate *CA*

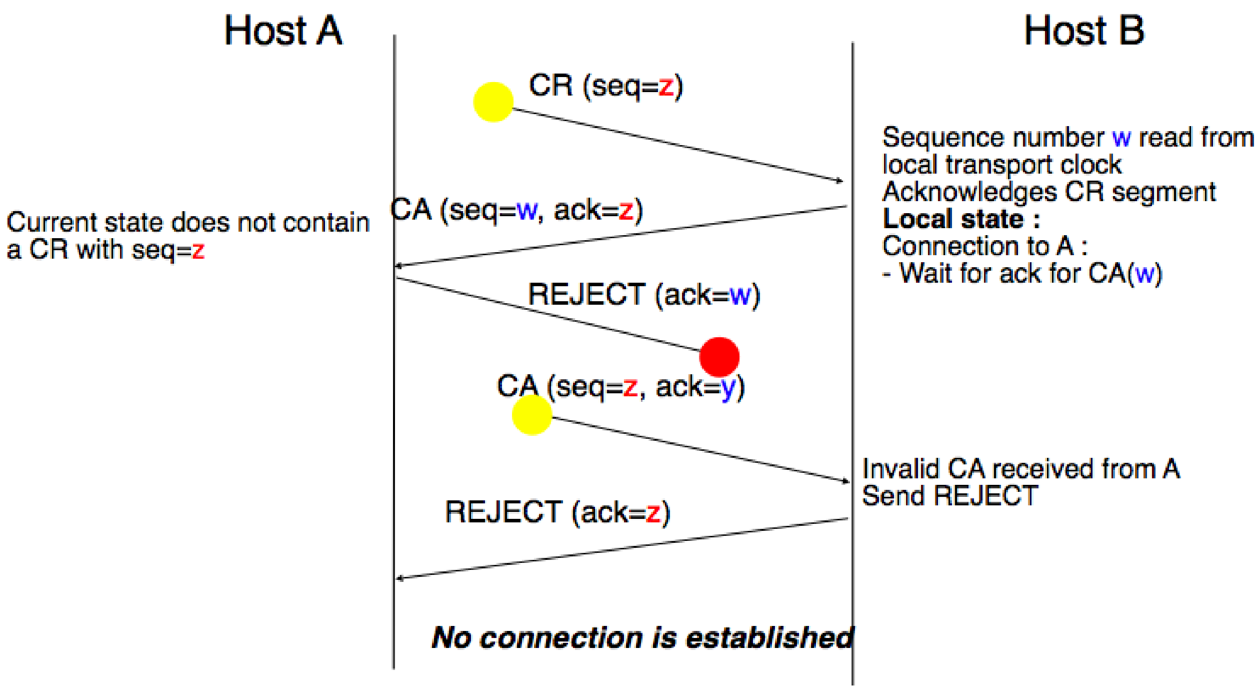
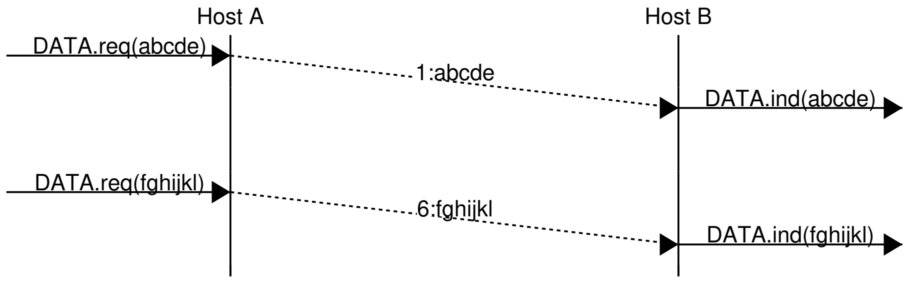


Fig. 47: Three-way handshake : recovery from duplicates *CR* and *CA*

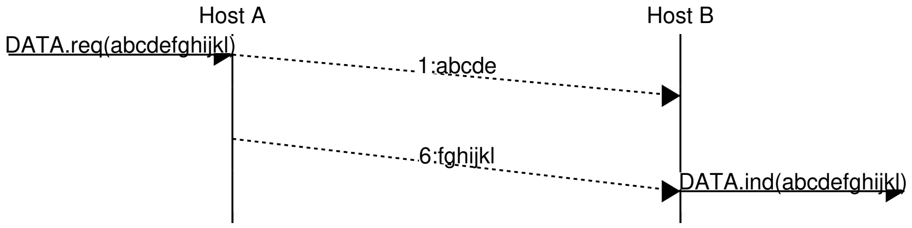
To deal with all these characteristics of the network layer, we need to adapt the techniques that we have introduced in the datalink layer.

The first point which is common between the two layers is that both use CRCs or checksums to detect transmission errors. Each segment contains a CRC/checksum which is computed over the entire segment (header and payload) by the sender and inserted in the header. The receiver recomputes the CRC/checksum for each received segment and discards all segments with an invalid CRC.

Reliable transport protocols also use sequence numbers and acknowledgment numbers. While reliable protocols in the datalink layer use one sequence number per frame, reliable transport protocols consider all the data transmitted as a stream of bytes. In these protocols, the sequence number placed in the segment header corresponds to the position of the first byte of the payload in the bytestream. This sequence number allows detecting losses but also enables the receiver to reorder the out-of-sequence segments. This is illustrated in the figure below.



Using sequence numbers to count bytes has also one advantage when the transport layer needs to fragment SDUs in several segments. The figure below shows the fragmentation of a large SDU in two segments. Upon reception of the segments, the receiver will use the sequence numbers to correctly reorder the data.



Compared to reliable protocols in the datalink layer, reliable transport protocols encode their sequence numbers using more bits. 32 bits and 64 bits sequence numbers are frequent in the transport layer while some datalink layer protocols encode their sequence numbers in an 8 bits field. This large sequence number space is motivated by two reasons. First, since the sequence number is incremented for each transmitted byte, a single segment may consume one or several thousands of sequence numbers. Second, a reliable transport protocol must be able to detect delayed segments. This can only be done if the number of bytes transmitted during the MSL period is smaller than the sequence number space. Otherwise, there is a risk of accepting duplicate segments.

Go-back-n and *selective repeat* can be used in the transport layer as in the datalink layer. Since the network layer does not guarantee an in-order delivery of the packets, a transport entity should always store the segments that it receives out-of-sequence. For this reason, most transport protocols will opt for some form of selective repeat mechanism.

In the datalink layer, the sliding window has usually a fixed size which depends on the amount of buffers allocated to the datalink layer entity. Such a datalink layer entity usually serves one or a few network layer entities. In the transport layer, the situation is different. A single transport layer entity serves a large and varying number of application processes. Each transport layer entity manages a pool of buffers that needs to be shared between all these processes. Transport entity are usually implemented inside the operating system kernel and shares memory with other parts of the system. Furthermore, a transport layer entity must support several (possibly hundreds or thousands) of transport connections at the same time. This implies that the memory which can be used to support the sending or the receiving buffer of a transport connection may change during the lifetime of the connection[3] . Thus, a transport protocol must

[3] For a discussion on how the sending buffer can change, see e.g. [SMM1998]

allow the sender and the receiver to adjust their window sizes.

To deal with this issue, transport protocols allow the receiver to advertise the current size of its receiving window in all the acknowledgments that it sends. The receiving window advertised by the receiver bounds the size of the sending buffer used by the sender. In practice, the sender maintains two state variables : *swin*, the size of its sending window (that may be adjusted by the system) and *rwin*, the size of the receiving window advertised by the receiver. At any time, the number of unacknowledged segments cannot be larger than $\min(swin, rwin)$[4] . The utilization of dynamic windows is illustrated in the figure below.

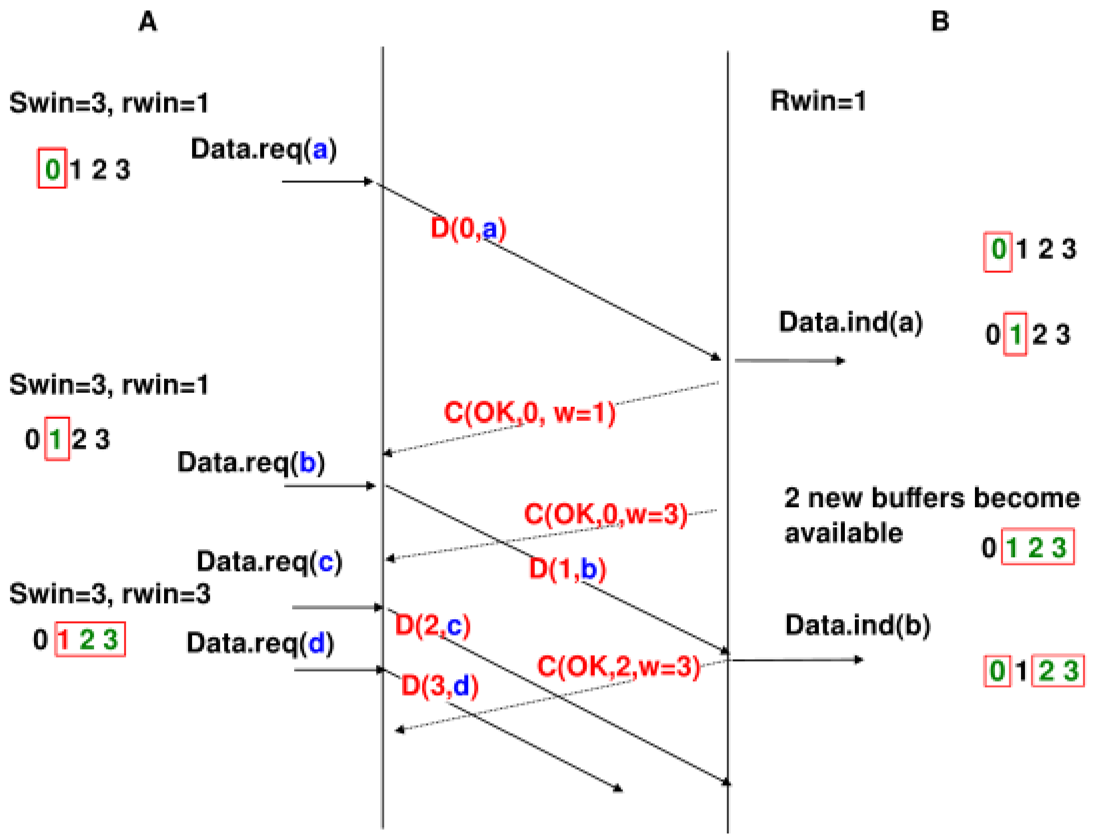


Fig. 48: Dynamic receiving window

The receiver may adjust its advertised receive window based on its current memory consumption, but also to limit the bandwidth used by the sender. In practice, the receive buffer can also shrink as the application may not able to process the received data quickly enough. In this case, the receive buffer may be completely full and the advertised receive window may shrink to *0*. When the sender receives an acknowledgment with a receive window set to *0*, it is blocked until it receives an acknowledgment with a positive receive window. Unfortunately, as shown in the figure below, the loss of this acknowledgment could cause a deadlock as the sender waits for an acknowledgment while the receiver is waiting for a data segment.

To solve this problem, transport protocols rely on a special timer : the *persistence timer*. This timer is started by the sender whenever it receives an acknowledgment advertising a receive window set to *0*. When the timer expires, the sender retransmits an old segment in order to force the receiver to send a new acknowledgment, and hence send the current receive window size.

To conclude our description of the basic mechanisms found in transport protocols, we still need to discuss the impact of segments arriving in the wrong order. If two consecutive segments are reordered, the receiver relies on their sequence numbers to reorder them in its receive buffer. Unfortunately, as transport protocols reuse the same sequence number

[4] Note that if the receive window shrinks, it might happen that the sender has already sent a segment that is not anymore inside its window. This segment will be discarded by the receiver and the sender will retransmit it later.

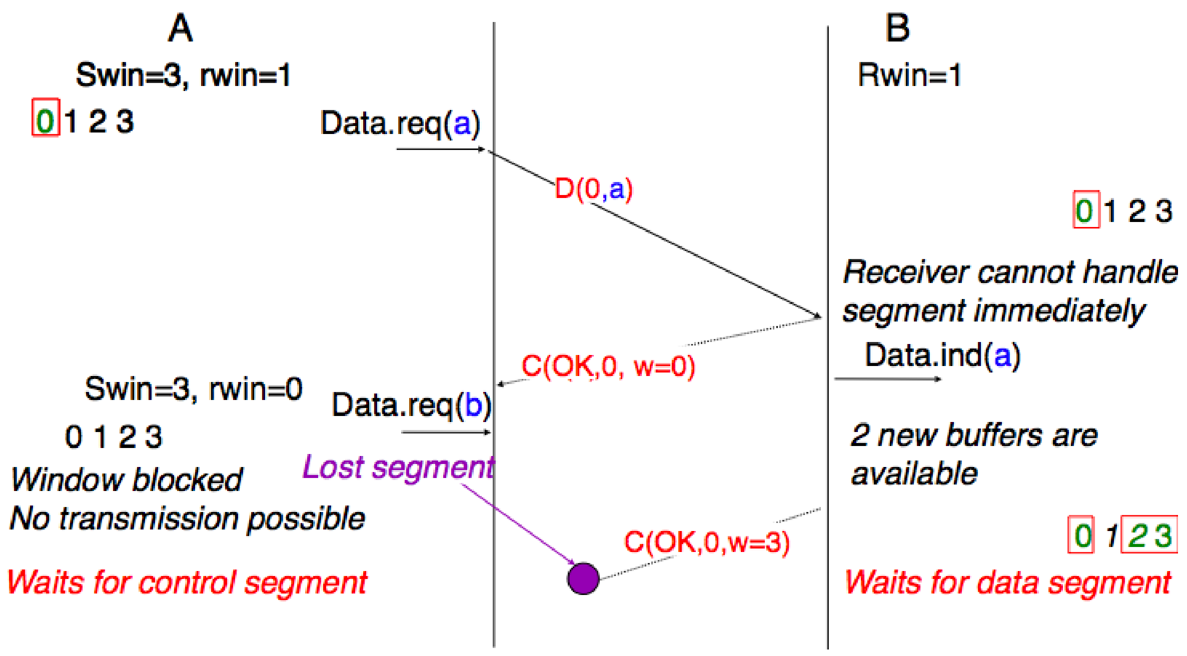


Fig. 49: Risk of deadlock with dynamic windows

for different segments, if a segment is delayed for a prolonged period of time, it might still be accepted by the receiver. This is illustrated in the figure below where segment *D(1,b)* is delayed.

To deal with this problem, transport protocols combine two solutions. First, they use 32 bits or more to encode the sequence number in the segment header. This increases the overhead, but also increases the delay between the transmission of two different segments having the same sequence number. Second, transport protocols require the network layer to enforce a *Maximum Segment Lifetime (MSL)*. The network layer must ensure that no packet remains in the network for more than MSL seconds. In the Internet the MSL is assumed[5] to be 2 minutes **RFC 793**. Note that this limits the maximum bandwidth of a transport protocol. If it uses n bits to encode its sequence numbers, then it cannot send more than 2^n segments every MSL seconds.

Connection release

When we discussed the connection-oriented service, we mentioned that there are two types of connection releases : *abrupt release* and *graceful release*.

The first solution to release a transport connection is to define a new control segment (e.g. the *DR* segment for Disconnection Request) and consider the connection to be released once this segment has been sent or received. This is illustrated in the figure below.

As the entity that sends the *DR* segment cannot know whether the other entity has already sent all its data on the connection, SDUs can be lost during such an *abrupt connection release*.

The second method to release a transport connection is to release independently the two directions of data transfer. Once a user of the transport service has sent all its SDUs, it performs a *DISCONNECT.req* for its direction of data transfer. The transport entity sends a control segment to request the release of the connection *after* the delivery of all previous SDUs to the remote user. This is usually done by placing in the *DR* the next sequence number and by delivering the *DISCONNECT.ind* only after all previous *DATA.ind*. The remote entity confirms the reception of the

[5] In reality, the Internet does not strictly enforce this MSL. However, it is reasonable to expect that most packets on the Internet will not remain in the network during more than 2 minutes. There are a few exceptions to this rule, such as **RFC 1149** whose implementation is described in http://www.blug.linux.no/rfc1149/ but there are few real links supporting **RFC 1149** in the Internet.

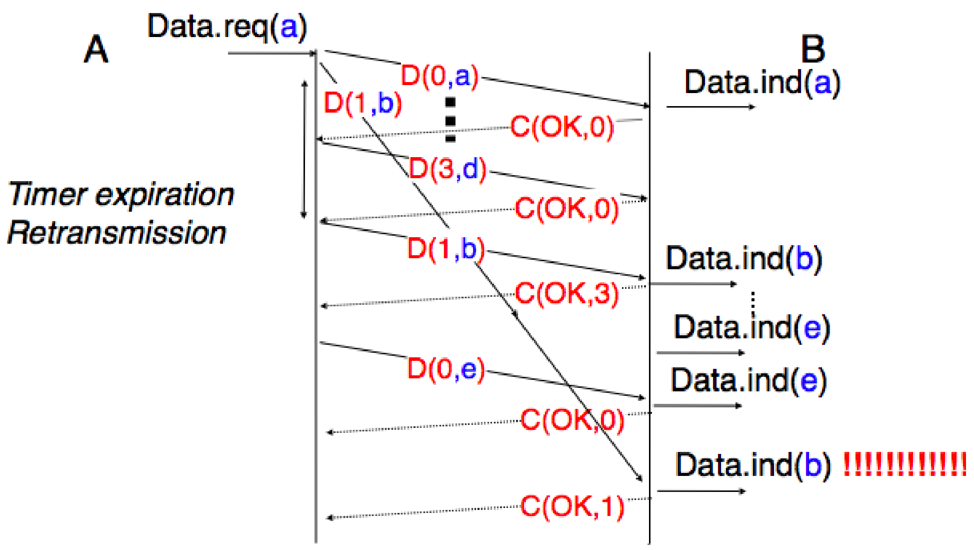


Fig. 50: Ambiguities caused by excessive delays

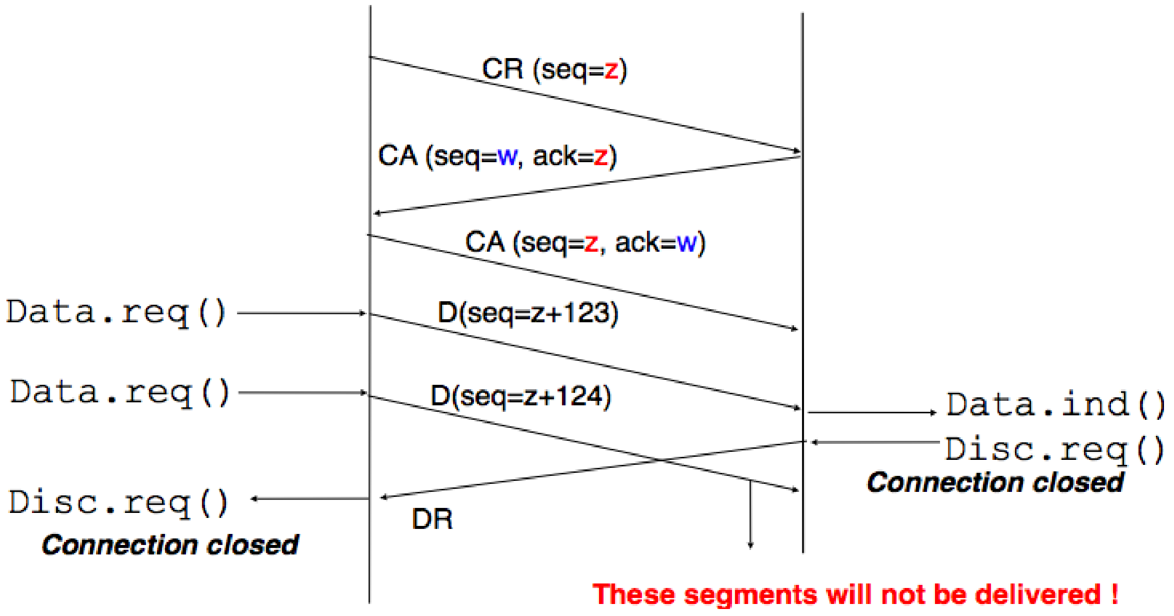


Fig. 51: Abrupt connection release

DR segment and the release of the corresponding direction of data transfer by returning an acknowledgment. This is illustrated in the figure below.

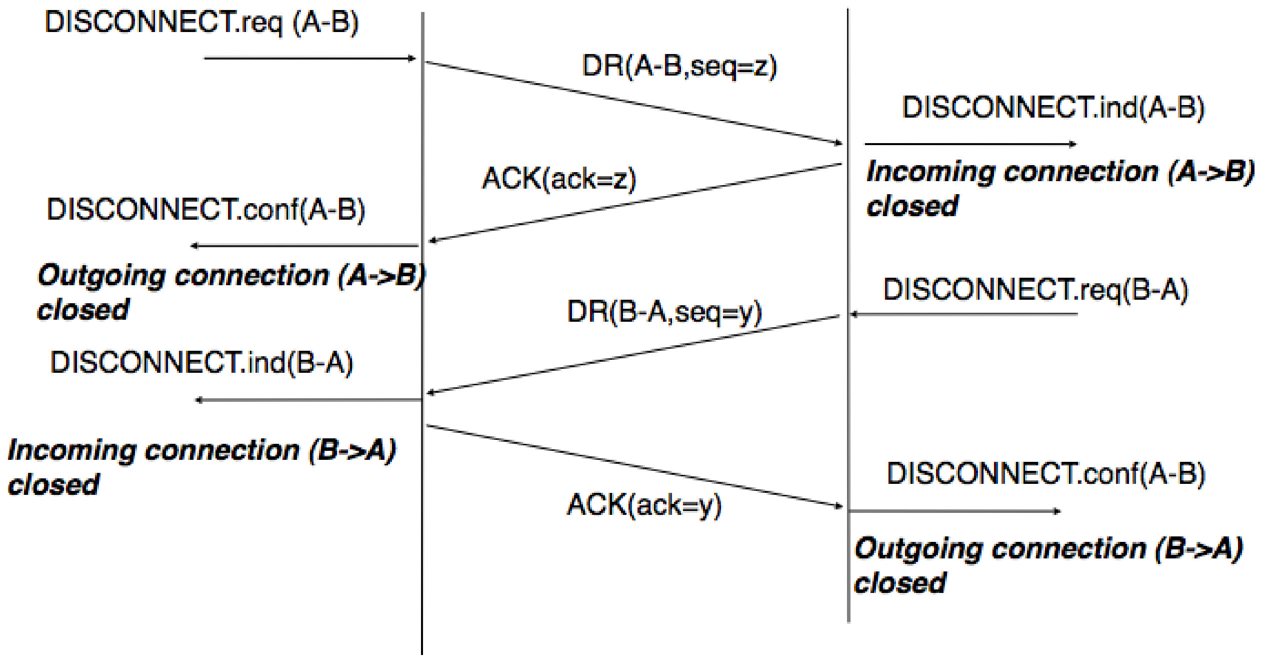


Fig. 52: Graceful connection release

2.5 Naming and addressing

The network and the transport layers rely on addresses that are encoded as fixed size bit strings. A network layer address uniquely identifies a host. Several transport layer entities can use the service of the same network layer. For example, a reliable transport protocol and a connectionless transport protocol can coexist on the same host. In this case, the network layer multiplexes the segments produced by these two protocols. This multiplexing is usually achieved by placing in the network packet header a field that indicates which transport protocol should process the segment. Given that there are few different transport protocols, this field does not need to be long. The port numbers play a similar role in the transport layer since they enable it to multiplex data from several application processes.

While addresses are natural for the network and transport layer entities, humans prefer to use names when interacting with network services. Names can be encoded as a character string and a mapping services allows applications to map a name into the corresponding address. Using names is friendlier for humans, but it also provides a level of indirection which is very useful in many situations.

In the early days of the Internet, only a few hosts (mainly minicomputers) were connected to the network. The most popular applications were *remote login* and file transfer. By 1983, there were already five hundred hosts attached to the Internet [Zakon]. Each of these hosts were identified by a unique address. Forcing human users to remember the addresses of the hosts that they wanted to use was not user-friendly. Humans prefer to remember names, and use them when needed. Using names as aliases for addresses is a common technique in Computer Science. It simplifies the development of applications and allows the developer to ignore the low level details. For example, by using a programming language instead of writing machine code, a developer can write software without knowing whether the variables that it uses are stored in memory or inside registers.

Because names are at a higher level than addresses, they allow (both in the example of programming above, and on the Internet) to treat addresses as mere technical identifiers, which can change at will. Only the names are stable.

The first solution that allowed applications to use names was the *hosts.txt* file. This file is similar to the symbol table found in compiled code. It contains the mapping between the name of each Internet host and its associated address[1]. It was maintained by SRI International that coordinated the Network Information Center (NIC). When a new host was connected to the network, the system administrator had to register its name and address at the NIC. The NIC updated the *hosts.txt* file on its server. All Internet hosts regularly retrieved the updated *hosts.txt* file from the SRI server. This file was stored at a well-known location on each Internet host (see **RFC 952**) and networked applications could use it to find the address corresponding to a name.

A *hosts.txt* file can be used when there are up to a few hundred hosts on the network. However, it is clearly not suitable for a network containing thousands or millions of hosts. A key issue in a large network is to define a suitable naming scheme. The ARPANet initially used a flat naming space, i.e. each host was assigned a unique name. To limit collisions between names, these names usually contained the name of the institution and a suffix to identify the host inside the institution (a kind of poor man's hierarchical naming scheme). On the ARPANet few institutions had several hosts connected to the network.

However, the limitations of a flat naming scheme became clear before the end of the ARPANet and **RFC 819** proposed a hierarchical naming scheme. While **RFC 819** discussed the possibility of organizing the names as a directed graph, the Internet opted for a tree structure capable of containing all names. In this tree, the top-level domains are those that are directly attached to the root. The first top-level domain was *.arpa*[2]. This top-level name was initially added as a suffix to the names of the hosts attached to the ARPANet and listed in the *hosts.txt* file. In 1984, the *.gov*, *.edu*, *.com*, *.mil* and *.org* generic top-level domain names were added. **RFC 1032** proposed the utilization of the two letter *ISO-3166* country codes as top-level domain names. Since *ISO-3166* defines a two letter code for each country recognized by the United Nations, this allowed all countries to automatically have a top-level domain. These domains include *.be* for Belgium, *.fr* for France, *.us* for the USA, *.ie* for Ireland or *.tv* for Tuvalu, a group of small islands in the Pacific or *.tm* for Turkmenistan. The set of top-level domain-names is managed by the Internet Corporation for Assigned Names and Numbers (*ICANN*). *ICANN* adds generic top-level domains that are not related to a country and the *.cat* top-level domain has been registered for the Catalan language. There are ongoing discussions within *ICANN* to increase the number of top-level domains.

Each top-level domain is managed by an organization that decides how sub-domain names can be registered. Most top-level domain names use a first-come first served system, and allow anyone to register domain names, but there are some exceptions. For example, *.gov* is reserved for the US government, *.int* is reserved for international organizations and names in the *.ca* are mainly reserved for companies or users that are present in Canada. The syntax of the domain

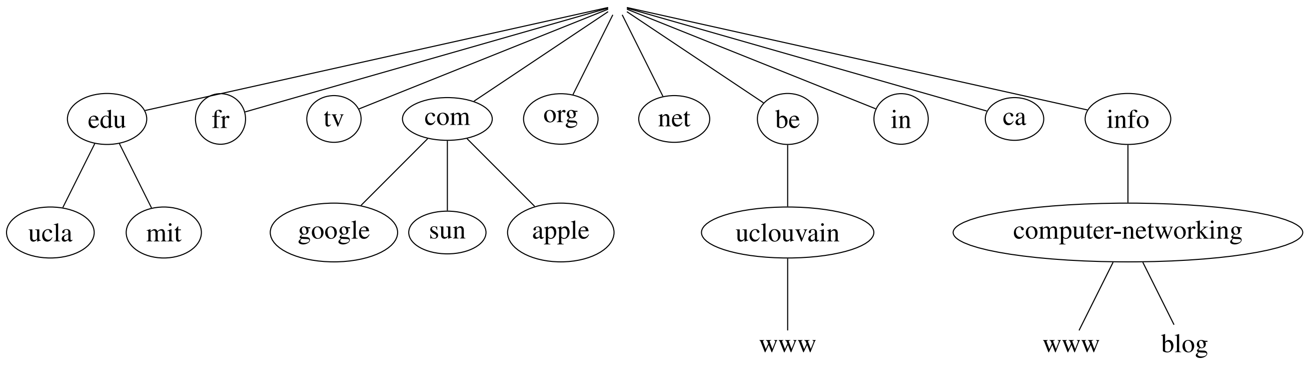


Fig. 53: The tree of domain names

names has been defined more precisely in **RFC 1035**. This document recommends the following *BNF* for fully qualified domain names (the domain names themselves have a much richer syntax).

[1] The *hosts.txt* file is not maintained anymore. A historical snapshot from April 1984 is available from http://ftp.univie.ac.at/netinfo/netinfo/hosts.txt

[2] See http://www.donelan.com/dnstimeline.html for a time line of DNS related developments.

Listing 1: BNF of the fully qualified host names

```
domain ::= subdomain | " "
subdomain ::= label | subdomain "." label
label ::= letter [ [ ldh-str ] let-dig ]
ldh-str ::= let-dig-hyp | let-dig-hyp ldh-str
let-dig-hyp ::= let-dig | "-"
let-dig ::= letter | digit
letter ::= any one of the 52 alphabetic characters A through Z in upper case and a␣
↪through z in lower case
digit ::= any one of the ten digits 0 through 9
```

This grammar specifies that a host name is an ordered list of labels separated by the dot (.) character. Each label can contain letters, numbers and the hyphen character (-)[3]. Fully qualified domain names are read from left to right. The first label is a hostname or a domain name followed by the hierarchy of domains and ending with the root implicitly at the right. The top-level domain name must be one of the registered TLDs[4]. For example, in the above figure, *www.computer-networking.info* corresponds to a host named *www* inside the *computer-networking* domain that belongs to the *info* top-level domain.

Note: Some visually similar characters have different character codes

The Domain Name System was created at a time when the Internet was mainly used in North America. The initial design assumed that all domain names would be composed of letters and digits **RFC 1035**. As Internet usage grew in other parts of the world, it became important to support non-ASCII characters. For this, extensions have been proposed to the Domain Name System **RFC 3490**. In a nutshell, the solution that is used to support Internationalized Domain Names works as follows. First, it is possible to use most of the Unicode characters to encode domain names and hostnames, with a few exceptions (for example, the dot character cannot be part of a name since it is used as a separator). Once a domain name has been encoded as a series of Unicode characters, it is then converted into a string that contains the `xn--` prefix and a sequence of ASCII characters. More details on these algorithms can be found in **RFC 3490** and **RFC 3492**.

The possibility of using all Unicode characters to create domain names opened a new form of attack called the homograph attack. This attack occurs when two character strings or domain names are visually similar but do not correspond to the same server. A simple example is https://G00GLE.COM and https://GOOGLE.COM. These two URLs are visually close but they correspond to different names (the first one does not point to a valid server[5]). With other Unicode characters, it is possible to construct domain names are visually equivalent to existing ones. See [Zhe2017] for additional details on this attack.

This hierarchical naming scheme is a key component of the Domain Name System (DNS). The DNS is a distributed database that contains mappings between fully qualified domain names and addresses. The DNS uses the client-server model. The clients are hosts or applications that need to retrieve the mapping for a given name. Each *nameserver* stores part of the distributed database and answers the queries sent by clients. There is at least one *nameserver* that is responsible for each domain. In the figure below, domains are represented by circles and there are three hosts inside domain *dom* (*h1*, *h2* and *h3*) and three hosts inside domain *a.sdom1.dom*. As shown in the figure below, a sub-domain may contain both host names and sub-domains. A *nameserver* that is responsible for domain *dom* can directly answer the following queries :

- the address of any host residing directly inside domain *dom* (e.g. *h2.dom* in the figure above)

[3] This specification evolved later to support domain names written by using other character sets than us-ASCII **RFC 5890**. This extension is important to support languages other than English, but a detailed discussion is outside the scope of this document.

[4] The official list of top-level domain names is maintained by *IANA* at http://data.iana.org/TLD/tlds-alpha-by-domain.txt Additional information about these domains may be found at http://en.wikipedia.org/wiki/List_of_Internet_top-level_domains

[5] It is interesting to note that to prevent any homograph attack, Google Inc. registered the *g00gle.com* domain name but does not apparently use it.

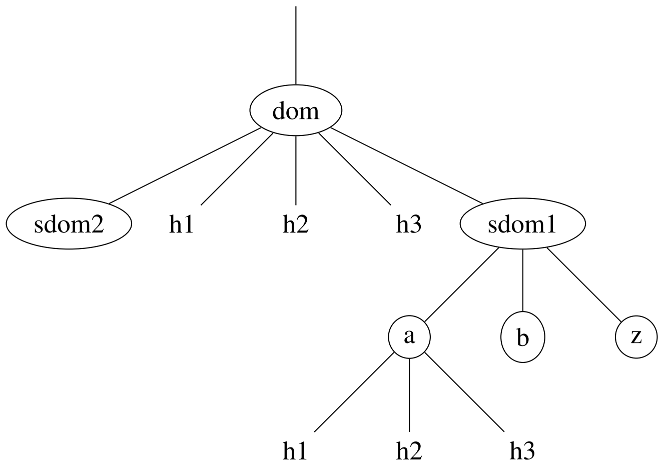


Fig. 54: A simple tree of domain names

- the nameserver(s) that are responsible for any direct sub-domain of domain *dom* (i.e. *sdom1.dom* and *sdom2.dom* in the figure above, but not *z.sdom1.dom*)

To retrieve the mapping for host *h2.dom*, a client sends its query to the name server that is responsible for domain *.dom*. The name server directly answers the query. To retrieve a mapping for *h3.a.sdom1.dom* a DNS client first sends a query to the name server that is responsible for the *.dom* domain. This nameserver returns the nameserver that is responsible for the *sdom1.dom* domain. This nameserver can now be contacted to obtain the nameserver that is responsible for the *a.sdom1.dom* domain. This nameserver can be contacted to retrieve the mapping for the *h3.a.sdom1.dom* name. Thanks to this structure, it is possible for a DNS client to obtain the mapping of any host inside the *.dom* domain or any of its subdomains. To ensure that any DNS client will be able to resolve any fully qualified domain name, there are special nameservers that are responsible for the root of the domain name hierarchy. These nameservers are called *root nameserver*.

Each root nameserver maintains the list[6] of all the nameservers that are responsible for each of the top-level domain names and their addresses[7]. All root nameservers cooperate and provide the same answers. By querying any of the root nameservers, a DNS client can obtain the nameserver that is responsible for any top-level-domain name. From this nameserver, it is possible to resolve any domain name.

To be able to contact the root nameservers, each DNS client must know their addresses. This implies, that DNS clients must maintain an up-to-date list of the addresses of the root nameservers. Without this list, it is impossible to contact the root nameservers. Forcing all Internet hosts to maintain the most recent version of this list would be difficult from an operational point of view. To solve this problem, the designers of the DNS introduced a special type of DNS server : the DNS resolvers. A *resolver* is a server that provides the name resolution service for a set of clients. A network usually contains a few resolvers. Each host in these networks is configured to send all its DNS queries via one of its local resolvers. These queries are called *recursive queries* as the *resolver* must recursively send requests through the hierarchy of nameservers to obtain the *answer*.

DNS resolvers have several advantages over letting each Internet host query directly nameservers. Firstly, regular Internet hosts do not need to maintain the up-to-date list of the addresses of the root servers. Secondly, regular Internet hosts do not need to send queries to nameservers all over the Internet. Furthermore, as a DNS resolver serves a large number of hosts, it can cache the received answers. This allows the resolver to quickly return answers for popular DNS queries and reduces the load on all DNS servers [JSBM2002].

[6] A copy of the information maintained by each root nameserver is available at http://www.internic.net/zones/root.zone

[7] Until February 2008, the root DNS servers only had IPv4 addresses. IPv6 addresses were slowly added to the root DNS servers to avoid creating problems as discussed in http://www.icann.org/en/committees/security/sac018.pdf As of February 2021, there remain a few DNS root servers that are still not reachable using IPv6. The full list is available at http://www.root-servers.org/

2.5.1 Benefits of names

In addition to being more human friendly, using names instead of addresses inside applications has several important benefits. To understand them, let us consider a popular application that provides information stored on servers. This application involves clients and servers. The server provides information upon requests from client processes. A first deployment of this application would be to rely only on addresses. In this case, the server process would be installed on one host and the clients would connect to this server to retrieve information. Such a deployment has several drawbacks :

- if the server process moves to another physical server, all clients must be informed about the new server address
- if there are many concurrent clients, the load of the server will increase without any possibility of adding another server without changing the server addresses used by the clients

Using names solves these problems and provides additional benefits. If the clients are configured with the name of the server, they will query the name service before contacting the server. The name service will resolve the name into the corresponding address. If a server process needs to move from one physical server to another, it suffices to update the name to address mapping on the name service to allow all clients to connect to the new server. The name service also enables the servers to better sustain the load. Assume a very popular server which is accessed by millions of users. This service cannot be provided by a single physical server due to performance limitations. Thanks to the utilization of names, it is possible to scale this service by mapping a given name to a set of addresses. When a client queries the name service for the server's name, the name service returns one of the addresses in the set. Various strategies can be used to select one particular address inside the set of addresses. A first strategy is to select a random address in the set. A second strategy is to maintain information about the load on the servers and return the address of the less loaded server. Note that the list of server addresses does not need to remain fixed. It is possible to add and remove addresses from the list to cope with load fluctuations. Another strategy is to infer the location of the client from the name request and return the address of the closest server.

Mapping a single name onto a set of addresses allows popular servers to dynamically scale. There are also benefits in mapping multiple names, possibly a large number of them, onto a single address. Consider the case of information servers run by individuals or SMEs. Some of these servers attract only a few clients per day. Using a single physical server for each of these services would be a waste of resources. A better approach is to use a single server for a set of services that are all identified by different names. This enables service providers to support a large number of servers processes, identified by different names, onto a single physical server. If one of these server processes becomes very popular, it will be possible to map its name onto a set of addresses to be able to sustain the load. There are some deployments where this mapping is done dynamically in function of the load.

Names provide a lot of flexibility compared to addresses. For the network, they play a similar role as variables in programming languages. No programmer using a high-level programming language would consider using hardcoded values instead of variables. For the same reasons, all networked applications should depend on names and avoid dealing with addresses as much as possible.

2.6 Sharing resources

A network is designed to support a potentially large number of users that exchange information with each other. These users produce and consume information which is exchanged through the network. To support its users, a network uses several types of resources. It is important to keep in mind the different resources that are shared inside the network.

The first and more important resource inside a network is the link bandwidth. There are two situations where link bandwidth needs to be shared between different users. The first situation is when several hosts are attached to the same physical link. This situation mainly occurs in Local Area Networks (LAN). A LAN is a network that efficiently interconnects several hosts (usually a few dozens to a few hundreds) in the same room, building or campus. Consider for example a network with five hosts. Any of these hosts needs to be able to exchange information with any of the other hosts. A first organization for this LAN is the full-mesh.

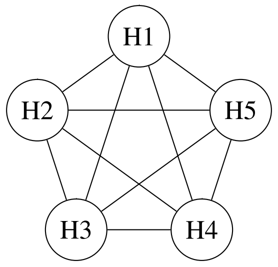


Fig. 55: A full-mesh network

The full-mesh is the most reliable and highest performing network to interconnect these five hosts. However, this network organization has two important drawbacks. First, if a network contains *n* hosts, then $\frac{n \times (n-1)}{2}$ links are required. If the network contains more than a few hosts, it becomes impossible to lay down the required physical links. Second, if the network contains *n* hosts, then each host must have $n - 1$ interfaces to terminate $n - 1$ links. This is beyond the capabilities of most hosts. Furthermore, if a new host is added to the network, new links have to be laid down and one interface has to be added to each participating host. However, full-mesh has the advantage of providing the lowest delay between the hosts and the best resiliency against link failures. In practice, full-mesh networks are rarely used except when there are few network nodes and resiliency is key.

The second possible physical organization, which is also used inside computers to connect different extension cards, is the bus. In a bus network, all hosts are attached to a shared medium, usually a cable through a single interface. When one host sends an electrical signal on the bus, the signal is received by all hosts attached to the bus. A drawback of bus-based networks is that if the bus is physically cut, then the network is split into two isolated networks. For this reason, bus-based networks are sometimes considered to be difficult to operate and maintain, especially when the cable is long and there are many places where it can break. Such a bus-based topology was used in early Ethernet networks.

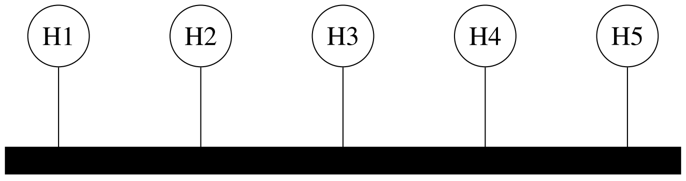


Fig. 56: A network organized as a bus

A third organization of a computer network is a star topology. In such networks, hosts have a single physical interface and there is one physical link between each host and the center of the star. The node at the center of the star can be either a piece of equipment that amplifies an electrical signal, or an active device, such as a piece of equipment that understands the format of the messages exchanged through the network. Of course, the failure of the central node implies the failure of the network. However, if one physical link fails (e.g. because the cable has been cut), then only one node is disconnected from the network. In practice, star-shaped networks are easier to operate and maintain than bus-shaped networks. Many network administrators also appreciate the fact that they can control the network from a central point. Administered from a Web interface, or through a console-like connection, the center of the star is a useful point of control (enabling or disabling devices) and an excellent observation point (usage statistics).

A fourth physical organization of a network is the ring topology. Like the bus organization, each host has a single physical interface connecting it to the ring. Any signal sent by a host on the ring will be received by all hosts attached to the ring. From a redundancy point of view, a single ring is not the best solution, as the signal only travels in one direction on the ring; thus if one of the links composing the ring is cut, the entire network fails. In practice, such rings have been used in local area networks, but are now often replaced by star-shaped networks. In metropolitan networks,

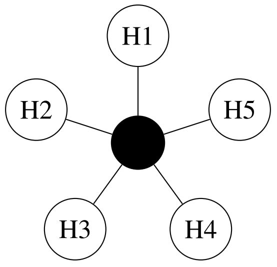


Fig. 57: A network organized as a star

rings are often used to interconnect multiple locations. In this case, two parallel links, composed of different cables, are often used for redundancy. With such a dual ring, when one ring fails all the traffic can be quickly switched to the other ring.

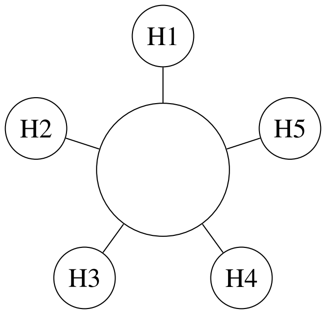


Fig. 58: A network organized as a ring

A fifth physical organization of a network is the tree. Such networks are typically used when a large number of customers must be connected in a very cost-effective manner. Cable TV networks are often organized as trees.

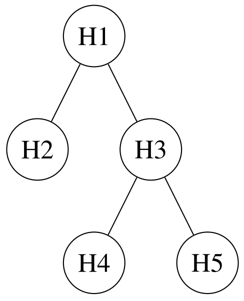


Fig. 59: A network organized as a tree

2.6.1 Sharing bandwidth

In all these networks, except the full-mesh, the link bandwidth is shared among all connected hosts. Various algorithms have been proposed and are used to efficiently share the access to this resource. We explain several of them in the Medium Access Control section below.

Note: Fairness in computer networks

Sharing resources is important to ensure that the network efficiently serves its user. In practice, there are many ways to share resources. Some resource sharing schemes consider that some users are more important than others and should obtain more resources. For example, on the roads, police cars and ambulances have priority. In some cities, traffic lanes are reserved for buses to promote public services, ... In computer networks, the same problem arise. Given that resources are limited, the network needs to enable users to efficiently share them. Before designing an efficient resource sharing scheme, one needs to first formalize its objectives. In computer networks, the most popular objective for resource sharing schemes is that they must be *fair*. In a simple situation, for example two hosts using a shared 2 Mbps link, the sharing scheme should allocate the same bandwidth to each user, in this case 1 Mbps. However, in a large networks, simply dividing the available resources by the number of users is not sufficient. Consider the network shown in the figure below where *A1* sends data to *A2*, *B1* to *B2*, ... In this network, how should we divide the bandwidth among the different flows ? A first approach would be to allocate the same bandwidth to each flow. In this case, each flow would obtain 5 Mbps and the link between *R2* and *R3* would not be fully loaded. Another approach would be to allocate 10 Mbps to *A1-A2*, 20 Mbps to *C1-C2* and nothing to *B1-B2*. This is clearly unfair.

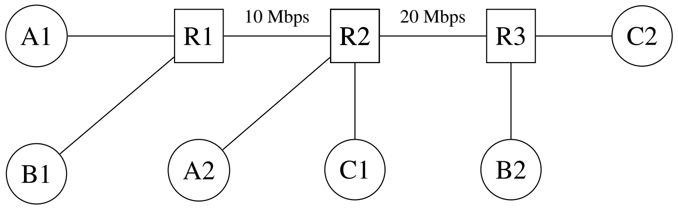


Fig. 60: A small network

In large networks, fairness is always a compromise. The most widely used definition of fairness is the *max-min fairness*. A bandwidth allocation in a network is said to be *max-min fair* if it is such that it is impossible to allocate more bandwidth to one of the flows without reducing the bandwidth of a flow that already has a smaller allocation than the flow that we want to increase. If the network is completely known, it is possible to derive a *max-min fair* allocation as follows. Initially, all flows have a null bandwidth and they are placed in the candidate set. The bandwidth allocation of all flows in the candidate set is increased until one link becomes congested. At this point, the flows that use the congested link have reached their maximum allocation. They are removed from the candidate set and the process continues until the candidate set becomes empty.

In the above network, the allocation of all flows would grow until *A1-A2* and *B1-B2* reach 5 Mbps. At this point, link *R1-R2* becomes congested and these two flows have reached their maximum. The allocation for flow *C1-C2* can increase until reaching 15 Mbps. At this point, link *R2-R3* is congested. To increase the bandwidth allocated to *C1-C2*, one would need to reduce the allocation to flow *B1-B2*. Similarly, the only way to increase the allocation to flow *B1-B2* would require a decrease of the allocation to *A1-A2*.

2.6.2 Network congestion

Sharing bandwidth among the hosts directly attached to a link is not the only sharing problem that occurs in computer networks. To understand the general problem, let us consider a very simple network which contains only point-to-point links. This network contains three hosts and two routers. All the links inside the network have the same capacity. For example, let us assume that all links have a bandwidth of 1000 bits per second and that the hosts send packets containing exactly one thousand bits.

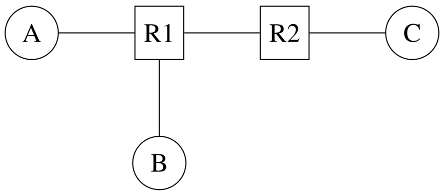


Fig. 61: A small network

In the network above, consider the case where host *A* is transmitting packets to destination *C*. *A* can send one packet per second and its packets will be delivered to *C*. Now, let us explore what happens when host *B* also starts to transmit a packet. Node *R1* will receive two packets that must be forwarded to *R2*. Unfortunately, due to the limited bandwidth on the *R1-R2* link, only one of these two packets can be transmitted. The outcome of the second packet will depend on the available buffers on *R1*. If *R1* has one available buffer, it could store the packet that has not been transmitted on the *R1-R2* link until the link becomes available. If *R1* does not have available buffers, then the packet needs to be discarded.

Besides the link bandwidth, the buffers on the network nodes are the second type of resource that needs to be shared inside the network. The node buffers play an important role in the operation of the network because that can be used to absorb transient traffic peaks. Consider again the example above. Assume that on average host *A* and host *B* send a group of three packets every ten seconds. Their combined transmission rate (0.6 packets per second) is, on average, lower than the network capacity (1 packet per second). However, if they both start to transmit at the same time, node *R1* will have to absorb a burst of packets. This burst of packets is a small *network congestion*. We will say that a network is congested, when the sum of the traffic demand from the hosts is larger than the network capacity $\sum demand > capacity$. This *network congestion* problem is one of the most difficult resource sharing problem in computer networks. *Congestion* occurs in almost all networks. Minimizing the amount of congestion is a key objective for many network operators. In most cases, they will have to accept transient congestion, i.e. congestion lasting a few seconds or perhaps minutes, but will want to prevent congestion that lasts days or months. For this, they can rely on a wide range of solutions. We briefly present some of these in the paragraphs below.

If *R1* has enough buffers, it will be able to absorb the load without having to discard packets. The packets sent by hosts *A* and *B* will reach their final destination *C*, but will experience a longer delay than when they are transmitting alone. The amount of buffering on the network node is the first parameter that a network operator can tune to control congestion inside his network. Given the decreasing cost of memory, one could be tempted to put as many buffers[1] as possible on the network nodes. Let us consider this case in the network above and assume that *R1* has infinite buffers. Assume now that hosts *A* and *B* try to transmit a file that corresponds to one thousand packets each. Both are using a reliable protocol that relies on go-back-n to recover from transmission errors. The transmission starts and packets start to accumulate in *R1*'s buffers. The presence of these packets in the buffers increases the delay between the transmission of a packet by *A* and the return of the corresponding acknowledgment. Given the increasing delay, host *A* (and *B* as well) will consider that some of the packets that it sent have been lost. These packets will be retransmitted and will enter the buffers of *R1*. The occupancy of the buffers of *R1* will continue to increase and the delays as well. This will cause new retransmissions, . . . In the end, only one file will be delivered (very slowly) to the destination, but the link *R1-R2* will transfer much more bytes than the size of the file due to the multiple copies of the same packets. This

[1] There are still some vendors that try to put as many buffers as possible on their routers. A recent example is the buffer bloat problem that plagues some low-end Internet routers [GN2011].

is known as the *congestion collapse* problem **RFC 896**. Congestion collapse is the nightmare for network operators. When it happens, the network carries packets without delivering useful data to the end users.

Note: Congestion collapse on the Internet

Congestion collapse is unfortunately not only an academic experience. Van Jacobson reports in [Jacobson1988] one of these events that affected him while he was working at the Lawrence Berkeley Laboratory (LBL). LBL was two network nodes away from the University of California in Berkeley. At that time, the link between the two sites had a bandwidth of 32 Kbps, but some hosts were already attached to 10 Mbps LANs. "In October 1986, the data throughput from LBL to UC Berkeley ... dropped from 32 Kbps to 40 bps. We were fascinated by this sudden factor-of-thousand drop in bandwidth and embarked on an investigation of why things had gotten so bad." This work lead to the development of various congestion control techniques that have allowed the Internet to continue to grow without experiencing widespread congestion collapse events.

Besides bandwidth and memory, a third resource that needs to be shared inside a network is the (packet) processing capacity. To forward a packet, a router needs bandwidth on the outgoing link, but it also needs to analyze the packet header to perform a lookup inside its forwarding table. Performing these lookup operations require resources such as CPU cycles or memory accesses. Routers are usually designed to be able to sustain a given packet processing rate, measured in packets per second[2].

Note: Packets per second versus bits per second

The performance of network nodes (either routers or switches) can be characterized by two key metrics :

- the node's capacity measured in bits per second
- the node's lookup performance measured in packets per second

The node's capacity in bits per second mainly depends on the physical interfaces that it uses and also on the capacity of the internal interconnection (bus, crossbar switch, ...) between the different interfaces inside the node. Many vendors, in particular for low-end devices will use the sum of the bandwidth of the nodes' interfaces as the node capacity in bits per second. Measurements do not always match this maximum theoretical capacity. A well designed network node will usually have a capacity in bits per second larger than the sum of its link capacities. Such nodes will usually reach this maximum capacity when forwarding large packets.

When a network node forwards small packets, its performance is usually limited by the number of lookup operations that it can perform every second. This lookup performance is measured in packets per second. The performance may depend on the length of the forwarded packets. The key performance factor is the number of minimal size packets that are forwarded by the node every second. This rate can lead to a capacity in bits per second which is much lower than the sum of the bandwidth of the node's links.

Let us now try to present a broad overview of the congestion problem in networks. We will assume that the network is composed of dedicated links having a fixed bandwidth[3]. A network contains hosts that generate and receive packets and nodes (routers and switches) that forward packets. Assuming that each host is connected via a single link to the network, the largest demand is $\sum AccessLinks$. In practice, this largest demand is never reached and the network will be engineered to sustain a much lower traffic demand. The difference between the worst-case traffic demand and the sustainable traffic demand can be large, up to several orders of magnitude. Fortunately, the hosts are not completely dumb and they can adapt their traffic demand to the current state of the network and the available bandwidth. For this, the hosts need to *sense* the current level of congestion and adjust their own traffic demand based on the estimated congestion. Network nodes can react in different ways to network congestion and hosts can sense the level of congestion

[2] Some examples of the performance of various types of commercial networks nodes (routers and switches) may be found in http://www.cisco.com/web/partners/downloads/765/tools/quickreference/routerperformance.pdf and http://www.cisco.com/web/partners/downloads/765/tools/quickreference/switchperformance.pdf

[3] Some networking technologies allow to adjust dynamically the bandwidth of links. For example, some devices can reduce their bandwidth to preserve energy. We ignore these technologies in this basic course and assume that all links used inside the network have a fixed bandwidth.

in different ways.

Let us first explore which mechanisms can be used inside a network to control congestion and how these mechanisms can influence the behavior of the end hosts.

As explained earlier, one of the first manifestation of congestion on network nodes is the saturation of the network links that leads to a growth in the occupancy of the buffers of the node. This growth of the buffer occupancy implies that some packets will spend more time in the buffer and thus in the network. If hosts measure the network delays (e.g. by measuring the round-trip-time between the transmission of a packet and the return of the corresponding acknowledgment) they could start to sense congestion. On low bandwidth links, a growth in the buffer occupancy can lead to an increase of the delays which can be easily measured by the end hosts. On high bandwidth links, a few packets inside the buffer will cause a small variation in the delay which may not necessarily be larger that the natural fluctuations of the delay measurements.

If the buffer's occupancy continues to grow, it will overflow and packets will need to be discarded. Discarding packets during congestion is the second possible reaction of a network node to congestion. Before looking at how a node can discard packets, it is interesting to discuss qualitatively the impact of the buffer occupancy on the reliable delivery of data through a network. This is illustrated by the figure below, adapted from [Jain1990].

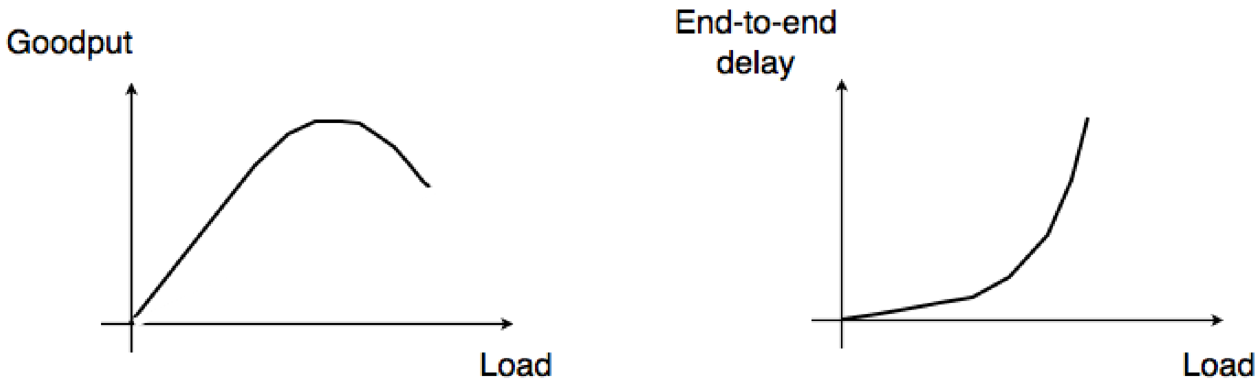


Fig. 62: Network congestion

When the network load is low, buffer occupancy and link utilization are low. The buffers on the network nodes are mainly used to absorb very short bursts of packets, but on average the traffic demand is lower than the network capacity. If the demand increases, the average buffer occupancy will increase as well. Measurements have shown that the total throughput increases as well. If the buffer occupancy is zero or very low, transmission opportunities on network links can be missed. This is not the case when the buffer occupancy is small but non zero. However, if the buffer occupancy continues to increase, the buffer becomes overloaded and the throughput does not increase anymore. When the buffer occupancy is close to the maximum, the throughput may decrease. This drop in throughput can be caused by excessive retransmissions of reliable protocols that incorrectly assume that previously sent packets have been lost while they are still waiting in the buffer. The network delay on the other hand increases with the buffer occupancy. In practice, a good operating point for a network buffer is a low occupancy to achieve high link utilization and also low delay for interactive applications.

Discarding packets is one of the signals that the network nodes can use to inform the hosts of the current level of congestion. Buffers on network nodes are usually used as FIFO queues to preserve packet ordering. Several *packet discard mechanisms* have been proposed for network nodes. These techniques basically answer two different questions :

- *What triggers a packet to be discarded ?* What are the conditions that lead a network node to decide to discard a packet? The simplest answer to this question is : *When the buffer is full*. Although this is a good congestion indication, it is probably not the best one from a performance viewpoint. An alternative is to discard packets when the buffer occupancy grows too much. In this case, it is likely that the buffer will become full shortly. Since packet discarding is an information that allows hosts to adapt their transmission rate, discarding packets early could allow hosts to react earlier and thus prevent congestion from happening.

- *Which packet(s) should be discarded ?* Once the network node has decided to discard packets, it needs to actually discard real packets.

By combining different answers to these questions, network researchers have developed different packet discard mechanisms.

- *Tail drop* is the simplest packet discard technique. When a buffer is full, the arriving packet is discarded. *Tail drop* can be easily implemented. This is, by far, the most widely used packet discard mechanism. However, it suffers from two important drawbacks. First, since *tail drop* discards packets only when the buffer is full, buffers tend to be congested and real-time applications may suffer from increased delays. Second, *tail drop* is blind when it discards a packet. It may discard a packet from a low bandwidth interactive flow while most of the buffer is used by large file transfers.
- *Drop from front* is an alternative packet discard technique. Instead of removing the arriving packet, it removes the packet that was at the head of the queue. Discarding this packet instead of the arriving one can have two advantages. First, it already stayed a long time in the buffer. Second, hosts should be able to detect the loss (and thus the congestion) earlier.
- *Probabilistic drop*. Various random drop techniques have been proposed. A frequently cited technique is *Random Early Discard* (RED) [FJ1993]. RED measures the average buffer occupancy and discards packets with a given probability when this average occupancy is too high. Compared to *tail drop* and *drop from front*, an advantage of *RED* is that thanks to the probabilistic drops, packets should be discarded from different flows in proportion of their bandwidth.

Discarding packets is a frequent reaction to network congestion. Unfortunately, discarding packets is not optimal since a packet which is discarded on a network node has already consumed resources on the upstream nodes. There are other ways for the network to inform the end hosts of the current congestion level. A first solution is to mark the packets when a node is congested. Several networking technologies have relied on this kind of packet marking.

In datagram networks, *Forward Explicit Congestion Notification* (FECN) can be used. One field of the packet header, typically one bit, is used to indicate congestion. When a host sends a packet, the congestion bit is unset. If the packet passes through a congested node, the congestion bit is set. The destination can then determine the current congestion level by measuring the fraction of the packets that it received with the congestion bit set. It may then return this information to the sending host to allow it to adapt its retransmission rate. Compared to packet discarding, the main advantage of FECN is that hosts can detect congestion explicitly without having to rely on packet losses.

In virtual circuit networks, packet marking can be improved if the return packets follow the reverse path of the forward packets. It this case, a network node can detect congestion on the forward path (e.g. due to the size of its buffer), but mark the packets on the return path. Marking the return packets (e.g. the acknowledgments used by reliable protocols) provides a faster feedback to the sending hosts compared to FECN. This technique is usually called *Backward Explicit Congestion Notification (BECN)*.

If the packet header does not contain any bit in the header to represent the current congestion level, an alternative is to allow the network nodes to send a control packet to the source to indicate the current congestion level. Some networking technologies use such control packets to explicitly regulate the transmission rate of sources. However, their usage is mainly restricted to small networks. In large networks, network nodes usually avoid using such control packets. These control packets are even considered to be dangerous in some networks. First, using them increases the network load when the network is congested. Second, while network nodes are optimized to forward packets, they are usually pretty slow at creating new packets.

Dropping and marking packets is not the only possible reaction of a router that becomes congested. A router could also selectively delay packets belonging to some flows. There are different algorithms that can be used by a router to delay packets. If the objective of the router is to fairly distribute to bandwidth of an output link among competing flows, one possibility is to organize the buffers of the router as a set of queues. For simplicity, let us assume that the router is capable of supporting a fixed number of concurrent flows, say *N*. One of the queues of the router is associated to each flow and when a packet arrives, it is placed at the tail of the corresponding queue. All the queues are controlled by a *scheduler*. A *scheduler* is an algorithm that is run each time there is an opportunity to transmit a packet on the outgoing link. Various schedulers have been proposed in the scientific literature and some are used in real routers.

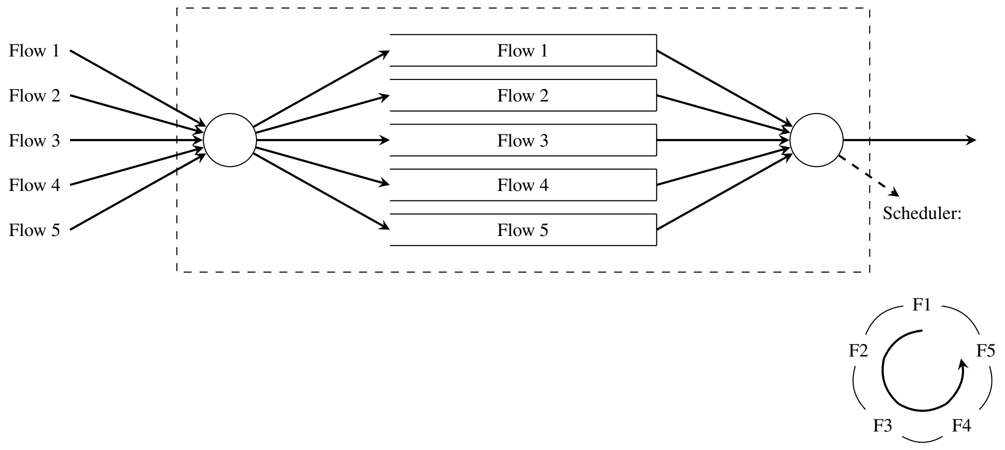


Fig. 63: A round-robin scheduler, where N = 5

A very simple scheduler is the *round-robin scheduler*. This scheduler serves all the queues in a round-robin fashion. If all flows send packets of the same size, then the round-robin scheduler fairly allocates the bandwidth among the different flows. Otherwise, it favors flows that are using larger packets. Extensions to the *round-robin scheduler* have been proposed to provide a fair distribution of the bandwidth with variable-length packets [SV1995] but these are outside the scope of this chapter.

```
# N queues
# state variable : next_queue
next_queue = 0
while True:
    if isEmpty(buffer):
        # Wait for next packet in buffer
        wait()
    if not(isEmpty(queue[next_queue])):
        # Send packet at head of next_queue
        p = remove_packet(queue[next_queue])
        send(p)
    next_queue=(next_queue + 1) % N
# end while
```

2.6.3 Distributing the load across the network

Delays, packet discards, packet markings and control packets are the main types of information that the network can exchange with the end hosts. Discarding packets is the main action that a network node can perform if the congestion is too severe. Besides tackling congestion at each node, it is also possible to divert some traffic flows from heavily loaded links to reduce congestion. Early routing algorithms [MRR1980] have used delay measurements to detect congestion between network nodes and update the link weights dynamically. By reflecting the delay perceived by applications in the link weights used for the shortest paths computation, these routing algorithms managed to dynamically change the forwarding paths in reaction to congestion. However, deployment experience showed that these dynamic routing algorithms could cause oscillations and did not necessarily lower congestion. Deployed datagram networks rarely use dynamic routing algorithms, except in some wireless networks. In datagram networks, the state of the art reaction to long term congestion, i.e. congestion lasting hours, days or more, is to measure the traffic demand and then select the link weights [FRT2002] that allow minimizing the maximum link loads. If the congestion lasts longer, changing the weights is not sufficient anymore and the network needs to be upgraded with additional or faster links. However, in Wide Area Networks, adding new links can take months.

In virtual circuit networks, another way to manage or prevent congestion is to limit the number of circuits that use the network at any time. This technique is usually called *connection admission control*. When a host requests the creation of a new circuit in the network, it specifies the destination and in some networking technologies the required bandwidth. With this information, the network can check whether there are enough resources available to reach this particular destination. If yes, the circuit is established. If not, the request is denied and the host will have to defer the creation of its virtual circuit. *Connection admission control* schemes are widely used in the telephone networks. In these networks, a busy tone corresponds to an unavailable destination or a congested network.

In datagram networks, this technique cannot be easily used since the basic assumption of such a network is that a host can send any packet towards any destination at any time. A host does not need to request the authorization of the network to send packets towards a particular destination.

Based on the feedback received from the network, the hosts can adjust their transmission rate. We discuss in section *Congestion control* some techniques that allow hosts to react to congestion.

Another way to share the network resources is to distribute the load across multiple links. Many techniques have been designed to spread the load over the network. As an illustration, let us briefly consider how the load can be shared when accessing some content. Consider a large and popular file such as the image of a Linux distribution or the upgrade of a commercial operating system that will be downloaded by many users. There are many ways to distribute this large file. A naive solution is to place one copy of the file on a server and allow all users to download this file from the server. If the file is popular and millions of users want to download it, the server will quickly become overloaded. There are two classes of solutions that can be used to serve a large number of users. A first approach is to store the file on servers whose name is known by the clients. Before retrieving the file, each client will query the name service to obtain the address of the server. If the file is available from many servers, the name service can provide different addresses to different clients. This will automatically spread the load since different clients will download the file from different servers. Most large content providers use such a solution to distribute large files or videos.

There is another solution that allows spreading the load among many sources without relying on the name service. The popular bittorent service is an example of this approach. With this solution, each file is divided in blocks of fixed size. To retrieve a file, a client needs to retrieve all the blocks that compose the file. However, nothing forces the client to retrieve all the blocks in sequence and from the same server. Each file is associated with metadata that indicates for each block a list of addresses of hosts that store this block. To retrieve a complete file, a client first downloads the metadata. Then, it tries to retrieve each block from one of the hosts that store the block. In practice, implementations often try to download several blocks in parallel. Once one block has been successfully downloaded, the next block can be requested. If a host is slow to provide one block or becomes unavailable, the client can contact another host listed in the metadata. Most deployments of bittorrent allow the clients to participate to the distribution of blocks. Once a client has downloaded one block, it contacts the server which stores the metadata to indicate that it can also provide this block. With this scheme, when a file is popular, its blocks are downloaded by many hosts that automatically participate in the distribution of the blocks. Thus, the number of *servers* that are capable of providing blocks from a popular file automatically increases with the file's popularity.

Now that we have provided a broad overview of the techniques that can be used to spread the load and allocate resources in the network, let us analyze two techniques in more details : Medium Access Control and Congestion control.

2.6.4 Medium Access Control algorithms

The common problem among Local Area Networks is how to efficiently share the available bandwidth. If two devices send a frame at the same time, the two electrical, optical or radio signals that correspond to these frames will appear at the same time on the transmission medium and a receiver will not be able to decode either frame. Such simultaneous transmissions are called *collisions*. A *collision* may involve frames transmitted by two or more devices attached to the Local Area Network. Collisions are the main cause of errors in wired Local Area Networks.

All Local Area Network technologies rely on a *Medium Access Control* algorithm to regulate the transmissions to either minimize or avoid collisions. There are two broad families of *Medium Access Control* algorithms :

1. *Deterministic* or *pessimistic* MAC algorithms. These algorithms assume that collisions are a very severe problem and that they must be completely avoided. These algorithms ensure that at any time, at most one device is allowed to send a frame on the LAN. This is usually achieved by using a distributed protocol which elects one device that is allowed to transmit at each time. A deterministic MAC algorithm ensures that no collision will happen, but there is some overhead in regulating the transmission of all the devices attached to the LAN.
2. *Stochastic* or *optimistic* MAC algorithms. These algorithms assume that collisions are part of the normal operation of a Local Area Network. They aim to minimize the number of collisions, but they do not try to avoid all collisions. Stochastic algorithms are usually easier to implement than deterministic ones.

We first discuss a simple deterministic MAC algorithm and then we describe several important optimistic algorithms, before coming back to a distributed and deterministic MAC algorithm.

Static allocation methods

A first solution to share the available resources among all the devices attached to one Local Area Network is to define, *a priori*, the distribution of the transmission resources among the different devices. If N devices need to share the transmission capacities of a LAN operating at b Mbps, each device could be allocated a bandwidth of $\frac{b}{N}$ Mbps.

Limited resources need to be shared in other environments than Local Area Networks. Since the first radio transmissions by Marconi more than one century ago, many applications that exchange information through radio signals have been developed. Each radio signal is an electromagnetic wave whose power is centered around a given frequency. The radio spectrum corresponds to frequencies ranging between roughly 3 KHz and 300 GHz. Frequency allocation plans negotiated among governments reserve most frequency ranges for specific applications such as broadcast radio, broadcast television, mobile communications, aeronautical radio navigation, amateur radio, satellite, etc. Each frequency range is then subdivided into channels and each channel can be reserved for a given application, e.g. a radio broadcaster in a given region.

Frequency Division Multiplexing (FDM) is a static allocation scheme in which a frequency is allocated to each device attached to the shared medium. As each device uses a different transmission frequency, collisions cannot occur. In optical networks, a variant of FDM called *Wavelength Division Multiplexing* (WDM) can be used. An optical fiber can transport light at different wavelengths without interference. With WDM, a different wavelength is allocated to each of the devices that share the same optical fiber.

Time Division Multiplexing (TDM) is a static bandwidth allocation method that was initially defined for the telephone network. In the fixed telephone network, a voice conversation is usually transmitted as a 64 Kbps signal. Thus, a telephone conservation generates 8 KBytes per second or one byte every 125 microseconds. Telephone conversations often need to be multiplexed together on a single line. For example, in Europe, thirty 64 Kbps voice signals are multiplexed over a single 2 Mbps (E1) line. This is done by using *Time Division Multiplexing* (TDM). TDM divides the transmission opportunities into slots. In the telephone network, a slot corresponds to 125 microseconds. A position inside each slot is reserved for each voice signal. The figure below illustrates TDM on a link that is used to carry four voice conversations. The vertical lines represent the slot boundaries and the letters the different voice conversations. One byte from each voice conversation is sent during each 125 microseconds slot. The byte corresponding to a given conversation is always sent at the same position in each slot.

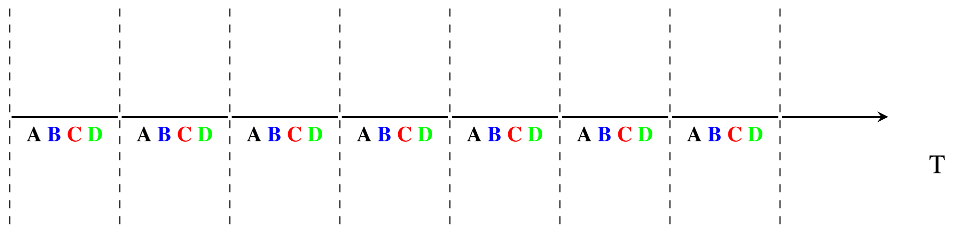


Fig. 64: Time-division multiplexing

TDM as shown above can be completely static, i.e. the same conversations always share the link, or dynamic. In the latter case, the two endpoints of the link must exchange messages specifying which conversation uses which byte inside each slot. Thanks to these control messages, it is possible to dynamically add and remove voice conversations from a given link.

TDM and FDM are widely used in telephone networks to support fixed bandwidth conversations. Using them in Local Area Networks that support computers would probably be inefficient. Computers usually do not send information at a fixed rate. Instead, they often have an on-off behavior. During the on period, the computer tries to send at the highest possible rate, e.g. to transfer a file. During the off period, which is often much longer than the on period, the computer does not transmit any packet. Using a static allocation scheme for computers attached to a LAN would lead to huge inefficiencies, as they would only be able to transmit at $\frac{1}{N}$ of the total bandwidth during their on period, despite the fact that the other computers are in their off period and thus do not need to transmit any information. The dynamic MAC algorithms discussed in the remainder of this chapter aim to solve this problem.

ALOHA

In the 1960s, computers were mainly mainframes with a few dozen terminals attached to them. These terminals were usually in the same building as the mainframe and were directly connected to it. In some cases, the terminals were installed in remote locations and connected through a *modem* attached to a *dial-up line*. The university of Hawaii chose a different organization. Instead of using telephone lines to connect the distant terminals, they developed the first *packet radio* technology [Abramson1970]. Until then, computer networks were built on top of either the telephone network or physical cables. ALOHANet showed that it is possible to use radio signals to interconnect computers.

The first version of ALOHANet, described in [Abramson1970], operated as follows. First, the terminals and the mainframe exchanged fixed-length frames composed of 704 bits. Each frame contained 80 8-bit characters, some control bits and parity information to detect transmission errors. Two channels in the 400 MHz range were reserved for the operation of ALOHANet. The first channel was used by the mainframe to send frames to all terminals. The second channel was shared among all terminals to send frames to the mainframe. As all terminals share the same transmission channel, there is a risk of collision. To deal with this problem as well as transmission errors, the mainframe verified the parity bits of the received frame and sent an acknowledgment on its channel for each correctly received frame. The terminals on the other hand had to retransmit the unacknowledged frames. As for TCP, retransmitting these frames immediately upon expiration of a fixed timeout is not a good approach as several terminals may retransmit their frames at the same time leading to a network collapse. A better approach, but still far from perfect, is for each terminal to wait a random amount of time after the expiration of its retransmission timeout. This avoids synchronization among multiple retransmitting terminals.

The pseudo-code below shows the operation of an ALOHANet terminal. We use this python syntax for all Medium Access Control algorithms described in this chapter. The algorithm is applied to each new frame that needs to be transmitted. It attempts to transmit a frame at most *max* times (*while loop*). Each transmission attempt is performed as follows. First, the frame is sent. Each frame is protected by a timeout. Then, the terminal waits for either a valid acknowledgment frame or the expiration of its timeout. If the terminal receives an acknowledgment, the frame has been delivered correctly and the algorithm terminates. Otherwise, the terminal waits for a random time and attempts to retransmit the frame.

```
# ALOHA
N = 1
while N <= max:
    send(frame)
    wait(ack_on_return_channel or timeout)
    if (ack_on_return_channel):
       break  # transmission was successful
    else:
             # timeout
             wait(random_time)
             N = N + 1
```

(continues on next page)

(continued from previous page)

```
else:
    # Too many transmission attempts
```

[Abramson1970] analyzed the performance of ALOHANet under particular assumptions and found that ALOHANet worked well when the channel was lightly loaded. In this case, the frames are rarely retransmitted and the *channel traffic*, i.e. the total number of (correct and retransmitted) frames transmitted per unit of time is close to the *channel utilization*, i.e. the number of correctly transmitted frames per unit of time. Unfortunately, the analysis also reveals that the *channel utilization* reaches its maximum at $\frac{1}{2 \times e} = 0.186$ times the channel bandwidth. At higher utilization, ALOHANet becomes unstable and the network collapses due to collided retransmissions.

Note: Amateur packet radio

Packet radio technologies have evolved in various directions since the first experiments performed at the University of Hawaii. The Amateur packet radio service developed by amateur radio operators is one of the descendants ALOHANet. Many amateur radio operators are very interested in new technologies and they often spend countless hours developing new antennas or transceivers. When the first personal computers appeared, several amateur radio operators designed radio modems and their own datalink layer protocols [KPD1985] [BNT1997]. This network grew and it was possible to connect to servers in several European countries by only using packet radio relays. Some amateur radio operators also developed TCP/IP protocol stacks that were used over the packet radio service. Some parts of the amateur packet radio network are connected to the global Internet and use the *44.0.0.0/8* IPv4 prefix.

Many improvements to ALOHANet have been proposed since the publication of [Abramson1970], and this technique, or some of its variants, are still found in wireless networks today. The slotted technique proposed in [Roberts1975] is important because it shows that a simple modification can significantly improve channel utilization. Instead of allowing all terminals to transmit at any time, [Roberts1975] proposed to divide time into slots and allow terminals to transmit only at the beginning of each slot. Each slot corresponds to the time required to transmit one fixed size frame. In practice, these slots can be imposed by a single clock that is received by all terminals. In ALOHANet, it could have been located on the central mainframe. The analysis in [Roberts1975] reveals that this simple modification improves the channel utilization by a factor of two.

Carrier Sense Multiple Access

ALOHA and slotted ALOHA can easily be implemented, but unfortunately, they can only be used in networks that are very lightly loaded. Designing a network for a very low utilization is possible, but it clearly increases the cost of the network. To overcome the problems of ALOHA, many Medium Access Control mechanisms have been proposed which improve channel utilization. Carrier Sense Multiple Access (CSMA) is a significant improvement compared to ALOHA. CSMA requires all nodes to listen to the transmission channel to verify that it is free before transmitting a frame [KT1975]. When a node senses the channel to be busy, it defers its transmission until the channel becomes free again. The pseudo-code below provides a more detailed description of the operation of CSMA.

```
# persistent CSMA
N = 1
while N <= max:
    wait(channel_becomes_free)
    send(frame)
    wait(ack or timeout)
    if ack:
       break  # transmission was successful
    else:
             # timeout
             N = N + 1
```

(continues on next page)

(continued from previous page)

```
else:
    # Too many transmission attempts
```

The above pseudo-code is often called *persistent CSMA* [KT1975] as the terminal will continuously listen to the channel and transmit its frame as soon as the channel becomes free. Another important variant of CSMA is the *non-persistent CSMA* [KT1975]. The main difference between persistent and non-persistent CSMA described in the pseudo-code below is that a non-persistent CSMA node does not continuously listen to the channel to determine when it becomes free. When a non-persistent CSMA terminal senses the transmission channel to be busy, it waits for a random time before sensing the channel again. This improves channel utilization compared to persistent CSMA. With persistent CSMA, when two terminals sense the channel to be busy, they will both transmit (and thus cause a collision) as soon as the channel becomes free. With non-persistent CSMA, this synchronization does not occur, as the terminals wait a random time after having sensed the transmission channel. However, the higher channel utilization achieved by non-persistent CSMA comes at the expense of a slightly higher waiting time in the terminals when the network is lightly loaded.

```
# Non persistent CSMA
N = 1
while N <= max:
    listen(channel)
    if free(channel):
        send(frame)
        wait(ack or timeout)
        if received(ack):
           break  # transmission was successful
        else:
                  # timeout
                  N = N + 1
    else:
        wait(random_time)
else:
    # Too many transmission attempts
```

[KT1975] analyzes in detail the performance of several CSMA variants. Under some assumptions about the transmission channel and the traffic, the analysis compares ALOHA, slotted ALOHA, persistent and non-persistent CSMA. Under these assumptions, ALOHA achieves a channel utilization of only 18.4% of the channel capacity. Slotted ALOHA is able to use 36.6% of this capacity. Persistent CSMA improves the utilization by reaching 52.9% of the capacity while non-persistent CSMA achieves 81.5% of the channel capacity.

Carrier Sense Multiple Access with Collision Detection

CSMA improves channel utilization compared to ALOHA. However, the performance can still be improved, especially in wired networks. Consider the situation of two terminals that are connected to the same cable. This cable could, for example, be a coaxial cable as in the early days of Ethernet [Metcalfe1976]. It could also be built with twisted pairs. Before extending CSMA, it is useful to understand, more intuitively, how frames are transmitted in such a network and how collisions can occur. The figure below illustrates the physical transmission of a frame on such a cable. To transmit its frame, host A must send an electrical signal on the shared medium. The first step is thus to begin the transmission of the electrical signal. This is point *(1)* in the figure below. This electrical signal will travel along the cable. Although electrical signals travel fast, we know that information cannot travel faster than the speed of light (i.e. 300.000 kilometers/second). On a coaxial cable, an electrical signal is slightly slower than the speed of light and 200.000 kilometers per second is a reasonable estimation. This implies that if the cable has a length of one kilometer, the electrical signal will need 5 microseconds to travel from one end of the cable to the other. The ends of coaxial cables are equipped with termination points that ensure that the electrical signal is not reflected back to its source. This is illustrated at point *(3)* in the figure, where the electrical signal has reached the left endpoint and host B. At this point, B starts to receive the frame being transmitted by A. Notice that there is a delay between the transmission of a bit on host A and its reception by host B. If there were other hosts attached to the cable, they would receive the first bit of the frame at slightly different times. As we will see later, this timing difference is a key problem for MAC algorithms. At point *(4)*, the electrical signal has reached both ends of the cable and occupies it completely. Host A continues to transmit the electrical signal until the end of the frame. As shown at point *(5)*, when the sending host stops its transmission, the electrical signal corresponding to the end of the frame leaves the coaxial cable. The channel becomes empty again once the entire electrical signal has been removed from the cable.

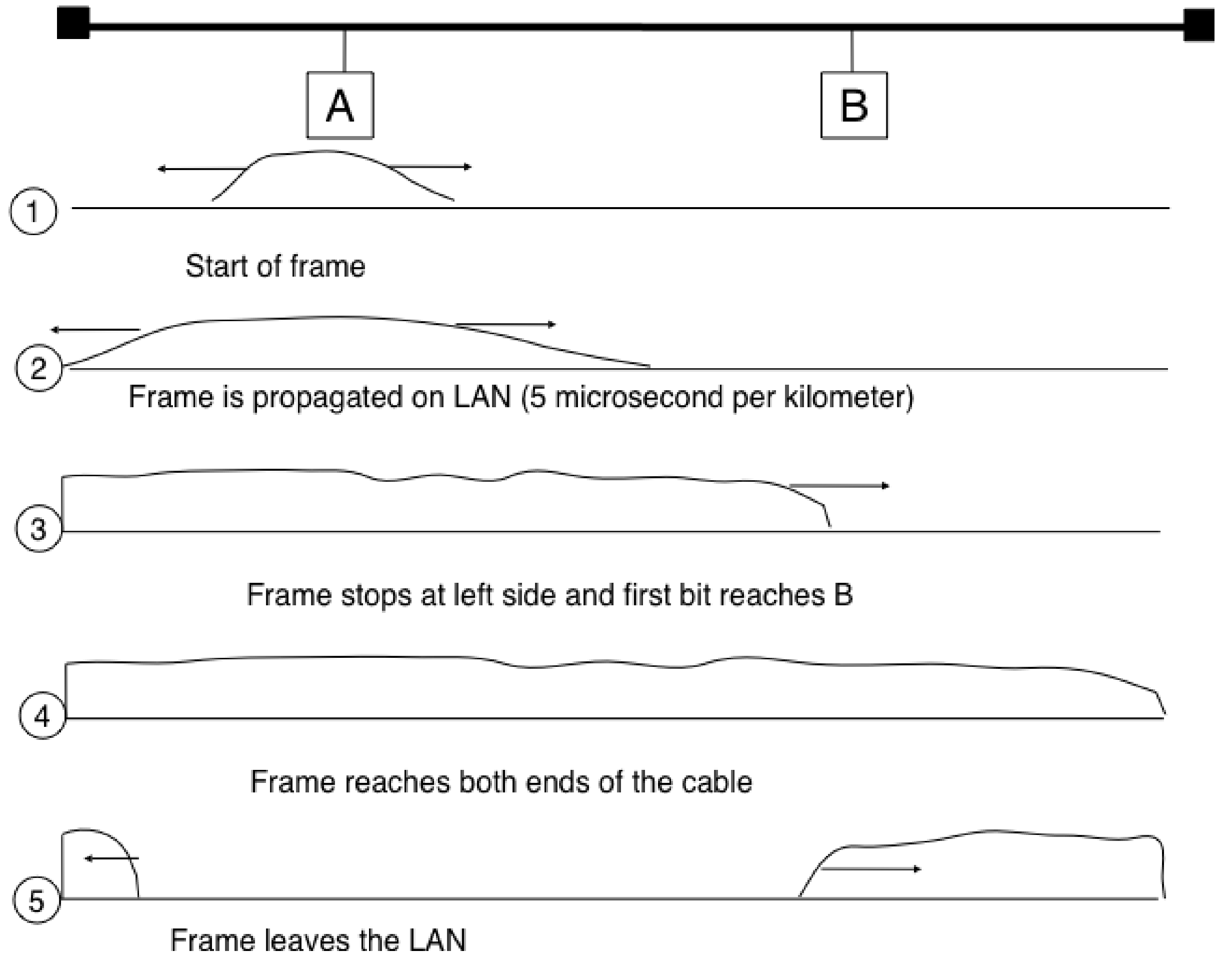

A
B
1
Start of frame
2
Frame is propagated on LAN (5 microsecond per kilometer)
3
Frame stops at left side and first bit reaches B
4
Frame reaches both ends of the cable
5
Frame leaves the LAN

Fig. 65: Frame transmission on a shared bus

Now that we have looked at how a frame is actually transmitted as an electrical signal on a shared bus, it is interesting to look in more detail at what happens when two hosts transmit a frame at almost the same time. This is illustrated in the figure below, where hosts A and B start their transmission at the same time (point *(1)*). At this time, if host C senses the channel, it will consider it to be free. This will not last a long time and at point *(2)* the electrical signals from both host A and host B reach host C. The combined electrical signal (shown graphically as the superposition of the two curves in the figure) cannot be decoded by host C. Host C detects a collision, as it receives a signal that it cannot decode. Since host C cannot decode the frames, it cannot determine which hosts are sending the colliding frames. Note that host A (and host B) will detect the collision after host C (point *(3)* in the figure below).

As shown above, hosts detect collisions when they receive an electrical signal that they cannot decode. In a wired network, a host is able to detect such a collision both while it is listening (e.g. like host C in the figure above) and also while it is sending its own frame. When a host transmits a frame, it can compare the electrical signal that it transmits with the electrical signal that it senses on the wire. At points *(1)* and *(2)* in the figure above, host A senses only its own signal. At point *(3)*, it senses an electrical signal that differs from its own signal and can thus detects the collision. At this point, its frame is corrupted and it can stop its transmission. The ability to detect collisions while transmitting

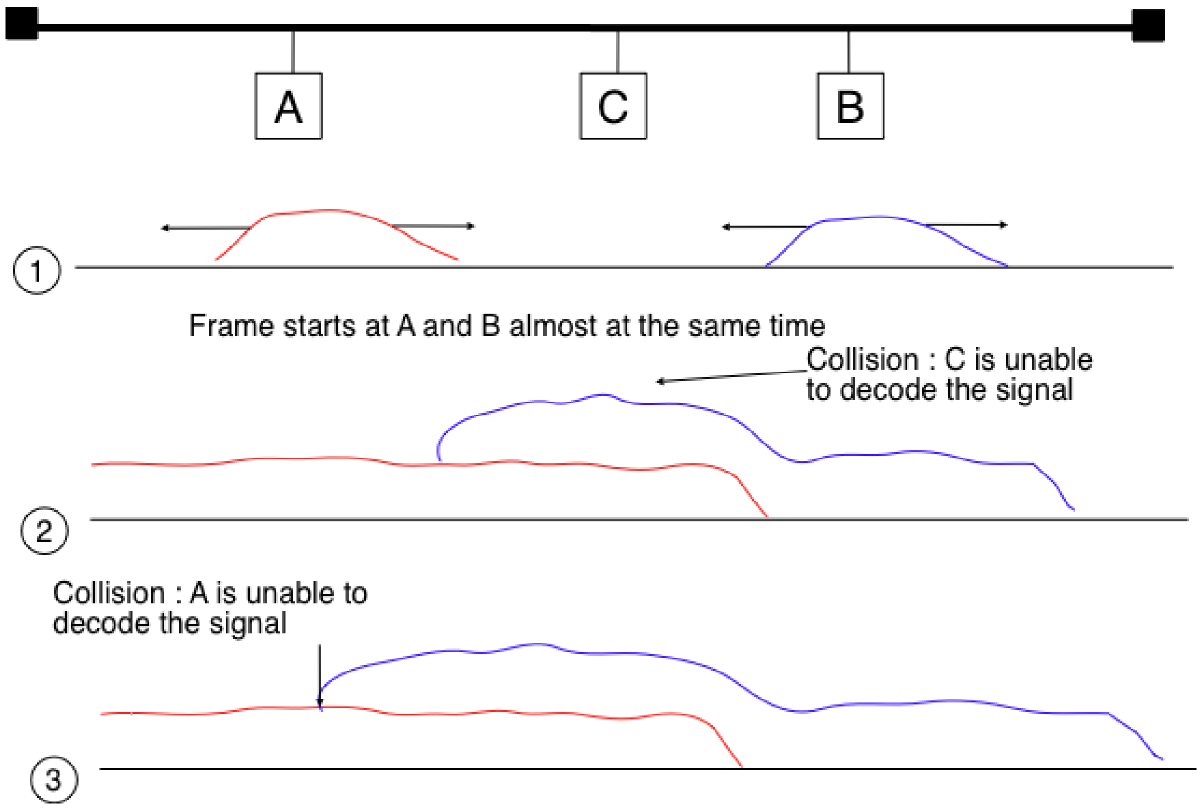

A
C
B
1
Frame starts at A and B almost at the same time
Collision : C is unable to decode the signal
2
Collision : A is unable to decode the signal
3

Fig. 66: Frame collision on a shared bus

is the starting point for the *Carrier Sense Multiple Access with Collision Detection (CSMA/CD)* Medium Access Control algorithm, which is used in Ethernet networks [Metcalfe1976] [IEEE802.3] . When an Ethernet host detects a collision while it is transmitting, it immediately stops its transmission. Compared with pure CSMA, CSMA/CD is an important improvement since when collisions occur, they only last until colliding hosts have detected it and stopped their transmission. In practice, when a host detects a collision, it sends a special jamming signal on the cable to ensure that all hosts have detected the collision.

To better understand these collisions, it is useful to analyze what would be the worst collision on a shared bus network. Let us consider a wire with two hosts attached at both ends, as shown in the figure below. Host A starts to transmit its frame and its electrical signal is propagated on the cable. Its propagation time depends on the physical length of the cable and the speed of the electrical signal. Let us use τ to represent this propagation delay in seconds. Slightly less than τ seconds after the beginning of the transmission of A's frame, B decides to start transmitting its own frame. After ϵ seconds, B senses A's frame, detects the collision and stops transmitting. The beginning of B's frame travels on the cable until it reaches host A. Host A can thus detect the collision at time $\tau - \epsilon + \tau \approx 2 \times \tau$. An important point to note is that a collision can only occur during the first $2 \times \tau$ seconds of its transmission. If a collision did not occur during this period, it cannot occur afterwards since the transmission channel is busy after τ seconds and CSMA/CD hosts sense the transmission channel before transmitting their frame.

Furthermore, on the wired networks where CSMA/CD is used, collisions are almost the only cause of transmission errors that affect frames. Transmission errors that only affect a few bits inside a frame seldom occur in these wired networks. For this reason, the designers of CSMA/CD chose to completely remove the acknowledgment frames in the datalink layer. When a host transmits a frame, it verifies whether its transmission has been affected by a collision. If not, given the negligible Bit Error Ratio of the underlying network, it assumes that the frame was received correctly by its destination. Otherwise the frame is retransmitted after some delay.

Removing acknowledgments is an interesting optimization as it reduces the number of frames that are exchanged on the network and the number of frames that need to be processed by the hosts. However, to use this optimization, we

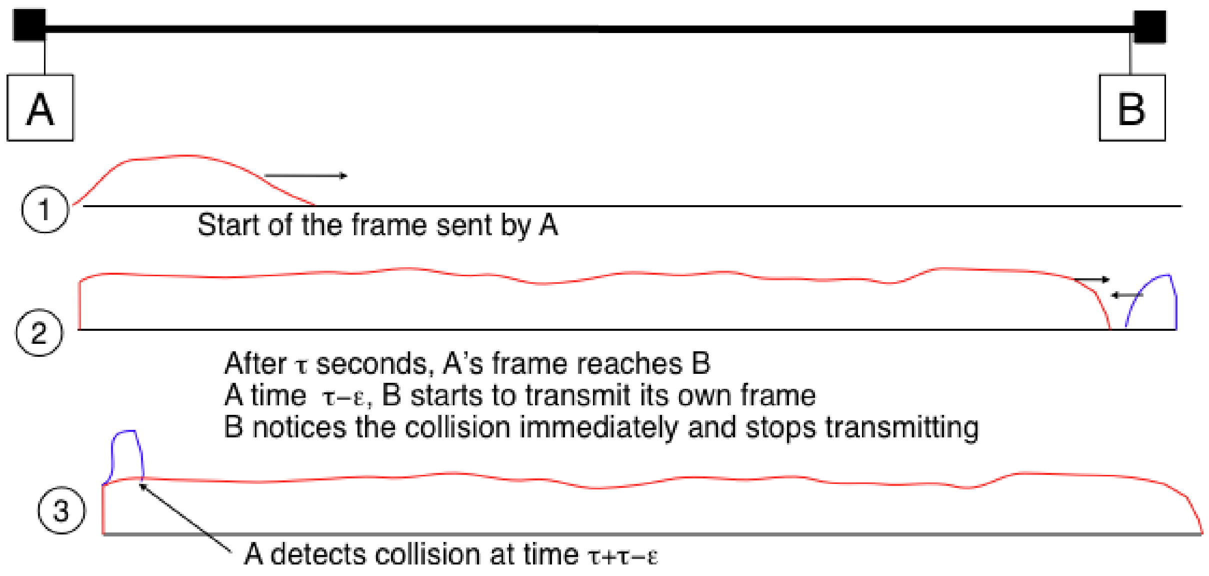

A
B
1
Start of the frame sent by A
2
After τ seconds, A's frame reaches B
A time τ−ε, B starts to transmit its own frame
B notices the collision immediately and stops transmitting
3
A detects collision at time τ+τ−ε

Fig. 67: The worst collision on a shared bus

must ensure that all hosts will be able to detect all the collisions that affect their frames. The problem is important for short frames. Let us consider two hosts, A and B, that are sending a small frame to host C as illustrated in the figure below. If the frames sent by A and B are very short, the situation illustrated below may occur. Hosts A and B send their frame and stop transmitting (point *(1)*). When the two short frames arrive at the location of host C, they collide and host C cannot decode them (point *(2)*). The two frames are absorbed by the ends of the wire. Neither host A nor host B have detected the collision. They both consider their frame to have been received correctly by its destination.

To solve this problem, networks using CSMA/CD require hosts to transmit for at least $2 \times \tau$ seconds. Since the network transmission speed is fixed for a given network technology, this implies that a technology that uses CSMA/CD enforces a minimum frame size. In the most popular CSMA/CD technology, Ethernet, $2 \times \tau$ is called the *slot time*[4].

The last innovation introduced by CSMA/CD is the computation of the retransmission timeout. As for ALOHA, this timeout cannot be fixed, otherwise hosts could become synchronized and always retransmit at the same time. Setting such a timeout is always a compromise between the network access delay and the amount of collisions. A short timeout would lead to a low network access delay but with a higher risk of collisions. On the other hand, a long timeout would cause a long network access delay but a lower risk of collisions. The *binary exponential back-off* algorithm was introduced in CSMA/CD networks to solve this problem.

To understand *binary exponential back-off*, let us consider a collision caused by exactly two hosts. Once it has detected the collision, a host can either retransmit its frame immediately or defer its transmission for some time. If each colliding host flips a coin to decide whether to retransmit immediately or to defer its retransmission, four cases are possible :

1. Both hosts retransmit immediately and a new collision occurs
2. The first host retransmits immediately and the second defers its retransmission
3. The second host retransmits immediately and the first defers its retransmission
4. Both hosts defer their retransmission and a new collision occurs

In the second and third cases, both hosts have flipped different coins. The delay chosen by the host that defers its retransmission should be long enough to ensure that its retransmission will not collide with the immediate retransmission of the other host. However the delay should not be longer than the time necessary to avoid the collision, because

[4] This name should not be confused with the duration of a transmission slot in slotted ALOHA. In CSMA/CD networks, the slot time is the time during which a collision can occur at the beginning of the transmission of a frame. In slotted ALOHA, the duration of a slot is the transmission time of an entire fixed-size frame.

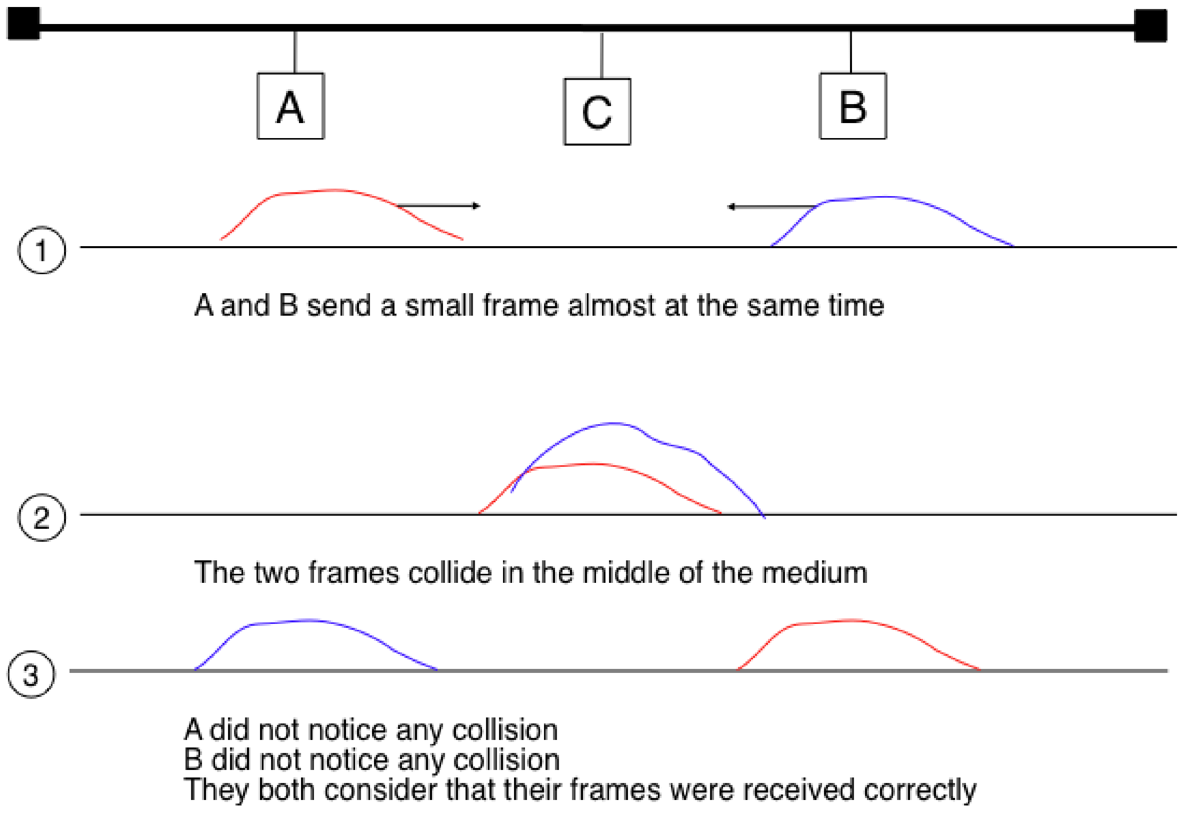

A
C
B
1
A and B send a small frame almost at the same time
2
The two frames collide in the middle of the medium
3
A did not notice any collision
B did not notice any collision
They both consider that their frames were received correctly

Fig. 68: The short-frame collision problem

if both hosts decide to defer their transmission, the network will be idle during this delay. The *slot time* is the optimal delay since it is the shortest delay that ensures that the first host will be able to retransmit its frame completely without any collision.

If two hosts are competing, the algorithm above will avoid a second collision 50% of the time. However, if the network is heavily loaded, several hosts may be competing at the same time. In this case, the hosts should be able to automatically adapt their retransmission delay. The *binary exponential back-off* performs this adaptation based on the number of collisions that have affected a frame. After the first collision, the host flips a coin and waits 0 or 1 *slot time*. After the second collision, it generates a random number and waits 0, 1, 2 or 3 *slot times*, etc. The duration of the waiting time is doubled after each collision. The complete pseudo-code for the CSMA/CD algorithm is shown in the figure below.

```
# CSMA/CD pseudo-code
N = 1
while N <= max:
    wait(channel_becomes_free)
    send(frame)
    wait_until (end_of_frame) or (collision)
    if collision detected:
        stop_transmitting()
        send(jamming)
        k = min(10, N)
        r = random(0, 2**k - 1)
        wait(r * slotTime)
        N = N + 1
    else:
```

(continues on next page)

(continued from previous page)

```
        wait(inter-frame_delay)
        break  # transmission was successful
else:
    # Too many transmission attempts
```

The inter-frame delay used in this pseudo-code is a short delay corresponding to the time required by a network adapter to switch from transmit to receive mode. It is also used to prevent a host from sending a continuous stream of frames without leaving any transmission opportunities for other hosts on the network. This contributes to the fairness of CSMA/CD. Despite this delay, there are still conditions where CSMA/CD is not completely fair [RY1994]. Consider for example a network with two hosts : a server sending long frames and a client sending acknowledgments. Measurements reported in [RY1994] have shown that there are situations where the client could suffer from repeated collisions that lead it to wait for long periods of time due to the exponential back-off algorithm.

Carrier Sense Multiple Access with Collision Avoidance

The *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA) Medium Access Control algorithm was designed for the popular WiFi wireless network technology [IEEE802.11]. CSMA/CA also senses the transmission channel before transmitting a frame. Furthermore, CSMA/CA tries to avoid collisions by carefully tuning the timers used by CSMA/CA devices.

CSMA/CA uses acknowledgments like CSMA. Each frame contains a sequence number and a CRC. The CRC is used to detect transmission errors while the sequence number is used to avoid frame duplication. When a device receives a correct frame, it returns a special acknowledgment frame to the sender. CSMA/CA introduces a small delay, named *Short Inter Frame Spacing* (SIFS), between the reception of a frame and the transmission of the acknowledgment frame. This delay corresponds to the time that is required to switch the radio of a device between the reception and transmission modes.

Compared to CSMA, CSMA/CA defines more precisely when a device is allowed to send a frame. First, CSMA/CA defines two delays : *DIFS* and *EIFS*. To send a frame, a device must first wait until the channel has been idle for at least the *Distributed Coordination Function Inter Frame Space* (DIFS) if the previous frame was received correctly. However, if the previously received frame was corrupted, this indicates that there are collisions and the device must sense the channel idle for at least the *Extended Inter Frame Space* (EIFS), with $SIFS < DIFS < EIFS$. The exact values for SIFS, DIFS and EIFS depend on the underlying physical layer [IEEE802.11].

The figure below shows the basic operation of CSMA/CA devices. Before transmitting, host *A* verifies that the channel is empty for a long enough period. Then, its sends its data frame. After checking the validity of the received frame, the recipient sends an acknowledgment frame after a short SIFS delay. Host *C*, which does not participate in the frame exchange, senses the channel to be busy at the beginning of the data frame. Host *C* can use this information to determine how long the channel will be busy for. Note that as $SIFS < DIFS < EIFS$, even a device that would start to sense the channel immediately after the last bit of the data frame could not decide to transmit its own frame during the transmission of the acknowledgment frame.

The main difficulty with CSMA/CA is when two or more devices transmit at the same time and cause collisions. This is illustrated in the figure below, assuming a fixed timeout after the transmission of a data frame. With CSMA/CA, the timeout after the transmission of a data frame is very small, since it corresponds to the SIFS plus the time required to transmit the acknowledgment frame.

To deal with this problem, CSMA/CA relies on a backoff timer. This backoff timer is a random delay that is chosen by each device in a range that depends on the number of retransmissions for the current frame. The range grows exponentially with the retransmissions as in CSMA/CD. The minimum range for the backoff timer is $[0, 7 * slotTime]$ where the *slotTime* is a parameter that depends on the underlying physical layer. Compared to CSMA/CD's exponential backoff, there are two important differences to notice. First, the initial range for the backoff timer is seven times larger. This is because it is impossible in CSMA/CA to detect collisions as they happen. With CSMA/CA, a collision may affect the entire frame while with CSMA/CD it can only affect the beginning of the frame. Second, a CSMA/CA device must regularly sense the transmission channel during its back off timer. If the channel becomes busy (i.e.

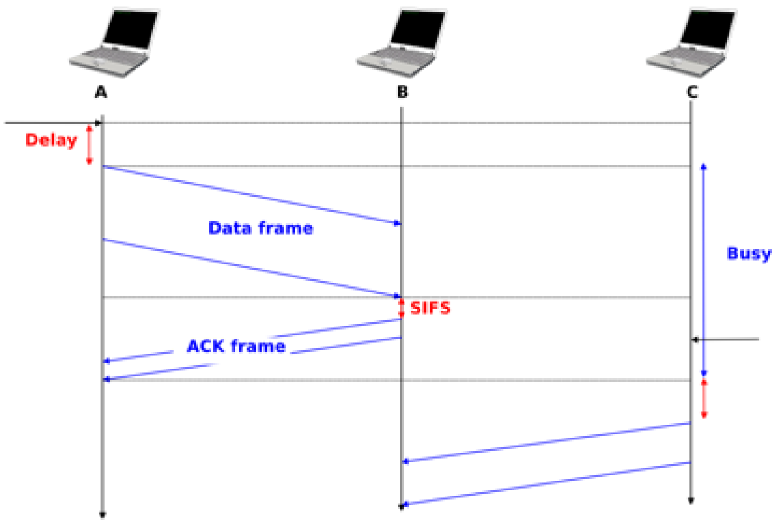


Fig. 69: Operation of a CSMA/CA device

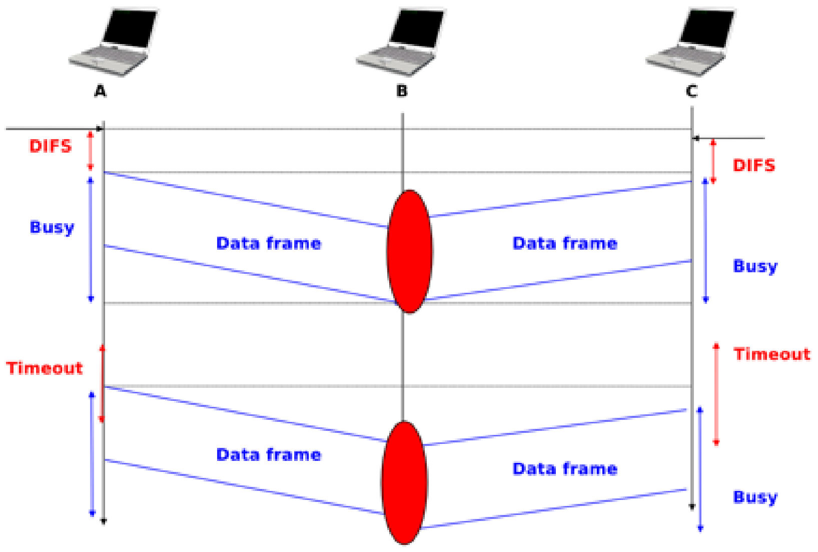


Fig. 70: Collisions with CSMA/CA

because another device is transmitting), then the back off timer must be frozen until the channel becomes free again. Once the channel becomes free, the back off timer is restarted. This is in contrast with CSMA/CD where the back off is recomputed after each collision. This is illustrated in the figure below. Host *A* chooses a smaller backoff than host *C*. When *C* senses the channel to be busy, it freezes its backoff timer and only restarts it once the channel is free again.

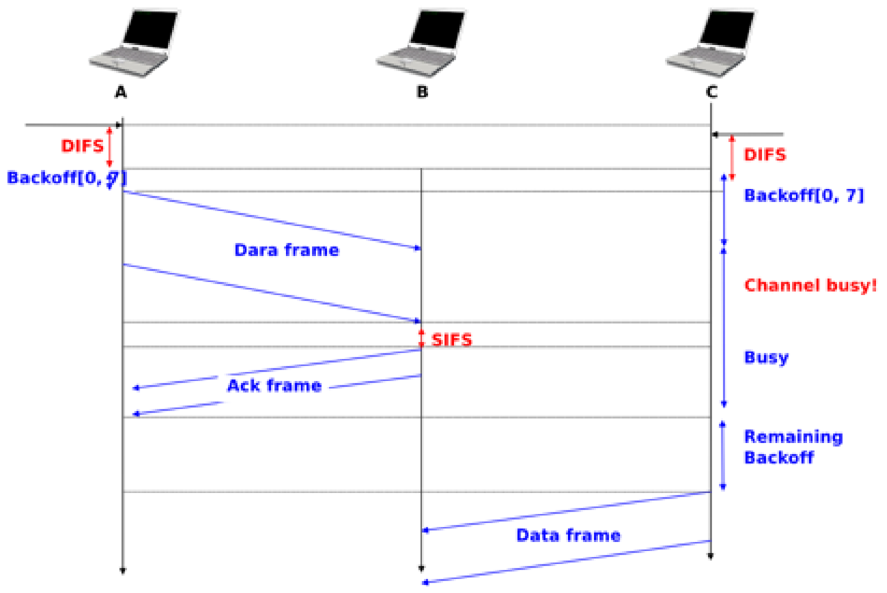


Fig. 71: Detailed example with CSMA/CA

The pseudo-code below summarizes the operation of a CSMA/CA device. The values of the SIFS, DIFS, EIFS and $slotTime$ depend on the underlying physical layer technology [IEEE802.11]

```
# CSMA/CA simplified pseudo-code
N=1
while N <= max:
    wait_until(free(channel))
    if correct(last_frame):
        wait(channel_free_during_t >= DIFS)
    else:
        wait(channel_free_during_t >= EIFS)

    backoff_time = int(random(0, min(255, 7 * ( 2 ** (N - 1))))) * slotTime
    wait(channel free during backoff_time)
    # backoff timer is frozen while channel is sensed to be busy
    send(frame)
    wait(ack or timeout)
    if received(ack)
        # frame received correctly
        break
    else:
        # retransmission required
        N = N + 1
else:
    # Too many transmission attempts
```

Another problem faced by wireless networks is often called the *hidden station problem*. In a wireless network, radio signals are not always propagated same way in all directions. For example, two devices separated by a wall may not be able to receive each other's signal while they could both be receiving the signal produced by a third host. This is illustrated in the figure below, but it can happen in other environments. For example, two devices that are on different sides of a hill may not be able to receive each other's signal while they are both able to receive the signal sent by a station at the top of the hill. Furthermore, the radio propagation conditions may change with time. For example, a truck may temporarily block the communication between two nearby devices.

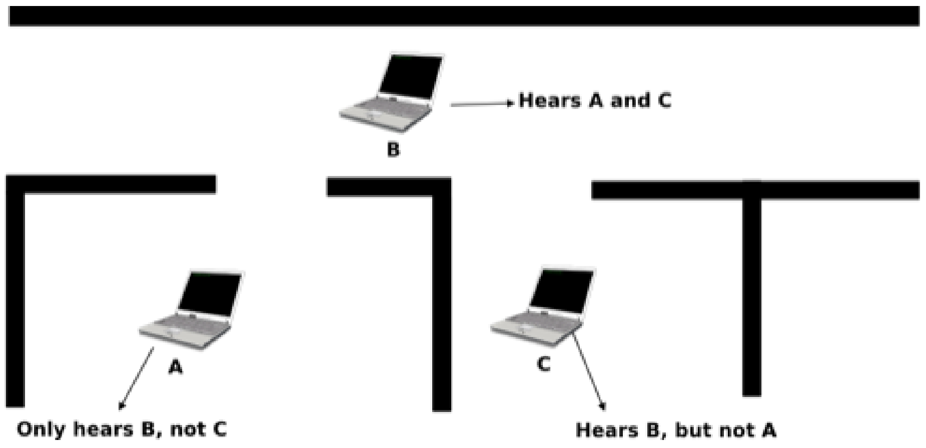


Fig. 72: The hidden station problem

To avoid collisions in these situations, CSMA/CA allows devices to reserve the transmission channel for some time. This is done by using two control frames : *Request To Send* (RTS) and *Clear To Send* (CTS). Both are very short frames to minimize the risk of collisions. To reserve the transmission channel, a device sends a RTS frame to the intended recipient of the data frame. The RTS frame contains the duration of the requested reservation. The recipient replies, after a SIFS delay, with a CTS frame which also contains the duration of the reservation. As the duration of the reservation has been sent in both RTS and CTS, all hosts that could collide with either the sender or the reception of the data frame are informed of the reservation. They can compute the total duration of the transmission and defer their access to the transmission channel until then. This is illustrated in the figure below where host *A* reserves the transmission channel to send a data frame to host *B*. Host *C* notices the reservation and defers its transmission.

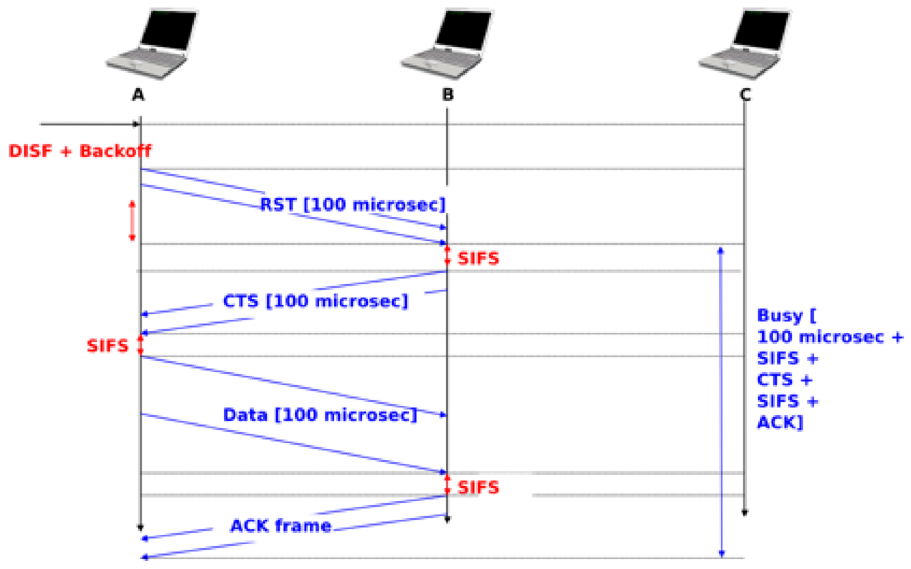


Fig. 73: Reservations with CSMA/CA

The utilization of the reservations with CSMA/CA is an optimization that is useful when collisions are frequent. If there are few collisions, the time required to transmit the RTS and CTS frames can become significant and in particular when short frames are exchanged. Some devices only turn on RTS/CTS after transmission errors.

Deterministic Medium Access Control algorithms

During the 1970s and 1980s, there were huge debates in the networking community about the best suited Medium Access Control algorithms for Local Area Networks. The optimistic algorithms that we have described until now were relatively easy to implement when they were designed. From a performance perspective, mathematical models and simulations showed the ability of these optimistic techniques to sustain load. However, none of the optimistic techniques are able to guarantee that a frame will be delivered within a given delay bound and some applications require predictable transmission delays. The deterministic MAC algorithms were considered by a fraction of the networking community as the best solution to fulfill the needs of Local Area Networks.

Both the proponents of the deterministic and the opportunistic techniques lobbied to develop standards for Local Area networks that would incorporate their solution. Instead of trying to find an impossible compromise between these diverging views, the IEEE 802 committee that was chartered to develop Local Area Network standards chose to work in parallel on three different LAN technologies and created three working groups. The IEEE 802.3 working group became responsible for CSMA/CD. The proponents of deterministic MAC algorithms agreed on the basic principle of exchanging special frames called tokens between devices to regulate the access to the transmission medium. However, they did not agree on the most suitable physical layout for the network. IBM argued in favor of Ring-shaped networks while the manufacturing industry, led by General Motors, argued in favor of a bus-shaped network. This led to the creation of the IEEE 802.4 working group to standardize Token Bus networks and the IEEE 802.5 working group to standardize Token Ring networks. Although these techniques are not widely used anymore today, the principles behind a token-based protocol are still important.

The IEEE 802.5 Token Ring technology is defined in [IEEE802.5]. We use Token Ring as an example to explain the principles of the token-based MAC algorithms in ring-shaped networks. Other ring-shaped networks include the defunct FDDI [Ross1989] or Resilient Pack Ring [DYGU2004] . A good survey of the early token ring networks may be found in [Bux1989] .

A Token Ring network is composed of a set of stations that are attached to a unidirectional ring. The basic principle of the Token Ring MAC algorithm is that two types of frames travel on the ring : tokens and data frames. When the Token Ring starts, one of the stations sends the token. The token is a small frame that represents the authorization to transmit data frames on the ring. To transmit a data frame on the ring, a station must first capture the token by removing it from the ring. As only one station can capture the token at a time, the station that owns the token can safely transmit a data frame on the ring without risking collisions. After having transmitted its frame, the station must remove it from the ring and resend the token so that other stations can transmit their own frames.

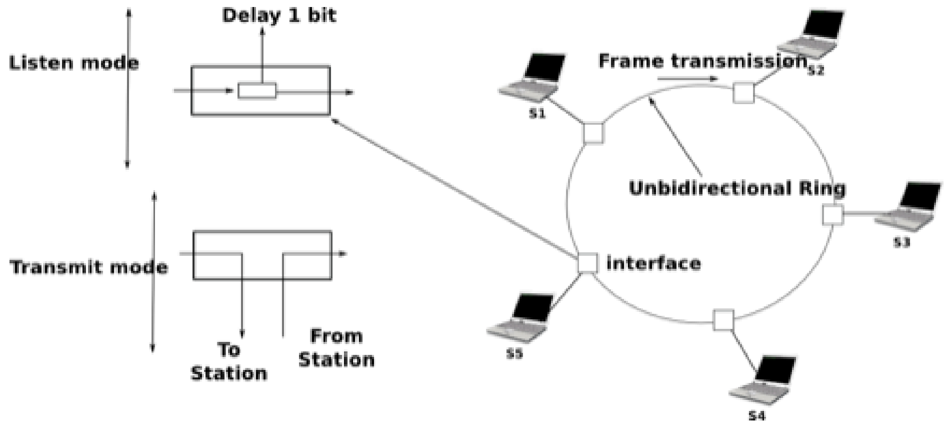


Fig. 74: A Token Ring network

While the basic principles of the Token Ring are simple, there are several subtle implementation details that add complexity to Token Ring networks. To understand these details let us analyze the operation of a Token Ring interface on a station. A Token Ring interface serves three different purposes. Like other LAN interfaces, it must be able to send and receive frames. In addition, a Token Ring interface is part of the ring, and as such, it must be able to forward the electrical signal that passes on the ring even when its station is powered off.

When powered-on, Token Ring interfaces operate in two different modes : *listen* and *transmit*. When operating in *listen* mode, a Token Ring interface receives an electrical signal from its upstream neighbor on the ring, introduces a delay equal to the transmission time of one bit on the ring and regenerates the signal before sending it to its downstream neighbor on the ring.

The first problem faced by a Token Ring network is that as the token represents the authorization to transmit, it must continuously travel on the ring when no data frame is being transmitted. Let us assume that a token has been produced and sent on the ring by one station. In Token Ring networks, the token is a 24 bits frame whose structure is shown below.

```
0                   1                   2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Start Delim.  |Access Control | Ending Delim. |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

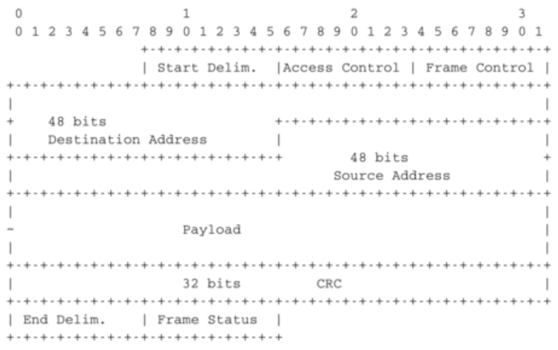
Fig. 75: 802.5 token format

The token is composed of three fields. First, the *Starting Delimiter* is the marker that indicates the beginning of a frame. The first Token Ring networks used Manchester coding and the *Starting Delimiter* contained both symbols representing *0* and symbols that do not represent bits. The last field is the *Ending Delimiter* which marks the end of the token. The *Access Control* field is present in all frames, and contains several flags. The most important is the *Token* bit that is set in token frames and reset in other frames.

Let us consider the five station network depicted in figure *A Token Ring network* above and assume that station *S1* sends a token. If we neglect the propagation delay on the inter-station links, as each station introduces a one bit delay, the first bit of the frame would return to *S1* while it sends the fifth bit of the token. If station *S1* is powered off at that time, only the first five bits of the token will travel on the ring. To avoid this problem, there is a special station called the *Monitor* on each Token Ring. To ensure that the token can travel forever on the ring, this *Monitor* inserts a delay that is equal to at least 24 bit transmission times. If station *S3* was the *Monitor* in figure *A Token Ring network*, *S1* would have been able to transmit the entire token before receiving the first bit of the token from its upstream neighbor.

Now that we have explained how the token can be forwarded on the ring, let us analyze how a station can capture a token to transmit a data frame. For this, we need some information about the format of the data frames. An 802.5 data frame begins with the *Starting Delimiter* followed by the *Access Control* field whose *Token* bit is reset, a *Frame Control* field that enables the definition of several types of frames, destination and source address, a payload, a CRC, the *Ending Delimiter* and a *Frame Status* field. The format of the Token Ring data frames is illustrated below.

To capture a token, a station must operate in *Listen* mode. In this mode, the station receives bits from its upstream neighbor. If the bits correspond to a data frame, they must be forwarded to the downstream neighbor. If they correspond to a token, the station can capture it and transmit its data frame. Both the data frame and the token are encoded as a bit string beginning with the *Starting Delimiter* followed by the *Access Control* field. When the station receives the first bit of a *Starting Delimiter*, it cannot know whether this is a data frame or a token and must forward the entire delimiter to its downstream neighbor. It is only when it receives the fourth bit of the *Access Control* field (i.e. the *Token* bit) that the station knows whether the frame is a data frame or a token. If the *Token* bit is reset, it indicates a data frame and the remaining bits of the data frame must be forwarded to the downstream station. Otherwise (*Token* bit is set), this is a token and the station can capture it by resetting the bit that is currently in its buffer. Thanks to this modification, the beginning of the token is now the beginning of a data frame and the station can switch to *Transmit* mode and send its data frame starting at the fifth bit of the *Access Control* field. Thus, the one-bit delay introduced by each Token Ring station plays a key role in enabling the stations to efficiently capture the token.



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
                +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                | Start Delim.  |Access Control | Frame Control |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+     48 bits                   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Destination Address       |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+        48 bits                +
|                                   Source Address              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                        Payload                                |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        32 bits          CRC                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| End Delim.    | Frame Status  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 76: 802.5 data frame format

After having transmitted its data frame, the station must remain in *Transmit* mode until it has received the last bit of its own data frame. This ensures that the bits sent by a station do not remain in the network forever. A data frame sent by a station in a Token Ring network passes in front of all stations attached to the network. Each station can detect the data frame and analyze the destination address to possibly capture the frame.

The text above describes the basic operation of a Token Ring network when all stations work correctly. Unfortunately, a real Token Ring network must be able to handle various types of anomalies and this increases the complexity of Token Ring stations. We briefly list the problems and outline their solutions below. A detailed description of the operation of Token Ring stations may be found in [IEEE802.5]. The first problem is when all the stations attached to the network start. One of them must bootstrap the network by sending the first token. For this, all stations implement a distributed election mechanism that is used to select the *Monitor*. Any station can become a *Monitor*. The *Monitor* manages the Token Ring network and ensures that it operates correctly. Its first role is to introduce a delay of 24 bit transmission times to ensure that the token can travel smoothly on the ring. Second, the *Monitor* sends the first token on the ring. It must also verify that the token passes regularly. According to the Token Ring standard [IEEE802.5], a station cannot retain the token to transmit data frames for a duration longer than the *Token Holding Time* (THT) (slightly less than 10 milliseconds). On a network containing *N* stations, the *Monitor* must receive the token at least every $N \times THT$ seconds. If the *Monitor* does not receive a token during such a period, it cuts the ring for some time and then re-initializes the ring and sends a token.

Several other anomalies may occur in a Token Ring network. For example, a station could capture a token and be powered off before having resent the token. Another station could have captured the token, sent its data frame and be powered off before receiving all of its data frame. In this case, the bit string corresponding to the end of a frame would remain in the ring without being removed by its sender. Several techniques are defined in [IEEE802.5] to allow the *Monitor* to handle all these problems. If unfortunately, the *Monitor* fails, another station will be elected to become the new *Monitor*.

2.6.5 Congestion control

Most networks contain links having different bandwidth. Some hosts can use low bandwidth wireless networks. Some servers are attached via 10 Gbps interfaces and inter-router links may vary from a few tens of kilobits per second up to hundred Gbps. Despite these huge differences in performance, any host should be able to efficiently exchange segments with a high-end server.

To understand this problem better, let us consider the scenario shown in the figure below, where a server (*A*) attached to a *10 Mbps* link needs to reliably transfer segments to another computer (*C*) through a path that contains a *2 Mbps* link.

In this network, the segments sent by the server reach router *R1*. *R1* forwards the segments towards router *R2*. Router *R1* can potentially receive segments at *10 Mbps*, but it can only forward them at *2 Mbps* to router *R2* and then to host *C*.

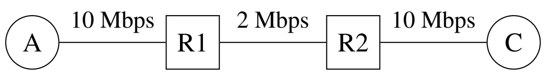


Fig. 77: Reliable transport with heterogeneous links

Router *R1* includes buffers that allow it to store the packets that cannot immediately be forwarded to their destination. To understand the operation of a reliable transport protocol in this environment, let us consider a simplified model of this network where host *A* is attached to a *10 Mbps* link to a queue that represents the buffers of router *R1*. This queue is emptied at a rate of *2 Mbps*.

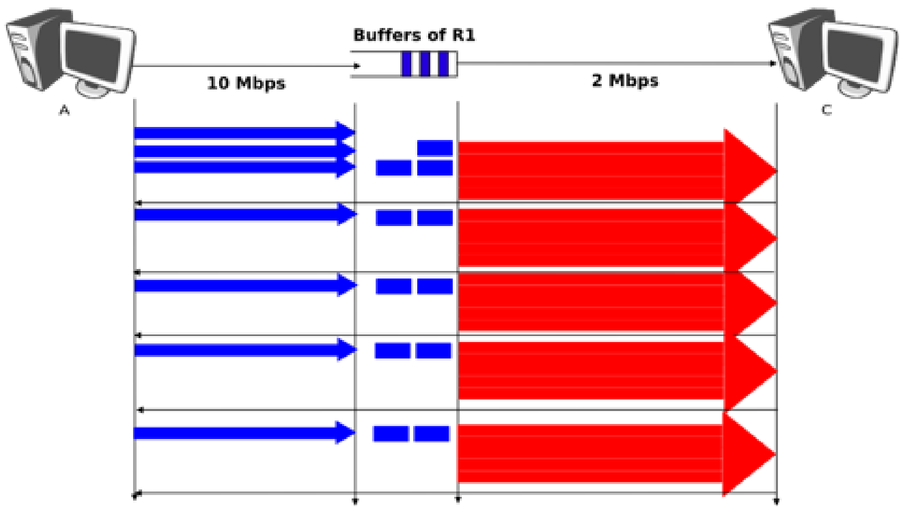


Fig. 78: Self clocking

Let us consider that host *A* uses a window of three segments. It thus sends three back-to-back segments at *10 Mbps* and then waits for an acknowledgment. Host *A* stops sending segments when its window is full. These segments reach the buffers of router *R1*. The first segment stored in this buffer is sent by router *R1* at a rate of *2 Mbps* to the destination host. Upon reception of this segment, the destination sends an acknowledgment. This acknowledgment allows host *A* to transmit a new segment. This segment is stored in the buffers of router *R1* while it is transmitting the second segment that was sent by host *A*… Thus, after the transmission of the first window of segments, the reliable transport protocol sends one data segment after the reception of each acknowledgment returned by the destination. In practice, the acknowledgments sent by the destination serve as a kind of *clock* that allows the sending host to adapt its transmission rate to the rate at which segments are received by the destination. This *self-clocking* is the first mechanism that allows a window-based reliable transport protocol to adapt to heterogeneous networks [Jacobson1988]. It depends on the availability of buffers to store the segments that have been sent by the sender but have not yet been transmitted to the destination.

However, transport protocols are not only used in this environment. In the global Internet, a large number of hosts send segments to a large number of receivers. For example, let us consider the network depicted below which is similar to the one discussed in [Jacobson1988] and **RFC 896**. In this network, we assume that the buffers of the router are infinite to ensure that no packet is lost.

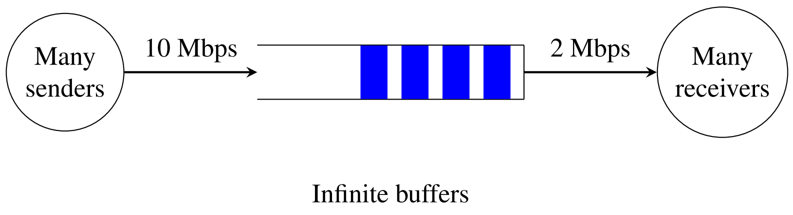


Fig. 79: The congestion collapse problem

If many senders are attached to the left part of the network above, they all send a window full of segments. These segments are stored in the buffers of the router before being transmitted towards their destination. If there are many senders on the left part of the network, the occupancy of the buffers quickly grows. A consequence of the buffer occupancy is that the round-trip-time, measured by the transport protocol, between the sender and the receiver increases. Consider a network where 10,000 bits segments are sent. When the buffer is empty, such a segment requires 1 millisecond to be transmitted on the *10 Mbps* link and 5 milliseconds to be the transmitted on the *2 Mbps* link. Thus, the measured round-trip-time measured is roughly 6 milliseconds if we ignore the propagation delay on the links. If the buffer contains 100 segments, the round-trip-time becomes $1 + 100 \times 5 + 5$ milliseconds as new segments are only transmitted on the *2 Mbps* link once all previous segments have been transmitted. Unfortunately, if the reliable transport protocol uses a retransmission timer and performs *go-back-n* to recover from transmission errors it will retransmit a full window of segments. This increases the occupancy of the buffer and the delay through the buffer… Furthermore, the buffer may store and send on the low bandwidth links several retransmissions of the same segment. This problem is called *congestion collapse*. It occurred several times during the late 1980s on the Internet [Jacobson1988].

The *congestion collapse* is a problem that all heterogeneous networks face. Different mechanisms have been proposed in the scientific literature to avoid or control network congestion. Some of them have been implemented and deployed in real networks. To understand this problem in more detail, let us first consider a simple network with two hosts attached to a high bandwidth link that are sending segments to destination *C* attached to a low bandwidth link as depicted below.

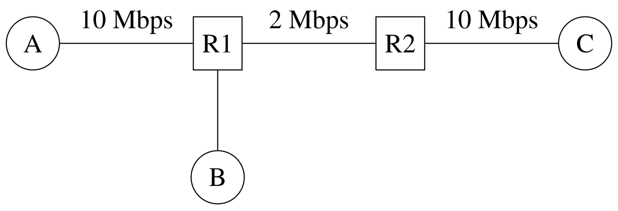


Fig. 80: The congestion problem

To avoid *congestion collapse*, the hosts must regulate their transmission rate[5] by using a *congestion control* mechanism. Such a mechanism can be implemented in the transport layer or in the network layer. In TCP/IP networks, it is implemented in the transport layer, but other technologies such as *Asynchronous Transfer Mode (ATM)* or *Frame Relay* include congestion control mechanisms in lower layers.

Let us first consider the simple problem of a set of i hosts that share a single bottleneck link as shown in the example above. In this network, the congestion control scheme must achieve the following objectives [CJ1989] :

1. The congestion control scheme must *avoid congestion*. In practice, this means that the bottleneck link cannot be overloaded. If $r_i(t)$ is the transmission rate allocated to host i at time t and R the

[5] In this section, we focus on congestion control mechanisms that regulate the transmission rate of the hosts. Other types of mechanisms have been proposed in the literature. For example, *credit-based* flow-control has been proposed to avoid congestion in ATM networks [KR1995]. With a credit-based mechanism, hosts can only send packets once they have received credits from the routers and the credits depend on the occupancy of the router's buffers.

bandwidth of the bottleneck link, then the congestion control scheme should ensure that, on average, $\forall t \sum r_i(t) \leq R$.

2. The congestion control scheme must be *efficient*. The bottleneck link is usually both a shared and an expensive resource. Usually, bottleneck links are wide area links that are much more expensive to upgrade than the local area networks. The congestion control scheme should ensure that such links are efficiently used. Mathematically, the control scheme should ensure that $\forall t \sum r_i(t) \approx R$.
3. The congestion control scheme should be *fair*. Most congestion schemes aim at achieving *max-min fairness*. An allocation of transmission rates to sources is said to be *max-min fair* if :

- no link in the network is congested
- the rate allocated to source j cannot be increased without decreasing the rate allocated to a source i whose allocation is smaller than the rate allocated to source j [Leboudec2008] .

Depending on the network, a *max-min fair allocation* may not always exist. In practice, *max-min fairness* is an ideal objective that cannot necessarily be achieved. When there is a single bottleneck link as in the example above, *max-min fairness* implies that each source should be allocated the same transmission rate.

To visualize the different rate allocations, it is useful to consider the graph shown below. In this graph, we plot on the *x-axis* (resp. *y-axis*) the rate allocated to host *B* (resp. *A*). A point in the graph (r_B, r_A) corresponds to a possible allocation of the transmission rates. Since there is a *2 Mbps* bottleneck link in this network, the graph can be divided into two regions. The lower left part of the graph contains all allocations (r_B, r_A) such that the bottleneck link is not congested $(r_A + r_B < 2)$. The right border of this region is the *efficiency line*, i.e. the set of allocations that completely utilize the bottleneck link $(r_A + r_B = 2)$. Finally, the *fairness line* is the set of fair allocations.

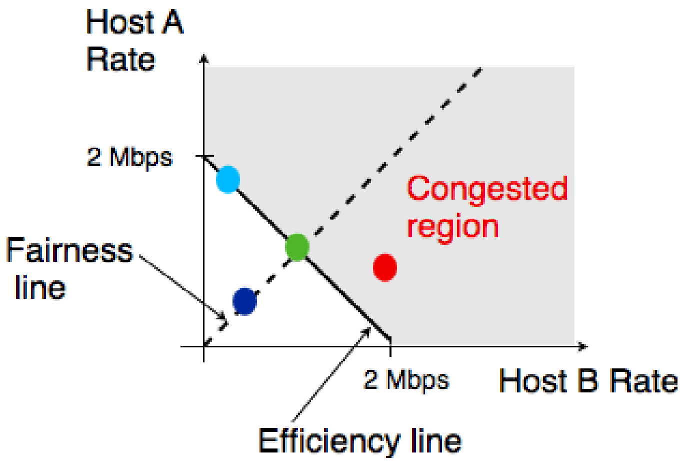


Fig. 81: Possible allocated transmission rates

As shown in the graph above, a rate allocation may be fair but not efficient (e.g. $r_A = 0.7, r_B = 0.7$), fair and efficient (e.g. $r_A = 1, r_B = 1$) or efficient but not fair (e.g. $r_A = 1.5, r_B = 0.5$). Ideally, the allocation should be both fair and efficient. Unfortunately, maintaining such an allocation with fluctuations in the number of flows that use the network is a challenging problem. Furthermore, there might be several thousands flows that pass through the same link[6].

To deal with these fluctuations in demand, which result in fluctuations in the available bandwidth, computer networks use a congestion control scheme. This congestion control scheme should achieve the three objectives listed above. Some congestion control schemes rely on a close cooperation between the end hosts and the routers, while others are mainly implemented on the end hosts with limited support from the routers.

[6] For example, the measurements performed in the Sprint network in 2004 reported more than 10k active TCP connections on a link, see https://research.sprintlabs.com/packstat/packetoverview.php. More recent information about backbone links may be obtained from caida 's real-time measurements, see e.g. http://www.caida.org/data/realtime/passive/

A congestion control scheme can be modeled as an algorithm that adapts the transmission rate ($r_i(t)$) of host i based on the feedback received from the network. Different types of feedback are possible. The simplest scheme is a binary feedback [CJ1989] [Jacobson1988] where the hosts simply learn whether the network is congested or not. Some congestion control schemes allow the network to regularly send an allocated transmission rate in Mbps to each host [BF1995].

Let us focus on the binary feedback scheme which is the most widely used today. Intuitively, the congestion control scheme should decrease the transmission rate of a host when congestion has been detected in the network, in order to avoid congestion collapse. Furthermore, the hosts should increase their transmission rate when the network is not congested. Otherwise, the hosts would not be able to efficiently utilize the network. The rate allocated to each host fluctuates with time, depending on the feedback received from the network. The figure below illustrates the evolution of the transmission rates allocated to two hosts in our simple network. Initially, two hosts have a low allocation, but this is not efficient. The allocations increase until the network becomes congested. At this point, the hosts decrease their transmission rate to avoid congestion collapse. If the congestion control scheme works well, after some time the allocations should become both fair and efficient.

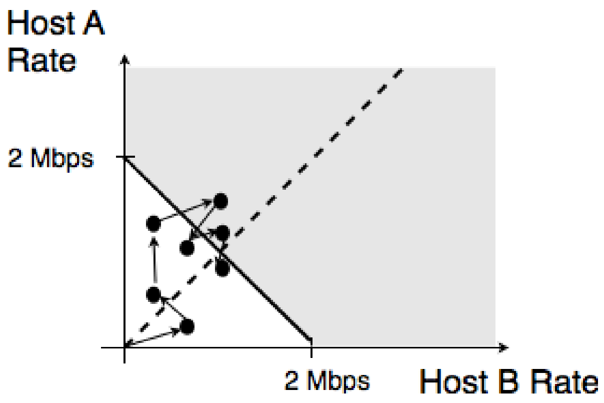


Fig. 82: Evolution of the transmission rates

Various types of rate adaption algorithms are possible. Dah Ming Chiu and Raj Jain have analyzed, in [CJ1989], different types of algorithms that can be used by a source to adapt its transmission rate to the feedback received from the network. Intuitively, such a rate adaptation algorithm increases the transmission rate when the network is not congested (ensure that the network is efficiently used) and decrease the transmission rate when the network is congested (to avoid congestion collapse).

The simplest form of feedback that the network can send to a source is a binary feedback (the network is congested or not congested). In this case, a *linear* rate adaptation algorithm can be expressed as :

- $rate(t+1) = \alpha_C + \beta_C rate(t)$ when the network is congested
- $rate(t+1) = \alpha_N + \beta_N rate(t)$ when the network is *not* congested

With a linear adaption algorithm, $\alpha_C, \alpha_N, \beta_C$ and β_N are constants. The analysis of [CJ1989] shows that to be fair and efficient, such a binary rate adaption mechanism must rely on *Additive Increase and Multiplicative Decrease*. When the network is not congested, the hosts should slowly increase their transmission rate ($\beta_N = 1\ and\ \alpha_N > 0$). When the network is congested, the hosts must multiplicatively decrease their transmission rate ($\beta_C < 1\ and\ \alpha_C = 0$). Such an AIMD rate adaptation algorithm can be implemented by the pseudo-code below.

```
# Additive Increase Multiplicative Decrease
if congestion:
    rate = rate * betaC     # multiplicative decrease, betaC<1
else
    rate = rate + alphaN     # additive increase, alphaN > 0
```

Note: Which binary feedback ?

Two types of binary feedback are possible in computer networks. A first solution is to rely on implicit feedback. This is the solution chosen for TCP. TCP's congestion control scheme [Jacobson1988] does not require any cooperation from the router. It only assumes that they use buffers and that they discard packets when there is congestion. TCP uses the segment losses as an indication of congestion. When there are no losses, the network is assumed to be not congested. This implies that congestion is the main cause of packet losses. This is true in wired networks, but unfortunately not always true in wireless networks. Another solution is to rely on explicit feedback. This is the solution proposed in the DECBit congestion control scheme [RJ1995] and used in Frame Relay and ATM networks. This explicit feedback can be implemented in two ways. A first solution would be to define a special message that could be sent by routers to hosts when they are congested. Unfortunately, generating such messages may increase the amount of congestion in the network. Such a congestion indication packet is thus discouraged **RFC 1812**. A better approach is to allow the intermediate routers to indicate, in the packets that they forward, their current congestion status. Binary feedback can be encoded by using one bit in the packet header. With such a scheme, congested routers set a special bit in the packets that they forward while non-congested routers leave this bit unmodified. The destination host returns the congestion status of the network in the acknowledgments that it sends. Details about such a solution in IP networks may be found in **RFC 3168**. Unfortunately, as of this writing, this solution is still not deployed despite its potential benefits.

Congestion control with a window-based transport protocol

AIMD controls congestion by adjusting the transmission rate of the sources in reaction to the current congestion level. If the network is not congested, the transmission rate increases. If congestion is detected, the transmission rate is multiplicatively decreased. In practice, directly adjusting the transmission rate can be difficult since it requires the utilization of fine grained timers. In reliable transport protocols, an alternative is to dynamically adjust the sending window. This is the solution chosen for protocols like TCP and SCTP that will be described in more details later. To understand how window-based protocols can adjust their transmission rate, let us consider the very simple scenario of a reliable transport protocol that uses *go-back-n*. Consider the very simple scenario shown in the figure below.

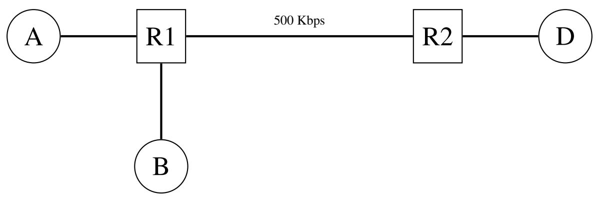


Fig. 83: A simple network with hosts sharing a bottleneck link

The links between the hosts and the routers have a bandwidth of 1 Mbps while the link between the two routers has a bandwidth of 500 Kbps. There is no significant propagation delay in this network. For simplicity, assume that hosts *A* and *B* send 1000 bits packets. The transmission of such a packet on a *host-router* (resp. *router-router*) link requires 1 msec (resp. 2 msec). If there is no traffic in the network, the round-trip-time measured by host *A* to reach *D* is slightly larger than 4 msec. Let us observe the flow of packets with different window sizes to understand the relationship between sending window and transmission rate.

Consider first a window of one segment. This segment takes 4 msec to reach host *D*. The destination replies with an acknowledgment and the next segment can be transmitted. With such a sending window, the transmission rate is roughly 250 segments per second or 250 Kbps. This is illustrated in the figure below where each square of the grid corresponds to one millisecond.

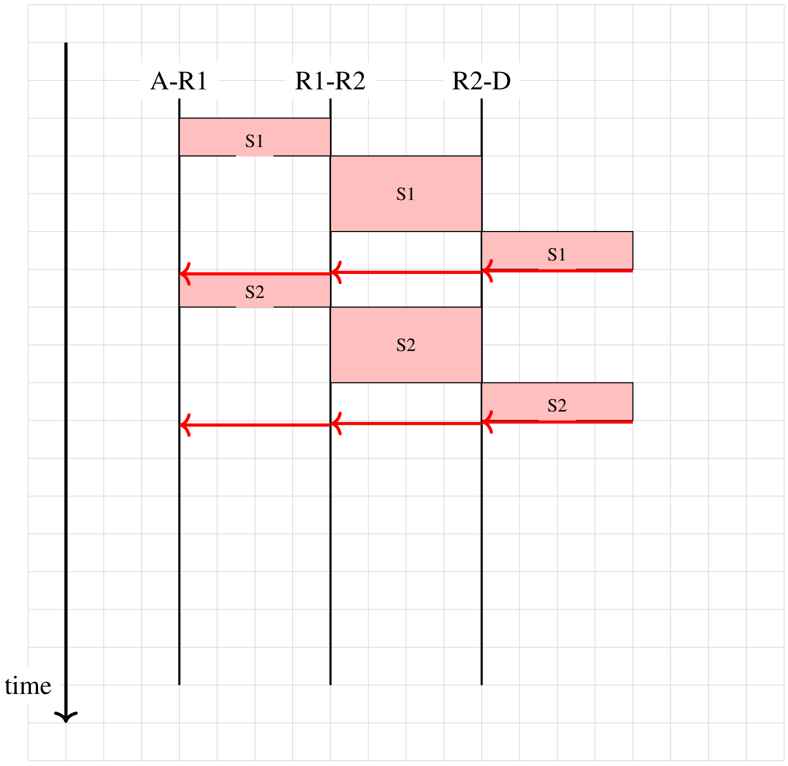


Fig. 84: Go-back-n transfer from A to D, window of one segment

Consider now a window of two segments. Host *A* can send two segments within 2 msec on its 1 Mbps link. If the first segment is sent at time t_0, it reaches host *D* at $t_0 + 4$. Host *D* replies with an acknowledgment that opens the sending window on host *A* and enables it to transmit a new segment. In the meantime, the second segment was buffered by router *R1*. It reaches host *D* at $t_0 + 6$ and an acknowledgment is returned. With a window of two segments, host *A* transmits at roughly 500 Kbps, i.e. the transmission rate of the bottleneck link.

Our last example is a window of four segments. These segments are sent at t_0, $t_0 + 1$, $t_0 + 2$ and $t_0 + 3$. The first segment reaches host *D* at $t_0 + 4$. Host *D* replies to this segment by sending an acknowledgment that enables host *A* to transmit its fifth segment. This segment reaches router *R1* at $t_0 + 5$. At that time, router *R1* is transmitting the third segment to router *R2* and the fourth segment is still in its buffers. At time $t_0 + 6$, host *D* receives the second segment and returns the corresponding acknowledgment. This acknowledgment enables host *A* to send its sixth segment. This segment reaches router *R1* at roughly $t_0 + 7$. At that time, the router starts to transmit the fourth segment to router *R2*. Since link *R1-R2* can only sustain 500 Kbps, packets will accumulate in the buffers of *R1*. On average, there will be two packets waiting in the buffers of *R1*. The presence of these two packets will induce an increase of the round-trip-time as measured by the transport protocol. While the first segment was acknowledged within 4 msec, the fifth segment (*data(4)*) that was transmitted at time $t_0 + 4$ is only acknowledged at time $t_0 + 11$. On average, the sender transmits at 500 Kbps, but the utilization of a large window induces a longer delay through the network.

From the above example, we can adjust the transmission rate by adjusting the sending window of a reliable transport protocol. A reliable transport protocol cannot send data faster than $\frac{window}{rtt}$ segments per second where $window$ is the current sending window. To control the transmission rate, we introduce a *congestion window*. This congestion window limits the sending window. At any time, the sending window is restricted to $\min(swin, cwin)$, where *swin* is the sending window and *cwin* the current *congestion window*. Of course, the window is further constrained by the receive window advertised by the remote peer. With the utilization of a congestion window, a simple reliable transport

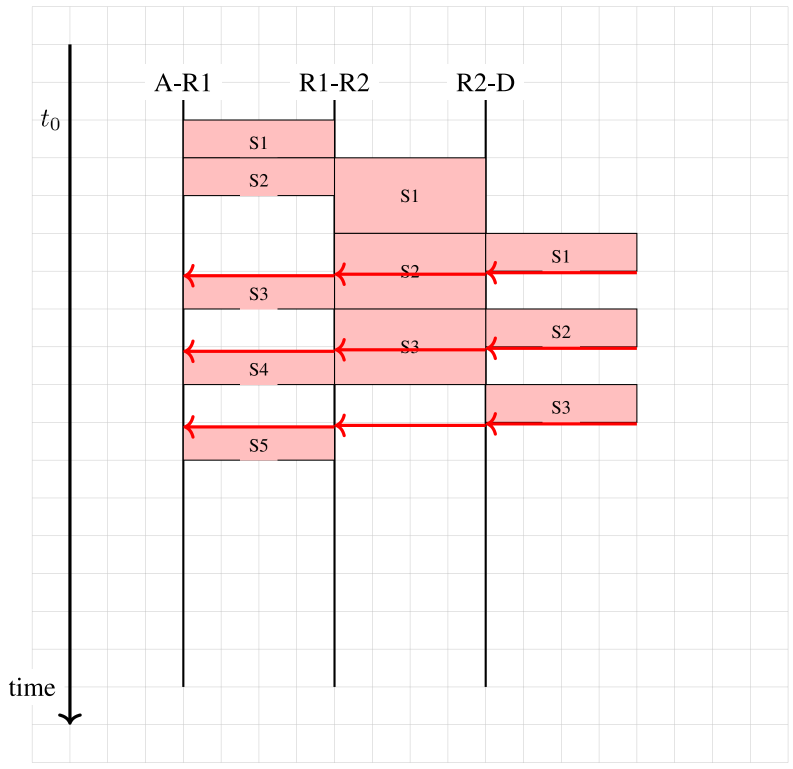


Fig. 85: Go-back-n transfer from A to D, window of two segments

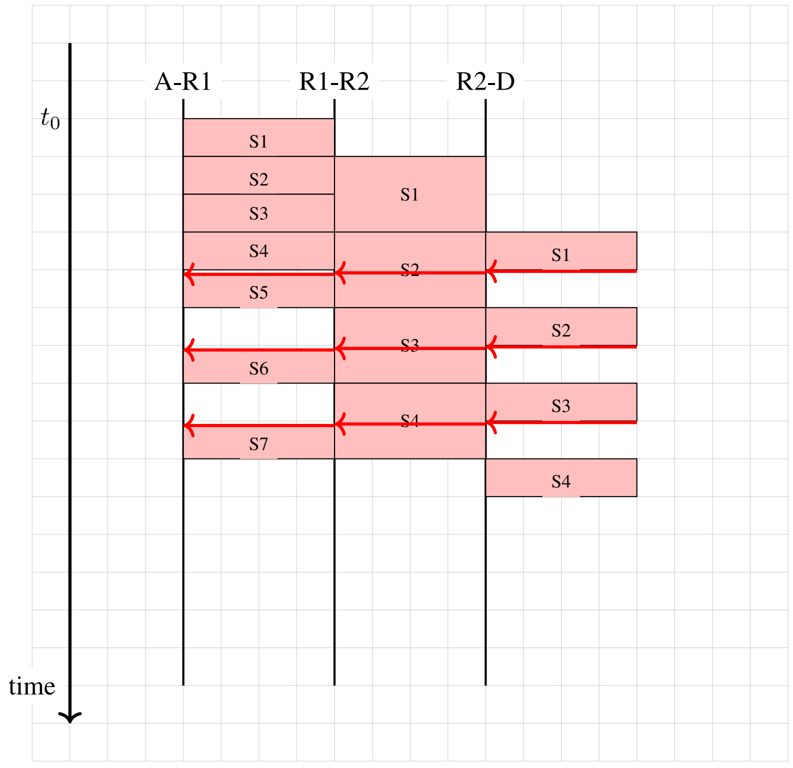


Fig. 86: Go-back-n transfer from A to D, window of four segments

protocol that uses fixed size segments could implement *AIMD* as follows.

For the *Additive Increase* part our simple protocol would simply increase its *congestion window* by one segment every round-trip-time. The *Multiplicative Decrease* part of *AIMD* could be implemented by halving the congestion window when congestion is detected. For simplicity, we assume that congestion is detected thanks to a binary feedback and that no segments are lost. We will discuss in more details how losses affect a real transport protocol like TCP in later sections.

A congestion control scheme for our simple transport protocol could be implemented as follows.

```
# Initialisation
cwin = 1  # congestion window measured in segments

# Ack arrival
if ack_received:
    if newack:  # new ack, no congestion
        # increase cwin by one every rtt
        cwin = cwin + (1/cwin)
    else:
        # no increase

if congestion_detected:
    cwin = cwin / 2 # only once per rtt
```

In the above pseudocode, *cwin* contains the congestion window stored as a real number of segments. This congestion window is updated upon the arrival of each acknowledgment and when congestion is detected. For simplicity, we assume that *cwin* is stored as a floating point number but only full segments can be transmitted.

As an illustration, let us consider the network scenario above and assume that the router implements the DECBit binary feedback scheme [RJ1995]. This scheme uses a form of Forward Explicit Congestion Notification and a router marks the congestion bit in arriving packets when its buffer contains one or more packets. In the figure below, we use a * to indicate a marked packet.

When the connection starts, its congestion window is set to one segment. Segment *S0* is sent an acknowledgment at roughly $t_0 + 4$. The congestion window is increased by one segment and *S1* and *S2* are transmitted at time $t_0 + 4$ and $t_0 + 5$. The corresponding acknowledgments are received at times $t_0 + 8$ and $t_0 + 10$. Upon reception of this last acknowledgment, the congestion window reaches *3* and segments can be sent (*S4* and *S5*). When segment *S6* reaches router *R1*, its buffers already contain *S5*. The packet containing *S6* is thus marked to inform the sender of the congestion. Note that the sender will only notice the congestion once it receives the corresponding acknowledgment at $t_0 + 18$. In the meantime, the congestion window continues to increase. At $t_0 + 16$, upon reception of the acknowledgment for *S5*, it reaches *4*. When congestion is detected, the congestion window is decreased down to *2*. This explains the idle time between the reception of the acknowledgment for *S*6* and the transmission of *S10*.

In practice, a router is connected to multiple input links. The figure below shows an example with two hosts.

In general, the links have a non-zero delay. This is illustrated in the figure below where a delay has been added on the link between *R* and *C*.

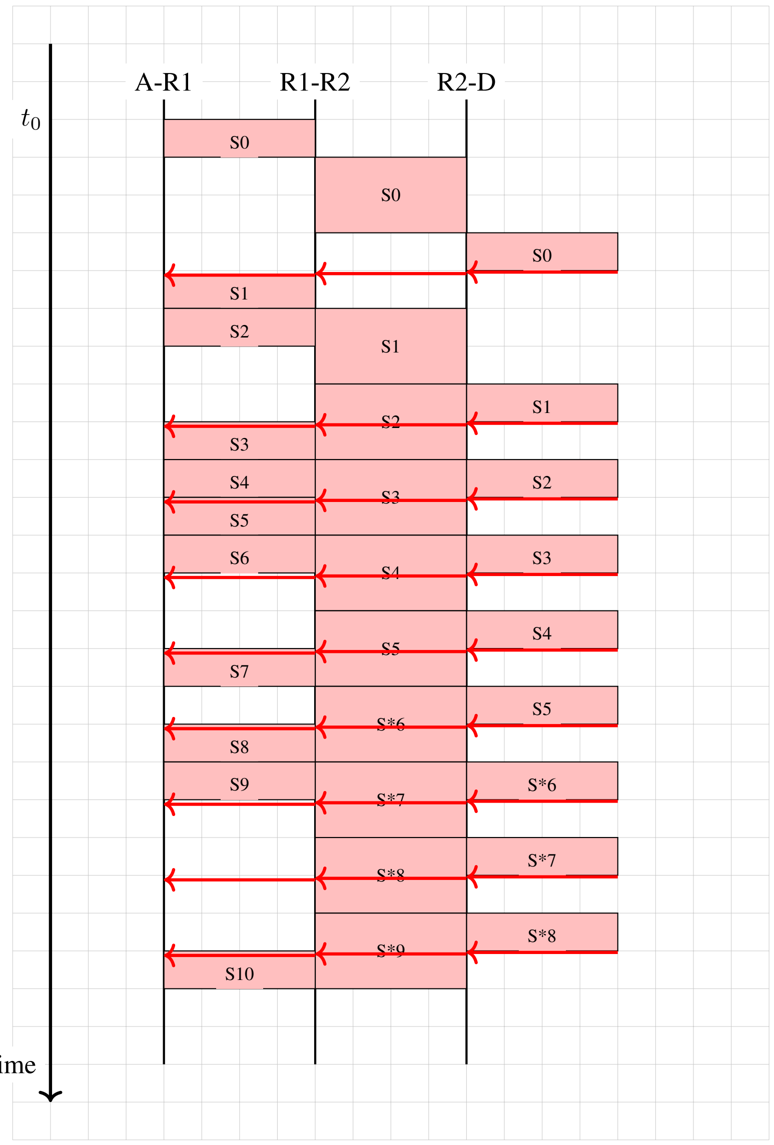


Fig. 87: Go-back-n transfer from A to D, with AIMD congestion control and DecBit binary feedback scheme

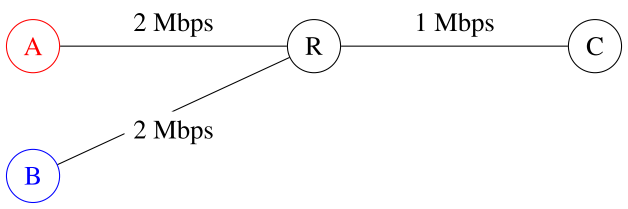


Fig. 88: A simple network with hosts sharing a bottleneck

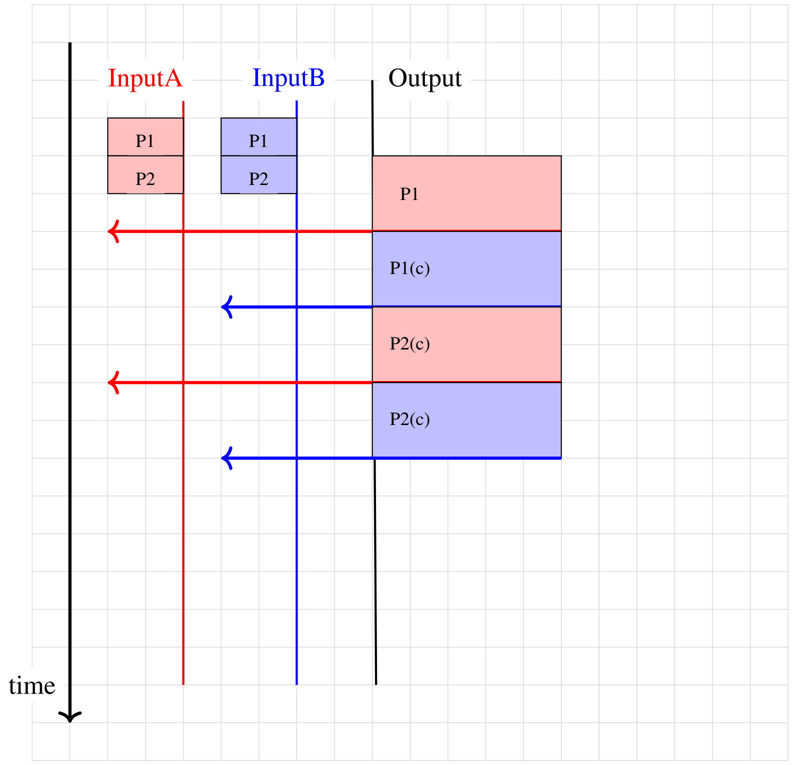


Fig. 89: R sharing the bottleneck link between different inputs

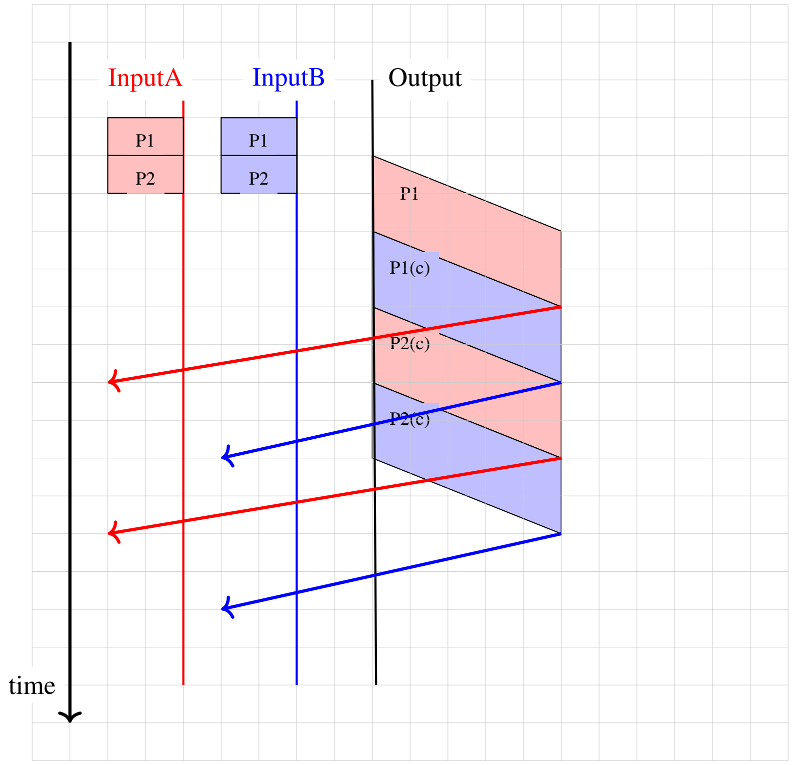


Fig. 90: R sharing the bottleneck link between different inputs

2.7 The reference models

Given the growing complexity of computer networks, during the 1970s network researchers proposed various reference models to facilitate the description of network protocols and services. Of these, the Open Systems Interconnection (OSI) model [Zimmermann80] was probably the most influential. It served as the basis for the standardization work performed within the *ISO* to develop global computer network standards. The reference model that we use in this book can be considered as a simplified version of the OSI reference model[1].

2.7.1 The five layers reference model

Our reference model is divided into five layers, as shown in the figure below.

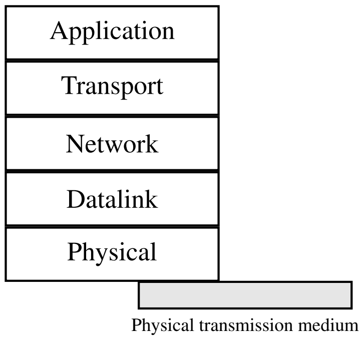


Fig. 91: The five layers of the reference model

2.7.2 The Physical layer

Starting from the bottom, the first layer is the Physical layer. Two communicating devices are linked through a physical medium. This physical medium is used to transfer an electrical or optical signal between two directly connected devices.

An important point to note about the Physical layer is the service that it provides. This service is usually an unreliable service that allows the users of the Physical layer to exchange bits. The unit of information transfer in the Physical layer is the bit. The Physical layer service is unreliable because :

- the Physical layer may change, e.g. due to electromagnetic interference, the value of a bit being transmitted
- the Physical layer may deliver *more* bits to the receiver than the bits sent by the sender
- the Physical layer may deliver *fewer* bits to the receiver than the bits sent by the sender

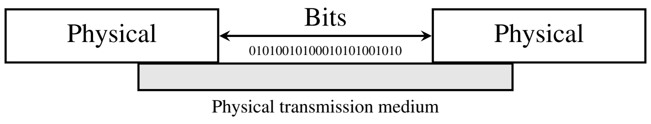


Fig. 92: The Physical layer

[1] An interesting historical discussion of the OSI-TCP/IP debate may be found in [Russel06]

2.7.3 The Datalink layer

The *Datalink layer* builds on the service provided by the underlying physical layer. The *Datalink layer* allows two hosts that are directly connected through the physical layer to exchange information. The unit of information exchanged between two entities in the *Datalink layer* is a frame. A frame is a finite sequence of bits. Some *Datalink layers* use variable-length frames while others only use fixed-length frames. Some *Datalink layers* provide a connection-oriented service while others provide a connectionless service. Some *Datalink layers* provide reliable delivery while others do not guarantee the correct delivery of the information.

An important point to note about the *Datalink layer* is that although the figure below indicates that two entities of the *Datalink layer* exchange frames directly, in reality this is slightly different. When the *Datalink layer* entity on the left needs to transmit a frame, it issues as many *Data.request* primitives to the underlying *physical layer* as there are bits in the frame. The physical layer will then convert the sequence of bits in an electromagnetic or optical signal that will be sent over the physical medium. The *physical layer* on the right hand side of the figure will decode the received signal, recover the bits and issue the corresponding *Data.indication* primitives to its *Datalink layer* entity. If there are no transmission errors, this entity will receive the frame sent earlier.

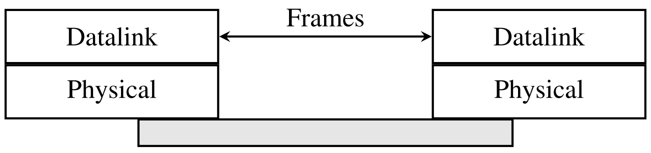


Fig. 93: The Datalink layer

2.7.4 The Network layer

The *Datalink layer* allows directly connected hosts to exchange information, but it is often necessary to exchange information between hosts that are not attached to the same physical medium. This is the task of the *network layer*. The *network layer* is built above the *datalink layer*. Network layer entities exchange *packets*. A *packet* is a finite sequence of bytes that is transported by the datalink layer inside one or more frames. A packet usually contains information about its origin and its destination, and usually passes through several intermediate devices called routers on its way from its origin to its destination.

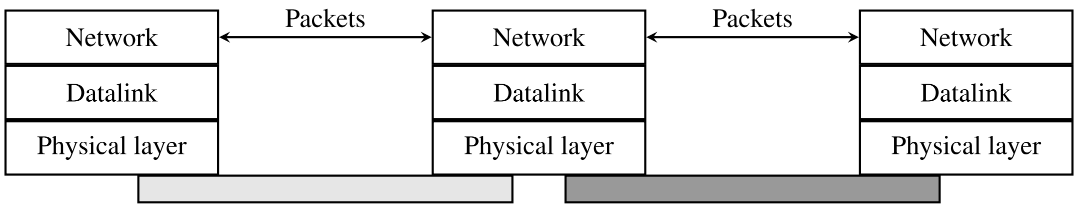


Fig. 94: The network layer

2.7.5 The Transport layer

The network layer enables hosts to reach each others. However, different communication flows can take place between the same hosts. These communication flows might have different needs (some require reliable delivery, other not) and need to be distinguished. Ensuring an identification of a communication flow between two given hosts is the task of the *transport layer*. *Transport layer* entities exchange *segments*. A segment is a finite sequence of bytes that are transported inside one or more packets. A transport layer entity issues segments (or sometimes part of segments) as *Data.request* to the underlying network layer entity.

There are different types of transport layers. The most widely used transport layers on the Internet are *TCP*, that provides a reliable connection-oriented bytestream transport service, and *UDP*, that provides an unreliable connection-less transport service.

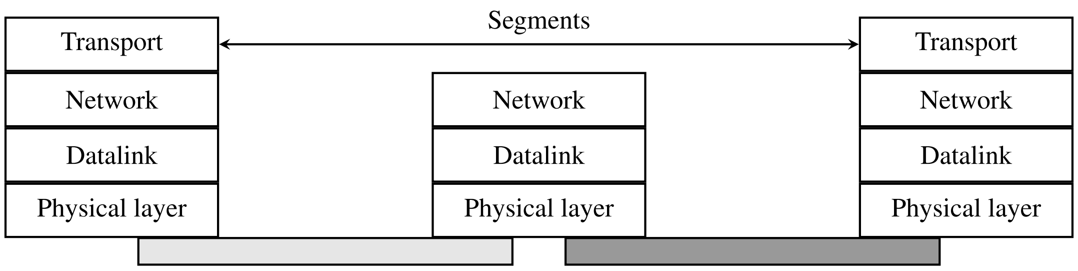


Fig. 95: The transport layer

2.7.6 The Application layer

The upper layer of our architecture is the *Application layer*. This layer includes all the mechanisms and data structures that are necessary for the applications. We will use Application Data Unit (ADU) or the generic Service Data Unit (SDU) term to indicate the data exchanged between two entities of the Application layer.

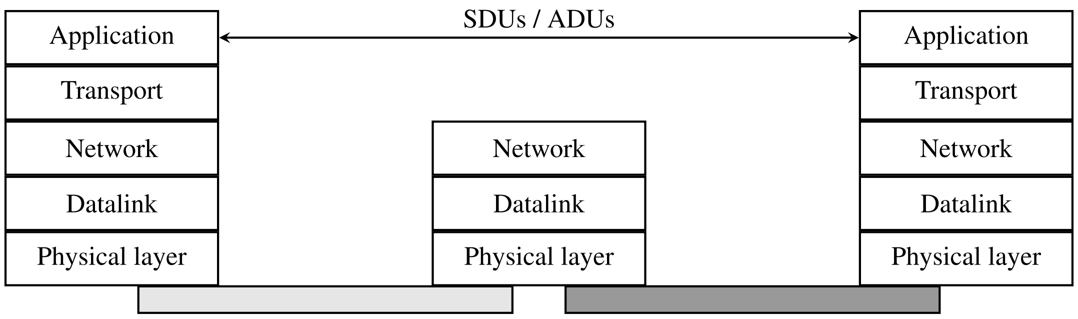


Fig. 96: The Application layer

In the remaining chapters of this text, we will often refer to the information exchanged between entities located in different layers. To avoid any confusion, we will stick to the terminology defined earlier, i.e. :

- physical layer entities exchange bits
- datalink layer entities exchange *frames*
- network layer entities exchange *packets*

- transport layer entities exchange *segments*
- application layer entities exchange *SDUs*

2.7.7 Reference models

Two reference models have been successful in the networking community : the OSI reference model and the TCP/IP reference model. We discuss them briefly in this section.

The TCP/IP reference model

In contrast with OSI, the TCP/IP community did not spend a lot of effort defining a detailed reference model; in fact, the goals of the Internet architecture were only documented after TCP/IP had been deployed [Clark88]. **RFC 1122**, which defines the requirements for Internet hosts, mentions four different layers. Starting from the top, these are :

- the Application layer
- the Transport layer
- the Internet layer which is equivalent to the network layer of our reference model
- the Link layer which combines the functions of the physical and datalink layers of our five-layer reference model

Besides this difference in the lower layers, the TCP/IP reference model is very close to the five layers that we use throughout this document.

The OSI reference model

Compared to the five layers reference model explained above, the *OSI* reference model defined in [X200] is divided in seven layers. The four lower layers are similar to the four lower layers described above. The OSI reference model refined the application layer by dividing it in three layers :

- the *Session layer*. The Session layer contains the protocols and mechanisms that are necessary to organize and to synchronize the dialogue and to manage the data exchange of presentation layer entities. While one of the main functions of the transport layer is to cope with the unreliability of the network layer, the session's layer objective is to hide the possible failures of transport-level connections to the upper layer higher. For this, the Session Layer provides services that allow establishing a session-connection, to support in-order data exchange (including mechanisms that allow recovering from the abrupt release of an underlying transport connection), and to release the connection in an orderly manner.
- the *Presentation layer* was designed to cope with the different ways of representing information on computers. There are many differences in the way computer store information. Some computers store integers as 32 bits field, others use 64 bits field and the same problem arises with floating point number. For textual information, this is even more complex with the many different character codes that have been used[2]. The situation is even more complex when considering the exchange of structured information such as database records. To solve this problem, the Presentation layer provides a common representation of the data transferred. The *ASN.1* notation was designed for the Presentation layer and is still used today by some protocols.
- the *Application layer* that contains the mechanisms that do not fit in neither the Presentation nor the Session layer. The OSI Application layer was itself further divided in several generic service elements.

[2] There is now a rough consensus for the greater use of the Unicode character format. Unicode can represent more than 100,000 different characters from the known written languages on Earth. Maybe one day, all computers will only use Unicode to represent all their stored characters and Unicode could become the standard format to exchange characters, but we are not yet at this stage today.

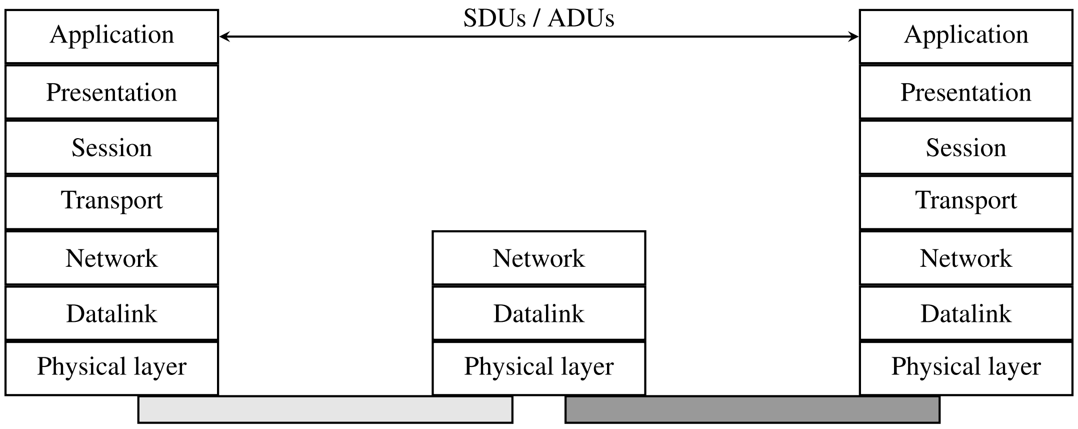


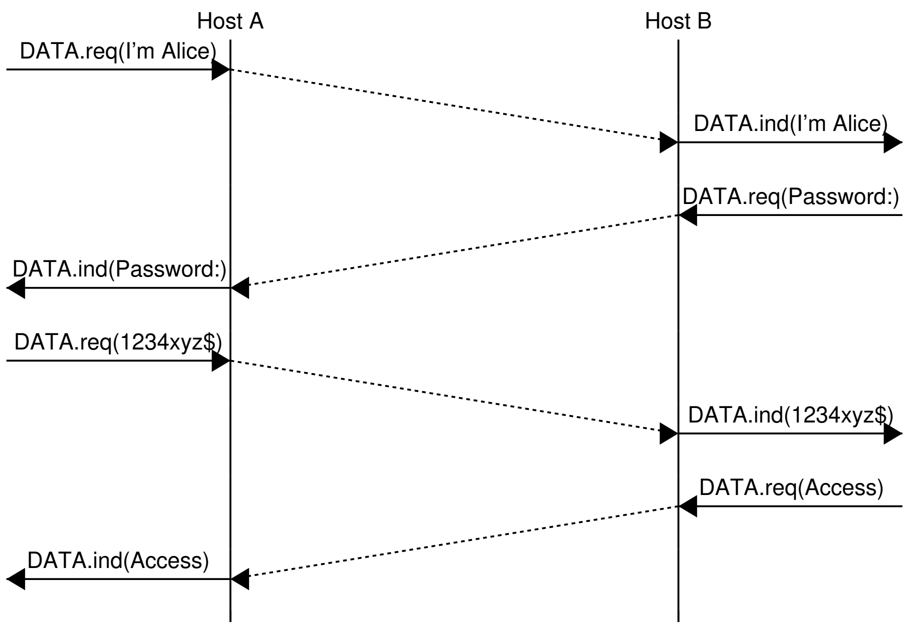
Fig. 97: The seven layers of the OSI reference model

2.8 Network security

In the early days, data networks were mainly used by researchers and security was not a concern. A few users were connected and capable of using the network. Almost all the devices attached to the network were openly accessible and users were trusted. As the utilization of the networks grew, security concerns started to appear. In universities, researchers and professors did not always trust their students and required some forms of access control. On standalone computers, the common access control mechanism is the password. A *username* is assigned to each user and when this user wants to access the computer, he or she needs to provide his/her *username* and his/her *password*. Most passwords are composed of a sequence of characters. The strength of the password is function of the difficulty of guessing the characters chosen by each user. Various guidelines have been defined on how to select a good password[1]. Some systems require regular modifications of the passwords chosen by their users.

When the first computers were attached to data networks, applications were developed to enable them to access to remote computers through the network. To authenticate the remote users, these applications have also relied on usernames and passwords. When a user connects to a distant computer, she sends her username through the network and then provides her password to confirm her *identity*. This authentication scheme is presented in the time sequence diagram below.

[1] The wikipedia page on passwords provides many of these references : https://en.wikipedia.org/wiki/Password_strength



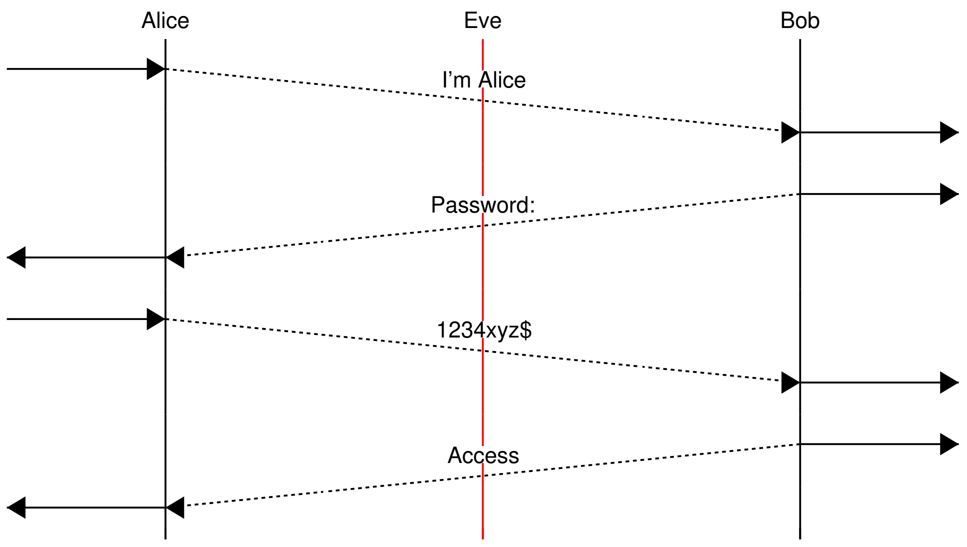
Note: Alice and Bob

Alice and Bob are the first names that are used in examples for security techniques. They first appeared in a seminal paper by Diffie and Hellman [DH1976]. Since then, Alice and Bob are the most frequently used names to represent the users who interact with a network. Other characters such as Eve or Mallory have been added over the years. We will explain their respective roles later.

2.8.1 Threats

When analyzing security issues in computer networks, it is useful to reason about the capabilities of the attacker who wants to exploit some breach in the security of the network. There are different types of attackers. Some have generic capabilities, others are specific to a given technology or network protocol. In this section, we discuss some important threats that a network architect must take into account.

The first type of attacker is called the *passive attacker*. A *passive attacker* is someone able to observe and usually store the information (e.g. the packets) exchanged in a given network or subset of it (e.g. a specific link). This attacker has access to all the data passing through this specific link. This is the most basic type of attacker and many network technologies are vulnerable to such attacks. In the above example, a passive attacker could easily capture the password sent by Alice and reuse it later to be authenticated as Alice on the remote computer. This is illustrated in the figure below where we do not show anymore the `DATA.req` and `DATA.ind` primitives but only the messages exchanged. Throughout this chapter, we will always use *Eve* as a user who is able to eavesdrop the data passing in front of her.

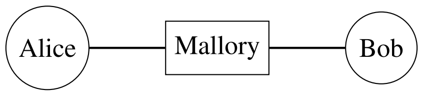


In the above example, *Eve* can capture all the packets exchanged by Bob and Alice. This implies that Eve can discover Alice's username and Alice's password. With this information, Eve can then authenticate as Alice on Bob's computer and do whatever Alice is authorized to do. This is a major problem from a security point of view. To prevent this attack, Alice should never send her password in clear over a network where someone could eavesdrop the information. In some networks, such as an open wireless network, an attacker can easily collect all the data sent by a particular user. In other networks, this is a bit more complex depending on the network technology used, but various software packages exist to automate this process. As will be described later, the best approach to prevent this type of attack is to rely on cryptographic techniques to ensure that passwords are never sent in clear.

Note: Pervasive monitoring

In the previous example, we have explained how Eve could capture data from a particular user. This is not the only attack of this type. In 2013, based on documents collected by Edward Snowden, the press revealed that several governmental agencies were collecting lots of data on various links that compose the global Internet [Greenwald2014]. Thanks to this massive amount of data, these governmental agencies have been able to extract lots of information about the behavior of Internet users. Like Eve, they are in a position to extract passwords, usernames and other privacy sensitive data from all the packets that they have captured. However, it seems that these agencies were often more interested in various meta data, e.g. information showing with whom a given user communicates than the actual data exchanged. These revelations have shocked the Internet community and the Internet Engineering Task Force that manages the standardization of Internet protocols has declared in **RFC 7258** that such pervasive monitoring is an attack that need to be countered in the development of new protocols. Several new protocols and extensions to existing ones are being developed to counter these attacks.

Eavesdropping and pervasive monitoring are not the only possible attacks against a network. Another type of attacker is the active attacker. In the literature, these attacks are often called *Man in the middle* or *MITM* attacks. Such attacks occur when one user, let us call him *Mallory*, has managed to configure the network so that he can both capture and modify the packets exchanged by two users. The simplest scenario is when Mallory controls a router that is on the path used by both Alice and Bob. For example, Alice could be connected to a WiFi access router controlled by Mallory and Bob would be a regular server on the Internet.



As Mallory receives all the packets sent by both Bob and Alice, he can modify them at will. For example, he could modify the commands sent by Alice to the server managed by Bob and change the responses sent by the server. This type of attack is very powerful and sometimes difficult to counter without relying on advanced cryptographic techniques.

The last type of attack that we consider in this introduction are the *Denial of Service* or DoS attacks. During such an attack, the attacker generates enough packets to saturate a given service and prevent it from operating correctly. The simplest Denial of Service attack is to send more packets that the bandwidth of the link that attaches the target to the network. The target could be a single server, a company or even an entire country. If these packets all come from the same source, then the victim can identify the attacker and contact the law enforcement authorities. In practice, such denial of service attacks do not originate from a single source. The attacker usually compromises a (possibly very large) set of sources and forces them to send packets to saturate a given target. Since the attacking traffic comes from a wide range of sources, it is difficult for the victim to locate the culprit and also to counter the attack. Saturating a link is the simplest example of *Distributed Denial of Service (DDoS)* attacks.

In practice, there is a possibility of denial of service attacks as soon as there is a limited resource somewhere in the network. This resource can be the bandwidth of a link, but it could also be the computational power of a server, its memory or even the size of tables used by a given protocol implementation. Defending against real DoS attacks can be difficult, especially if the attacker controls a large number of sources that are used to launch the attacks. In terms of bandwidth, DoS attacks composed of a few Gbps to a few tens of Gbps of traffic are frequent on the Internet. In 2015, github.com suffered from a distributed DoS that reached a top bandwidth of 400 Gbps according to some reports.

When designing network protocols and applications that will be deployed on a large scale, it is important to take those DDoS attacks into account. Attackers use different strategies to launch DDoS attacks. Some have managed to gain control of a large number of sources by injecting malware on them. Others, and this is where protocol designers have an important role to play, simply exploit design flaws in some protocols. Consider a simple request-response protocol where the client sends a request and the server replies with a response. Often the response is larger or much larger than the request sent by the client. Consider that such a simple protocol is used over a datagram network. When Alice sends a datagram to Bob containing her request, Bob extracts both the request and Alice's address from the packet. He then sends his response in a single packet destined to Alice. Mallory would like to create a DoS attack against Alice without being identified. Since he has studied the specification of this protocol, he can send a request to Bob inside a packet having Alice's address as its source address. Bob will process the request and send his (large) response to Alice. If the response has the same size as the request, Mallory is producing a *reflection attack* since his packets are reflected by Bob. Alice would think that she is attacked by Bob. If there are many servers that operate the same service as Bob, Mallory could hide behind a large number of such reflectors. Unfortunately, the reflection attack can also become an amplification attack. This happens when the response sent by Bob is larger than the request that it has received. If the response is k times larger than the request, then when Mallory consumes 1 Gbps of bandwidth to send requests, his victim receives k Gbps of attack traffic. Such amplification attacks are a very important problem and protocol designers should ensure that they never send a large response before having received the proof that the request that they have received originated from the source indicated in the request.

2.8.2 Cryptographic primitives

Cryptography techniques have initially been defined and used by spies and armies to exchange secret information in manner that ensures that adversaries cannot decode the information even if they capture the message or the person carrying the message. A wide range of techniques have been defined. The first techniques relied on their secrecy to operate. One of the first encryption schemes is attributed to Julius Caesar. When he sent confidential information to his generals, he would encode each message by replacing each letter with another letter that is n positions after this letter in the alphabet. For example, the message *SECRET* becomes *VHFUHW* when encoded using Caesar's cipher. This technique could have puzzled some soldiers during Caesar's wars, but today even young kids can recover the original message from the ciphered one.

The security of the Caesar cipher depends on the confidentiality of the algorithm, but experience has shown that it is impossible to assume that an algorithm will remain secret, even for military applications. Instead, cryptographic techniques must be designed by assuming that the algorithm will be public and known to anyone. However, its behavior must be controlled by a small parameter, known as the key, that will only be known by the users who need to communicate secretly. This principle is attributed to Auguste Kerckhoff, a French cryptographer who first documented it :

A cryptographic algorithm should be secure even if the attacker knows everything about the system, except one parameter known as the secret key.

This principle is important because it remains the basic assumption of all cryptographers. Any system that relies on the secrecy of its algorithm to be considered secure is doomed to fail and be broken one day.

With the Kerckhoff principle, we can now discuss a simple but powerful encryption scheme that relies on the *XOR* logic operation. This operation is easily implemented in hardware and is supported by all microprocessors. Given a secret, K, it is possible to encode a message *M* by computing $C_M = K \oplus M$. The receiver of this messages can recover the original message as since $M = K \oplus (K \oplus M)$. This *XOR* operation is the key operation of the perfect cipher that is also called the Vernam cipher or the one-time pad. This cipher relies on a key that contains purely random bits. The encrypted message is then produced by XORing all the bits of the message with all the bits of the key. Since the key is random, it is impossible for an attacker to recover the original text (or plain text) from the encrypted one. From a security viewpoint, the one-time-pad is the best solution provided that the key is as long as the message.

Unfortunately, it is difficult to use this cipher in practice since the key must be as long as the message that needs to be transmitted. If the key is smaller than the message and the message is divided into blocks that have the same length as the key, then the scheme becomes less secure since the same key is used to decrypt different parts of the message. In practice, *XOR* is often one of the basic operations used by encryption schemes. To be usable, the deployed encryption schemes use keys that are composed of a small number of bits, typically 56, 64, 128, 256, . . .

A secret key encryption scheme is a perfectly reversible functions, i.e. given an encryption function *E*, there is an associated decryption function *D* such that $\forall k \forall M : D(K, E(M, K)) = M$.

Various secret key cryptographic functions have been proposed, implemented and deployed. The most popular ones are :

- DES, the Data Encryption Standard that became a standard in 1977 and has been widely used by industry. It uses 56 bits keys that are not considered sufficiently secure nowadays since attackers can launch brute-force attacks by testing all possible keys. Triple DES combines three 56 bits keys, making the brute force attacks more difficult.
- RC4 is an encryption scheme defined in the late 1980s by Ron Rivest for RSA Security. Given the speed of its software implementation, it has been included in various protocols and implementations. However, cryptographers have identified several weaknesses in this algorithm. It is now deprecated and should not be used anymore **RFC 7465**.
- AES or the Advanced Encryption Standard is an encryption scheme that was designed by the Belgian cryptographers Joan Daemen and Vincent Rijmen in 2001 [DR2002]. This algorithm has been standardized by the U.S. National Institute of Standards and Technology (NIST). It is now used by a wide range of applications and various hardware and software implementations exist. Many microprocessors include special instructions that ease the implementation of AES. AES divides the message to be encrypted in blocks of 128 bits and uses keys of length 128, 192 or 256 bits. The block size and the key length are important parameters of an encryption scheme. The block size indicates the smallest message that can be encrypted and forces the sender to divide each message in blocks of the supported size. If the message is larger than an integer number of blocks, then the message must be padded before being encrypted and this padding must be removed after decryption. The key size indicates the resistance of the encryption scheme against brute force attacks, i.e. attacks where the attacker tries all possible keys to find the correct one.

AES is widely used as of this writing, but other secret key encryption schemes continue to appear. ChaCha20, proposed by D. Bernstein is now used by several internet protocols **RFC 7539**. A detailed discussion of encryption schemes is outside the scope of this book. We will consider encryption schemes as black boxes whose operation depends on a single key. A detailed overview of several of these schemes may be found in [MVV2011].

In the 1970s, Diffie and Hellman proposed in their seminal paper [DH1976], a different type of encryption : *public key cryptography*. In public key cryptography, each user has two different keys :

- a public key (K_{pub}) that he can distribute to everyone
- a private key (K_{priv}) that he needs to store in a secure manner and never reveal to anyone

These two keys are generated together and they are linked by a complex mathematical relationship that is such that it is computationally difficult to compute K_{priv} from K_{pub}.

A public key cryptographic scheme is a combination of two functions :

- an encryption function, E_p, that takes a key and a message as parameters
- a decryption function, D_p, that takes a key and a message as parameters

The public key is used to encrypt a message so that it can only be read by the intended recipient. For example, let us consider two users : Alice and Bob. Alice (resp. Bob) uses the keys A_{priv} and A_{pub} (resp. B_{priv} and B_{pub}). To send a secure message *M* to Alice, Bob computes $CM = E_p(A_{pub}, M)$ and Alice can decrypt it by using $D_p(A_{priv}, CM) = D_p(A_{priv}, E_p(A_{pub}, M)) = M$.

Several public key encryption schemes have been proposed. Two of them have reached wide deployment :

- The Rivest Shamir Adleman (RSA) algorithm[2] proposed in [RSA1978] that relies on modular exponentiation with large integers.
- The Elliptic Curve Cryptography techniques[3] that rely on special properties of elliptic curves.

Another interesting property of public key cryptography is its ability to compute *signatures* that can be used to authenticate a message. This capability comes from the utilization of two different keys that are linked together. If Alice wants to sign a message *M*, she can compute $SM = E_p(A_{priv}, M)$. Anyone who receives this signed messaged can extract its content as $D_p(A_{pub}, SM) = D_p(A_{pub}, E_p(A_{priv}, M)) = M$. Everyone can use A_{pub} to check that the message was signed by using Alice's private key (A_{priv}). Since this key is only known by Alice, the ability to decrypt *SM* is a proof that the message was signed by Alice herself.

In practice, encrypting a message to sign it can be computationally costly, in particular if the message is a large file. A faster solution would be to summarize the document and only sign the summary of the document. A naive approach could be based on a checksum or CRC computed over the message. Alice would then compute $C = Checksum(M)$ and $SC = E_p(A_{priv}, C)$. She would then send both *M* and *SC* to the recipient of the message who can easily compute *C* from *SC* and verify the authenticity of the message. Unfortunately, this solution does not protect Alice and the message's recipient against a man-in-the-middle attack. If Mallory can intercept the message sent by Alice, he can easily modify Alice's message and tweak it so that it has the same checksum as the original one. The CRCs, although more complex to compute, suffer from the same problem.

To efficiently sign messages, Alice needs to be able to compute a summary of her message in a way that makes prohibits an attacker from generating a different message that has the same summary. *Cryptographic hash functions* were designed to solve this problem. The ideal hash function is a function that returns a different number for every possible input. In practice, it is impossible to find such a function. Cryptographic hash functions are an approximation of this perfect summarization function. They compute a summary of a given message in 128, 160, 256 bits or more. They also exhibit the *avalanche effect*. This effect indicates that a small change in the message causes a large change in the hash value. Finally hash functions are very difficult to invert. Knowing a hash value, it is computationally very difficult to find the corresponding input message. Several hash functions have been proposed by cryptographers. The most popular ones are :

- MD5, originally proposed in **RFC 1321**. It has been used in a wide range of applications. In 2010, attacks against MD5 were published and this hash function is now deprecated.
- SHA-1 is a cryptographic hash function that was standardized by the NIST in 1995. It outputs 160 bits results. It is now used in a variety of network protocols.
- SHA-2 is another family of cryptographic hash functions designed by the NIST. Different variants of SHA-2 can produce has values of 224, 256, 384 or 512 bits.

[2] A detailed explanation of the operation of the RSA algorithm is outside the scope of this e-book. Various tutorials such as the RSA page on wikipedia provide examples and tutorial information.

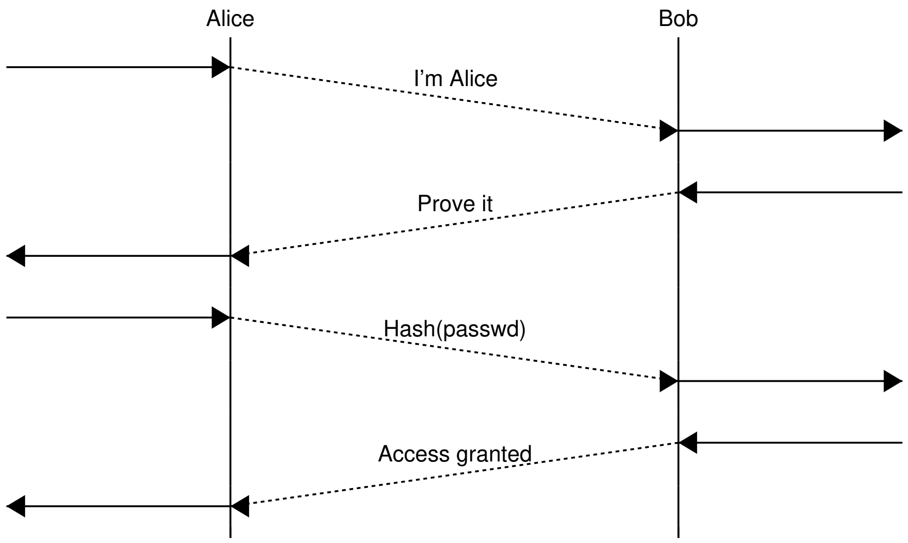
[3] A detailed explanation of the ECC cryptosystems is outside the scope of this e-book. A simple introduction may be found on Andrea Corbellini's blog. There have been deployments of ECC recently because ECC schemes usually require shorter keys than RSA and consume less CPU.

Another important point about cryptographic algorithms is that often these algorithms require random numbers to operate correctly (e.g. to generate keys). Generating good random numbers is difficult and any implementation of cryptographic algorithms should also include a secure random number generator. **RFC 4086** provides useful recommendations.

2.8.3 Cryptographic protocols

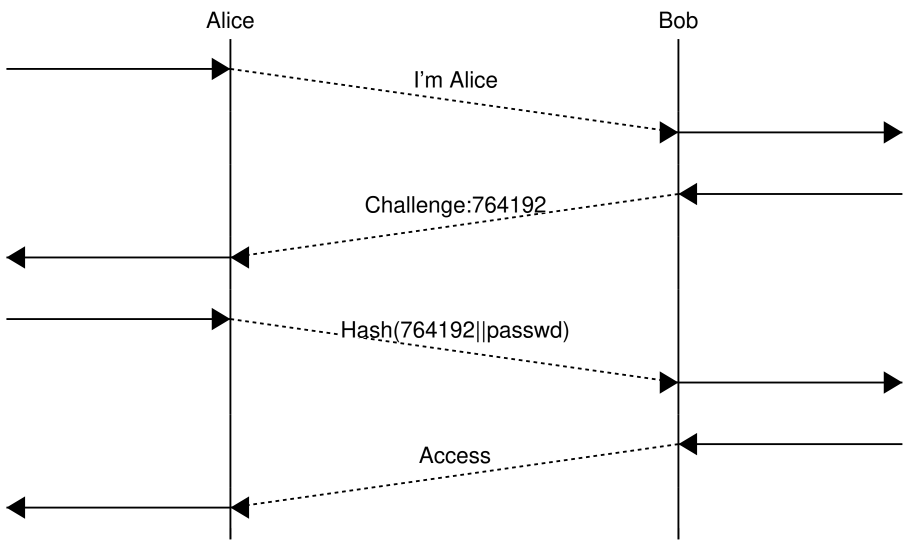
We can now combine the cryptographic operations described in the previous section to build some protocols to securely exchange information. Let us first go back to the problem of authenticating Alice on Bob's computer. We have shown earlier that using a simple password for this purpose is insecure in the presence of attackers.

A naive approach would be to rely on hash functions. Since hash functions are non-invertible, Alice and Bob could decide to use them to exchange Alice's password in a secure manner. Then, Alice could be authenticated by using the following exchange.



Since the hash function cannot be inverted, an eavesdropper cannot extract Alice's password by simply observing the data exchanged. However, Alice's real password is not the objective of an attacker. The main objective for Mallory is to be authenticated as Alice. If Mallory can capture *Hash(passwd)*, he can simply replay this data, without being able to invert the hash function. This is called a *replay attack*.

To counter this replay attack, we need to ensure that Alice never sends the same information twice to Bob. A possible mode of operation is shown below.



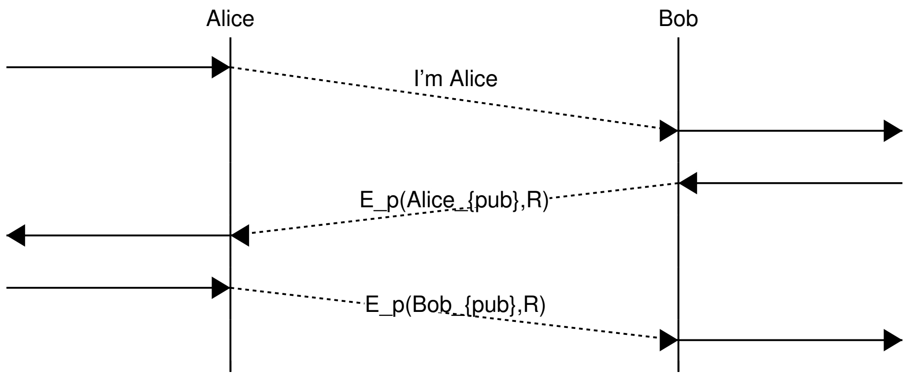
To authenticate herself, Alice sends her user identifier to Bob. Bob replies with a random number as a challenge to verify that Alice knows the shared secret (i.e. her password). Alice replies with the result of the computation of a hash function (e.g. SHA-1) over a string that is the concatenation between the random number chosen by Bob and Alice's password. The random number chosen by Bob is often called a *nonce* since this is a number that should only be used once. Bob performs the same computation locally and can check the message returned by Alice. This type of authentication scheme has been used in various protocols. It prevents replay attacks. If Eve captures the messages exchanged by Alice and Bob, she cannot recover Alice's password from the messages exchanged since hash functions are non-invertible. Furthermore, she cannot replay the hashed value since Bob will always send a different nonce.

Unfortunately, this solution forces Bob to store Alice's password in clear. Any breach in the security of Bob's computer would reveal Alice's password. Such breaches unfortunately occur and some of them have led to the dissemination of millions of passwords.

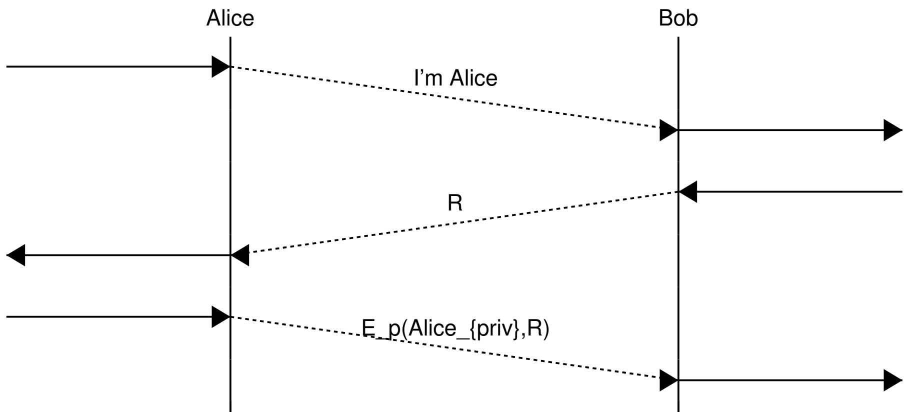
A better approach would be to authenticate Alice without storing her password in clear on Bob's computer. For this, Alice computes a *hash chain* as proposed by Lamport in [Lamport1981]. A hash chain is a sequence of applications of a hash function (*H*) on an input string. If Alice's password is *P*, then her 10 steps hash chain is : $H(H(H(H(H(H(H(H(H(H(P))))))))))$. The result of this hash chain will be stored on Bob's computer together with the value *10*. This number is the maximum number of remaining authentications for Alice on Bob's computer. To authenticate Alice, Bob sends the remaining number of authentications, i.e. *10* in this example. Since Alice knows her password, *P*, she can compute $H^9(P) = H(H(H(H(H(H(H(H(H(P)))))))))$ and send this information to Bob. Bob computes the hash of the value received from Alice ($H(H^9(P))$) and verifies that this value is equal to the value stored in his database. It then decrements the number of authorized authentications and stores $H^9(P)$ in his database. Bob is now ready for the next authentication of Alice. When the number of authorized authentications reaches zero, the hash chain needs to be reinitialized. If Eve captures ($H^n(P)$), she cannot use it to authenticate herself as Alice on Bob's computer because Bob will have decremented its number of authorized authentications. Furthermore, given that hash functions are not invertible, Eve cannot compute $H^{n-1}(P)$ from $H^n(P)$.

The two protocols above prevent eavesdropping attacks, but not man-in-the-middle attacks. If Mallory can intercept the messages sent by Alice, he could force her to reveal $H^n(P)$ and then use this information to authenticate as Alice on Bob's computer. In practice, hash chains should only be used when the communicating users know that there cannot be any man-in-the-middle on their communication.

Public key cryptography provides another possibility to allow Alice to authenticate herself on Bob's computer. Assume again that Alice and Bob know each other from previous encounters. Alice knows Bob's public key (Bob_{pub}) and Bob also knows Alice's key ($Alice_{pub}$). To authenticate herself, Alice could send her user identifier. Bob would reply with a random number encrypted with Alice's public key : $E_p(Alice_{pub}, R)$. Alice can decrypt this message to recover *R* and sends $E_p(Bob_{pub}, R)$. Bob decrypts the nonce and confirms that Alice knows $Alice_{priv}$. If an eavesdropper captures the messages exchanged, he cannot recover the value *R* which could be used as a key to encrypt the information with a secret key algorithm. This is illustrated in the time sequence diagram below.

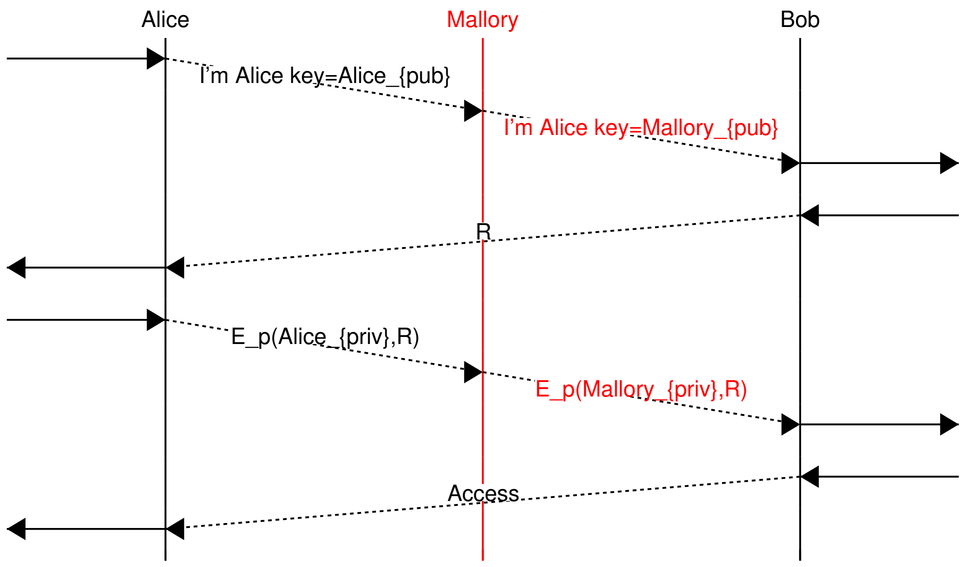


A drawback of this approach is that Bob is forced to perform two public key computations : one encryption to send the random nonce to Alice and one decryption to recover the nonce encrypted by Alice. If these computations are costly from a CPU viewpoint, this creates a risk of Denial of Service Attacks were attackers could try to access Bob's computer and force it to perform such costly computations. Bob is more at risk than Alice in this situation and he should not perform complex operations before being sure that he is talking with Alice. An alternative is shown in the time sequence diagram below.



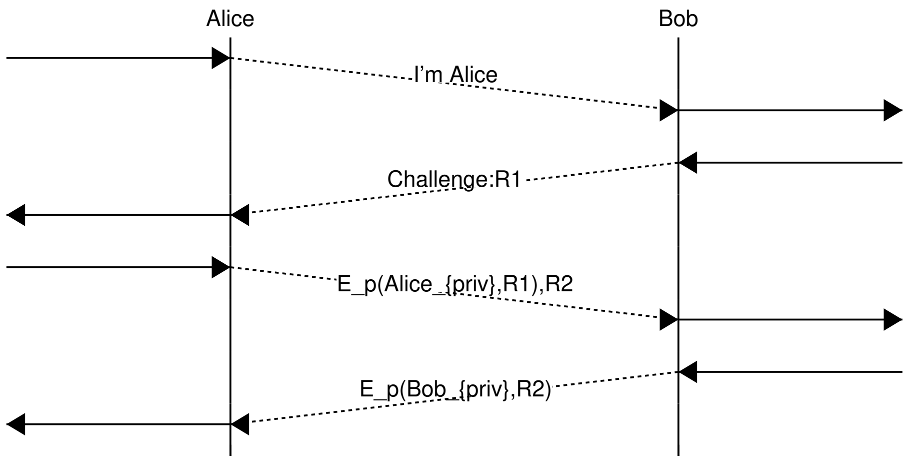
Here, Bob simply sends a random nonce to Alice and verifies her signature. Since the random nonce and the signature could be captured by an eavesdropper, they cannot be used as a secret key to encrypt further data. However Bob could propose a secret key and send it encrypted with Alice's public key in response to the signed nonce that he received.

The solution described above works provided that Bob and Alice know their respective public keys before communicating. Otherwise, the protocol is not secure against man-in-the-middle attackers. Consider Mallory sitting in the middle between Alice and Bob and assume that neither Alice nor Bob knows the other's public key.

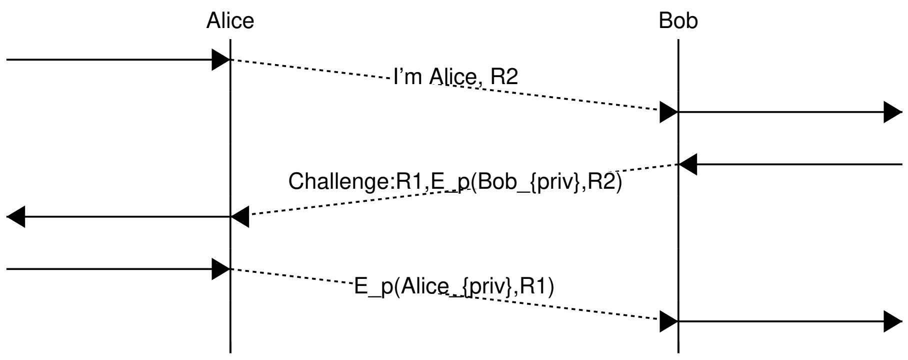


In the above example, Alice sends her public key, ($Alice_{pub}$), in her first message together with her identity. Mallory intercepts the message and replaces Alice's key with his own key, ($Mallory_{pub}$). Bob replies with a nonce, *R*. Alice then signs the random nonce to prove that she knows $Alice_{priv}$. Mallory discards the information and instead computes $E_p(Mallory_{priv}, R)$. Bob now thinks that he is discussing with Alice while Mallory sits in the middle.

There are situations where symmetric authentication is required. In this case, each user must perform some computation with his/her private key. A possible exchange is the following. Alice sends her certificate to Bob. Bob replies with a nonce, $R1$, and provides his certificate. Alice encrypts $R1$ with her private key and generates a nonce, $R2$. Bob verifies Alice's computation and encrypts $R2$ with his private key. Alice verifies the computation and both have been authenticated.

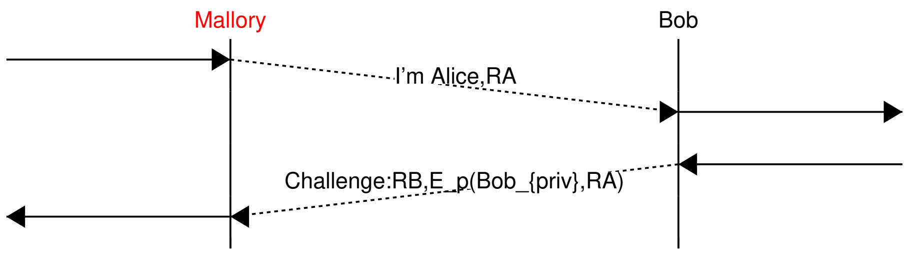


The protocol described above works, but it takes a long time for Bob to authenticate Alice and for Alice to authenticate Bob. A faster authentication could be the following.

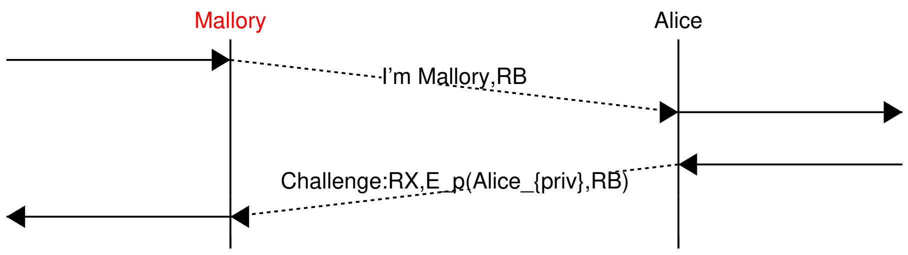


Alice sends her random nonce, $R2$. Bob signs $R2$ and sends his nonce : $R1$. Alice signs $R1$ and both are authenticated.

Now consider that Mallory wants to be authenticated as Alice. The above protocol has a subtle flaw that could be exploited by Mallory. This flaw can be exploited if Alice and Bob can act as both client and server. Knowing this, Mallory could operate as follows. Mallory starts an authentication with Bob faking himself as Alice. He sends a first message to Bob including Alice's identity.



In this exchange, Bob authenticates himself by signing the RA nonce that was sent by Mallory. Now, to authenticate as Alice, Mallory needs to compute the signature of nonce RB with Alice's private key. Mallory does not know Alice's key, but he could exploit the protocol to force Alice to perform the required computation. For this, Mallory can start an authentication to Alice as shown below.



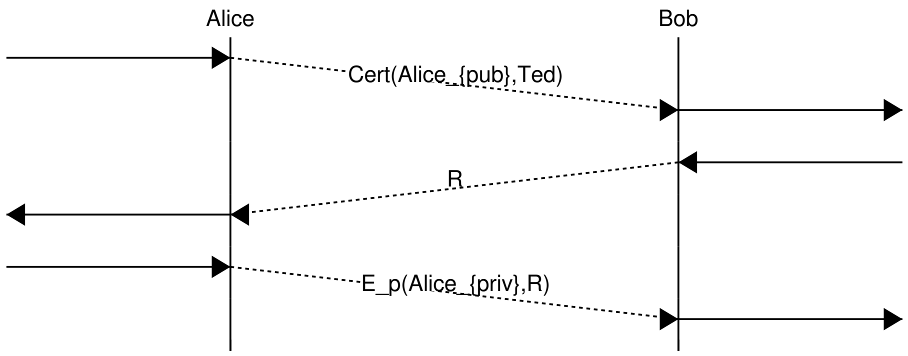
In this example, Mallory has forced Alice to compute $E_p(Alice_{priv}, RB)$ which is the information required to finalize the first exchange and be authenticated as Alice. This illustrates a common problem with authentication schemes when the same information can be used for different purposes. The problem comes from the fact that Alice agrees to compute her signature on a nonce chosen by Bob (and relayed by Mallory). This problem occurs if the nonce is a simple integer without any structure. If the nonce includes some structure such as some information about Alice and Bob's identities or even a single bit indicating whether the nonce was chosen by a user acting as a client (i.e. starting the authentication) or as a server, then the protocol is not vulnerable anymore.

To cope with some of the above mentioned problems, public-key cryptography is usually combined with certificates. A *certificate* is a data structure that includes a signature from a trusted third party. A simple explanation of the utilization of certificates is to consider that Alice and Bob both know Ted. Ted is trusted by these two users and both have stored Ted's public key : Ted_{pub}. Since they both know Ted's key, he can issue certificates. A certificate is mainly a cryptographic link between the identity of a user and his/her public key. Such a certificate can be computed in different ways. A simple solution is for Ted to generate a file that contains the following information for each certified user :

- his/her identity
- his/her public key
- a hash of the entire file signed with Ted's private key

Then, knowing Ted's public key, anyone can verify the validity of a certificate. When a user sends his/her public key, he/she must also attach the certificate to prove the link between his/her identity and the public key. In practice, certificates are more complex than this. Certificates will often be used to authenticate the server and sometimes to authenticate the client.

A possible protocol could then be the following. Alice sends $Cert(Alice_{pub}, Ted)$. Bob replies with a random nonce.



Until now, we have only discussed the authentication problem. This is an important but not sufficient step to have a secure communication between two users through an insecure network. To securely exchange information, Alice and Bob need to both :

- mutually authenticate each other
- agree on a way to encrypt the messages that they will exchange

Let us first explore how this could be realized by using public-key cryptography. We assume that Alice and Bob have both a public-private key pair and the corresponding certificates signed by a trusted third party : Ted.

A possible protocol would be the following. Alice sends $Cert(Alice_{pub}, Ted)$. This certificate provides Alice's identity and her public key. Bob replies with the certificate containing his own public key : $Cert(Bob_{pub}, Ted)$. At this point, they both know the other public key and could use it to send encrypted messages. Alice would send $E_p(Bob_{pub}, M1)$ and Bob would send $E_p(Alice_{pub}, M2)$. In practice, using public key encryption techniques to encrypt a large number of messages is inefficient because these cryptosystems require a large number of computations. It is more efficient to use secret key cryptosystems for most of the data and only use a public key cryptosystem to encrypt the random secret keys that will be used by the secret key encryption scheme.

2.8.4 Key exchange

When users want to communicate securely through a network, they need to exchange information such as the keys that will be used by an encryption algorithm even in the presence of an eavesdropper. The most widely used algorithm that allows two users to safely exchange an integer in the presence of an eavesdropper is the one proposed by Diffie and Hellman [DH1976]. It operates with (large) integers. Two of them are public, the modulus, p, which is prime and the base, g, which must be a primitive root of p. The communicating users select a random integer, a for Alice and b for Bob. The exchange starts as :

- Alice selects a random integer, a and sends $A = g^a \mod p$ to Bob
- Bob selects a random integer, b and sends $B = g^b \mod p$ to Alice
- From her knowledge of a and B, Alice can compute $Secret = B^a \mod p = (g^b \mod p)^a \mod p = g^{a \times b} \mod p$

- From is knowledge of b and A, Bob can compute $Secret = A^b \mod p = (g^a \mod p)^b \mod p = g^{a \times b} \mod p$

The security of this protocol relies on the difficulty of computing discrete logarithms, i.e. from the knowledge of A (resp. B), it is very difficult to extract $\log(A) = \log(g^a \mod p) = a$ (resp. $\log(B) = \log(g^b \mod p) = b$).

An example of the utilization of the Diffie-Hellman key exchange is shown below. Before starting the exchange, Alice and Bob agree on a modulus ($p = 23$) and a base ($g = 5$). These two numbers are public. They are typically part of the standard that defines the protocol that uses the key exchange.

- Alice chooses a secret integer : $a = 8$ and sends $A = g^a \mod p = 5^8 \mod 23 = 16$ to Bob
- Bob chooses a secret integer : $b = 13$ and sends $B = g^b \mod p = 5^{13} \mod 23 = 21$ to Alice
- Alice computes $S_A = B^a \mod p = 21^8 \mod 23 = 3$
- Bob computes $S_B = A^b \mod p = 16^{13} \mod 23 = 3$

Alice and Bob have agreed on the secret information 3 without having sent it explicitly through the network. If the integers used are large enough and have good properties, then even Eve who can capture all the messages sent by Alice and Bob cannot recover the secret key that they have exchanged. There is no formal proof of the security of the algorithm, but mathematicians have tried to solve similar problems with integers during centuries without finding an efficient algorithm. As long as the integers that are used are random and large enough, the only possible attack for Eve is to test all possible integers that could have been chosen by Alice and Bob. This is computationally very expensive. This algorithm is widely used in security protocols to agree on a secret key.

Unfortunately, the Diffie-Hellman key exchange alone cannot cope with man-in-the middle attacks. Consider Mallory who sits in the middle between Alice and Bob and can easily capture and modify their messages. The modulus and the base are public. They are thus known by Mallory as well. He could then operate as follows :

- Alice chooses a secret integer and sends $A = g^a \mod p$ to Mallory
- Mallory generates a secret integer, m and sends $M = g^m \mod p$ to Bob
- Bob chooses a secret integer and sends $B = g^b \mod p$ to Mallory
- Mallory computes $S_A = A^m \mod p$ and $S_B = B^m \mod p$
- Alice computes $S_A = M^a \mod p$ and uses this key to communicate with Mallory (acting as Bob)
- Bob computes $S_B = M^b \mod p$ and uses this key to communicate with Mallory (acting as Alice)

When Alice sends a message, she encrypts it with S_A. Mallory decrypts it with S_A and encrypts the plaintext with S_B. When Bob receives the message, he can decrypt it by using S_B.

To safely use the Diffie-Hellman key exchange, Alice and Bob must use an *authenticated* exchange. Some of the information sent by Alice or Bob must be signed with a public key known by the other user. In practice, it is often important for Alice to authenticate Bob. If Bob has a certificated signed by Ted, the authenticated key exchange could be organized as follows.

- Alice chooses a secret integer : a and sends $A = g^a \mod p$ to Bob
- Bob chooses a secret integer : b, computes $B = g^b \bmod p$ and sends $Cert(Bob, Bob_{pub}, Ted), E_p(Bob_{priv}, B)$ to Alice
- Alice checks the signature (with Bob_{pub}) and the certificate and computes $S_A = B^a \mod p$
- Bob computes $S_B = A^b \mod p$

This prevents the attack mentioned above since Mallory cannot create a fake certificate and cannot sign a value by using Bob's private key. Given the risk of man-in-the-middle attacks, the Diffie-Hellman key exchange mechanism should never be used without authentication.

CHAPTER

THREE

PART 2: PROTOCOLS

3.1 The application layer

Networked applications rely on the transport service. As explained earlier, there are two main types of transport services :

- the *connectionless* service
- the *connection-oriented* or *byte-stream* service

The connectionless service allows applications to easily exchange messages or Service Data Units. On the Internet, this service is provided by the UDP protocol that will be explained in the next chapter. The connectionless transport service on the Internet is unreliable, but is able to detect transmission errors. This implies that an application will not receive data that has been corrupted due to transmission errors.

The connectionless transport service allows networked application to exchange messages. Several networked applications may be running at the same time on a single host. Each of these applications must be able to exchange SDUs with remote applications. To enable these exchanges of SDUs, each networked application running on a host is identified by the following information :

- the *host* on which the application is running
- the *port number* on which the application *listens* for SDUs

On the Internet, the *port number* is an integer and the *host* is identified by its network address. There are two types of Internet addresses :

- *IP version 4* addresses that are 32 bits wide
- *IP version 6* addresses that are 128 bits wide

IPv4 addresses are usually represented by using a dotted decimal representation where each decimal number corresponds to one byte of the address, e.g. *203.0.113.56*. IPv6 addresses are usually represented as a set of hexadecimal numbers separated by semicolons, e.g. *2001:db8:3080:2:217:f2ff:fed6:65c0*. Today, most Internet hosts have one IPv4 address. A small fraction of them also have an IPv6 address. In the future, we can expect that more and more hosts will have IPv6 addresses and that some of them will not have an IPv4 address anymore. A host that only has an IPv4 address cannot communicate with a host having only an IPv6 address. The figure below illustrates two applications that are using the datagram service provided by UDP on hosts that are using IPv4 addresses.

Note: Textual representation of IPv6 addresses

It is sometimes necessary to write IPv6 addresses in text format, e.g. when manually configuring addresses or for documentation purposes. The preferred format for writing IPv6 addresses is *x:x:x:x:x:x:x:x*, where the *x* ‘s are hexadecimal digits representing the eight 16-bit parts of the address. Here are a few examples of IPv6 addresses :

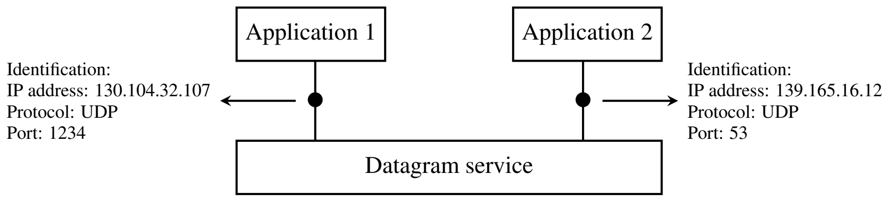


Fig. 1: The connectionless or datagram service

- *abcd:ef01:2345:6789:abcd:ef01:2345:6789*
- *2001:db8:0:0:8:800:200c:417a*
- *fe80:0:0:0:219:e3ff:fed7:1204*

IPv6 addresses often contain a long sequence of bits set to *0*. In this case, a compact notation has been defined. With this notation, *::* is used to indicate one or more groups of 16 bits blocks containing only bits set to *0*. For example,

- *2001:db8:0:0:8:800:200c:417a* is represented as *2001:db8::8:800:200c:417a*
- *ff01:0:0:0:0:0:0:101* is represented as *ff01::101*
- *0:0:0:0:0:0:0:1* is represented as *::1*
- *0:0:0:0:0:0:0:0* is represented as *::*

The second transport service is the connection-oriented service. On the Internet, this service is often called the *byte-stream service* as it creates a reliable byte stream between the two applications that are linked by a transport connection. Like the datagram service, the networked applications that use the byte-stream service are identified by the host on which they run and a port number. These hosts can be identified by an address or a name. The figure below illustrates two applications that are using the byte-stream service provided by the TCP protocol on IPv6 hosts. The byte stream service provided by TCP is reliable and bidirectional.

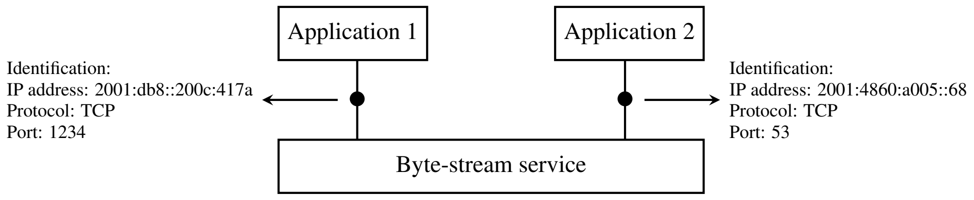


Fig. 2: The connection-oriented or byte-stream service

3.2 The Domain Name System

We have already explained the main principles that underlie the utilization of names on the Internet and their mapping to addresses in section *Naming and addressing*.

The last component of the Domain Name System is the DNS protocol. The original DNS protocol runs above both the datagram and the bytestream services. In practice, the datagram service is used when short queries and responses are exchanged, and the bytestream service is used when longer responses are expected. In this section, we first focus on the utilization of the DNS protocol above the datagram service. We will discuss later other recently proposed protocols to carry DNS information.

DNS messages are composed of five parts that are named sections in **RFC 1035**. The first three sections are mandatory and the last two sections are optional. The first section of a DNS message is its *Header*. It contains information about the message type and the content of the other sections. The second section contains the *Question* sent to the nameserver or resolver. The third section contains the *Answer* to the *Question*. When a client sends a DNS query, the *Answer* section is empty. The fourth section, named *Authority*, contains information about the servers that can provide an authoritative answer if required. The last section contains additional information that is supplied by the resolver or nameserver but was not requested in the question.

The header of DNS messages is composed of 12 bytes. The figure below presents its structure.

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      ID                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    QDCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ANCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    NSCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ARCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Fig. 3: The DNS header

The *Transaction ID* (transaction identifier) is a 16-bits random value chosen by the client. When a client sends a question to a DNS server, it remembers the question and its identifier. When a server returns an answer, it returns in the *Transaction ID* field the identifier chosen by the client. Thanks to this identifier, the client can match the received answer with the question that it sent.

The DNS header contains a series of flags. The *QR* flag is used to distinguish between queries and responses. It is set to *0* in DNS queries and *1* in DNS answers. The *Opcode* is used to specify the query type. For instance, a *standard query* is used when a client sends a *name* and the server returns the corresponding *data*. An update request is used when the client sends a *name* and new *data* and the server then updates its database.

The *AA* bit is set when the server that sent the response has *authority* for the domain name found in the question section. In the original DNS deployments, two types of servers were considered : *authoritative* servers and *non-authoritative* servers. The *authoritative* servers are managed by the system administrators responsible for a given domain. They always store the most recent information about a domain. *Non-authoritative* servers are servers or resolvers that store DNS information about external domains without being managed by the owners of a domain. They may thus provide answers that are out of date. From a security point of view, the *authoritative* bit is not an absolute indication about the validity of an answer. Securing the Domain Name System is a complex problem that was only addressed satisfactorily recently by the utilization of cryptographic signatures in the DNSSEC extensions to DNS described in **RFC 4033**.

The *RD* (recursion desired) bit is set by a client when it sends a query to a resolver. Such a query is said to be *recursive* because the resolver will recursively traverse the DNS hierarchy to retrieve the answer on behalf of the client. In the past, all resolvers were configured to perform recursive queries on behalf of any Internet host. However, this exposes the resolvers to several security risks. The simplest one is that the resolver could become overloaded by having too many recursive queries to process. Most resolvers[1] only allow recursive queries from clients belonging to

[1] Some DNS resolvers allow any host to send queries. Google operates a public DNS resolver at addresses *2001:4860:4860::8888* and *2001:4860:4860::8844*. Other companies provide similar services.

their company or network and discard all other recursive queries. The *RA* bit indicates whether the server supports recursion. The *RCODE* is used to distinguish between different types of errors. See **RFC 1035** for additional details. The last four fields indicate the size of the *Question*, *Answer*, *Authority* and *Additional* sections of the DNS message.

The last four sections of the DNS message contain *Resource Records* (RR). All RRs have the same top level format shown in the figure below.

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
/                                               /
/                      NAME                     /
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      TYPE                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     CLASS                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      TTL                      |
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                   RDLENGTH                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--|
/                     RDATA                     /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Fig. 4: DNS Resource Records

In a *Resource Record* (*RR*), the *Name* indicates the name of the node to which this resource record pertains. The two-)bytes *Type* field indicates the type of resource record. The *Class* field was used to support the utilization of the DNS in other environments than the Internet. The *IN Class* refers to Internet names.

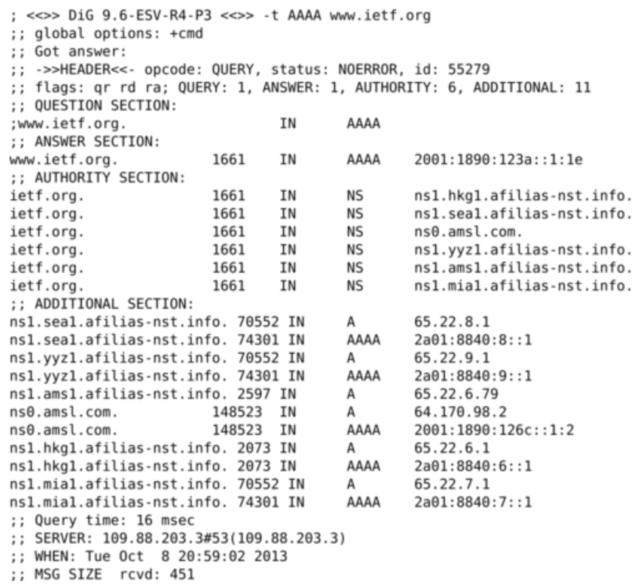
The *TTL* field indicates the lifetime of the *Resource Record* in seconds. This field is set by the server that returns an answer and indicates for how long a client or a resolver can store the *Resource Record* inside its cache. A long *TTL* indicates a stable *RR*. Some companies use short *TTL* values for mobile hosts and also for popular servers. For example, a web hosting company that wants to spread the load over a pool of hundred servers can configure its nameservers to return different answers to different clients. If each answer has a small *TTL*, the clients will be forced to send DNS queries regularly. The nameserver will reply to these queries by supplying the address of the less loaded server.

The *RDLength* field is the length of the *RData* field that contains the information of the type specified in the *Type* field.

Several types of DNS RR are used in practice. The *A* type encodes the IPv4 address that corresponds to the specified name. The *AAAA* type encodes the IPv6 address that corresponds to the specified name. A *NS* record contains the name of the DNS server that is responsible for a given domain. For example, a query for the *AAAA* record associated to the *www.ietf.org* name returned the following answer.

This answer contains several pieces of information. First, the name *www.ietf.org* is associated to IP address *2001:1890:123a::1:1e*. Second, the *ietf.org* domain is managed by six different nameservers. Five of these nameservers are reachable via IPv4 and IPv6.

CNAME (or canonical names) are used to define aliases. For example *www.example.com* could be a *CNAME* for *pc12.example.com* that is the actual name of the server on which the web server for *www.example.com* runs.



```
; <<>> DiG 9.6-ESV-R4-P3 <<>> -t AAAA www.ietf.org
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 55279
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 6, ADDITIONAL: 11
;; QUESTION SECTION:
;www.ietf.org.                  IN      AAAA
;; ANSWER SECTION:
www.ietf.org.           1661    IN      AAAA    2001:1890:123a::1:1e
;; AUTHORITY SECTION:
ietf.org.               1661    IN      NS      ns1.hkg1.afilias-nst.info.
ietf.org.               1661    IN      NS      ns1.sea1.afilias-nst.info.
ietf.org.               1661    IN      NS      ns0.amsl.com.
ietf.org.               1661    IN      NS      ns1.yyz1.afilias-nst.info.
ietf.org.               1661    IN      NS      ns1.ams1.afilias-nst.info.
ietf.org.               1661    IN      NS      ns1.mia1.afilias-nst.info.
;; ADDITIONAL SECTION:
ns1.sea1.afilias-nst.info. 70552 IN     A       65.22.8.1
ns1.sea1.afilias-nst.info. 74301 IN     AAAA    2a01:8840:8::1
ns1.yyz1.afilias-nst.info. 70552 IN     A       65.22.9.1
ns1.yyz1.afilias-nst.info. 74301 IN     AAAA    2a01:8840:9::1
ns1.ams1.afilias-nst.info. 2597 IN      A       65.22.6.79
ns0.amsl.com.           148523  IN      A       64.170.98.2
ns0.amsl.com.           148523  IN      AAAA    2001:1890:126c::1:2
ns1.hkg1.afilias-nst.info. 2073 IN      A       65.22.6.1
ns1.hkg1.afilias-nst.info. 2073 IN      AAAA    2a01:8840:6::1
ns1.mia1.afilias-nst.info. 70552 IN     A       65.22.7.1
ns1.mia1.afilias-nst.info. 74301 IN     AAAA    2a01:8840:7::1
;; Query time: 16 msec
;; SERVER: 109.88.203.3#53(109.88.203.3)
;; WHEN: Tue Oct  8 20:59:02 2013
;; MSG SIZE  rcvd: 451
```

Fig. 5: Query for the *AAAA* record of *www.ietf.org*

Note: Reverse DNS

The DNS is mainly used to find the address that corresponds to a given name. However, it is sometimes useful to obtain the name that corresponds to an IP address. This done by using the *PTR* (*pointer*) *RR*. The *RData* part of a *PTR RR* contains the name while the *Name* part of the *RR* contains the IP address encoded in the *in-addr.arpa* domain. IPv4 addresses are encoded in the *in-addr.arpa* by reversing the four digits that compose the dotted decimal representation of the address. For example, consider IPv4 address *192.0.2.11*. The hostname associated to this address can be found by requesting the *PTR RR* that corresponds to *11.2.0.192.in-addr.arpa*. A similar solution is used to support IPv6 addresses **RFC 3596**, but slightly more complex given the length of the IPv6 addresses. For example, consider IPv6 address *2001:1890:123a::1:1e*. To obtain the name that corresponds to this address, we need first to convert it in a reverse dotted decimal notation : *e.1.0.0.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.a.3.2.1.0.9.8.1.1.0.0.2*. In this notation, each character between dots corresponds to one nibble, i.e. four bits. The low-order byte (*e*) appears first and the high order (*2*) last. To obtain the name that corresponds to this address, one needs to append the *ip6.arpa* domain name and query for *e.1.0.0.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.a.3.2.1.0.9.8.1.1.0.0.2.ip6.arpa*. In practice, tools and libraries do the conversion automatically and the user does not need to worry about it.

An important point to note regarding the Domain Name System it that it is extensible. Thanks to the *Type* and *RDLength* fields, the format of the Resource Records can easily be extended. Furthermore, a DNS implementation that receives a new Resource Record that it does not understand can ignore the record while still being able to process the other parts of the message. This allows, for example, a DNS server that only supports IPv6 to safely ignore the IPv4 addresses listed in the DNS reply for *www.ietf.org* while still being able to correctly parse the Resource Records that it understands. This allowed the Domain Name System to evolve over the years while still preserving the backward compatibility with already deployed DNS implementations.

3.3 Electronic mail

Electronic mail, or email, is a very popular application in computer networks such as the Internet. Email appeared in the early 1970s and allows users to exchange text based messages. Initially, it was mainly used to exchange short messages, but over the years its usage has grown. It is now not only used to exchange small, but also long messages that can be composed of several parts as we will see later.

Before looking at the details of Internet email, let us consider a simple scenario illustrated in the figure below, where Alice sends an email to Bob. Alice prepares her email by using an email clients and sends it to her email server. Alice's email server extracts Bob's address from the email and delivers the message to Bob's server. Bob retrieves Alice's message on his server and reads it by using his favorite email client or through his webmail interface.

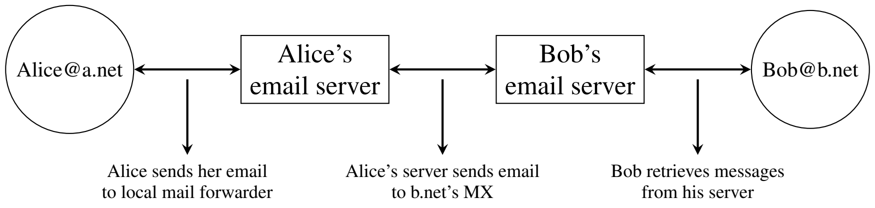

Alice@a.net
Alice's email server
Bob's email server
Bob@b.net
Alice sends her email to local mail forwarder
Alice's server sends email to b.net's MX
Bob retrieves messages from his server

Fig. 6: Simplified architecture of the Internet email

The email system that we consider in this book is composed of four components :

- a message format, that defines how valid email messages are encoded
- protocols, that allow hosts and servers to exchange email messages
- client software, that allows users to easily create and read email messages
- software, that allows servers to efficiently exchange email messages

We will first discuss the format of email messages followed by the protocols that are used on today's Internet to exchange and retrieve emails. Other email systems have been developed in the past [Bush1993] [Genilloud1990] [GC2000], but today most email solutions have migrated to the Internet email. Information about the software that is used to compose and deliver emails may be found on wikipedia among others, for both email clients and email servers. More detailed information about the full Internet Mail Architecture may be found in **RFC 5598**.

Email messages, like postal mail, are composed of two parts :

- a *header* that plays the same role as the letterhead in regular mail. It contains metadata about the message.
- the *body* that contains the message itself.

Email messages are entirely composed of lines of ASCII characters. Each line can contain up to 998 characters and is terminated by the *CR* and *LF* control characters **RFC 5322**. The lines that compose the *header* appear before the message *body*. An empty line, containing only the *CR* and *LF* characters, marks the end of the *header*. This is illustrated in the figure below.

The email header contains several lines that all begin with a keyword followed by a colon and additional information. The format of email messages and the different types of header lines are defined in **RFC 5322**. Two of these header lines are mandatory and must appear in all email messages :

- The sender address. This header line starts with *From:*. This contains the (optional) name of the sender followed by its email address between < and >. Email addresses are always composed of a username followed by the @ sign and a domain name.
- The date. This header line starts with *Date:*. **RFC 5322** precisely defines the format used to encode a date.

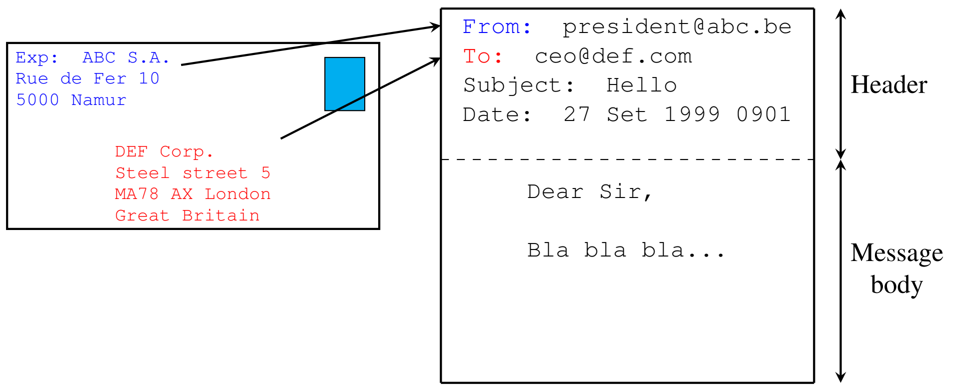


Fig. 7: The structure of email messages

Other header lines appear in most email messages. The *Subject:* header line allows the sender to indicate the topic discussed in the email. Three types of header lines can be used to specify the recipients of a message :

- the *To:* header line contains the email addresses of the primary recipients of the message[1]. Several addresses can be separated by using commas.
- the *cc:* header line is used by the sender to provide a list of email addresses that must receive a carbon copy of the message. Several addresses can be listed in this header line, separated by commas. All recipients of the email message receive the *To:* and *cc:* header lines.
- the *bcc:* header line is used by the sender to provide a list of comma separated email addresses that must receive a blind carbon copy of the message. The *bcc:* header line is not delivered to the recipients of the email message.

A simple email message containing the *From:*, *To:*, *Subject:* and *Date:* header lines and two lines of body is shown below.

```
From: Bob Smith <Bob@machine.example>
To: Alice Doe <alice@example.net>, Alice Smith <Alice@machine.example>
Subject: Hello
Date: Mon, 8 Mar 2010 19:55:06 -0600

This is the "Hello world" of email messages.
This is the second line of the body
```

Note the empty line after the *Date:* header line; this empty line contains only the *CR* and *LF* characters, and marks the boundary between the header and the body of the message.

Several other optional header lines are defined in **RFC 5322** and elsewhere[2]. Furthermore, many email clients and servers define their own header lines starting from *X-*. Several of the optional header lines defined in **RFC 5322** are worth being discussed here :

- the *Message-Id:* header line is used to associate a "unique" identifier to each email. Email identifiers are usually structured like *string@domain* where *string* is a unique character string or sequence number chosen by the sender of the email and *domain* the domain name of the sender. Since domain names are unique, a host can generate globally unique message identifiers concatenating a locally unique identifier with its domain name.
- the *In-reply-to:* header line is used when a message was created in reply to a previous message. In this case, the end of the *In-reply-to:* line contains the identifier of the original message.

[1] It could be surprising that the *To:* is not mandatory inside an email message. While most email messages will contain this header line an email that does not contain a *To:* header line and that relies on the *bcc:* to specify the recipient is valid as well.

[2] The list of all standard email header lines may be found at http://www.iana.org/assignments/message-headers/message-header-index.html

- the *Received:* header line is used when an email message is processed by several servers before reaching its destination. Each intermediate email server adds a *Received:* header line. These header lines are useful to debug problems in delivering email messages.

The figure below shows the header lines of one email message. The message originated at a host named *wira.firstpr.com.au* and was received by *smtp3.sgsi.ucl.ac.be*. The *Received:* lines have been wrapped for readability.

```
Received: from smtp3.sgsi.ucl.ac.be (Unknown [10.1.5.3])
    by mmp.sipr-dc.ucl.ac.be
    (Sun Java(tm) System Messaging Server 7u3-15.01 64bit (built Feb 12 2010))
    with ESMTP id <0KYY00L85LI5JLE0@mmp.sipr-dc.ucl.ac.be>; Mon,
    08 Mar 2010 11:37:17 +0100 (CET)
Received: from mail.ietf.org (mail.ietf.org [64.170.98.32])
    by smtp3.sgsi.ucl.ac.be (Postfix) with ESMTP id B92351C60D7; Mon,
    08 Mar 2010 11:36:51 +0100 (CET)
Received: from [127.0.0.1] (localhost [127.0.0.1])      by core3.amsl.com (Postfix)
    with ESMTP id F066A3A68B9; Mon, 08 Mar 2010 02:36:38 -0800 (PST)
Received: from localhost (localhost [127.0.0.1])       by core3.amsl.com (Postfix)
    with ESMTP id A1E6C3A681B  for <rrg@core3.amsl.com>; Mon,
    08 Mar 2010 02:36:37 -0800 (PST)
Received: from mail.ietf.org ([64.170.98.32])
    by localhost (core3.amsl.com [127.0.0.1]) (amavisd-new, port 10024)
    with ESMTP id erw8ih2v8VQa for <rrg@core3.amsl.com>; Mon,
    08 Mar 2010 02:36:36 -0800 (PST)
Received: from gair.firstpr.com.au (gair.firstpr.com.au [150.101.162.123])
    by core3.amsl.com (Postfix) with ESMTP id 03E893A67ED      for <rrg@irtf.org>;␣
↪Mon,
    08 Mar 2010 02:36:35 -0800 (PST)
Received: from [10.0.0.6] (wira.firstpr.com.au [10.0.0.6])
    by gair.firstpr.com.au (Postfix) with ESMTP id D0A49175B63; Mon,
    08 Mar 2010 21:36:37 +1100 (EST)
Date: Mon, 08 Mar 2010 21:36:38 +1100
From: Robin Whittle <rw@firstpr.com.au>
Subject: Re: [rrg] Recommendation and what happens next
In-reply-to: <C7B9C21A.4FAB%tony.li@tony.li>
To: RRG <rrg@irtf.org>
Message-id: <4B94D336.7030504@firstpr.com.au>

Message content removed
```

Initially, email was used to exchange small messages of ASCII text between computer scientists. However, with the growth of the Internet, supporting only ASCII text became a severe limitation for two reasons. First of all, non-English speakers wanted to write emails in their native language that often required more characters than those of the ASCII character table. Second, many users wanted to send other content than just ASCII text by email such as binary files, images or sound.

To solve this problem, the IETF developed the Multipurpose Internet Mail Extensions (*MIME*). These extensions were carefully designed to allow Internet email to carry non-ASCII characters and binary files without breaking the email servers that were deployed at that time. This requirement for backward compatibility forced the MIME designers to develop extensions to the existing email message format **RFC 822** instead of defining a completely new format that would have been better suited to support the new types of emails.

RFC 2045 defines three new types of header lines to support MIME :

- The *MIME-Version:* header line indicates the version of the MIME specification that was used to encode the email message. The current version of MIME is 1.0. Other versions of MIME may be defined in the future. Thanks to this header line, the software that processes email messages will be able to adapt to the MIME version used to encode the message. Messages that do not contain this header are supposed to be formatted according to the original **RFC 822** specification.

- The *Content-Type:* header line indicates the type of data that is carried inside the message (see below).
- The *Content-Transfer-Encoding:* header line is used to specify how the message has been encoded. When MIME was designed, some email servers were only able to process messages containing characters encoded using the 7 bits ASCII character set. MIME allows the utilization of other character encodings.

Inside the email header, the *Content-Type:* header line indicates how the MIME email message is structured. **RFC 2046** defines the utilization of this header line. The two most common structures for MIME messages are :

- *Content-Type: multipart/mixed*. This header line indicates that the MIME message contains several independent parts. For example, such a message may contain a part in plain text and a binary file.
- *Content-Type: multipart/alternative*. This header line indicates that the MIME message contains several representations of the same information. For example, a *multipart/alternative* message may contain both a plain text and an HTML version of the same text.

To support these two types of MIME messages, the recipient of a message must be able to extract the different parts from the message. In **RFC 822**, an empty line was used to separate the header lines from the body. Using an empty line to separate the different parts of an email body would be difficult as the body of email messages often contains one or more empty lines. Another possible option would be to define a special line, e.g. **-LAST_LINE-** to mark the boundary between two parts of a MIME message. Unfortunately, this is not possible as some emails may contain this string in their body (e.g. emails sent to students to explain the format of MIME messages). To solve this problem, the *Content-Type:* header line contains a second parameter that specifies the string that has been used by the sender of the MIME message to delineate the different parts. In practice, this string is often chosen randomly by the mail client.

The email message below, copied from **RFC 2046** shows a MIME message containing two parts that are both in plain text and encoded using the ASCII character set. The string *simple boundary* is defined in the *Content-Type:* header as the marker for the boundary between two successive parts. Another example of MIME messages may be found in **RFC 2046**.

```
Date: Mon, 20 Sep 1999 16:33:16 +0200
From: Nathaniel Borenstein <nsb@bellcore.com>
To: Ned Freed <ned@innosoft.com>
Subject: Test
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="simple boundary"

preamble, to be ignored

--simple boundary
Content-Type: text/plain; charset=us-ascii

First part

--simple boundary
Content-Type: text/plain; charset=us-ascii

Second part
--simple boundary
```

The *Content-Type:* header can also be used inside a MIME part. In this case, it indicates the type of data placed in this part. Each data type is specified as a type followed by a subtype. A detailed description may be found in **RFC 2046**. Some of the most popular *Content-Type:* header lines are :

- *text*. The message part contains information in textual format. There are several subtypes : *text/plain* for regular ASCII text, *text/html* defined in **RFC 2854** for documents in *HTML* format or the *text/enriched* format defined in **RFC 1896**. The *Content-Type:* header line may contain a second parameter that specifies the character set used to encode the text. *charset=us-ascii* is the standard ASCII character table. Other frequent character sets include *charset=UTF8* or *charset=iso-8859-1*. The list of standard character sets is maintained by *IANA*.

- *image*. The message part contains a binary representation of an image. The subtype indicates the format of the image such as gif, jpg or png.
- *audio*. The message part contains an audio clip. The subtype indicates the format of the audio clip like wav or mp3.
- *video*. The message part contains a video clip. The subtype indicates the format of the video clip like avi or mp4.
- *application*. The message part contains binary information that was produced by the particular application listed as the subtype. Email clients use the subtype to launch the application that is able to decode the received binary information.

Note: From ASCII to Unicode

The first computers used different techniques to represent characters in memory and on disk. During the 1960s, computers began to exchange information via tape or telephone lines. Unfortunately, each vendor had its own proprietary character set and exchanging data between computers from different vendors was often difficult. The 7 bits ASCII character table **RFC 20** was adopted by several vendors and by many Internet protocols. However, ASCII became a problem with the internationalization of the Internet and the desire of more and more users to use character sets that support their own written language. A first attempt at solving this problem was the definition of the ISO-8859 character sets by *ISO*. This family of standards specified various character sets that allowed the representation of many European written languages by using 8 bits characters. Unfortunately, an 8-bits character set is not sufficient to support some widely used languages, such as those used in Asian countries. Fortunately, at the end of the 1980s, several computer scientists proposed to develop a standard that supports all written languages used on Earth today. The Unicode standard [Unicode] has now been adopted by most computer and software vendors. For example, Java always uses Unicode to manipulate characters, Python can handle both ASCII and Unicode characters. Internet applications are slowly moving towards complete support for the Unicode character sets, but moving from ASCII to Unicode is an important change that can have a huge impact on current deployed implementations. See, for example, the work to completely internationalize email **RFC 4952** and domain names **RFC 5890**.

The last MIME header line is *Content-Transfer-Encoding:*. This header line is used after the *Content-Type:* header line, within a message part, and specifies how the message part has been encoded. The default encoding is to use 7 bits ASCII. The most frequent encodings are *quoted-printable* and *Base64*. Both support encoding a sequence of bytes into a set of ASCII lines that can be safely transmitted by email servers. *quoted-printable* is defined in **RFC 2045**. We briefly describe *base64* which is defined in **RFC 2045** and **RFC 4648**.

Base64 divides the sequence of bytes to be encoded into groups of three bytes (with the last group possibly being partially filled). Each group of three bytes is then divided into four six-bit fields and each six bit field is encoded as a character from the table below.

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w		
15	P	32	g	49	x		
16	Q	33	h	50	y		

The example below, from **RFC 4648**, illustrates the *Base64* encoding.

Input data	*0x14fb9c03d97e*
8-bit	00010100 11111011 10011100 00000011 11011001 01111110
6-bit	000101 001111 101110 011100 000000 111101 100101 111110
Decimal	5 15 46 28 0 61 37 62
Encoding	F P u c A 9 l +

The last point to be discussed about *base64* is what happens when the length of the sequence of bytes to be encoded is not a multiple of three. In this case, the last group of bytes may contain one or two bytes instead of three. *Base64* reserves the = character as a padding character. This character is used once when the last group contains two bytes and twice when it contains one byte as illustrated by the two examples below.

Input data	0x14
8-bit	00010100
6-bit	000101 000000
Decimal	5 0
Encoding	F A = =

Input data	0x14b9
8-bit	00010100 11111011
6-bit	000101 001111 101100
Decimal	5 15 44
Encoding	F P s =

Now that we have explained the format of the email messages, we can discuss how these messages can be exchanged through the Internet. The figure below illustrates the protocols that are used when *Alice* sends an email message to *Bob*. *Alice* prepares her email with an email client or on a webmail interface. To send her email to *Bob*, *Alice*'s client will use the Simple Mail Transfer Protocol (*SMTP*) to deliver her message to her SMTP server. *Alice*'s email client is configured with the name of the default SMTP server for her domain. There is usually at least one SMTP server per domain. To deliver the message, *Alice*'s SMTP server must find the SMTP server that contains *Bob*'s mailbox. This can be done by using the Mail eXchange (MX) records of the DNS. A set of MX records can be associated to each domain. Each MX record contains a numerical preference and the fully qualified domain name of a SMTP server that is able to deliver email messages destined to all valid email addresses of this domain. The DNS can return several MX records for a given domain. In this case, the server with the lowest numerical preference is used first **RFC 2821**. If this server is not reachable, the second most preferred server is used etc. *Bob*'s SMTP server will store the message sent by *Alice* until *Bob* retrieves it using a webmail interface or protocols such as the Post Office Protocol (*POP*) or the Internet Message Access Protocol (*IMAP*).

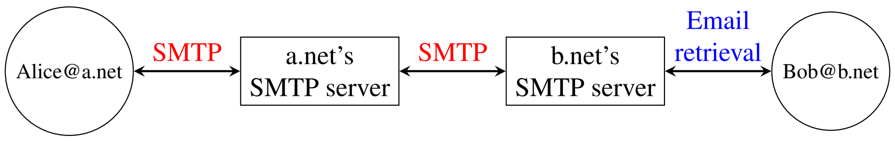


Fig. 8: Email delivery protocols

3.3.1 The Simple Mail Transfer Protocol

The Simple Mail Transfer Protocol (*SMTP*) defined in **RFC 5321** is a client-server protocol. The SMTP specification distinguishes between five types of processes involved in the delivery of email messages. Email messages are composed on a Mail User Agent (MUA). The MUA is usually either an email client or a webmail. The MUA sends the email message to a Mail Submission Agent (MSA). The MSA processes the received email and forwards it to the Mail Transmission Agent (MTA). The MTA is responsible for the transmission of the email, directly or via intermediate MTAs to the MTA of the destination domain. This destination MTA will then forward the message to the Mail Delivery Agent (MDA) where it will be accessed by the recipient's MUA. SMTP is used for the interactions between MUA and MSA[3], MSA-MTA and MTA-MTA.

SMTP is a text-based protocol like many other application-layer protocols on the Internet. It relies on the byte-stream service. Servers listen on port *25*. Clients send commands that are each composed of one line of ASCII text terminated by *CR+LF*. Servers reply by sending ASCII lines that contain a three digit numerical error/success code and optional comments.

The SMTP protocol, like most text-based protocols, is specified as a *BNF*. The full BNF is defined in **RFC 5321**. The main SMTP commands are defined by the BNF rules shown in the figure below.

```
ehlo = "EHLO" SP Domain CRLF
mail = "MAIL FROM:" Path CRLF
rcpt = "RCPT TO:" ( "<Postmaster@" Domain ">" / "<Postmaster>" / Path ) CRLF
data = "DATA" CRLF
quit = "QUIT" CRLF
Path           = "<" Mailbox ">"
Domain         = sub-domain *("." sub-domain)
sub-domain     = Let-dig [Ldh-str]
Let-dig        = ALPHA / DIGIT
Ldh-str        = *( ALPHA / DIGIT / "-" ) Let-dig
Mailbox        = Local-part "@" Domain
Local-part     = Dot-string
Dot-string     = Atom *("."  Atom)
Atom           = 1*atext
```

Fig. 9: BNF specification of the SMTP commands

In this BNF, *atext* corresponds to printable ASCII characters. This BNF rule is defined in **RFC 5322**. The five main

[3] During the last years, many Internet Service Providers, campus and enterprise networks have deployed SMTP extensions **RFC 4954** on their MSAs. These extensions force the MUAs to be authenticated before the MSA accepts an email message from the MUA.

commands are *EHLO*[4], *MAIL FROM:*, *RCPT TO:*, *DATA* and *QUIT*. *Postmaster* is the alias of the system administrator who is responsible for a given domain or SMTP server. All domains must have a *Postmaster* alias.

The SMTP responses are defined by the BNF shown in the figure below.

```
Greeting       = "220 " Domain [ SP textstring ] CRLF
textstring     = 1*atext
Reply-line     = *( Reply-code "-" [ textstring ] CRLF )
                 Reply-code [ SP textstring ] CRLF
Reply-code     = %x32-35 %x30-35 %x30-39
```

Fig. 10: BNF specification of the SMTP responses

SMTP servers use structured reply codes containing three digits and an optional comment. The first digit of the reply code indicates whether the command was successful or not. A reply code of *2xy* indicates that the command has been accepted. A reply code of *3xy* indicates that the command has been accepted, but additional information from the client is expected. A reply code of *4xy* indicates a transient negative reply. This means that for some reason, which is indicated by either the other digits or the comment, the command cannot be processed immediately, but there is some hope that the problem will only be transient. This is basically telling the client to try the same command again later. In contrast, a reply code of *5xy* indicates a permanent failure or error. In this case, it is useless for the client to retry the same command later. Other application layer protocols such as FTP **RFC 959** or HTTP **RFC 2616** use a similar structure for their reply codes. Additional details about the other reply codes may be found in **RFC 5321**.

Examples of SMTP reply codes include the following :

```
220  <domain> Service ready
221  <domain> Service closing transmission channel
250  Requested mail action okay, completed
354  Start mail input; end with <CRLF>.<CRLF>
421  <domain> Service not available, closing transmission channel
450  Requested mail action not taken: mailbox unavailable
452  Requested action not taken: insufficient system storage
500  Syntax error, command unrecognized
501  Syntax error in parameters or arguments
502  Command not implemented
503  Bad sequence of commands
550  Requested action not taken: mailbox unavailable
```

Reply code *220* is used by the server as the first message when it agrees to interact with the client. Reply code *221* is sent by the server before closing the underlying transport connection. Reply code *250* is the standard positive reply that indicates the success of the previous command. Reply code *354* indicates that the client can start transmitting its email message. Reply code *421* is returned when there is a problem (e.g. lack of memory/disk resources) that prevents the server from accepting the transport connection. Reply codes *450* and *452* indicate that the destination mailbox is temporarily unavailable, for various reasons, while reply code *550* indicates that the mailbox does not exist or cannot be used for policy reasons. The *500* to *503* reply codes correspond to errors in the commands sent by the client. The *503* reply code would be sent by the server when the client sends commands in an incorrect order (e.g. the client tries to send an email before providing the destination address of the message).

The transfer of an email message is performed in three phases. During the first phase, the client opens a transport connection with the server. Once the connection has been established, the client and the server exchange greetings messages (*EHLO* command). Most servers insist on receiving valid greeting messages and some of them drop the underlying transport connection if they do not receive a valid greeting. Once the greetings have been exchanged, the email transfer phase can start. During this phase, the client transfers one or more email messages by indicating

[4] The first versions of SMTP used *HELO* as the first command sent by a client to a SMTP server. When SMTP was extended to support newer features such as 8 bits characters, it was necessary to allow a server to recognize whether it was interacting with a client that supported the extensions or not. *EHLO* became mandatory with the publication of **RFC 2821**.

the email address of the sender (*MAIL FROM:* command), the email address of the recipient (*RCPT TO:* command) followed by the headers and the body of the email message (*DATA* command). Once the client has finished sending all its queued email messages to the SMTP server, it terminates the SMTP association (*QUIT* command).

A successful transfer of an email message is shown below

```
S: 220 smtp.example.com ESMTP MTA information
C: EHLO mta.example.org
S: 250 Hello mta.example.org, glad to meet you
C: MAIL FROM:<alice@example.org>
S: 250 Ok
C: RCPT TO:<bob@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: From: "Alice Doe" <alice@example.org>
C: To: Bob Smith <bob@example.com>
C: Date: Mon, 9 Mar 2010 18:22:32 +0100
C: Subject: Hello
C:
C: Hello Bob
C: This is a small message containing 4 lines of text.
C: Best regards,
C: Alice
C: .
S: 250 Ok: queued as 12345
C: QUIT
S: 221 Bye
```

In the example above, the MTA running on *mta.example.org* opens a TCP connection to the SMTP server on host *smtp.example.com*. The lines prefixed with *S:* (resp. *C:*) are the responses sent by the server (resp. the commands sent by the client). The server sends its greetings as soon as the TCP connection has been established. The client then sends the *EHLO* command with its fully qualified domain name. The server replies with reply-code *250* and sends its greetings. The SMTP association can now be used to exchange an email.

To send an email, the client must first provide the address of the recipient with *RCPT TO:*. Then it uses the *MAIL FROM:* with the address of the sender. Both the recipient and the sender are accepted by the server. The client can now issue the *DATA* command to start the transfer of the email message. After having received the *354* reply code, the client sends the headers and the body of its email message. The client indicates the end of the message by sending a line containing only the . (dot) character[5]. The server confirms that the email message has been queued for delivery or transmission with a reply code of *250*. The client issues the *QUIT* command to close the session and the server confirms with reply-code *221*, before closing the TCP connection.

Note: Open SMTP relays and spam

Since its creation in 1971, email has been a very useful tool that is used by many users to exchange lots of information. In the early days, all SMTP servers were open and anyone could use them to forward emails towards their final destination. Unfortunately, over the years, some unscrupulous users have found ways to use email for marketing purposes or to send malware. The first documented abuse of email for marketing purposes occurred in 1978 when a marketer who worked for a computer vendor sent a marketing email to many ARPANET users. At that time, the ARPANET could only be used for research purposes and this was an abuse of the acceptable use policy. Unfortunately, given the extremely low cost of sending emails, the problem of unsolicited emails has not stopped. Unsolicited emails are now called spam and a study carried out by ENISA in 2009 reveals that 95% of email was spam and this number seems to continue to grow. This places a burden on the email infrastructure of Internet Service Providers and large companies that need to process many useless messages.

[5] This implies that a valid email message cannot contain a line with one dot followed by *CR* and *LF*. If a user types such a line in an email, his email client will automatically add a space character before or after the dot when sending the message over SMTP.

Given the amount of spam messages, SMTP servers are no longer open **RFC 5068**. Several extensions to SMTP have been developed in recent years to deal with this problem. For example, the SMTP authentication scheme defined in **RFC 4954** can be used by an SMTP server to authenticate a client. Several techniques have also been proposed to allow SMTP servers to *authenticate* the messages sent by their users **RFC 4870 RFC 4871** .

3.3.2 The Post Office Protocol

When the first versions of SMTP were designed, the Internet was composed of minicomputers that were used by an entire university department or research lab. These minicomputers were used by many users at the same time. Email was mainly used to send messages from a user on a given host to another user on a remote host. At that time, SMTP was the only protocol involved in the delivery of the emails as all hosts attached to the network were running an SMTP server. On such hosts, an email destined to local users was delivered by placing the email in a special directory or file owned by the user. However, the introduction of personal computers in the 1980s changed this environment. Initially, users of these personal computers used applications such as *telnet* to open a remote session on the local *minicomputer* to read their email. This was not user-friendly. A better solution appeared with the development of user friendly email client applications on personal computers. Several protocols were designed to allow these client applications to retrieve the email messages destined to a user from his/her server. Two of these protocols became popular and are still used today. The Post Office Protocol (POP), defined in **RFC 1939**, is the simplest one. It allows a client to download all the messages destined to a given user from his/her email server. We describe POP briefly in this section. The second protocol is the Internet Message Access Protocol (IMAP), defined in **RFC 3501**. IMAP is more powerful, but also more complex than POP. IMAP was designed to allow client applications to efficiently access, in real-time, to messages stored in various folders on servers. IMAP assumes that all the messages of a given user are stored on a server and provides the functions that are necessary to search, download, delete or filter messages.

POP is another example of a simple line-based protocol. POP runs above the bytestream service. A POP server usually listens to port 110. A POP session is composed of three parts : an *authorisation* phase during which the server verifies the client's credential, a *transaction* phase during which the client downloads messages and an *update* phase that concludes the session. The client sends commands and the server replies are prefixed by *+OK* to indicate a successful command or by *-ERR* to indicate errors.

When a client opens a transport connection with the POP server, the latter sends as banner an ASCII-line starting with *+OK*. The POP session is at that time in the *authorisation* phase. In this phase, the client can send its username (resp. password) with the *USER* (resp. *PASS*) command. The server replies with *+OK* if the username (resp. password) is valid and *-ERR* otherwise.

Once the username and password have been validated, the POP session enters in the *transaction* phase. In this phase, the client can issue several commands. The *STAT* command is used to retrieve the status of the server. Upon reception of this command, the server replies with a line that contains *+OK* followed by the number of messages in the mailbox and the total size of the mailbox in bytes. The *RETR* command, followed by a space and an integer, is used to retrieve the nth message of the mailbox. The *DELE* command is used to mark for deletion the nth message of the mailbox.

Once the client has retrieved and possibly deleted the emails contained in the mailbox, it must issue the *QUIT* command. This command terminates the POP session and allows the server to delete all the messages that have been marked for deletion by using the *DELE* command.

The figure below provides a simple POP session. All lines prefixed with *C:* (resp. *S:*) are sent by the client (resp. server).

```
S:    +OK POP3 server ready
C:    USER alice
S:    +OK
C     PASS 12345pass
S:    +OK alice's maildrop has 2 messages (620 octets)
C:    STAT
S:    +OK 2 620
```

(continues on next page)

(continued from previous page)

```
C:    LIST
S:    +OK 2 messages (620 octets)
S:    1 120
S:    2 500
S:    .
C:    RETR 1
S:    +OK 120 octets
S:    <the POP3 server sends message 1>
S:    .
C:    DELE 1
S:    +OK message 1 deleted
C:    QUIT
S:    +OK POP3 server signing off (1 message left)
```

In this example, a POP client contacts a POP server on behalf of the user named *alice*. Note that in this example, Alice's password is sent in clear by the client. This implies that if someone is able to capture the packets sent by Alice, he will know Alice's password[6]. Then Alice's client issues the *STAT* command to know the number of messages that are stored in her mailbox. It then retrieves and deletes the first message of the mailbox.

3.4 The HyperText Transfer Protocol

In the early days, the Internet was mainly used for remote terminal access with telnet, email and file transfer. The default file transfer protocol, *FTP*, defined in **RFC 959** was widely used. *FTP* clients and servers are still included in some operating systems.

Many *FTP* clients offered a user interface similar to a Unix shell and allowed clients to browse the file system on the server and to send and retrieve files. *FTP* servers can be configured in two modes :

- *authenticated* : in this mode, the ftp server only accepts users with a valid user name and password. Once authenticated, they can access the files and directories according to their permissions
- *anonymous* : in this mode, clients supply the *anonymous* user identifier and their email address as password. These clients are granted access to a special zone of the file system that only contains public files.

FTP was very popular in the 1990s and early 2000s, but today it has mostly been superseded by more recent protocols. Authenticated access to files is mainly done by using the Secure Shell (ssh) protocol defined in **RFC 4251** and supported by clients such as scp or sftp. Nowadays, anonymous access is mainly provided by web protocols.

In the late 1980s, high energy physicists working at CERN had to efficiently exchange documents about their ongoing and planned experiments. Tim Berners-Lee evaluated several of the documents sharing techniques that were available at that time [B1989]. As none of the existing solutions met CERN's requirements, they chose to develop a completely new document sharing system. This system was initially called the *mesh*. It was quickly renamed the *world wide web*. The starting point for the *world wide web* are hypertext documents. An hypertext document is a document that contains references (hyperlinks) to other documents that the reader can immediately access. Hypertext was not invented for the world wide web. The idea of hypertext documents was proposed in 1945 [Bush1945] and the first experiments were done during the 1960s [Nelson1965] [Myers1998] . Compared to the hypertext documents that were used in the late 1980s, the main innovation introduced by the *world wide web* was to allow hyperlinks to reference documents stored on different remote machines.

A document sharing system such as the *world wide web* is composed of three important parts.

1. A standardized addressing scheme that unambiguously identifies documents

[6] **RFC 1939** defines the APOP authentication scheme that is not vulnerable to such attacks.

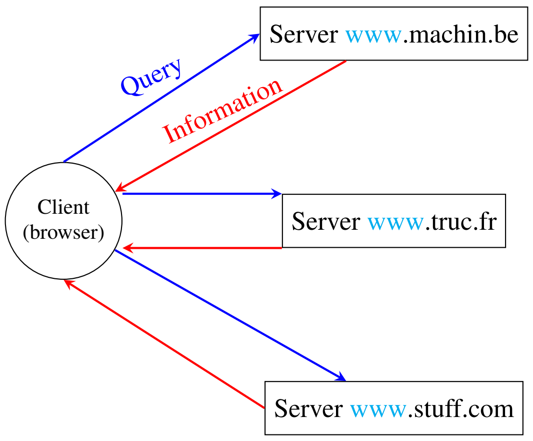


Fig. 11: World-wide web clients and servers

2. A standard document format : the HyperText Markup Language
3. A standardized protocol to efficiently retrieve the documents stored on a server

---

Note: Open standards and open implementations

Open standards play a key role in the success of the *world wide web* as we know it today. Without open standards, the world wide web would have never reached its current size. In addition to open standards, another important factor for the success of the web was the availability of open and efficient implementations of these standards. When CERN started to work on the *web*, their objective was to build a running system that could be used by physicists. They developed open-source implementations of the first web servers and web clients. These open-source implementations were powerful and could be used as is, by institutions willing to share information. They were also extended by other developers who contributed to new features. For example, the NCSA added support for images in their Mosaic browser that was eventually used to create Netscape Communications and the first commercial browsers and servers.

The first components of the *world wide web* are the Uniform Resource Identifiers (URI), defined in **RFC 3986**. A URI is a character string that unambiguously identifies a resource on the world wide web. Here is a subset of the BNF for URIs

```
URI            = scheme ":" "//" authority path [ "?" query ] [ "#" fragment ]
scheme         = ALPHA *( ALPHA / DIGIT / "+" / "-" / "." )
authority      = [ userinfo "@" ] host [ ":" port ]
query          = *( pchar / "/" / "?" )
fragment       = *( pchar / "/" / "?" )
pchar          = unreserved / pct-encoded / sub-delims / ":" / "@"
query          = *( pchar / "/" / "?" )
fragment       = *( pchar / "/" / "?" )
pct-encoded    = "%" HEXDIG HEXDIG
unreserved     = ALPHA / DIGIT / "-" / "." / "_" / "~"
reserved       = gen-delims / sub-delims
gen-delims     = ":" / "/" / "?" / "#" / "[" / "]" / "@"
sub-delims     = "!" / "$" / "&" / "'" / "(" / ")" / "*" / "+" / "," / ";" / "="
```

The first component of a URI is its *scheme*. A *scheme* can be seen as a selector, indicating the meaning of the fields after it. In practice, the scheme often identifies the application-layer protocol that must be used by the client to retrieve the document, but it is not always the case. Some schemes do not imply a protocol at all and some do not indicate a retrievable document[1]. The most frequent schemes are *http* and *https*. We focus on *http* in this section. A URI scheme

[1] An example of a non-retrievable URI is *urn:isbn:0-380-81593-1* which is an unique identifier for a book, through the urn scheme (see **RFC**

can be defined for almost any application layer protocol[2]. The characters *:* and *//* follow the *scheme* of any URI.

The second part of the URI is the *authority*. With retrievable URIs, this includes the DNS name or the IP address of the server where the document can be retrieved using the protocol specified via the *scheme*. This name can be preceded by some information about the user (e.g. a user name) who is requesting the information. Earlier definitions of the URI allowed the specification of a user name and a password before the @ character (**RFC 1738**), but this is now deprecated as placing a password inside a URI is insecure. The host name can be followed by the semicolon character and a port number. A default port number is defined for some protocols and the port number should only be included in the URI if a non-default port number is used (for other protocols, techniques like service DNS records can used).

The third part of the URI is the path to the document. This path is structured as filenames on a Unix host (but it does not imply that the files are indeed stored this way on the server). If the path is not specified, the server will return a default document. The last two optional parts of the URI are used to provide a query parameter and indicate a specific part (e.g. a section in an article) of the requested document. Sample URIs are shown below.

```
http://tools.ietf.org/html/rfc3986.html
mailto:infobot@example.com?subject=current-issue
http://docs.python.org/library/basehttpserver.html?highlight=http#BaseHTTPServer.
↪BaseHTTPRequestHandler
telnet://[2001:db8:3080:3::2]:2323/
ftp://cnn.example.com&story=breaking_news@10.0.0.1/top_story.htm
```

The first URI corresponds to a document named *rfc3986.html* that is stored on the server named *tools.ietf.org* and can be accessed by using the *http* protocol on its default port. The second URI corresponds to an email message, with subject *current-issue*, that will be sent to user *infobot* in domain *example.com*. The *mailto:* URI scheme is defined in **RFC 2368**. The third URI references the portion *BaseHTTPServer.BaseHTTPRequestHandler* of the document *basehttpserver.html* that is stored in the *library* directory on the *docs.python.org* server. This document can be retrieved by using the *http* protocol. The query parameter *highlight=http* is associated to this URI. The fourth example is a server that operates the telnet protocol, uses IPv6 address *2001:db8:3080:3::2* and is reachable on port 2323. The last URI is somewhat special. Most users will assume that it corresponds to a document stored on the *cnn.example.com* server. However, to parse this URI, it is important to remember that the @ character is used to separate the user name from the host name in the authorization part of a URI. This implies that the URI points to a document named *top_story.htm* on the host having IPv4 address *10.0.0.1*. The document will be retrieved by using the *ftp* protocol with the user name set to *cnn.example.com&story=breaking_news*.

The second component of the *word wide web* is the HyperText Markup Language (HTML). HTML defines the format of the documents that are exchanged on the *web*. The first version of HTML was derived from the Standard Generalized Markup Language (SGML) that was standardized in 1986 by *ISO*. SGML was designed to support large documents maintained by government, law firms or aerospace companies that must be shared efficiently in a machine-readable manner. These industries require documents to remain readable and editable for tens of years and insisted on a standardized format supported by multiple vendors. Today, SGML is no longer widely used beyond specific applications, but its descendants including *HTML* and *XML* are now widespread.

A markup language is a structured way of adding annotations about the formatting of the document within the document itself. Example markup languages include troff, which is used to write the Unix man pages or Latex. HTML uses markers to annotate text and a document is composed of *HTML elements*. Each element is usually composed of three parts: a start tag that potentially includes some specific attributes, some text (often including other elements), and an end tag. A HTML tag is a keyword enclosed in angle brackets. The generic form of an HTML element is

```
<tag>Some text to be displayed</tag>
```

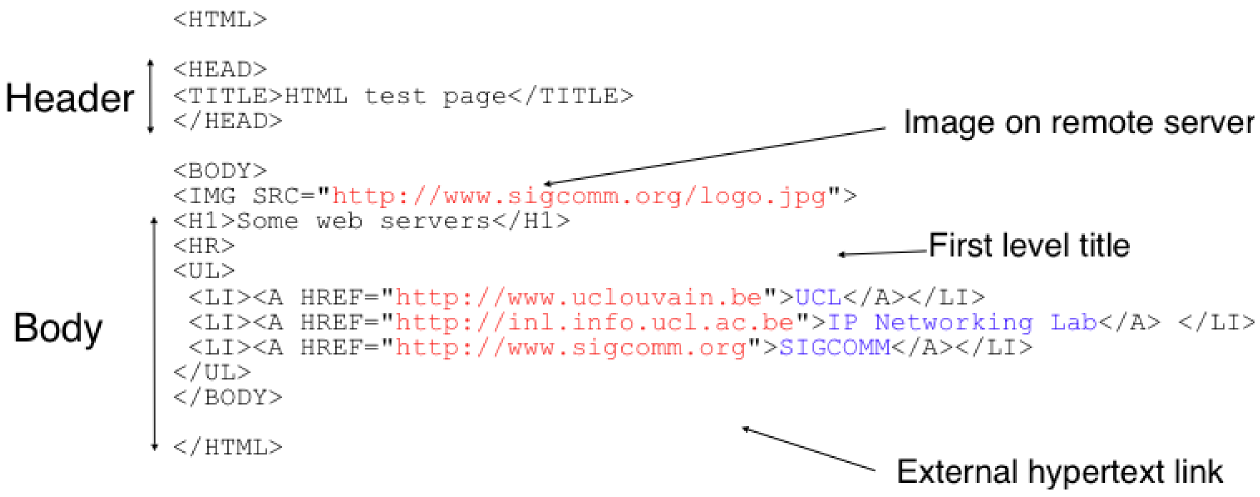
More complex HTML elements can also include optional attributes in the start tag

3187). Of course, any URI can be made retrievable via a dedicated server or a new protocol but this one has no explicit protocol. Same thing for the scheme tag (see **RFC 4151**), often used in Web syndication (see **RFC 4287** about the Atom syndication format). Even when the scheme is retrievable (for instance with *http*), it is often used only as an identifier, not as a way to get a resource. See http://norman.walsh.name/2006/07/25/namesAndAddresses for a good explanation.

[2] The list of standard URI schemes is maintained by IANA at http://www.iana.org/assignments/uri-schemes.html

```
<tag attribute1="value1" attribute2="value2">some text to be displayed</tag>
```

The HTML document shown below is composed of two parts: a header, delineated by the *<head>* and *</head>* markers, and a body (between the *<body>* and *</body>* markers). In the example below, the header only contains a title, but other types of information can be included in the header. The body contains an image, some text and a list with three hyperlinks. The image is included in the web page by indicating its URI between brackets inside the *<img src="... ">* marker. It is important to note that the image can reside on any server. The client will automatically download it when rendering the web page. The *<h1>... </h1>* marker is used to specify the first level of headings. The *<ul>* marker indicates an unnumbered list while the *<li>* marker indicates a list item. The *<a href="URI">text</a>* indicates a hyperlink. The *text* will be underlined in the rendered web page and the client will fetch the specified URI when the user clicks on the link.



```
<HTML>

<HEAD>
<TITLE>HTML test page</TITLE>
</HEAD>

<BODY>
<IMG SRC="http://www.sigcomm.org/logo.jpg">
<H1>Some web servers</H1>
<HR>
<UL>
 <LI><A HREF="http://www.uclouvain.be">UCL</A></LI>
 <LI><A HREF="http://inl.info.ucl.ac.be">IP Networking Lab</A> </LI>
 <LI><A HREF="http://www.sigcomm.org">SIGCOMM</A></LI>
</UL>
</BODY>

</HTML>
```

Fig. 12: A simple HTML page

Over the years, various extensions to HTML have been proposed and implemented. These include the specification of style sheets that adjust the layout of the document and the possibility of adding or referencing javascript code. Additional details about the various extensions to HTML may be found in the official specifications maintained by W3C.

The third component of the *world wide web* is the HyperText Transfer Protocol (HTTP). HTTP is a text-based protocol like SMTP. The client sends a request and the server returns a response. HTTP runs above the bytestream service and HTTP servers listen by default on port *80*. The design of HTTP has largely been inspired by the Internet email protocols. Each HTTP request contains three parts :

- a *method*, that indicates the type of request, a URI, and the version of the HTTP protocol used by the client
- a *header*, that is used by the client to specify optional parameters for the request. An empty line is used to mark the end of the header
- an optional MIME document attached to the request

The response sent by the server also contains three parts :

- a *status line* , that indicates whether the request was successful or not
- a *header*, that contains additional information about the response. The response header ends with an empty line.
- a MIME document

Several types of method can be used in HTTP requests. The three most important ones are :

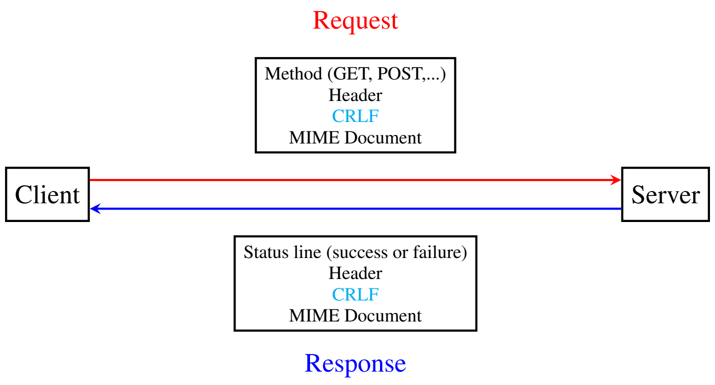


Fig. 13: HTTP requests and responses

- the *GET* method is the most popular one. It is used to retrieve a document from a server. The *GET* method is encoded as *GET* followed by the path of the URI of the requested document and the version of HTTP used by the client. For example, to retrieve the http://www.w3.org/MarkUp/ URI, a client must open a TCP connection on port *80* with host *www.w3.org* and send a HTTP request containing the following line:

```
GET /MarkUp/ HTTP/1.0
```

- the *HEAD* method is a variant of the *GET* method that allows the retrieval of the header lines for a given URI without retrieving the entire document. It can be used by a client to verify if a document exists, for instance.
- the *POST* method can be used by a client to send a document to a server. The document is attached to the HTTP request as a MIME document.

HTTP clients and servers can include different HTTP headers in HTTP requests and responses. Each HTTP header is encoded as a single ASCII-line terminated by *CR* and *LF*. Several of these headers are briefly described below. A detailed discussion of the standard headers may be found in **RFC 1945**. The MIME headers can appear in both HTTP requests and HTTP responses.

- the *Content-Length:* header is the *MIME* header that indicates the length of the MIME document in bytes.
- the *Content-Type:* header is the *MIME* header that indicates the type of the attached MIME document. HTML pages use the *text/html* type.
- the *Content-Encoding:* header indicates how the *MIME document* has been encoded. For example, this header would be set to *x-gzip* for a document compressed using the gzip software.

RFC 1945 and **RFC 2616** define headers that are specific to HTTP responses. These server headers include:

- the *Server:* header indicates the version of the web server that has generated the HTTP response. Some servers provide information about their software release and optional modules that they use. For security reasons, some system administrators disable these headers to avoid revealing too much information about their server to potential attackers.
- the *Date:* header indicates when the HTTP response has been produced by the server.
- the *Last-Modified:* header indicates the date and time of the last modification of the document attached to the HTTP response.

Similarly, the following header lines can only appear inside HTTP requests sent by a client:

- the *User-Agent:* header provides information about the client that has generated the HTTP request. Some servers analyze this header line and return different headers and sometimes different documents for different user agents.

- the *If-Modified-Since:* header is followed by a date. It enables clients to cache in memory or on disk recent or most frequently used documents. When a client needs to request a URI from a server, it first checks whether the document is already in its cache. If it is, the client sends an HTTP request with the *If-Modified-Since:* header indicating the date of the cached document. The server will only return the document attached to the HTTP response if it is newer than the version stored in the client's cache.
- the *Referrer:* header is followed by a URI. It indicates the URI of the document that the client visited before sending this HTTP request. Thanks to this header, the server can know the URI of the document containing the hyperlink followed by the client, if any. This information is very useful to measure the impact of advertisements containing hyperlinks placed on websites.
- the *Host:* header contains the fully qualified domain name of the URI being requested.

Note: The importance of the *Host:* header line

The first version of HTTP did not include the *Host:* header line. This was a severe limitation for web hosting companies. For example consider a web hosting company that wants to serve both *web.example.com* and *www.example.net* on the same physical server. Both web sites contain a */index.html* document. When a client sends a request for either *http://web.example.com/index.html* or *http://www.example.net/index.html*, the HTTP 1.0 request contains the following line :

```
GET /index.html HTTP/1.0
```

By parsing this line, a server cannot determine which *index.html* file is requested. Thanks to the *Host:* header line, the server knows whether the request is for *http://web.example.com/index.html* or *http://www.dummy.net/index.html*. Without the *Host:* header, this is impossible. The *Host:* header line allowed web hosting companies to develop their business by supporting a large number of independent web servers on the same physical server.

The status line of the HTTP response begins with the version of HTTP used by the server (usually *HTTP/1.0* defined in **RFC 1945** or *HTTP/1.1* defined in **RFC 2616**) followed by a three digit status code and additional information in English. HTTP status codes have a similar structure as the reply codes used by SMTP:

- All status codes starting with digit *2* indicate a valid response. *200 Ok* indicates that the HTTP request was successfully processed by the server and that the response is valid.
- All status codes starting with digit *3* indicate that the requested document is no longer available on the server. *301 Moved Permanently* indicates that the requested document is no longer available on this server. A *Location:* header containing the new URI of the requested document is inserted in the HTTP response. *304 Not Modified* is used in response to an HTTP request containing the *If-Modified-Since:* header. This status line is used by the server if the document stored on the server is not more recent than the date indicated in the *If-Modified-Since:* header.
- All status codes starting with digit *4* indicate that the server has detected an error in the HTTP request sent by the client. *400 Bad Request* indicates a syntax error in the HTTP request. *404 Not Found* indicates that the requested document does not exist on the server.
- All status codes starting with digit *5* indicate an error on the server. *500 Internal Server Error* indicates that the server could not process the request due to an error on the server itself.

In both HTTP requests and responses, the MIME document refers to a representation of the document with the MIME headers indicating the type of document and its size.

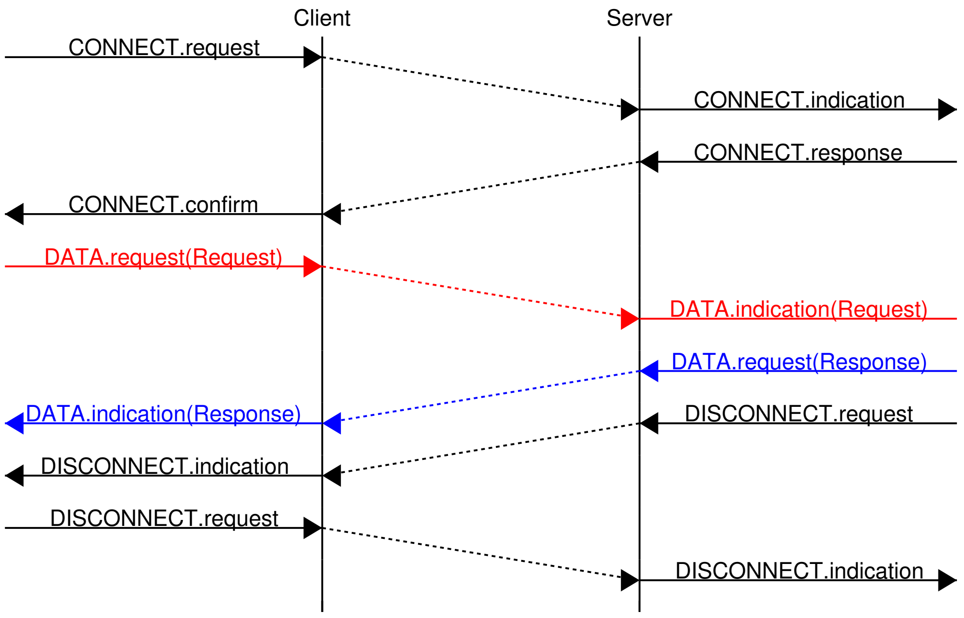
As an illustration of HTTP/1.0, the transcript below shows a HTTP request for http://www.ietf.org and the corresponding HTTP response. The HTTP request was sent using the curl command line tool. The *User-Agent:* header line contains more information about this client software. There is no MIME document attached to this HTTP request, and it ends with a blank line.

```
GET / HTTP/1.0
User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l␣
↪zlib/1.2.3
Host: www.ietf.org
```

The HTTP response indicates the version of the server software used with the modules included. The *Last-Modified:* header indicates that the requested document was modified about one week before the request. A HTML document (not shown) is attached to the response. Note the blank line between the header of the HTTP response and the attached MIME document. The *Server:* header line has been truncated in this output.

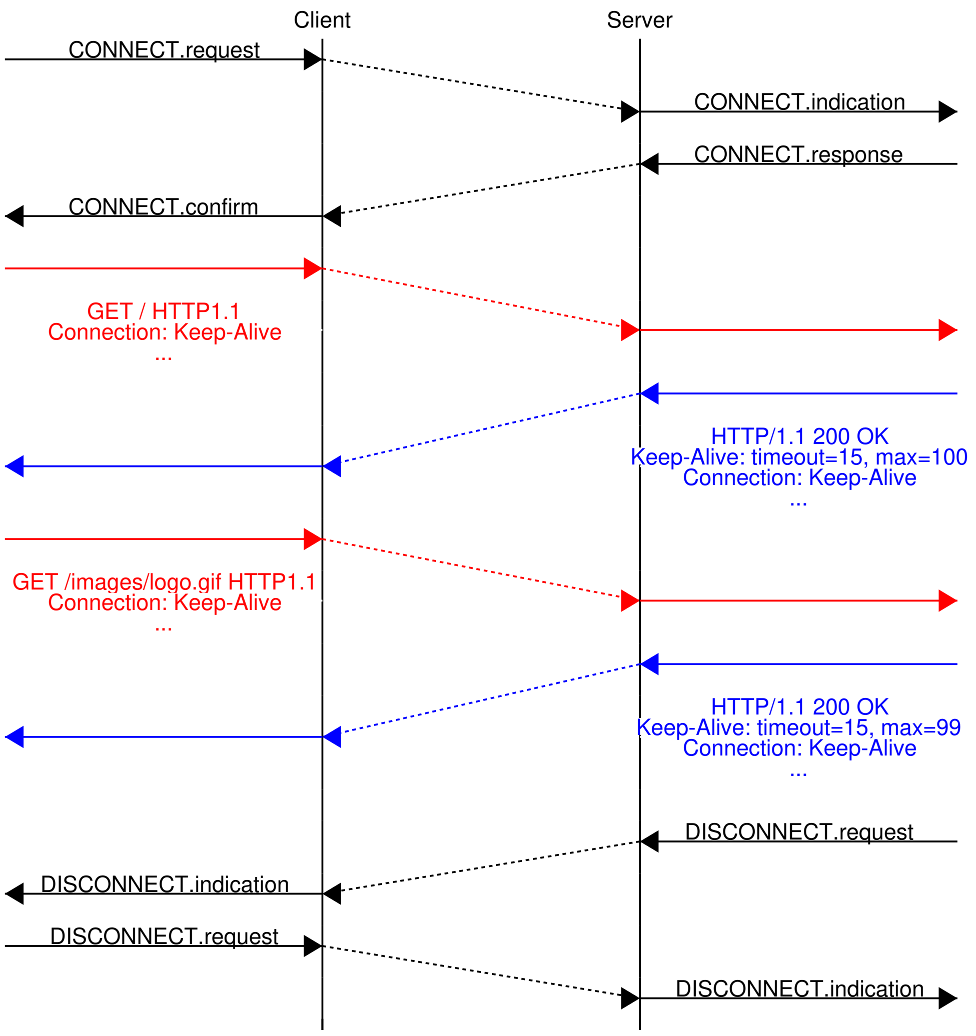
```
HTTP/1.1 200 OK
Date: Mon, 15 Mar 2010 13:40:38 GMT
Server: Apache/2.2.4 (Linux/SUSE) mod_ssl/2.2.4 OpenSSL/0.9.8e (truncated)
Last-Modified: Tue, 09 Mar 2010 21:26:53 GMT
Content-Length: 17019
Content-Type: text/html

<!DOCTYPE HTML PUBLIC .../HTML>
```

HTTP was initially designed to share text documents. For this reason, and to ease the implementation of clients and servers, the designers of HTTP chose to open a TCP connection for each HTTP request. This implies that a client must open one TCP connection for each URI that it wants to retrieve from a server as illustrated on the figure below, showing HTTP 1.0 and the underlying TCP connection. For a web page containing only text documents this was a reasonable design choice as the client usually remains idle while the (human) user is reading the retrieved document.



However, as the web evolved to support richer documents containing images, opening a TCP connection for each URI became a performance problem [Mogul1995]. Indeed, besides its HTML part, a web page may include dozens of images or more. Forcing the client to open a TCP connection for each component of a web page has two important drawbacks. First, the client and the server must exchange packets to open and close a TCP connection as we will see later. This increases the network overhead and the total delay of completely retrieving all the components of a web page. Second, a large number of established TCP connections may be a performance bottleneck on servers.

This problem was solved by extending HTTP to support persistent TCP connections **RFC 2616**. A persistent connection is a TCP connection over which a client may send several HTTP requests. This is illustrated in the figure below showing the persistent connection of HTTP 1.1.



To allow the clients and servers to control the utilization of these persistent TCP connections, HTTP 1.1 **RFC 2616** defines several new HTTP headers:

- The *Connection:* header is used with the *Keep-Alive* argument by the client to indicate that it expects the underlying TCP connection to be persistent. When this header is used with the *Close* argument, it indicates that the entity that sent it will close the underlying TCP connection at the end of the HTTP response.
- The *Keep-Alive:* header is used by the server to inform the client about how it agrees to use the persistent connection. A typical *Keep-Alive:* contains two parameters: the maximum number of requests that the server agrees to serve on the underlying TCP connection and the timeout (in seconds) after which the server will close an idle connection

The example below shows the operation of HTTP/1.1 over a persistent TCP connection to retrieve three URIs stored on the same server. Once the connection has been established, the client sends its first request with the *Connection: Keep-Alive* header to request a persistent connection.

```
GET / HTTP/1.1
Host: www.kame.net
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2; en-us)
Connection: Keep-Alive
```

The server replies with the *Connection: Keep-Alive* header and indicates that it accepts a maximum of 100 HTTP requests over this connection and that it will close the connection if it remains idle for 15 seconds.

```
HTTP/1.1 200 OK
Date: Fri, 19 Mar 2010 09:23:37 GMT
Server: Apache/2.0.63 (FreeBSD) PHP/5.2.12 with Suhosin-Patch
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Length: 3462
Content-Type: text/html

<html>...   </html>
```

The client sends a second request for the style sheet of the retrieved web page.

```
GET /style.css HTTP/1.1
Host: www.kame.net
Referer: http://www.kame.net/
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2; en-us)
Connection: keep-alive
```

The server replies with the requested style sheet and maintains the persistent connection. Note that the server only accepts 99 remaining HTTP requests over this persistent connection.

```
HTTP/1.1 200 OK
Date: Fri, 19 Mar 2010 09:23:37 GMT
Server: Apache/2.0.63 (FreeBSD) PHP/5.2.12 with Suhosin-Patch
Last-Modified: Mon, 10 Apr 2006 05:06:39 GMT
Content-Length: 2235
Keep-Alive: timeout=15, max=99
Connection: Keep-Alive
Content-Type: text/css

...
```

Then the client requested the web server's icon[3]. This server does not contain such an icon and thus replies with a *404* HTTP status. However, the underlying TCP connection is not closed immediately.

```
GET /favicon.ico HTTP/1.1
Host: www.kame.net
Referer: http://www.kame.net/
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2; en-us)
Connection: keep-alive

HTTP/1.1 404 Not Found
Date: Fri, 19 Mar 2010 09:23:40 GMT
Server: Apache/2.0.63 (FreeBSD) PHP/5.2.12 with Suhosin-Patch
Content-Length: 318
Keep-Alive: timeout=15, max=98
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN"> ...
```

As illustrated above, a client can send several HTTP requests over the same persistent TCP connection. However, it is important to note that all of these HTTP requests are considered to be independent by the server. Each HTTP request must be self-contained. This implies that each request must include all the header lines that are required by the

[3] Favorite icons are small icons that are used to represent web servers in the toolbar of Internet browsers. Microsoft added this feature in their browsers without taking into account the W3C standards. See http://www.w3.org/2005/10/howto-favicon for a discussion on how to cleanly support such favorite icons.

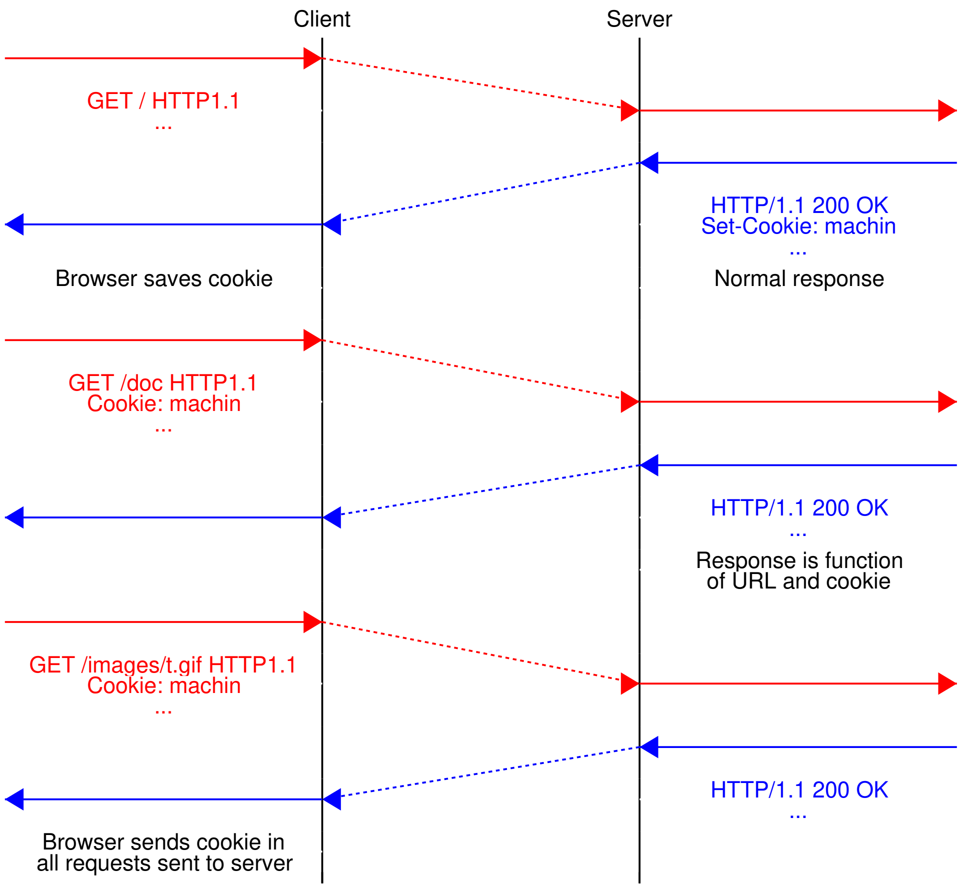
server to understand the request. The independence of these requests is one of the key design choices of HTTP. As a consequence of this design choice, when a server processes a HTTP request, it does not use any other information than what is contained in the request itself. This explains why the client adds its *User-Agent:* header in all of the HTTP requests that it sends over the persistent TCP connection.

However, in practice, some servers want to provide content tuned for each user. For example, some servers can provide information in several languages. Other servers want to provide advertisements that are targeted to different types of users. To do this, servers need to maintain some information about the preferences of each user and use this information to produce content matching the user's preferences. HTTP contains several mechanisms to solve this problem. We discuss three of them below.

A first solution is to force the users to be authenticated. This was the solution used by *FTP* to control the files that each user could access. Initially, user names and passwords could be included inside URIs **RFC 1738**. However, placing passwords in the clear in a potentially publicly visible URI is completely insecure and this usage has now been deprecated **RFC 3986**. HTTP supports several extension headers **RFC 2617** that can be used by a server to request the authentication of the client by providing his/her credentials. However, user names and passwords have not been popular on web servers as they force human users to remember one user name and one password per server. Remembering a password is acceptable when a user needs to access protected content, but users will not accept to remember a unique user name and password for each web sites that they visit.

A second solution to allow servers to tune that content to the needs and capabilities of the user is to rely on the different types of *Accept-** HTTP headers. For example, the *Accept-Language:* header can be used by the client to indicate its preferred languages. Unfortunately, in practice this header is usually set based on the default language of the browser and it is difficult for a user to indicate the language it prefers by selecting options for each visited web server.

The third and widely adopted solution are HTTP cookies. HTTP cookies were initially developed as a private extension by Netscape. They are now part of the standard **RFC 6265**. In a nutshell, a cookie is a short string that is chosen by a server to represent a given client. Two HTTP headers are used : *Cookie:* and *Set-Cookie:*. When a server receives an HTTP request from a new client (i.e. an HTTP request that does not contain the *Cookie:* header), it generates a cookie for the client and includes it in the *Set-Cookie:* header of the returned HTTP response. The *Set-Cookie:* header contains several additional parameters including the domain names for which the cookie is valid. The client stores all received cookies on disk and every time it sends an HTTP request, it verifies whether it already knows a cookie for this domain. If so, it attaches the *Cookie:* header to the HTTP request. This is illustrated in the figure below with HTTP 1.1, but cookies also work with HTTP 1.0.



Note: Privacy issues with HTTP cookies

The HTTP cookies introduced by Netscape are key for large e-commerce websites. However, they have also raised many discussions concerning their potential misuses. Consider *ad.com*, a company that delivers lots of advertisements on web sites. A web site that wishes to include *ad.com*'s advertisements next to its content will add links to *ad.com* inside its HTML pages. If *ad.com* is used by many web sites, *ad.com* could be able to track the interests of all the users that visit its client websites and use this information to provide targeted advertisements. Privacy advocates have even sued online advertisement companies to force them to comply with the privacy regulations. More recent related technologies also raise privacy concerns.

3.5 Making HTTP faster

During the last decade, a growing number of services have been supported by world wide web servers. The web protocols are not only used to deliver static documents, they are also used to deliver streaming music or video. They also enable clients to use interactive applications including games or productivity applications. These services and applications have more stringent performance requirements than the delivery of static documents. Many researchers and companies have proposed solutions to improve the performance of web services and protocols during the last decade [KR2001] [WBK2014]. We discuss a subset of them in this section.

A first way to improve the performance of the web protocols is to tune the servers that provide content. In the early days, documents were stored on a single server. Clients established TCP connections to this server to retrieve each document. This architecture evolved in several directions. A first way to speedup web services is to avoid unnecessary transmissions. Thanks to the *HEAD* method and the *If-Modified-Since:* header, web browsers can verify that they have the most recent version of a document in their cache.

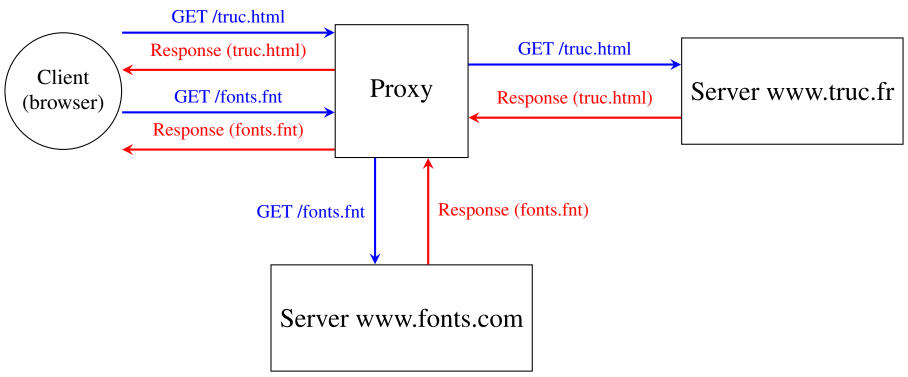


Fig. 14: Proxies relay client requests to servers and return the received responses

Caches can also be used inside the network. To understand their benefits, let us consider an SME with a dozen of employees that are connected to the Internet through a low-speed link. These employees often access similar web sites. Consider that Alice and Bob want to browse today's local newspaper. Their browsers will both retrieve the newspaper's website through the low bandwidth link and store the main documents in their cache. Unfortunately, the same information passes twice over the low-speed link. Some companies have deployed web proxies to cope with this problem. A web proxy is a server that resides in the enterprise network. All the employee's browsers are configured to send their HTTP requests to this proxy. When such a proxy receives a request, it checks whether the content is already stored inside its own cache. If so, it returns it directly. Otherwise, the request is sent to the remote server and the information is stored in the proxy cache. By reducing the number of web objects that are exchanged over low-speed links, such proxies can significantly improve performance. Some companies also use them to control the websites that are contacted by their employees and sometimes block illegitimate accesses.

Proxies can also be located in front of servers. In this case, they are called reverse-proxies. Consider a dynamic web server that produces web pages by assembling information stored in different databases. When this server receives a request, it must send multiple queries to its databases and then create the HTML document. These queries and the creation of the HTML document take time and this limits the number of requests that our server can sustain. Many content providers would place a reverse proxy in front of such a server. The DNS servers are configured to point to the reverse proxy. Upon reception of a request, the reverse proxy first checks whether the response is already stored in its cache. If so, it can return it to the client without interacting with the official server. Otherwise, the reverse proxy contacts the server and then returns the response to the client.

These reverse proxies can also be used to spread the load among different servers. In the above example, consider that a server needs 10 milliseconds to process each request and that it must handle them sequentially. Such a server cannot support more than 100 requests per second. If the service becomes popular, then the content provider will need to deploy several servers. These servers could serve the same reverse proxy.

Note: Serving content from multiple servers

When a web user interacts with *www.service.net*, she expects that all the information comes from the *www.service.net* server. If the service is popular, there are probably tens, hundreds, thousands or more physical servers that support this service. Still, the user has the illusion that she is interacting with a single server. Several techniques have been deployed by content providers to scale web services. Consider a simple service that serves text documents from *N* different servers. There are different ways to architect such a service.

A first approach is to store all files on each physical server and rely on the DNS to distribute the load among them. Each physical server has its own IP address and when the DNS server receives a query for *www.service.net*, it returns the IP address of one of them. Some DNS servers use Round-Robin to return one of these IP addresses. Others measure the load of the physical servers and return the address of the less loaded one. Another possibility is to locate the physical servers in different regions and configure the DNS server to return the IP address of the server that is geographically closer to the client's IP address.

A second approach is to rely on *k* reverse proxies and *N-k* servers. The servers store the content and the proxies cache the most frequently used files. The proxies can be geographically close to the clients while the servers can reside in the datacenters of the content provider. The DNS server can also distribute the load among the different proxies or return the geographically closest proxy. An important point to note about reverse proxies is that they receive HTTP requests from clients and send HTTP requests to the original servers that host the content. Several companies, usually called Content Distribution Networks, have deployed such reverse proxies throughout the world to cache web content next to the end-users. A good description of such a CDN may be found in [NSS2010].

A second way to improve the web performance is to reduce the time required to retrieve web objects. While the first web servers returned an HTML documents with possibly a few images, today's rich web servers return one HTML document with associated style sheets, javascript code, images, fonts, … Some of these web objects come from the original server while others are hosted on different servers. Today, a typical web page contains almost 2 MBytes of data on average. The size of the web pages continues to grow according to statistics collected by *httparchive.org*. Web pages targeted to mobile devices are slightly smaller.

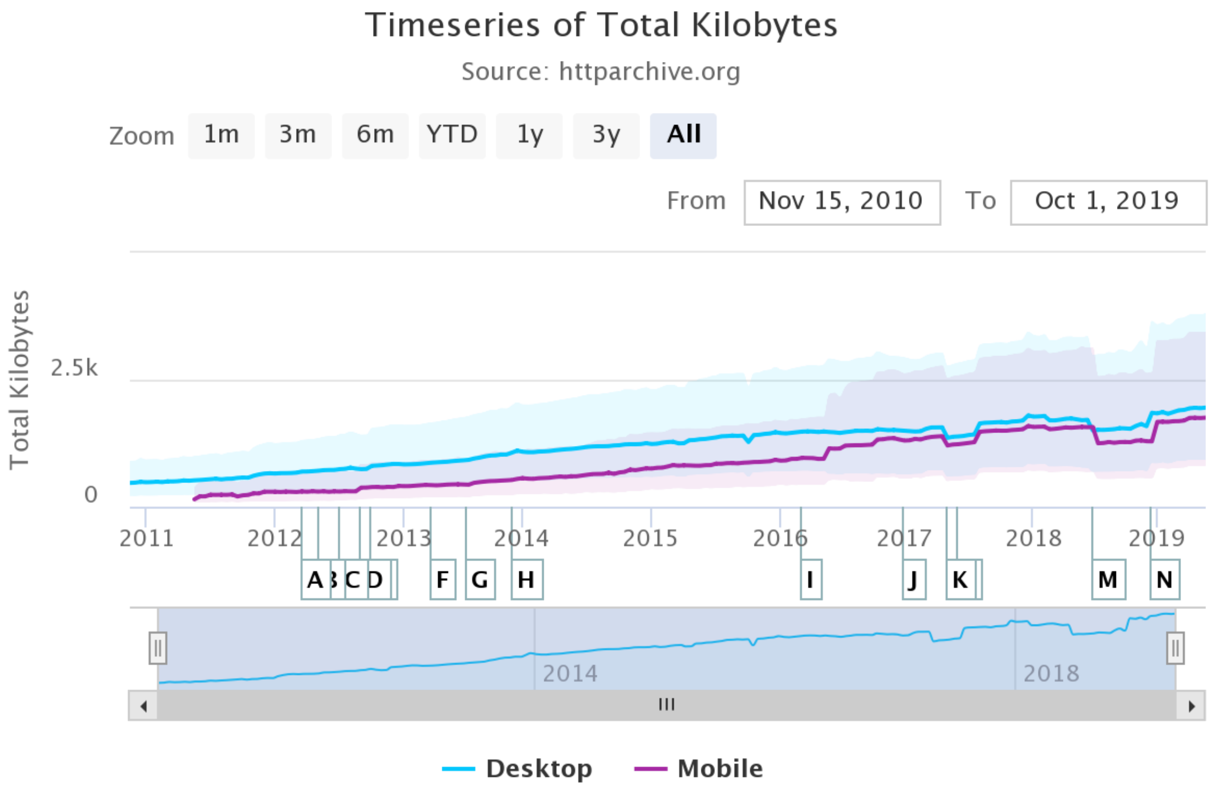


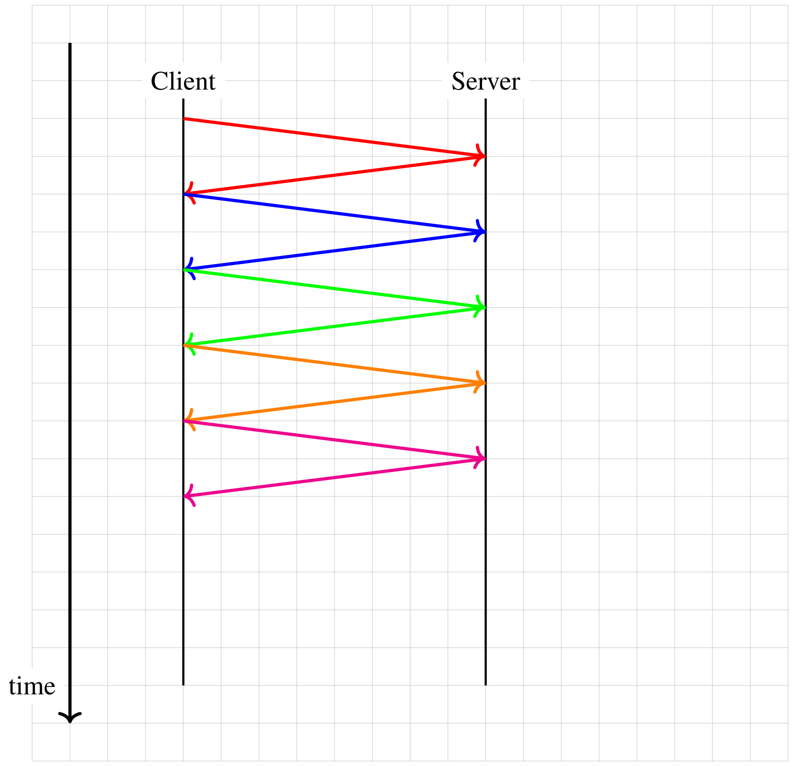
Fig. 15: Evolution of the size of the web pages (source: https://httparchive.org/reports/page-weight)

A closer look at the average web page shows that it contains, on average, 27 KBytes of HTML, 120 KBytes of fonts, 60 KBytes of CSS information, almost 1 MBytes of images and more than 400 KBytes of javascript. Each of these web page requires about 70 different HTTP requests. In other words, a browser needs to send on average 70 requests to retrieve a complete web page.

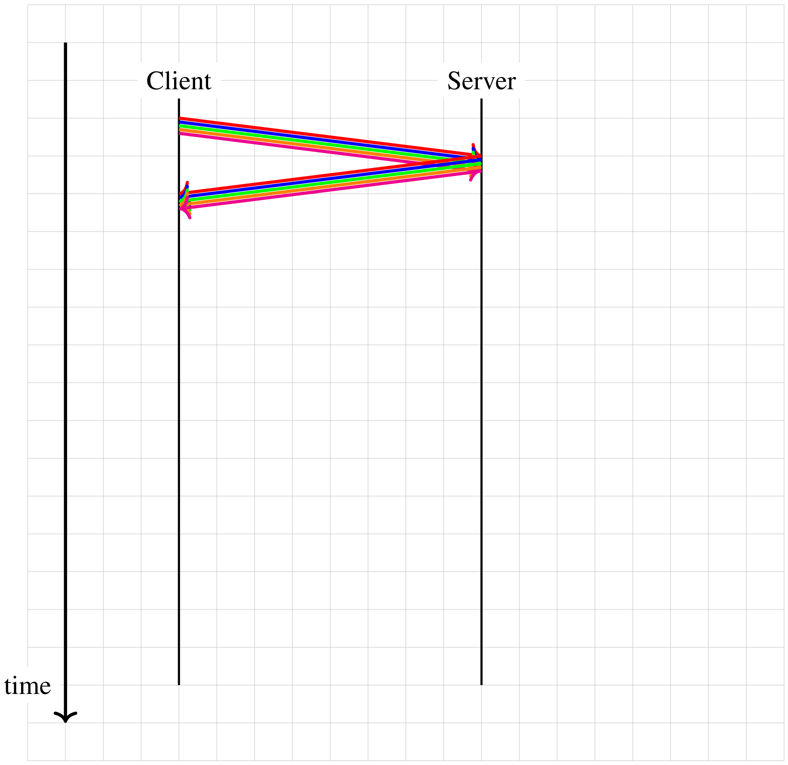
Two directions have been explored to improve the delivery of these web pages. The first direction is to tune the HTTP protocol. The second approach is to change the entire network stack. We will discuss this approach after having covered the entire stack.

One of the limitations of HTTP from a performance viewpoint is that the requests that are sent by a browser must be sequential. Typically, a browser requests the HTML page. Once the page has been retrieved, the browser parses it to identify all the objects that it references and requests them one after each other. The web page can only be displayed to the user once all the required web objects have been retrieved. This implies that the browser must wait until the reception of each response before sending the next request. Another possibility is to allow the browser to send multiple requests without waiting for their corresponding responses. This approach is called *pipelining* in **RFC 7230**.

To understand the benefits of pipelining, let us consider a simple but illustrative example. A client needs to retrieve 5 web objects that are each 100 bytes. The underlying transport connection has a 1 Gbps bandwidth but a one-way delay of 100 msec. A normal HTTP/1.x client would send the first request, wait 200 msec to receive the answer, then send another request… It would need one entire second to retrieve the five web objects. This is illustrated in the figure below.



With *pipelining*, the client sends the five requests immediately and receives the five responses after 200 msec. The figure below illustrates the benefits of *pipelining*.



However, as explained in **RFC 7230**, there is one important limitation to *pipelining*. It can only be used to serve HTTP requests that are idempotent, i.e. none of the requests must depend on any of the previous requests in the pipeline. It turned out that it was difficult for web browsers to correctly support this requirement and very few of them have implemented *pipelining*[1].

Another limitation of HTTP/1.1 is that all commands and parameters are encoded as ASCII strings. Using ASCII strings makes it easy to write simple clients or debug problems by observing packets. Unfortunately, the burden is placed on servers that need to include complex parsers that accept a wide range of partially compliant implementations. Furthermore, the flexibility of the ASCII encoding has enabled some classes of security attacks on servers [CWE444].

To cope with these two problems, the IETF HTTP working group developed version 2.0 of HTTP. HTTP/2.0 diverges from HTTP/1.1 in two important ways. First, HTTP/2.0 relies on binary encoding which is both more compact and easier to parse. Second, HTTP/2.0 supports multiple streams, which makes it possible to simultaneously transfer different web objects over a single transport connection. Furthermore, HTTP/2.0 also compresses the HTTP headers to reduce the amount of data transferred. This technique is described in **RFC 7541** but is not discussed in this chapter.

Let us first examine how HTTP/2.0 structures the bytestream of the underlying connection.

```
+-----------------------------------------------+
|                 Length (24)                   |
+---------------+---------------+---------------+
|   Type (8)    |   Flags (8)   |
+-+-------------+---------------+-------------------------------+
|R|                 Stream Identifier (31)                      |
+=+=============================================================+
|                   Frame Payload (0...)                      ...
+---------------------------------------------------------------+
```

Fig. 16: The HTTP/2.0 Frame header

The information exchanged over an HTTP/2.0 session is composed of frames. A frame starts with a 9 bytes-long header that carries several types of information. The HTTP/2.0 frames have a variable length. The *Length* field of the header contains the length of the frame payload in bytes. As this field is encoded as a 24 bits field, an HTTP/2.0 frame cannot be longer than $2^{24} - 1$ bytes. It should be noted that **RFC 7540** assumes a maximum size of 2^{14} bytes,

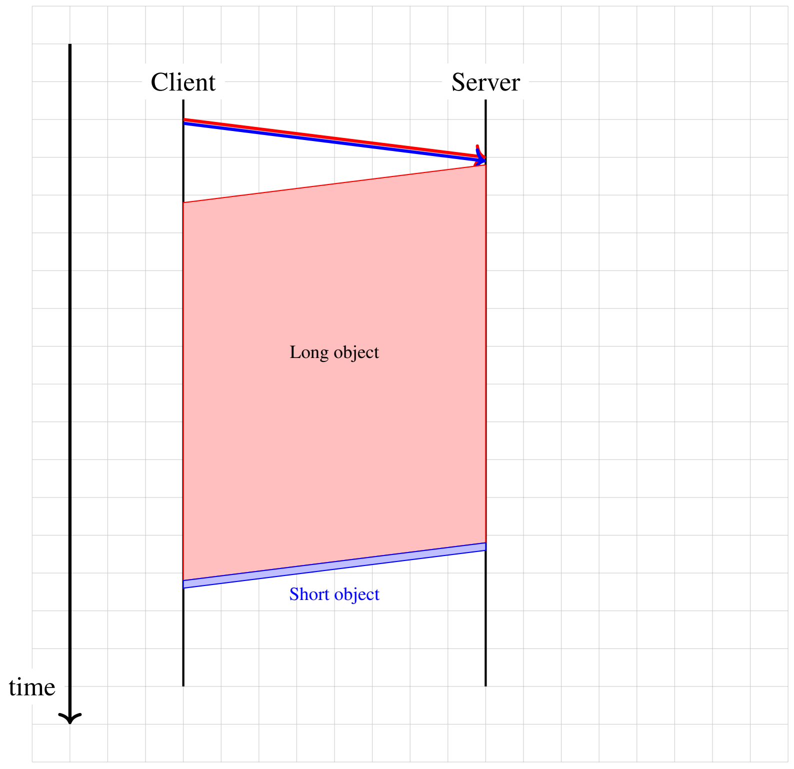
[1] See https://en.wikipedia.org/wiki/HTTP_pipelining for additional information.

i.e. 16,384 bytes for the HTTP/2.0 frame payload unless a longer maximum frame length has been negotiated at the beginning of the session using the HTTP/2.0 *Settings* frame that will be described later. The next field of the frames header indicates the frame type. The first frame types are *Data* which contains data from web objects and *Headers* containing HTTP/2.0 headers. When a client retrieves a web object from a server, it always receives an HTTP/2.0 *Headers* frame followed by an HTTP/2.0 *Data* frame. The *Headers* frame information contains essentially the same HTTP headers as the ones supported by HTTP/1.1, but those are encoded by leveraging a data compression technique that minimizes the number of bytes required to transmit them.

Other frame types are described later. The *Flags* are used for some frame types and the *R* bit must be set to zero. The last important field of the *HTTP/2.0 Frame* header is the *Stream Identifier*. With HTTP/2.0, the bytestream of the underlying transport connection is divided in independent streams that are identified by an integer. The odd (resp. even) stream identifiers are managed by the client (resp. server). This enables the server (or the client) to multiplex data corresponding to different frames over a single bytestream.

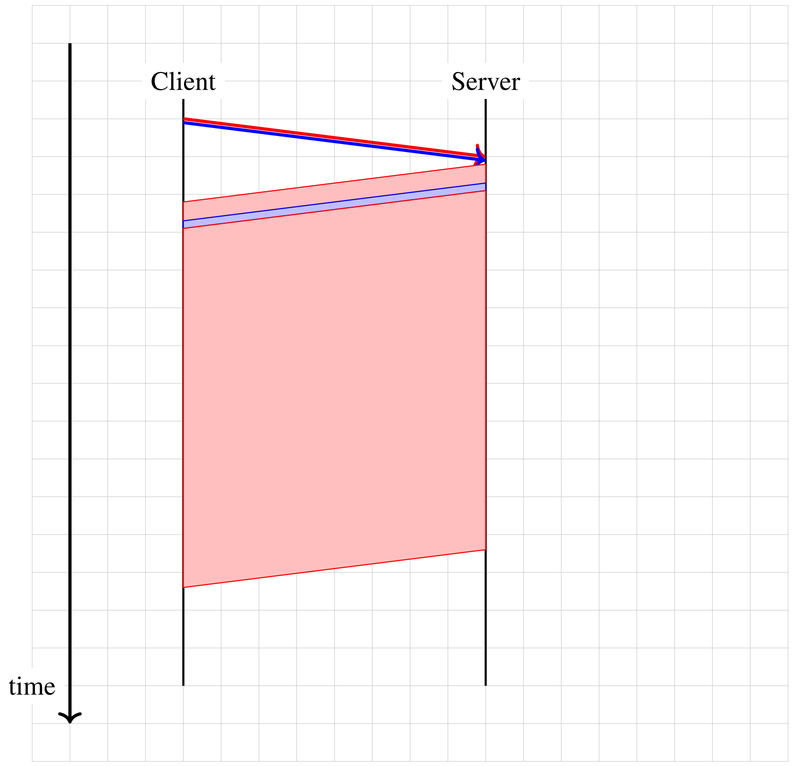
This multiplexing capability is probably the most important feature of HTTP/2.0 from a performance viewpoint. To understand its benefits, let us consider a client that retrieves two web objects over a 1 Mbps connection. The two requests are sent together by the client. The first object is 125 bytes long, while the second is 12500 bytes long. In this case, the server will first return the first object as a single frame and the second will be sent in the subsequent frame.

Consider now that the first object is 12500 bytes long and the second 125 bytes long. With a 1 Mbps connection, this object will use the underlying connection during 100 milliseconds. The client will thus need to wait 100 milliseconds to retrieve the second object. This is the *Head of Line* (HoL) blocking problem that affects the performance of many web services. If the short web object is a javascript code that requests other web objects, its retrieval may be critical to display the retrieved web page.



With HTTP/2.0 frames, the server could send the first 1250 bytes of the long object during 10 milliseconds, then send a second frame that contains the short object during one millisecond and later send a longer frame that contains the remaining 11250 bytes of the long object. In this case, the client has received the short object after 10 milliseconds. Given the HTTP/2.0 streams, the transmission of long web objects does not anymore blocks the transmission of shorter ones.

The length of the HTTP/2.0 frames obviously affects how different web objects can be multiplexed over the underlying transport connection. If HTTP/2.0 frames are long, the overhead of the frame header is minimal, but long frames can block short web objects. On the other hand, if the frame length is small, then the overhead due to the HTTP/2.0 frame header could become significant.



The HTTP/2.0 streams can provide performance benefits, but they also increase the complexity of the implementations since an HTTP/2.0 receiver must be able to simultaneously process frames that correspond to different web objects. This complexity mainly resides on the client side. The HTTP/2.0 protocol includes several techniques that enable clients to manage the utilization of the HTTP/2.0 session.

The first frame that a client sends over an HTTP/2.0 session is the *Settings* frame. This is a control frame that indicates some parameters that the client proposes for this session. Several of these parameters are defined in **RFC 7540**. The most important ones are probably the *SETTINGS_MAX_FRAME_SIZE* that specifies the maximum length of the HTTP/2.0 frames that this implementation supports and the *SETTINGS_MAX_CONCURRENT_STREAMS* that specifies the maximum number of parallel streams that this implementation can manage. The *SETTINGS_MAX_FRAME_SIZE* must be at least 2^{14} bytes but can go up to $2^{24} - 1$ bytes. There is no minimum value for *SETTINGS_MAX_CONCURRENT_STREAMS*, but **RFC 7540** recommends to support at least 100 different stream identifiers.

By using multiple streams, the server can multiplex different web objects over the same underlying transport connection. However, these objects are only sent in response to requests from clients. There are some situations where the server might know in advance that the client will request a given object. It could speedup the transfer by sending it before having received a client request. This is the *push* feature of HTTP/2.0. A server can independently push web objects to a client without having received any request. This feature can only be used by the server if the client has enabled it by sending *SETTINGS_ENABLE_PUSH* in its *Settings* frame. A classical use case for this *push* feature is to enable a server to automatically send an object which cannot be cached by the client, such as a dynamic javascript code, when another web object that references it is requested. However, measurement studies indicate that very few web servers seem to have adopted this feature [ZWH2018].

Another feature of HTTP/2.0 is that it is possible to assign different priorities to different streams. A high priority stream should carry more *Data* frames than a lower priority ones. The HTTP/2.0 specification defines *Priority* frames which can be used for this purpose.

As the server can send multiple objects at the same time, there is a risk of overloading the client buffers. To cope with this potential problem, HTTP/2.0 includes its own flow control mechanism. When an HTTP/2.0 session starts, a receiver agrees to receive up to 65,535 bytes over this connection (unless it has indicated a different initial window in its *Settings* frame). This limits the amount of data that a sender can transmit over the HTTP/2.0 session. The receiver can advertise a large receive window by sending a *Window_Update* frame at any time. This flow control mechanism can be applied to the entire connection or to a specific stream. In practice, using a small *HTTP/2.0 window* could severely limit the throughput over an HTTP/2.0 session.

HTTP/2.0 includes much more than what we have covered in this short introduction. There is for example a *Ping* frame that allows measuring the round-trip-time between a client and a server or the *GoAway* frame that indicates the termination of an HTTP/2.0 session. This frame contains an error code that indicates why the session has been terminated. Several error codes are defined in **RFC 7540**, including *ENHANCE_YOUR_CALM* that is used to indicate that the other endpoint exhibits an behavior that could cause excessive load.

Note: Detecting whether a server supports HTTP/2.0

HTTP/2.0 is a new version of the HTTP protocol that still uses port 80. When a client contacts an HTTP server, it must be able to determine whether it supports HTTP/1.x or HTTP/2.0. If the client sends a binary encoded HTTP/2.0 request to a server that only supports the ASCII encoded HTTP/1.x, it could cause problems on the server and even crash it. To minimize the risk of crashing HTTP/1.x servers, an HTTP/2.0 session starts like an HTTP/1.1 session and the first request contains the *Connection*, *Upgrade* and *HTTP2-Settings* headers. An example of such a request to upgrade the version of HTTP is shown below.

```
GET /robots.txt HTTP/1.1
Host: nghttp2.org
User-Agent: curl/7.52.1
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: AAMAAABkAARAAAAA
```

The *HTTP2-Settings* line contains the HTTP/2.0 settings frame that the client would server over an HTTP/2.0 session encoded in Base64. The server replies with a response that indicates that it has accepted to upgrade the connection to HTTP/2.0. A sample response is shown below.

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c
```

Finally, the client and the server need to confirm the utilization of HTTP/2.0. A client confirms this by sending the following Magic string *PRI * HTTP/2.0rnrnSMrnrn* or *0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a* in hex. This string is followed by a SETTINGS frame. The server must send a possibly empty SETTINGS frame.

3.6 Remote Procedure Calls

In the previous sections, we have described several protocols that enable humans to exchange messages and access to remote documents. This is not the only usage of computer networks and in many situations applications use the network to exchange information with other applications. When an application needs to perform a large computation on a host, it can sometimes be useful to request computations from other hosts. Many distributed systems have been built by distributing applications on different hosts and using *Remote Procedure Calls* as a basic building block.

In traditional programming languages, *procedure calls* allow programmers to better structure their code. Each procedure is identified by a name, a return type and a set of parameters. When a procedure is called, the current flow of program execution is diverted to execute the procedure. This procedure uses the provided parameters to perform its computation and returns one or more values. This programming model was designed with a single host in mind. In a nutshell, most programming languages support it as follows :

1. The caller places the values of the parameters at a location (register, stack, . . .) where the callee can access them
2. The caller transfers the control of execution to the callee's procedure
3. The callee accesses the parameters and performs the requested computation
4. The callee places the return value(s) at a location (register, stack, . . .) where the caller can access them
5. The callee returns the control of execution to the caller's

This model was developed with a single host in mind. How should it be modified if the caller and the callee are different hosts connected through a network ? Since the two hosts can be different, the two main problems are the fact they do not share the same memory and that they do not necessarily use the same representation for numbers, characters, . . . Let us examine how the five steps identified above can be supported through a network.

The first problem to be solved is how to transfer the information from the caller to the callee. This problem is not simple and includes two sub-problems. The first sub-problem is the encoding of the information. How to encode the values of the parameters so that they can be transferred correctly through the network ? The second problem is how to reach the callee through the network ? The callee is identified by a procedure name, but to use the transport service, we need to convert this name into an address and a port number.

3.6.1 Encoding data

The encoding problem exists in a wide range of applications. In the previous sections, we have described how character-based encodings are used by email and HTTP. Although standard encoding techniques such as ASN.1 [Dubuisson2000] have been defined to cover most application needs, many applications have defined their specific encoding. *Remote Procedure Call* are no exception to this rule. The three most popular encoding methods are probably XDR **RFC 1832** used by ONC-RPC **RFC 1831**, XML, used by XML-RPC and JSON **RFC 4627**.

The eXternal Data Representation (XDR) Standard, defined in **RFC 1832** is an early specification that describes how information exchanged during Remote Procedure Calls should be encoded before being transmitted through a network. Since the transport service enables transfering a block of bytes (with the connectionless service) or a stream of bytes (by using the connection-oriented service), XDR maps each datatype onto a sequence of bytes. The caller encodes each data in the appropriate sequence and the callee decodes the received information. Here are a few examples extracted from **RFC 1832** to illustrate how this encoding/decoding can be performed.

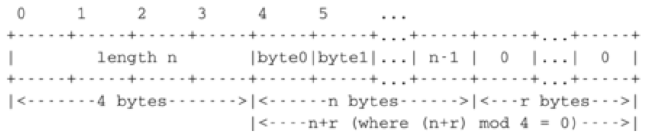
For basic data types, **RFC 1832** simply maps their representation into a sequence of bytes. For example a 32 bits integer is transmitted as follows (with the most significant byte first, which corresponds to big-endian encoding).

```
  (MSB)                   (LSB)
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
+-------+-------+-------+-------+
<------------32 bits------------>
```

XDR also supports 64 bits integers and booleans. The booleans are mapped onto integers (*0* for *false* and *1* for *true*). For the floating point numbers, the encoding defined in the IEEE standard is used.

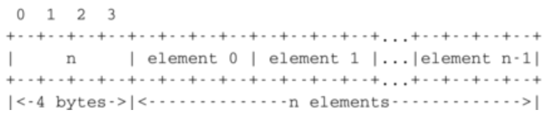
```
+------+------+------+------+------+------+------+------+
|byte 0|byte 1|byte 2|byte 3|byte 4|byte 5|byte 6|byte 7|
S|    E   |                 F                          |
+------+------+------+------+------+------+------+------+
1|<--11-->|<-----------------52 bits------------------->|
<-----------------------64 bits------------------------->
```

In this representation, the first bit (S) is the sign (*0* represents positive). The next 11 bits represent the exponent of the number (E), in base 2, and the remaining 52 bits are the fractional part of the number (F). The floating point number that corresponds to this representation is $(-1)^S \times 2^{E-1023} \times 1.F$. XDR also allows encoding complex data types. A first example is the string of bytes. A string of bytes is composed of two parts : a length (encoded as an integer) and a sequence of bytes. For performance reasons, the encoding of a string is aligned to 32 bits boundaries. This implies that some padding bytes may be inserted during the encoding operation is the length of the string is not a multiple of 4. The structure of the string is shown below (source **RFC 1832**).



```
  0     1     2     3     4     5   ...
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|        length n       |byte0|byte1|...| n-1 |  0  |...|  0  |
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|<-------4 bytes------->|<------n bytes------>|<---r bytes--->|
                        |<----n+r (where (n+r) mod 4 = 0)---->|
```

In some situations, it is necessary to encode fixed or variable length arrays. XDR **RFC 1832** supports such arrays. For example, the encoding below corresponds to a variable length array containing n elements. The encoded representation starts with an integer that contains the number of elements and follows with all elements in sequence. It is also possible to encode a fixed-length array. In this case, the first integer (the *n* field) is missing.



```
  0  1  2  3
+--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
|     n     | element 0 | element 1 |...|element n-1|
+--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
|<-4 bytes->|<--------------n elements------------->|
```

XDR also supports the definition of unions, structures, . . . Additional details are provided in **RFC 1832**.

A second popular method to encode data is the JavaScript Object Notation (JSON). This syntax was initially defined to allow applications written in JavaScript to exchange data, but it has now wider usages. JSON **RFC 4627** is a text-based representation. The simplest data type is the integer. It is represented as a sequence of digits in ASCII. Strings can also be encoding by using JSON. A JSON string always starts and ends with a quote character (“) as in the C language. As in the C language, some characters (like “ or \) must be escaped if they appear in a string. **RFC 4627** describes this in details. Booleans are also supported by using the strings *false* and *true*. Like XDR, JSON supports more complex data types. A structure or object is defined as a comma separated list of elements enclosed in curly brackets. **RFC 4627** provides the following example as an illustration.

```
{
   "Image": {
       "Width":  800,
       "Height": 600,
       "Title":  "View from 15th Floor",
       "Thumbnail": {
           "Url":    "http://www.example.com/image/481989943",
           "Height": 125,
           "Width":  100
       },
       "ID": 1234
     }
}
```

This object has one field named *Image*. It has five attributes. The first one, *Width*, is an integer set to 800. The third one is a string. The fourth attribute, *Thumbnail* is also an object composed of three different attributes, one string and two integers. JSON can also be used to encode arrays or lists. In this case, square brackets are used as delimiters. The snippet below shows an array which contains the prime integers that are smaller than ten.

```
{
  "Primes" : [ 2, 3, 5, 7 ]
}
```

Compared with XDR, the main advantage of JSON is that the transfer syntax is easily readable by a human. However, this comes at the expense of a less compact encoding. Some data encoded in JSON will usually take more space than when it is encoded with XDR. More compact encoding schemes have been defined, see e.g. [BH2013] and the references therein.

3.6.2 Reaching the callee

The second sub-problem is how to reach the callee. A simple solution to this problem is to make sure that the callee listens on a specific port on the remote machine and then exchange information with this server process. This is the solution chosen for JSON-RPC [JSON-RPC2]. JSON-RPC can be used over the connectionless or the connection-oriented transport service. A JSON-RPC request contains the following fields:

- *jsonrpc*: a string indicating the version of the protocol used. This is important to allow the protocol to evolve in the future.
- *method*: a string that contains the name of the procedure which is invoked
- *params*: a structure that contains the values of the parameters that are passed to the method
- *id*: an identifier chosen by the caller

The JSON-RPC is encoded as a JSON object. For example, the example below shows an invocation of a method called *sum* with *1* and *3* as parameters.

```
{ "jsonrpc": "2.0", "method": "sum", "params": [1, 3], "id": 1 }
```

Upon reception of this JSON structure, the callee parses the object, locates the corresponding method and passes the parameters. This method returns a response which is also encoded as a JSON structure. This response contains the following fields:

- *jsonrpc*: a string indicating the version of the protocol used to encode the response
- *id*: the same identifier as the identifier chosen by the caller
- *result*: if the request succeeded, this member contains the result of the request (in our example, value *4*).
- *error*: if the method called does not exist or its execution causes an error, the *result* element will be replaced by an *error* element which contains the following members :
  - *code*: a number that indicates the type of error. Several error codes are defined in [JSON-RPC2]. For example, *-32700* indicates an error in parsing the request, *-32602* indicates invalid parameters and *-32601* indicates that the method could not be found on the server. Other error codes are listed in [JSON-RPC2].
  - *message*: a string (limited to one sentence) that provides a short description of the error.
  - *data*: an optional field that provides additional information about the error.

Coming back to our example with the call for the *sum* procedure, it would return the following JSON structure.

```
{ "jsonrpc": "2.0", "result": 4, "id": 1 }
```

If the *sum* method is not implemented on the server, it would reply with the following response.

```
{ "jsonrpc": "2.0", "error": {"code": -32601, "message": "Method not found"}, "id": "1
↪" }
```

The *id* field, which is present in the request and the response plays the same role as the identifier field in the DNS message. It allows the caller to match the response with the request that it sent. This *id* is very important when JSON-RPC is used over the connectionless transport service which is unreliable. If a request is sent, it may need to be retransmitted and it is possible that a callee will receive twice the same request (e.g. if the response for the first request was lost). In the DNS, when a request is lost, it can be retransmitted without causing any difficulty. However with remote procedure calls in general, losses can cause some problems. Consider a method which is used to deposit money on a bank account. If the request is lost, it will be retransmitted and the deposit will be eventually performed. However, if the response is lost, the caller will also retransmit its request. This request will be received by the callee that will deposit the money again. To prevent this problem from affecting the application, either the programmer must ensure that the remote procedures that it calls can be safely called multiple times or the application must verify whether the request has been transmitted earlier. In most deployments, the programmers use remote methods that can be safely called multiple times without breaking the application logic.

ONC-RPC uses a more complex method to allow a caller to reach the callee. On a host, server processes can run on different ports and given the limited number of port values (2^{16} per host on the Internet), it is impossible to reserve one port number for each method. The solution used in ONC-RPC **RFC 1831** is to use a special method which is called the *portmapper* **RFC 1833**. The *portmapper* is a kind of directory that runs on a server that hosts methods. The *portmapper* runs on a standard port (*111* for ONC-RPC **RFC 1833**). A server process that implements a method registers its method on the local *portmapper*. When a caller needs to call a method on a remote server, it first contacts the *portmapper* to obtain the port number of the server process which implements the method. The response from the portmapper allows it to directly contact the server process which implements the method.

3.7 Remote login

One of the initial motivations for building computer networks was to allow users to access remote computers over the networks. In the 1960s and 1970s, the mainframes and the emerging minicomputers were composed of a central unit and a set of terminals connected through serial lines or modems. The simplest protocol that was designed to access remote computers over a network is probably *telnet* **RFC 854**. *telnet* runs over TCP and a telnet server listens on port *23* by default. The TCP connection used by telnet is bidirectional, both the client and the server can send data over it. The data exchanged over such a connection is essentially the characters that are typed by the user on the client machine and the text output of the processes running on the server machine with a few exceptions (e.g. control characters, characters to control the terminal like VT-100, . . .) . The default character set for telnet is the ASCII character set, but the extensions specified in **RFC 5198** support the utilization of Unicode characters.

From a security viewpoint, the main drawback of *telnet* is that all the information, including the usernames, passwords and commands, is sent in cleartext over a TCP connection. This implies that an eavesdropper could easily capture the passwords used by anyone on an unprotected network. Various software tools exist to automate this collection of information. For this reason, *telnet* is rarely used today to access remote computers. It is usually replaced by *ssh* or similar protocols.

3.7.1 The secure shell (ssh)

The secure shell protocol was designed in the mid 1990s by T. Ylonen to counter the eavesdropping attacks against *telnet* and similar protocols [Ylonen1996]. *ssh* became quickly popular and system administrators encouraged its usage. The original version of *ssh* was freely available. After a few years, his author created a company to distribute it commercially, but other programmers continued to develop an open-source version of *ssh* called OpenSSH. Over the years, *ssh* evolved and became a flexible applicable whose usage extends beyond remote login to support features such as file transfers, protocol tunneling, . . . In this section, we only discuss the basic features of *ssh* and explain how it differs from *telnet*. Entire books have been written to describe *ssh* in details [BS2005]. An overview of the protocol appeared in [Stallings2009].

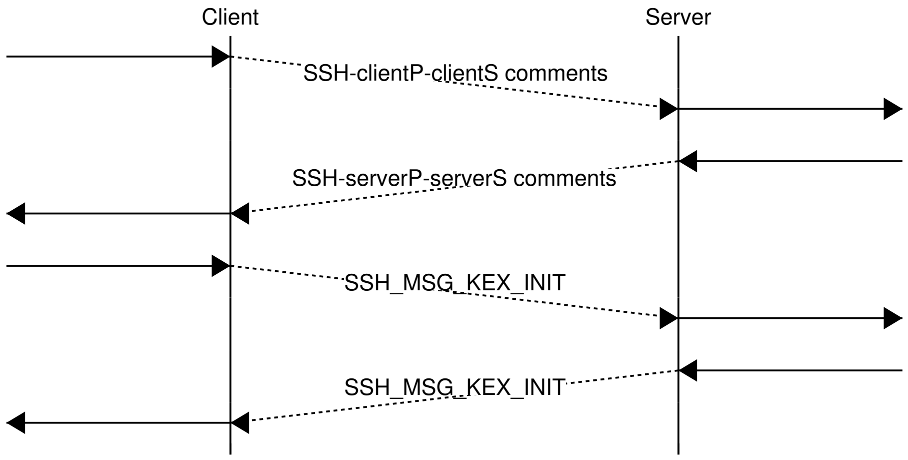
The *ssh* protocol runs directly above the TCP protocol. Once the TCP bytestream has been established, the client and the server exchange messages. The first message exchanged is an ASCII line that announces the version of the protocol and the version of the software implementation used by the client and the server. These two lines are useful when debugging interoperability problems and other issues.

The next message is the `SSH_MSG_KEX_INIT` message that is used to negotiate the cryptographic algorithms that will be used for the `ssh` session. It is very important for security protocols to include mechanisms that enable a negotiation of the cryptographic algorithms that are used. First, these algorithms provide different levels of security. Some algorithms might be considered totally secure and are recommended today while they could become deprecated a few years later after the publication of some attacks. Second, these algorithms provide different levels of performance and have different CPU and memory impacts.

In practice, an `ssh` implementation supports four types of cryptographic algorithms :

- key exchange
- encryption
- Message Authentication Code (MAC)
- compression

The IANA maintains a list of the cryptographic algorithms that can be used by `ssh` implementations. For each type of algorithm, the client provides an ordered list of the algorithms that it supports and agrees to use. The server compares the received list with its own list. The outcome of the negotiation is a set of four algorithms[1] that will be combined for this session.



This negotiation of the cryptographic algorithms allows the implementations to evolve when new algorithms are proposed. If a client is upgraded, it can announce a new algorithm as its preferred one even if the server is not yet upgraded.

Once the cryptographic algorithms have been negotiated, the key exchange algorithm is used to negotiate a secret key that will be shared by the client and the server. These key exchange algorithms include some variations over the basic algorithms. As an example, let us analyze how the Diffie Hellman key exchange algorithm is used within the `ssh` protocol. In this case, each host has both a private and a public key.

- the client generates the random number a and sends $A = g^a \mod p$ to the server
- the server generates the random number b. It then computes $B = g^b \mod p$, $K = B^a \mod p$ and signs with its private key $hash(V_{Client}||V_{Server}||KEX\_INIT_{Client}||KEX\_INIT_{Server}||Server_{pub}||A||B||K)$ where V_{Server} (resp. V_{Client}) is the initial messages sent by the client (resp. server), KEX_INIT_{Client} (resp. KEX_INIT_{Server}) is the key exchange message sent by the client (resp. server) and A, B and K are the messages of the Diffie Hellman key exchange
- the client can recompute $K = A^b \mod p$ and verify the signature provided by the server

This is a slightly modified authenticated Diffie Hellman key exchange with two interesting points. The first point is that when the server authenticates the key exchange it does not provide a certificate. This is because `ssh` assumes that

[1] For some of the algorithms, it is possible to negotiate the utilization of no algorithm. This happens frequently for the compression algorithm that is not always used. For this, both the client and the server must announce `null` in their ordered list of supported algorithms.

the client will store inside its cache the public key of the servers that it uses on a regular basis. This assumption is valid for a protocol like `ssh` because users typically use it to interact with a small number of servers, typically a few or a few tens. Storing this information does not require a lot of storage. In practice, most `ssh` clients will accept to connect to remote servers without knowing their public key before the connection. In this case, the client issues a warning to the user who can decide to accept or reject the key. This warning can be associated with a fingerprint of the key, either as a sequence of letters or as an ASCII art which can be posted on the web or elsewhere[2] by the system administrator of the server. If a client connects to a server whose public key does not match the stored one, a stronger warning is issued because this could indicate a man-in-the-middle attack or that the remote server has been compromised. It can also indicate that the server has been upgraded and that a new key has been generated during this upgrade.

The second point is that the server authenticates not only the result of the Diffie Hellman exchange but also a hash of all the information sent and received during the exchange. This is important to prevent *downgrade attacks*. A *downgrade attack* is an attack where an active attacker modifies the messages sent by the communicating hosts (typically the client) to request the utilization of weaker encryption algorithms. Consider a client that supports two encryption schemes. The preferred one uses 128 bits secret keys and the second one is an old encryption scheme that uses 48 bits keys. This second algorithm is kept for backward compatibility with older implementations. If an attacker can remove the preferred algorithm from the list of encryption algorithms supported by the client, he can force the server to use a weaker encryption scheme that will be easier to break. Thanks to the hash that covers all the messages exchanged by the server, the downgrade attack cannot occur against `ssh`. Algorithm agility is a key requirement for security protocols that need to evolve when encryption algorithms are broken by researchers. This agility cannot be used without care and signing a hash of all the messages exchanged is a technique that is frequently used to prevent downgrade attacks.

Note: Single use keys

Thanks to the Diffie Hellman key exchange, the client and the servers share key K. A naive implementation would probably directly use this key for all the cryptographic algorithms that have been negotiated for this session. Like most security protocols, `ssh` does not directly use key K. Instead, it uses the negotiated hash function with different parameters[3] to allow the client and the servers to compute six keys from K :

- a key used by the client (resp. server) to encrypt the data that it sends
- a key used by the client (resp. server) to authenticate the data that it sends
- a key used by the client (resp. server) to initialize the negotiated encryption scheme (if required by this scheme)

It is common practice among designers of security protocols to never use the same key for different purposes. For example, allowing the client and the server to use the same key to encrypt data could enable an attacker to launch a replay attack by sending to the client data that it has itself encrypted.

At this point, all the messages sent over the TCP connection will be encrypted with the negotiated keys. The `ssh` protocol uses messages that are encoded according to the Binary Packet Protocol defined in **RFC 4253**. Each of these messages contains the following information :

- `length` : this is the length of the message in bytes, excluding the MAC and length fields
- `padding length` : this is the number of random bytes that have been added at the end of the message.
- `payload` : the data (after optional compression) passed by the user
- `padding` : random bytes added in each message (at least four) to ensure that the message length is a multiple of the block size used by the negotiated encryption algorithm
- `MAC` : this field is present if a Message Authentication Code has been negotiated for the session (in practice, using `ssh` without authentication is risky and this field should always be present). Note that to compute the MAC, an `ssh` implementation must maintain a message counter. This counter is incremented by

[2] For example, **RFC 4255** describes a DNS record that can be used to associate an `ssh` fingerprint to a DNS name.
[3] The exact algorithms used for the computation of these keys are defined in **RFC 4253**

one every time a message is sent and the MAC is computed with the negotiated authentication algorithm using the MAC key over the concatenation of the message counter and the cleartext message. The message counter is not transmitted, but the recipient can easily recover its value. The `MAC` is computed as $mac = MAC(key, sequence\_number||unencrypted\_message)$ where the key is the negotiated authentication key.

Note: Authenticating messages with HMAC

ssh is one example of a protocol that uses Message Authentication Codes (MAC) to authenticates the messages that are sent. A naive implementation of such a MAC would be to simply use a hash function like SHA-1. However, such a construction would not be safe from a security viewpoint. Internet protocols usually rely on the HMAC construction defined in **RFC 2104**. It works with any hash function (*H*) and a key (*K*). As an example, let us consider HMAC with the SHA-1 hash function. SHA-1 uses 20 bytes blocks and the block size will play an important role in the operation of HMAC. We first require the key to be as long as the block size. Since this key is the output of the key generation algorithm, this is one parameter of this algorithm.

HMAC uses two padding strings : *ipad* (resp. *opad*) which is a string containing 20 times byte `0x36` (resp. byte `0x5C`). The HMAC is then computed as $H[K \oplus opad, H(K \oplus ipad, data)]$ where $\oplus$ denotes the bitwise XOR operation. This computation has been shown to be stronger than the naive $H(K, data)$ against some types of cryptographic attacks.

Among the various features of the `ssh` protocol, it is interesting to mention how users are authenticated by the server. The `ssh` protocol supports the classical username/password authentication (but both the username and the password are transmitted over the secure encrypted channel). In addition, `ssh` supports two authentication mechanisms that rely on public keys. To use the first one, each user needs to generate his/her own public/private key pair and store the public key on the server. To be authenticated, the user needs to sign a message containing his/her public key by using his/her private key. The server can easily verify the validity of the signature since it already knows the user's public key. The second authentication scheme is designed for hosts that trust each other. Each host has a public/private key pair and stores the public keys of the other hosts that it trusts. This is typically used in environments such as university labs where each user could access any of the available computers. If Alice has logged on `computer1` and wants to execute a command on `computer2`, she can create an `ssh` session on this computer and type (again) her password. With the host-based authentication scheme, `computer1` signs a message with its private key to confirm that it has already authenticated Alice. `computer2` would then accept Alice's session without asking her credentials.

The `ssh` protocol includes other features that are beyond the scope of this book. Additional details may be found in [BS2005].

3.8 Transport Layer Security

The Transport Layer Security family of protocols were initially proposed under the name Secure Socket Layer (SSL). The first deployments used this name and many researchers still refer to this security protocol as SSL [FKC1996]. In this chapter, we use the official name that was standardized by the IETF: TLS for *Transport Layer Security*.

The TLS protocol was designed to be usable by a wide range of applications that use the transport layer to reliably exchange information. TLS is mainly used over the TCP protocol. There are variants of TLS that operate over SCTP **RFC 3436** or UDP **RFC 6347**, but these are outside the scope of this chapter.

A TLS session operates over a TCP connection. TLS is responsible for the encryption and the authentication of the SDUs exchanged by the application layer protocol while TCP provides the reliable delivery of this encrypted and authenticated bytestream. TLS is used by many different application layer protocols. The most frequent ones are HTTP (HTTP over TLS is called HTTPS), SMTP **RFC 3207** or POP and IMAP **RFC 2595**, but proprietary application-layer protocols also use TLS [AM2019].

A TLS session can be initiated in two different ways. First, the application can use a dedicated TCP port number for application layer protocol x-over-TLS. This is the solution used by many HTTP servers that reserve port 443 for HTTP over TLS. This solution works, but it requires to reserve two ports for each application : one where the application-layer protocol is used directly over TCP and another one where the application-layer protocol is used over TLS. Given the limited number of TCP ports that are available, this is not a scalable solution. The table below provides some of the reserved port numbers for application layer protocols on top of TLS.

Application	TCP port	TLS port
POP3	110	995
IMAP	143	993
NNTP	119	563
HTTP	80	443
FTP	21	990

A second approach to initiate a TLS session is to use the standard TCP port number for the application layer protocol and define a special message in this protocol to trigger the start of the TLS session. This is the solution used for SMTP with the `STARTTLS` message. This extension to SMTP **RFC 3207** defines the new STARTTLS command. The client can issue this command to indicate to the server that it wants to start a TLS session as shown in the example below captured during a session on port 25.

```
220 server.example.org ESMTP
EHLO client.example.net
250-server.example.org
250-PIPELINING
250-SIZE 250000000
250-ETRN
250-STARTTLS
250-ENHANCEDSTATUSCODES
250-8BITMIME
250 DSN
STARTTLS
220 2.0.0 Ready to start TLS
```

In the remaining parts of this chapter, we assume that the TLS session starts immediately after the establishment of the TCP connection. This corresponds to the deployments on web servers. We focus our presentation of TLS on this very popular use case. TLS is a complex protocol that supports other features than the one used by web servers. A more detailed presentation of TLS may be found in [KPS2002] and [Ristic2015].

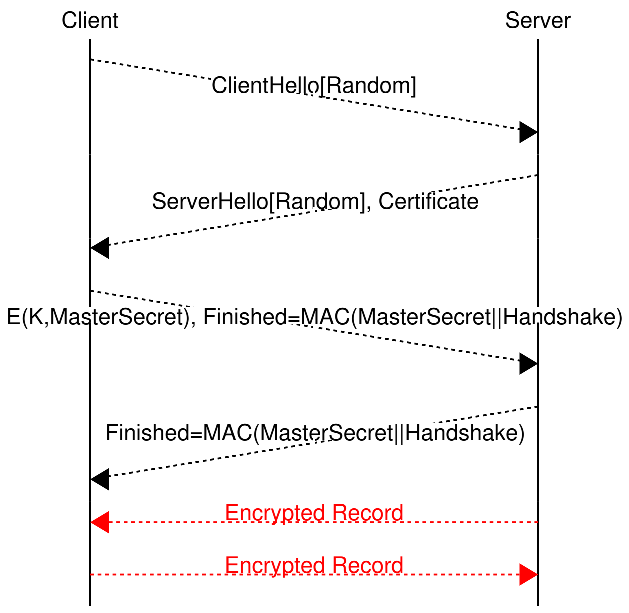
A TLS session is divided in two phases: the handshake and the data transfer. During the handshake, the client and the server negotiate the security parameters and the keys that will be used to secure the data transfer. During the second phase, all the messages exchanged are encrypted and authenticated with the negotiated algorithms and keys.

3.8.1 The TLS handshake

When used to interact with a regular web server, the TLS handshake has three important objectives:

1. Securely negotiate the cryptographic algorithms that will be used by the client and the server over the TLS session
2. Verify that the client interacts with a valid server
3. Securely agree on the keys that will be used to encrypt and authenticate the messages exchanged over the TLS session

The TLS handshake is a four-way handshake illustrated in the figure below.


Client
Server
ClientHello[Random]
ServerHello[Random], Certificate
E(K,MasterSecret), Finished=MAC(MasterSecret||Handshake)
Finished=MAC(MasterSecret||Handshake)
Encrypted Record
Encrypted Record

In a nutshell, the client starts the TLS handshake by proposing a random nonce. The server replies with its random nonce and a certificate that binds its name to a public key. The client generates a MasterSecret that will be used later to derive the session keys and encrypts it with the public key of the server. It also generates a *Finished* message that contains a MAC of all the messages exchanged to allow the server to detect any modification of the messages sent by the client. The server also sends its own *Finished* message. At that point, the client and the server sent encrypted records thanks to the keys derived from the MasterSecret.

Let us first discuss the negotiation of the cryptographic algorithms and parameters. Like all security protocols, TLS includes some agility in its design since new cryptographic algorithms appear over the years and some older algorithms become deprecated once cryptanalysts find flaws. The TLS handshakes starts with the `ClientHello` message that is sent by the client. This message carries the following information :

- *Protocol version number*: this is the version of the TLS protocol supported by the client. The server should use the same version of the TLS protocol as the client, but may opt for an older version. Both versions 1.2 and 1.3 of TLS are deployed today. Older versions are being deprecated.
- *Random number*: security protocols rely on random numbers. The client sends a 32 bytes long random number where usually four of these bytes correspond to the client's clock. This random number is used, together with the server's random number, as a seed to generate the security keys.
- *Cipher suites* : this ordered list contains the set of cryptographic algorithms that are supported by the client, with the most preferred one listed first. In contrast with `ssh` that allows negotiating independent algorithms for encryption, key exchange and authentication, TLS relies on suites that combine these algorithms together. Many cryptographic suites have been defined for TLS. Various recommendations have been published on the security of some of these suites **RFC 7525**.
- *Compression algorithm* : the client may propose the utilization of a specific compression algorithm (e.g. zlib). In theory, compressing the data before encrypting it is an intelligent way to reduce the amount of data exchanged. Unfortunately, its implementation in TLS has caused several security problems [PHG2013]. For this reason, compression is usually disabled in TLS **RFC 7525**.
- *Extensions* : TLS supports various extensions in the `ClientHello` message. These extensions **RFC 6066** are important to allow the protocol to evolve, but many of them go beyond the scope of this chapter.

Note: The `Server Name Indication (SNI)`

The `Server Name Indication (SNI)` extension defined in **RFC 6066** is an important TLS extension for web servers. It is used by the client to indicate the name of the server that it wishes to contact. The IP address associated to this name has been queried from the DNS and used to establish the TCP connection. Why should the client indicate the server name in the TLS `ClientHello` ? The motivation is the same as for the `Host` header line in HTTP/1.0. With the SNI extension, a single TLS server can support several web sites that use different domain names. Thanks to the SNI extension, the server knows the concerned domain name at the start of the TLS session. Without this extension, hosting providers would have been forced use one IP address per TLS-enabled server.

The server replies to the `ClientHello` with several messages:

- the `ServerHello` message that contains the protocol version chosen by the server (assumed to be the same as the client version in this chapter), the 32 random bytes chosen by the server, the *Cipher Suite* selected by the server from the list advertised by the client and a *Session Id*. This *Session Id* is an identifier which is chosen by the server. It identifies the TLS session and the security parameters (algorithms and keys) negotiated for this session. It is used to support session resumption.
- the `Certificate` message provides the certificate (or usually a chain of certificates) that binds a domain name to the public key used by the server. TLS uses the server certificates to authenticate the server. It relies on a Public Key Infrastructure that is composed of a set of root certification authorities that issue certificates to certification authorities that in the end issue certificates to servers. TLS clients are usually configured with the public keys of several root certification authorities and use this information to validate the certificates that they receive from servers. For historical reasons, the TLS certificates are encoded in ASN.1 format. The details of the ASN.1 syntax [Dubuisson2000] are outside the scope of this book.
- the `ServerKeyExchange` message is used by the server to transmit the information that is required to perform the key exchange. The content of this message is function of the selected key exchange algorithm.
- the `ServerHelloDone` indicates that the server has sent all the messages for the first phase of the handshake.

At this point, it is time to describe the TLS key exchange. TLS supports different key exchange mechanisms that can be negotiated as part of the selection of the cipher suite. We focus on two of them to highlight their differences:

- `RSA`. This key exchange algorithm uses the encryption capabilities of the RSA public-key algorithm. The client has validated the server's public key thanks to the `Certificate` message. It then generates a (48 bytes) random number, encrypts it with the server public key and sends the encrypted number to the server in the `ClientKeyExchange` message. The server uses its private key to decrypt the random number. At this point, the client and the server share the same (48 bytes long) secret and use it to derive the secret keys required to encrypt and authenticate data in the second phase. With this key exchange algorithm, the server does not need to send a `ServerKeyExchange` message.
- `DHE_RSA`. This key exchange algorithm is the Ephemeral Diffie Hellman key exchange with RSA signatures to authenticate the key exchange. It operates as a classical authenticated Diffie Hellman key exchange. If this key exchange has been selected by the server, it sends its Diffie Hellman parameters in the `ServerKeyExchange` message and signs them with its private key. The client then continues the key exchange and sends the results of its own computation in the `ClientKeyExchange` message. `DHE_RSA` is thus an authenticated Diffie Hellman key exchange where the initial message is sent by the server (instead of the client as in our first example but since the protocol is symmetric, this does not matter).

An important difference between `DHE_RSA` and `RSA` is their reaction against attacks. `DHE_RSA` is considered by many to be stronger than `RSA` because it supports *Perfect Forward Secrecy*. This property is important against attackers that are able to eavesdrop all the (encrypted) data sent and received by a server. Consider that Terrence is such an attacker that has stored all the packets exchanged by Bob's server during the last six months. If he manages, by any means, to obtain Bob's private key, he will be able to decrypt all the keys used to secure the TLS sessions with Bob's server during this period. With `DHE_RSA`, a similar attack is less devastating. If Terrence knows Bob's private key, he will be able to launch a man-in-the-middle attack against future TLS sessions with Bob's server. However, he will not be able to recover the keys used for all the past sessions that he captured.

Note: Perfect Forward Secrecy

Perfect Forward Secrecy (PFS) is an important property for key exchange protocols. A protocol provides PFS if its design guarantees that the keys used for former sessions will not be compromised even if the private key of the server is compromised. This is a very important property. `DHE_RSA` provides Perfect Forward Secrecy, but the `RSA` key exchange does not provide this property. In practice, `DHE_RSA` is costly from a computational viewpoint. Recent implementations of TLS thus prefer `ECDHE_RSA` or `ECDHE_ECDSA` when Perfect Forward Secrecy is required.

All the information required for the key exchange has now been transmitted. There are two important messages that will be sent by the client and the server to conclude the handshake and start the data transfer phase.

The client sends the `ChangeCipherSpec` message followed by the `Finished` message. The `ChangeCipherSpec` message indicates that the client has received all the information required to generate the security keys for this TLS session. This messages can also appear later in the session to indicate a change in the encryption algorithms that are used, but this usage is outside the scope of this book. The `Finished` message is more important. It confirms to the server that the TLS handshake has been performed correctly and that no attacker has been able to modify the data sent by the client or the server. This is the first message that is encrypted with the selected security keys. It contains a hash of all the messages that were exchanged during the handshake.

The server also sends a `ChangeCipherSpec` message followed by a `Finished` message.

Note: TLS Cipher suites

A TLS cipher suite is usually represented as an ASCII string that starts with TLS and contains the acronym of the key exchange algorithm, the encryption scheme with the key size and its mode of operation and the authentication algorithm. For example, `TLS_DHE_RSA_WITH_AES_128_GCM_SHA256` is a TLS cipher suite that uses the `DHE_RSA` key exchange algorithm with 128 bits AES in GCM mode for encryption and SHA-256 for authentication. The official list of TLS cipher suites is maintained by IANA[1]. The NULL acronym indicates that no algorithm has been specified. For example, `TLS_ECDH_RSA_WITH_NULL_SHA` is a cipher suite that does not use any encryption but still uses the `ECDH_RSA` key exchange and `SHA` for authentication.

3.8.2 The TLS record protocol

The handshake is now finished. The client and the server will exchange authenticated and encrypted records. TLS defines different formats for the records depending on the cryptographic algorithms that have been negotiated for the session. A detailed discussion of these different types of records is outside the scope of this introduction. For illustration, we briefly describe one record format.

As other security protocols, TLS uses different keys to encrypt and authenticate records. These keys are derived from the MasterSecret that is either randomly generated by the client after the `RSA` key exchange or derived from the Diffie Hellman parameters after the `DH_RSA` key exchange. The exact algorithm used to derive the keys is defined in **RFC 5246**.

A TLS record is always composed of four different fields :

- a *Type* that indicates the type of record. The most frequent type is *application data* which corresponds to a record containing encrypted data. The other types are *handshake*, *change_cipher_spec* and *alert*.
- a *Protocol Version* field that indicates the version of the TLS protocol used. This version is composed of two sub fields : a major and a minor version number.
- a *Length* field. A TLS record cannot be longer than 16,384 bytes.
- a *TLSPlainText* that contains the encrypted data

TLS supports several methods to encrypted records. The selected method depends on the cryptographic algorithms that have been negotiated for the TLS session. A detailed presentation of the different methods that can be used to produce

[1] See http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4

the *TLSPlainText* from the user data is outside the scope of this book. As an example, we study one method: Stream Encryption. This method is used with cryptographic algorithms which can operate on a stream of bytes. The method starts with a sequence of bytes provided by the user application: the plain text. The first step is to compute the authentication code to verify the integrity of the data. For this, TLS computes $MAC(SeqNum, Header, PlainText)$ using HMAC where *SeqNum* is a sequence number which is incremented by one for each new TLS record transmitted. The *Header* is the header of the TLS record described above and *PlainText* is the information that needs to be encrypted. Note that the sequence number is maintained at the two endpoints of the TLS session, but it is not transmitted inside the TLS record. This sequence number is used to prevent replay attacks.

Note: MAC-then-encrypt or Encrypt-then-MAC

When secure protocols use Message Authentication and Encryption, they need to specify how these two algorithms are combined. A first solution, which is used by the current version of TLS, is to compute the authentication code and then encrypt both the data and the authentication code. A drawback of this approach is that the receiver of an encrypted TLS record must first attempt to decrypt data that has potentially been modified by an attacker before being able to verify the authenticity of the record. A better approach is for the sender to first encrypt the data and then compute the authentication code over the encrypted data. This is the encrypt-then-MAC approach proposed in **RFC 7366**. With encrypt-then-MAC, the receiver first checks the authentication code before attempting to decrypt the record.

3.8.3 Improving TLS

During the last two decades, the deployment of TLS has continued to grow. The early TLS servers were only used for critical services such as e-commerce websites or online banks. As CPU performance improved, it became much more cost-effective to use TLS to secure non-critical parts of web servers, including the delivery of HTML pages and even video services. There is now a growing number of applications that rely on TLS [AM2019].

In 2013, the statistics collected by the Firefox Telemetry project[3] revealed that 30% of the web pages loaded by Firefox users were done over HTTPS. In October 2019, 80% of the web pages are loaded over HTTPS. In six years, HTTPS became the dominant protocol to access web services. Another look at the deployment of HTTPS on web sites may be found in [Helme2019].

Measurement studies that analyzed the evolution of TLS over the years have identified several important changes in the TLS ecosystem [KRA2018]. First, the preferred cryptographic algorithms have changed. While RC4 was used by 60% of the connections in 2012, its usage has dropped since 2015. AEA started to be deployed in 2013 and is now used for more than 90% of the connections. The deployed versions of TLS have also changed. TLS 1.0 and TLS 1.1 are now rarely used. The deployment of TLS 1.2 started in 2013 and reached 70% of the connections in 2015. Version 1.3 of TLS, that is described below, is also widely deployed.

Another interesting fact is are the key exchange schemes. In 2012, RSA was the dominant solution, used by more than 80% of the observed connections [KRA2018]. In 2013, Edward Snowden revealed the surveillance activities of several governments. These revelations had a huge impact on the Internet community. The IETF, which standardizes Internet protocols, considered in **RFC 7258** that such pervasive monitoring was an attack. Since then, several IETF working groups have developed solutions to counter pervasive monitoring. One of these solutions is to encourage *Perfect Forward Security*. Within TLS, this implies replacing RSA by an authenticated Diffie Hellman key exchange such as ECDHE. Measurements indicate that since summer 2014, ECDHE is more popular than RSA. In 2018, more than 90% of the observed TLS connections used ECDHE.

The last point is the difficulty of deploying TLS servers [KMS2017]. When TLS servers are installed, the system administrator needs to obtain certificates and configure a range of servers. Initially, getting certificates was complex and costly, but initiatives such as https://letsencrypt.org have simplified this workflow.

[3] See https://letsencrypt.org/stats/ for a graph and https://docs.telemetry.mozilla.org/datasets/other/ssl/reference.html for additional information on the dataset

In 2014, the IETF TLS working started to work on the development of version 1.3 of the TLS protocol. Their main objectives [Rescorla2015] for this new version were:

- simplify the design by removing unused or unsafe protocol features
- improve the security of TLS by leveraging the lessons learned from TLS 1.2 and some documented attacks
- improve the privacy of the protocol
- reduce the latency of TLS

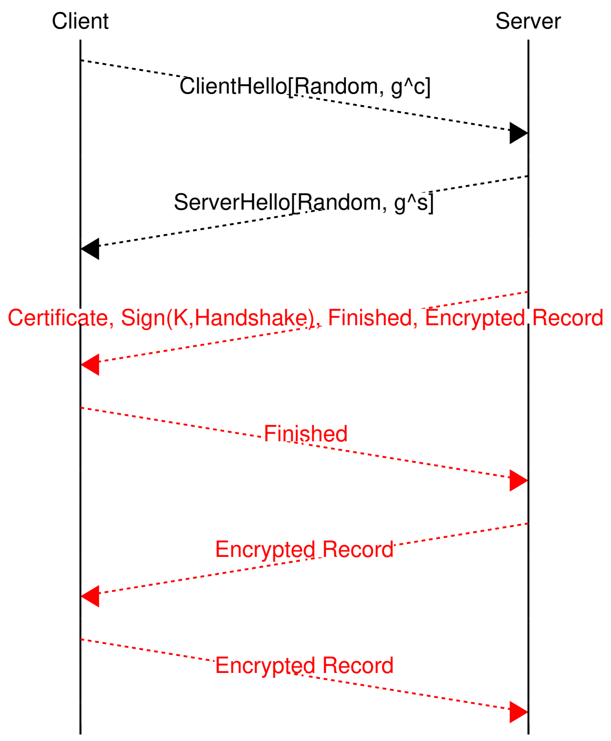
Since 2014, latency has become an important concern for web services. As access networks bandwidth continue to grow, latency is becoming a key factor that affects the performance of interactive web services. With TLS 1.2, the download of a web page requires a minimum of four round-trip-times, one to create the underlying TCP connection, one to exchange the ClientHello/ServerHello, one to exchange the keys and then one to send the HTTP GET and retrieve the response. This can be very long when the server is not near the client. TLS 1.3 aimed at reducing this handshake to one round-trip-time and even zero by placing some of the cryptographic handshake in the TCP handshake. This part will be discussed in the TCP chapter. We focus here on the reducing the TLS handshake to a single round-trip-time.

To simplify both the design and the implementations, TLS 1.3 uses only a small number of cipher suites. Five of them are specified in **RFC 8446** and `TLS_AES_128_GCM_SHA256` must be supported by all implementations. To ensure privacy, all cipher suites that did not provide Perfect Forward Secrecy have been removed. Compression has also been removed from TLS since several attacks on TLS 1.2 exploited its compression capability **RFC 7457**.

Note: Enterprises, privacy and TLS

By supporting only cipher suites that provide Perfect Forward Secrecy in TLS 1.3, the IETF aims at protecting the privacy of users against a wide range of attacks. However, this choice has resulted in intense debates in some enterprises. Some enterprises, notably in financial organizations, have deployed TLS, but wish to be able to decrypt TLS traffic for various security-related activities. These enterprises tried to lobby within the IETF to maintain RSA-based cipher suites that do not provide Perfect Forward Secrecy. Their arguments did not convince the IETF. Eventually, these enterprises moved to ETSI, another standardization body, and convinced them to adopt *entreprise TLS*, a variant of TLS 1.3 that does not provide Perfect Forward Secrecy [eTLS2018].

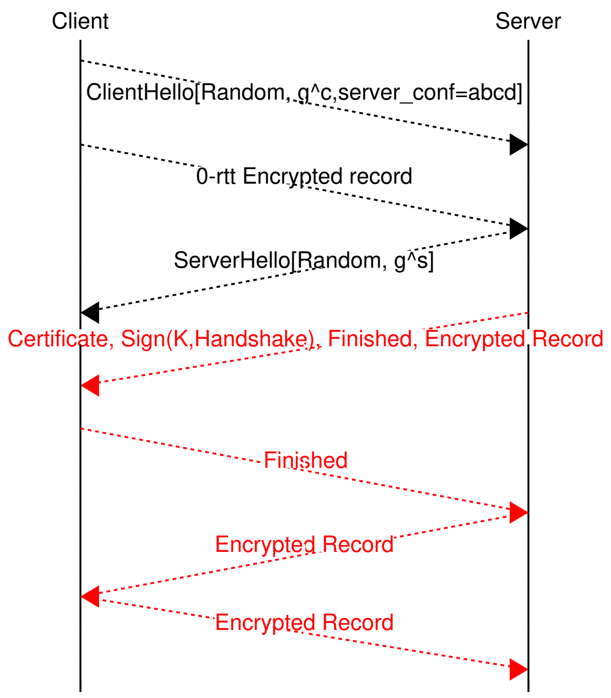
The TLS 1.3 handshake differs from the TLS 1.2 handshake in several ways. First, the TLS 1.3 handshake requires a single round-trip-time when the client connects for the first time to a server. To achieve this, the TLS designers look at the TLS 1.2 handshake in details and found that the first round-trip-time is mainly used to select the set of cryptographic algorithms and the cryptographic exchange scheme that will be used over the TLS session. TLS 1.3 drastically simplifies this negotiation by requiring to use the Diffie Hellman exchange with a small set of possible parameters. This means that the client can guess the parameters used by the server (i.e. the modulus, p and the base g) and immediately start the Diffie Hellman exchange. A simplified version of the TLS 1.3 handshake is shown in the figure below.



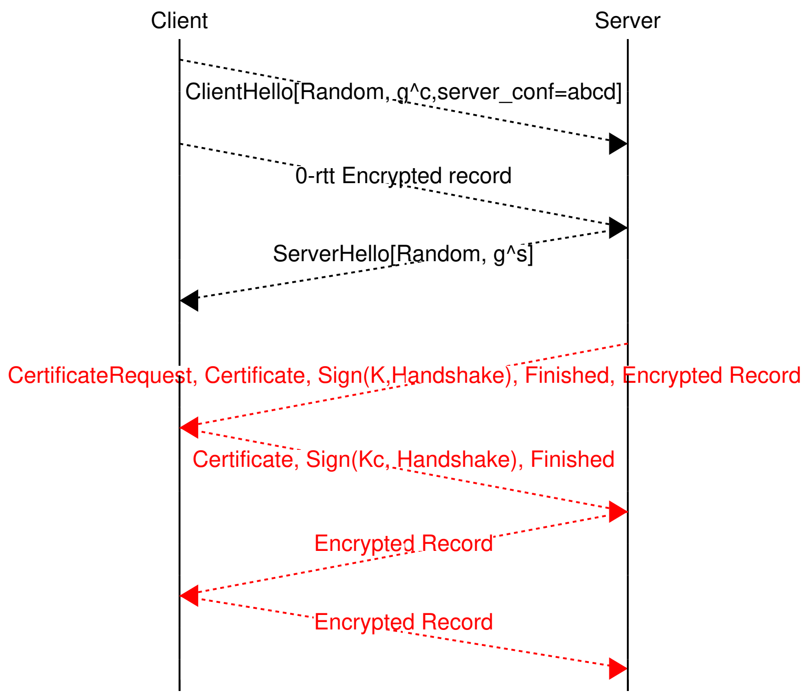
There are several important differences with the TLS 1.2 handshake. First, the Diffie Hellman key exchange is required in TLS 1.3 and this exchange is initiated by the client (before having validated the server identity). To initiate the Diffie Hellman key exchange, the client needs to guess the modulus and the base that can be accepted by the server. Either the client uses standard parameters that most server supports or the client remembers the last modulus/base that it used with this particular server. If the client guessed incorrectly, the server replies with the parameters that it expects and one round-trip-time is lost. When the server sends its *ServerHello*, it already knows the session key. This implies that the server can encrypt all subsequent messages. After one round-trip-time, all data exchanged over the TLS 1.3 session is encrypted and authenticated. In TLS 1.3, the server certificate is encrypted with the session key, as well as the *Finished* message. The server signs the handshake to confirm that it owns the public key of its certificate. If the server wants to send application data, it can already encrypt it and send it to the client. Upon reception of the server Certificate, the client verifies it and checks the signature of the handshake and the *Finished* message. The client confirms the end of the handshake by sending its own *Finished* message. At that time, the client can send encrypted data. This means that the client only had to wait one round-trip-time before sending encrypted data. This is much faster than with TLS 1.2.

For some applications, waiting one round-trip-time before being able to send data is too long. TLS 1.3 allows the client to send encrypted data immediately after the *ClientHello*, without having to wait for the *ServerHello* message. At this point in the handshake, the client cannot know the key that will be derived by the Diffie Hellman key exchange. The trick is that the server and the client need to have previously agreed on a *pre-shared-key*. This key could be negotiated out of band, but usually it was exchanged over a previous TLS session between the client and the server. Both the client and the server can store this key in their cache. When the client creates a new TLS session to a server, it checks whether it already knows a pre-shared key for this server. If so, the client announces the identifier of this key in its *ClientHello* message. Thanks to this identifier, the server can recover the key and use it to decrypt the 0-rtt Encrypted record. A simplified version of the 0-rtt TLS 1.3 handshake[2] is shown in the figure below.

[2] A detailed explanation of the TLS 1.3 handshake may be found at https://tls13.ulfheim.net/


Client
Server
ClientHello[Random, g^c,server_conf=abcd]
0-rtt Encrypted record
ServerHello[Random, g^s]
Certificate, Sign(K,Handshake), Finished, Encrypted Record
Finished
Encrypted Record
Encrypted Record

On the web, TLS clients use certificates to authenticate servers but the clients are not authenticated. However, there are environments such as enterprise networks where servers may need to authenticate clients as well. A popular deployment is to authenticate remote clients who wish to access the enterprise network through a Virtual Private Network service. Some of these services run above TLS (or more precisely a variant of TLS named DTLS that runs above UDP [MoR2004] but is outside the scope of this chapter). In such services, each client is authenticated thanks to a public key and a certificate that is trusted by the servers. To establish a TLS session, such a client needs to prove that it owns the public key associated with the certificate. This is done by the server thanks to the CertificateRequest message. The TLS handshake becomes the following one:


Client
Server
ClientHello[Random, g^c,server_conf=abcd]
0-rtt Encrypted record
ServerHello[Random, g^s]
CertificateRequest, Certificate, Sign(K,Handshake), Finished, Encrypted Record
Certificate, Sign(Kc, Handshake), Finished
Encrypted Record
Encrypted Record

The server sends a CertificatRequest message. The client returns its certificate and signs the Handshake with is private key. This confirms to the server that the client owns the public key indicated in its certificate.

There are many more differences between TLS 1.2 and TLS 1.3. Additional details may be found in their respective specifications, **RFC 5246** and **RFC 8446**.

3.9 Securing the Domain Name System

The Domain Name System provides a critical service in the Internet infrastructure since it maps the domain names that are used by end users onto IP addresses. Since end users rely on names to identify the servers that they connect to, any incorrect information distributed by the DNS would direct end users' connections to invalid destinations. Unfortunately, several attacks of this kind occurred in the past. A detailed analysis of the security threats against the DNS appeared in **RFC 3833**. We consider three of these threats in this section and leave the others to **RFC 3833**.

The first type of attack is *eavesdropping*. An attacker who can capture packets sent to a DNS resolver or a DNS server can gain valuable information about the DNS names that are used by a given end user. If the attacker can capture all the packets sent to a DNS resolver, he/she can collect a lot of meta data about the domain names used by the end user. Preventing this type of attack has not been an objective of the initial design of the DNS. There are currently discussions with the IETF to carry DNS messages over TLS sessions to protect against such attacks. However, these solutions are not yet widely deployed.

The second type of attack is the *man-in-the-middle* attack. Consider that Alice is sending DNS requests to her DNS resolver. Unfortunately, Mallory sits in front of this resolver and can capture and modify all the packets sent by Alice to her resolver. In this case, Mallory can easily modify the DNS responses sent by the resolver to redirect Alice's packets to a different IP address controlled by Mallory. This enables Mallory to observe (and possibly modify) all the packets sent and received by Alice. In practice, executing this attack is not simple since DNS resolvers are usually installed in protected datacenters. However, if Mallory controls the WiFi access point that Alice uses to access the Internet, he could easily modify the packets on this access point and some software packages automate this type of attacks.

If Mallory cannot control a router on the path between Alice and her resolver, she could still launch a different attack. To understand this attack, it is important to correctly understand how the DNS protocol operates and the roles of the different fields of the DNS header which is reproduced in the figure below.

```
                                    1  1  1  1  1  1
      0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                      ID                       |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                    QDCOUNT                    |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                    ANCOUNT                    |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                    NSCOUNT                    |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                    ARCOUNT                    |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Fig. 17: DNS header

The first field of the header is the *Identification* field. When Alice sends a DNS request, she places a 16-bits integer in this field and remembers it. When she receives a response, she uses this *Identification* field to locate the initial DNS request that she sent. The response is only used if its *Identification* matches a pending DNS request (containing the same question).

Mallory has studied the DNS protocol and understands how it works. If he can predict a popular domain for which Alice will regularly send DNS requests, then he can prepare a set of DNS responses that map the name requested by Alice to an IP address controlled by Mallory instead of the legitimate DNS response. Each DNS response has a different *Identification*. Since there are only 65,536 values for the *Identification* field, it is possible for Mallory to send them to Alice hoping that one of them will be received while Alice is waiting for a DNS response with the same identifier. In the past, it was difficult to send 65,536 DNS responses quickly enough. However, with the high speed links that are available today, this is not an issue anymore. A second concern for Mallory is that he must be able to send the DNS responses as if they were coming directly from the DNS resolver. This implies that Mallory must be able to send IP packets that appear to originate from a different address. Although networks should be configured to prevent this type of attack, this is not always the case and there are networks where it is possible for a host to send packets with a different source IP address[1]. If the attack targets a single end user, e.g. Alice, this is annoying for this user. However, if the attacker can target a DNS resolver that serves an entire company or an entire ISP, the impact of the attack can be much larger in particular if the injected DNS response carries a long TTL and thus resides in the resolver's cache for a long period of time.

Fortunately, designers of DNS servers and resolvers have found solutions to mitigate this type of attack. The easiest approach would have been to update the format of the DNS requests and responses to include a larger *Identifier* field. Unfortunately, this elegant solution was not possible with the DNS because the DNS messages do not include any version number that would have enabled such a change. Since the DNS messages are exchanged inside UDP segments, the DNS developers found an alternate solution to counter this attack. There are two ways for the DNS library used by Alice to send her DNS requests. A first solution is to bind one UDP source port and always send the DNS requests from this source port (the destination port is always port `53`). The advantage of this solution is that Alice's DNS library can easily receive the DNS responses by listening to her chosen port. Unfortunately, once the attacker has found the source port used by Alice, he only needs to send 65,536 DNS responses to inject an invalid response. Fortunately, Alice can send her DNS requests in a different way. Instead of using the same source port for all DNS requests, she can use a different source port for each request. In practice, each DNS request will be sent from a different source port. From an implementation viewpoint, this implies that Alice's DNS library will need to listen to one different port number for each pending DNS request. This increases the complexity of her implementation. From a security viewpoint there is a clear benefit since the attacker needs to guess both the 16 bits *Identifier* and the 16 bits *UDP source port* to inject a fake DNS response. To generate all possible DNS responses, the attacker would need to generate almost 2^{32} different messages, which is excessive in today's networks. Most DNS implementations use this second approach to prevent these cache poisoning attacks.

These attacks affect the DNS messages that are exchanged between a client and its resolver or between a resolver and name servers. Another type of attack exploits the possibility of providing several resource records inside one DNS response. A frequent optimization used by DNS servers and resolvers is to include several related resource records in each response. For example, if a client sends a DNS query for an *NS* record, it usually receives in the response both the queried record, i.e. the name of the DNS server that serves the queried domain, and the IP addresses of this server. Some DNS servers return several *NS* records and the associated IP addresses. The *cache poisoning* attack exploits this DNS optimization.

Let us illustrate it on an example. Assume that Alice frequently uses the *example.net* domain and in particular the web server whose name is *www.example.net*. Mallory would like to redirect the TCP connections established by Alice towards *www.example.net* to one IP address that he controls. Assume that Mallory controls the *mallory.net* domain. Mallory can tune the DNS server of his domain and add special DNS records to the responses that it sends. An attack could go roughly as follows. Mallory forces Alice to visit the *www.mallory.net* web site. He can achieve this by sending a spam message to Alice or buying advertisements on a web site visited by Alice and redirect one of these advertisements to *www.mallory.net*. When visiting the advertisement, Alice's DNS resolver will send a DNS request for *www.mallory.net*. Since Mallory control the DNS server, he can easily add in the response a *AAAA* record

[1] See http://spoofer.caida.org/summary.php for an ongoing measurement study that analyses the networks where an attacker could send packets with any source IP address.

that associates *www.example.net* to the IP address controlled by Mallory. If Alice's DNS library does not check the returned response, the cache entry for *www.example.net* will be replaced by the *AAAA* record sent by Mallory.

To cope with these security threats and improve the security of the DNS, the IETF has defined several extensions that are known as DNSSEC. DNSSEC exploits public-key cryptography to authenticate the content of the DNS records that are sent by DNS servers and resolvers. DNSEC is defined in three main documents **RFC 4033**, **RFC 4034**, **RFC 4035**. With DNSSEC, each DNS zone uses one public-private key pair. This key pair is only used to sign and authenticate DNS records. The DNS records are not encrypted and DNSSEC does not provide any confidentiality. Other DNS extensions are being developed to ensure the confidentiality of the information exchanged between a client and its resolvers **RFC 7626**. Some of these extensions exchange DNS records over a TLS session which provides the required confidentiality, but they are not yet deployed and outside the scope of this chapter.

DNSSEC defines four new types of DNS records that are used together to authenticate the information distributed by the DNS.

- the *DNSKEY* record allows storing the public key associated with a zone. This record is encoded as a TLV and includes a *Base64* representation of the key and the identification of the public key algorithm. This allows the *DNSKEY* record to support different public key algorithms.
- the *RRSIG* record is used to encode the signature of a DNS record. This record contains several sub-fields. The most important ones are the algorithm used to generate the signature, the identifier of the public key used to sign the record, the original TTL of the signed record and the validity period for the signature.
- the *DS* record contains a hash of a public key. It is used by a parent zone to certify the public key used by one of its child zones.
- the *NSEC* record is used when non-existent domain names are queried. Its usage will be explained later

The simplest way to understand the operation of DNSSEC is to rely on a simple example. Let us consider the *example.org* domain and assume that Alice wants to retrieve the *AAAA* record for *www.example.org* using DNSSEC.

The security of DNSSEC relies on *anchored keys*. An *anchored key* is a public key that is considered as trusted by a resolver. In our example, we assume that Alice's resolver has obtained the public key of the servers that manage the root zone in a secure way. This key has been distributed outside of the DNS, e.g. it has been published in a newspaper or has been received in a sealed letter.

To obtain an authenticated record for *www.example.org*, Alice's resolver first needs to retrieve the *NS* which is responsible for the *.org* Top-Level Domain (TLD). This record is served by the DNS root server and Alice's resolver can retrieve the signature (*RRSIG* record) for this *NS* record. Since Alice knows the *DNSKEY* of the root, she can verify the validity of this signature.

The next step is to contact *ns.org*, the *NS* responsible for the *.org* TLD to retrieve the *NS* record for the *example.org* domain. This record is accompanied by a *RRSIG* record that authenticates it. This *RRSIG* record is signed with the key of the *.org* domain. Alice's resolver can retrieve this public key as the *DNSKEY* record for the *.org*, but how can it trust this key since it is distributed by using the DNS and could have been modified by attackers ? DNSSEC solves this problem by using the *DS* record that is stored in the parent zone (in this case, the root zone). This record contains a hash of a public key that is signed with a *RRSIG* signature. Since Alice's resolver's trusts the root key, it can validate the signature of the *DS* record for the *.org* domain. It can then retrieve the *DNSKEY* record for this domain from the DNS and compare the hash of this key with the *DS* record. If they match, the public key of the *.org* domain can be trusted. The same technique is used to obtain and validate the key of the *example.org* domain. Once this key is trusted, Alice's resolver can request the *AAAA* record for *www.example.org* and validate its signature.

Thanks to the *DS* record, a resolver can validate the public keys of client zones as long as their is a chain of *DS* -> *DNSKEY* records from an anchored key. If the resolver trusts the public key of the root zone, it can validate all DNS replies for which this chain exists.

There are several details of the operation of DNSSEC that are worth being discussed. First, a server that supports DNSSEC must have a public-private key pair. The public key is distributed with the *DNSKEY* record. The private key is never distributed and it does not even need to be stored on the server that uses the public key. DNSSEC does not require the DNSSEC servers to perform any operation that requires a private key in real time. All the *RRSIG* records can be computed offline, possibly on a different server than the server that returns the DNSSEC replies. The initial motivation for this design choice was the CPU complexity of computing the *RRSIG* signatures for zones that contain millions of records. In the early days of DNSSEC, this was an operational constraint. Today, this is less an issue, but avoiding costly signature operations in real time has two important benefits. First, this reduces the risk of denial of service attacks since an attacker cannot force a DNSSEC server to perform computationally intensive signing operations. Second, the private key can be stored offline, which means that even if an attacker gains access to the DNSSEC server, it cannot retrieve its private key. Using offline signatures for the *RRSIG* records has some practical implications that are reflected in the content of this record. First, each *RRSIG* record contains the original TTL of the signed record. When DNS resolvers cache records, they change the value of the TTL of these cached records and then return the modified records to their clients. When a resolver receives a signed DNS record, it must replace the received TTL of the record with the original TTL (and check that the received TTL is smaller than the original one) before checking the signature. Second, the *RRSIG* records contain a validity period, i.e. a starting time and an ending time for the validity of the signature. This period is specified as two timestamps. This period is only the validity of the signature. It does not affect the TTL of the signed record and is independent from the TTL. In practice, the validity period is important to allow DNS server operators to update their public/private keys. When such a key is changed, e.g. because the private could have been compromised, there is some period of time during which records signed with the two keys coexist in the network. The validity period allows ensuring that old signatures do not remain in DNS caches for ever.

The last record introduced by DNSSEC is the *NSEC* record. It is used to authenticate a negative response returned by a DNS server. If a resolver requests a domain name that is not defined in the zone, the server replies with an error message. The designers of the original version of the DNS thought that these errors would not be very frequent and resolvers were not required to cache those negative responses. However, operational experience showed that queries for invalid domain names are more frequent than initially expected and a large fraction of the load on some servers is caused by repeated queries for invalid names. Typical examples include queries for invalid TLDs to the root DNS servers or queries caused by configuration errors [WF2003]. Current DNS deployments allow resolvers to cache those negative answers to reduce the load on the entire DNS **RFC 2308**.

The simplest way to allow a DNSSEC server to return signed negative responses would be for the server to return a signed response that contains the received query and some information indicating the error. The client could then easily check the validity of the negative response. Unfortunately, this would force the DNSSEC server to generate signatures in real time. This implies that the private key must be stored in the server memory, which leads to risks if an attacker can take control of the server. Furthermore, those signatures are computationally complex and a simple denial of service attack would be to send invalid queries to a DNSSEC server.

Given the above security risks, DNSSEC opted for a different approach that allows the negative replies to be authenticated by using offline signatures. The *NSEC* record exploits the lexicographical ordering of all the domain names. To understand its usage, consider a simple domain that contains three names (the associated *AAAA* and other records that are not shown) :

```
alpha.example.org
beta.example.org
gamma.example.org
```

In this domain, the DNSSEC server adds three *NSEC* records. A *RRSIG* signature is also computed for each of these records.

```
alpha.example.org
alpha.example.org NSEC beta.example.org

beta.example.org
beta.example.org NSEC gamma.example.org

gamma.example.org
gamma.example.org NSEC alpha.example.org
```

If a resolver queries *delta.example.org*, the server will parse its zone. If this name were present, it would have been placed, in lexicographical order, between the *beta.example.org* and the *gamma.example.org* names. To confirm that the *delta.example.org* name does not exist, the server returns the *NSEC* record for *beta.example.org* that indicates that the next valid name after *beta.example.org* is *gamma.example.org*. If the server receives a query for *pi.example.org*, this is the *NSEC* record for *gamma.example.org* that will be returned. Since this record contains a name that is before *pi.example.org* in lexicographical order, this indicates that *pi.example.org* does not exist.

3.10 Internet transport protocols

Transport protocols rely on the service provided by the network layer. On the Internet, the network layer provides a connectionless service. The network layer identifies each (interface of a) host by using an IP address. It enables hosts to transmit packets that contain up to 64 KBytes of payload to any destination reachable through the network. The network layer does not guarantee the delivery of information, cannot detect transmission errors and does not preserve sequence integrity.

Several transport protocols have been designed to provide a richer service to the applications. The two most widely deployed transport protocols on the Internet are the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). A third important transport protocol, the Stream Control Transmission Protocol (SCTP) **RFC 4960** appeared in the early 2000s. It is currently used by some particular applications such as signaling in Voice over IP networks. SCTP was described in the second edition of the e-book.

The Real Time Transport Protocol (RTP), defined in **RFC 3550** is another important protocol that is used by many multimedia applications. It includes functions that belong to the transport layer, but also functions that are related to the encoding of the information. Due to space limitations, we do not discuss it in this section.

3.11 The User Datagram Protocol

The User Datagram Protocol (UDP) is defined in **RFC 768**. It provides an unreliable connectionless transport service on top of the unreliable network layer connectionless service. The main characteristics of the UDP service are :

- the UDP service cannot deliver SDUs that are larger than 65467 bytes[1]
- the UDP service does not guarantee the delivery of SDUs (losses can occur and SDUs can arrive out-of-sequence)
- the UDP service will not deliver a corrupted SDU to the destination

Compared to the connectionless network layer service, the main advantage of the UDP service is that it allows several applications running on a host to exchange SDUs with several other applications running on remote hosts. Let us consider two hosts, e.g. a client and a server. The network layer service allows the client to send information to the server, but if an application running on the client wants to contact a particular application running on the server, then an additional addressing mechanism is required other than the IP address that identifies a host, in order to differentiate the application running on a host. This additional addressing is provided by *port numbers*. When a server using UDP is enabled on a host, this server registers a *port number*. This *port number* will be used by the clients to contact the server process via UDP.

The figure below shows a typical usage of the UDP port numbers. The client process uses port number *1234* while the server process uses port number *5678*. When the client sends a request, it is identified as originating from port number *1234* on the client host and destined to port number *5678* on the server host. When the server process replies to this request, the server's UDP implementation will send the reply as originating from port *5678* on the server host and destined to port *1234* on the client host.

[1] This limitation is due to the fact that the network layer cannot transport packets that are larger than 64 KBytes. As UDP does not include any segmentation/reassembly mechanism, it cannot split a SDU before sending it. The UDP header consumes 8 bytes and the IPv6 header 60. With IPv4, the IPv4 header only consumes 20 bytes and thus the maximum UDP payload size is 65507 bytes.

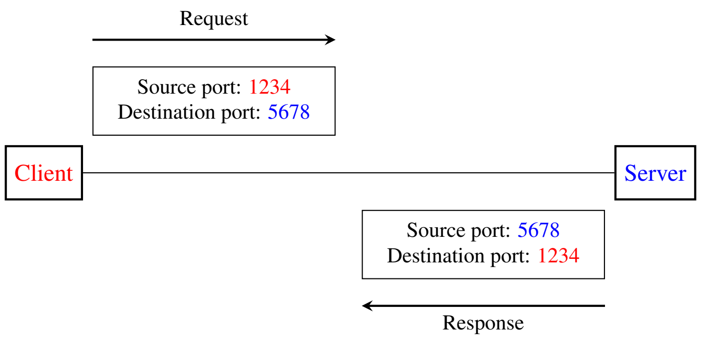


Fig. 18: Usage of the UDP port numbers

UDP uses a single segment format shown in the figure below.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Length              |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 19: UDP Header Format

The UDP header contains four fields :

- a 16 bits source port
- a 16 bits destination port
- a 16 bits length field
- a 16 bits checksum

As the port numbers are encoded as a 16 bits field, there can be up to only 65535 different server processes that are bound to a different UDP port at the same time on a given server. In practice, this limit is never reached. However, it is worth noticing that most implementations divide the range of allowed UDP port numbers into three different ranges :

- the privileged port numbers (1 < port < 1024)
- the ephemeral port numbers (officially[2] 49152 <= port <= 65535)
- the registered port numbers (officially 1024 <= port < 49152)

In most Unix variants, only processes having system administrator privileges can be bound to port numbers smaller than *1024*. Well-known servers such as *DNS*, *NTP* or *RPC* use privileged port numbers. When a client needs to use UDP, it usually does not require a specific port number. In this case, the UDP implementation will allocate the first available port number in the ephemeral range. The range of registered port numbers should be used by servers. In theory, developers of network servers should register their port number officially through IANA[3], but few developers do this.

Note: Computation of the UDP checksum

The checksum of the UDP segment is computed over :

[2] A discussion of the ephemeral port ranges used by different TCP/UDP implementations may be found in http://www.ncftp.com/ncftpd/doc/misc/ephemeral_ports.html

[3] The complete list of allocated port numbers is maintained by IANA . It may be downloaded from http://www.iana.org/assignments/port-numbers

- a pseudo header **RFC 2460** containing the source address, the destination address, the packet length encoded as a 32 bits number and a 32 bits bit field containing the three most significant bytes set to 0 and the low order byte set to 17
- the entire UDP segment, including its header

Several types of applications rely on UDP. As a rule of thumb, UDP is used for applications where delay must be minimized or losses can be recovered by the application itself. A first class of the UDP-based applications are applications where the client sends a short request and expects a quick and short answer. The *DNS* is an example of a UDP application that is often used in the wide area. However, in local area networks, many distributed systems rely on Remote Procedure Call (*RPC*) that is often used on top of UDP. In Unix environments, the Network File System (*NFS*) is built on top of RPC and runs frequently on top of UDP. A second class of UDP-based applications are the interactive computer games that need to frequently exchange small messages, such as the player's location or their recent actions. Many of these games use UDP to minimize the delay and can recover from losses. A third class of applications are multimedia applications such as interactive Voice over IP or interactive Video over IP. These interactive applications expect a delay shorter than about 200 milliseconds between the sender and the receiver and can recover from losses directly inside the application.

3.12 The Transmission Control Protocol

The Transmission Control Protocol (TCP) was initially defined in **RFC 793**. Several parts of the protocol have been improved since the publication of the original protocol specification[1]. However, the basics of the protocol remain and an implementation that only supports **RFC 793** should inter-operate with today's implementation.

TCP provides a reliable bytestream, connection-oriented transport service on top of the unreliable connectionless network service provided by *IP*. TCP is used by a large number of applications, including :

- Email (*SMTP*, *POP*, *IMAP*)
- World wide web (*HTTP*, . . .)
- Most file transfer protocols (*ftp*, peer-to-peer file sharing applications , . . .)
- remote computer access : *telnet*, *ssh*, *X11*, *VNC*, . . .
- non-interactive multimedia applications (flash, . . .)

On the global Internet, most of the applications used in the wide area rely on TCP. Many studies[2] have reported that TCP was responsible for more than 90% of the data exchanged in the global Internet.

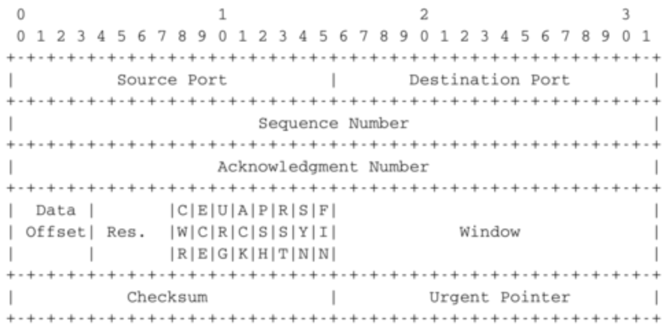
To provide this service, TCP relies on a simple segment format that is shown in the figure below. Each TCP segment contains a header described below and, optionally, a payload. The default length of the TCP header is twenty bytes, but some TCP headers contain options.

A TCP header contains the following fields :

- the *source and destination ports*. The source and destination ports play an important role in TCP, as they allow the identification of the connection to which a TCP segment belongs. When a client opens a TCP connection, it typically selects an ephemeral TCP port number as its source port and contacts the server by using the server's port number. All the segments that are sent by the client on this connection have the same source and destination ports. The server sends segments that contain as source (resp. destination) port, the destination (resp. source) port of the segments sent by the client (see figure *Utilization of the TCP source and destination ports*). A TCP connection is always identified by four pieces of information :

[1] A detailed presentation of all standardization documents concerning TCP may be found in **RFC 4614**

[2] Several researchers have analyzed the utilization of TCP and UDP in the global Internet. Most of these studies have been performed by collecting all the packets transmitted over a given link during a period of a few hours or days and then analyzing their headers to infer the transport protocol used, the type of application, . . . Recent studies include http://www.caida.org/research/traffic-analysis/tcpudpratio/, https://research.sprintlabs.com/packstat/packetoverview.php or http://www.nanog.org/meetings/nanog43/presentations/Labovitz_internetstats_N43.pdf

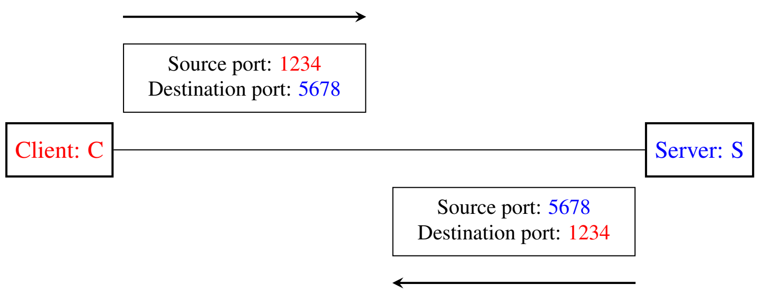


```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |       |C|E|U|A|P|R|S|F|                               |
| Offset| Res.  |W|C|R|C|S|S|Y|I|            Window             |
|       |       |R|E|G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 20: TCP header format

  - the address of the client
  - the address of the server
  - the port chosen by the client
  - the port chosen by the server

- the *sequence number* (32 bits), *acknowledgment number* (32 bits) and *window* (16 bits) fields are used to provide a reliable data transfer, using a window-based protocol. In a TCP bytestream, each byte of the stream consumes one sequence number. Their usage is described in more detail in section *TCP reliable data transfer*
- the *Urgent pointer* is used to indicate that some data should be considered as urgent in a TCP bytestream. However, it is rarely used in practice and will not be described here. Additional details about the utilization of this pointer may be found in **RFC 793**, **RFC 1122** or [Stevens1994]
- the flags field contains a set of bit flags that indicate how a segment should be interpreted by the TCP entity receiving it :
  - the *SYN* flag is used during connection establishment
  - the *FIN* flag is used during connection release
  - the *RST* is used in case of problems or when an invalid segment has been received
  - when the *ACK* flag is set, it indicates that the *acknowledgment* field contains a valid number. Otherwise, the content of the *acknowledgment* field must be ignored by the receiver
  - the *URG* flag is used together with the *Urgent pointer*
  - the *PSH* flag is used as a notification from the sender to indicate to the receiver that it should pass all the data it has received to the receiving process. However, in practice TCP implementations do not allow TCP users to indicate when the *PSH* flag should be set.
- the *checksum* field contains the value of the Internet checksum computed over the entire TCP segment and a pseudo-header as with UDP
- the *Reserved* field was initially reserved for future utilization. It is now used by **RFC 3168**.
- the *TCP Header Length* (THL) or *Data Offset* field is a four bits field that indicates the size of the TCP header in 32 bit words. The maximum size of the TCP header is thus 64 bytes.
- the *Optional header extension* is used to add optional information to the TCP header. Thanks to this header extension, it is possible to add new fields to the TCP header that were not planned in the original specification. This allowed TCP to evolve since the early eighties. The details of the TCP header extension are explained in sections *TCP connection establishment* and *TCP reliable data transfer*.



Established TCP connections on Client

Local IP	Remote IP	Local Port	Remote Port
C	S	1234	5678

Established TCP connections on Server

Local IP	Remote IP	Local Port	Remote Port
S	C	5678	1234

Fig. 21: Utilization of the TCP source and destination ports

The rest of this section is organized as follows. We first explain the establishment and the release of a TCP connection, then we discuss the mechanisms that are used by TCP to provide a reliable bytestream service. We end the section with a discussion of network congestion and explain the mechanisms that TCP uses to avoid congestion collapse.

3.12.1 TCP connection establishment

A TCP connection is established by using a three-way handshake. The connection establishment phase uses the *sequence number*, the *acknowledgment number* and the *SYN* flag. When a TCP connection is established, the two communicating hosts negotiate the initial sequence number to be used in both directions of the connection. For this, each TCP entity maintains a 32 bits counter, which is supposed to be incremented by one at least every 4 microseconds and after each connection establishment[3]. When a client host wants to open a TCP connection with a server host, it creates a TCP segment with :

- the *SYN* flag set
- the *sequence number* set to the current value of the 32 bits counter of the client host's TCP entity

Upon reception of this segment (which is often called a *SYN segment*), the server host replies with a segment containing :

- the *SYN* flag set
- the *sequence number* set to the current value of the 32 bits counter of the server host's TCP entity
- the *ACK* flag set
- the *acknowledgment number* set to the *sequence number* of the received *SYN* segment incremented by 1 $\pmod{2^{32}}$. When a TCP entity sends a segment having *x+1* as acknowledgment number, this indicates that it has received all data up to and including sequence number *x* and that it is expecting data having sequence number *x+1*. As the *SYN* flag was set in a segment having sequence number *x*, this implies that setting the *SYN* flag in a segment consumes one sequence number.

This segment is often called a *SYN+ACK* segment. The acknowledgment confirms to the client that the server has correctly received the *SYN* segment. The *sequence number* of the *SYN+ACK* segment is used by the server host to verify that the *client* has received the segment. Upon reception of the *SYN+ACK* segment, the client host replies with a segment containing :

- the *ACK* flag set

[3] This 32 bits counter was specified in **RFC 793**. A 32 bits counter that is incremented every 4 microseconds wraps in about 4.5 hours. This period is much larger than the Maximum Segment Lifetime that is fixed at 2 minutes in the Internet (**RFC 791**, **RFC 1122**).

- the *acknowledgment number* set to the *sequence number* of the received *SYN+ACK* segment incremented by 1 $(\bmod 2^{32})$

At this point, the TCP connection is open and both the client and the server are allowed to send TCP segments containing data. This is illustrated in the figure below.

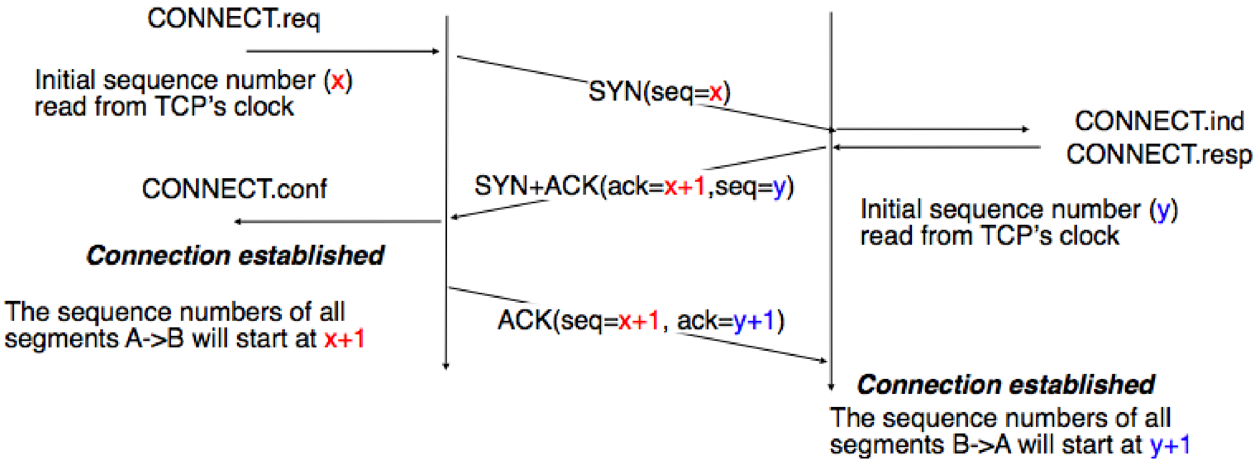

CONNECT.req
Initial sequence number (x) read from TCP's clock
SYN(seq=x)
CONNECT.ind
CONNECT.resp
SYN+ACK(ack=x+1,seq=y)
CONNECT.conf
Initial sequence number (y) read from TCP's clock
Connection established
The sequence numbers of all segments A->B will start at x+1
ACK(seq=x+1, ack=y+1)
Connection established
The sequence numbers of all segments B->A will start at y+1

Fig. 22: Establishment of a TCP connection

In the figure above, the connection is considered to be established by the client once it has received the *SYN+ACK* segment, while the server considers the connection to be established upon reception of the *ACK* segment. The first data segment sent by the client (server) has its *sequence number* set to *x+1* (resp. *y+1*).

Note: Computing TCP's initial sequence number

In the original TCP specification **RFC 793**, each TCP entity maintained a clock to compute the initial sequence number (*ISN*) placed in the *SYN* and *SYN+ACK* segments. This made the ISN predictable and caused a security issue. The typical security problem was the following. Consider a server that trusts a host based on its IP address and allows the system administrator to log in from this host without giving a password[4]. Consider now an attacker who knows this particular configuration and is able to send IP packets having the client's address as source. He can send fake TCP segments to the server, but does not receive the server's answers. If he can predict the *ISN* that is chosen by the server, he can send a fake *SYN* segment and shortly after the fake *ACK* segment confirming the reception of the *SYN+ACK* segment sent by the server. Once the TCP connection is open, he can use it to send any command to the server. To counter this attack, current TCP implementations add randomness to the *ISN*. One of the solutions, proposed in **RFC 1948** is to compute the *ISN* as

```
ISN = M + H(localhost, localport, remotehost, remoteport, secret).
```

where *M* is the current value of the TCP clock and *H* is a cryptographic hash function. *localhost* and *remotehost* (resp. *localport* and *remoteport*) are the IP addresses (port numbers) of the local and remote host and *secret* is a random number only known by the server. This method allows the server to use different ISNs for different clients at the same time. Measurements performed with the first implementations of this technique showed that it was difficult to implement it correctly, but today's TCP implementation now generate good ISNs.

A server could, of course, refuse to open a TCP connection upon reception of a *SYN* segment. This refusal may be due to various reasons. There may be no server process that is listening on the destination port of the *SYN* segment. The server could always refuse connection establishments from this particular client (e.g. due to security reasons) or

[4] On many departmental networks containing Unix workstations, it was common to allow users on one of the hosts to use `rlogin` **RFC 1258** to run commands on any of the workstations of the network without giving any password. In this case, the remote workstation "authenticated" the client host based on its IP address. This was a bad practice from a security viewpoint.

the server may not have enough resources to accept a new TCP connection at that time. In this case, the server would reply with a TCP segment having its *RST* flag set and containing the *sequence number* of the received *SYN* segment incremented by one as its *acknowledgment number*. This is illustrated in the figure below. We discuss the other usages of the TCP *RST* flag later (see *TCP connection release*).

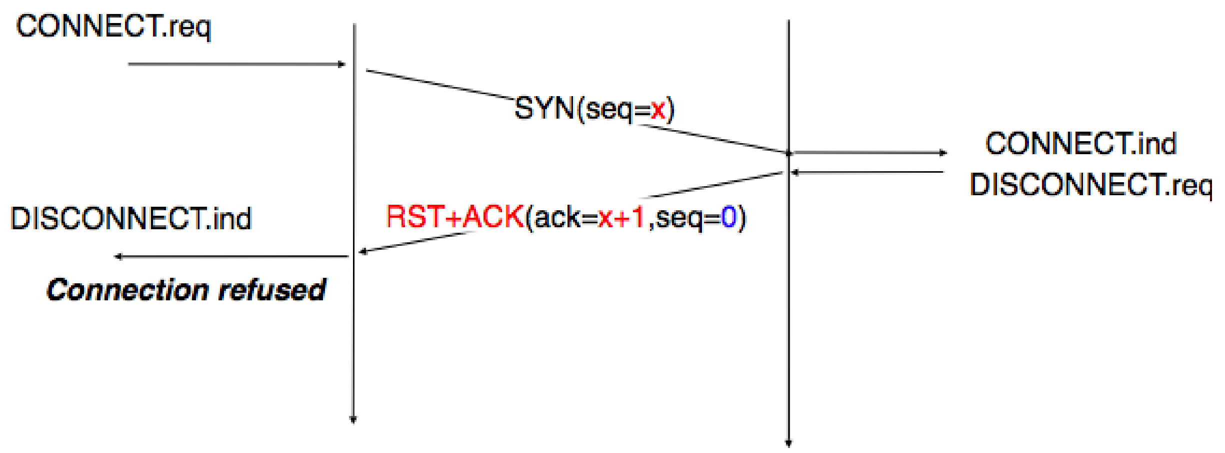


Fig. 23: TCP connection establishment rejected by peer

TCP connection establishment can be described as the four state Finite State Machine shown below. In this FSM, *!X* (resp. *?Y*) indicates the transmission of segment *X* (resp. reception of segment *Y*) during the corresponding transition. *Init* is the initial state.

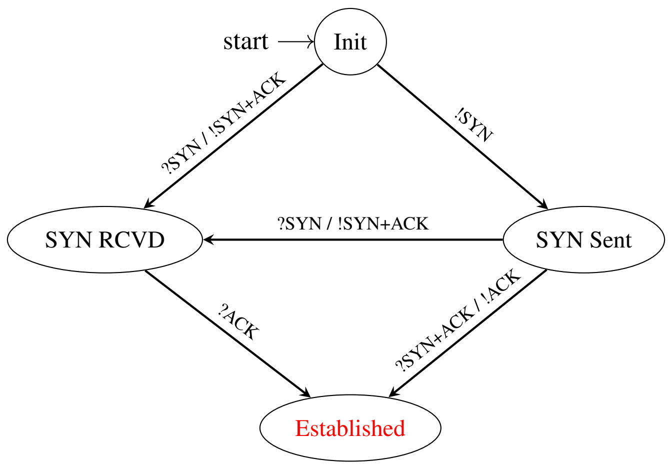


Fig. 24: TCP FSM for connection establishment

A client host starts in the *Init* state. It then sends a *SYN* segment and enters the *SYN Sent* state where it waits for a *SYN+ACK* segment. Then, it replies with an *ACK* segment and enters the *Established* state where data can be exchanged. On the other hand, a server host starts in the *Init* state. When a server process starts to listen to a destination port, the underlying TCP entity creates a TCP control block and a queue to process incoming *SYN* segments. Upon reception of a *SYN* segment, the server's TCP entity replies with a *SYN+ACK* and enters the *SYN RCVD* state. It remains in this state until it receives an *ACK* segment that acknowledges its *SYN+ACK* segment, with this it then enters the *Established* state.

Apart from these two paths in the TCP connection establishment FSM, there is a third path that corresponds to the case when both the client and the server send a *SYN* segment to open a TCP connection[5]. In this case, the client and the server send a *SYN* segment and enter the *SYN Sent* state. Upon reception of the *SYN* segment sent by the other host, they reply by sending a *SYN+ACK* segment and enter the *SYN RCVD* state. The *SYN+ACK* that arrives from the other host allows it to transition to the *Established* state. The figure below illustrates such a simultaneous establishment of a TCP connection.

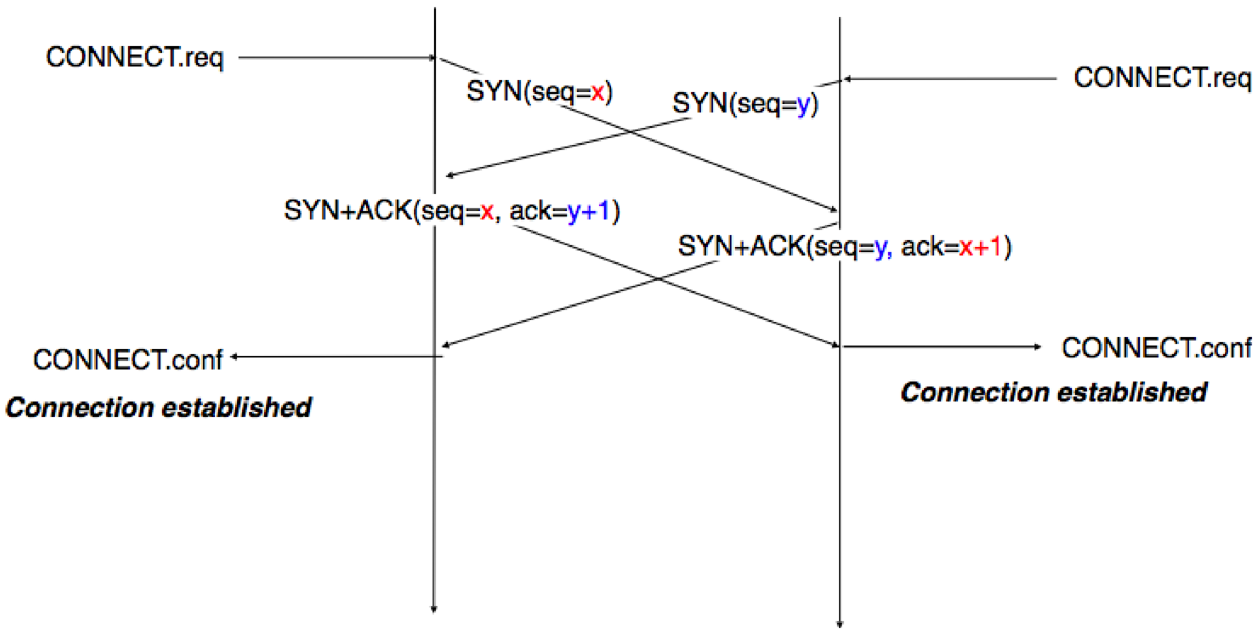


Fig. 25: Simultaneous establishment of a TCP connection

Denial of Service attacks

When a TCP entity opens a TCP connection, it creates a Transmission Control Block (*TCB*). The TCB contains the entire state that is maintained by the TCP entity for each TCP connection. During connection establishment, the TCB contains the local IP address, the remote IP address, the local port number, the remote port number, the current local sequence number and the last sequence number received from the remote entity. Until the mid 1990s, TCP implementations had a limit on the number of TCP connections that could be in the *SYN RCVD* state at a given time. Many implementations set this limit to about 100 TCBs. This limit was considered sufficient even for heavily load http servers given the small delay between the reception of a *SYN* segment and the reception of the *ACK* segment that terminates the establishment of the TCP connection. When the limit of 100 TCBs in the *SYN Rcvd* state is reached, the TCP entity discards all received TCP *SYN* segments that do not correspond to an existing TCB.

This limit of 100 TCBs in the *SYN Rcvd* state was chosen to protect the TCP entity from the risk of overloading its memory with too many TCBs in the *SYN Rcvd* state. However, it was also the reason for a new type of Denial of Service (DoS) attack **RFC 4987**. A DoS attack is defined as an attack where an attacker can render a resource unavailable in the network. For example, an attacker may cause a DoS attack on a 2 Mbps link used by a company by sending more than 2 Mbps of packets through this link. In this case, the DoS attack was more subtle. As a TCP entity discards all received *SYN* segments as soon as it has 100 TCBs in the *SYN Rcvd* state, an attacker simply had to send a few 100 *SYN* segments every second to a server and never reply to the received *SYN+ACK* segments. To avoid being caught, attackers were of course sending these *SYN* segments with a different address than their own IP address[6]. On most TCP implementations, once a TCB entered the *SYN Rcvd* state, it remained in this state for

[5] Of course, such a simultaneous TCP establishment can only occur if the source port chosen by the client is equal to the destination port chosen by the server. This may happen when a host can serve both as a client as a server or in peer-to-peer applications when the communicating hosts do not use ephemeral port numbers.

several seconds, waiting for a retransmission of the initial *SYN* segment. This attack was later called a *SYN flood* attack and the servers of the ISP named panix were among the first to be affected by this attack.

To avoid the *SYN flood* attacks, recent TCP implementations no longer enter the *SYN Rcvd* state upon reception of a *SYN segment*. Instead, they reply directly with a *SYN+ACK* segment and wait until the reception of a valid *ACK*. This implementation trick is only possible if the TCP implementation is able to verify that the received *ACK* segment acknowledges the *SYN+ACK* segment sent earlier without storing the initial sequence number of this *SYN+ACK* segment in a TCB. The solution to solve this problem, which is known as SYN cookies is to compute the 32 bits of the *ISN* as follows :

- the high order bits contain the low order bits of a counter that is incremented slowly
- the low order bits contain a hash value computed over the local and remote IP addresses and ports and a random secret only known to the server

The advantage of the SYN cookies is that by using them, the server does not need to create a *TCB* upon reception of the *SYN* segment and can still check the returned *ACK* segment by recomputing the *SYN cookie*. The main disadvantage is that they are not fully compatible with the TCP options. This is why they are not enabled by default on a typical system.

Retransmitting the first *SYN* segment

As IP provides an unreliable connectionless service, the *SYN* and *SYN+ACK* segments sent to open a TCP connection could be lost. Current TCP implementations start a retransmission timer when they send the first *SYN* segment. This timer is often set to three seconds for the first retransmission and then doubles after each retransmission **RFC 2988**. TCP implementations also enforce a maximum number of retransmissions for the initial *SYN* segment.

As explained earlier, TCP segments may contain an optional header extension. In the *SYN* and *SYN+ACK* segments, these options are used to negotiate some parameters and the utilization of extensions to the basic TCP specification.

The first parameter which is negotiated during the establishment of a TCP connection is the Maximum Segment Size (*MSS*). The MSS is the size of the largest segment that a TCP entity is able to process. According to **RFC 879**, all TCP implementations must be able to receive TCP segments containing 536 bytes of payload. However, most TCP implementations are able to process larger segments. Such TCP implementations use the TCP MSS Option in the *SYN/SYN+ACK* segment to indicate the largest segment they are able to process. The MSS value indicates the maximum size of the payload of the TCP segments. The client (resp. server) stores in its *TCB* the MSS value announced by the server (resp. the client).

Another utilization of TCP options during connection establishment is to enable TCP extensions. For example, consider **RFC 1323** (which is discussed in *TCP reliable data transfer*). **RFC 1323** defines TCP extensions to support timestamps and larger windows. If the client supports **RFC 1323**, it adds a **RFC 1323** option to its *SYN* segment. If the server understands this **RFC 1323** option and wishes to use it, it replies with a **RFC 1323** option in the *SYN+ACK* segment and the extension defined in **RFC 1323** is used throughout the TCP connection. Otherwise, if the server's *SYN+ACK* does not contain the **RFC 1323** option, the client is not allowed to use this extension and the corresponding TCP header options throughout the TCP connection. TCP's option mechanism is flexible and it allows the extension of TCP while maintaining compatibility with older implementations.

The TCP options are encoded by using a Type Length Value format where :

- the first byte indicates the *type* of the option.
- the second byte indicates the total length of the option (including the first two bytes) in bytes
- the last bytes are specific for each type of option

[6] Sending a packet with a different source IP address than the address allocated to the host is called sending a *spoofed packet*.

RFC 793 defines the Maximum Segment Size (MSS) TCP option that must be understood by all TCP implementations. This option (type 2) has a length of 4 bytes and contains a 16 bits word that indicates the MSS supported by the sender of the *SYN* segment. The MSS option can only be used in TCP segments having the *SYN* flag set.

RFC 793 also defines two special options that must be supported by all TCP implementations. The first option is *End of option*. It is encoded as a single byte having value *0x00* and can be used to ensure that the TCP header extension ends on a 32 bits boundary. The *No-Operation* option, encoded as a single byte having value *0x01*, can be used when the TCP header extension contains several TCP options that should be aligned on 32 bit boundaries. All other options[7] are encoded using the TLV format.

Note: The robustness principle

The handling of the TCP options by TCP implementations is one of the many applications of the *robustness principle* which is usually attributed to Jon Postel and is often quoted as *"Be liberal in what you accept, and conservative in what you send"* **RFC 1122**.

Concerning the TCP options, the robustness principle implies that a TCP implementation should be able to accept TCP options that it does not understand, in particular in received *SYN* segments, and that it should be able to parse any received segment without crashing, even if the segment contains an unknown TCP option. Furthermore, a server should not send in the *SYN+ACK* segment or later, options that have not been proposed by the client in the *SYN* segment.

3.12.2 TCP reliable data transfer

The original TCP data transfer mechanisms were defined in **RFC 793**. Based on the experience of using TCP on the growing global Internet, this part of the TCP specification has been updated and improved several times, always while preserving the backward compatibility with older TCP implementations. In this section, we review the main data transfer mechanisms used by TCP.

TCP is a window-based transport protocol that provides a bi-directional byte stream service. This has several implications on the fields of the TCP header and the mechanisms used by TCP. The three fields of the TCP header are :

- *sequence number*. TCP uses a 32 bits sequence number. The *sequence number* placed in the header of a TCP segment containing data is the sequence number of the first byte of the payload of the TCP segment.
- *acknowledgment number*. TCP uses cumulative positive acknowledgments. Each TCP segment contains the *sequence number* of the next byte that the sender of the acknowledgment expects to receive from the remote host. In theory, the *acknowledgment number* is only valid if the *ACK* flag of the TCP header is set. In practice, almost all[8] TCP segments have their *ACK* flag set.
- *window*. a TCP receiver uses this 16 bits field to indicate the current size of its receive window expressed in bytes.

Note: The Transmission Control Block

For each established TCP connection, a TCP implementation must maintain a Transmission Control Block (*TCB*). A TCB contains all the information required to send and receive segments on this connection **RFC 793**. This includes[9] :

- the local IP address
- the remote IP address

[7] The full list of all TCP options may be found at http://www.iana.org/assignments/tcp-parameters/

[8] In practice, only the *SYN* segment do not have their *ACK* flag set.

[9] A complete TCP implementation contains additional information in its TCB, notably to support the *urgent* pointer. However, this part of TCP is not discussed in this book. Refer to **RFC 793** and **RFC 2140** for more details about the TCB.

- the local TCP port number
- the remote TCP port number
- the current state of the TCP FSM
- the *maximum segment size* (MSS)
- *snd.nxt* : the sequence number of the next byte in the byte stream (the first byte of a new data segment that you send uses this sequence number)
- *snd.una* : the earliest sequence number that has been sent but has not yet been acknowledged
- *snd.wnd* : the current size of the sending window (in bytes)
- *rcv.nxt* : the sequence number of the next byte that is expected to be received from the remote host
- *rcv.wnd* : the current size of the receive window advertised by the remote host
- *sending buffer* : a buffer used to store all unacknowledged data
- *receiving buffer* : a buffer to store all data received from the remote host that has not yet been delivered to the user. Data may be stored in the *receiving buffer* because either it was not received in sequence or because the user is too slow to process it

The original TCP specification can be summarized as a transport protocol that provides a byte stream service and uses *go-back-n* with a *selective-repeat* reception strategy.

To send new data on an established connection, a TCP entity performs the following operations on the corresponding TCB. It first checks that the *sending buffer* does not contain more data than the receive window advertised by the remote host (*rcv.wnd*). If the window is not full, up to *MSS* bytes of data are placed in the payload of a TCP segment. The *sequence number* of this segment is the sequence number of the first byte of the payload. It is set to the first available sequence number : *snd.nxt* and *snd.nxt* is incremented by the length of the payload of the TCP segment. The *acknowledgment number* of this segment is set to the current value of *rcv.nxt* and the *window* field of the TCP segment is computed based on the current occupancy of the *receiving buffer*. The data is kept in the *sending buffer* in case it needs to be retransmitted later.

When a TCP segment with the *ACK* flag set is received, the following operations are performed. *rcv.wnd* is set to the value of the *window* field of the received segment. The *acknowledgment number* is compared to *snd.una*. The newly acknowledged data is removed from the *sending buffer* and *snd.una* is updated. If the TCP segment contained data, the *sequence number* is compared to *rcv.nxt*. If they are equal, the segment was received in sequence and the data can be delivered to the user and *rcv.nxt* is updated. The contents of the *receiving buffer* is checked to see whether other data already present in this buffer can be delivered in sequence to the user. If so, *rcv.nxt* is updated again. Otherwise, the segment's payload is placed in the *receiving buffer*.

Segment transmission strategies

In a transport protocol such as TCP that offers a bytestream, a practical issue that was left as an implementation choice in **RFC 793** is to decide when a new TCP segment containing data must be sent. There are two simple and extreme implementation choices. The first implementation choice is to send a TCP segment as soon as the user has requested the transmission of some data. This allows TCP to provide a low delay service. However, if the user is sending data one byte at a time, TCP would place each user byte in a segment containing 20 bytes of TCP header[10]. This is a huge overhead that is not acceptable in wide area networks. A second simple solution would be to only transmit a new TCP segment once the user has produced MSS bytes of data. This solution reduces the overhead, but at the cost of a potentially very high delay.

[10] This TCP segment is then placed in an IP header. We describe IPv6 in the next chapter. The minimum size of the IPv6 (resp. IPv4) header is 40 bytes (resp. 20 bytes).

An elegant solution to this problem was proposed by John Nagle in **RFC 896**. John Nagle observed that the overhead caused by the TCP header was a problem in wide area connections, but less in local area connections where the available bandwidth is usually higher. He proposed the following rules to decide to send a new data segment when a new data has been produced by the user or a new *ack* segment has been received.

```
if rcv.wnd >= MSS and len(data) >= MSS:
    send one MSS-sized segment
else:
    if there are unacknowledged data:
        place data in buffer until acknowledgment has been received
    else:
        send one TCP segment containing all buffered data
```

The first rule ensures that a TCP connection used for bulk data transfer always sends full TCP segments. The second rule sends one partially filled TCP segment every round-trip-time.

This algorithm, called the Nagle algorithm, takes a few lines of code in all TCP implementations. These lines of code have a huge impact on the packets that are exchanged in TCP/IP networks. Researchers have analyzed the distribution of the packet sizes by capturing and analyzing all the packets passing through a given link. These studies have shown several important results :

- in TCP/IP networks, a large fraction of the packets are TCP segments that contain only an acknowledgment. These packets usually account for 40-50% of the packets passing through the studied link
- in TCP/IP networks, most of the bytes are exchanged in long packets, usually packets containing about 1440 bytes of payload which is the default MSS for hosts attached to an Ethernet network, the most popular type of LAN

Recent measurements indicate that these packet size distributions are still valid in today's Internet, although the packet distribution tends to become bi-modal with small packets corresponding to TCP pure acknowledgments and large 1440-bytes packets carrying most of the user data [SMASU2012].

3.12.3 TCP windows

From a performance point of view, one of the main limitations of the original TCP specification is the 16 bits *window* field in the TCP header. As this field indicates the current size of the receive window in bytes, it limits the TCP receive window at 65535 bytes. This limitation was not a severe problem when TCP was designed since at that time high-speed wide area networks offered a maximum bandwidth of 56 kbps. However, in today's network, this limitation is not acceptable anymore. The table below provides the rough[11] maximum throughput that can be achieved by a TCP connection with a 64 KBytes window in function of the connection's round-trip-time

RTT	Maximum Throughput
1 msec	524 Mbps
10 msec	52.4 Mbps
100 msec	5.24 Mbps
500 msec	1.05 Mbps

To solve this problem, a backward compatible extension that allows TCP to use larger receive windows was proposed in **RFC 1323**. Today, most TCP implementations support this option. The basic idea is that instead of storing *snd.wnd* and *rcv.wnd* as 16 bits integers in the TCB, they should be stored as 32 bits integers. As the TCP segment header only contains 16 bits to place the window field, it is impossible to copy the value of *snd.wnd* in each sent TCP segment. Instead the header contains *snd.wnd* >> S where S is the scaling factor ($0 \leq S \leq 14$) negotiated during connection establishment. The client adds its proposed scaling factor as a TCP option in the *SYN* segment. If the server supports

[11] A precise estimation of the maximum bandwidth that can be achieved by a TCP connection should take into account the overhead of the TCP and IP headers as well.

RFC 1323, it places in the *SYN+ACK* segment the scaling factor that it uses when advertising its own receive window. The local and remote scaling factors are included in the *TCB*. If the server does not support **RFC 1323**, it ignores the received option and no scaling is applied.

By using the window scaling extensions defined in **RFC 1323**, TCP implementations can use a receive buffer of up to 1 GByte. With such a receive buffer, the maximum throughput that can be achieved by a single TCP connection becomes :

RTT	Maximum Throughput
1 msec	8590 Gbps
10 msec	859 Gbps
100 msec	86 Gbps
500 msec	17 Gbps

These throughputs are acceptable in today's networks. However, there are already servers having 10 Gbps interfaces... Early TCP implementations had fixed receiving and sending buffers[12]. Today's high performance implementations are able to automatically adjust the size of the sending and receiving buffer to better support high bandwidth flows [SMM1998].

3.12.4 TCP's retransmission timeout

In a go-back-n transport protocol such as TCP, the retransmission timeout must be correctly set in order to achieve good performance. On one hand, if the retransmission timeout expires too early, then bandwidth is wasted by retransmitting segments that have already been correctly received. On the other hand, if the retransmission timeout expires too late, then bandwidth is wasted because the sender is idle waiting for the expiration of its retransmission timeout.

A good setting of the retransmission timeout clearly depends on an accurate estimation of the round-trip-time of each TCP connection. The round-trip-time differs between TCP connections, but may also change during the lifetime of a single connection. For example, the figure below shows the evolution of the round-trip-time between two hosts during a period of 45 seconds.

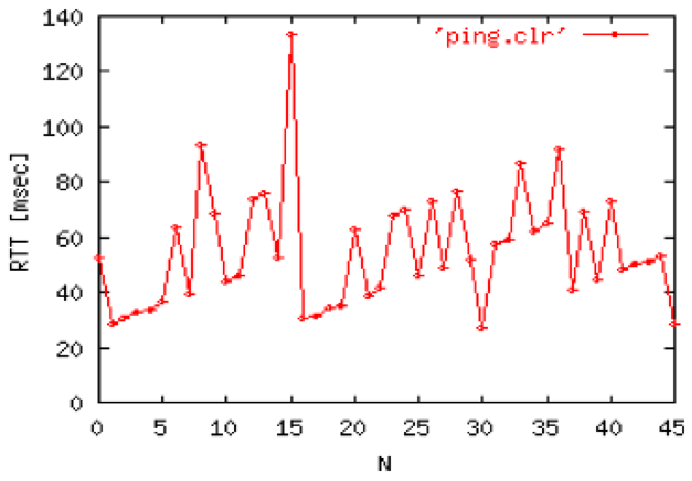


Fig. 26: Evolution of the round-trip-time between two hosts

The easiest solution to measure the round-trip-time on a TCP connection is to measure the delay between the transmission of a data segment and the reception of a corresponding acknowledgment[13]. As illustrated in the figure below, this measurement works well when there are no segment losses.

[12] See http://fasterdata.es.net/tuning.html for more information on how to tune a TCP implementation

[13] In theory, a TCP implementation could store the timestamp of each data segment transmitted and compute a new estimate for the round-trip-

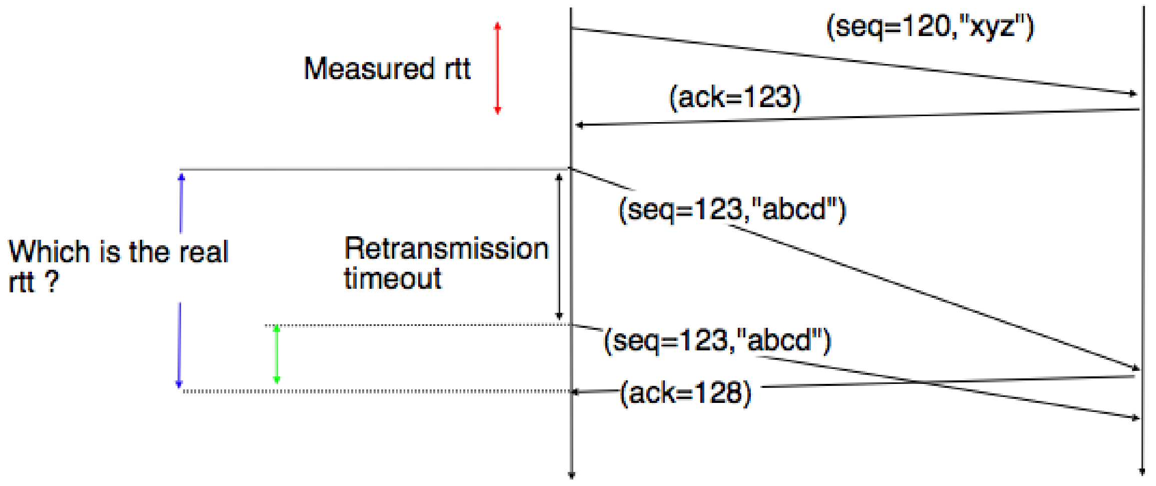


Fig. 27: How to measure the round-trip-time ?

However, when a data segment is lost, as illustrated in the bottom part of the figure, the measurement is ambiguous as the sender cannot determine whether the received acknowledgment was triggered by the first transmission of segment *123* or its retransmission. Using incorrect round-trip-time estimations could lead to incorrect values of the retransmission timeout. For this reason, Phil Karn and Craig Partridge proposed, in [KP91], to ignore the round-trip-time measurements performed during retransmissions.

To avoid this ambiguity in the estimation of the round-trip-time when segments are retransmitted, recent TCP implementations rely on the *timestamp option* defined in **RFC 1323**. This option allows a TCP sender to place two 32 bit timestamps in each TCP segment that it sends. The first timestamp, TS Value (*TSval*) is chosen by the sender of the segment. It could for example be the current value of its real-time clock[14]. The second value, TS Echo Reply (*TSecr*), is the last *TSval* that was received from the remote host and stored in the *TCB*. The figure below shows how the utilization of this timestamp option allows for the disambiguation of the round-trip-time measurement when there are retransmissions.

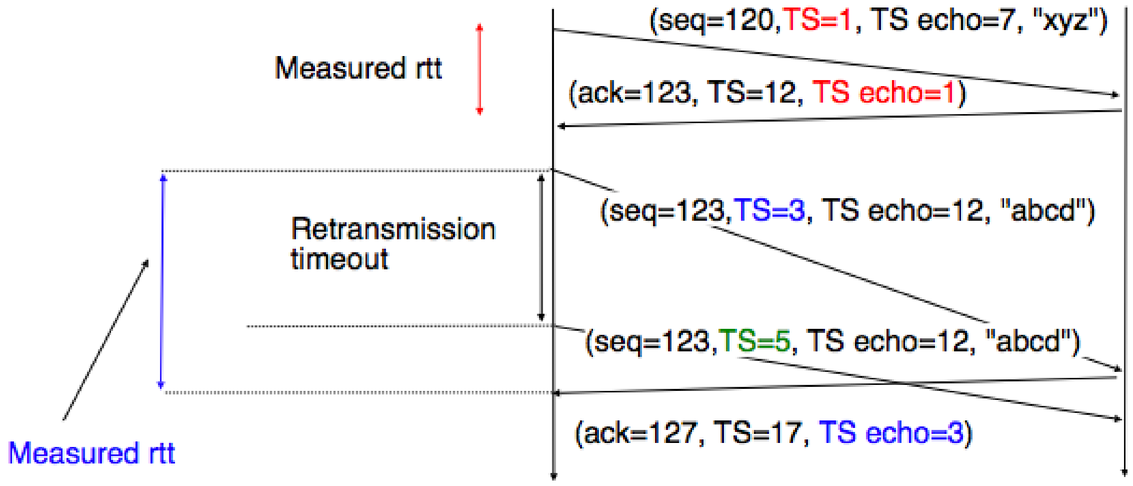

Measured rtt
(seq=120,TS=1, TS echo=7, "xyz")
(ack=123, TS=12, TS echo=1)
Retransmission timeout
(seq=123,TS=3, TS echo=12, "abcd")
(seq=123,TS=5, TS echo=12, "abcd")
(ack=127, TS=17, TS echo=3)
Measured rtt

Fig. 28: Disambiguating round-trip-time measurements with the **RFC 1323** timestamp option

time upon reception of the corresponding acknowledgment. However, using such frequent measurements introduces a lot of noise in practice and many implementations still measure the round-trip-time once per round-trip-time by recording the transmission time of one segment at a time **RFC 2988**

[14] Some security experts have raised concerns that using the real-time clock to set the *TSval* in the timestamp option can leak information such as the system's up-time. Solutions proposed to solve this problem may be found in [CNPI09]

Once the round-trip-time measurements have been collected for a given TCP connection, the TCP entity must compute the retransmission timeout. As the round-trip-time measurements may change during the lifetime of a connection, the retransmission timeout may also change. At the beginning of a connection[15], the TCP entity that sends a *SYN* segment does not know the round-trip-time to reach the remote host and the initial retransmission timeout is usually set to 3 seconds **RFC 2988**.

The original TCP specification proposed in **RFC 793** to include two additional variables in the TCB :

- *srtt* : the smoothed round-trip-time computed as $srtt = (\alpha \times srtt) + ((1 - \alpha) \times rtt)$ where rtt is the round-trip-time measured according to the above procedure and α a smoothing factor (e.g. 0.8 or 0.9)
- *rto* : the retransmission timeout is computed as $rto = \min(60, max(1, \beta \times srtt))$ where β is used to take into account the delay variance (value : 1.3 to 2.0). The *60* and *1* constants are used to ensure that the *rto* is not larger than one minute nor smaller than 1 second.

However, in practice, this computation for the retransmission timeout did not work well. The main problem was that the computed *rto* did not correctly take into account the variations in the measured round-trip-time. *Van Jacobson* proposed in his seminal paper [Jacobson1988] an improved algorithm to compute the *rto* and implemented it in the BSD Unix distribution. This algorithm is now part of the TCP standard **RFC 2988**.

Jacobson's algorithm uses two state variables, *srtt* the smoothed *rtt* and *rttvar* the estimation of the variance of the *rtt* and two parameters : α and β. When a TCP connection starts, the first *rto* is set to *3* seconds. When a first estimation of the *rtt* is available, the *srtt*, *rttvar* and *rto* are computed as follows :

```
srtt = rtt
rttvar = rtt/2
rto = srtt + 4*rttvar
```

Then, when other rtt measurements are collected, *srtt* and *rttvar* are updated as follows :

$$rttvar = (1 - \beta) \times rttvar + \beta \times |srtt - rtt|$$

$$srtt = (1 - \alpha) \times srtt + \alpha \times rtt$$

$$rto = srtt + 4 \times rttvar$$

The proposed values for the parameters are $\alpha = \frac{1}{8}$ and $\beta = \frac{1}{4}$. This allows a TCP implementation, implemented in the kernel, to perform the *rtt* computation by using shift operations instead of the more costly floating point operations [Jacobson1988]. The figure below illustrates the computation of the *rto* upon *rtt* changes.

3.12.5 Advanced retransmission strategies

The default go-back-n retransmission strategy was defined in **RFC 793**. When the retransmission timer expires, TCP retransmits the first unacknowledged segment (i.e. the one having sequence number *snd.una*). After each expiration of the retransmission timeout, **RFC 2988** recommends to double the value of the retransmission timeout. This is called an *exponential backoff*. This doubling of the retransmission timeout after a retransmission was included in TCP to deal with issues such as network/receiver overload and incorrect initial estimations of the retransmission timeout. If the same segment is retransmitted several times, the retransmission timeout is doubled after every retransmission until it reaches a configured maximum. **RFC 2988** suggests a maximum retransmission timeout of at least 60 seconds. Once the retransmission timeout reaches this configured maximum, the remote host is considered to be unreachable and the TCP connection is closed.

This retransmission strategy has been refined based on the experience of using TCP on the Internet. The first refinement was a clarification of the strategy used to send acknowledgments. As TCP uses piggybacking, the easiest and less costly method to send acknowledgments is to place them in the data segments sent in the other direction. However,

[15] As a TCP client often establishes several parallel or successive connections with the same server, **RFC 2140** has proposed to reuse for a new connection some information that was collected in the TCB of a previous connection, such as the measured rtt. However, this solution has not been widely implemented.

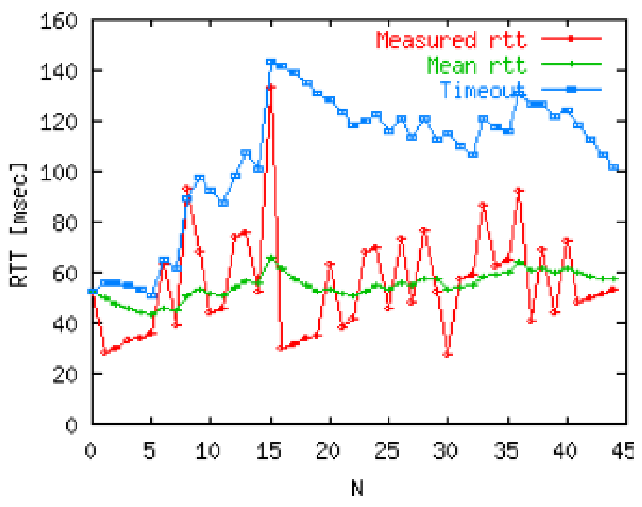


Fig. 29: Example computation of the *rto*

few application layer protocols exchange data in both directions at the same time and thus this method rarely works. For an application that is sending data segments in one direction only, the remote TCP entity returns empty TCP segments whose only useful information is their acknowledgment number. This may cause a large overhead in wide area network if a pure *ACK* segment is sent in response to each received data segment. Most TCP implementations use a *delayed acknowledgment* strategy. This strategy ensures that piggybacking is used whenever possible, otherwise pure *ACK* segments are sent for every second received data segments when there are no losses. When there are losses or reordering, *ACK* segments are more important for the sender and they are sent immediately **RFC 813 RFC 1122**. This strategy relies on a new timer with a short delay (e.g. 50 milliseconds) and one additional flag in the TCB. It can be implemented as follows.

```
reception of a data segment:
    if pkt.seq == rcv.nxt:  # segment received in sequence
        if delayed_ack:
            send pure ack segment
            delayed_ack = False
            ack_timer.cancel()
        else:
            delayed_ack = True
            ack_timer.start()
    else:  # out of sequence segment
        send pure ack segment
        if delayed_ack:
            delayed_ack = False
            ack_timer.cancel()

transmission of a data segment:  # piggyback ack
    if delayed_ack:
        delayed_ack = False
        ack_timer.cancel()

acktimer expiration:
    send pure ack segment
    delayed_ack = False
```

Due to this delayed acknowledgment strategy, during a bulk transfer, a TCP implementation usually acknowledges every second TCP segment received.

The default go-back-n retransmission strategy used by TCP has the advantage of being simple to implement, in particular on the receiver side, but when there are losses, a go-back-n strategy provides a lower performance than a selective repeat strategy. The TCP developers have designed several extensions to TCP to allow it to use a selective repeat strategy while maintaining backward compatibility with older TCP implementations. These TCP extensions assume that the receiver is able to buffer the segments that it receives out-of-sequence.

The first extension that was proposed is the fast retransmit heuristic. This extension can be implemented on TCP senders and thus does not require any change to the protocol. It only assumes that the TCP receiver is able to buffer out-of-sequence segments.

From a performance point of view, one issue with TCP's *retransmission timeout* is that when there are isolated segment losses, the TCP sender often remains idle waiting for the expiration of its retransmission timeouts. Such isolated losses are frequent in the global Internet [Paxson99]. A heuristic to deal with isolated losses without waiting for the expiration of the retransmission timeout has been included in many TCP implementations since the early 1990s. To understand this heuristic, let us consider the figure below that shows the segments exchanged over a TCP connection when an isolated segment is lost.

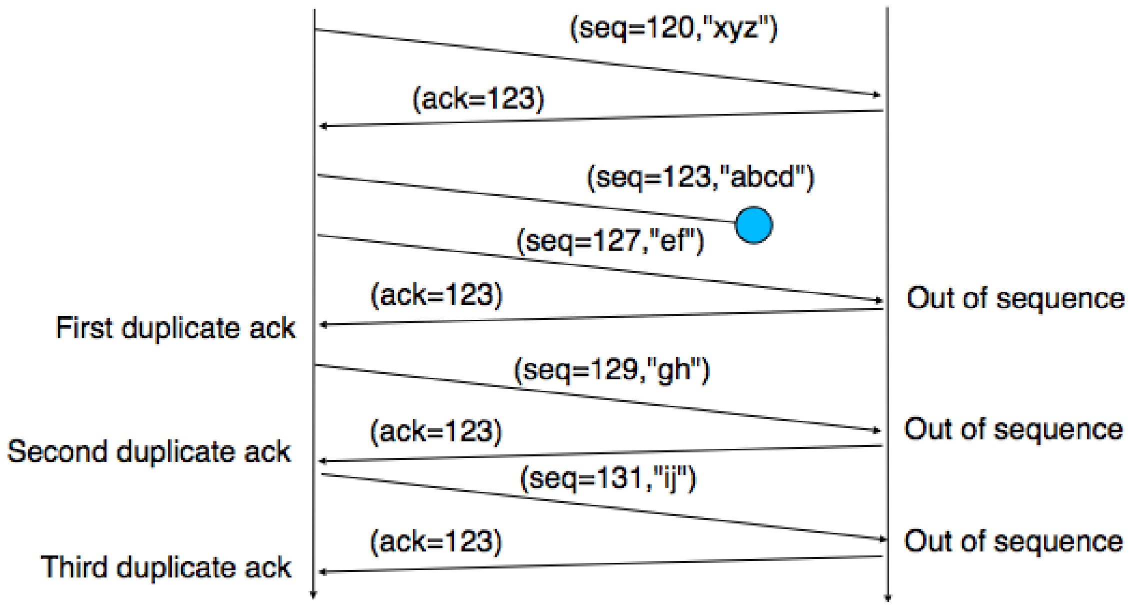


Fig. 30: Detecting isolated segment losses

As shown above, when an isolated segment is lost the sender receives several *duplicate acknowledgments* since the TCP receiver immediately sends a pure acknowledgment when it receives an out-of-sequence segment. A duplicate acknowledgment is an acknowledgment that contains the same *acknowledgment number* as a previous segment. A single duplicate acknowledgment does not necessarily imply that a segment was lost, as a simple reordering of the segments may cause duplicate acknowledgments as well. Measurements [Paxson99] have shown that segment reordering is frequent in the Internet. Based on these observations, the *fast retransmit* heuristic has been included in most TCP implementations. It can be implemented as follows.

```
ack arrival:
    if tcp.ack == snd.una:  # duplicate acknowledgment
        dupacks += 1
        if dupacks == 3:
            retransmit segment(snd.una)
    else:
        dupacks = 0
        # process acknowledgment
```

This heuristic requires an additional variable in the TCB (*dupacks*). Most implementations set the default number of duplicate acknowledgments that trigger a retransmission to 3. It is now part of the standard TCP specification **RFC 2581**. The *fast retransmit* heuristic improves the TCP performance provided that isolated segments are lost and the current window is large enough to allow the sender to send three duplicate acknowledgments.

The figure below illustrates the operation of the *fast retransmit* heuristic.

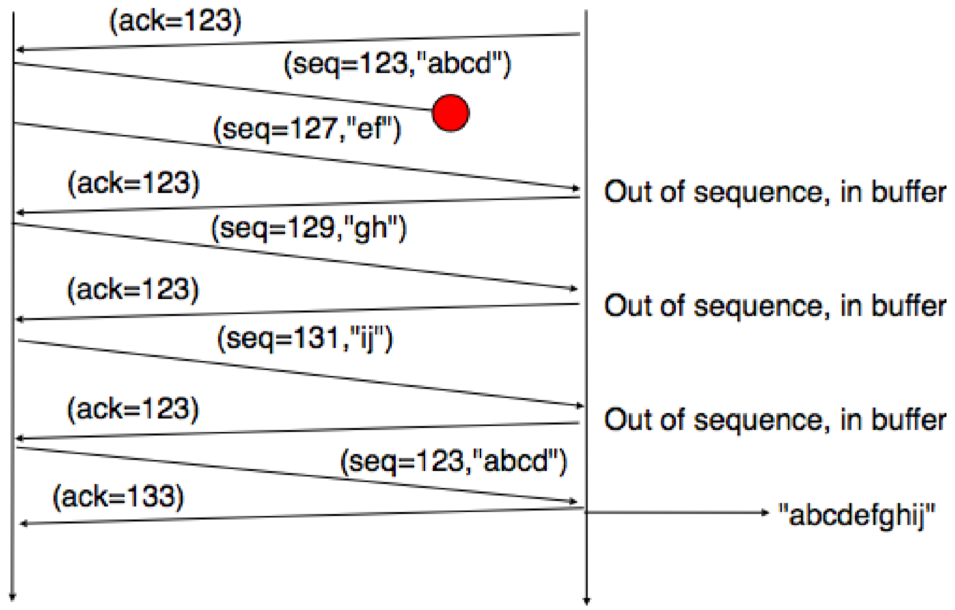


Fig. 31: TCP fast retransmit heuristics

When losses are not isolated or when the windows are small, the performance of the *fast retransmit* heuristic decreases. In such environments, it is necessary to allow a TCP sender to use a selective repeat strategy instead of the default go-back-n strategy. Implementing selective-repeat requires a change to the TCP protocol as the receiver needs to be able to inform the sender of the out-of-order segments that it has already received. This can be done by using the Selective Acknowledgments (SACK) option defined in **RFC 2018**. This TCP option is negotiated during the establishment of a TCP connection. If both TCP hosts support the option, SACK blocks can be attached by the receiver to the segments that it sends. SACK blocks allow a TCP receiver to indicate the blocks of data that it has received correctly but out of sequence. The figure below illustrates the utilization of the SACK blocks.

A SACK option contains one or more blocks. A block corresponds to all the sequence numbers between the *left edge* and the *right edge* of the block. The two edges of the block are encoded as 32 bit numbers (the same size as the TCP sequence number) in an SACK option. As the SACK option contains one byte to encode its type and one byte for its length, a SACK option containing b blocks is encoded as a sequence of $2 + 8 \times b$ bytes. In practice, the size of the SACK option can be problematic as the optional TCP header extension cannot be longer than 40 bytes. As the SACK option is usually combined with the **RFC 1323** timestamp extension, this implies that a TCP segment cannot usually contain more than three SACK blocks. This limitation implies that a TCP receiver cannot always place in the SACK option that it sends, information about all the received blocks.

To deal with the limited size of the SACK option, a TCP receiver currently having more than 3 blocks inside its receiving buffer must select the blocks to place in the SACK option. A good heuristic is to put in the SACK option the blocks that have most recently changed, as the sender is likely to be already aware of the older blocks.

When a sender receives a SACK option indicating a new block and thus a new possible segment loss, it usually does not retransmit the missing segments immediately. To deal with reordering, a TCP sender can use a heuristic similar to *fast retransmit* by retransmitting a gap only once it has received three SACK options indicating this gap. It should be noted that the SACK option does not supersede the *acknowledgment number* of the TCP header. A TCP sender can only remove data from its sending buffer once they have been acknowledged by TCP's cumulative acknowledgments. This design was chosen for two reasons. First, it allows the receiver to discard parts of its receiving buffer when it is

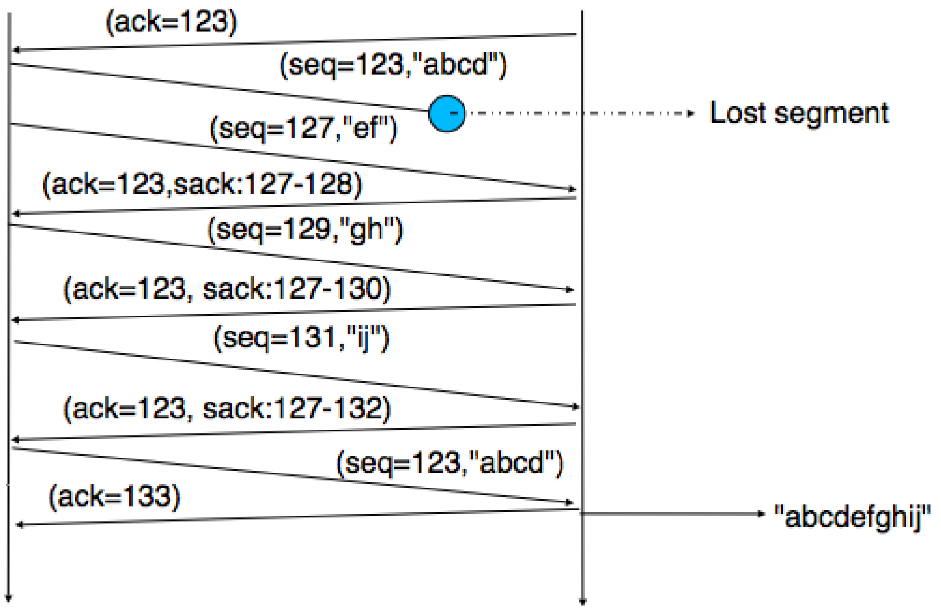


Fig. 32: TCP selective acknowledgments

running out of memory without loosing data. Second, as the SACK option is not transmitted reliably, the cumulative acknowledgments are still required to deal with losses of *ACK* segments carrying only SACK information. Thus, the SACK option only serves as a hint to allow the sender to optimize its retransmissions.

As explained earlier, the TCP Timestamp option **RFC 1323** prevents ambiguities while collecting round-trip-time measurements. It plays another very important role in today's high-bandwidth networks. Since TCP uses 32 bits long sequence numbers, the sequence numbers wrap after the transmission of 4 GBytes of data. With 10 Gbps and soon 100 Gbps interfaces, TCP only needs to transmit during a few seconds before reusing the same sequence number. Given that the Maximum Segment Lifetime is still 2 minutes, several packets, belonging to the same TCP connection could use the same sequence number. If one of these packets is severely delayed through the network, it could reappear at the same time as a packet with the same TCP sequence number. To prevent this problem, most modern TCP implementations associate a TCP timestamp option to each segment on transmission. When a TCP stack receives a TCP segment, it checks that its TCP timestamp is valid and if not the segment is discarded **RFC 7323**.

3.12.6 TCP connection release

TCP, like most connection-oriented transport protocols, supports two types of connection releases :

- graceful connection release, where each TCP user can release its own direction of data transfer after having transmitted all data
- abrupt connection release, where either one user closes both directions of data transfer or one TCP entity is forced to close the connection (e.g., because the remote host does not reply anymore or due to lack of resources)

The abrupt connection release mechanism is very simple and relies on a single segment having the *RST* bit set. A TCP segment containing the *RST* bit can be sent for the following reasons :

- a non-*SYN* segment was received for a non-existing TCP connection **RFC 793**
- by extension, some implementations respond with an *RST* segment to a segment that is received on an existing connection but with an invalid header **RFC 3360**. This causes the corresponding connection to be closed and has caused security attacks **RFC 4953**

- by extension, some implementations send an *RST* segment when they need to close an existing TCP connection (e.g., because there are not enough resources to support this connection or because the remote host is considered to be unreachable). Measurements have shown that this usage of TCP *RST* is widespread [AW05]

When an *RST* segment is sent by a TCP entity, it should contain the current value of the *sequence number* for the connection (or 0 if it does not belong to any existing connection) and the *acknowledgment number* should be set to the next expected in-sequence *sequence number* on this connection.

Note: TCP *RST* wars

The designers of TCP implementations should ensure that two TCP entities never enter a TCP *RST* war where host *A* is sending a *RST* segment in response to a previous *RST* segment that was sent by host *B* in response to a TCP *RST* segment sent by host *A* ... To avoid such an infinite exchange of *RST* segments that do not carry data, a TCP entity is *never* allowed to send a *RST* segment in response to another *RST* segment.

The normal way of terminating a TCP connection is by using the graceful TCP connection release. This mechanism uses the *FIN* flag of the TCP header and allows each host to release its own direction of data transfer. As for the *SYN* flag, the utilization of the *FIN* flag in the TCP header consumes one sequence number. The figure *FSM for TCP connection release* shows the part of the TCP FSM used when a TCP connection is released.

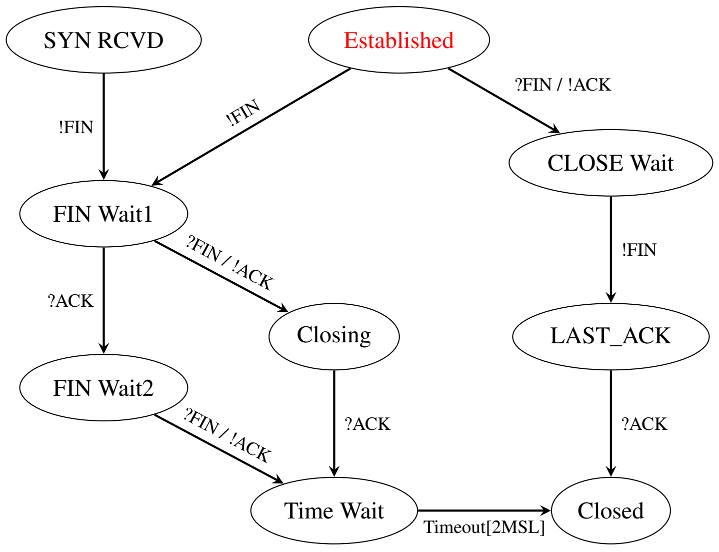


Fig. 33: FSM for TCP connection release

Starting from the *Established* state, there are two main paths through this FSM.

The first path is when the host receives a segment with sequence number *x* and the *FIN* flag set. The utilization of the *FIN* flag indicates that the byte before *sequence number x* was the last byte of the byte stream sent by the remote host. Once all of the data has been delivered to the user, the TCP entity sends an *ACK* segment whose *ack* field is set to $(x + 1) \pmod{2^{32}}$ to acknowledge the *FIN* segment. The *FIN* segment is subject to the same retransmission mechanisms as a normal TCP segment. In particular, its transmission is protected by the retransmission timer. At this point, the TCP connection enters the *CLOSE_WAIT* state. In this state, the host can still send data to the remote host. Once all its data have been sent, it sends a *FIN* segment and enter the *LAST_ACK* state. In this state, the TCP entity waits for the acknowledgment of its *FIN* segment. It may still retransmit unacknowledged data segments, e.g., if the retransmission timer expires. Upon reception of the acknowledgment for the *FIN* segment, the TCP connection is completely closed and its *TCB* can be discarded.

The second path is when the host has transmitted all data. Assume that the last transmitted sequence number is z. Then, the host sends a *FIN* segment with sequence number $(z+1) \pmod{2^{32}}$ and enters the *FIN_WAIT1* state. In this state, it can retransmit unacknowledged segments but cannot send new data segments. It waits for an acknowledgment of its *FIN* segment (i.e. sequence number $(z+1) \pmod{2^{32}}$), but may receive a *FIN* segment sent by the remote host. In the first case, the TCP connection enters the *FIN_WAIT2* state. In this state, new data segments from the remote host are still accepted until the reception of the *FIN* segment. The acknowledgment for this *FIN* segment is sent once all data received before the *FIN* segment have been delivered to the user and the connection enters the *TIME_WAIT* state. In the second case, a *FIN* segment is received and the connection enters the *Closing* state once all data received from the remote host have been delivered to the user. In this state, no new data segments can be sent and the host waits for an acknowledgment of its *FIN* segment before entering the *TIME_WAIT* state.

The *TIME_WAIT* state is different from the other states of the TCP FSM. A TCP entity enters this state after having sent the last *ACK* segment on a TCP connection. This segment indicates to the remote host that all the data that it has sent have been correctly received and that it can safely release the TCP connection and discard the corresponding *TCB*. After having sent the last *ACK* segment, a TCP connection enters the *TIME_WAIT* and remains in this state for $2 * MSL$ seconds. During this period, the TCB of the connection is maintained. This ensures that the TCP entity that sent the last *ACK* maintains enough state to be able to retransmit this segment if this *ACK* segment is lost and the remote host retransmits its last *FIN* segment or another one. The delay of $2 * MSL$ seconds ensures that any duplicate segments on the connection would be handled correctly without causing the transmission of an *RST* segment. Without the *TIME_WAIT* state and the $2 * MSL$ seconds delay, the connection release would not be graceful when the last *ACK* segment is lost.

Note: TIME_WAIT on busy TCP servers

The $2 * MSL$ seconds delay in the *TIME_WAIT* state is an important operational problem on servers having thousands of simultaneously opened TCP connections [FTY99]. Consider for example a busy web server that processes 10.000 TCP connections every second. If each of these connections remains in the *TIME_WAIT* state for 4 minutes, this implies that the server would have to maintain more than 2 million TCBs at any time. For this reason, some TCP implementations prefer to perform an abrupt connection release by sending a *RST* segment to close the connection [AW05] and immediately discard the corresponding *TCB*. However, if the *RST* segment is lost, the remote host continues to maintain a *TCB* for a connection that no longer exists. This optimization reduces the number of TCBs maintained by the host sending the *RST* segment but at the potential cost of increased processing on the remote host when the *RST* segment is lost.

3.13 Congestion control

In an internetwork, i.e. a networking composed of different types of networks (such as the Internet), congestion control could be implemented either in the network layer or the transport layer. The congestion problem was clearly identified in the later 1980s and the researchers who developed techniques to solve the problem opted for a solution in the transport layer. Adding congestion control to the transport layer makes sense since this layer provides a reliable data transfer and avoiding congestion is a factor in this reliable delivery. The transport layer already deals with heterogeneous networks thanks to its *self-clocking* property that we have already described. In this section, we explain how congestion control has been added to TCP and how this mechanism could be improved in the future.

The TCP congestion control scheme was initially proposed by Van Jacobson in [Jacobson1988]. The current specification may be found in **RFC 5681**. TCP relies on *Additive Increase and Multiplicative Decrease (AIMD)*. To implement *AIMD*, a TCP host must be able to control its transmission rate. A first approach would be to use timers and adjust their expiration times in function of the rate imposed by *AIMD*. Unfortunately, maintaining such timers for a large number of TCP connections can be difficult. Instead, Van Jacobson noted that the rate of TCP congestion can be artificially controlled by constraining its sending window. A TCP connection cannot send data faster than $\frac{window}{rtt}$ where $window$ is the maximum between the host's sending window and the window advertised by the receiver.

TCP's congestion control scheme is based on a *congestion window*. The current value of the congestion window (*cwnd*) is stored in the TCB of each TCP connection and the window that can be used by the sender is constrained by $\min(cwnd, rwin, swin)$ where $swin$ is the current sending window and $rwin$ the last received receive window. The *Additive Increase* part of the TCP congestion control increments the congestion window by *MSS* bytes every round-trip-time. In the TCP literature, this phase is often called the *congestion avoidance* phase. The *Multiplicative Decrease* part of the TCP congestion control divides the current value of the congestion window once congestion has been detected.

When a TCP connection begins, the sending host does not know whether the part of the network that it uses to reach the destination is congested or not. To avoid causing too much congestion, it must start with a small congestion window. [Jacobson1988] recommends an initial window of MSS bytes. As the additive increase part of the TCP congestion control scheme increments the congestion window by MSS bytes every round-trip-time, the TCP connection may have to wait many round-trip-times before being able to efficiently use the available bandwidth. This is especially important in environments where the $bandwidth \times rtt$ product is high. To avoid waiting too many round-trip-times before reaching a congestion window that is large enough to efficiently utilize the network, the TCP congestion control scheme includes the *slow-start* algorithm. The objective of the TCP *slow-start* phase is to quickly reach an acceptable value for the *cwnd*. During *slow-start*, the congestion window is doubled every round-trip-time. The *slow-start* algorithm uses an additional variable in the TCB : *ssthresh* (*slow-start threshold*). The *ssthresh* is an estimation of the last value of the *cwnd* that did not cause congestion. It is initialized at the sending window and is updated after each congestion event.

A key question that must be answered by any congestion control scheme is how congestion is detected. The first implementations of the TCP congestion control scheme opted for a simple and pragmatic approach : packet losses indicate congestion. If the network is congested, router buffers are full and packets are discarded. In wired networks, packet losses are mainly caused by congestion. In wireless networks, packets can be lost due to transmission errors and for other reasons that are independent of congestion. TCP already detects segment losses to ensure a reliable delivery. The TCP congestion control scheme distinguishes between two types of congestion :

- *mild congestion*. TCP considers that the network is lightly congested if it receives three duplicate acknowledgments and performs a fast retransmit. If the fast retransmit is successful, this implies that only one segment has been lost. In this case, TCP performs multiplicative decrease and the congestion window is divided by *2*. The slow-start threshold is set to the new value of the congestion window.
- *severe congestion*. TCP considers that the network is severely congested when its retransmission timer expires. In this case, TCP retransmits the first segment, sets the slow-start threshold to 50% of the congestion window. The congestion window is reset to its initial value and TCP performs a slow-start.

The figure below illustrates the evolution of the congestion window when there is severe congestion. At the beginning of the connection, the sender performs *slow-start* until the first segments are lost and the retransmission timer expires. At this time, the *ssthresh* is set to half of the current congestion window and the congestion window is reset at one segment. The lost segments are retransmitted as the sender again performs slow-start until the congestion window reaches the *sshtresh*. It then switches to congestion avoidance and the congestion window increases linearly until segments are lost and the retransmission timer expires.

The figure below illustrates the evolution of the congestion window when the network is lightly congested and all lost segments can be retransmitted using fast retransmit. The sender begins with a slow-start. A segment is lost but successfully retransmitted by a fast retransmit. The congestion window is divided by 2 and the sender immediately enters congestion avoidance as this was a mild congestion.

Most TCP implementations update the congestion window when they receive an acknowledgment. If we assume that the receiver acknowledges each received segment and the sender only sends MSS sized segments, the TCP congestion control scheme can be implemented using the simplified pseudo-code[1] below. This pseudocode includes the optimization proposed in **RFC 3042** that allows a sender to send new unsent data upon reception of the first or second duplicate acknowledgment. The reception of each of these acknowledgment indicates that one segment has left the network and thus additional data can be sent without causing more congestion. Note that the congestion window is *not* increased upon reception of these first duplicate acknowledgments.

[1] In this pseudo-code, we assume that TCP uses unlimited sequence and acknowledgment numbers. Furthermore, we do not detail how the *cwnd* is adjusted after the retransmission of the lost segment by fast retransmit. Additional details may be found in **RFC 5681**.

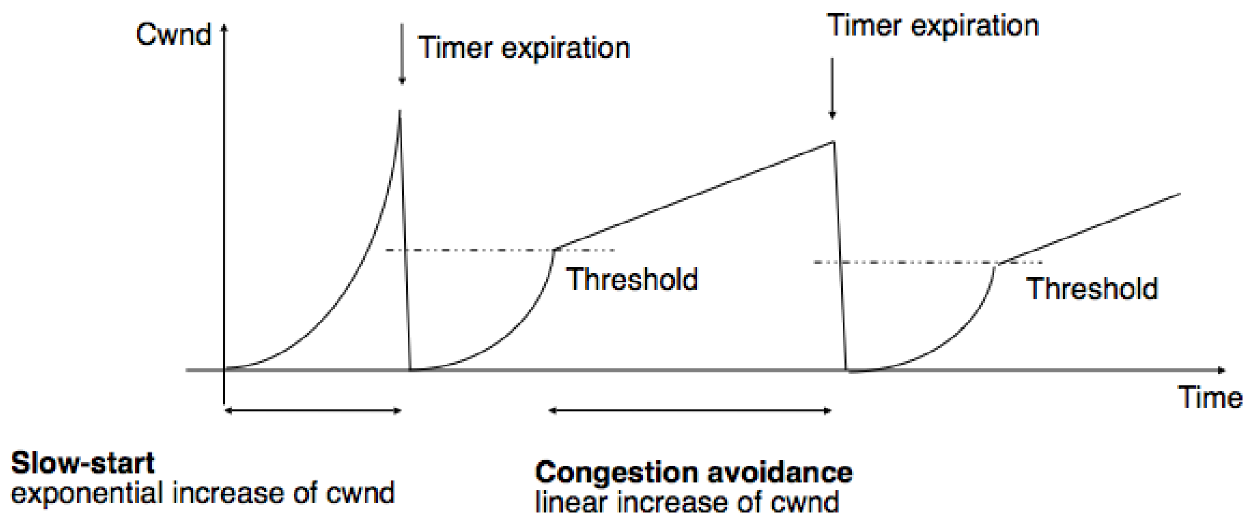


Fig. 34: Evaluation of the TCP congestion window with severe congestion

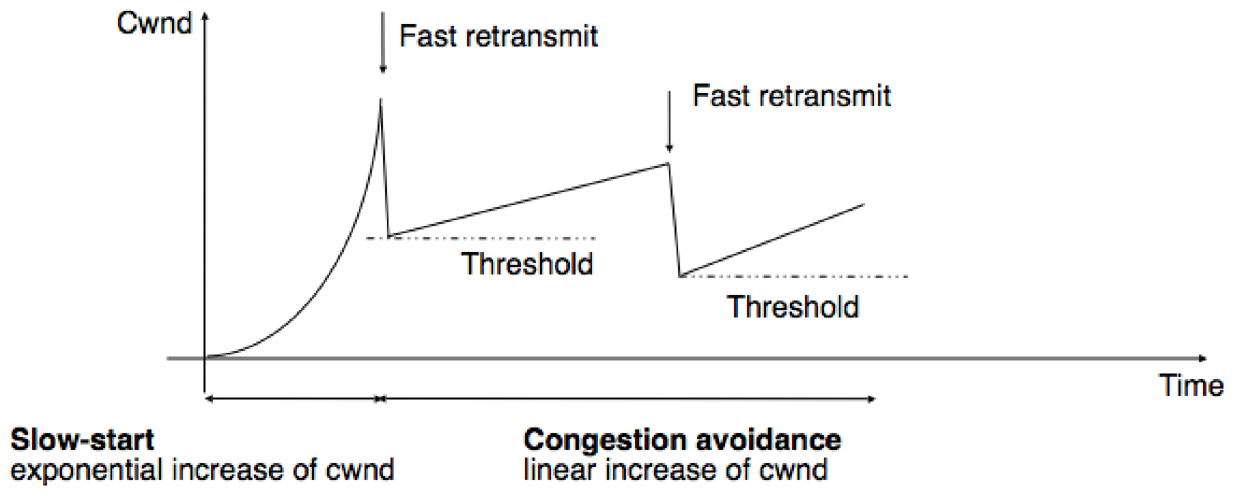


Fig. 35: Evaluation of the TCP congestion window when the network is lightly congested

```
# Initialization
cwnd = MSS  # congestion window in bytes
ssthresh= swin # in bytes

# Ack arrival
if tcp.ack > snd.una:  # new ack, no congestion
    if dupacks == 0:  # not currently recovering from loss
        if cwnd < ssthresh:
            # slow-start : quickly increase cwnd
            # double cwnd every rtt
            cwnd = cwnd + MSS
        else:
            # congestion avoidance : slowly increase cwnd
            # increase cwnd by one mss every rtt
            cwnd = cwnd + MSS * (MSS / cwnd)
    else:  # recovering from loss
        cwnd = ssthresh  # deflate cwnd RFC5681
        dupacks = 0
else:  # duplicate or old ack
    if tcp.ack == snd.una:  # duplicate acknowledgment
        dupacks += 1
        if dupacks == 1 or dupacks == 2:
            send_next_unacked_segment  # RFC3042
        if dupacks == 3:
            retransmitsegment(snd.una)
            ssthresh = max(cwnd/2, 2*MSS)
            cwnd = ssthresh
        if dupacks > 3:  # RFC5681
            cwnd = cwnd + MSS  # inflate cwnd
    else:
        # ack for old segment, ignored
        pass

Expiration of the retransmission timer:
    send(snd.una)  # retransmit first lost segment
    sshtresh = max(cwnd/2, 2*MSS)
    cwnd = MSS
```

Furthermore when a TCP connection has been idle for more than its current retransmission timer, it should reset its congestion window to the congestion window size that it uses when the connection begins, as it no longer knows the current congestion state of the network.

Note: Initial congestion window

The original TCP congestion control mechanism proposed in [Jacobson1988] recommended that each TCP connection should begin by setting $cwnd = MSS$. However, in today's higher bandwidth networks, using such a small initial congestion window severely affects the performance for short TCP connections, such as those used by web servers. In 2002, **RFC 3390** allowed an initial congestion window of about 4 KBytes, which corresponds to 3 segments in many environments. Recently, researchers from Google proposed to further increase the initial window up to 15 KBytes [DRC+2010]. The measurements that they collected show that this increase would not significantly increase congestion but would significantly reduce the latency of short HTTP responses. Unsurprisingly, the chosen initial window corresponds to the average size of an HTTP response from a search engine. This proposed modification has been adopted in **RFC 6928** and TCP implementations support it.

3.13.1 Controlling congestion without losing data

In today's Internet, congestion is controlled by regularly sending packets at a higher rate than the network capacity. These packets fill the buffers of the routers and are eventually discarded. But shortly after, TCP senders retransmit packets containing exactly the same data. This is potentially a waste of resources since these successive retransmissions consume resources upstream of the router that discards the packets. Packet losses are not the only signal to detect congestion inside the network. An alternative is to allow routers to explicitly indicate their current level of congestion when forwarding packets. This approach was proposed in the late 1980s [RJ1995] and used in some networks. Unfortunately, it took almost a decade before the Internet community agreed to consider this approach. In the mean time, a large number of TCP implementations and routers were deployed on the Internet.

As explained earlier, Explicit Congestion Notification **RFC 3168** improves the detection of congestion by allowing routers to explicitly mark packets when they are lightly congested. In theory, a single bit in the packet header [RJ1995] is sufficient to support this congestion control scheme. When a host receives a marked packet, it returns the congestion information to the source that adapts its transmission rate accordingly. Although the idea is relatively simple, deploying it on the entire Internet has proven to be challenging [KNT2013]. It is interesting to analyze the different factors that have hindered the deployment of this technique.

The first difficulty in adding Explicit Congestion Notification (ECN) in TCP/IP network was to modify the format of the network packet and transport segment headers to carry the required information. In the network layer, one bit was required to allow the routers to mark the packets they forward during congestion periods. In the IP network layer, this bit is called the *Congestion Experienced* (*CE*) bit and is part of the packet header. However, using a single bit to mark packets is not sufficient. Consider a simple scenario with two sources, one congested router and one destination. Assume that the first sender and the destination support ECN, but not the second sender. If the router is congested it will mark packets from both senders. The first sender will react to the packet markings by reducing its transmission rate. However since the second sender does not support ECN, it will not react to the markings. Furthermore, this sender could continue to increase its transmission rate, which would lead to more packets being marked and the first source would decrease again its transmission rate, … In the end, the sources that implement ECN are penalized compared to the sources that do not implement it. This unfairness issue is a major hurdle to widely deploy ECN on the public Internet[2]. The solution proposed in **RFC 3168** to deal with this problem is to use a second bit in the network packet header. This bit, called the *ECN-capable transport* (ECT) bit, indicates whether the packet contains a segment produced by a transport protocol that supports ECN or not. Transport protocols that support ECN set the ECT bit in all packets. When a router is congested, it first verifies whether the ECT bit is set. In this case, the CE bit of the packet is set to indicate congestion. Otherwise, the packet is discarded. This eases the deployment of ECN[3].

The second difficulty is how to allow the receiver to inform the sender of the reception of network packets marked with the *CE* bit. In reliable transport protocols like TCP and SCTP, the acknowledgments can be used to provide this feedback. For TCP, two options were possible : change some bits in the TCP segment header or define a new TCP option to carry this information. The designers of ECN opted for reusing spare bits in the TCP header. More precisely, two TCP flags have been added in the TCP header to support ECN. The *ECN-Echo* (ECE) is set in the acknowledgments when the *CE* was set in packets received on the forward path.

```
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               |               | C | E | U | A | P | R | S | F |
| Header Length |    Reserved   | W | C | R | C | S | S | Y | I |
|               |               | R | E | G | K | H | T | N | N |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Fig. 36: The TCP flags

The third difficulty is to allow an ECN-capable sender to detect whether the remote host also supports ECN. This is a classical negotiation of extensions to a transport protocol. In TCP, this could have been solved by defining a new TCP

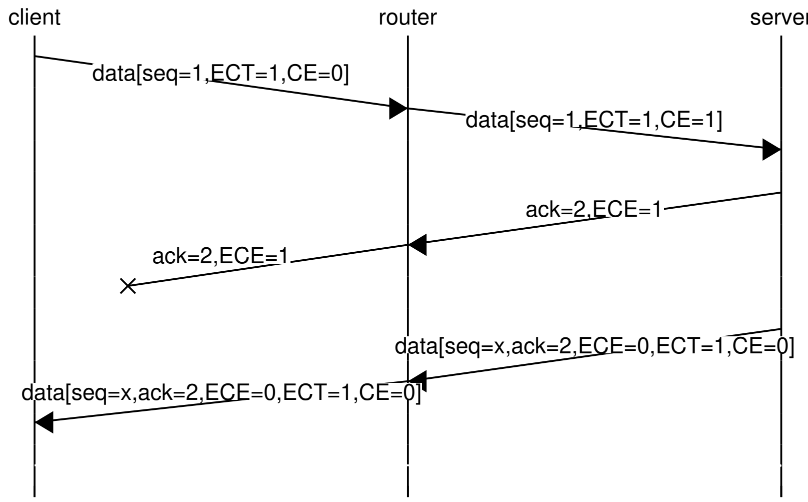
[2] In enterprise networks or datacenters, the situation is different since a single company typically controls all the sources and all the routers. In such networks it is possible to ensure that all hosts and routers have been upgraded before turning on ECN on the routers.

[3] With the ECT bit, the deployment issue with ECN is solved provided that all sources cooperate. If some sources do not support ECN but still set the ECT bit in the packets that they sent, they will have an unfair advantage over the sources that correctly react to packet markings. Several solutions have been proposed to deal with this problem **RFC 3540**, but they are outside the scope of this book.

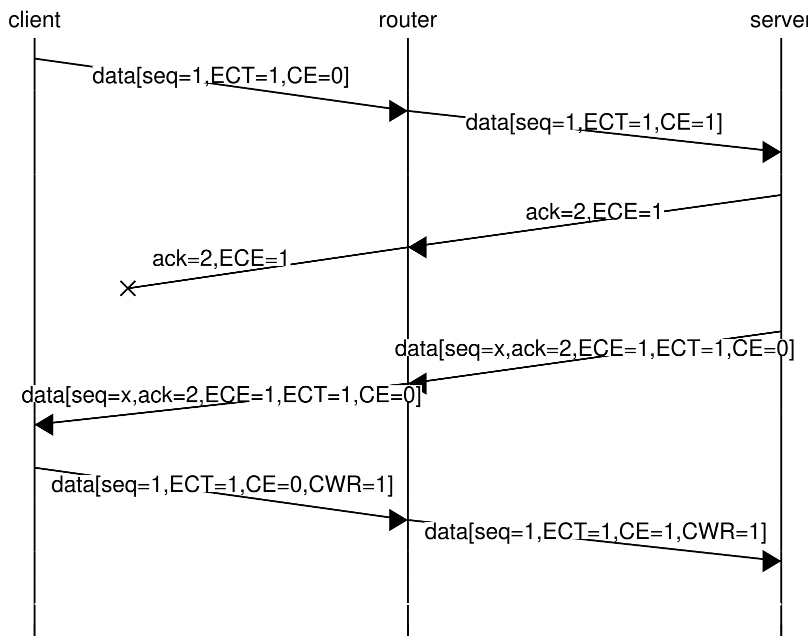
option used during the three-way handshake. To avoid wasting space in the TCP options, the designers of ECN opted in **RFC 3168** for using the *ECN-Echo* and *CWR* bits in the TCP header to perform this negotiation. In the end, the result is the same with fewer bits exchanged.

Thanks to the *ECT*, *CE* and *ECE*, routers can mark packets during congestion and receivers can return the congestion information back to the TCP senders. However, these three bits are not sufficient to allow a server to reliably send the *ECE* bit to a TCP sender. TCP acknowledgments are not sent reliably. A TCP acknowledgment always contains the next expected sequence number. Since TCP acknowledgments are cumulative, the loss of one acknowledgment is recovered by the correct reception of a subsequent acknowledgment.

If TCP acknowledgments are overloaded to carry the *ECE* bit, the situation is different. Consider the example shown in the figure below. A client sends packets to a server through a router. In the example below, the first packet is marked. The server returns an acknowledgment with the *ECE* bit set. Unfortunately, this acknowledgment is lost and never reaches the client. Shortly after, the server sends a data segment that also carries a cumulative acknowledgment. This acknowledgment confirms the reception of the data to the client, but it did not receive the congestion information through the *ECE* bit.



To solve this problem, **RFC 3168** uses an additional bit in the TCP header : the *Congestion Window Reduced* (CWR) bit.


client
router
server
data[seq=1,ECT=1,CE=0]
data[seq=1,ECT=1,CE=1]
ack=2,ECE=1
ack=2,ECE=1
data[seq=x,ack=2,ECE=1,ECT=1,CE=0]
data[seq=x,ack=2,ECE=1,ECT=1,CE=0]
data[seq=1,ECT=1,CE=0,CWR=1]
data[seq=1,ECT=1,CE=1,CWR=1]

The *CWR* bit of the TCP header provides some form of acknowledgment for the *ECE* bit. When a TCP receiver detects a packet marked with the *CE* bit, it sets the *ECE* bit in all segments that it returns to the sender. Upon reception of an acknowledgment with the *ECE* bit set, the sender reduces its congestion window to reflect a mild congestion and sets the *CWR* bit. This bit remains set as long as the segments received contained the *ECE* bit set. A sender should only react once per round-trip-time to marked packets.

The last point that needs to be discussed about Explicit Congestion Notification is the algorithm that is used by routers to detect congestion. On a router, congestion manifests itself by the number of packets that are stored inside the router buffers. As explained earlier, we need to distinguish between two types of routers :

- routers that have a single FIFO queue
- routers that have several queues served by a round-robin scheduler

Routers that use a single queue measure their buffer occupancy as the number of bytes of packets stored in the queue[4]. A first method to detect congestion is to measure the instantaneous buffer occupancy and consider the router to be congested as soon as this occupancy is above a threshold. Typical values of the threshold could be 40% of the total buffer. Measuring the instantaneous buffer occupancy is simple since it only requires one counter. However, this value is fragile from a control viewpoint since it changes frequently. A better solution is to measure the *average* buffer occupancy and consider the router to be congested when this average occupancy is too high. Random Early Detection (RED) [FJ1993] is an algorithm that was designed to support Explicit Congestion Notification. In addition to measuring the average buffer occupancy, it also uses probabilistic marking. When the router is congested, the arriving packets are marked with a probability that increases with the average buffer occupancy. The main advantage of using probabilistic marking instead of marking all arriving packets is that flows will be marked in proportion of the number of packets that they transmit. If the router marks 10% of the arriving packets when congested, then a large flow that sends hundred packets per second will be marked 10 times while a flow that only sends one packet per second will not be marked. This probabilistic marking allows marking packets in proportion of their usage of the network resources.

If the router uses several queues served by a scheduler, the situation is different. If a large and a small flow are competing for bandwidth, the scheduler will already favor the small flow that is not using its fair share of the bandwidth. The queue for the small flow will be almost empty while the queue for the large flow will build up. On routers using such schedulers, a good way of marking the packets is to set a threshold on the occupancy of each queue and mark the packets that arrive in a particular queue as soon as its occupancy is above the configured threshold.

[4] The buffers of a router can be implemented as variable or fixed-length slots. If the router uses variable length slots to store the queued packets, then the occupancy is usually measured in bytes. Some routers have use fixed-length slots with each slot large enough to store a maximum-length packet. In this case, the buffer occupancy is measured in packets.

3.13.2 Modeling TCP congestion control

Thanks to its congestion control scheme, TCP adapts its transmission rate to the losses that occur in the network. Intuitively, the TCP transmission rate decreases when the percentage of losses increases. Researchers have proposed detailed models that allow the prediction of the throughput of a TCP connection when losses occur [MSMO1997] . To have some intuition about the factors that affect the performance of TCP, let us consider a very simple model. Its assumptions are not completely realistic, but it gives us good intuition without requiring complex mathematics.

This model considers a hypothetical TCP connection that suffers from equally spaced segment losses. If p is the segment loss ratio, then the TCP connection successfully transfers $\frac{1}{p} - 1$ segments and the next segment is lost. If we ignore the slow-start at the beginning of the connection, TCP in this environment is always in congestion avoidance as there are only isolated losses that can be recovered by using fast retransmit. The evolution of the congestion window is thus as shown in the figure below. Note that the *x-axis* of this figure represents time measured in units of one round-trip-time, which is supposed to be constant in the model, and the *y-axis* represents the size of the congestion window measured in MSS-sized segments.

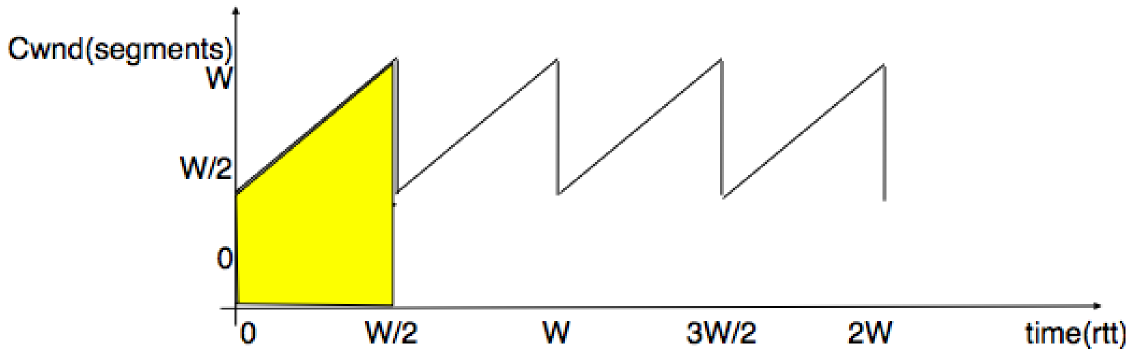


Fig. 37: Evolution of the congestion window with regular losses

As the losses are equally spaced, the congestion window always starts at some value ($\frac{W}{2}$), and is incremented by one MSS every round-trip-time until it reaches twice this value (W). At this point, a segment is retransmitted and the cycle starts again. If the congestion window is measured in MSS-sized segments, a cycle lasts $\frac{W}{2}$ round-trip-times. The bandwidth of the TCP connection is the number of bytes that have been transmitted during a given period of time. During a cycle, the number of segments that are sent on the TCP connection is equal to the area of the yellow trapeze in the figure. Its area is thus :

$$area = (\frac{W}{2})^2 + \frac{1}{2} \times (\frac{W}{2})^2 = \frac{3 \times W^2}{8}$$

However, given the regular losses that we consider, the number of segments that are sent between two losses (i.e. during a cycle) is by definition equal to $\frac{1}{p}$. Thus, $W = \sqrt{\frac{8}{3 \times p}} = \frac{k}{\sqrt{p}}$. The throughput (in bytes per second) of the TCP connection is equal to the number of segments transmitted divided by the duration of the cycle :

$$Throughput = \frac{area \times MSS}{time} = \frac{\frac{3 \times W^2}{8}}{\frac{W}{2} \times rtt}$$ or, after having eliminated W, $$Throughput = \sqrt{\frac{3}{2}} \times \frac{MSS}{rtt \times \sqrt{p}}$$

More detailed models and the analysis of simulations have shown that a first order model of the TCP throughput when losses occur was $Throughput \approx \frac{k \times MSS}{rtt \times \sqrt{p}}$. This is an important result which shows that :

- TCP connections with a small round-trip-time can achieve a higher throughput than TCP connections having a longer round-trip-time when losses occur. This implies that the TCP congestion control scheme is not completely fair since it favors the connections that have the shorter round-trip-times.
- TCP connections that use a large MSS can achieve a higher throughput that the TCP connections that use a shorter MSS. This creates another source of unfairness between TCP connections. However, it should be noted that today most hosts are using almost the same MSS, roughly 1460 bytes.

In general, the maximum throughput that can be achieved by a TCP connection depends on its maximum window size and the round-trip-time if there are no losses. If there are losses, it depends on the MSS, the round-trip-time and the loss ratio.

$$Throughput < \min(\frac{window}{rtt}, \frac{k \times MSS}{rtt \times \sqrt{p}})$$

Note: The TCP congestion control zoo

The first TCP congestion control scheme was proposed by Van Jacobson in [Jacobson1988]. In addition to writing the scientific paper, Van Jacobson also implemented the slow-start and congestion avoidance schemes in release 4.3 *Tahoe* of the BSD Unix distributed by the University of Berkeley. Later, he improved the congestion control by adding the fast retransmit and the fast recovery mechanisms in the *Reno* release of 4.3 BSD Unix. Since then, many researchers have proposed, simulated and implemented modifications to the TCP congestion control scheme. Some of these modifications are still used today, e.g. :

- *NewReno* (**RFC 3782**), which was proposed as an improvement of the fast recovery mechanism in the *Reno* implementation.
- *TCP Vegas*, which uses changes in the round-trip-time to estimate congestion in order to avoid it [BOP1994]. This is one of the examples of the delay-based congestion control algorithms. A Vegas sender continuously measures the evolution of the round-trip-time and slows down when the round-trip-time increases significantly. This enables Vegas to prevent congestion when used alone. Unfortunately, if Vegas senders compete with more aggressive TCP congestion control schemes that only react to losses, Vegas senders may have difficulties to user their fair share of the available bandwidth.
- *CUBIC*, which was designed for high bandwidth links and is the default congestion control scheme in Linux since the Linux 2.6.19 kernel [HRX2008]. It is now used by several operating systems and is becoming the default congestion control scheme **RFC 8312**. A key difference between CUBIC and the TCP congestion control scheme described in this chapter is that CUBIC is much more aggressive when probing the network. Instead of relying on additive increase after a fast recovery, a CUBIC sender adjusts its congestion by using a cubic function. Thanks to this function, the congestion windows grows faster. This is particularly important in high-bandwidth delay networks.
- *BBR*, which is being developed by Google researchers and is included in recent Linux kernels [CCG+2016]. BBR periodically estimates the available bandwidth and the round-trip-times. To adapt to changes in network conditions, BBR regularly tries to send at 1.25 times the current bandwidth. This enables BBR senders to probe the network, but can also cause large amount of losses. Recent scientific articles indicate that BBR is unfair to other congestion control schemes in specific conditions [WMSS2019].

A wide range of congestion control schemes have been proposed in the scientific literature and several of them have been widely deployed. A detailed comparison of these congestion control schemes is outside the scope of this chapter. A recent survey paper describing many of the implemented TCP congestion control schemes may be found in [TKU2019].

3.14 The network layer

The main objective of the network layer is to allow hosts, connected to different networks, to exchange information through intermediate systems called *router*. The unit of information in the network layer is called a *packet*.

Before explaining the network layer in detail, it is useful to begin by analyzing the service provided by the *datalink* layer. There are many variants of the datalink layer. Some provide a connection-oriented service while others provide a connectionless service. In this section, we focus on connectionless datalink layer services as they are the most widely

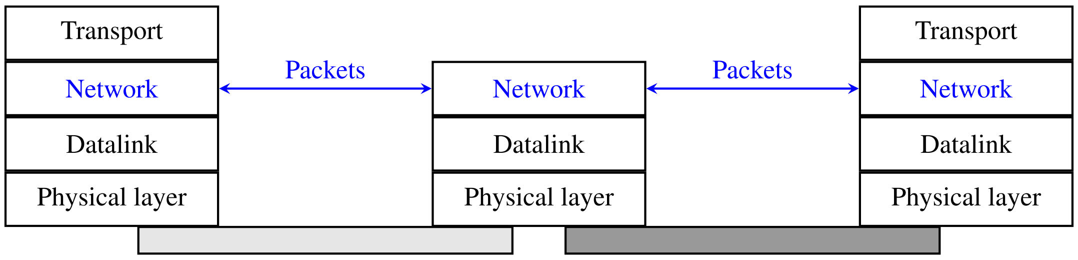


Fig. 38: The network layer in the reference model

used. Using a connection-oriented datalink layer causes some problems that are beyond the scope of this chapter. See **RFC 3819** for a discussion on this topic.

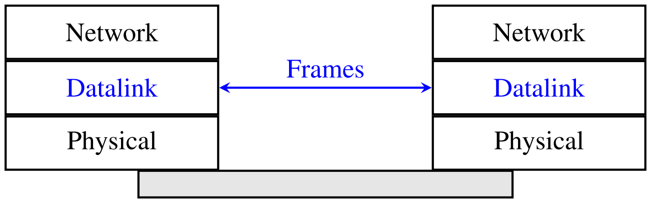


Fig. 39: The point-to-point datalink layer

There are three main types of datalink layers. The simplest datalink layer is when there are only two communicating systems that are directly connected through the physical layer. Such a datalink layer is used when there is a point-to-point link between the two communicating systems. The two systems can be hosts or routers. PPP (Point-to-Point Protocol), defined in **RFC 1661**, is an example of such a point-to-point datalink layer. Datalink layers exchange *frames* and a datalink *frame* sent by a datalink layer entity on the left is transmitted through the physical layer, so that it can reach the datalink layer entity on the right. Point-to-point datalink layers can either provide an unreliable service (frames can be corrupted or lost) or a reliable service (in this case, the datalink layer includes retransmission mechanisms similar to the ones used in the transport layer). The unreliable service is frequently used above physical layers (e.g. optical fiber, twisted pairs) having a low bit error ratio while reliability mechanisms are often used in wireless networks to recover locally from transmission errors.

The second type of datalink layer is the one used in Local Area Networks (LAN). Conceptually, a LAN is a set of communicating devices such that any two devices can directly exchange frames through the datalink layer. Both hosts and routers can be connected to a LAN. Some LANs only connect a few devices, but there are LANs that can connect hundreds or even thousands of devices.

In the next chapter, we describe the organization and the operation of Local Area Networks. An important difference between the point-to-point datalink layers and the datalink layers used in LANs is that in a LAN, each communicating device is identified by a unique *datalink layer address*. This address is usually embedded in the hardware of the device and different types of LANs use different types of datalink layer addresses. Most LANs use 48-bits long addresses that are usually called *MAC* addresses. A communicating device attached to a LAN can send a datalink frame to any other communicating device that is attached to the same LAN. Most LANs also support special broadcast and multicast datalink layer addresses. A frame sent to the broadcast address of the LAN is delivered to all communicating devices that are attached to the LAN. The multicast addresses are used to identify groups of communicating devices. When a frame is sent towards a multicast datalink layer address, it is delivered by the LAN to all communicating devices that belong to the corresponding group.

The third type of datalink layers are used in Non-Broadcast Multi-Access (NBMA) networks. These networks are used to interconnect devices like a LAN. All devices attached to an NBMA network are identified by a unique datalink layer address. However, and this is the main difference between an NBMA network and a traditional LAN, the NBMA

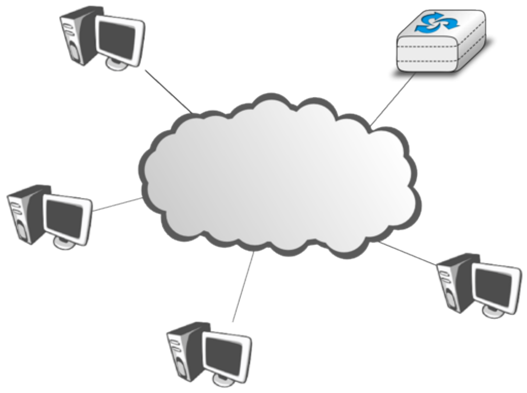


Fig. 40: A local area network

service only supports unicast. The datalink layer service provided by an NBMA network supports neither broadcast nor multicast.

Unfortunately no datalink layer is able to send frames of unlimited side. Each datalink layer is characterized by a maximum frame size. There are more than a dozen different datalink layers and unfortunately most of them use a different maximum frame size. The network layer must cope with the heterogeneity of the datalink layer.

3.14.1 IP version 6

In the late 1980s and early 1990s the growth of the Internet was causing several operational problems on routers. Many of these routers had a single CPU and up to 1 MByte of RAM to store their operating system, packet buffers and routing tables. Given the rate of allocation of IPv4 prefixes to companies and universities willing to join the Internet, the routing tables where growing very quickly and some feared that all IPv4 prefixes would quickly be allocated. In 1987, a study cited in **RFC 1752**, estimated that there would be 100,000 networks in the near future. In August 1990, estimates indicated that the class B space would be exhausted by March 1994. Two types of solution were developed to solve this problem. The first short term solution was the introduction of Classless Inter Domain Routing (*CIDR*). A second short term solution was the Network Address Translation (*NAT*) mechanism, defined in **RFC 1631**. NAT allowed multiple hosts to share a single public IPv4 address.

However, in parallel with these short-term solutions, which have allowed the IPv4 Internet to continue to be usable until now, the Internet Engineering Task Force started working on developing a replacement for IPv4. This work started with an open call for proposals, outlined in **RFC 1550**. Several groups responded to this call with proposals for a next generation Internet Protocol (IPng) :

- TUBA proposed in **RFC 1347** and **RFC 1561**
- PIP proposed in **RFC 1621**
- SIPP proposed in **RFC 1710**

The IETF decided to pursue the development of IPng based on the SIPP proposal. As IP version *5* was already used by the experimental ST-2 protocol defined in **RFC 1819**, the successor of IP version 4 is IP version 6. The initial IP version 6 defined in **RFC 1752** was designed based on the following assumptions :

- IPv6 addresses are encoded as a 128 bits field
- The IPv6 header has a simple format that can easily be parsed by hardware devices
- A host should be able to configure its IPv6 address automatically
- Security must be part of IPv6

Note: The IPng address size

When the work on IPng started, it was clear that 32 bits was too small to encode an IPng address and all proposals used longer addresses. However, there were many discussions about the most suitable address length. A first approach, proposed by SIPP in **RFC 1710**, was to use 64 bit addresses. A 64 bits address space was 4 billion times larger than the IPv4 address space and, furthermore, from an implementation perspective, 64 bit CPUs were being considered and 64 bit addresses would naturally fit inside their registers. Another approach was to use an existing address format. This was the TUBA proposal (**RFC 1347**) that reuses the ISO CLNP 20 bytes addresses. The 20 bytes addresses provided room for growth, but using ISO CLNP was not favored by the IETF partially due to political reasons, despite the fact that mature CLNP implementations were already available. 128 bits appeared to be a reasonable compromise at that time.

IPv6 addressing architecture

The experience of IPv4 revealed that the scalability of a network layer protocol heavily depends on its addressing architecture. The designers of IPv6 spent a lot of effort defining its addressing architecture **RFC 3513**. All IPv6 addresses are 128 bits wide. This implies that there are $340,282,366,920,938,463,463,374,607,431,768,211,456(3.4 \times 10^{38})$ different IPv6 addresses. As the surface of the Earth is about 510,072,000 km^2, this implies that there are about 6.67×10^{23} IPv6 addresses per square meter on Earth. Compared to IPv4, which offers only 8 addresses per square kilometer, this is a significant improvement on paper.

Note: Textual representation of IPv6 addresses

It is sometimes necessary to write IPv6 addresses in text format, e.g. when manually configuring addresses or for documentation purposes. The preferred format for writing IPv6 addresses is `x:x:x:x:x:x:x:x`, where the `x` 's are hexadecimal digits representing the eight 16-bit parts of the address. Here are a few examples of IPv6 addresses :

- `abcd:ef01:2345:6789:abcd:ef01:2345:6789`
- `2001:db8:0:0:8:800:200c:417a`
- `fe80:0:0:0:219:e3ff:fed7:1204`

IPv6 addresses often contain a long sequence of bits set to `0`. In this case, a compact notation has been defined. With this notation, *::* is used to indicate one or more groups of 16 bits blocks containing only bits set to *0*. For example,

- `2001:db8:0:0:8:800:200c:417a` is represented as `2001:db8::8:800:200c:417a`
- `ff01:0:0:0:0:0:0:101` is represented as `ff01::101`
- `0:0:0:0:0:0:0:1` is represented as `::1`
- `0:0:0:0:0:0:0:0` is represented as `::`

An IPv6 prefix can be represented as *address/length*, where *length* is the length of the prefix in bits. For example, the three notations below correspond to the same IPv6 prefix :

- `2001:0db8:0000:cd30:0000:0000:0000:0000` / `60`
- `2001:0db8::cd30:0:0:0:0` / `60`
- `2001:0db8:0:cd30::` / `60`

IPv6 supports unicast, multicast and anycast addresses. An IPv6 unicast address is used to identify one datalink-layer interface on a host. If a host has several datalink layer interfaces (e.g. an Ethernet interface and a WiFi interface), then it needs several IPv6 addresses. In general, an IPv6 unicast address is structured as shown in the figure below.

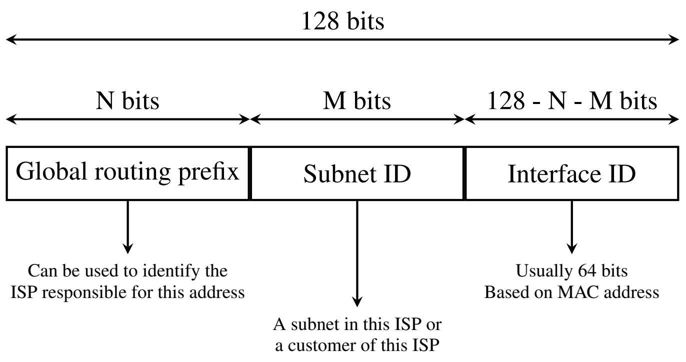

128 bits
N bits
M bits
128 - N - M bits
Global routing prefix
Subnet ID
Interface ID
Can be used to identify the ISP responsible for this address
A subnet in this ISP or a customer of this ISP
Usually 64 bits Based on MAC address

Fig. 41: Structure of IPv6 unicast addresses

An IPv6 unicast address is composed of three parts :

1. A *global routing prefix* that is assigned to the Internet Service Provider that owns this block of addresses
2. A *subnet identifier* that identifies a customer of the ISP
3. An *interface identifier* that identifies a particular interface on a host

The subnet identifier plays a key role in the scalability of network layer addressing architecture. An important point to be defined in a network layer protocol is the allocation of the network layer addresses. A naive allocation scheme would be to provide an address to each host when the host is attached to the Internet on a first come first served basis. With this solution, a host in Belgium could have address `2001:db8::1` while another host located in Africa would use address `2001:db8::2`. Unfortunately, this would force all routers on the Internet to maintain one route towards each host. In the network layer, scalability is often a function of the number of routes stored on the router. A network will usually work better if its routers store fewer routes and network administrators usually try to minimize the number of routes that are known by their routers. For this, they often divide their network prefix in smaller blocks. For example, consider a company with three campuses, a large one and two smaller ones. The network administrator would probably divide his block of addresses as follows :

- the bottom half is used for the large campus
- the top half is divided in two smaller blocks, one for each small campus

Inside each campus, the same division can be done, for example on a per building basis, starting from the buildings that host the largest number of nodes, e.g. the company datacenter. In each building, the same division can be done on a per floor basis, ... The advantage of such a hierarchical allocation of the addresses is that the routers in the large campus only need one route to reach a router in the smaller campus. The routers in the large campus would know more routes about the buildings in their campus, but they do not need to know the details of the organization of each smaller campus.

To preserve the scalability of the routing system, it is important to minimize the number of routes that are stored on each router. A router cannot store and maintain one route for each of the almost 1 billion hosts that are connected to today's Internet. Routers should only maintain routes towards blocks of addresses and not towards individual hosts. For this, hosts are grouped in *subnets* based on their location in the network. A typical subnet groups all the hosts that are part of the same enterprise. An enterprise network is usually composed of several LANs interconnected by routers. A small block of addresses from the Enterprise's block is usually assigned to each LAN.

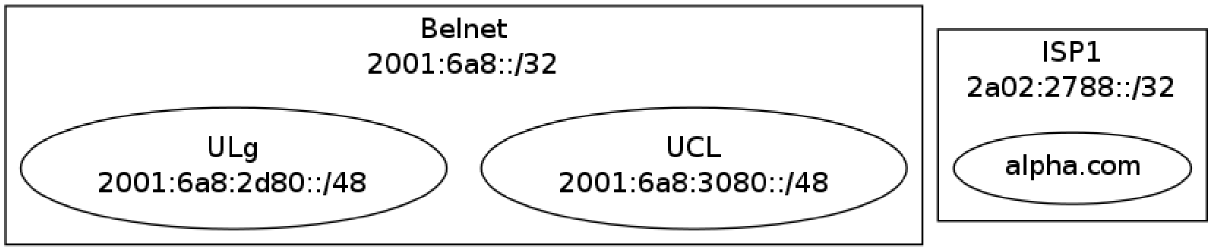
In today's deployments, interface identifiers are always 64 bits wide. This implies that while there are 2^{128} different IPv6 addresses, they must be grouped in 2^{64} subnets. This could appear as a waste of resources, however using 64 bits for the host identifier allows IPv6 addresses to be auto-configured and also provides some benefits from a security point of view, as explained in section *ICMPv6*.

In practice, there are several types of IPv6 unicast address. Most of the IPv6 unicast addresses are allocated in blocks under the responsibility of IANA. The current IPv6 allocations are part of the *2000::/3* address block. Regional Internet Registries (RIR) such as RIPE in Europe, ARIN in North-America or AfriNIC in Africa have each received a block of IPv6 addresses that they sub-allocate to Internet Service Providers in their region. The ISPs then sub-allocate addresses to their customers.

When considering the allocation of IPv6 addresses, two types of address allocations are often distinguished. The RIRs allocate *provider-independent (PI)* addresses. PI addresses are usually allocated to Internet Service Providers and large companies that are connected to at least two different ISPs [CSP2009]. Once a PI address block has been allocated to a company, this company can use its address block with the provider of its choice and change its provider at will. Internet Service Providers allocate *provider-aggregatable (PA)* address blocks from their own PI address block to their customers. A company that is connected to only one ISP should only use PA addresses. The drawback of PA addresses is that when a company using a PA address block changes its provider, it needs to change all the addresses that it uses. This can be a nightmare from an operational perspective and many companies are lobbying to obtain *PI* address blocks even if they are small and connected to a single provider. The typical size of the IPv6 address blocks are :

- `/32` for an Internet Service Provider
- `/48` for a single company
- `/56` for small user sites
- `/64` for a single user (e.g. a home user connected via ADSL)
- `/128` in the rare case when it is known that no more than one host will be attached

There is one difficulty with the utilization of these IPv6 prefixes. Consider Belnet, the Belgian research ISP that has been allocated the `2001:6a8::/32` prefix. Universities are connected to Belnet. UCLouvain uses prefix `2001:6a8:3080::/48` while the University of Liege uses `2001:6a8:2d80::/48`. A commercial ISP uses prefix `2a02:2788::/32`. Both Belnet and the commercial ISP are connected to the global Internet.



The Belnet network advertises prefix `2001:6a8::/32` that includes the prefixes from both UCLouvain and ULg. These two subnetworks can be easily reached from any internet connected host. After a few years, UCLouvain decides to increase the redundancy of its Internet connectivity and buys transit service from ISP1. A direct link between UCLouvain and the commercial ISP appears on the network and UCLouvain expects to receive packets from both Belnet and the commercial ISP.

Now, consider how a router inside `alpha.com` would reach a host in the `UCLouvain` network. This router has two routes towards `2001:6a8:3080::1`. The first one, for prefix `2001:6a8:3080::/48` is via the direct link between the commercial ISP and UCLouvain. The second one, for prefix `2001:6a8::/32` is via the Internet and Belnet. Since **RFC 1519** when a router knows several routes towards the same destination address, it must forward packets along the route having the longest prefix length. In the case of `2001:6a8:3080::1`, this is the route `2001:6a8:3080::/48` that is used to forward the packet. This forwarding rule is called the *longest prefix match* or the *more specific match*. All IP routers implement this forwarding rule.

To understand the *longest prefix match* forwarding, consider the IPv6 routing below.

```
Destination                         Gateway
::/0                                fe80::dead:beef
::1                                 ::1
2a02:2788:2c4:16f::/64              eth0
2001:6a8:3080::/48                  fe80::bad:cafe
2001:6a8:2d80::/48                  fe80::bad:bad
2001:6a8::/32                       fe80::aaaa:bbbb
```

With the longest match rule, the route `::/0` plays a particular role. As this route has a prefix length of *0* bits, it matches all destination addresses. This route is often called the *default* route.

- a packet with destination `2a02:2788:2c4:16f::1` received by router *R* is destined to a host on interface `eth0`.
- a packet with destination `2001:6a8:3080::1234` matches three routes : `::/0`, `2001:6a8::/32` and `2001:6a8:3080::/48`. The packet is forwarded via gateway `fe80::bad:cafe`
- a packet with destination `2001:1890:123a::1:1e` matches one route : `::/0`. The packet is forwarded via `fe80::dead:beef`
- a packet with destination `2001:6a8:3880:40::2` matches two routes : `2001:6a8::/32` and `::/0`. The packet is forwarded via `fe80::aaaa:bbbb`

The longest prefix match can be implemented by using different data structures One possibility is to use a trie. Details on how to implement efficient packet forwarding algorithms may be found in [Varghese2005].

For the companies that want to use IPv6 without being connected to the IPv6 Internet, **RFC 4193** defines the *Unique Local Unicast (ULA)* addresses (`fc00::/7`). These ULA addresses play a similar role as the private IPv4 addresses defined in **RFC 1918**. However, the size of the `fc00::/7` address block allows ULA to be much more flexible than private IPv4 addresses.

Furthermore, the IETF has reserved some IPv6 addresses for a special usage. The two most important ones are :

- `0:0:0:0:0:0:0:1` (`::1` in compact form) is the IPv6 loopback address. This is the address of a logical interface that is always up and running on IPv6 enabled hosts.
- `0:0:0:0:0:0:0:0` (`::` in compact form) is the unspecified IPv6 address. This is the IPv6 address that a host can use as source address when trying to acquire an official address.

The last type of unicast IPv6 addresses are the *Link Local Unicast* addresses. These addresses are part of the *fe80::/10* address block and are defined in **RFC 4291**. Each host can compute its own link local address by concatenating the *fe80::/64* prefix with the 64 bits identifier of its interface. Link local addresses can be used when hosts that are attached to the same link (or local area network) need to exchange packets. They are used notably for address discovery and auto-configuration purposes. Their usage is restricted to each link and a router cannot forward a packet whose source or destination address is a link local address. Link local addresses have also been defined for IPv4 **RFC 3927**. However, the IPv4 link local addresses are only used when a host cannot obtain a regular IPv4 address, e.g. on an isolated LAN.

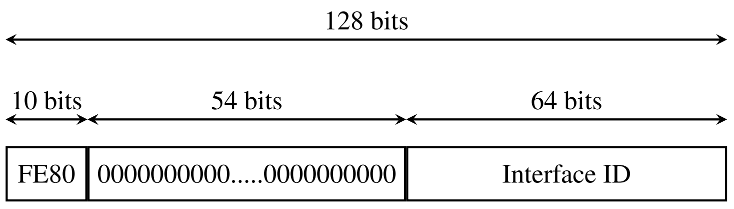
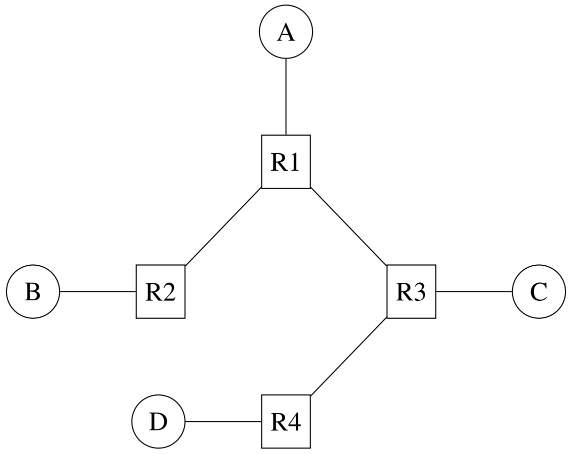


Fig. 42: IPv6 link local address structure

Note: All IPv6 hosts have several addresses

An important consequence of the IPv6 unicast addressing architecture and the utilization of link-local addresses is that each IPv6 host has several IPv6 addresses. This implies that all IPv6 stacks must be able to handle multiple IPv6 addresses.

The addresses described above are unicast addresses. These addresses are used to identify (interfaces on) hosts and routers. They can appear as source and destination addresses in the IPv6 packets. When a host sends a packet towards a unicast address, this packet is delivered by the network to its final destination. There are situations, such as when delivering video or television signal to a large number of receivers, where it is useful to have a network that can efficiently deliver the same packet to a large number of receivers. This is the *multicast* service. A multicast service can be provided in a LAN. In this case, a multicast address identifies a set of receivers and each frame sent towards this address is delivered to all receivers in the group. Multicast can also be used in a network containing routers and hosts. In this case, a multicast address identifies also a group of receivers and the network delivers efficiently each multicast packet to all members of the group. Consider for example the network below.



Assume that `B` and `D` are part of a multicast group. If `A` sends a multicast packet towards this group, then `R1` will replicate the packet to forward it to `R2` and `R3`. `R2` would forward the packet towards `B`. `R3` would forward the packet towards `R4` that would deliver it to `D`.

Finally, **RFC 4291** defines the structure of the IPv6 multicast addresses[1]. This structure is depicted in the figure below.

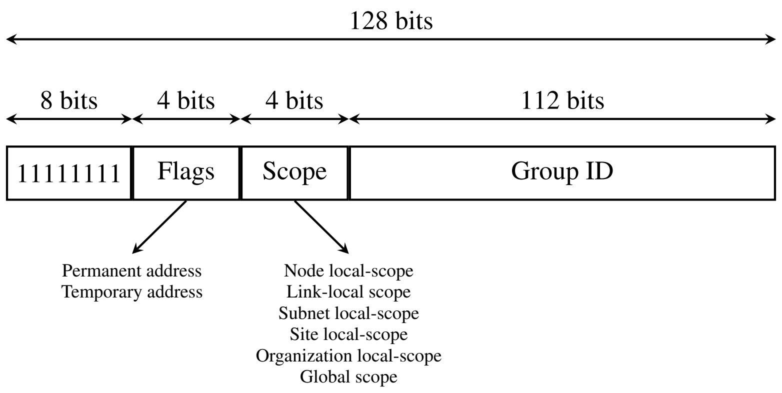


Fig. 43: IPv6 multicast address structure

[1] The full list of allocated IPv6 multicast addresses is available at http://www.iana.org/assignments/ipv6-multicast-addresses

The low order 112 bits of an IPv6 multicast address are the group's identifier. The high order bits are used as a marker to distinguish multicast addresses from unicast addresses. Notably, the 4-bit *Flags* field indicates whether the address is temporary or permanent. Finally, the *Scope* field indicates the boundaries of the forwarding of packets destined to a particular address. A link-local scope indicates that a router should not forward a packet destined to such a multicast address. An organization local-scope indicates that a packet sent to such a multicast destination address should not leave the organization. Finally the global scope is intended for multicast groups spanning the global Internet.

Among these addresses, some are well known. For example, all hosts automatically belong to the `ff02::1` multicast group while all routers automatically belong to the `ff02::2` multicast group. A detailed discussion of IPv6 multicast is outside the scope of this chapter.

IPv6 packet format

The IPv6 packet format was heavily inspired by the packet format proposed for the SIPP protocol in **RFC 1710**. The standard IPv6 header defined in **RFC 2460** occupies 40 bytes and contains 8 different fields, as shown in the figure below.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version| Traffic Class |           Flow Label                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Payload Length        |  Next Header  |   Hop Limit   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                         Source Address                        +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                      Destination Address                      +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 44: The IP version 6 header (**RFC 2460**)

Apart from the source and destination addresses, the IPv6 header contains the following fields :

- *Version* : a 4 bits field set to *6* and intended to allow IP to evolve in the future if needed
- *Traffic class* : this 8 bits field indicates the type of service expected by this packet and contains the `CE` and `ECT` flags that are used by *Explicit Congestion Notification*
- *Flow Label* : this field was initially intended to be used to tag packets belonging to the same *flow*. A recent document, **RFC 6437** describes some possible usages of this field, but it is too early to tell whether it will be really used.
- *Payload Length* : this is the size of the packet payload in bytes. As the length is encoded as a 16 bits field, an IPv6 packet can contain up to 65535 bytes of payload.
- *Next Header* : this 8-bit field indicates the type[2] of header that follows the IPv6 header. It can be a transport layer header (e.g. *6* for TCP or *17* for UDP) or an IPv6 option.
- *Hop Limit* : this 8-bit field indicates the number of routers that can forward the packet. It is decremented by one by each router and prevents packets from looping forever inside the network.

[2] The IANA maintains the list of all allocated Next Header types at http://www.iana.org/assignments/protocol-numbers/

It is interesting to note that there is no checksum inside the IPv6 header. This is mainly because all datalink layers and transport protocols include a checksum or a CRC to protect their frames/segments against transmission errors. Adding a checksum in the IPv6 header would have forced each router to recompute the checksum of all packets, with limited benefit in detecting errors. In practice, an IP checksum allows for catching errors that occur inside routers (e.g. due to memory corruption) before the packet reaches its destination. However, this benefit was found to be too small given the reliability of current memories and the cost of computing the checksum on each router[3].

When a host receives an IPv6 packet, it needs to determine which transport protocol (UDP, TCP, SCTP, . . .) needs to handle the payload of the packet. This is the first role of the *Next header* field. The IANA which manages the allocation of Internet resources and protocol parameters, maintains an official list of transport protocols[2]. The following protocol numbers are reserved :

- `TCP` uses *Next Header* number `6`
- `UDP` uses *Next Header* number `17`
- `SCTP` uses *Next Header* number `132`

For example, an IPv6 packet that contains an TCP segment would appear as shown in the figure below.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version| Traffic Class |           Flow Label                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Payload Length        |       6       |   Hop Limit   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                         Source Address                        +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                      Destination Address                      +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |       |C|E|U|A|P|R|S|F|                               |
| Offset| Res.  |W|C|R|C|S|S|Y|I|            Window             |
|       |       |R|E|G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| ...
```

Fig. 45: An IPv6 packet containing an TCP segment

However, the *Next header* has broader usages than simply indicating the transport protocol which is responsible for the packet payload. An IPv6 packet can contain a chain of headers and the last one indicates the transport protocol that is responsible for the packet payload. Supporting a chain of headers is a clever design from an extensibility viewpoint. As we will see, this chain of headers has several usages.

RFC 2460 defines several types of IPv6 extension headers that could be added to an IPv6 packet :

[3] When IPv4 was designed, the situation was different. The IPv4 header includes a checksum that only covers the network header. This checksum is computed by the source and updated by all intermediate routers that decrement the TTL, which is the IPv4 equivalent of the *HopLimit* used by IPv6.

- *Hop-by-Hop Options* header. This option is processed by routers and hosts.
- *Destination Options* header. This option is processed only by hosts.
- *Routing* header. This option is processed by some nodes.
- *Fragment* header. This option is processed only by hosts.
- *Authentication* header. This option is processed only by hosts.
- *Encapsulating Security Payload*. This option is processed only by hosts.

The last two headers are used to add security above IPv6 and implement IPSec. They are described in **RFC 2402** and **RFC 2406** and are outside the scope of this document.

The *Hop-by-Hop Options* header was designed to make IPv6 easily extensible. In theory, this option could be used to define new fields that were not foreseen when IPv6 was designed. It is intended to be processed by both routers and hosts. Deploying an extension to a network protocol can be difficult in practice since some nodes already support the extensions while others still use the old version and do not understand the extension. To deal with this issue, the IPv6 designers opted for a Type-Length-Value encoding of these IPv6 options. The *Hop-by-Hop Options* header is encoded as shown below.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next Header  |  Hdr Ext Len  |  Opt. Type    |  Opt. Len     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
.                                                               .
.                            Options                            .
.                                                               .
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 46: The IPv6 *Hop-by-Hop Options* header

In this optional header, the *Next Header* field is used to support the chain of headers. It indicates the type of the next header in the chain. IPv6 headers have different lengths. The *Hdr Ext Len* field indicates the total length of the option header in bytes. The *Opt. Type* field indicates the type of option. These types are encoded such that their high order bits specify how the header needs to be handled by nodes that do not recognize it. The following values are defined for the two high order bits :

- `00` : if a node does not recognize this header, it can be safely skipped and the processing continues with the subsequent header
- `01` : if a node does not recognize this header, the packet must be discarded
- `10` (resp. `11`) : if a node does not recognize this header, it must return a control packet (ICMP, see later) back to the source (resp. except if the destination was a multicast address)

This encoding allows the designers of protocol extensions to specify whether the option must be supported by all nodes on a path or not. Still, deploying such an extension can be difficult in practice.

Two *hop-by-hop* options have been defined. **RFC 2675** specifies the jumbogram that enables IPv6 to support packets containing a payload larger than 65535 bytes. These jumbo packets have their *payload length* set to *0* and the jumbogram option contains the packet length as a 32 bits field. Such packets can only be sent from a source to a destination if all the routers on the path support this option. However, as of this writing it does not seem that the jumbogram option has been implemented. The router alert option defined in **RFC 2711** is the second example of a *hop-by-hop* option. The packets that contain this option should be processed in a special way by intermediate routers. This option is used for IP packets that carry Resource Reservation Protocol (RSVP) messages, but this is outside the scope of this book.

The *Destinations Option* header uses the same format as the *Hop-by-Hop Options* header. It has some usages, e.g. to support mobile nodes **RFC 6275**, but these are outside the scope of this document.

The *Fragment Options* header is more important. An important problem in the network layer is the ability to handle heterogeneous datalink layers. Most datalink layer technologies can only transmit and receive frames that are shorter than a given maximum frame size. Unfortunately, all datalink layer technologies use different maximum frames sizes.

Each datalink layer has its own characteristics and as indicated earlier, each datalink layer is characterized by a maximum frame size. From IP's point of view, a datalink layer interface is characterized by its *Maximum Transmission Unit (MTU)*. The MTU of an interface is the largest packet (including header) that it can send. The table below provides some common MTU sizes.

Datalink layer	MTU
Ethernet	1500 bytes
WiFi	2272 bytes
ATM (AAL5)	9180 bytes
802.15.4	102 or 81 bytes
Token Ring	4464 bytes
FDDI	4352 bytes

Although IPv6 can send 64 KBytes long packets, few datalink layer technologies that are used today are able to send a 64 KBytes packet inside a frame. Furthermore, as illustrated in the figure below, another problem is that a host may send a packet that would be too large for one of the datalink layers used by the intermediate routers.

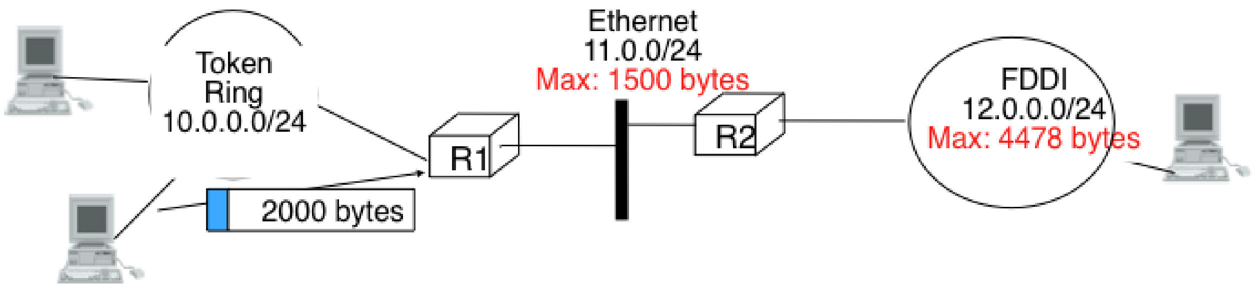


Fig. 47: The need for fragmentation and reassembly

To solve these problems, IPv6 includes a packet fragmentation and reassembly mechanism. In IPv4, fragmentation was performed by both the hosts and the intermediate routers. However, experience with IPv4 has shown that fragmenting packets in routers was costly [KM1995]. For this reason, the developers of IPv6 have decided that routers would not fragment packets anymore. In IPv6, fragmentation is only performed by the source host. If a source has to send a packet which is larger than the MTU of the outgoing interface, the packet needs to be fragmented before being transmitted. In IPv6, each packet fragment is an IPv6 packet that includes the *Fragmentation* header. This header is included by the source in each packet fragment. The receiver uses them to reassemble the received fragments.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next Header  |   Reserved    |      Fragment Offset    |Res|M|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Identification                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 48: IPv6 fragmentation header

If a router receives a packet that is too long to be forwarded, the packet is dropped and the router returns an ICMPv6 message to inform the sender of the problem. The sender can then either fragment the packet or perform Path MTU discovery. In IPv6, packet fragmentation is performed only by the source by using IPv6 options.

In IPv6, fragmentation is performed exclusively by the source host and relies on the fragmentation header. This 64 bits header is composed of six fields :

- a *Next Header* field that indicates the type of the header that follows the fragmentation header

- two *Reserved* fields set to *0*.
- the *Fragment Offset* is a 13-bit unsigned integer that contains the offset, in 8 bytes units, of the data following this header, relative to the start of the original packet.
- the *More* flag, which is set to *0* in the last fragment of a packet and to *1* in all other fragments.
- the 32-bit *Identification* field indicates to which original packet a fragment belongs. When a host sends fragmented packets, it should ensure that it does not reuse the same *identification* field for packets sent to the same destination during a period of *MSL* seconds. This is easier with the 32 bits *identification* used in the IPv6 fragmentation header, than with the 16 bits *identification* field of the IPv4 header.

Some IPv6 implementations send the fragments of a packet in increasing fragment offset order, starting from the first fragment. Others send the fragments in reverse order, starting from the last fragment. The latter solution can be advantageous for the host that needs to reassemble the fragments, as it can easily allocate the buffer required to reassemble all fragments of the packet upon reception of the last fragment. When a host receives the first fragment of an IPv6 packet, it cannot know a priori the length of the entire IPv6 packet.

The figure below provides an example of a fragmented IPv6 packet containing a UDP segment. The *Next Header* type reserved for the IPv6 fragmentation option is 44.

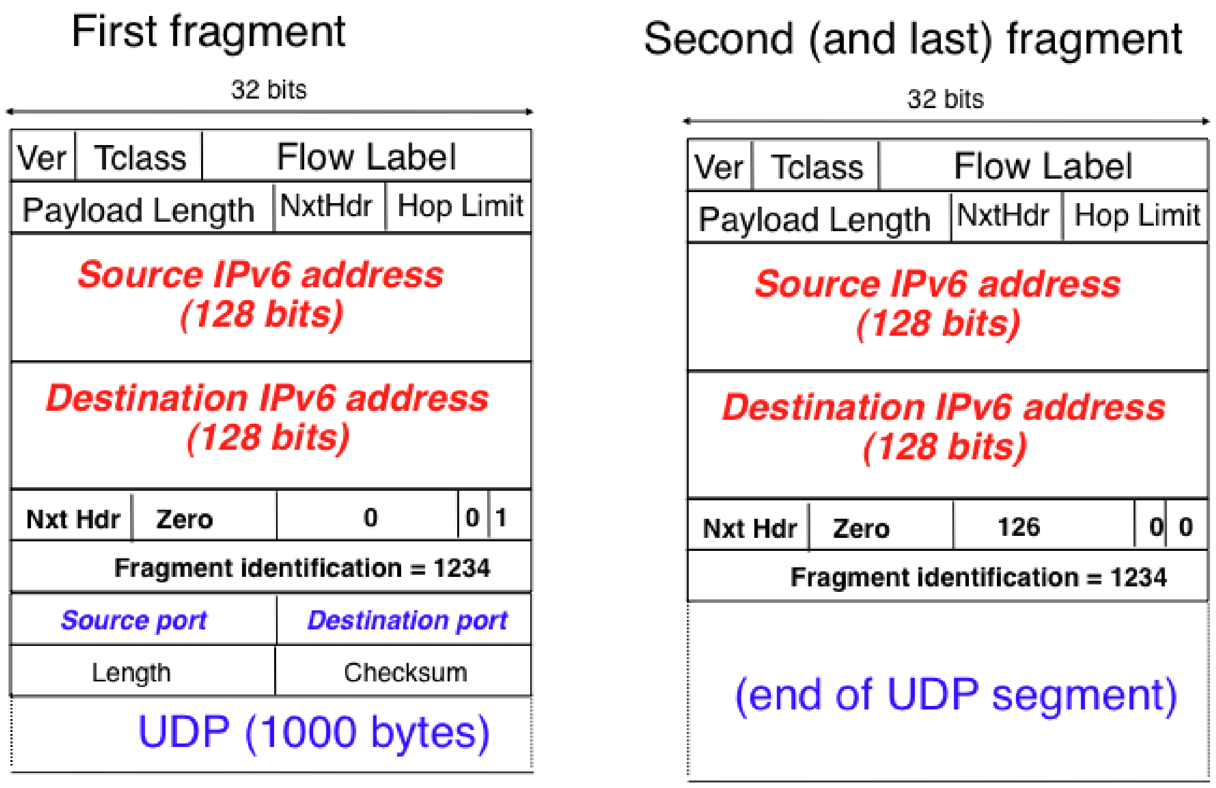


Fig. 49: IPv6 fragmentation example

The following pseudo-code details the IPv6 fragmentation, assuming that the packet does not contain options.

```
# mtu : maximum size of the packet (including header) of outgoing link
# In Scapy-like notation (see https://github.com/secdev/scapy)
if p.len < mtu:
    send(p)
else:
    # packet is too large
```

(continues on next page)

(continued from previous page)

```
    # 40 refers to the size of the IPv6 header
    maxpayload = 8 * int((mtu - 40) / 8)  # must be n times 8 bytes
    # packet must be fragmented
    payload = p[IPv6].payload
    pos = 0
    id = globalCounter
    globalCounter += 1
    while len(payload) > 0:
        if len(payload) > maxpayload:
            toSend = IPv6(dst=p.dst, src=p.src, plen=mtu,
                          hlim=p.hlim, nh=44)/IPv6ExtHdrFrament(
                              id=id, offset=p.offset + (pos/8), m=True,
                              nh=p.nh)/payload[0:maxpayload]
            pos = pos + maxpayload
            payload = payload[maxpayload+1:]
        else:
            # The last fragment
            toSend = IPv6(dst=p.dst, src=p.src, plen=len(payload),
                          hlim=p.hlim, nh=44)/IPv6ExtHdrFrament(
                              id=id, offset=p.offset + (pos/8), m=False,
                              nh=p.nh)/payload
            payload = []

        forward(toSend)
```

In the above pseudocode, we maintain a single 32 bits counter that is incremented for each packet that needs to be fragmented. Other implementations to compute the packet identification are possible. **RFC 2460** only requires that two fragmented packets that are sent within the MSL between the same pair of hosts have different identifications.

The fragments of an IPv6 packet may arrive at the destination in any order, as each fragment is forwarded independently in the network and may follow different paths. Furthermore, some fragments may be lost and never reach the destination.

The reassembly algorithm used by the destination host is roughly as follows. First, the destination can verify whether a received IPv6 packet is a fragment or not by checking whether it contains a fragment header. If so, all fragments with the some identification must be reassembled together. The reassembly algorithm relies on the *Identification* field of the received fragments to associate a fragment with the corresponding packet being reassembled. Furthermore, the *Fragment Offset* field indicates the position of the fragment payload in the original non-fragmented packet. Finally, the packet with the *M* flag reset allows the destination to determine the total length of the original non-fragmented packet.

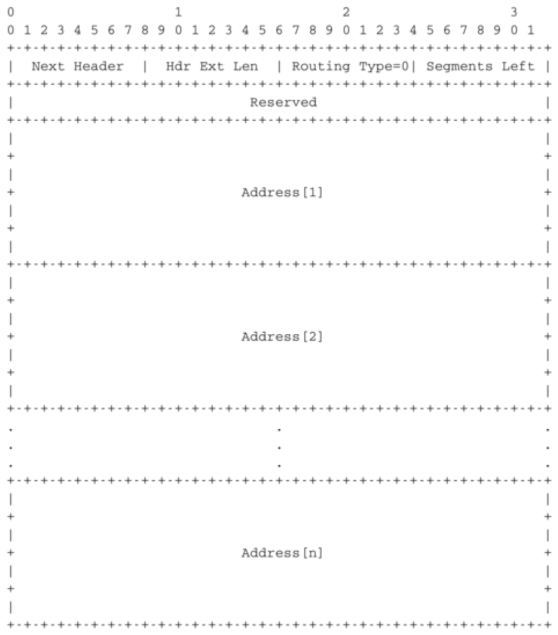
Note that the reassembly algorithm must deal with the unreliability of the IP network. This implies that a fragment may be duplicated or a fragment may never reach the destination. The destination can easily detect fragment duplication thanks to the *Fragment Offset*. To deal with fragment losses, the reassembly algorithm must bind the time during which the fragments of a packet are stored in its buffer while the packet is being reassembled. This can be implemented by starting a timer when the first fragment of a packet is received. If the packet has not been reassembled upon expiration of the timer, all fragments are discarded and the packet is considered to be lost.

Note: Header compression on low bandwidth links

Given the size of the IPv6 header, it can cause huge overhead on low bandwidth links, especially when small packets are exchanged such as for Voice over IP applications. In such environments, several techniques can be used to reduce the overhead. A first solution is to use data compression in the datalink layer to compress all the information exchanged [Thomborson1992]. These techniques are similar to the data compression algorithms used in tools such as `compress(1)` or `gzip(1)` **RFC 1951**. They compress streams of bits without taking advantage of the fact that these streams contain IP packets with a known structure. A second solution is to compress the IP and TCP header.

These header compression techniques, such as the one defined in **RFC 5795** take advantage of the redundancy found in successive packets from the same flow to significantly reduce the size of the protocol headers. Another solution is to define a compressed encoding of the IPv6 header that matches the capabilities of the underlying datalink layer **RFC 4944**.

The last type of *IPv6 header extension* is the *Routing* header. The `type 0` routing header defined in **RFC 2460** is an example of an IPv6 option that must be processed by some routers. This option is encoded as shown below.



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next Header  |  Hdr Ext Len  | Routing Type=0| Segments Left |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Reserved                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                           Address[1]                          +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                           Address[2]                          +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.                               .                               .
.                               .                               .
.                               .                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                           Address[n]                          +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 50: The Type 0 routing header (**RFC 2460**)

The type 0 routing option was intended to allow a host to indicate a loose source route that should be followed by a packet by specifying the addresses of some of the routers that must forward this packet. Unfortunately, further work with this routing header, including an entertaining demonstration with scapy [BE2007] , revealed severe security problems with this routing header. For this reason, loose source routing with the type 0 routing header has been removed from the IPv6 specification **RFC 5095**.

3.14.2 ICMP version 6

It is sometimes necessary for intermediate routers or the destination host to inform the sender of the packet of a problem that occurred while processing a packet. In the TCP/IP protocol suite, this reporting is done by the Internet Control Message Protocol (ICMP). ICMPv6 is defined in **RFC 4443**. It is used both to report problems that occurred while processing an IPv6 packet, but also to distribute addresses.

ICMPv6 messages are carried inside IPv6 packets (the *Next Header* field for ICMPv6 is `58`). Each ICMP message contains a 32 bits header with an 8 bits *type* field, a *code* field and a 16 bits checksum computed over the entire ICMPv6 message. The message body contains a copy of the IPv6 packet in error.

ICMPv6 specifies two classes of messages : error messages that indicate a problem in handling a packet and informational messages. Four types of error messages are defined in **RFC 4443** :

- **1** [*Destination Unreachable*. Such an ICMPv6 message is sent when the destination address of a packet is unreachable. The *code* field of the ICMP header contains additional information about the type of unreachability. The following codes are specified in **RFC 4443**]

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Code      |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         Message Body                          +
|                                                               |
```

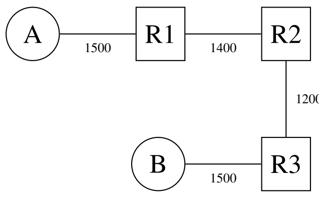
Fig. 51: ICMP version 6 packet format

  - `0` : No route to destination. This indicates that the router that sent the ICMPv6 message did not have a route towards the packet's destination
  - `1` : Communication with destination administratively prohibited. This indicates that a firewall has refused to forward the packet towards its final destination.
  - `2` : Beyond scope of source address. This message can be sent if the source is using link-local addresses to reach a global unicast address outside its subnet.
  - `3` : Address unreachable. This message indicates that the packet reached the subnet of the destination, but the host that owns this destination address cannot be reached.
  - `4` : Port unreachable. This message indicates that the IPv6 packet was received by the destination, but there was no application listening to the specified port.

- `2` : Packet Too Big. The router that was to send the ICMPv6 message received an IPv6 packet that is larger than the MTU of the outgoing link. The ICMPv6 message contains the MTU of this link in bytes. This allows the sending host to implement Path MTU discovery **RFC 1981**
- `3` : Time Exceeded. This error message can be sent either by a router or by a host. A router would set *code* to *0* to report the reception of a packet whose *Hop Limit* reached *0*. A host would set *code* to *1* to report that it was unable to reassemble received IPv6 fragments.
- `4` : Parameter Problem. This ICMPv6 message is used to report either the reception of an IPv6 packet with an erroneous header field (code *0*) or an unknown *Next Header* or IP option (codes *1* and *2*). In this case, the message body contains the erroneous IPv6 packet and the first 32 bits of the message body contain a pointer to the error.

The *Destination Unreachable* ICMP error message is returned when a packet cannot be forwarded to its final destination. The first four ICMPv6 error messages (type `1`, codes `0-3`) are generated by routers while hosts may return code `4` when there is no application bound to the corresponding port number.

The *Packet Too Big* ICMP messages enable the source host to discover the MTU size that it can safely use to reach a given destination. To understand its operation, consider the (academic) scenario shown in the figure below. In this figure, the labels on each link represent the maximum packet size supported by this link.



If `A` sends a 1500 bytes packet, `R1` will return an ICMPv6 error message indicating a maximum packet length of 1400 bytes. `A` would then fragment the packet before retransmitting it. The small fragment would go through, but the large fragment will be refused by `R2` that would return an ICMPv6 error message. `A` can fragment again the packet and send it to the final destination as two fragments.

In practice, an IPv6 implementation does not store the transmitted packets to be able to retransmit them if needed. However, since TCP (and SCTP) buffer the segments that they transmit, a similar approach can be used in transport protocols to detect the largest MTU on a path towards a given destination. This technique is called PathMTU Discovery **RFC 1981**.

When a TCP segment is transported in an IP packet that is fragmented in the network, the loss of a single fragment forces TCP to retransmit the entire segment (and thus all the fragments). If TCP was able to send only packets that do not require fragmentation in the network, it could retransmit only the information that was lost in the network. In addition, IP reassembly causes several challenges at high speed as discussed in **RFC 4963**. Using IP fragmentation to allow UDP applications to exchange large messages raises several security issues [KPS2003].

ICMPv6 is used by TCP implementations to discover the largest MTU size that is allowed to reach a destination host without causing network fragmentation. A TCP implementation parses the *Packets Too Big* ICMP messages that it receives. These ICMP messages contain the MTU of the router's outgoing link in their *Data* field. Upon reception of such an ICMP message, the source TCP implementation adjusts its Maximum Segment Size (MSS) so that the packets containing the segments that it sends can be forwarded by this router without requiring fragmentation.

Two types of informational ICMPv6 messages are defined in **RFC 4443** : *echo request* and *echo reply*, which are used to test the reachability of a destination by using `ping6(8)`. Each host is supposed[4] to reply with an ICMP *Echo reply* message when it receives an ICMP *Echo request* message. A sample usage of `ping6(8)` is shown below.

```
#ping6 www.ietf.org
PING6(56=40+8+8 bytes) 2001:6a8:3080:2:3403:bbf4:edae:afc3 --> 2001:1890:123a::1:1e
16 bytes from 2001:1890:123a::1:1e, icmp_seq=0 hlim=49 time=156.905 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=1 hlim=49 time=155.618 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=2 hlim=49 time=155.808 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=3 hlim=49 time=155.325 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=4 hlim=49 time=155.493 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=5 hlim=49 time=155.801 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=6 hlim=49 time=155.660 ms
16 bytes from 2001:1890:123a::1:1e, icmp_seq=7 hlim=49 time=155.869 ms
^C
--- www.ietf.org ping6 statistics ---
8 packets transmitted, 8 packets received, 0.0% packet loss
round-trip min/avg/max/std-dev = 155.325/155.810/156.905/0.447 ms
```

Another very useful debugging tool is `traceroute6(8)`. The traceroute man page describes this tool as *"print the route packets take to network host"*. traceroute uses the *Time exceeded* ICMP messages to discover the intermediate routers on the path towards a destination. The principle behind traceroute is very simple. When a router receives an IP packet whose *Hop Limit* is set to `1` it is forced to return to the sending host a *Time exceeded* ICMP message containing the header and the first bytes of the discarded packet. To discover all routers on a network path, a simple solution is to first send a packet whose *Hop Limit* is set to *1*, then a packet whose *Hop Limit* is set to *2*, etc. A sample traceroute6 output is shown below.

```
#traceroute6 www.ietf.org
traceroute6 to www.ietf.org (2001:1890:1112:1::20) from␣
↪2001:6a8:3080:2:217:f2ff:fed6:65c0, 30 hops max, 12 byte packets
 1  2001:6a8:3080:2::1  13.821 ms  0.301 ms  0.324 ms
 2  2001:6a8:3000:8000::1  0.651 ms  0.51 ms  0.495 ms
 3  10ge.cr2.bruvil.belnet.net  3.402 ms  3.34 ms  3.33 ms
 4  10ge.cr2.brueve.belnet.net  3.668 ms 10ge.cr2.brueve.belnet.net  3.988 ms 10ge.
↪cr2.brueve.belnet.net  3.699 ms
 5  belnet.rt1.ams.nl.geant2.net  10.598 ms  7.214 ms  10.082 ms
 6  so-7-0-0.rt2.cop.dk.geant2.net  20.19 ms  20.002 ms  20.064 ms
 7  kbn-ipv6-b1.ipv6.telia.net  21.078 ms  20.868 ms  20.864 ms
 8  s-ipv6-b1-link.ipv6.telia.net  31.312 ms  31.113 ms  31.411 ms
 9  s-ipv6-b1-link.ipv6.telia.net  61.986 ms  61.988 ms  61.994 ms
```

(continues on next page)

[4] Until a few years ago, all hosts replied to *Echo request* ICMP messages. However, due to the security problems that have affected TCP/IP implementations, many of these implementations can now be configured to disable answering *Echo request* ICMP messages.

(continued from previous page)

```
10  2001:1890:61:8909::1  121.716 ms  121.779 ms  121.177 ms
11  2001:1890:61:9117::2  203.709 ms  203.305 ms  203.07 ms
12  mail.ietf.org  204.172 ms  203.755 ms  203.748 ms
```

Note: Rate limitation of ICMP messages

High-end hardware based routers use special purpose chips on their interfaces to forward IPv6 packets at line rate. These chips are optimized to process *correct* IP packets. They are not able to create ICMP messages at line rate. When such a chip receives an IP packet that triggers an ICMP message, it interrupts the main CPU of the router and the software running on this CPU processes the packet. This CPU is much slower than the hardware acceleration found on the interfaces [Gill2004]. It would be overloaded if it had to process IP packets at line rate and generate one ICMP message for each received packet. To protect this CPU, high-end routers limit the rate at which the hardware can interrupt the main CPU and thus the rate at which ICMP messages can be generated. This implies that not all erroneous IP packets cause the transmission of an ICMP message. The risk of overloading the main CPU of the router is also the reason why using hop-by-hop IPv6 options, including the router alert option is discouraged[5].

3.14.3 The IPv6 subnet

Until now, we have focused our discussion on the utilization of IPv6 on point-to-point links. Although there are point-to-point links in the Internet, mainly between routers and sometimes hosts, most of the hosts are attached to datalink layer networks such as Ethernet LANs or WiFi networks. These datalink layer networks play an important role in today's Internet and have heavily influenced the design of the operation of IPv6. To understand IPv6 and ICMPv6 completely, we first need to correctly understand the key principles behind these datalink layer technologies.

As explained earlier, devices attached to a Local Area Network can directly exchange frames among themselves. For this, each datalink layer interface on a device (host, router, ...) attached to such a network is identified by a MAC address. Each datalink layer interface includes a unique hardwired MAC address. MAC addresses are allocated to manufacturers in blocks and interface is numbered with a unique address. Thanks to the global unicity of the MAC addresses, the datalink layer service can assume that two hosts attached to a LAN have different addresses. Most LANs provide an unreliable connectionless service and a datalink layer frame has a header containing :

- the source MAC address
- the destination MAC address
- some multiplexing information to indicate the network layer protocol that is responsible for the payload of the frame

LANs also provide a broadcast and a multicast service. The broadcast service enables a device to send a single frame to all the devices attached to the same LAN. This is done by reserving a special broadcast MAC address (typically all bits of the address are set to one). To broadcast a frame, a device simply needs to send a frame whose destination is the broadcast address. All devices attached to the datalink network will receive the frame.

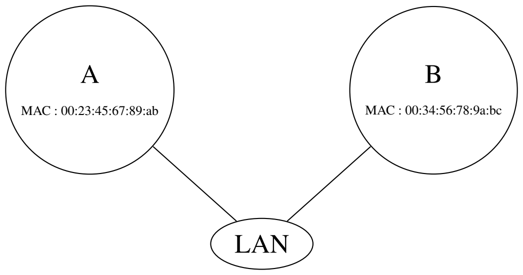
The broadcast service allows easily reaching all devices attached to a datalink layer network. It has been widely used to support IP version 4. A drawback of using the broadcast service to support a network layer protocol is that a broadcast frame that contains a network layer packet is always delivered to all devices attached to the datalink network, even if some of these devices do not support the network layer protocol. The multicast service is a useful alternative to the broadcast service. To understand its operation, it is important to understand how a datalink layer interface operates. In shared media LANs, all devices are attached to the same physical medium and all frames are delivered to all devices. When such a frame is received by a datalink layer interface, it compares the destination address with the MAC address of the device. If the two addresses match, or the destination address is the broadcast address, the frame is destined

[5] For a discussion of the issues with the router alert IP option, see http://tools.ietf.org/html/draft-rahman-rtg-router-alert-dangerous-00 or http://tools.ietf.org/html/draft-rahman-rtg-router-alert-considerations-03

to the device and its payload is delivered to the network layer protocol. The multicast service exploits this principle. A multicast address is a logical address. To receive frames destined to a multicast address in a shared media LAN, a device captures all frames having this multicast address as their destination. All IPv6 nodes are capable of capturing datalink layer frames destined to different multicast addresses.

Interactions between IPv6 and the datalink layer

IPv6 hosts and routers frequently interact with the datalink layer service. To understand the main interactions, it is useful to analyze all the packets that are exchanged when a simple network containing a few hosts and routers is built. Let us first start with a LAN containing two hosts[6].



Hosts `A` and `B` are attached to the same datalink layer network. They can thus exchange frames by using the MAC addresses shown in the figure above. To be able to use IPv6 to exchange packets, they need to have an IPv6 address. One possibility would be to manually configure an IPv6 address on each host. However, IPv6 provides a better solution thanks to the *link-local* IPv6 addresses. A *link-local* IPv6 address is an address that is composed by concatenating the `fe80:://64` prefix with the MAC address of the device. In the example above, host A would use IPv6 *link-local* address `fe80::0223:45FF:FE67:89ab` and host B `fe80::0234:56FF:FE78:9abc`. With these two IPv6 addresses, the hosts can exchange IPv6 packets.

Note: Converting MAC addresses in host identifiers

Appendix A of **RFC 4291** provides the algorithm used to convert a 48 bits MAC address into a 64 bits host identifier. This algorithm builds upon the structure of the MAC addresses. A MAC address is represented as shown in the figure below.

```
|0              1|1              3|3              4|
|0              5|6              1|2              7|
+----------------+----------------+----------------+
|cccccc0gcccccccc|ccccccccmmmmmmmm|mmmmmmmmmmmmmmmm|
+----------------+----------------+----------------+
```

Fig. 52: A MAC address

MAC addresses are allocated in blocks of 2^{20}. When a company registers for a block of MAC addresses, it receives an identifier. company identifier is then used to populated the *c* bits of the MAC addresses. The company can allocate all addresses in starting with this prefix and manages the *m* bits as it wishes.

[6] For simplicity, you assume that each datalink layer interface is assigned a 64 bits MAC address. As we will see later, today's datalink layer technologies mainly use 48 bits MAC addresses, but the smaller addresses can easily be converted into 64 bits addresses.

```
|0              1|1              3|3              4|4              6|
|0              5|6              1|2              7|8              3|
+----------------+----------------+----------------+----------------+
|cccccc1gcccccccc|cccccccc11111111|11111110mmmmmmmm|mmmmmmmmmmmmmmmm|
+----------------+----------------+----------------+----------------+
```

Fig. 53: A MAC address converted into a 64 bits host identifier

Inside a MAC address, the two bits indicated as *0* and *g* in the figure above play a special role. The first bit indicates whether the address is universal or local. The *g* bit indicates whether this is a multicast address or a unicast address. The MAC address can be converted into a 64 bits host identifier by flipping the value of the *0* bit and inserting `FFFE`, i.e. `1111111111111110` in binary, in the middle of the address as shown in the figure below. The *c*, *m* and *g* bits of the MAC address are not modified.

The next step is to connect the LAN to the Internet. For this, a router is attached to the LAN.

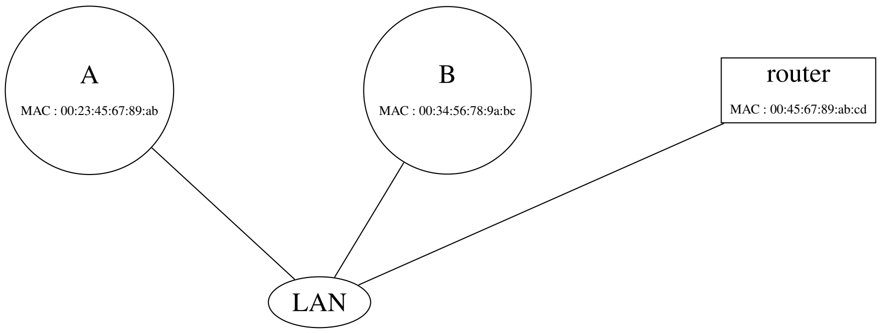


Fig. 54: A simple IPv6 network with one router

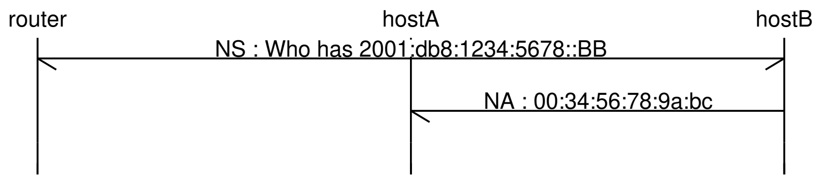
Assume that the LAN containing the two hosts and the router is assigned prefix `2001:db8:1234:5678/64`. A first solution to configure the IPv6 addresses in this network is to assign them manually. A possible assignment is :

- `2001:db8:1234:5678::1` is assigned to `router`
- `2001:db8:1234:5678::AA` is assigned to `hostA`
- `2001:db8:1234:5678::BB` is assigned to `hostB`

To be able to exchange IPv6 packets with `hostB`, `hostA` needs to know the MAC address of the interface of `hostB` on the LAN. This is the *address resolution* problem. In IPv6, this problem is solved by using the Neighbor Discovery Protocol (NDP). NDP is specified in **RFC 4861**. This protocol is part of ICMPv6 and uses the multicast datalink layer service.

NDP allows a host to discover the MAC address used by any other host attached to the same LAN. NDP operates in two steps. First, the querier sends a multicast ICMPv6 Neighbor Solicitation message that contains as parameter the queried IPv6 address. This multicast ICMPv6 NS is placed inside a multicast frame[7]. The queried node receives the frame, parses it and replies with a unicast ICMPv6 Neighbor Advertisement that provides its own IPv6 and MAC addresses. Upon reception of the Neighbor Advertisement message, the querier stores the mapping between the IPv6 and the MAC address inside its NDP table. This table is a data structure that maintains a cache of the recently received Neighbor Advertisement. Thanks to this cache, a host only needs to send a Neighbor Solicitation message for the first packet that it sends to a given host. After this initial packet, the NDP table can provide the mapping between the destination IPv6 address and the corresponding MAC address.

[7] **RFC 4291** and **RFC 4861** explain in more details how the IPv6 multicast address is determined from the target IPv6 unicast address. These details are outside the scope of this book, but may matter if you try to understand a packet trace.



The NS message can also be used to verify the reachability of a host in the local subnet. For this usage, NS messages can be sent in unicast since other nodes on the subnet do not need to process the message.

When an entry in the NDP table times out on a host, it may either be deleted or the host may try to validate it by sending the NS message again.

This is not the only usage of the Neighbor Solicitation and Neighbor Advertisement messages. They are also used to detect the utilization of duplicate addresses. In the network above, consider what happens when a new host is connected to the LAN. If this host is configured by mistake with the same address as `hostA` (i.e. `2001:db8:1234:5678::AA`), problems could occur. Indeed, if two hosts have the same IPv6 address on the LAN, but different MAC addresses, it will be difficult to correctly reach them. IPv6 anticipated this problem and includes a *Duplicate Address Detection* Algorithm (DAD). When an IPv6 address[8] is configured on a host, by any means, the host must verify the uniqueness of this address on the LAN. For this, it multicasts an ICMPv6 Neighbor Solicitation that queries the network for its newly configured address. The IPv6 source address of this NS is set to `::` (i.e. the reserved unassigned address) if the host does not already have an IPv6 address on this subnet). If the NS does not receive any answer, the new address is considered to be unique and can safely be used. Otherwise, the new address is refused and an error message should be returned to the system administrator or a new IPv6 address should be generated. The *Duplicate Address Detection* Algorithm can prevent various operational problems that are often difficult to debug.

Few users manually configure the IPv6 addresses on their hosts. They prefer to rely on protocols that can automatically configure their IPv6 addresses. IPv6 supports two such protocols : DHCPv6 and the Stateless Address Autoconfiguration (SLAAC).

The Stateless Address Autoconfiguration (SLAAC) mechanism defined in **RFC 4862** enables hosts to automatically configure their addresses without maintaining any state. When a host boots, it derives its identifier from its datalink layer address[9] as explained earlier and concatenates this 64 bits identifier to the *FE80::/64* prefix to obtain its link-local IPv6 address. It then multicasts a Neighbor Solicitation with its link-local address as a target to verify whether another host is using the same link-local address on this subnet. If it receives a Neighbor Advertisement indicating that the link-local address is used by another host, it generates another 64 bits identifier and sends again a Neighbor Solicitation. If there is no answer, the host considers its link-local address to be valid. This address will be used as the source address for all NDP messages sent on the subnet.

To automatically configure its global IPv6 address, the host must know the globally routable IPv6 prefix that is used on the local subnet. IPv6 routers regularly multicast ICMPv6 Router Advertisement messages that indicate the IPv6 prefix assigned to the subnet. The Router Advertisement message contains several interesting fields.

This message is sent from the link-local address of the router on the subnet. Its destination is the IPv6 multicast address that targets all IPv6 enabled hosts (i.e. `ff02::1`). The *Cur Hop Limit* field, if different from zero, allows specifying the default *Hop Limit* that hosts should use when sending IPv6 packets from this subnet. `64` is a frequently used value. The *M* and *O* bits are used to indicate that some information can be obtained from DHCPv6. The *Router Lifetime* parameter provides the expected lifetime (in seconds) of the sending router acting as a default router. This lifetime enables planning the replacement of a router by another one in the same subnet. The *Reachable Time* and the *Retrans Timer* parameter are used to configure the utilization of the NDP protocol on the hosts attached to the subnet.

Several options can be included in the Router Advertisement message. The simplest one is the MTU option that indicates the MTU to be used within the subnet. Thanks to this option, it is possible to ensure that all devices attached

[8] The DAD algorithm is also used with *link-local* addresses.

[9] Using a datalink layer address to derive a 64 bits identifier for each host raises privacy concerns as the host will always use the same identifier. Attackers could use this to track hosts on the Internet. An extension to the Stateless Address Configuration mechanism that does not raise privacy concerns is defined in **RFC 4941**. These privacy extensions allow a host to generate its 64 bits identifier randomly every time it attaches to a subnet. It then becomes impossible for an attacker to use the 64-bits identifier to track a host.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Code      |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Cur Hop Limit |M|O|  Reserved |       Router Lifetime         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Reachable Time                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Retrans Timer                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Options ...
+-+-+-+-+-+-+-+-+-+-+-+-
```

Fig. 55: Format of the ICMPv6 Router Advertisement message

to the same subnet use the same MTU. Otherwise, operational problems could occur. The *Prefix* option is more important. It provides information about the prefix(es) that is (are) advertised by the router on the subnet.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |    Length     | Prefix Length |L|A| Reserved1 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Valid Lifetime                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Preferred Lifetime                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Reserved2                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                            Prefix                             +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 56: The Prefix information option

The key information placed in this option are the prefix and its length. This allows the hosts attached to the subnet to automatically configure their own IPv6 address. The *Valid* and *Preferred Lifetimes* provide information about the expected lifetime of the prefixes. Associating some time validity to prefixes is a good practice from an operational viewpoint. There are some situations where the prefix assigned to a subnet needs to change without impacting the hosts attached to the subnet. This is often called the IPv6 renumbering problem in the literature **RFC 7010**. A very simple scenario is the following. An SME subscribes to one ISP. Its router is attached to another router of this ISP and advertises a prefix assigned by the ISP. The SME is composed of a single subnet and all its hosts rely on stateless address configuration. After a few years, the SME decides to change of network provider. It connects its router to the second ISP and receives a different prefix from this ISP. At this point, two prefixes are advertised on the SME's subnet. The old prefix can be advertised with a short lifetime to ensure that hosts will stop using it while the new one is advertised with a longer lifetime. After sometime, the router stops advertising the old prefix and the hosts stop using it. The old prefix can now be returned back to the first ISP. In larger networks, renumbering an IPv6 remains a difficult operational problem [LeB2009].

Upon reception of this message, the host can derive its global IPv6 address by concatenating its 64 bits identifier with the received prefix. It concludes the SLAAC by sending a Neighbor Solicitation message targeted at its global IPv6 address to ensure that no other host is using the same IPv6 address.

Note: Router Advertisements and Hop Limits

ICMPv6 Router Advertisements messages are regularly sent by routers. They are destined to all devices attached to the local subnet and no router should ever forward them to another subnet. Still, these messages are sent inside IPv6 packets whose *Hop Limit* is always set to `255`. Given that the packet should not be forwarded outside of the local subnet, the reader could expect instead a *Hop Limit* set to `1`. Using a *Hop Limit* set to `255` provides one important benefit from a security viewpoint and this hack has been adapted in several Internet protocols. When a host receives a *Router Advertisement* message, it expects that this message has been generated by a router attached to the same subnet. Using a *Hop Limit* of `255` provides a simple check for this. If the message was generated by an attacker outside the subnet, it would reach the subnet with a decremented *Hop Limit*. Checking that the *Hop Limit* is set to `255` is a simple[10] verification that the packet was generated on this particular subnet. **RFC 5082** provides other examples of protocols that use this hack and discuss its limitations.

Routers regularly send Router Advertisement messages. These messages are triggered by a timer that is often set at approximately 30 seconds. Usually, hosts wait for the arrival of a Router Advertisement message to configure their address. This implies that hosts could sometimes need to wait 30 seconds before being able to configure their address. If this delay is too long, a host can also send a *Router Solicitation* message. This message is sent towards the multicast address that corresponds to all IPv6 routers (i.e. `FF01::2`) and the default router will reply.

The last point that needs to be explained about ICMPv6 is the *Redirect* message. This message is used when there is more than one router on a subnet as shown in the figure below.

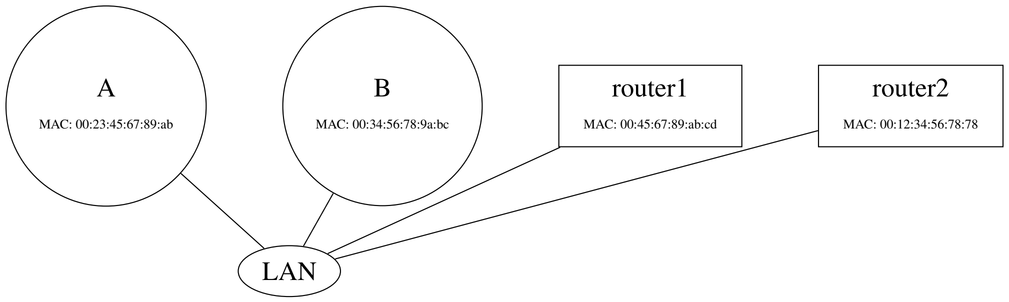


Fig. 57: A simple IPv6 network with two routers

In this network, `router1` is the default router for all hosts. The second router, `router2` provides connectivity to a specific IPv6 subnet, e.g. `2001:db8:abcd::/48`. These two routers attached to the same subnet can be used in different ways. First, it is possible to manually configure the routing tables on all hosts to add a route towards `2001:db8:abcd::/48` via `router2`. Unfortunately, forcing such manual configuration boils down all the benefits of using address auto-configuration in IPv6. The second approach is to automatically configure a default route via `router1` on all hosts. With such route, when a host needs to send a packet to any address within `2001:db8:abcd::/48`, it will send it to `router1`. `router1` would consult its routing table and find that the packet needs to be sent again on the subnet to reach `router2`. This is a waste of time. A better approach would be to enable the hosts to automatically learn the new route. This is possible thanks to the ICMPv6 *Redirect* message. When `router1` receives a packet that needs to be forwarded back on the same interface, it replies with a *Redirect* message that indicates that the packet should have been sent via `router2`. Upon reception of a *Redirect* message, the host updates it forwarding table to include a new transient entry for the destination reported in the message. A timeout is usually associated with this transient entry to automatically delete it after some time.

An alternative is the Dynamic Host Configuration Protocol (DHCP) defined in **RFC 2131** and **RFC 3315**. DHCP allows a host to automatically retrieve its assigned IPv6 address, but relies on server. A DHCP server is associated to each subnet[11]. Each DHCP server manages a pool of IPv6 addresses assigned to the subnet. When a host is first attached to the subnet, it sends a DHCP request message in a UDP segment (the DHCP server listens on port 67). As

[10] Using a *Hop Limit* of `255` prevents one family of attacks against ICMPv6, but other attacks still remain possible. A detailed discussion of the security issues with IPv6 is outside the scope of this book. It is possible to secure NDP by using the *Cryptographically Generated IPv6 Addresses* (CGA) defined in **RFC 3972**. The Secure Neighbor Discovery Protocol is defined in **RFC 3971**. A detailed discussion of the security of IPv6 may be found in [HV2008].

[11] In practice, there is usually one DHCP server per group of subnets and the routers capture on each subnet the DHCP messages and forward them to the DHCP server.

the host knows neither its IPv6 address nor the IPv6 address of the DHCP server, this UDP segment is sent inside a multicast packet target at the DHCP servers. The DHCP request may contain various options such as the name of the host, its datalink layer address, etc. The server captures the DHCP request and selects an unassigned address in its address pool. It then sends the assigned IPv6 address in a DHCP reply message which contains the datalink layer address of the host and additional information such as the subnet mask, the address of the default router or the address of the DNS resolver. The DHCP reply also specifies the lifetime of the address allocation. This forces the host to renew its address allocation once it expires. Thanks to the limited lease time, IP addresses are automatically returned to the pool of addresses when hosts are powered off.

Both SLAAC and DHCPv6 can be extended to provide additional information beyond the IPv6 prefix/address. For example, **RFC 6106** defines options for the ICMPv6 ND message that can carry the IPv6 address of the recursive DNS resolver and a list of default domain search suffixes. It is also possible to combine SLAAC with DHCPv6. **RFC 3736** defines a stateless variant of DHCPv6 that can be used to distribute DNS information while SLAAC is used to distribute the prefixes.

3.15 Routing in IP networks

In a large IP network such as the global Internet, routers need to exchange routing information. The Internet is an interconnection of networks, often called domains, that are under different responsibilities. As of this writing, the Internet is composed on more than 40,000 different domains and this number is still growing[1]. A domain can be a small enterprise that manages a few routers in a single building, a larger enterprise with a hundred routers at multiple locations, or a large Internet Service Provider managing thousands of routers. Two classes of routing protocols are used to allow these domains to efficiently exchange routing information.

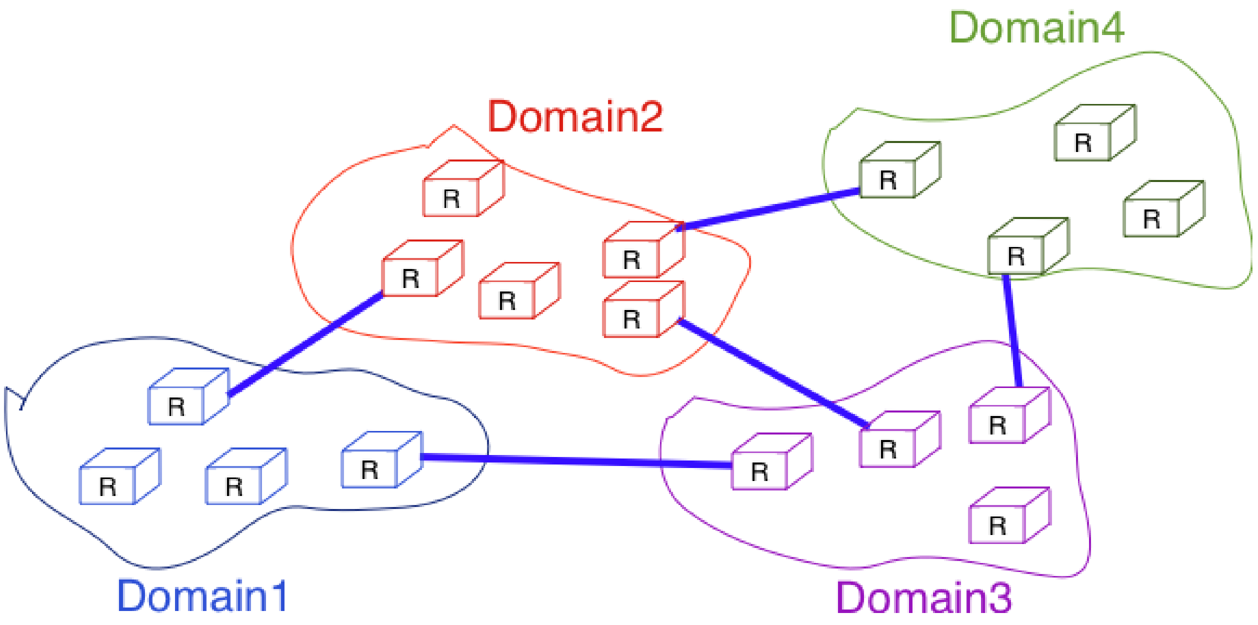


Fig. 58: Organisation of a small Internet

The first class of routing protocols are the *intradomain routing protocols* (sometimes also called the interior gateway protocols or *IGP*). An intradomain routing protocol is used by all routers inside a domain to exchange routing information about the destinations that are reachable inside the domain. There are several intradomain routing protocols. Some domains use *RIP*, which is a distance vector protocol. Other domains use link-state routing protocols such as *OSPF* or *IS-IS*. Finally, some domains use static routing or proprietary protocols such as *IGRP* or *EIGRP*.

[1] See http://bgp.potaroo.net/index-as.html for reports on the evolution of the number of Autonomous Systems over time.

These intradomain routing protocols usually have two objectives. First, they distribute routing information that corresponds to the shortest path between two routers in the domain. Second, they should allow the routers to quickly recover from link and router failures.

The second class of routing protocols are the *interdomain routing protocols* (sometimes also called the exterior gateway protocols or *EGP*). The objective of an interdomain routing protocol is to distribute routing information between domains. For scalability reasons, an interdomain routing protocol must distribute aggregated routing information and considers each domain as a black box.

A very important difference between intradomain and interdomain routing are the *routing policies* that are used by each domain. Inside a single domain, all routers are considered equal, and when several routes are available to reach a given destination prefix, the best route is selected based on technical criteria such as the route with the shortest delay, the route with the minimum number of hops or the route with the highest bandwidth.

When we consider the interconnection of domains that are managed by different organizations, this is no longer true. Each domain implements its own routing policy. A routing policy is composed of three elements : an *import filter* that specifies which routes can be accepted by a domain, an *export filter* that specifies which routes can be advertised by a domain and a ranking algorithm that selects the best route when a domain knows several routes towards the same destination prefix. As we will see later, another important difference is that the objective of the interdomain routing protocol is to find the *cheapest* route towards each destination. There is only one interdomain routing protocol : *BGP*.

3.16 Intradomain routing

In this section, we briefly describe the key features of the two main intradomain unicast routing protocols : RIP and OSPF. The basic principles of distance vector and link-state routing have been presented earlier.

3.16.1 RIP

The Routing Information Protocol (RIP) is the simplest routing protocol that was standardized for the TCP/IP protocol suite. RIP is defined in **RFC 2453**. Additional information about RIP may be found in [Malkin1999].

RIP routers periodically exchange RIP messages. The format of these messages is shown below. A RIP message is sent inside a UDP segment whose destination port is set to *521*. A RIP message contains several fields. The *command* field indicates whether the RIP message is a request or a response. When a router boots, its routing table is empty and it cannot forward any packet. To speedup the discovery of the network, it can send a request message to the RIP IPv6 multicast address, `FF02::9`. All RIP routers listen to this multicast address and any router attached to the subnet will reply by sending its own routing table as a sequence of RIP messages. In steady state, routers multicast one of more RIP response messages every 30 seconds. These messages contain the distance vectors that summarize the router's routing table. The current version of RIP is version 2 defined in **RFC 2453** for IPv4 and **RFC 2080** for IPv6.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  command (1)  |  version (1)  |       must be zero (2)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                 Route Table Entry 1 (20)                      ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                         ...                                   ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                 Route Table Entry N (20)                      ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 59: The RIP message format

Each RIP message contains a set of route entries. Each route entry is encoded as a 20 bytes field whose format is shown below. RIP was initially designed to be suitable for different network layer protocols. Some implementations of RIP were used in XNS or IPX networks **RFC 2453**. The format of the route entries used by **RFC 2080** is shown below. *Prefix length* is the length of the subnet identifier in bits and the *metric* is encoded as one byte. The maximum metric supported by RIP is *15*.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                        IPv6 prefix (16)                       ~
|                                                               |
+---------------------------------------------------------------+
|         route tag (2)         | prefix len (1)|  metric (1)   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 60: Format of the RIP IPv6 route entries

Note: A note on timers

The first RIP implementations sent their distance vector exactly every 30 seconds. This worked well in most networks, but some researchers noticed that routers were sometimes overloaded because they were processing too many distance vectors at the same time [FJ1994]. They collected packet traces in these networks and found that after some time the routers' timers became synchronized, i.e. almost all routers were sending their distance vectors at almost the same time. This synchronization of the transmission times of the distance vectors caused an overload on the routers' CPU but also increased the convergence time of the protocol in some cases. This was mainly due to the fact that all routers set their timers to the same expiration time after having processed the received distance vectors. Sally Floyd and Van Jacobson proposed in [FJ1994] a simple solution to solve this synchronization problem. Instead of advertising their distance vector exactly after 30 seconds, a router should send its next distance vector after a delay chosen randomly in the [15,45] interval **RFC 2080**. This randomization of the delays prevents the synchronization that occurs with a fixed delay and is now a recommended practice for protocol designers.

3.16.2 OSPF

Link-state routing protocols are used in IP networks. Open Shortest Path First (OSPF), defined in **RFC 2328**, is the link state routing protocol that has been standardized by the IETF. The last version of OSPF, which supports IPv6, is defined in **RFC 5340**. OSPF is frequently used in enterprise networks and in some ISP networks. However, ISP networks often use the IS-IS link-state routing protocol [ISO10589] , which was developed for the ISO CLNP protocol but was adapted to be used in IP **RFC 1195** networks before the finalization of the standardization of OSPF. A detailed analysis of ISIS and OSPF may be found in [BMO2006] and [Perlman2000]. Additional information about OSPF may be found in [Moy1998].

Compared to the basics of link-state routing protocols that we discussed in section *Link state routing*, there are some particularities of OSPF that are worth discussing. First, in a large network, flooding the information about all routers and links to thousands of routers or more may be costly as each router needs to store all the information about the entire network. A better approach would be to introduce hierarchical routing. Hierarchical routing divides the network into regions. All the routers inside a region have detailed information about the topology of the region but only learn aggregated information about the topology of the other regions and their interconnections. OSPF supports a restricted variant of hierarchical routing. In OSPF's terminology, a region is called an *area*.

OSPF imposes restrictions on how a network can be divided into areas. An area is a set of routers and links that are grouped together. Usually, the topology of an area is chosen so that a packet sent by one router inside the area can reach any other router in the area without leaving the area[2] . An OSPF area contains two types of routers **RFC 2328**:

[2] OSPF can support *virtual links* to connect routers together that belong to the same area but are not directly connected. However, this goes beyond this introduction to OSPF.

- Internal router : A router whose directly connected networks belong to the area
- Area border routers : A router that is attached to several areas.

For example, the network shown in the figure below has been divided into three areas : *area 0*, containing routers *RA*, *RB*, *RC* and *RD*, *area 1*, containing routers *R1*, *R3*, *R4*, *R5* and *RA*, and *area 2* containing *R7*, *R8*, *R9*, *R10*, *RB* and *RC*. OSPF areas are identified by a 32 bit integer, which is sometimes represented as an IP address. Among the OSPF areas, *area 0*, also called the *backbone area*, has a special role. The backbone area groups all the area border routers (routers *RA*, *RB* and *RC* in the figure below) and the routers that are directly connected to the backbone routers but do not belong to another area (router *RD* in the figure below). An important restriction imposed by OSPF is that the path between two routers that belong to two different areas (e.g. *R1* and *R8* in the figure below) must pass through the backbone area.

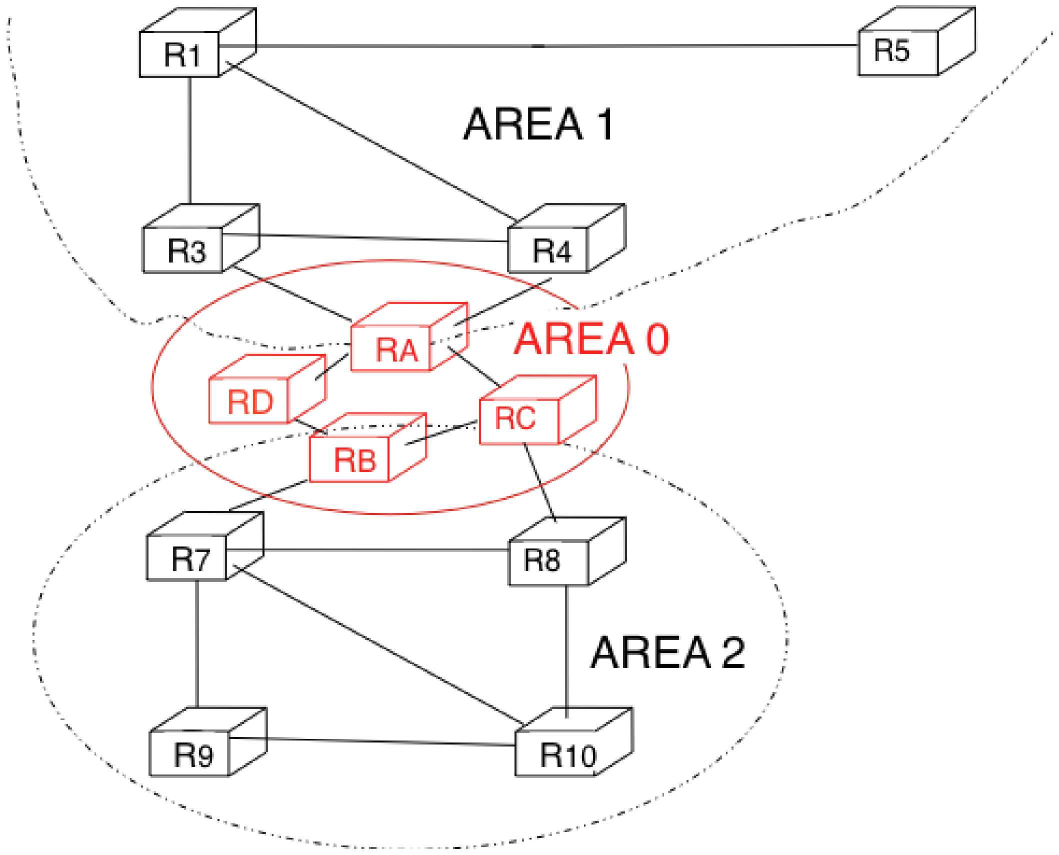


Fig. 61: OSPF areas

Inside each non-backbone area, routers distribute the topology of the area by exchanging link state packets with the other routers in the area. The internal routers do not know the topology of other areas, but each router knows how to reach the backbone area. Inside an area, the routers only exchange link-state packets for all destinations that are reachable inside the area. In OSPF, the inter-area routing is done by exchanging distance vectors. This is illustrated by the network topology shown below.

Let us first consider OSPF routing inside *area 2*. All routers in the area learn a route towards *2001:db8:1234::/48* and *2001:db8:5678::/48*. The two area border routers, *RB* and *RC*, create network summary advertisements. Assuming that all links have a unit link metric, these would be:

- *RB* advertises *2001:db8:1234::/48* at a distance of *2* and *2001:db8:5678::/48* at a distance of *3*
- *RC* advertises *2001:db8:5678::/48* at a distance of *2* and *2001:db8:1234::/48* at a distance of *3*

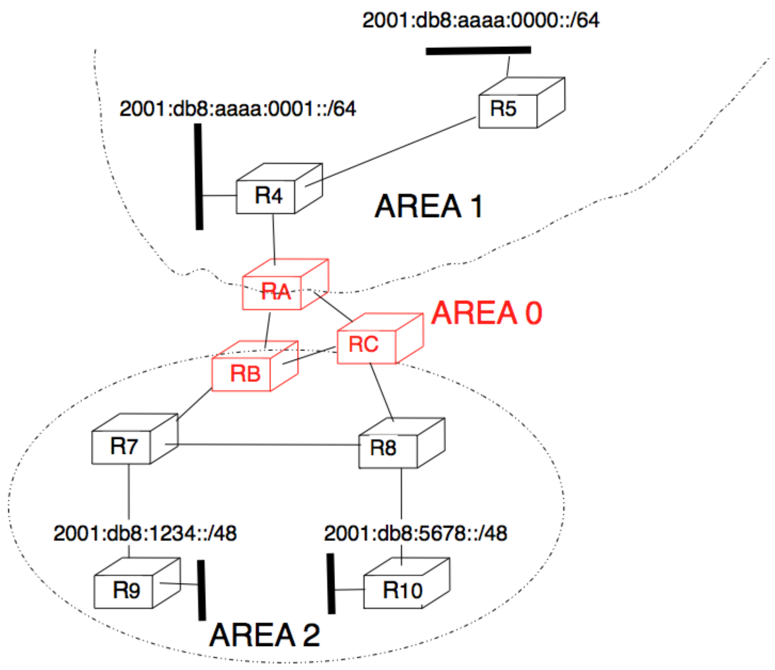


Fig. 62: Hierarchical routing with OSPF

These summary advertisements are flooded through the backbone area attached to routers *RB* and *RC*. In its routing table, router *RA* selects the summary advertised by *RB* to reach *2001:db8:1234::/48* and the summary advertised by *RC* to reach *2001:db8:5678::/48*. Inside *area 1*, router *RA* advertises a summary indicating that *2001:db8:1234::/48* and *2001:db8:5678::/48* are both at a distance of *3* from itself.

On the other hand, consider the prefixes *2001:db8:aaaa:0000::/64* and *2001:db8:aaaa:0001::/64* that are inside *area 1*. Router *RA* is the only area border router that is attached to this area. This router can create two different network summary advertisements :

- *2001:db8:aaaa:0000::/64* at a distance of *1* and *2001:db8:aaaa:0001::/64* at a distance of *2* from *RA*
- *2001:db8:aaaa:0000::/63* at a distance of *2* from *RA*

The first summary advertisement provides precise information about the distance used to reach each prefix. However, all routers in the network have to maintain a route towards *2001:db8:aaaa:0000::/64* and a route towards *2001:db8:aaaa:0001::/64* that are both via router *RA*. The second advertisement would improve the scalability of OSPF by reducing the number of routes that are advertised across area boundaries. However, in practice this requires manual configuration on the border routers.

The second OSPF particularity that is worth discussing is the support of Local Area Networks (LAN). As shown in the example below, several routers may be attached to the same LAN. A first solution to support such a LAN with a

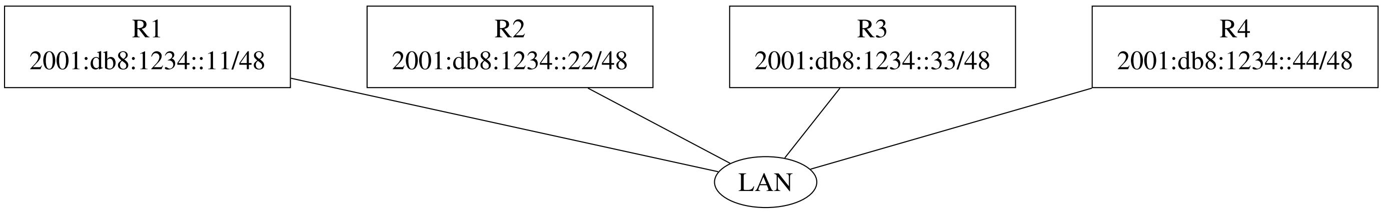


Fig. 63: A LAN with routers

link-state routing protocol would be to consider that a LAN is equivalent to a full-mesh of point-to-point links as if each router can directly reach any other router on the LAN. However, this approach has two important drawbacks :

1. Each router must exchange HELLOs and link state packets with all the other routers on the LAN. This increases the number of OSPF packets that are sent and processed by each router.
2. Remote routers, when looking at the topology distributed by OSPF, consider that there is a full-mesh of links between all the LAN routers. Such a full-mesh implies a lot of redundancy in case of failure, while in practice the entire LAN may completely fail. In case of a failure of the entire LAN, all routers need to detect the failures and flood link state packets before the LAN is completely removed from the OSPF topology by remote routers.

To better represent LANs and reduce the number of OSPF packets that are exchanged, OSPF handles LAN differently. When OSPF routers boot on a LAN, they elect[3] one of them as the *Designated Router (DR)* **RFC 2328**. The *DR* router *represents* the local area network, and advertises the LAN's subnet. Furthermore, LAN routers only exchange HELLO packets with the *DR*. Thanks to the utilization of a *DR*, the topology of the LAN appears as a set of point-to-point links connected to the *DR* router.

Note: How to quickly detect a link failure ?

Network operators expect an OSPF network to be able to quickly recover from link or router failures [VPD2004]. In an OSPF network, the recovery after a failure is performed in three steps [FFEB2005] :

- the routers that are adjacent to the failure detect it quickly. The default solution is to rely on the regular exchange of HELLO packets. However, the interval between successive HELLOs is often set to 10 seconds... Setting the HELLO timer down to a few milliseconds is difficult as HELLO packets are created and processed by the main

[3] The OSPF Designated Router election procedure is defined in **RFC 2328**. Each router can be configured with a router priority that influences the election process since the router with the highest priority is preferred when an election is run.

CPU of the routers and these routers cannot easily generate and process a HELLO packet every millisecond on each of their interfaces. A better solution is to use a dedicated failure detection protocol such as the Bidirectional Forwarding Detection (BFD) protocol defined in [KW2009] that can be implemented directly on the router interfaces. Another solution to be able to detect the failure is to instrument the physical and the datalink layer so that they can interrupt the router when a link fails. Unfortunately, such a solution cannot be used on all types of physical and datalink layers.

- the routers that have detected the failure flood their updated link state packets in the network
- all routers update their routing table

A last, but operationally important, point needs to be discussed about intradomain routing protocols such as OSPF and IS-IS. Intradomain routing protocols always select the shortest path for each destination. In practice, there are often several equal paths towards the same destination. When a router computes several equal cost paths towards one destination, it can use these paths in different ways.

A first approach is to select one of the equal cost paths (e.g. the first or the last path found by the SPF computation) and install it in the forwarding table. In this case, only one path is used to reach each destination.

A second approach is to install all equal cost paths[4] in the forwarding table and load-balance the packets on the different paths. Consider the case where a router has *N* different outgoing interfaces to reach destination *d*. A first possibility to load-balance the traffic among these interfaces is to use *round-robin*. *Round-robin* allows equally balancing the packets among the *N* outgoing interfaces. This equal load-balancing is important in practice because it allows better spreading the load throughout the network. However, few networks use this *round-robin* strategy to load-balance traffic on routers. The main drawback of *round-robin* is that packets that belong to the same flow (e.g. TCP connection) may be forwarded over different paths. If packets belonging to the same TCP connection are sent over different paths, they will probably experience different delays and arrive out-of-sequence at their destination. When a TCP receiver detects out-of-order segments, it sends duplicate acknowledgments that may cause the sender to initiate a fast retransmission and enter congestion avoidance. Thus, out-of-order segments may lead to lower TCP performance. This is annoying for a load-balancing technique whose objective is to improve the network performance by spreading the load.

To efficiently spread the load over different paths, routers need to implement *per-flow* load-balancing. This implies that they must forward all the packets that belong to the same flow on the same path. Since a TCP connection is always identified by the four-tuple (source and destination addresses, source and destination ports), one possibility would be to select an outgoing interface upon arrival of the first packet of the flow and store this decision in the router's memory. Unfortunately, such a solution does not scale since the required memory grows with the number of TCP connections that pass through the router.

Fortunately, it is possible to perform *per-flow* load balancing without maintaining any state on the router. Most routers today use hash functions for this purpose **RFC 2991**. When a packet arrives, the router extracts the Next Header information and the four-tuple from the packet and computes :

$$hash(NextHeader, IP_{src}, IP_{dst}, Port_{src}, Port_{dst}) \pmod N$$

In this formula, N is the number of outgoing interfaces on the equal cost paths towards the packet's destination. Various hash functions are possible, including CRC, checksum or MD5 **RFC 2991**. Since the hash function is computed over the four-tuple, the same hash value will be computed for all packets belonging to the same flow. This prevents reordering due to load balancing inside the network. Most routers support this kind of load-balancing today [ACO+2006].

[4] In some networks, there are several dozens of paths towards a given destination. Some routers, due to hardware limitations, cannot install more than 8 or 16 paths in their forwarding table. In this case, a subset of the computed paths is installed in the forwarding table.

3.17 Interdomain routing

As explained earlier, the Internet is composed of more than 45,000 different networks[1] called *domains*. Each domain is composed of a group of routers and hosts that are managed by the same organization. Example domains include belnet, sprint, level3, geant, abilene, cisco or google . . .

Each domain contains a set of routers. From a routing point of view, these domains can be divided into two classes : the *transit* and the *stub* domains. A *stub* domain sends and receives packets whose source or destination are one of its own hosts. A *transit* domain is a domain that provides a transit service for other domains, i.e. the routers in this domain forward packets whose source and destination do not belong to the transit domain. As of this writing, about 85% of the domains in the Internet are stub domains[2]. A *stub* domain that is connected to a single transit domain is called a *single-homed stub* (e.g., *S1* in the figure below.). A *multihomed stub* is a *stub* domain connected to two or more transit providers (e.g., *S2*).

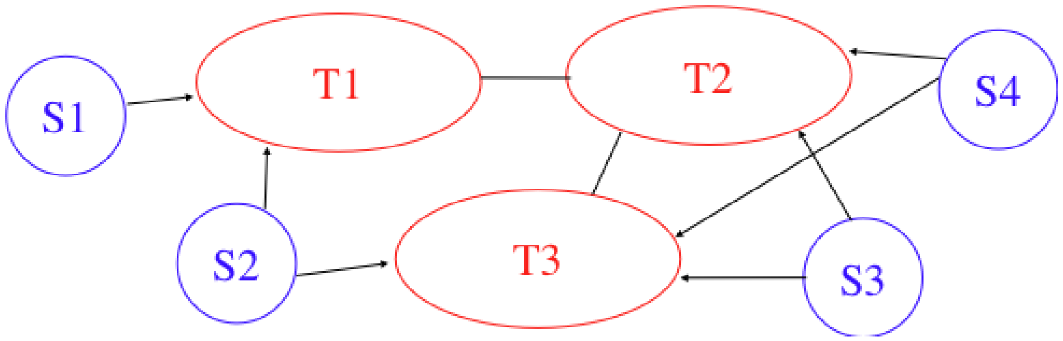


Fig. 64: Transit and stub domains

The stub domains can be further classified by considering whether they mainly send or receive packets. An *access-rich* stub domain is a domain that contains hosts that mainly receive packets. Typical examples include small ADSL- or cable modem-based Internet Service Providers or enterprise networks. On the other hand, a *content-rich* stub domain is a domain that mainly produces packets. Examples of *content-rich* stub domains include google, yahoo, microsoft, facebook or content distribution networks such as akamai or limelight. For the last few years, we have seen a rapid growth of these *content-rich* stub domains. Recent measurements [ATLAS2009] indicate that a growing fraction of all the packets exchanged on the Internet are produced in the data centers managed by these content providers.

Domains need to be interconnected to allow a host inside a domain to exchange IP packets with hosts located in other domains. From a physical perspective, domains can be interconnected in two different ways. The first solution is to directly connect a router belonging to the first domain with a router inside the second domain. Such links between domains are called private interdomain links or *private peering links*. In practice, for redundancy or performance reasons, distinct physical links are usually established between different routers in the two domains that are interconnected.

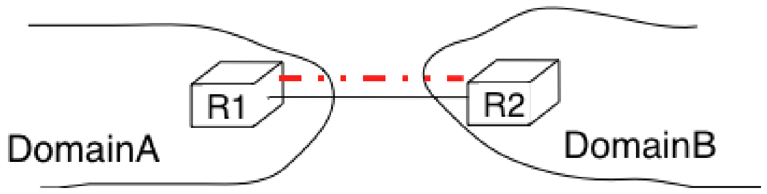


Fig. 65: Interconnection of two domains via a private peering link

[1] An analysis of the evolution of the number of domains on the global Internet during the last ten years may be found in http://www.potaroo.net/tools/asn32/.

[2] Several web sites collect and analyze data about the evolution of BGP in the global Internet. http://bgp.potaroo.net provides lots of statistics and analyzes that are updated daily.

Such *private peering links* are useful when, for example, an enterprise or university network needs to be connected to its Internet Service Provider. However, some domains are connected to hundreds of other domains[3] . For some of these domains, using only private peering links would be too costly. A better solution to allow many domains to cheaply interconnect are the *Internet eXchange Points* (*IXP*). An *IXP* is usually some space in a data center that hosts routers belonging to different domains. A domain willing to exchange packets with other domains present at the *IXP* installs one of its routers on the *IXP* and connects it to other routers inside its own network. The IXP contains a Local Area Network to which all the participating routers are connected. When two domains that are present at the IXP wish[4] to exchange packets, they simply use the Local Area Network. IXPs are very popular in Europe and many Internet Service Providers and Content providers are present in these IXPs.

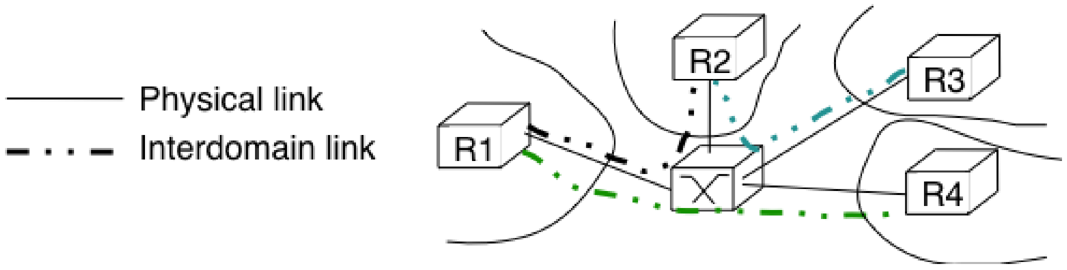


Fig. 66: Interconnection of two domains at an Internet eXchange Point

In the early days of the Internet, domains would simply exchange all the routes they know to allow a host inside one domain to reach any host in the global Internet. However, in today's highly commercial Internet, this is no longer true as interdomain routing mainly needs to take into account the economical relationships between the domains. Furthermore, while intradomain routing usually prefers some routes over others based on their technical merits (e.g. prefer route with the minimum number of hops, prefer route with the minimum delay, prefer high bandwidth routes over low bandwidth ones, etc) interdomain routing mainly deals with economical issues. For interdomain routing, the cost of using a route is often more important than the quality of the route measured by its delay or bandwidth.

There are different types of economical relationships that can exist between domains. Interdomain routing converts these relationships into peering relationships between domains that are connected via peering links.

The first category of peering relationship is the *customer->provider* relationship. Such a relationship is used when a customer domain pays an Internet Service Provider to be able to exchange packets with the global Internet over an interdomain link. A similar relationship is used when a small Internet Service Provider pays a larger Internet Service Provider to exchange packets with the global Internet.

To understand the *customer->provider* relationship, let us consider the simple internetwork shown in the figure above. In this internetwork, *AS7* is a stub domain that is connected to one provider : *AS4*. The contract between *AS4* and *AS7* allows a host inside *AS7* to exchange packets with any host in the internetwork. To enable this exchange of packets, *AS7* must know a route towards any domain and all the domains of the internetwork must know a route via *AS4* that allows them to reach hosts inside *AS7*. From a routing perspective, the commercial contract between *AS7* and *AS4* leads to the following routes being exchanged :

- over a *customer->provider* relationship, the *customer* domain advertises to its *provider* its own prefixes and all the routes that it has learned from its own customers.
- over a *provider->customer* relationship, the *provider* advertises all the routes that it knows to its *customer*.

The second rule ensures that the customer domain receives a route towards all destinations that are reachable via its provider. The first rule allows the prefixes of the customer domain to be distributed throughout the Internet.

Coming back to the figure above, *AS4* advertises to its two providers *AS1* and *AS2* its own prefixes and the routes learned from its customer, *AS7*. On the other hand, *AS4* advertises to *AS7* all the routes that it knows.

[3] See http://as-rank.caida.org/ for an analysis of the interconnections between domains based on measurements collected in the global Internet.

[4] Two routers that are attached to the same IXP exchange packets only when the owners of their domains have an economical incentive to exchange packets on this IXP. Usually, a router on an IXP is only able to exchange packets with a small fraction of the routers that are present on the same IXP.

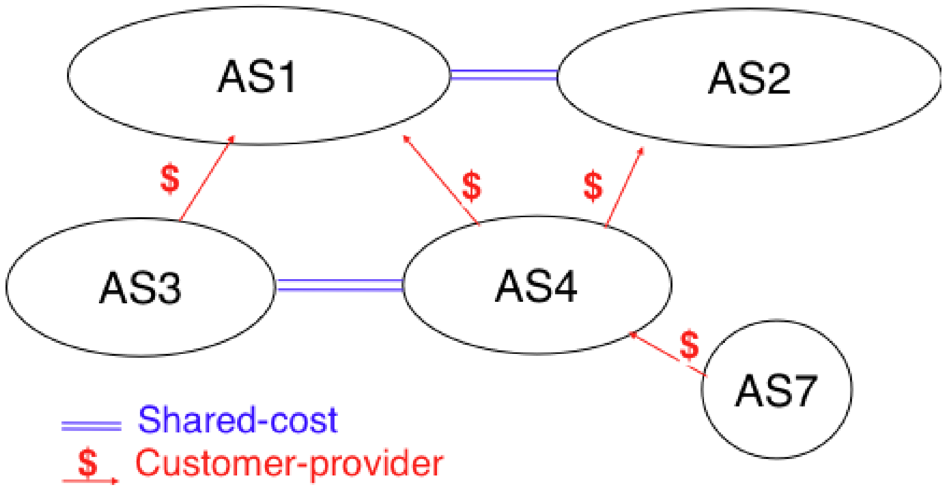


Fig. 67: A simple Internet with peering relationships

The second type of peering relationship is the *shared-cost* peering relationship. Such a relationship usually does not involve a payment from one domain to the other in contrast with the *customer->provider* relationship. A *shared-cost* peering relationship is usually established between domains having a similar size and geographic coverage. For example, consider the figure above. If *AS3* and *AS4* exchange many packets via *AS1*, they both need to pay *AS1*. A cheaper alternative for *AS3* and *AS4* would be to establish a *shared-cost* peering. Such a peering can be established at IXPs where both *AS3* and *AS4* are present or by using private peering links. This *shared-cost* peering should be used to exchange packets between hosts inside *AS3* and hosts inside *AS4*. However, *AS3* does not want to receive on the *AS3-AS4 shared-cost* peering links packets whose destination belongs to *AS1* as *AS3* would have to pay to send these packets to *AS1*.

From a routing perspective, over a *shared-cost* peering relationship a domain only advertises its internal routes/prefixes and the routes that it has learned from its customers. This restriction ensures that only packets destined to the local domain or one of its customers is received over the *shared-cost* peering relationship. This implies that the routes that have been learned from a provider or from another *shared-cost* peer is not advertised over a *shared-cost* peering relationship. This is motivated by economical reasons. If a domain were to advertise the routes that it learned from a provider over a *shared-cost* peering relationship that does not bring revenue, it would have allowed its *shared-cost* peer to use the link with its provider without any payment. If a domain were to advertise the routes it learned over a *shared cost* peering over another *shared-cost* peering relationship, it would have allowed these *shared-cost* peers to use its own network (which may span one or more continents) freely to exchange packets.

Finally, the last type of peering relationship is the *sibling*. Such a relationship is used when two domains exchange all their routes in both directions. In practice, such a relationship is only used between domains that belong to the same company.

These different types of relationships are implemented in the *interdomain routing policies* defined by each domain. The *interdomain routing policy* of a domain is composed of three main parts :

- the *import filter* that specifies, for each peering relationship, the routes that can be accepted from the neighboring domain (the non-acceptable routes are ignored and the domain never uses them to forward packets)
- the *export filter* that specifies, for each peering relationship, the routes that can be advertised to the neighboring domain
- the *ranking* algorithm that is used to select the best route among all the routes that the domain has received towards the same destination prefix

A domain's import and export filters can be defined by using the Route Policy Specification Language (RPSL) specified in **RFC 2622** [GAVE1999] . Some Internet Service Providers, notably in Europe, use RPSL to document[5] their import and export policies. Several tools help to easily convert a RPSL policy into router commands.

The figure below provides a simple example of import and export filters for two domains in a simple internetwork. In RPSL, the keyword *ANY* is used to replace any route from any domain. It is typically used by a provider to indicate that it announces all its routes to a customer over a *provider->customer* relationship. This is the case for *AS4*'s export policy. The example below clearly shows the difference between a *provider->customer* and a *shared-cost* peering relationship. *AS4*'s export filter indicates that it announces only its internal routes (*AS4*) and the routes learned from its clients (*AS7*) over its *shared-cost* peering with *AS3*, while it advertises all the routes that it uses (including the routes learned from *AS3*) to *AS7*.

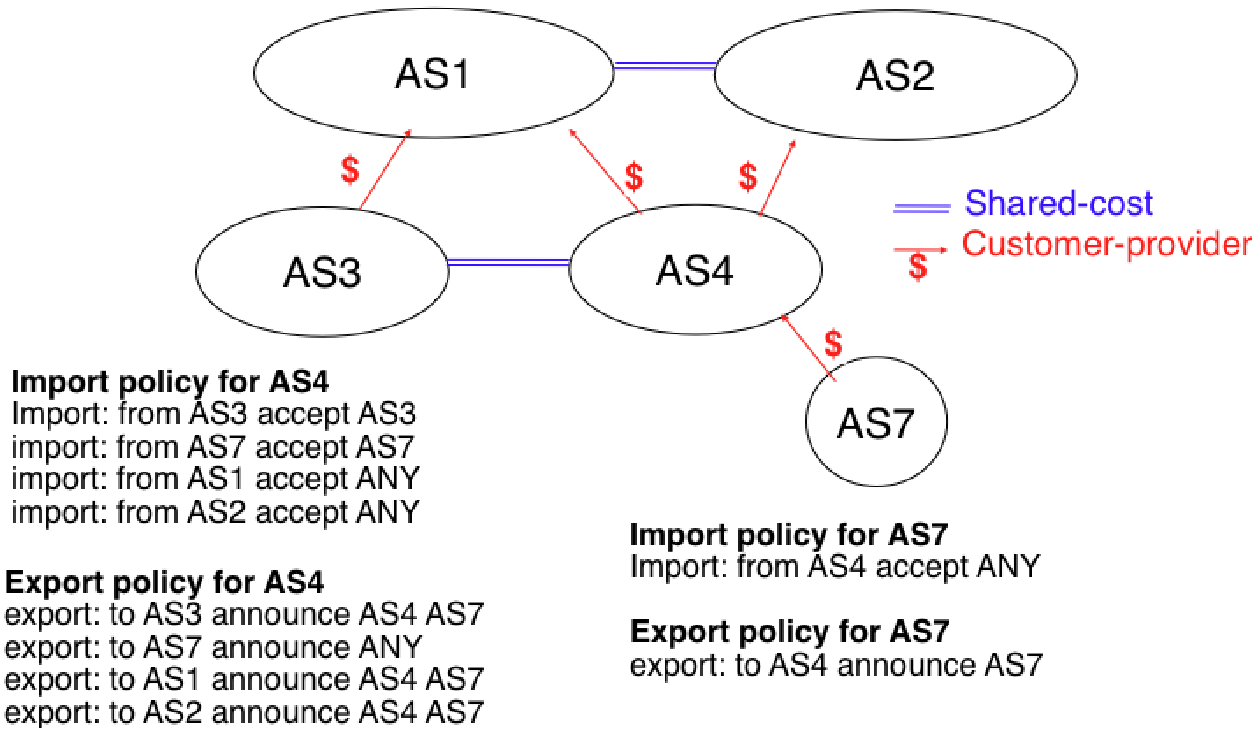

AS1
AS2
AS3
AS4
AS7
$
$
$
$
Shared-cost
Customer-provider
$
Import policy for AS4
Import: from AS3 accept AS3
import: from AS7 accept AS7
import: from AS1 accept ANY
import: from AS2 accept ANY
Export policy for AS4
export: to AS3 announce AS4 AS7
export: to AS7 announce ANY
export: to AS1 announce AS4 AS7
export: to AS2 announce AS4 AS7
Import policy for AS7
Import: from AS4 accept ANY
Export policy for AS7
export: to AS4 announce AS7

Fig. 68: Import and export policies

3.17.1 The Border Gateway Protocol

The Internet uses a single interdomain routing protocol : the Border Gateway Protocol (BGP). The current version of BGP is defined in **RFC 4271**. BGP differs from the intradomain routing protocols that we have already discussed in several ways. First, BGP is a *path-vector* protocol. When a BGP router advertises a route towards a prefix, it announces the IP prefix and the interdomain path used to reach this prefix. From BGP's point of view, each domain is identified by a unique *Autonomous System* (AS) number[6] and the interdomain path contains the AS numbers of the transit domains that are used to reach the associated prefix. This interdomain path is called the *AS Path*. Thanks to these AS-Paths, BGP does not suffer from the count-to-infinity problems that affect distance vector routing protocols. Furthermore, the AS-Path can be used to implement some routing policies. Another difference between BGP and the intradomain routing protocols is that a BGP router does not send the entire contents of its routing table to its neighbors regularly. Given the size of the global Internet, routers would be overloaded by the number of BGP messages that they would need to process. BGP uses incremental updates, i.e. it only announces the routes that have changed to its neighbors.

[5] See ftp://ftp.ripe.net/ripe/dbase for the RIPE database that contains the import and export policies of many European ISPs.

[6] In this text, we consider Autonomous System and domain as synonyms. In practice, a domain may be divided into several Autonomous Systems, but we ignore this detail.

The figure below shows a simple example of the BGP routes that are exchanged between domains. In this example, prefix *2001:db8:cafe::/48* is announced by *AS1*. *AS1* advertises a BGP route towards this prefix to *AS2*. The AS-Path of this route indicates that *AS1* is the originator of the prefix. When *AS4* receives the BGP route from *AS1*, it re-announces it to *AS2* and adds its AS number to the AS-Path. *AS2* has learned two routes towards prefix *2001:db8:cafe::/48*. It compares the two routes and prefers the route learned from *AS4* based on its own ranking algorithm. *AS2* advertises to *AS5* a route towards *2001:db8:cafe::/48* with its AS-Path set to *AS2:AS4:AS1*. Thanks to the AS-Path, *AS5* knows that if it sends a packet towards *2001:db8:cafe::/48* the packet first passes through *AS2*, then through *AS4* before reaching its destination inside *AS1*.

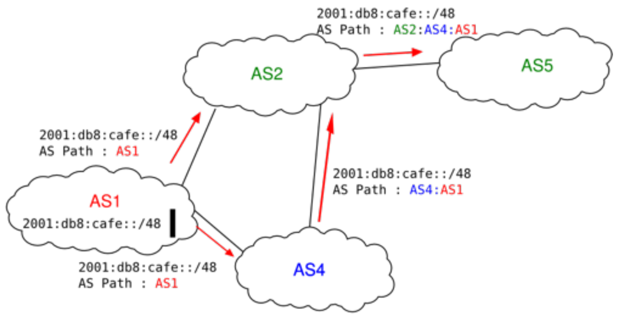


Fig. 69: Simple exchange of BGP routes

BGP routers exchange routes over BGP sessions. A BGP session is established between two routers belonging to two different domains that are directly connected. As explained earlier, the physical connection between the two routers can be implemented as a private peering link or over an Internet eXchange Point. A BGP session between two adjacent routers runs above a TCP connection (the default BGP port is 179). In contrast with intradomain routing protocols that exchange IP packets or UDP segments, BGP runs above TCP because TCP ensures a reliable delivery of the BGP messages sent by each router without forcing the routers to implement acknowledgments, checksums, etc. Furthermore, the two routers consider the peering link to be up as long as the BGP session and the underlying TCP connection remain up[7]. The two endpoints of a BGP session are called *BGP peers*.

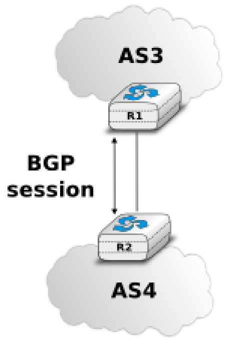


Fig. 70: A BGP peering session between two directly connected routers

In practice, to establish a BGP session between routers *R1* and *R2* in the figure above, the network administrator of *AS3* must first configure on *R1* the IP address of *R2* on the *R1-R2* link and the AS number of *R2*. Router *R1* then regularly tries to establish the BGP session with *R2*. *R2* only agrees to establish the BGP session with *R1* once it has been configured with the IP address of *R1* and its AS number. For security reasons, a router never establishes a BGP session that has not been manually configured on the router.

The BGP protocol **RFC 4271** defines several types of messages that can be exchanged over a BGP session :

[7] The BGP sessions and the underlying TCP connection are typically established by the routers when they boot based on information found in their configuration. The BGP sessions are rarely released, except if the corresponding peering link fails or one of the endpoints crashes or needs to be rebooted.

- *OPEN* : this message is sent as soon as the TCP connection between the two routers has been established. It initializes the BGP session and allows the negotiation of some options. Details about this message may be found in **RFC 4271**.
- *NOTIFICATION* : this message is used to terminate a BGP session, usually because an error has been detected by the BGP peer. A router that sends or receives a *NOTIFICATION* message immediately shutdowns the corresponding BGP session.
- *UPDATE*: this message is used to advertise new or modified routes or to withdraw previously advertised routes.
- *KEEPALIVE* : this message is used to ensure a regular exchange of messages on the BGP session, even when no route changes. When a BGP router has not sent an *UPDATE* message during the last 30 seconds, it shall send a *KEEPALIVE* message to confirm to the other peer that it is still up. If a peer does not receive any BGP message during a period of 90 seconds[8], the BGP session is considered to be down and all the routes learned over this session are withdrawn.

As explained earlier, BGP relies on incremental updates. This implies that when a BGP session starts, each router first sends BGP *UPDATE* messages to advertise to the other peer all the exportable routes that it knows. Once all these routes have been advertised, the BGP router only sends BGP *UPDATE* messages about a prefix if the route is new, one of its attributes has changed or the route became unreachable and must be withdrawn. The BGP *UPDATE* message allows BGP routers to efficiently exchange such information while minimizing the number of bytes exchanged. Each *UPDATE* message contains :

- a list of IP prefixes that are withdrawn
- a list of IP prefixes that are (re-)advertised
- the set of attributes (e.g. AS-Path) associated to the advertised prefixes

In the remainder of this chapter, and although all routing information is exchanged using BGP *UPDATE* messages, we assume for simplicity that a BGP message contains only information about one prefix and we use the words :

- *Withdraw message* to indicate a BGP *UPDATE* message containing one route that is withdrawn
- *Update message* to indicate a BGP *UPDATE* containing a new or updated route towards one destination prefix with its attributes

From a conceptual point of view, a BGP router connected to *N* BGP peers, can be described as being composed of four parts as shown in the figure below.

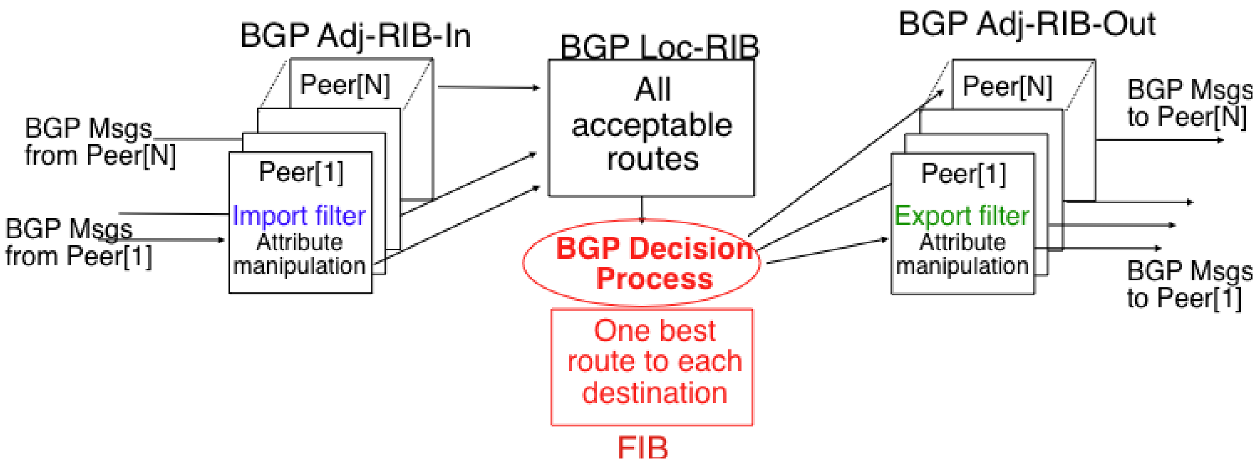


Fig. 71: Organization of a BGP router

[8] 90 seconds is the default delay recommended by **RFC 4271**. However, two BGP peers can negotiate a different timer during the establishment of their BGP session. Using a too small interval to detect BGP session failures is not recommended. BFD [KW2009] can be used to replace BGP's KEEPALIVE mechanism if fast detection of interdomain link failures is required.

In this figure, the router receives BGP messages on the left part of the figure, processes these messages and possibly sends BGP messages on the right part of the figure. A BGP router contains three important data structures :

- the *Adj-RIB-In* contains the BGP routes that have been received from each BGP peer. The routes in the *Adj-RIB-In* are filtered by the *import filter* before being placed in the *BGP-Loc-RIB*. There is one *import filter* per BGP peer.
- the *Local Routing Information Base* (*Loc-RIB*) contains all the routes that are considered as acceptable by the router. The *Loc-RIB* may contain several routes, learned from different BGP peers, towards the same destination prefix.
- the *Forwarding Information Base* (*FIB*) is used by the dataplane to forward packets towards their destination. The *FIB* contains, for each destination, the best route that has been selected by the *BGP decision process*. This decision process is an algorithm that selects, for each destination prefix, the best route according to the router's ranking algorithm that is part of its policy.
- the *Adj-RIB-Out* contains the BGP routes that have been advertised to each BGP peer. The *Adj-RIB-Out* for a given peer is built by applying the peer's *export filter* on the routes that have been installed in the *FIB*. There is one *export filter* per BGP peer. For this reason, the Adj-RIB-Out of a peer may contain different routes than the Adj-RIB-Out of another peer.

When a BGP session starts, the routers first exchange *OPEN* messages to negotiate the options that apply throughout the entire session. Then, each router extracts from its FIB the routes to be advertised to the peer. It is important to note that, for each known destination prefix, a BGP router can only advertise to a peer the route that it has itself installed inside its *FIB*. The routes that are advertised to a peer must pass the peer's *export filter*. The *export filter* is a set of rules that define which routes can be advertised over the corresponding session, possibly after having modified some of its attributes. One *export filter* is associated to each BGP session. For example, on a *shared-cost peering*, the *export filter* only selects the internal routes and the routes that have been learned from a *customer*. The pseudo-code below shows the initialization of a BGP session.

```python
def initialize_BGP_session(remoteAS, remoteIP):
    # Initialize and start BGP session
    # Send BGP OPEN Message to RemoteIP on port 179
    # Follow BGP state machine
    # Advertise local routes and routes learned from peers
    for d in BGPLocRIB:
        msg = build_BGP_update(d)
        msg_to_send = apply_export_filter(remoteAS, msg)
        if msg_to_send is not None:
            send_update(msg_to_send, remoteAS, remoteIP)
    # Entire RIB has now been sent. New updates will be sent
    # to reflect local or distant changes in routers.
```

In the above pseudo-code, the *build_BGP_update(d)* procedure extracts from the *BGP Loc-RIB* the best path towards destination *d* (i.e. the route installed in the FIB) and prepares the corresponding BGP *UPDATE* message. This message is then passed to the *export filter* that returns *None* if the route cannot be advertised to the peer or the (possibly modified) BGP *UPDATE* message to be advertised. BGP routers allow network administrators to specify very complex *export filters*, see e.g. [WMS2004]. A simple *export filter* that implements the equivalent of *split horizon* is shown below.

```python
def apply_export_filter(remoteAS, bgpMsg):
    # Check if RemoteAS already received route
    if remoteAS in bgpMsg.ASPath:
        bgpMsg = None
        # Many additional export policies can be configured:
        # accept or refuse the bgpMsg, modify selected attributes
        # inside bgpMsg, ...
    return bgpMsg
```

At this point, the remote router has received all the exportable BGP routes. After this initial exchange, the router only sends *BGP UPDATE* messages when there is a change (addition of a route, removal of a route or change in the attributes of a route) in one of these exportable routes. Such a change can happen when the router receives a BGP message. The pseudo-code below summarizes the processing of these BGP messages.

```
def bgp_message_received(msg, remoteAS):
    filtered_msg = apply_import_filter(msg, remoteAS)
    if filtered_msg is None: # msg is not acceptable
        return

    if is_update(msg):
        old_route = best_route(msg.prefix)
        insert_in_RIB(msg)
        run_decision_process(RIB)
        if best_route(msg.prefix) != old_route:
            # Best route changed
            out_msg = build_BGP_message(msg.prefix)
            to_send = apply_export_filter(remoteAS, out_msg)
            if to_send is not None:
                # Announce best route
                send_update(to_send, remoteAS, remoteIP)
            elif old_route is not None:
                # Withdraw the route
                send_withdraw(msg.prefix, remoteAS, remoteIP)

    else:  # msg is WITHDRAW
        old_route = best_route(msg.prefix)
        remove_from_rib(msg)
        run_decision_process(RIB)
        if best_route(msg.prefix) != old_route:
            # Best route changed
            out_msg = build_BGP_message(msg.prefix)
            to_send = apply_export_filter(remoteAS, out_msg)
            if to_send is not None:
                # There is still one best route
                # towards msg.prefix
                send_update(to_send, remoteAS, remoteIP)
            elif old_route is not None:
                # No best route anymore
                send_withdraw(msg.prefix, remoteAS, remoteIP)
```

When a BGP message is received, the router first applies the peer's *import filter* to verify whether the message is acceptable or not. If the message is not acceptable, the processing stops. The pseudo-code below shows a simple *import filter*. This *import filter* accepts all routes, except those that already contain the local AS in their AS-Path. If such a route was used, it would cause a routing loop. Another example of an *import filter* would be a filter used by an Internet Service Provider on a session with a customer to only accept routes towards the IP prefixes assigned to the customer by the provider. On real routers, *import filters* can be much more complex and some *import filters* modify the attributes of the received BGP *UPDATE* [WMS2004] .

```
def apply_import_filter(remoteAS, bgpMsg):
    if my_AS in bgpMsg.ASPath:
        bgpMsg = None
        # Many additional import policies can be configured:
        # accept or refuse the bgpMsg, modify selected
        # attributes inside bgpMsg,...
    return bgpMsg
```

Note: The bogon filters

Another example of frequently used *import filters* are the filters that Internet Service Providers use to ignore bogon routes. In the ISP community, a bogon route is a route that should not be advertised on the global Internet. Typical examples include the documentation IPv6 prefix (*2001:db8::/32* used for most examples in this book), the loopback address (*::1/128*) or the IPv6 prefixes that have not yet been allocated by IANA. A well managed BGP router should ensure that it never advertises bogons on the global Internet. Detailed information about these bogons may be found in [IMHM2013].

If the import filter accepts the BGP message, the pseudo-code distinguishes two cases. If this is an *Update message* for prefix *p*, this can be a new route for this prefix or a modification of the route's attributes. The router first retrieves from its *RIB* the best route towards prefix *p*. Then, the new route is inserted in the *RIB* and the *BGP decision process* is run to find whether the best route towards destination *p* changes. A BGP message only needs to be sent to the router's peers if the best route has changed. For each peer, the router applies the *export filter* to verify whether the route can be advertised. If yes, the filtered BGP message is sent. Otherwise, a *Withdraw message* is sent. When the router receives a *Withdraw message*, it also verifies whether the removal of the route from its *RIB* caused its best route towards this prefix to change. It should be noted that, depending on the content of the *RIB* and the *export filters*, a BGP router may need to send a *Withdraw message* to a peer after having received an *Update message* from another peer and conversely.

Let us now discuss in more detail the operation of BGP in an IPv6 network. For this, let us consider the simple network composed of three routers located in three different ASes and shown in the figure below.

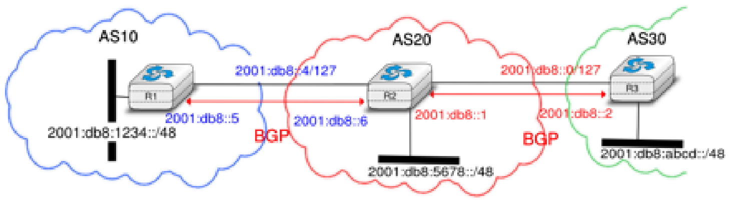


Fig. 72: Utilization of the BGP nexthop attribute

This network contains three routers : *R1*, *R2* and *R3*. Each router is attached to a local IPv6 subnet that it advertises using BGP. There are two BGP sessions, one between *R1* and *R2* and the second between *R2* and *R3*. A */127* subnet is used on each interdomain link (*2001:db8::4/127* on *R1-R2* and *2001:db8::0/127* on *R2-R3*) in conformance with the latest recommendation **RFC 6164**. The BGP sessions run above TCP connections established between the neighboring routers (e.g. *2001:db8::5 - 2001:db8::6* for the *R1-R2* session).

Let us assume that the *R1-R2* BGP session is the first to be established. A *BGP Update* message sent on such a session contains three fields :

- the advertised prefix
- the *BGP nexthop*
- the attributes including the AS-Path

We use the notation *U(prefix, nexthop, attributes)* to represent such a *BGP Update* message in this section. Similarly, *W(prefix)* represents a *BGP withdraw* for the specified prefix. Once the *R1-R2* session has been established, *R1* sends *U(2001:db8:1234::/48,2001:db8::5,AS10)* to *R2* and *R2* sends *U(2001:db8:5678:/48,2001:db8::6,AS20)*. At this point, *R1* can reach *2001:db8:5678::/48* via *2001:db8::6* and *R2* can reach *2001:db8:1234::/48* via *2001:db8::5*.

Once the *R2-R3* has been established, *R3* sends *U(2001:db8:acbd::/48,2001:db8::2,AS30)*. *R2* announces on the *R2-R3* session all the routes inside its RIB. It thus sends to *R3* : *U(2001:db8:1234::/48,2001:db8::1,AS20:AS10)* and *U(2001:db8:5678::/48,2001:db8::1,AS20)*. Note that when *R2* advertises the route that it learned from *R1*, it updates the BGP nexthop and adds its AS number to the AS-Path. *R2* also sends

U(2001:db8:abcd::48,2001:db8::6,AS20:AS30) to *R1* on the *R1-R3* session. At this point, all BGP routes have been exchanged and all routers can reach *2001:db8::1234/48*, *2001:db8:5678::/48* and *2001:db8:abcd::/48*.

If the link between *R2* and *R3* fails, *R3* detects the failure as it did not receive *KEEPALIVE* messages recently from *R2*. At this time, *R3* removes from its RIB all the routes learned over the *R2-R3* BGP session. *R2* also removes from its RIB the routes learned from *R3*. *R2* also sends *W(2001:db8:acbd::/48)* to *R1* over the *R1-R3* BGP session since it does not have a route anymore towards this prefix.

Note: Origin of the routes advertised by a BGP router

A frequent practical question about the operation of BGP is how a BGP router decides to originate or advertise a route for the first time. In practice, this occurs in two situations :

- the router has been manually configured by the network operator to always advertise one or several routes on a BGP session. For example, on the BGP session between UCLouvain and its provider, belnet , UCLouvain's router always advertises the *2001:6a8:3080/48* IPv6 prefix assigned to the campus network
- the router has been configured by the network operator to advertise over its BGP session some of the routes that it learns with its intradomain routing protocol. For example, an enterprise router may advertise over a BGP session with its provider the routes to remote sites when these routes are reachable and advertised by the intradomain routing protocol

The first solution is the most frequent. Advertising routes learned from an intradomain routing protocol is not recommended, this is because if the route flaps[9], this would cause a large number of BGP messages being exchanged in the global Internet.

The BGP decision process

Besides the import and export filters, a key difference between BGP and the intradomain routing protocols is that each domain can define its own ranking algorithm to determine which route is chosen to forward packets when several routes have been learned towards the same prefix. This ranking depends on several BGP attributes that can be attached to a BGP route.

The first BGP attribute that is used to rank BGP routes is the *local-preference* (local-pref) attribute. This attribute is an unsigned integer that is attached to each BGP route received over an eBGP session by the associated import filter.

When comparing routes towards the same destination prefix, a BGP router always prefers the routes with the highest *local-pref*. If the BGP router knows several routes with the same *local-pref*, it prefers among the routes having this *local-pref* the ones with the shortest AS-Path.

The *local-pref* attribute is often used to prefer some routes over others.

A common utilization of *local-pref* is to support backup links. Consider the situation depicted in the figure below. *AS1* would always like to use the high bandwidth link to send and receive packets via *AS2* and only use the backup link upon failure of the primary one.

As BGP routers always prefer the routes with the highest *local-pref* attribute, this policy can be implemented using the following import filter on *R1*

```
import: from  AS2 RA at R1 set localpref=100;
        from  AS2 RB at R1 set localpref=200;
        accept ANY
```

With this import filter, all the BGP routes learned from *RB* over the high bandwidth links are preferred over the routes learned over the backup link. If the primary link fails, the corresponding routes are removed from *R1*'s RIB and *R1*

[9] A link is said to be flapping if it switches several times between an operational state and a disabled state within a short period of time. A router attached to such a link would need to frequently send routing messages.

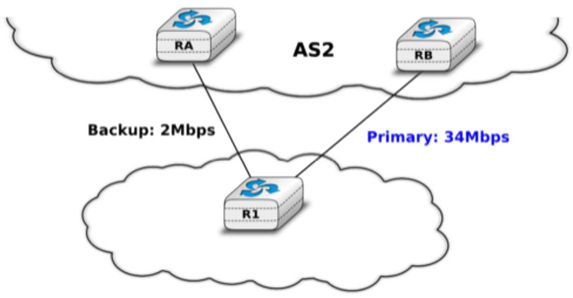


Fig. 73: How to create a backup link with BGP ?

uses the route learned from *RA*. *R1* reuses the routes via *RB* as soon as they are advertised by *RB* once the *R1*-*RB* link comes back.

The import filter above modifies the selection of the BGP routes inside *AS1*. Thus, it influences the route followed by the packets forwarded by *AS1*. In addition to using the primary link to send packets, *AS1* would like to receive its packets via the high bandwidth link. For this, *AS2* also needs to set the *local-pref* attribute in its import filter.

```
import: from  AS1 R1 at RA set localpref=100;
        from  AS1 R1 at RB set localpref=200;
        accept AS1
```

Sometimes, the *local-pref* attribute is used to prefer a *cheap* link compared to a more expensive one. For example, in the network below, *AS1* could wish to send and receive packets mainly via its interdomain link with *AS4*.

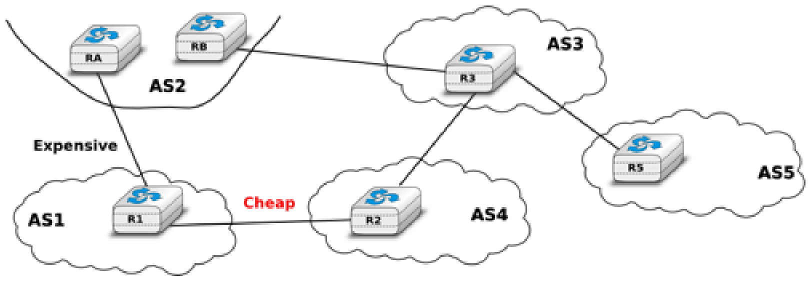


Fig. 74: How to prefer a cheap link over an more expensive one ?

AS1 can install the following import filter on *R1* to ensure that it always sends packets via *R2* when it has learned a route via *AS2* and another via *AS4*.

```
import: from  AS2 RA at R1 set localpref=100;
        from  AS4 R2 at R1 set localpref=200;
        accept ANY
```

However, this import filter does not influence how *AS3* , for example, prefers some routes over others. If the link between *AS3* and *AS2* is less expensive than the link between *AS3* and *AS4*, *AS3* could send all its packets via *AS2* and *AS1* would receive packets over its expensive link. An important point to remember about *local-pref* is that it can be used to prefer some routes over others to send packets, but it has no influence on the routes followed by received packets.

Another important utilization of the *local-pref* attribute is to support the *customer->provider* and *shared-cost* peering relationships. From an economic point of view, there is an important difference between these three types of peering relationships. A domain usually earns money when it sends packets over a *provider->customer* relationship. On the other hand, it must pay its provider when it sends packets over a *customer->provider* relationship. Using a *shared-cost* peering to send packets is usually neutral from an economic perspective. To take into account these economic issues, domains usually configure the import filters on their routers as follows :

- insert a high *local-pref* attribute in the routes learned from a customer
- insert a medium *local-pref* attribute in the routes learned over a shared-cost peering
- insert a low *local-pref* attribute in the routes learned from a provider

With such an import filter, the routers of a domain always prefer to reach destinations via their customers whenever such a route exists. Otherwise, they prefer to use *shared-cost* peering relationships and they only send packets via their providers when they do not know any alternate route. A consequence of setting the *local-pref* attribute like this is that Internet paths are often asymmetrical. Consider for example the internetwork shown in the figure below.

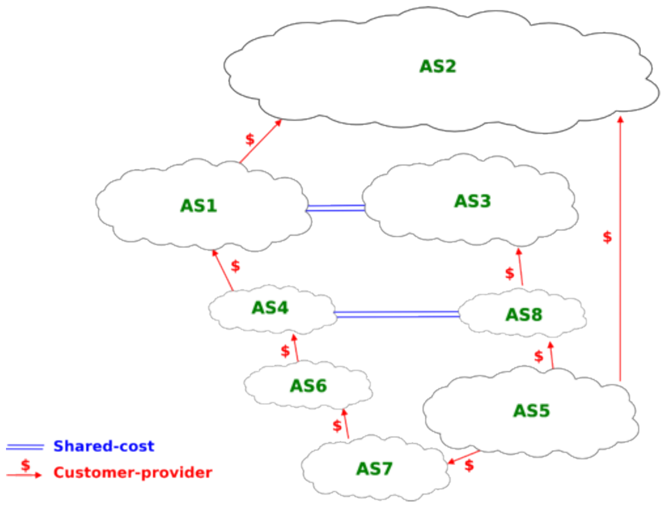


Fig. 75: Asymmetry of Internet paths

Consider in this internetwork the routes available inside *AS1* to reach *AS5*. *AS1* learns the *AS4:AS6:AS7:AS5* path from *AS4*, the *AS3:AS8:AS5* path from *AS3* and the *AS2:AS5* path from *AS2*. The first path is chosen since it was learned from a customer. *AS5* on the other hand receives three paths towards *AS1* via its providers. It may select any of these paths to reach *AS1* , depending on how it prefers one provider over the others.

BGP convergence

In the previous sections, we have explained the operation of BGP routers. Compared to intradomain routing protocols, a key feature of BGP is its ability to support interdomain routing policies that are defined by each domain as its import and export filters and ranking process. A domain can define its own routing policies and router vendors have implemented many configuration tweaks to support complex routing policies. However, the routing policy chosen by a domain may interfere with the routing policy chosen by another domain. To understand this issue, let us first consider the simple internetwork shown below.

In this internetwork, we focus on the route towards *2001:db8::1234/48* which is advertised by *AS1*. Let us also assume that *AS3* (resp. *AS4*) prefers, e.g. for economic reasons, a route learned from *AS4* (*AS3*) over a route learned from *AS1*. When *AS1* sends *U(2001:db8::1234/48,AS1)* to *AS3* and *AS4*, three sequences of exchanges of BGP messages are possible :

1. *AS3* sends first *U(2001:db8:1234/48,AS3:AS1)* to *AS4*. *AS4* has learned two routes towards *2001:db8:1234/48*. It runs its BGP decision process and selects the route via *AS3* and does not advertise a route to *AS3*
2. *AS4* first sends *U(2001:db8:1234/48,AS4:AS1)* to *AS3*. *AS3* has learned two routes towards *2001:db8:1234/48*. It runs its BGP decision process and selects the route via *AS4* and does not advertise a route to *AS4*

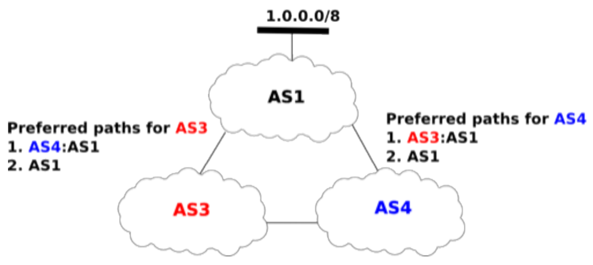


Fig. 76: The disagree internetwork

3. *AS3* sends *U(2001:db8:1234/48,AS3:AS1)* to *AS4* and, at the same time, *AS4* sends *U(2001:db8:1234/48,AS4:AS1)*. *AS3* prefers the route via *AS4* and thus sends *W(2001:db8:1234/48)* to *AS4*. In the mean time, *AS4* prefers the route via *AS3* and thus sends *W(2001:db8:1234/48)* to *AS3*. Upon reception of the *BGP Withdraws*, *AS3* and *AS4* only know the direct route towards *2001:db8:1234/48*. *AS3* (resp. *AS4*) sends *U(2001:db8:1234/48,AS3:AS1)* (resp. *U(2001:db8:1234/48,AS4:AS1)*) to *AS4* (resp. *AS3*). *AS3* and *AS4* could in theory continue to exchange BGP messages for ever. In practice, one of them sends one message faster than the other and BGP converges.

The example above has shown that the routes selected by BGP routers may sometimes depend on the ordering of the BGP messages that are exchanged. Other similar scenarios may be found in **RFC 4264**.

From an operational perspective, the above configuration is annoying since the network operators cannot easily predict which paths are chosen. Unfortunately, there are even more annoying BGP configurations. For example, let us consider the configuration below which is often named *Bad Gadget* [GW1999]

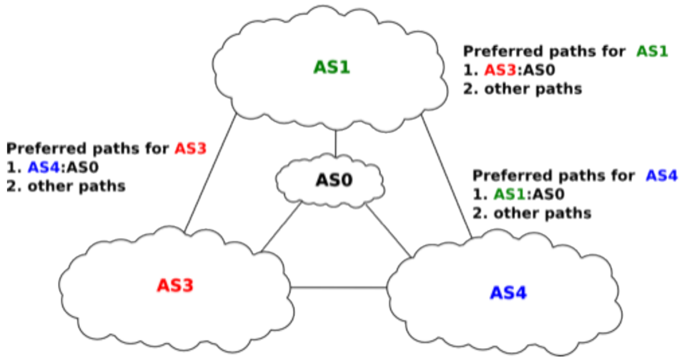


Fig. 77: The bad gadget internetwork

In this internetwork, there are four ASes. *AS0* advertises one route towards one prefix and we only analyze the routes towards this prefix. The routing preferences of *AS1*, *AS3* and *AS4* are the following :

- *AS1* prefers the path *AS3:AS0* over all other paths
- *AS3* prefers the path *AS4:AS0* over all other paths
- *AS4* prefers the path *AS1:AS0* over all other paths

AS0 sends *U(p,AS0)* to *AS1*, *AS3* and *AS4*. As this is the only route known by *AS1*, *AS3* and *AS4* towards *p*, they all select the direct path. Let us now consider one possible exchange of BGP messages :

1. *AS1* sends *U(p, AS1:AS0)* to *AS3* and *AS4*. *AS4* selects the path via *AS1* since this is its preferred path. *AS3* still uses the direct path.
2. *AS4* advertises *U(p,AS4:AS1:AS0)* to *AS3*.

3. *AS3* sends *U(p, AS3:AS0)* to *AS1* and *AS4*. *AS1* selects the path via *AS3* since this is its preferred path. *AS4* still uses the path via *AS1*.
4. As *AS1* has changed its path, it sends *U(p,AS1:AS3:AS0)* to *AS4* and *W(p)* to *AS3* since its new path is via *AS3*. *AS4* switches back to the direct path.
5. *AS4* sends *U(p,AS4:AS0)* to *AS1* and *AS3*. *AS3* prefers the path via *AS4*.
6. *AS3* sends *U(p,AS3:AS4:AS0)* to *AS1* and *W(p)* to *AS4*. *AS1* switches back to the direct path and we are back at the first step.

This example shows that the convergence of BGP is unfortunately not always guaranteed as some interdomain routing policies may interfere with each other in complex ways. [GW1999] have shown that checking for global convergence is either NP-complete or NP-hard. See [GSW2002] for a more detailed discussion.

Fortunately, there are some operational guidelines [GR2001] [GGR2001] that can guarantee BGP convergence in the global Internet. To ensure that BGP will converge, these guidelines consider that there are two types of peering relationships : *customer->provider* and *shared-cost*. In this case, BGP convergence is guaranteed provided that the following conditions are fulfilled :

1. The topology composed of all the directed *customer->provider* peering links is a graph that does not contain any cycle
2. An AS always prefers a route received from a *customer* over a route received from a *shared-cost* peer or a *provider*.

The first guideline implies that the provider of the provider of *ASx* cannot be a customer of *ASx*. Such a relationship would not make sense from an economic perspective as it would imply circular payments. Furthermore, providers are usually larger than customers.

The second guideline also corresponds to economic preferences. Since a provider earns money when sending packets to one of its customers, it makes sense to prefer such customer learned routes over routes learned from providers. [GR2001] also shows that BGP convergence is guaranteed even if an AS associates the same preference to routes learned from a *shared-cost* peer and routes learned from a customer.

From a theoretical perspective, these guidelines should be verified automatically to ensure that BGP will always converge in the global Internet. However, such a verification cannot be performed in practice because this would force all domains to disclose their routing policies (and few are willing to do so) and furthermore the problem is known to be NP-hard [GW1999].

In practice, researchers and operators expect that these guidelines are verified[10] in most domains. Thanks to the large amount of BGP data that has been collected by operators and researchers[11], several studies have analyzed the AS-level topology of the Internet. [SARK2002] is one of the first analysis. More recent studies include [COZ2008] and [DKF+2007]

Based on these studies and [ATLAS2009], the AS-level Internet topology can be summarized as shown in the figure below.

The domains on the Internet can be divided in about four categories according to their role and their position in the AS-level topology.

Due to this organization of the Internet and due to the BGP decision process, most AS-level paths on the Internet have a length of 3-5 AS hops.

[10] Researchers such as [MUF+2007] have shown that modeling the Internet topology at the AS-level requires more than the *shared-cost* and *customer->provider* peering relationships. However, there is no publicly available model that goes beyond these classical peering relationships.

[11] BGP data is often collected by establishing BGP sessions between Unix hosts running a BGP daemon and BGP routers in different ASes. The Unix hosts stores all BGP messages received and regular dumps of its BGP routing table. See http://www.routeviews.org, http://www.ripe.net/ris, http://bgp.potaroo.net or http://irl.cs.ucla.edu/topology/.

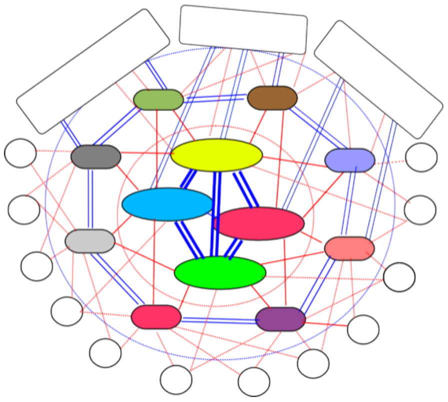


Fig. 78: The layered structure of the global Internet

3.18 Datalink layer technologies

In this section, we review the key characteristics of several datalink layer technologies. We discuss in more detail the technologies that are widely used today. A detailed survey of all datalink layer technologies would be outside the scope of this book.

3.18.1 The Point-to-Point Protocol

Many point-to-point datalink layers[1] have been developed, starting in the 1960s. In this section, we focus on the protocols that are often used to transport IP packets between hosts or routers that are directly connected by a point-to-point link. This link can be a dedicated physical cable, a leased line through the telephone network or a dial-up connection with modems on the two communicating hosts.

The first solution to transport IP packets over a serial line was proposed in **RFC 1055** and is known as *Serial Line IP* (SLIP). SLIP is a simple character stuffing technique applied to IP packets. SLIP defines two special characters : *END* (decimal 192) and *ESC* (decimal 219). *END* appears at the beginning and at the end of each transmitted IP packet and the sender adds *ESC* before each *END* character inside each transmitted IP packet. SLIP only supports the transmission of IP packets and it assumes that the two communicating hosts/routers have been manually configured with each other's IP address. SLIP was mainly used over links offering bandwidth of often less than 20 Kbps. On such a low bandwidth link, sending 20 bytes of IP header followed by 20 bytes of TCP header for each TCP segment takes a lot of time. This initiated the development of a family of compression techniques to efficiently compress the TCP/IP headers. The first header compression technique proposed in **RFC 1144** was designed to exploit the redundancy between several consecutive segments that belong to the same TCP connection. In all these segments, the IP addresses and port numbers are always the same. Furthermore, fields such as the sequence and acknowledgment numbers do not change in a random way. **RFC 1144** defined simple techniques to reduce the redundancy found in successive segments. The development of header compression techniques continued and there are still improvements being developed now **RFC 5795**.

While SLIP was implemented and used in some environments, it had several limitations discussed in **RFC 1055**. The *Point-to-Point Protocol* (PPP) was designed shortly after and is specified in **RFC 1548**. PPP aims to support IP and other network layer protocols over various types of serial lines. PPP is in fact a family of three protocols that are used together :

1. The *Point-to-Point Protocol* defines the framing technique to transport network layer packets.
2. The *Link Control Protocol* that is used to negotiate options and authenticate the session by using username and password or other types of credentials

[1] LAPB and HDLC were widely used datalink layer protocols.

3. The *Network Control Protocol* that is specific for each network layer protocol. It is used to negotiate options that are specific for each protocol. For example, IPv4's NCP **RFC 1548** can negotiate the IPv4 address to be used, the IPv4 address of the DNS resolver. IPv6's NCP is defined in **RFC 5072**.

The PPP framing **RFC 1662** was inspired by the datalink layer protocols standardized by ITU-T and ISO. A typical PPP frame is composed of the fields shown in the figure below. A PPP frame starts with a one byte flag containing *01111110*. PPP can use bit stuffing or character stuffing depending on the environment where the protocol is used. The address and control fields are present for backward compatibility reasons. The 16 bit Protocol field contains the identifier[2] of the network layer protocol that is carried in the PPP frame. *0x002d* is used for an IPv4 packet compressed with **RFC 1144** while *0x002f* is used for an uncompressed IPv4 packet. *0xc021* is used by the Link Control Protocol, *0xc023* is used by the Password Authentication Protocol (PAP). *0x0057* is used for IPv6 packets. PPP supports variable length packets, but LCP can negotiate a maximum packet length. The PPP frame ends with a Frame Check Sequence. The default is a 16 bits CRC, but some implementations can negotiate a 32 bits CRC. The frame ends with the *01111110* flag.

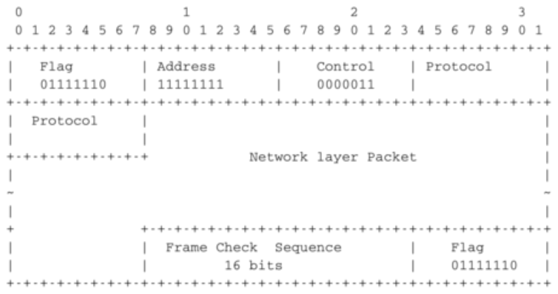


Fig. 79: PPP frame format

PPP played a key role in allowing Internet Service Providers to provide dial-up access over modems in the late 1990s and early 2000s. ISPs operated modem banks connected to the telephone network. For these ISPs, a key issue was to authenticate each user connected through the telephone network. This authentication was performed by using the *Extensible Authentication Protocol* (EAP) defined in **RFC 3748**. EAP is a simple, but extensible protocol that was initially used by access routers to authenticate the users connected through dial-up lines. Several authentication methods, starting from the simple username/password pairs to more complex schemes have been defined and implemented. When ISPs started to upgrade their physical infrastructure to provide Internet access over *Asymmetric Digital Subscriber Lines* (ADSL), they tried to reuse their existing authentication (and billing) systems. To meet these requirements, the IETF developed specifications to allow PPP frames to be transported over other networks than the point-to-point links for which PPP was designed. Nowadays, most ADSL deployments use PPP over either ATM **RFC 2364** or Ethernet **RFC 2516**.

3.18.2 Ethernet

Ethernet was designed in the 1970s at the Palo Alto Research Center [Metcalfe1976]. The first prototype[3] used a coaxial cable as the shared medium and 3 Mbps of bandwidth. Ethernet was improved during the late 1970s and in the 1980s, Digital Equipment, Intel and Xerox published the first official Ethernet specification [DIX]. This specification defines several important parameters for Ethernet networks. The first decision was to standardize the commercial Ethernet at 10 Mbps. The second decision was the duration of the *slot time*. In Ethernet, a long *slot time* enables networks to span a long distance but forces the host to use a larger minimum frame size. The compromise was a *slot time* of 51.2 microseconds, which corresponds to a minimum frame size of 64 bytes.

The third decision was the frame format. The experimental 3 Mbps Ethernet network built at Xerox used short frames containing 8 bit source and destination addresses fields, a 16 bit type indication, up to 554 bytes of payload and a 16 bit CRC. Using 8 bit addresses was suitable for an experimental network, but it was clearly too small for commercial

[2] The IANA maintains the registry of all assigned PPP protocol fields at : http://www.iana.org/assignments/ppp-numbers
[3] Additional information about the history of the Ethernet technology may be found at http://ethernethistory.typepad.com/

deployments. Although the initial Ethernet specification [DIX] only allowed up to 1024 hosts on an Ethernet network, it also recommended three important changes compared to the networking technologies that were available at that time. The first change was to require each host attached to an Ethernet network to have a globally unique datalink layer address. Until then, datalink layer addresses were manually configured on each host. [DP1981] went against that state of the art and noted "*Suitable installation-specific administrative procedures are also needed for assigning numbers to hosts on a network. If a host is moved from one network to another it may be necessary to change its host number if its former number is in use on the new network. This is easier said than done, as each network must have an administrator who must record the continuously changing state of the system (often on a piece of paper tacked to the wall !). It is anticipated that in future office environments, hosts locations will change as often as telephones are changed in present-day offices.*" The second change introduced by Ethernet was to encode each address as a 48 bits field [DP1981]. 48 bit addresses were huge compared to the networking technologies available in the 1980s, but the huge address space had several advantages [DP1981] including the ability to allocate large blocks of addresses to manufacturers. Eventually, other LAN technologies opted for 48 bits addresses as well [IEEE802] . The third change introduced by Ethernet was the definition of *broadcast* and *multicast* addresses. The need for *multicast* Ethernet was foreseen in [DP1981] and thanks to the size of the addressing space it was possible to reserve a large block of multicast addresses for each manufacturer.

The datalink layer addresses used in Ethernet networks are often called MAC addresses. They are structured as shown in the figure below. The first bit of the address indicates whether the address identifies a network adapter or a multicast group. The upper 24 bits are used to encode an Organization Unique Identifier (OUI). This OUI identifies a block of addresses that has been allocated by the secretariat[4] that is responsible for the uniqueness of Ethernet addresses to a manufacturer. Once a manufacturer has received an OUI, it can build and sell products with one of the 16 million addresses in this block.

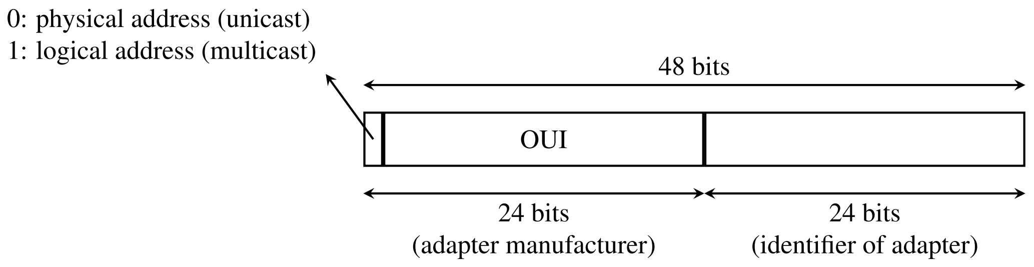


Fig. 80: 48 bits Ethernet address format

The original 10 Mbps Ethernet specification [DIX] defined a simple frame format where each frame is composed of five fields. The Ethernet frame starts with a preamble (not shown in the figure below) that is used by the physical layer of the receiver to synchronize its clock with the sender's clock. The first field of the frame is the destination address. As this address is placed at the beginning of the frame, an Ethernet interface can quickly verify whether it is the frame recipient and if not, cancel the processing of the arriving frame. The second field is the source address. While the destination address can be either a unicast or a multicast/broadcast address, the source address must always be a unicast address. The third field is a 16 bits integer that indicates which type of network layer packet is carried inside the frame. This field is often called the *EtherType*. Frequently used *EtherType* values[5] include *0x0800* for IPv4, *0x86DD* for IPv6[6] and *0x806* for the Address Resolution Protocol (ARP).

[4] Initially, the OUIs were allocated by Xerox [DP1981]. However, once Ethernet became an IEEE and later an ISO standard, the allocation of the OUIs moved to IEEE. The list of all OUI allocations may be found at http://standards.ieee.org/regauth/oui/index.shtml

[5] The official list of all assigned Ethernet type values is available from http://standards.ieee.org/regauth/ethertype/eth.txt

[6] The attentive reader may question the need for different *EtherTypes* for IPv4 and IPv6 while the IP header already contains a version field that can be used to distinguish between IPv4 and IPv6 packets. Theoretically, IPv4 and IPv6 could have used the same *EtherType*. Unfortunately, developers of the early IPv6 implementations found that some devices did not check the version field of the IPv4 packets that they received and parsed frames whose *EtherType* was set to *0x0800* as IPv4 packets. Sending IPv6 packets to such devices would have caused disruptions. To avoid this problem, the IETF decided to apply for a distinct *EtherType* value for IPv6. Such a choice is now mandated by **RFC 6274** (section 3.1), although we can find a funny counter-example in **RFC 6214**.

The fourth part of the Ethernet frame is the payload. The minimum length of the payload is 46 bytes to ensure a minimum frame size, including the header of 512 bits. The Ethernet payload cannot be longer than 1500 bytes. This size was found reasonable when the first Ethernet specification was written. At that time, Xerox had been using its experimental 3 Mbps Ethernet that offered 554 bytes of payload and **RFC 1122** required a minimum MTU of 572 bytes for IPv4. 1500 bytes was large enough to support these needs without forcing the network adapters to contain overly large memories. Furthermore, simulations and measurement studies performed in Ethernet networks revealed that CSMA/CD was able to achieve a very high utilization. This is illustrated in the figure below based on [SH1980], which shows the channel utilization achieved in Ethernet networks containing different numbers of hosts that are sending frames of different sizes.

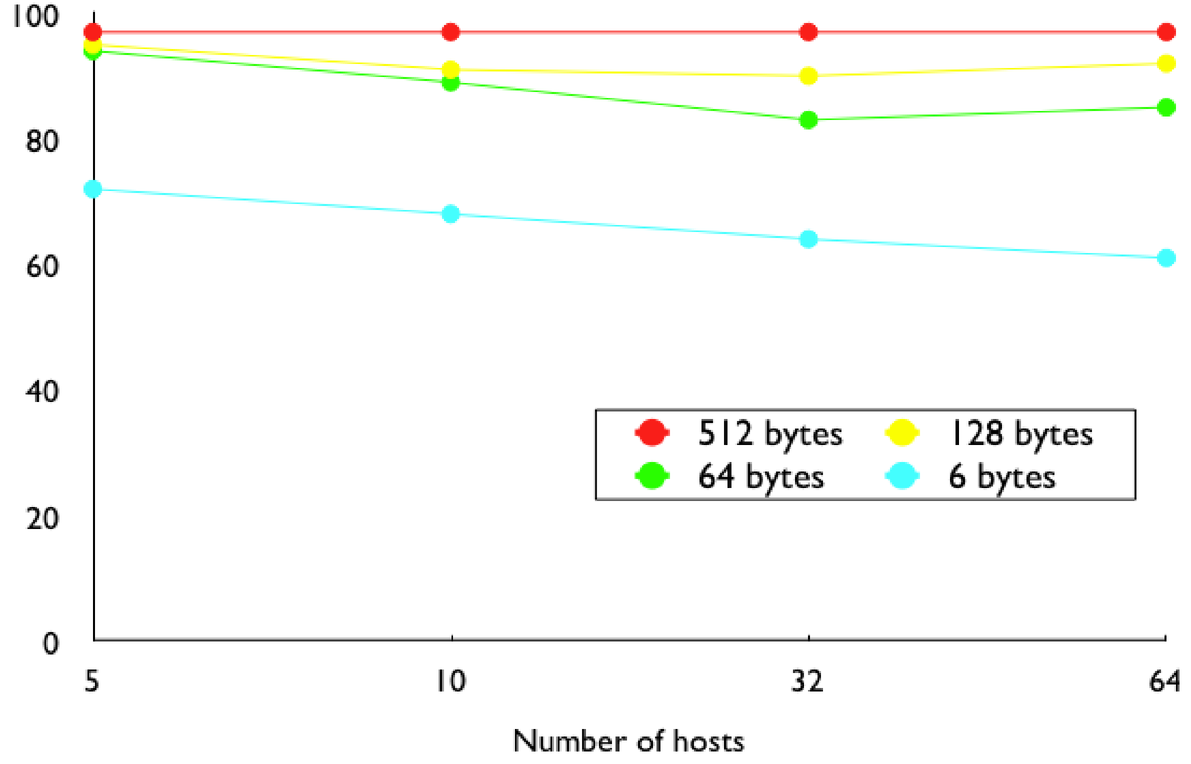


Fig. 81: Impact of the frame length on the maximum channel utilisation [SH1980]

The last field of the Ethernet frame is a 32 bit Cyclical Redundancy Check (CRC). This CRC is able to catch a much larger number of transmission errors than the Internet checksum used by IP, UDP and TCP [SGP98]. The format of the Ethernet frame is shown below.

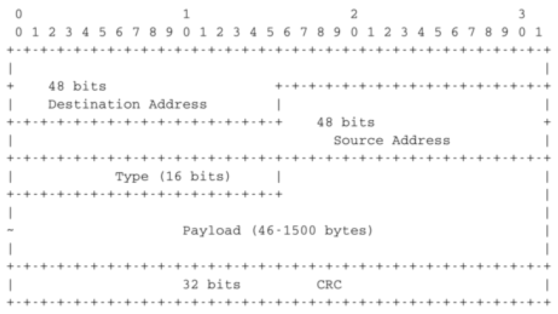


Fig. 82: Ethernet DIX frame format

Note: Where should the CRC be located in a frame ?

The transport and datalink layers usually chose different strategies to place their CRCs or checksums. Transport layer protocols usually place their CRCs or checksums in the segment header. Datalink layer protocols sometimes place their CRC in the frame header, but often in a trailer at the end of the frame. This choice reflects implementation assumptions, but also influences performance **RFC 893**. When the CRC is placed in the trailer, as in Ethernet, the datalink layer can compute it while transmitting the frame and insert it at the end of the transmission. All Ethernet interfaces use this optimization today. When the checksum is placed in the header, as in a TCP segment, it is impossible for the network interface to compute it while transmitting the segment. Some network interfaces provide hardware assistance to compute the TCP checksum, but this is more complex than if the TCP checksum were placed in the trailer[7].

The Ethernet frame format shown above is specified in [DIX]. This is the format used to send both IPv4 **RFC 894** and IPv6 packets **RFC 2464**. After the publication of [DIX], the Institute of Electrical and Electronic Engineers (IEEE) began to standardize several Local Area Network technologies. IEEE worked on several LAN technologies, starting with Ethernet, Token Ring and Token Bus. These three technologies were completely different, but they all agreed to use the 48 bits MAC addresses specified initially for Ethernet [IEEE802] . While developing its Ethernet standard [IEEE802.3], the IEEE 802.3 working group was confronted with a problem. Ethernet mandated a minimum payload size of 46 bytes, while some companies were looking for a LAN technology that could transparently transport short frames containing only a few bytes of payload. Such a frame can be sent by an Ethernet host by padding it to ensure that the payload is at least 46 bytes long. However since the Ethernet header [DIX] does not contain a length field, it is impossible for the receiver to determine how many useful bytes were placed inside the payload field. To solve this problem, the IEEE decided to replace the *Type* field of the Ethernet [DIX] header with a length field[8]. This *Length* field contains the number of useful bytes in the frame payload. The payload must still contain at least 46 bytes, but padding bytes are added by the sender and removed by the receiver. In order to add the *Length* field without significantly changing the frame format, IEEE had to remove the *Type* field. Without this field, it is impossible for a receiving host to identify the type of network layer packet inside a received frame. To solve this new problem, IEEE developed a completely new sublayer called the Logical Link Control [IEEE802.2]. Several protocols were defined in this sublayer. One of them provided a slightly different version of the *Type* field of the original Ethernet frame format. Another contained acknowledgments and retransmissions to provide a reliable service... In practice, [IEEE802.2] is never used to support IP in Ethernet networks. The figure below shows the official [IEEE802.3] frame format.

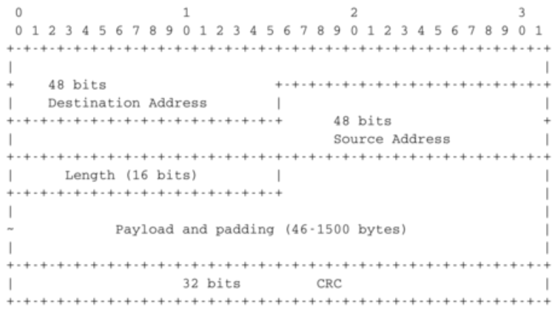


Fig. 83: Ethernet 802.3 frame format

Note: What is the Ethernet service ?

An Ethernet network provides an unreliable connectionless service. It supports three different transmission modes :

[7] These network interfaces compute the TCP checksum while a segment is transferred from the host memory to the network interface [SH2004].

[8] Fortunately, IEEE was able to define the [IEEE802.3] frame format while maintaining backward compatibility with the Ethernet [DIX] frame format. The trick was to only assign values above 1500 as *EtherType* values. When a host receives a frame, it can determine whether the frame's format by checking its *EtherType/Length* field. A value lower smaller than *1501* is clearly a length indicator and thus an [IEEE802.3] frame. A value larger than *1501* can only be type and thus a [DIX] frame.

unicast, *multicast* and *broadcast*. While the Ethernet service is unreliable in theory, a good Ethernet network should, in practice, provide a service that:

- delivers frames to their destination with a very high probability of successful delivery
- does not reorder the transmitted frames

The first property is a consequence of the utilization of CSMA/CD. The second property is a consequence of the physical organization of the Ethernet network as a shared bus. These two properties are important and all revisions to the Ethernet technology have preserved them.

Several physical layers have been defined for Ethernet networks. The first physical layer, usually called 10Base5, provided 10 Mbps over a thick coaxial cable. The characteristics of the cable and the transceivers that were used then enabled the utilization of 500 meter long segments. A 10Base5 network can also include repeaters between segments.

The second physical layer was 10Base2. This physical layer used a thin coaxial cable that was easier to install than the 10Base5 cable, but could not be longer than 185 meters. A 10BaseF physical layer was also defined to transport Ethernet over point-to-point optical links. The major change to the physical layer was the support of twisted pairs in the 10BaseT specification. Twisted pair cables are traditionally used to support the telephone service in office buildings. Most office buildings today are equipped with structured cabling. Several twisted pair cables are installed between any room and a central telecom closet per building or per floor in large buildings. These telecom closets act as concentration points for the telephone service but also for LANs.

The introduction of the twisted pairs led to two major changes to Ethernet. The first change concerns the physical topology of the network. 10Base2 and 10Base5 networks are shared buses, the coaxial cable typically passes through each room that contains a connected computer. A 10BaseT network is a star-shaped network. All the devices connected to the network are attached to a twisted pair cable that ends in the telecom closet. From a maintenance perspective, this is a major improvement. The cable is a weak point in 10Base2 and 10Base5 networks. Any physical damage on the cable broke the entire network and when such a failure occurred, the network administrator had to manually check the entire cable to detect where it was damaged. With 10BaseT, when one twisted pair is damaged, only the device connected to this twisted pair is affected and this does not affect the other devices. The second major change introduced by 10BaseT was that is was impossible to build a 10BaseT network by simply connecting all the twisted pairs together. All the twisted pairs must be connected to a relay that operates in the physical layer. This relay is called an *Ethernet hub*. A *hub* is thus a physical layer relay that receives an electrical signal on one of its interfaces, regenerates the signal and transmits it over all its other interfaces. Some *hubs* are also able to convert the electrical signal from one physical layer to another (e.g. 10BaseT to 10Base2 conversion).

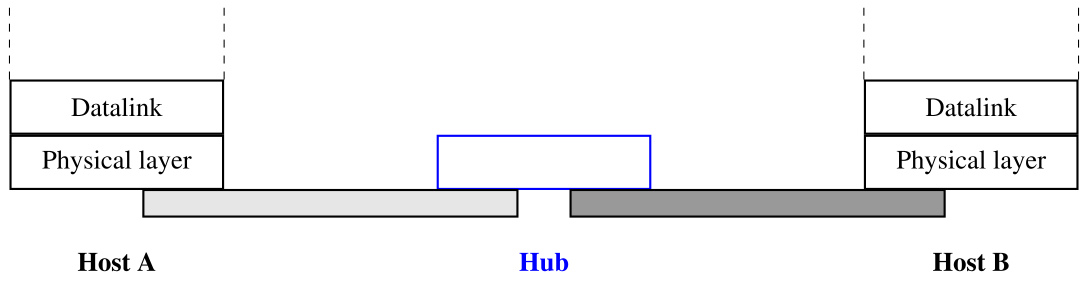


Fig. 84: Ethernet hubs in the reference model

Computers can directly be attached to Ethernet hubs. Ethernet hubs themselves can be attached to other Ethernet hubs to build a larger network. However, some important guidelines must be followed when building a complex network with hubs. First, the network topology must be a tree. As hubs are relays in the physical layer, adding a link between *Hub2* and *Hub3* in the network below would create an electrical shortcut that would completely disrupt the network. This implies that there cannot be any redundancy in a hub-based network. A failure of a hub or of a link between two hubs would partition the network into two isolated networks. Second, as hubs are relays in the physical layer, collisions can happen and must be handled by CSMA/CD as in a 10Base5 network. This implies that the maximum delay between any pair of devices in the network cannot be longer than the 51.2 microseconds *slot time*. If the delay is longer, collisions between short frames may not be correctly detected. This constraint limits the geographical spread of 10BaseT networks containing hubs.

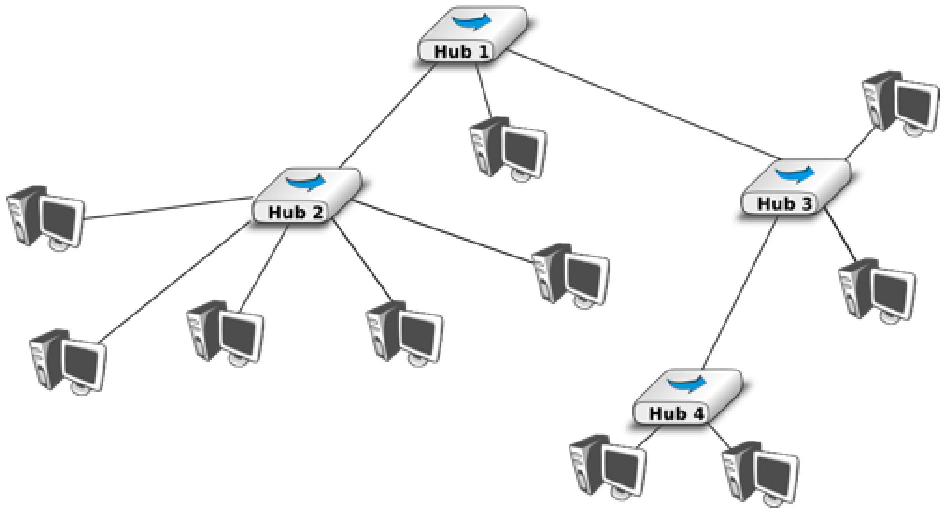


Fig. 85: A hierarchical Ethernet network composed of hubs

In the late 1980s, 10 Mbps became too slow for some applications and network manufacturers developed several LAN technologies that offered higher bandwidth, such as the 100 Mbps FDDI LAN that used optical fibers. As the development of 10Base5, 10Base2 and 10BaseT had shown that Ethernet could be adapted to different physical layers, several manufacturers started to work on 100 Mbps Ethernet and convinced IEEE to standardize this new technology that was initially called *Fast Ethernet*. *Fast Ethernet* was designed under two constraints. First, *Fast Ethernet* had to support twisted pairs. Although it was easier from a physical layer perspective to support higher bandwidth on coaxial cables than on twisted pairs, coaxial cables were a nightmare from deployment and maintenance perspectives. Second, *Fast Ethernet* had to be perfectly compatible with the existing 10 Mbps Ethernet to allow *Fast Ethernet* technology to be used initially as a backbone technology to interconnect 10 Mbps Ethernet networks. This forced *Fast Ethernet* to use exactly the same frame format as 10 Mbps Ethernet. This implied that the minimum *Fast Ethernet* frame size remained at 512 bits. To preserve CSMA/CD with this minimum frame size and 100 Mbps instead of 10 Mbps, the duration of the *slot time* was decreased to 5.12 microseconds.

The evolution of Ethernet did not stop. In 1998, the IEEE published a first standard to provide Gigabit Ethernet over optical fibers. Several other types of physical layers were added afterwards. The 10 Gigabit Ethernet standard appeared in 2002. Work is ongoing to develop standards for 40 Gigabit and 100 Gigabit Ethernet and some are thinking about Terabit Ethernet. The table below lists the main Ethernet standards. A more detailed list may be found at http://en.wikipedia.org/wiki/Ethernet_physical_layer.

Standard	Comments
10Base5	Thick coaxial cable, 500m
10Base2	Thin coaxial cable, 185m
10BaseT	Two pairs of category 3+ UTP
10Base-F	10 Mb/s over optical fiber
100Base-Tx	Category 5 UTP or STP, 100 m maximum
100Base-FX	Two multi-mode optical fiber, 2 km maximum
1000Base-CX	Two pairs shielded twisted pair, 25m maximum
1000Base-SX	Two multi-mode or single mode optical fibers with lasers
10 Gbps	Optical fiber but also Category 6 UTP
40-100 Gbps	Optical fiber (experiences are performed with copper)

Ethernet Switches

Increasing the physical layer bandwidth as in *Fast Ethernet* was only one of the solutions to improve the performance of Ethernet LANs. A second solution was to replace the hubs with more intelligent devices. As *Ethernet hubs* operate in the physical layer, they can only regenerate the electrical signal to extend the geographical reach of the network. From a performance perspective, it would be more interesting to have devices that operate in the datalink layer and can analyze the destination address of each frame and forward the frames selectively on the link that leads to the destination. Such devices are usually called *Ethernet switches*[9]. An *Ethernet switch* is a relay that operates in the datalink layer as is illustrated in the figure below.

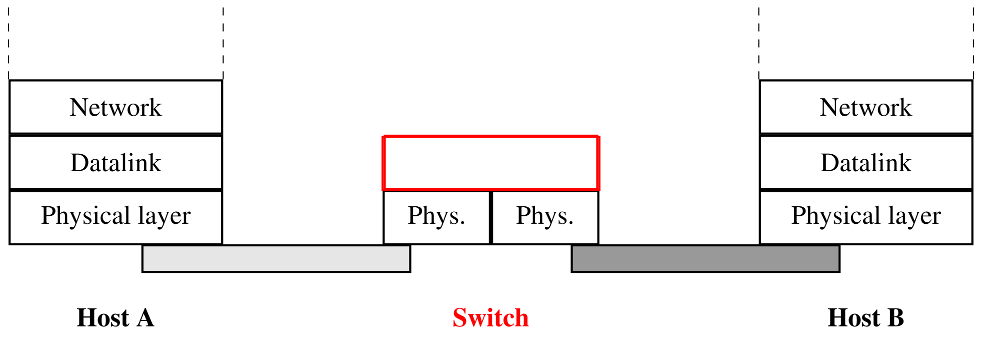


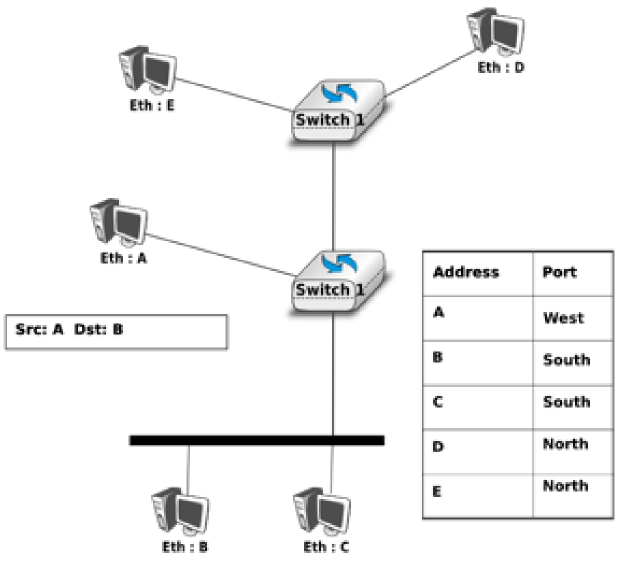
Fig. 86: Ethernet switches in the reference model

An *Ethernet switch* understands the format of the Ethernet frames and can selectively forward frames over each interface. For this, each *Ethernet switch* maintains a *MAC address table*. This table contains, for each MAC address known by the switch, the identifier of the switch's port over which a frame sent towards this address must be forwarded to reach its destination. This is illustrated below with the *MAC address table* of the bottom switch. When the switch receives a frame destined to address *B*, it forwards the frame on its South port. If it receives a frame destined to address *D*, it forwards it only on its North port.

One of the selling points of Ethernet networks is that, thanks to the utilization of 48 bits MAC addresses, an Ethernet LAN is plug and play at the datalink layer. When two hosts are attached to the same Ethernet segment or hub, they can immediately exchange Ethernet frames without requiring any configuration. It is important to retain this plug and play capability for Ethernet switches as well. This implies that Ethernet switches must be able to build their MAC address table automatically without requiring any manual configuration. This automatic configuration is performed by the *MAC address learning* algorithm that runs on each Ethernet switch. This algorithm extracts the source address of the received frames and remembers the port over which a frame from each source Ethernet address has been received. This information is inserted into the MAC address table that the switch uses to forward frames. This allows the switch to automatically learn the ports that it can use to reach each destination address, provided that this host has previously sent at least one frame. This is not a problem since most upper layer protocols use acknowledgments at some layer and thus even an Ethernet printer sends Ethernet frames as well.

The pseudo-code below details how an Ethernet switch forwards Ethernet frames. It first updates its *MAC address table* with the source address of the frame. The *MAC address table* used by some switches also contains a timestamp that is updated each time a frame is received from each known source address. This timestamp is used to remove from the *MAC address table* entries that have not been active during the last *n* minutes. This limits the growth of the *MAC address table*, but also allows hosts to move from one port to another. The switch uses its *MAC address table* to forward the received unicast frame. If there is an entry for the frame's destination address in the *MAC address table*, the frame is forwarded selectively on the port listed in this entry. Otherwise, the switch does not know how to reach the destination address and it must forward the frame on all its ports except the port from which the frame has been

[9] The first Ethernet relays that operated in the datalink layers were called *bridges*. In practice, the main difference between switches and bridges is that bridges were usually implemented in software while switches are hardware-based devices. Throughout this text, we always use *switch* when referring to a relay in the datalink layer, but you might still see the word *bridge*.



Address	Port
A	West
B	South
C	South
D	North
E	North

Fig. 87: Operation of Ethernet switches

received. This ensures that the frame will reach its destination, at the expense of some unnecessary transmissions. These unnecessary transmissions will only last until the destination has sent its first frame. Multicast and Broadcast frames are also forwarded in a similar way.

```
# Arrival of frame F on port P
# Table : MAC address table dictionary : addr->port
# Ports : list of all ports on the switch
src = F.SourceAddress
dst = F.DestinationAddress
Table[src] = P  #  src heard on port P
if is_unicast(dst):
    if dst in Table:
        forward_frame(F, Table[dst])
    else:
        for o in Ports:
          if o != P:
               forward_frame(F, o)
else:
    # multicast or broadcast destination
    for o in Ports:
        if o != P:
            forward_frame(F, o)
```

Note: Security issues with Ethernet hubs and switches

From a security perspective, Ethernet hubs have the same drawbacks as the older coaxial cable. A host attached to a hub will be able to capture all the frames exchanged between any pair of hosts attached to the same hub. Ethernet switches are much better from this perspective thanks to the selective forwarding, a host will usually only receive the frames destined to itself as well as the multicast, broadcast and unknown frames. However, this does not imply that switches are completely secure. There are, unfortunately, attacks against Ethernet switches. From a security perspective, the *MAC address table* is one of the fragile elements of an Ethernet switch. This table has a fixed size. Some low-end switches can store a few tens or a few hundreds of addresses while higher-end switches can store tens of thousands of addresses or more. From a security point of view, a limited resource can be the target of Denial of Service attacks. Unfortunately, such attacks are also possible on Ethernet switches. A malicious host could overflow the *MAC address table* of the switch by generating thousands of frames with random source addresses. Once the *MAC address table* is full, the switch needs to broadcast all the frames that it receives. At this point, an attacker will receive unicast frames that are not destined to its address. The ARP attack discussed in the previous chapter could also occur with Ethernet switches [Vyncke2007]. Recent switches implement several types of defenses against these attacks, but they need to be carefully configured by the network administrator. See [Vyncke2007] for a detailed discussion on security issues with Ethernet switches.

The *MAC address learning* algorithm combined with the forwarding algorithm work well in a tree-shaped network such as the one shown above. However, to deal with link and switch failures, network administrators often add redundant links to ensure that their network remains connected even after a failure. Let us consider what happens in the Ethernet network shown in the figure below.

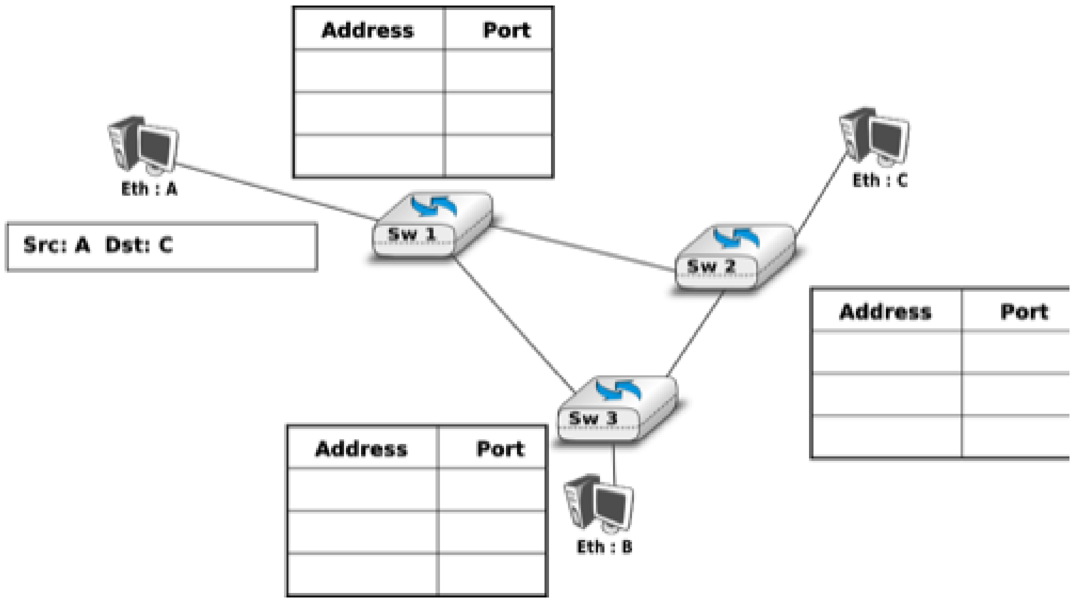


Fig. 88: Ethernet switches in a loop

When all switches boot, their *MAC address table* is empty. Assume that host *A* sends a frame towards host *C*. Upon reception of this frame, switch1 updates its *MAC address table* to remember that address *A* is reachable via its West port. As there is no entry for address *C* in switch1's *MAC address table*, the frame is forwarded to both switch2 and switch3. When switch2 receives the frame, its updates its *MAC address table* for address *A* and forwards the frame to host *C* as well as to switch3. switch3 has thus received two copies of the same frame. As switch3 does not know how to reach the destination address, it forwards the frame received from switch1 to switch2 and the frame received from switch2 to switch1… The single frame sent by host *A* will be continuously duplicated by the switches until their *MAC address table* contains an entry for address *C*. Quickly, all the available link bandwidth will be used to forward all the copies of this frame. As Ethernet does not contain any *TTL* or *HopLimit*, this loop will never stop.

The *MAC address learning* algorithm allows switches to be plug-and-play. Unfortunately, the loops that arise when the network topology is not a tree are a severe problem. Forcing the switches to only be used in tree-shaped networks as hubs would be a severe limitation. To solve this problem, the inventors of Ethernet switches have developed the *Spanning Tree Protocol*. This protocol allows switches to automatically disable ports on Ethernet switches to ensure that the network does not contain any cycle that could cause frames to loop forever.

The Spanning Tree Protocol (802.1d)

The *Spanning Tree Protocol* (STP), proposed in [Perlman1985], is a distributed protocol that is used by switches to reduce the network topology to a spanning tree, so that there are no cycles in the topology. For example, consider the network shown in the figure below. In this figure, each bold line corresponds to an Ethernet to which two Ethernet switches are attached. This network contains several cycles that must be broken to allow Ethernet switches, using the MAC address learning algorithm, to exchange frames.

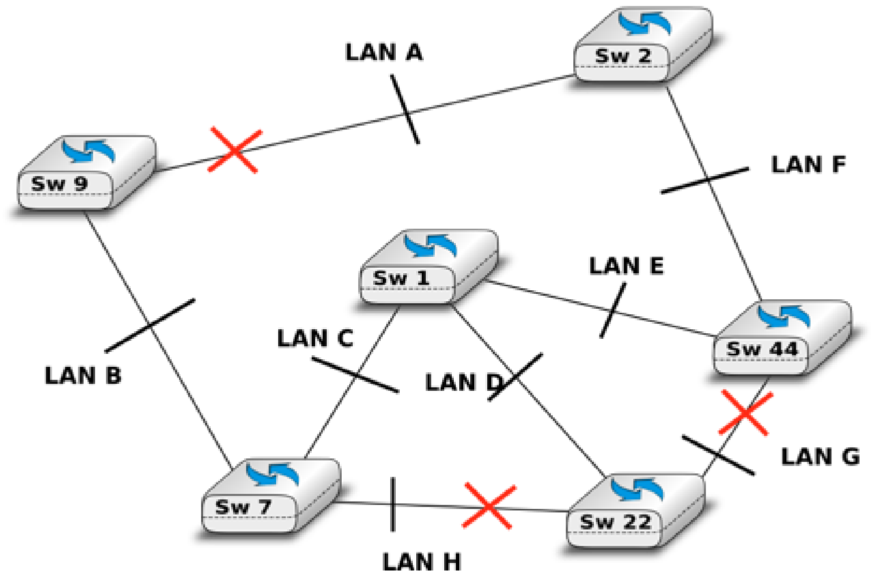


Fig. 89: Spanning tree computed in a switched Ethernet network

In this network, the STP will compute the following spanning tree. *Switch1* will be the root of the tree. All the interfaces of *Switch1*, *Switch2* and *Switch7* are part of the spanning tree. Only the interface connected to *LAN B* will be active on *Switch9*. *LAN H* will only be served by *Switch7* and the port of *Switch44* on *LAN G* will be disabled. A frame originating on *LAN B* and destined for *LAN A* will be forwarded by *Switch7* on *LAN C*, then by *Switch1* on *LAN E*, then by *Switch44* on *LAN F* and eventually by *Switch2* on *LAN A*.

Switches running the *Spanning Tree Protocol* exchange *BPDUs*. These *BPDUs* are always sent as frames with destination MAC address as the *ALL_BRIDGES* reserved multicast MAC address. Each switch has a unique 64 bit *identifier*. To ensure uniqueness, the lower 48 bits of the identifier are set to the unique MAC address allocated to the switch by its manufacturer. The high order 16 bits of the switch identifier can be configured by the network administrator to influence the topology of the spanning tree. The default value for these high order bits is 32768.

The switches exchange *BPDUs* to build the spanning tree. Intuitively, the spanning tree is built by first selecting the switch with the smallest *identifier* as the root of the tree. The branches of the spanning tree are then composed of the shortest paths that allow all of the switches that compose the network to be reached. The *BPDUs* exchanged by the switches contain the following information :

- the *identifier* of the root switch (*R*)
- the *cost* of the shortest path between the switch that sent the *BPDU* and the root switch (*c*)
- the *identifier* of the switch that sent the *BPDU* (*T*)
- the number of the switch port over which the *BPDU* was sent (*p*)

We will use the notation *<R,c,T,p>* to represent a *BPDU* whose *root identifier* is *R*, *cost* is *c* and that was sent from the port *p* of switch *T*. The construction of the spanning tree depends on an ordering relationship among the *BPDUs*. This ordering relationship could be implemented by the Python function below.

```python
# returns True if bpdu b1 is better than bpdu b2
def better(b1, b2):
    return ((b1.R < b2.R) or
            ((b1.R == b2.R) and (b1.c < b2.c)) or
            ((b1.R == b2.R) and (b1.c == b2.c) and (b1.T < b2.T)) or
            ((b1.R == b2.R) and (b1.c == b2.c) and (b1.T == b2.T) and (b1.p < b2.p)))
```

In addition to the *identifier* discussed above, the network administrator can also configure a *cost* to be associated to each switch port. Usually, the *cost* of a port depends on its bandwidth and the [IEEE802.1d] standard recommends the values below. Of course, the network administrator may choose other values. We will use the notation *cost[p]* to indicate the cost associated to port *p* in this section.

Bandwidth	Cost
10 Mbps	2000000
100 Mbps	200000
1 Gbps	20000
10 Gbps	2000
100 Gbps	200

The *Spanning Tree Protocol* uses its own terminology that we illustrate in the figure above. A switch port can be in three different states : *Root*, *Designated* and *Blocked*. All the ports of the *root* switch are in the *Designated* state. The state of the ports on the other switches is determined based on the *BPDU* received on each port.

The *Spanning Tree Protocol* uses the ordering relationship to build the spanning tree. Each switch listens to *BPDUs* on its ports. When *BPDU = <R,c,T,p>* is received on port *q*, the switch computes the port's *root priority vector*: *V[q] = <R,c+cost[q],T,p,q>* , where *cost[q]* is the cost associated to the port over which the *BPDU* was received. The switch stores in a table the last *root priority vector* received on each port. The switch then compares its own *identifier* with the smallest *root identifier* stored in this table. If its own *identifier* is smaller, then the switch is the root of the spanning tree and is, by definition, at a distance *0* of the root. The *BPDU* of the switch is then *<R,0,R,p>*, where *R* is the switch *identifier* and *p* will be set to the port number over which the *BPDU* is sent.

Otherwise, the switch chooses the best priority vector from its table, *bv = <R,c+cost[q'],T,p,q'>*. The port *q'*, over which this best root priority vector was learned, is the switch port that is closest to the *root* switch. This port becomes the *Root* port of the switch. There is only one *Root* port per switch (except for the *Root* switches whose ports are all *Designated*). The switch can then compute its own *BPDU* as *BPDU = <R,c',S,p>* , where *R* is the *root identifier*, *c'* the cost of the best root priority vector, *S* the identifier of the switch and *p* will be replaced by the number of the port over which the *BPDU* will be sent.

To determine the state of its other ports, the switch compares its own *BPDU* with the last *BPDU* received on each port. Note that the comparison is done by using the *BPDUs* and not the *root priority vectors*. If the switch's *BPDU* is better than the last *BPDU* of this port, the port becomes a *Designated* port. Otherwise, the port becomes a *Blocked* port.

The state of each port is important when considering the transmission of *BPDUs*. The root switch regularly sends its own *BPDU* over all of its (*Designated*) ports. This *BPDU* is received on the *Root* port of all the switches that are directly connected to the *root switch*. Each of these switches computes its own *BPDU* and sends this *BPDU* over all its *Designated* ports. These *BPDUs* are then received on the *Root* port of downstream switches, which then compute their own *BPDU*, etc. When the network topology is stable, switches send their own *BPDU* on all their *Designated* ports, once they receive a *BPDU* on their *Root* port. No *BPDU* is sent on a *Blocked* port. Switches listen for *BPDUs* on their *Blocked* and *Designated* ports, but no *BPDU* should be received over these ports when the topology is stable. The utilization of the ports for both *BPDUs* and data frames is summarized in the table below.

Port state	Receives BPDUs	Sends BPDU	Handles data frames
Blocked	yes	no	no
Root	yes	no	yes
Designated	yes	yes	yes

To illustrate the operation of the *Spanning Tree Protocol*, let us consider the simple network topology in the figure below.

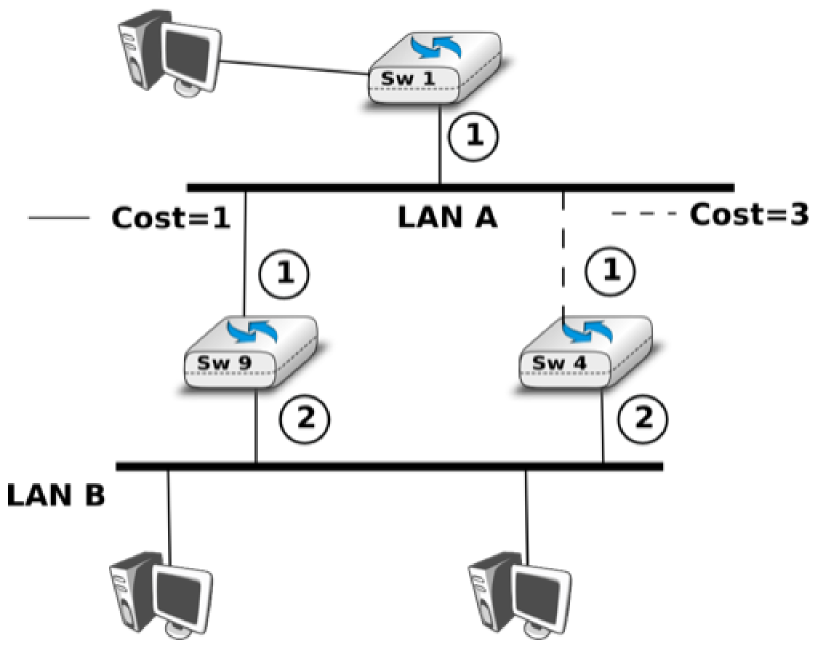


Fig. 90: A simple Spanning tree computed in a switched Ethernet network

Assume that *Switch4* is the first to boot. It sends its own *BPDU = <4,0,4,1>* (resp. *BPDU = <4,0,4,2>*) on port 1 (resp. port 2). When *Switch1* boots, it sends *BPDU = <1,0,1,1>*. This *BPDU* is received by *Switch4*, which updates its *BPDU* and root priority vector tables and computes a new *BPDU = <1,3,4,1>* (resp. *BPDU = <1,3,4,2>*) on port 1 (resp. port 2). Indeed, there is only one root priority vector and hence, it is the best one. Port 1 of *Switch4* becomes the *Root* port while its second port is still in the *Designated* state.

Assume now that *Switch9* boots and immediately receives *Switch1* 's *BPDU* on port 1. *Switch9* computes its own *BPDU = <1,1,9,1>* (resp. *BPDU = <1,1,9,2>*) on port 1 (resp. port 2) and port 1 becomes the *Root* port of this switch. The *BPDU* is sent on port 2 of *Switch9* and reaches *Switch4*. *Switch4* compares the priority vectors. It notices that the last computed vector (i.e., *V[2] = <1,2,9,2,2>*) is better than *V[1] = <1,3,1,1,1>*. Thus, *Switch4*'s *BPDU* is recomputed and port 2 becomes the *Root* port of *Switch4*. *Switch4* compares its new *BPDU = <1,2,4,p>* with the last *BPDU* received on each port (except for the *Root* port). Port 1 becomes a *Blocked* port on *Switch4* because the *BPDU=<1,0,1,1>* received on this port is better.

During the computation of the spanning tree, switches discard all received data frames, as at that time the network topology is not guaranteed to be loop-free. Once that topology has been stable for some time, the switches again start to use the MAC learning algorithm to forward data frames. Only the *Root* and *Designated* ports are used to forward data frames. Switches discard all the data frames received on their *Blocked* ports and never forward frames on these ports.

Switches, ports and links can fail in a switched Ethernet network. When a failure occurs, the switches must be able to recompute the spanning tree to recover from the failure. The *Spanning Tree Protocol* relies on regular transmissions of the *BPDUs* to detect these failures. A *BPDU* contains two additional fields : the *Age* of the *BPDU* and the *Maximum Age*. The *Age* contains the amount of time that has passed since the root switch initially originated the *BPDU*. The root switch sends its *BPDU* with an *Age* of zero and each switch that computes its own *BPDU* increments its *Age* by one. The *Age* of the *BPDUs* stored on a switch's table is also incremented every second. A *BPDU* expires when its *Age* reaches the *Maximum Age*. When the network is stable, this does not happen as *BPDU* s are regularly sent by the *root* switch and downstream switches. However, if the *root* fails or the network becomes partitioned, *BPDU* will expire and switches will recompute their own *BPDU* and restart the *Spanning Tree Protocol*. Once a topology change has been detected, the forwarding of the data frames stops as the topology is not guaranteed to be loop-free. Additional details about the reaction to failures may be found in [IEEE802.1d].

Virtual LANs

Another important advantage of Ethernet switches is the ability to create Virtual Local Area Networks (VLANs). A virtual LAN can be defined as a *set of ports attached to one or more Ethernet switches*. A switch can support several VLANs and it runs one MAC learning algorithm for each Virtual LAN. When a switch receives a frame with an unknown or a multicast destination, it forwards it over all the ports that belong to the same Virtual LAN but not over the ports that belong to other Virtual LANs. Similarly, when a switch learns a source address on a port, it associates it to the Virtual LAN of this port and uses this information only when forwarding frames on this Virtual LAN.

The figure below illustrates a switched Ethernet network with three Virtual LANs. *VLAN2* and *VLAN3* only require a local configuration of switch *S1*. Host *C* can exchange frames with host *D*, but not with hosts that are outside of its VLAN. *VLAN1* is more complex as there are ports of this VLAN on several switches. To support such VLANs, local configuration is not sufficient anymore. When a switch receives a frame from another switch, it must be able to determine the VLAN in which the frame originated to use the correct MAC table to forward the frame. This is done by assigning an identifier to each Virtual LAN and placing this identifier inside the headers of the frames that are exchanged between switches.

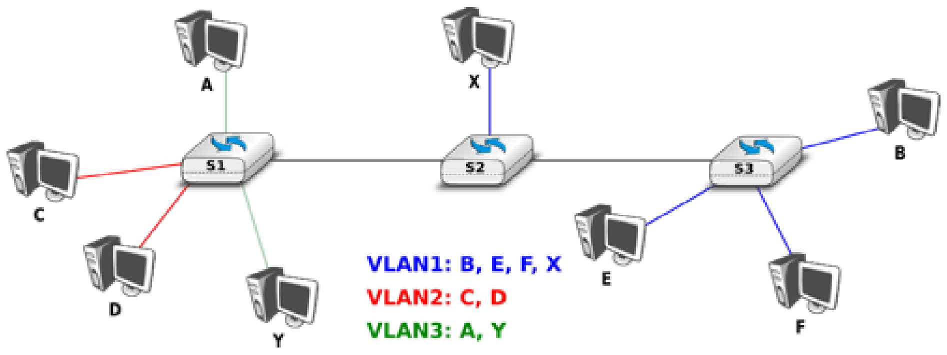


Fig. 91: Virtual Local Area Networks in a switched Ethernet network

IEEE defined in the [IEEE802.1q] standard a special header to encode the VLAN identifiers. This 32 bit header includes a 12 bit VLAN field that contains the VLAN identifier of each frame. The format of the [IEEE802.1q] header is described below.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Tag Protocol Identifier    | PCP |C|     VLAN Identifier   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 92: Format of the 802.1q header

The [IEEE802.1q] header is inserted immediately after the source MAC address in the Ethernet frame (i.e. before the EtherType field). The maximum frame size is increased by 4 bytes. It is encoded in 32 bits and contains four fields. The Tag Protocol Identifier is set to *0x8100* to allow the receiver to detect the presence of this additional header. The *Priority Code Point* (PCP) is a three bit field that is used to support different transmission priorities for the frame. Value *0* is the lowest priority and value *7* the highest. Frames with a higher priority can expect to be forwarded earlier than frames having a lower priority. The *C* bit is used for compatibility between Ethernet and Token Ring networks. The last 12 bits of the 802.1q header contain the VLAN identifier. Value *0* indicates that the frame does not belong to any VLAN while value *0xFFF* is reserved. This implies that 4094 different VLAN identifiers can be used in an Ethernet network.

3.18.3 802.11 wireless networks

The radio spectrum is a limited resource that must be shared by everyone. During most of the twentieth century, governments and international organizations have regulated most of the radio spectrum. This regulation controls the utilization of the radio spectrum, in order to prevent interference among different users. A company that wants to use a frequency range in a given region must apply for a license from the regulator. Most regulators charge a fee for the utilization of the radio spectrum and some governments have encouraged competition among companies bidding for the same frequency to increase the license fees.

In the 1970s, after the first experiments with ALOHANet, interest in wireless networks grew. Many experiments were done on and outside the ARPANet. One of these experiments was the first mobile phone , which was developed and tested in 1973. This experimental mobile phone was the starting point for the first generation analog mobile phones. Given the growing demand for mobile phones, it was clear that the analog mobile phone technology was not sufficient to support a large number of users. To support more users and new services, researchers in several countries worked on the development of digital mobile telephones. In 1987, several European countries decided to develop the standards for a common cellular telephone system across Europe : the *Global System for Mobile Communications* (GSM). Since then, the standards have evolved and more than three billion users are connected to GSM networks today.

While most of the frequency ranges of the radio spectrum are reserved for specific applications and require a special license, there are a few exceptions. These exceptions are known as the Industrial, Scientific and Medical (ISM) radio bands. These bands can be used for industrial, scientific and medical applications without requiring a license from the regulator. For example, some radio-controlled models use the 27 MHz ISM band and some cordless telephones operate in the 915 MHz ISM. In 1985, the 2.400-2.500 GHz band was added to the list of ISM bands. This frequency range corresponds to the frequencies that are emitted by microwave ovens. Sharing this band with licensed applications would have likely caused interference, given the large number of microwave ovens that are used. Despite the risk of interference with microwave ovens, the opening of the 2.400-2.500 GHz allowed the networking industry to develop several wireless network techniques to allow computers to exchange data without using cables. In this section, we discuss in more detail the most popular one, i.e. the WiFi [IEEE802.11] family of wireless networks. Other wireless networking techniques such as BlueTooth or HiperLAN use the same frequency range.

Today, WiFi is a very popular wireless networking technology. There are more than several hundreds of millions of WiFi devices. The development of this technology started in the late 1980s with the WaveLAN proprietary wireless network. WaveLAN operated at 2 Mbps and used different frequency bands in different regions of the world. In the early 1990s, the IEEE created the 802.11 working group to standardize a family of wireless network technologies. This working group was very prolific and produced several wireless networking standards that use different frequency ranges and different physical layers. The table below provides a summary of the main 802.11 standards.

Standard	Frequency	Typical throughput	Max bandwidth	Range (m) indoor/outdoor
802.11	2.4 GHz	0.9 Mbps	2 Mbps	20/100
802.11a	5 GHz	23 Mbps	54 Mbps	35/120
802.11b	2.4 GHz	4.3 Mbps	11 Mbps	38/140
802.11g	2.4 GHz	19 Mbps	54 Mbps	38/140
802.11n	2.4/5 GHz	74 Mbps	150 Mbps	70/250

When developing its family of standards, the IEEE 802.11 working group took a similar approach as the IEEE 802.3 working group that developed various types of physical layers for Ethernet networks. 802.11 networks use the CSMA/CA Medium Access Control technique described earlier and they all assume the same architecture and use the same frame format.

The architecture of WiFi networks is slightly different from the Local Area Networks that we have discussed until now. There are, in practice, two main types of WiFi networks : *independent* or *adhoc* networks and *infrastructure* networks[10]. An *independent* or *adhoc* network is composed of a set of devices that communicate with each other. These devices play the same role and the *adhoc* network is usually not connected to the global Internet. *Adhoc*

[10] The 802.11 working group defined the *basic service set (BSS)* as a group of devices that communicate with each other. We continue to use *network* when referring to a set of devices that communicate.

networks are used when for example a few laptops need to exchange information or to connect a computer with a WiFi printer.

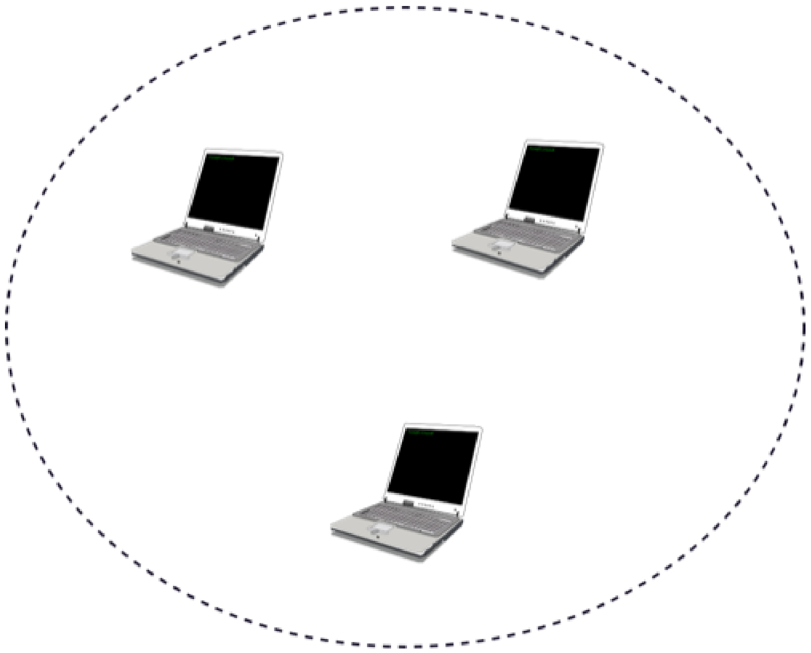


Fig. 93: An 802.11 independent or adhoc network

Most WiFi networks are *infrastructure* networks. An *infrastructure* network contains one or more *access points* that are attached to a fixed Local Area Network (usually an Ethernet network) that is connected to other networks such as the Internet. The figure below shows such a network with two access points and four WiFi devices. Each WiFi device is associated to one access point and uses this access point as a relay to exchange frames with the devices that are associated to another access point or reachable through the LAN.

An 802.11 access point is a relay that operates in the datalink layer like switches. The figure below represents the layers of the reference model that are involved when a WiFi host communicates with a host attached to an Ethernet network through an access point.

802.11 devices exchange variable length frames, which have a slightly different structure than the simple frame format used in Ethernet LANs. We review the key parts of the 802.11 frames. Additional details may be found in [IEEE802.11] and [Gast2002] . An 802.11 frame contains a fixed length header, a variable length payload that may contain up 2324 bytes of user data and a 32 bits CRC. Although the payload can contain up to 2324 bytes, most 802.11 deployments use a maximum payload size of 1500 bytes as they are used in *infrastructure* networks attached to Ethernet LANs. An 802.11 data frame is shown below.

The first part of the 802.11 header is the 16 bit *Frame Control* field. This field contains flags that indicate the type of frame (data frame, RTS/CTS, acknowledgment, management frames, etc), whether the frame is sent to or from a fixed LAN, etc [IEEE802.11]. The *Duration* is a 16 bit field that is used to reserve the transmission channel. In data frames, the *Duration* field is usually set to the time required to transmit one acknowledgment frame after a SIFS delay. Note that the *Duration* field must be set to zero in multicast and broadcast frames. As these frames are not acknowledged, there is no need to reserve the transmission channel after their transmission. The *Sequence control* field contains a 12 bits sequence number that is incremented for each data frame and a 4 bits fragment number.

The astute reader may have noticed that the 802.11 data frames contain three 48-bits address fields[11] . This is surprising

[11] In fact, the [IEEE802.11] frame format contains a fourth optional address field. This fourth address is only used when an 802.11 wireless network is used to interconnect bridges attached to two classical LAN networks.

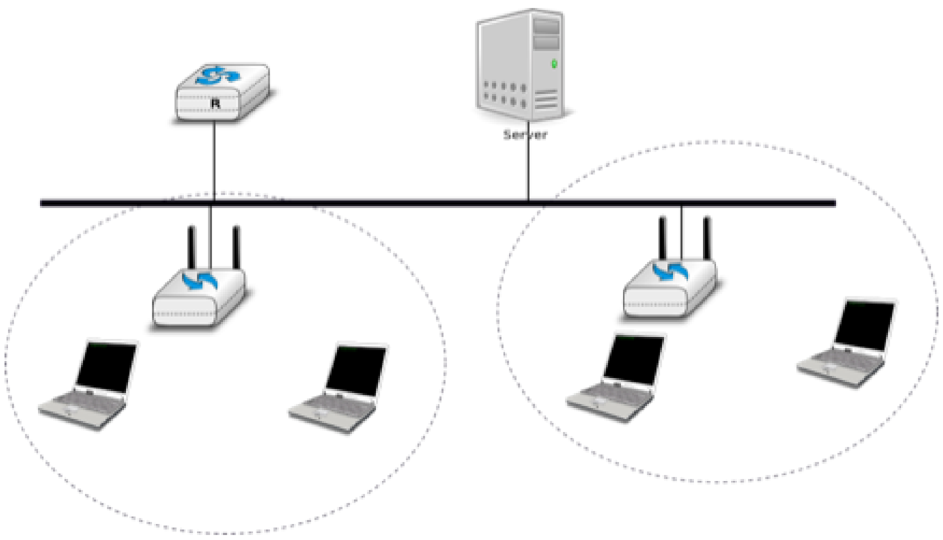


Fig. 94: An 802.11 infrastructure network

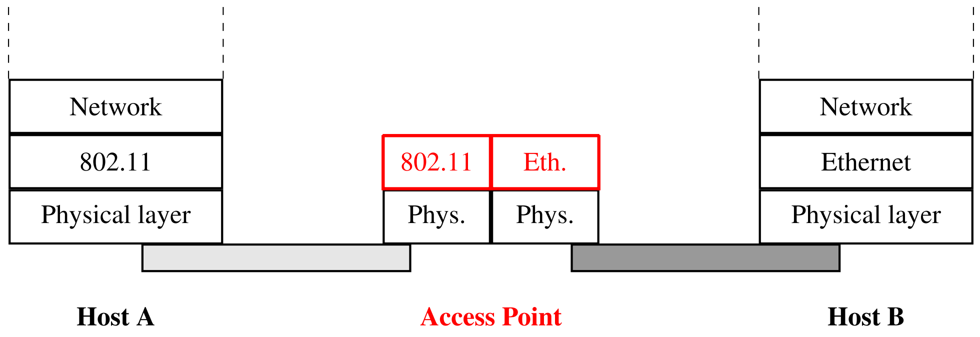


Fig. 95: An 802.11 access point

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Frame  Control         |           Duration            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|      Address 1 (48 bits)      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+      Address 2 (48 bits)      |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|      Address 3 (48 bits)      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               |       Sequence control        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                      Payload (0-2324 bytes)                   ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         32 bits CRC                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 96: 802.11 data frame format

compared to other protocols in the network and datalink layers whose headers only contain a source and a destination address. The need for a third address in the 802.11 header comes from the *infrastructure* networks. In such a network, frames are usually exchanged between routers and servers attached to the LAN and WiFi devices attached to one of the access points. The role of the three address fields is specified by bit flags in the *Frame Control* field.

When a frame is sent from a WiFi device to a server attached to the same LAN as the access point, the first address of the frame is set to the MAC address of the access point, the second address is set to the MAC address of the source WiFi device and the third address is the address of the final destination on the LAN. When the server replies, it sends an Ethernet frame whose source address is its MAC address and the destination address is the MAC address of the WiFi device. This frame is captured by the access point that converts the Ethernet header into an 802.11 frame header. The 802.11 frame sent by the access point contains three addresses : the first address is the MAC address of the destination WiFi device, the second address is the MAC address of the access point and the third address the MAC address of the server that sent the frame.

802.11 control frames are simpler than data frames. They contain a *Frame Control*, a *Duration* field and one or two addresses. The acknowledgment frames are very small. They only contain the address of the destination of the acknowledgment. There is no source address and no *Sequence Control* field in the acknowledgment frames. This is because the acknowledgment frame can easily be associated to the previous frame that it acknowledges. Indeed, each unicast data frame contains a *Duration* field that is used to reserve the transmission channel to ensure that no collision will affect the acknowledgment frame. The *Sequence Control* field is mainly used by the receiver to remove duplicate frames. Duplicate frames are detected as follows. Each data frame contains a 12 bits sequence number in the *Sequence Control* field and the *Frame Control* field contains the *Retry* bit flag that is set when a frame is transmitted. Each 802.11 receiver stores the most recent sequence number received from each source address in frames whose *Retry* bit is reset. Upon reception of a frame with the *Retry* bit set, the receiver verifies its sequence number to determine whether it is a duplicated frame or not.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Frame Control         |          Duration             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|       Receiver Address        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               |              CRC              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         CRC  (cont.)          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 97: IEEE 802.11 ACK and CTS frames

802.11 RTS/CTS frames are used to reserve the transmission channel, in order to transmit one data frame and its acknowledgment. The RTS frames contain a *Duration* and the transmitter and receiver addresses. The *Duration* field of the RTS frame indicates the duration of the entire reservation (i.e. the time required to transmit the CTS, the data frame, the acknowledgments and the required SIFS delays). The CTS frame has the same format as the acknowledgment frame.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Frame Control         |          Duration             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|       Receiver Address        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+      Transmitter Address      |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          32 bits CRC                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 98: IEEE 802.11 RTS frame format

Note: The 802.11 service

Despite the utilization of acknowledgments, the 802.11 layer only provides an unreliable connectionless service like

Ethernet networks that do not use acknowledgments. The 802.11 acknowledgments are used to minimize the probability of frame duplication. They do not guarantee that all frames will be correctly received by their recipients. Like Ethernet, 802.11 networks provide a high probability of successful delivery of the frames, not a guarantee. Furthermore, it should be noted that 802.11 networks do not use acknowledgments for multicast and broadcast frames. This implies that in practice such frames are more likely to suffer from transmission errors than unicast frames.

In addition to the data and control frames that we have briefly described above, 802.11 networks use several types of management frames. These management frames are used for various purposes. We briefly describe some of these frames below. A detailed discussion may be found in [IEEE802.11] and [Gast2002].

A first type of management frames are the *beacon* frames. These frames are broadcasted regularly by access points. Each *beacon frame* contains information about the capabilities of the access point (e.g. the supported 802.11 transmission rates) and a *Service Set Identity* (SSID). The SSID is a null-terminated ASCII string that can contain up to 32 characters. An access point may support several SSIDs and announce them in beacon frames. An access point may also choose to remain silent and not advertise beacon frames. In this case, WiFi stations may send *Probe request* frames to force the available access points to return a *Probe response* frame.

Note: IP over 802.11

Two types of encapsulation schemes were defined to support IP in Ethernet networks : the original encapsulation scheme, built above the Ethernet DIX format is defined in **RFC 894** and a second encapsulation **RFC 1042** scheme, built above the LLC/SNAP protocol [IEEE802.2]. In 802.11 networks, the situation is simpler and only the **RFC 1042** encapsulation is used. In practice, this encapsulation adds 6 bytes to the 802.11 header. The first four bytes correspond to the LLC/SNAP header. They are followed by the two bytes Ethernet Type field (*0x800* for IP and *0x806* for ARP). The figure below shows an IP packet encapsulated in an 802.11 frame.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                         802.11 header                         ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   SNAP/DSAP   |   SNAP/SSAP   |    Control    |   RFC 1042    |
|     0xAA      |     0xAA      |     0x03      |     0x00      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Type              |                               |
|             0x800             |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+                               |
|                                                               |
~                           IP packet                           ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          32 bits CRC                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 99: IP over IEEE 802.11

The second important utilization of the management frames is to allow a WiFi station to be associated with an access point. When a WiFi station starts, it listens to beacon frames to find the available SSIDs. To be allowed to send and receive frames via an access point, a WiFi station must be associated to this access point. If the access point does not use any security mechanism to secure the wireless transmission, the WiFi station simply sends an *Association request* frame to its preferred access point (usually the access point that it receives with the strongest radio signal). This frame contains some parameters chosen by the WiFi station and the SSID that it requests to join. The access point replies with an *Association response frame* if it accepts the WiFI station.

CHAPTER FOUR

PART 3: PRACTICE

4.1 Exercises

This section gathers a series of open questions that have been designed to enable the students to improve their understanding of the topics presented in the e-book. These open questions typically contain a small problem that needs to be solved by the student. They are designed so that they can lead to discussions with teaching assistants.

Several *Multiple Choice Questions* and simple exercises are included in the main text and supported by the https://inginious.org platform.

A good understanding of the topics covered by this e-book can only be obtained by solving the proposed exercises. Reading the e-book from the first to the last page is not sufficient to get a detailed knowledge of computer networking.

4.2 Reliable transfer

4.2.1 Open questions

1. Consider a b bits per second link between two hosts that has a propagation delay of t seconds. Derive a formula that computes the time elapsed between the transmission of the first bit of a d bytes frame from a sending host and the reception of the last bit of this frame on the receiving host.
2. Transmission links have sometimes different upstream and downstream bandwidths. A typical example are access networks that use ADSL (Asymmetric Digital Subscriber Lines). Consider two hosts connected via an ADSL link having an upstream bandwidth of 1 Mbps and a downstream bandwidth of 50 Mbps. The propagation delay between the two hosts is 10 milliseconds. What is the maximum throughput, expressed in frames/second, that the alternating bit protocol can obtain on this link if each data frame has a length of 125 bytes and acknowledgments are 25 bytes long. Same question if the protocol is modified to support 1500 bytes long data frames.
3. How would you set the duration of the retransmission timer in the alternating bit protocol ?
4. A version of the Alternating Bit Protocol supporting variable length frames uses a header that contains the following fields :
   - a *number* (0 or 1)
   - a *length* field that indicates the length of the data
   - a Cyclic Redundancy Check (*CRC*)

   To speedup the transmission of the frames, a student proposes to compute the CRC over the data part of the frame but not over the header. What do you think of this proposed solution ?

5. Derive a mathematical expression that provides the *goodput*, i.e. the amount of payload bytes that have been transmitted during a period of time, achieved by the Alternating Bit Protocol assuming that :
   - Each frame contains D bytes of data and c bytes of control information
   - Each acknowledgment contains c bytes of control information
   - The bandwidth of the two directions of the link is set to B bits per second
   - The delay between the two hosts is s seconds in both directions
   - there are no transmission errors
6. Consider a go-back-n sender and a go-back receiver that are directly connected with a 10 Mbps link that has a propagation delay of 100 milliseconds. Assume that the retransmission timer is set to three seconds. If the window has a length of 4 frames, draw a time-sequence diagram showing the transmission consisting of 10 data frames (each frame contains 10000 bits):

- when there are no losses
- when the third and seventh frames are lost
- when every second acknowledgment is discarded due to transmission errors

7. Same question when using selective repeat instead of go-back-n. Note that the answer is not necessarily the same.

4.2.2 Practice

Reliable protocols depend on error detection algorithms to detect transmission errors. The following questions will reinforce your understanding of these algorithms.

1. Reliable protocols rely on different types of checksums to verify whether frames have been affected by transmission errors. The most frequently used checksums are :
   - the Internet checksum used by UDP, TCP and other Internet protocols which is defined in **RFC 1071** and implemented in various libraries.
   - the 16 bits or the 32 bits Cyclical Redundancy Checks (CRC) that are often used on disks, in zip archives and in datalink layer protocols. See http://rosettacode.org/wiki/CRC-32 for CRC-32 implementations in various languages.
   - the Fletcher checksum [Fletcher1982], see https://en.wikipedia.org/wiki/Fletcher%27s_checksum for implementation details.

   By using your knowledge of the Internet checksum, can you find a transmission error that will not be detected by these checksums?
2. The Cyclic Redundancy Checks (CRCs) are efficient error detection codes that are able to detect :
   - all errors that affect an odd number of bits
   - all errors that affect a sequence of bits which is shorter than the length of the CRC

   Implement a small software that computes the CRC-32 for a text file. Then, modify the contents of the file to change an even number of bits or an odd number of bits inside the file. When modifying the file, remember that an ASCII file is composed of 8 bits characters that are encoded by using the ASCII table that you can find at : http://en.wikipedia.org/wiki/ASCII . You can also write a small program that produces binary files that are a small variation of each other.
3. Checksums and CRCs should not be confused with secure hash functions such as MD5 defined in **RFC 1321** or SHA-1 described in **RFC 4634**. Secure hash functions are used to ensure that files or sometimes frames/packets/segments have not been modified. Secure hash functions aim at detecting malicious changes

while checksums and CRCs only detect random transmission errors. Use the shasum or md5sum programs on Linux to perform the same tests as above.

4.2.3 Discussion questions

1. Consider two high-end servers connected back-to-back by using a 10 Gbps interface. If the delay between the two servers is one millisecond, what is the throughput that can be achieved by a reliable protocol that is using 10,000 bits frames and a window of

- one frame
- ten frames
- hundred frames

2. Is it possible for a go-back-n receiver to inter-operate with a selective-repeat sender ? Justify your answer.
3. Is it possible for a selective-repeat receiver to inter-operate with a go-back-n sender ? Justify your answer.
4. A go-back-n sender has sent 2^n data frames. All the frames have been received correctly and in-order by the receiver, but all the returned acknowledgments have been lost. Show by using a time sequence diagram (e.g. by considering a window of four frames) what happens in this case. Can you fix the problem on the go-back-n sender ?
5. Same question as above, but assume now that both the sender and the receiver implement selective repeat. Note that the answer can be different from the above question.

4.3 Using sockets for inter-process communication

Popular operating systems allow isolating different programs by executing them in separate *processes*. A *socket* is a tool provided by the operating system that enables two separated processes to communicate with each other. A socket takes the form of a file descriptor and can be seen as a communication pipe through which the communicating processes can exchange arbitrary information. In order to receive a message, a process must be attached to a specific *address* that the peer can use to reach it. The socket is a powerful abstraction as it allows processes to communicate

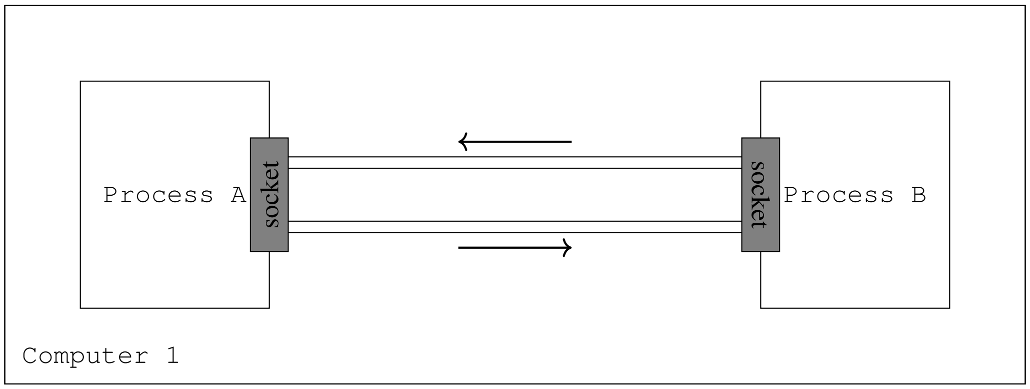


Fig. 1: Connecting two processes communicating on the same computer

even if they are located on different computers. In this specific cases, the inter-processes communication will go through a network. Networked applications were usually implemented by using the *socket API*. This API was designed when TCP/IP was first implemented in the Unix BSD operating system [Sechrest] [LFJLMT], and has served as the model for many APIs between applications and the networking stack in an operating system. Although the socket API is very popular, other APIs have also been developed. For example, the STREAMS API has been added to several Unix

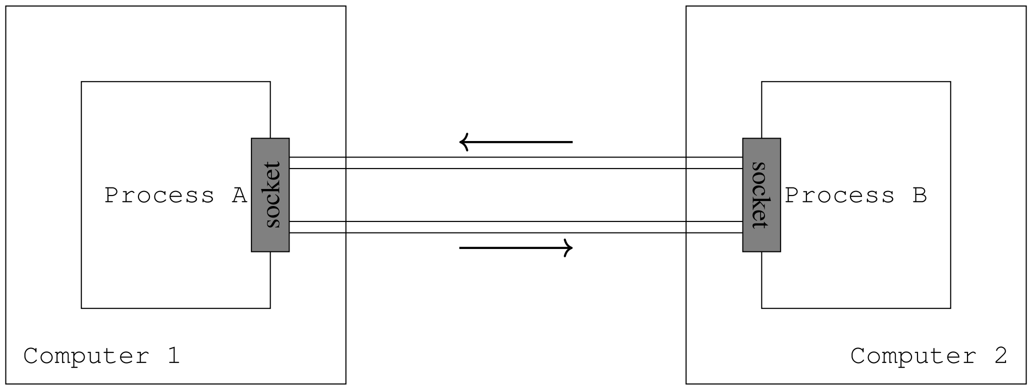


Fig. 2: Connecting two processes communicating on different computers

System V variants [Rago1993]. The socket API is supported by most programming languages and several textbooks have been devoted to it. Users of the C language can consult [DC2009], [Stevens1998], [SFR2004] or [Kerrisk2010]. The Java implementation of the socket API is described in [CD2008] and in the Java tutorial. In this section, we will use the C socket API to illustrate the key concepts.

The socket API is quite low-level and should be used only when you need a complete control of the network access. If your application simply needs, for instance, to retrieve data from a web server, there are much simpler and higher-level APIs.

A detailed discussion of the socket API is outside the scope of this section and the references cited above provide a detailed discussion of all the details of the socket API. As a starting point, it is interesting to compare the socket API with the service primitives that we have discussed in the previous chapter. Let us first consider the connectionless service that consists of the following two primitives :

- *DATA.request(destination,message)* is used to send a message to a specified destination. In this socket API, this corresponds to the `send` method.
- *DATA.indication(message)* is issued by the transport service to deliver a message to the application. In the socket API, this corresponds to the return of the `recv` method that is called by the application.

The *DATA* primitives are exchanged through a service access point. In the socket API, the equivalent to the service access point is the *socket*. A *socket* is a data structure which is maintained by the networking stack and is used by the application every time it needs to send or receive data through the networking stack.

4.3.1 Sending data to a peer using a socket

In order to reach a peer, a process must know its *address*. An address is a value that identifies a peer in a given network. There exists many different kinds of address families. For example, some of them allow reaching a peer using the file system on the computer. Some others enable communicating with a remote peer through a network. The socket API provides generic functions: the peer address is taken as a `struct sockaddr *`, which can point to any family of address. This is partly why sockets are a powerful abstraction.

The `sendto` system call allows to send data to a peer identified by its socket address through a given socket.

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct␣
↪sockaddr *dest_addr, socklen_t addrlen);
```

The first argument is the file descriptor of the socket that we use to perform the communication. `buf` is a buffer of length `len` containing the bytes to send to the peer. The usage of `flags` argument is out of the scope of this section and can be set to 0. `dest_addr` is the socket address of the destination to which we want to send the bytes, its length is passed using the `addrlen` argument.

In the following example, a C program is sending the bytes `'h'`, `'e'`, `'l'`, `'l'` and `'o'` to a remote process located at address `peer_addr`, using the already created socket `sock`.

```
int send_hello_to_peer(int sock, struct sockaddr *peer_addr, size_t peer_addr_len) {
    ssize_t sent = sendto(sock, "hello", strlen("hello"), 0, peer_addr, peer_addr_
↪len);
    if (sent == -1) {
        printf("could not send the message, error: %s\n", strerror(errno));
        return errno;
    }
    return 0;
}
```

As the `sendto` function is generic, this function will work correctly independently from the fact that the peer's address is defined as a path on the computer filesystem or a network address.

4.3.2 Receiving data from a peer using a socket

Operating systems allow assigning an address to a socket using the `bind` system call. This is useful when you want to receive messages from another program to which you announced your socket address. Once the address is assigned to the socket, the program can receive data from others using system calls such as `recv` and `read`. Note that we can use the `read` system call as the operating system provides a socket as a file descriptor.

The following program binds its socket to a given socket address and then waits for receiving new bytes, using the already created socket `sock`.

```
#define MAX_MESSAGE_SIZE 2500
int bind_and_receive_from_peer(int sock, struct sockaddr *local_addr, socklen_t local_
↪addr_len) {
    int err = bind(sock, local_addr, local_addr_len);  // assign our address to the␣
↪socket
    if (err == -1) {
        printf("could not bind on the socket, error: %s\n", strerror(errno));
        return errno;
    }
    char buffer[MAX_MESSAGE_SIZE];  // allocate a buffer of MAX_MESSAGE_SIZE bytes on␣
↪the stack
    ssize_t n_received = recv(sock, buffer, MAX_MESSAGE_SIZE, 0);   // equivalent to␣
↪do: read(sock, buffer, MAX_MESSAGE_SIZE);
    if (n_received == -1) {
        printf("could not receive the message, error: %s\n", strerror(errno));
        return errno;
    }

    // let's print what we received !
    printf("received %ld bytes:\n", n_received);
    for (int i = 0 ; i < n_received ; i++) {
        printf("0x%hhx ('%c') ", buffer[i], buffer[i]);
    }
    printf("\n");
    return 0;
}
```

Note: Depending on the socket address family, the operating system might implicitly assign an address to an unbound socket upon a call to `write`, `send` or `sendto`. While this is a useful behavior, describing it precisely is out of the scope of this section.

Warning: While the provided examples show the usage of a *char* array as the data buffer, implementers should **never** assume that it contains a string. C programs rely on the *char* type to refer to a 8-bit long value, and arbitrary binary values can be exchanged over the network (i.e., the `\0` value does not delimit the end of the data).

Using this code, the program will read and print an arbitrary message received from an arbitrary peer who knows the program's socket address. If we want to know the address of the peer that sent us the message, we can use the `recvfrom` system call. This is what a modified version of `bind_and_receive_from_peer` is doing below.

```c
#define MAX_MESSAGE_SIZE 2500
int bind_and_receive_from_peer_with_addr(int sock) {
    int err = bind(sock, local_addr, local_addr_len);  // assign our address to the␣
↪socket
    if (err == -1) {
        printf("could not bind on the socket, error: %s\n", strerror(errno));
        return errno;
    }
    struct sockaddr_storage peer_addr;  // allocate the peer's address on the stack.␣
↪It will be initialized when we receive a message
    socklen_t peer_addr_len = sizeof(struct sockaddr_storage); // variable that will␣
↪contain the length of the peer's address
    char buffer[MAX_MESSAGE_SIZE];  // allocate a buffer of MAX_MESSAGE_SIZE bytes on␣
↪the stack
    ssize_t n_received = recvfrom(sock, buffer, MAX_MESSAGE_SIZE, 0, (struct sockaddr␣
↪*) &peer_addr, &peer_addr_len);
    if (n_received == -1) {
        printf("could not receive the message, error: %s\n", strerror(errno));
        return errno;
    }

    // let's print what we received !
    printf("received %ld bytes:\n", n_received);
    for (int i = 0 ; i < n_received ; i++) {
        printf("0x%hhx ('%c') ", buffer[i], buffer[i]);
    }
    printf("\n");

    // let's now print the address of the peer
    uint8_t *peer_addr_bytes = (uint8_t *) &peer_addr;
    printf("the socket address of the peer is (%ld bytes):\n", peer_addr_len);
    for (int i = 0 ; i < peer_addr_len ; i++) {
        printf("0x%hhx ", peer_addr_bytes[i]);
    }
    printf("\n");
    return 0;
}
```

This function is now using the `recvfrom` system call that will also provide the address of the peer who sent the message. As addresses are generic and can have different sizes, `recvfrom` also tells us the size of the address that it has written.

4.3.3 `connect`: connecting a socket to a remote address

Operating systems enable linking a socket to a remote address so that every information sent through the socket will only be sent to this remote address, and the socket will only receive messages sent by this remote address. This can be done using the `connect` system call shown below.

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

This system call will assign the socket `sockfd` to the `addr` remote socket address. The process can then use the `send` and `write` system calls that do not to specify the destination socket address. Furthermore, the calls to `recv` and `read` will only deliver messages sent by this remote address. This is useful when we only care about the other peer messages.

The following program connects a socket to a remote address, sends a message and waits for a reply.

```
#define MAX_MESSAGE_SIZE 2500
int send_hello_to_and_read_reply_from_connected_peer(int sock, struct sockaddr *peer_
↪addr, size_t peer_addr_len) {
    int err = connect(sock, peer_addr, peer_addr_len); // connect the socket to the␣
↪peer
    if (err == -1) {
        printf("cound not connect the socket: %s\n", strerror(errno));
        return errno;
    }
    ssize_t written = write(sock, "hello", strlen("hello"));  // we can use the␣
↪generic write(2) system call: we do not need to specify the destination socket␣
↪address
    if (written == -1) {
        printf("could not send the message, error: %s\n", strerror(errno));
        return errno;
    }
    uint8_t buffer[MAX_MESSAGE_SIZE]; // allocate the receive buffer on the stack
    ssize_t amount_read = read(sock, buffer, MAX_MESSAGE_SIZE);
    if (amount_read == -1) {
        printf("could not read on the socket, error: %s\n", strerror(errno));
        return errno;
    }
    // let's print what we received !
    printf("received %ld bytes:\n", amount_read);
    for (int i = 0 ; i < amount_read ; i++) {
        printf("0x%hhx ('%c') ", buffer[i], buffer[i]);
    }
    return 0;
}
```

4.3.4 Creating a new socket to communicate through a network

Until now, we learned how to use sockets that were already created. When writing a whole program, you will have to create you own sockets and choose the concrete technology that it will use to communicate with others. In this section, we will create new sockets and allow a program to communicate with processes located on another computer using a network. The most recent standardized technology used to communicate through a network is the *IPv6* network protocol. In the IPv6 protocol, hosts are identified using *IPv6 addresses*. Modern operating systems allow IPv6 network communications between programs to be done using the socket API, just as we did in the previous sections.

A program can use the `socket` system call to create a new socket.

```
int socket(int domain, int type, int protocol)
```

The `domain` parameter specifies the address family that we will use to concretely perform the communication. For an IPv6 socket, the `domain` parameter will be set to the value `AF_INET6`, telling the operating system that we plan to communicate using IPv6 addresses. The `type` parameter specifies the communication guarantees that we need. For now, we will use the type `SOCK_DGRAM` which allows us to send *unreliable messages*. This means that each data that we send at each call of `sendto` will either be completely received or not received at all. The last parameter will be set to `0`. The following line creates a socket, telling the operating system that we want to communicate using IPv6 addresses and that we want to send unreliable messages.

```
int sock = socket(AF_INET6, SOCK_DGRAM, 0);
```

Sending a message to a remote peer using its IPv6 address

Now that we created an IPv6 socket, we can use it to reach another program if we know its IPv6 address. IPv6 addresses have a

- The `::1` IPv6 address identifies the computer on which the current program is running.
- The `2001:6a8:308f:9:0:82ff:fe68:e520` IPv6 address identifies the computer serving the `https://beta.computer-networking.info` website.

An IPv6 address often identifies a computer and not a program running on the computer. In order to identify a specific progran

- The IPv6 address of the computer
- The port number identifying the program running on the computer

A program can use the `struct sockaddr_in6` to represent IPv6 socket addresses. The following program creates a `struct sockaddr_in6` that identifies the program that reserved the port number `55555` on the computer identified by the `::1` IPv6 address.

```
struct sockaddr_in6 peer_addr;                    // allocate the address on the␣
↪stack
memset(&peer_addr, 0, sizeof(peer_addr));         // fill the address with 0-bytes␣
↪to avoid garbage-values
peer_addr.sin6_family = AF_INET6;                 // indicate that the address is␣
↪an IPv6 address
peer_addr.sin6_port = htons(55555);               // indicate that the programm is␣
↪running on port 55555
inet_pton(AF_INET6, "::1", &peer_addr.sin6_addr); // indicate that the program is␣
↪running on the computer identified by the ::1 IPv6 address
```

Now, we have built everything we need to send a message to the remote program. The `create_socket_and_send_message` function below assembles all the building blocks we created until now in order to send the message `"hello"` to the program running on port `55555` on the computer identified by the `::1` IPv6 address.

```
int create_socket_and_send_message() {
    int sock = socket(AF_INET6, SOCK_DGRAM, 0);          // create a socket using␣
↪IPv6 addresses
    if (sock == -1) {
        printf("could not create the IPv6 SOCK_DGRAM socket, error: %s\n",␣
↪strerror(errno));
        return errno;
    }
```

(continues on next page)

(continued from previous page)

```
    struct sockaddr_in6 peer_addr;                          // allocate the address␣
↪on the stack
    memset(&peer_addr, 0, sizeof(peer_addr));               // fill the address with␣
↪0-bytes to avoid garbage-values
    peer_addr.sin6_family = AF_INET6;                       // indicate that the␣
↪address is an IPv6 address
    peer_addr.sin6_port = htons(55555);                     // indicate that the␣
↪programm is running on port 55555
    inet_pton(AF_INET6, "::1", &peer_addr.sin6_addr);       // indicate that the␣
↪program is running on the computer identified by the ::1 IPv6 address

    send_hello_to_peer(sock, (struct sockaddr *) &peer_addr, sizeof(peer_addr));    //
↪ use the send_hello_to_peer function that we defined previously
    close(sock);                                            // release the resources␣
↪used by the socket
    return 0;
}
```

Note that we can reuse our `send_hello_to_peer` function without any modification as we wrote it to handle any kind of sockets, including sockets using the IPv6 network protocol.

4.3.5 Endianness: exchanging integers between different computers

Besides character strings, some applications also need to exchange 16 bits and 32 bits fields such as integers. A naive solution would have been to send the 16- or 32-bits field as it is encoded in the host's memory. Unfortunately, there are different methods to store 16- or 32-bits fields in memory. Some CPUs store the most significant byte of a 16-bits field in the first address of the field while others store the least significant byte at this location. When networked applications running on different CPUs exchange 16 bits fields, there are two possibilities to transfer them over the transport service :

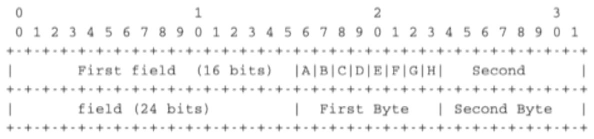
- send the most significant byte followed by the least significant byte
- send the least significant byte followed by the most significant byte

The first possibility was named *big-endian* in a note written by Cohen [Cohen1980] while the second was named *little-endian*. Vendors of CPUs that used *big-endian* in memory insisted on using *big-endian* encoding in networked applications while vendors of CPUs that used *little-endian* recommended the opposite. Several studies were written on the relative merits of each type of encoding, but the discussion became almost a religious issue [Cohen1980]. Eventually, the Internet chose the *big-endian* encoding, i.e. multi-byte fields are always transmitted by sending the most significant byte first, **RFC 791** refers to this encoding as the *network-byte order*. Most libraries[1] used to write networked applications contain functions to convert multi-byte fields from memory to the network byte order and the reverse.

Besides 16 and 32 bit words, some applications need to exchange data structures containing bit fields of various lengths. For example, a message may be composed of a 16 bits field followed by eight, one bit flags, a 24 bits field and two 8 bits bytes. Internet protocol specifications will define such a message by using a representation such as the one below. In this representation, each line corresponds to 32 bits and the vertical lines are used to delineate fields. The numbers above the lines indicate the bit positions in the 32-bits word, with the high order bit at position *0*.

The message mentioned above will be transmitted starting from the upper 32-bits word in network byte order. The first field is encoded in 16 bits. It is followed by eight one bit flags (*A-H*), a 24 bits field whose high order byte is shown in the first line and the two low order bytes appear in the second line followed by two one byte fields. This ASCII

[1] For example, the `htonl(3)` (resp. `ntohl(3)`) function the standard C library converts a 32-bits unsigned integer from the byte order used by the CPU to the network byte order (resp. from the network byte order to the CPU byte order). Similar functions exist in other programming languages.



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       First field  (16 bits)  |A|B|C|D|E|F|G|H|    Second     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       field (24 bits)         |  First Byte   | Second Byte   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 3: Message format

representation is frequently used when defining binary protocols. We will use it for all the binary protocols that are discussed in this book.

4.3.6 Exercises

Here are some exercises that will help you to learn how to use sockets.

During this course, you will be asked to implement a transport protocol running on Linux devices. To prepare yourself, try to implement the protocol described in the above tasks on your Linux personal machine. If you did these exercises correctly, most of your answers can be used as it (do not forget to include the required header files). In addition to the previously produced code, you will need

- to wrap the `create_and_send_message` in a `client` executable that can parse user arguments (the `getopt(3)` function might help) and appropriately call the wrapped function;
- to wrap the `recv_and_handle_message` server function in a `server` executable, similarly to what you have done with the `client` executable.

As an example, here is what you could have to invoke your programs.

```
# Let put the server on port 10000 (small port numbers are priviledged) and run it as␣
↪a daemon)
$ ./server :: 10000 &
# Let us call the client and request an addition result, as an int
$ ./client -op + ::1 10000 1 3 5 7 9
Result: 25
# Request now a multiplication, but returned as a string
$ ./client -op * -s ::1 10000 1 3 5 7 9
Result: 945
```

If you want to observe the packets exchanged over the network, use a packet dissector such as wireshark or tcpdump, listen the loopback interface (`lo`) and filter UDP packets using port 10000 (`udp.port==10000` in wireshark, `udp port 10000` with tcpdump).

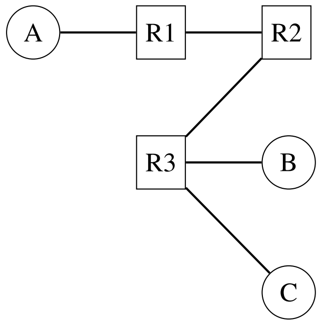
4.4 Building a network

4.4.1 Multiple-choice questions

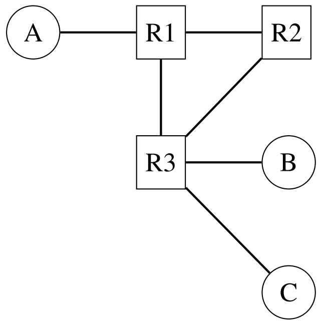
4.4.2 Open questions

1. In your daily life, you also use hierarchical and flat address spaces. Can you provide examples of these two types of addresses and discuss the benefits of using a hierarchical or flat addressing space in their particular context ?
2. The network below uses port forwarding with flat addresses. The network boots and all hosts start one after the other. Explain at each step how the packets are forwarded and how the port forwarding tables of the network

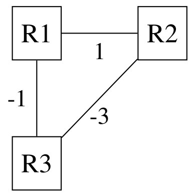
nodes are modified. Host *C* sends a packet to host *B*. Some time later, host *A* sends a packet to host *C*. Finally, host *B* sends a packet to host *A*.



3. Same question as above, but the network is modified as shown in the figure below.



4. Routing protocols used in data networks only use positive link weights. What would happen with a distance vector routing protocol in the network below that contains a negative link weight ?



5. When a network specialist designs a network, one of the problems that he needs to solve is to set the metrics the links in his network. In the USA, the Abilene network interconnects most of the research labs and universities. The figure below shows the topology of this network in 2009.

   In this network, assume that all the link weights are set to 1. What is the paths followed by a packet sent by the router located in *Los Angeles* to reach :

   - the router located in *New York*
   - the router located in *Washington* ?

   Is it possible to configure the link metrics so that the packets sent by the router located in *Los Angeles* to the routers located in respectively *New York* and *Washington* do not follow the same path ?

   Is it possible to configure the link weights so that the packets sent by the router located in *Los Angeles* to router located in *New York* follow one path while the packets sent by the router located in *New York* to the router located in *Los Angeles* follow a completely different path ?

   Assume that the routers located in *Denver* and *Kansas City* need to exchange lots of packets. Can you configure the link metrics such that the link between these two routers does not carry any packet sent by another router in the network ?

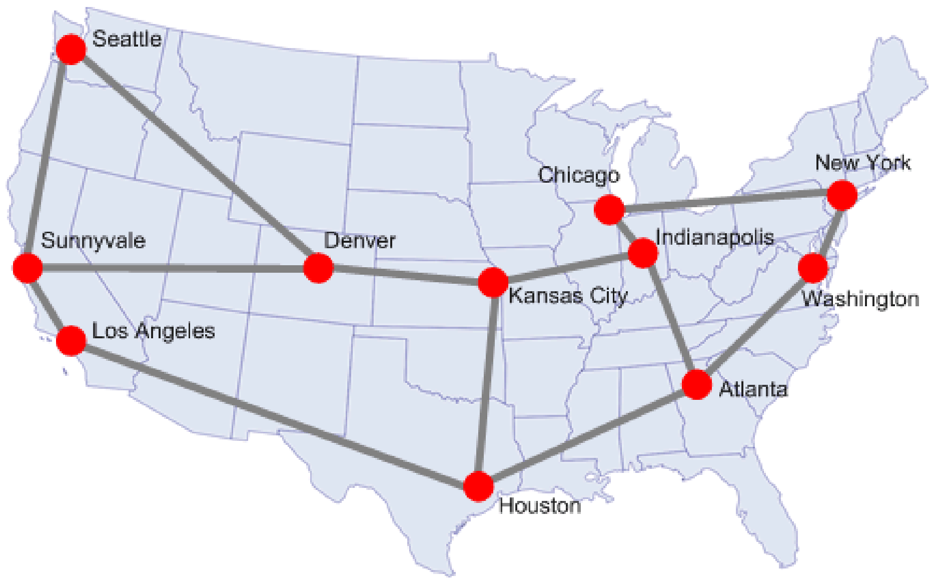
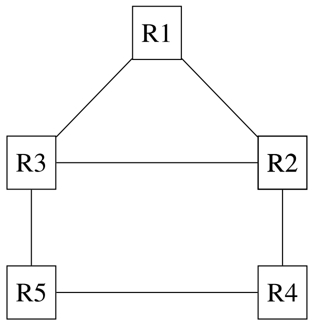


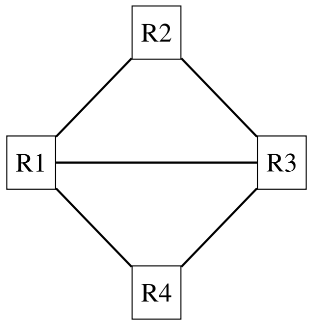
Fig. 4: The Abilene network

6. In the five nodes network shown below, can you configure the link metrics so that the packets sent by router *R5* to router *R1* use link *R3->R1* while the packets sent by router *R3* use links *R3->R2* and *R2->R1*?



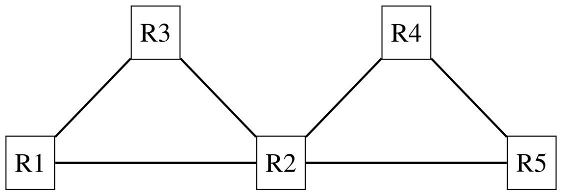
7. In the five nodes network shown above, can you configure the link weights so that the packets sent by router *R5* (resp. *R4*) follow the *R5->R3->R1* path (resp. *R4->R2->R3->R1*) ?

8. Consider the network shown in the figure below.



Assuming that the network uses source routing, what are the possible paths from *R1* to *R4* ?

9. Consider the network shown in the figure below.



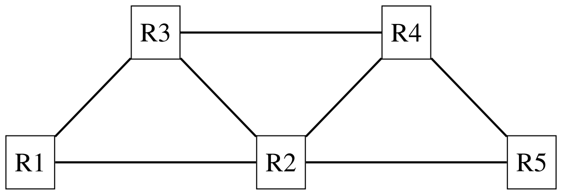
The network operator uses would like to have the following paths in this network :

- *R3->R2->R4->R5* and *R1->R2->R5*

Is it possible to achieve these paths and if so what are the required forwarding tables ?

Same question with virtual circuits.

10. Consider the network shown in the figure below.



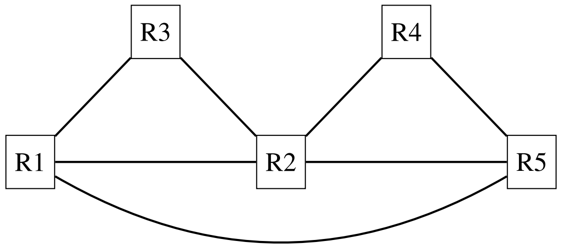
The network operator would like to use the following paths :

- *R1->R2->R4* and *R3->R2->R5->R4*

Are these paths possible with link-state or distance vector routing ? If yes, how do you configure the link weights. If no, explain your answer.

Same question with virtual circuits.

11. Consider the network shown in the figure below.



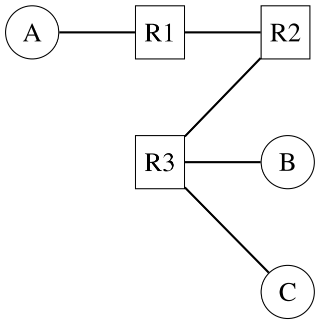
The network operator would like to use the following paths :

- *R1->R5->R4* and *R3->R2->R4*

Are these paths possible with link-state or distance vector routing ? If yes, how do configure the link weights. If no, explain your answer.

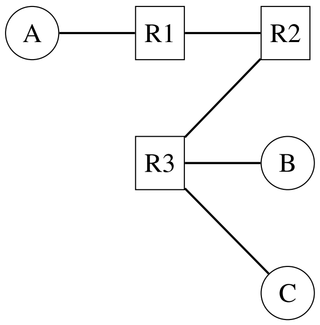
4.4.3 Discussion questions

1. The network below uses port forwarding tables. It has been running for several hours and all hosts have exchanged packets. What is the content of the port forwarding tables ?

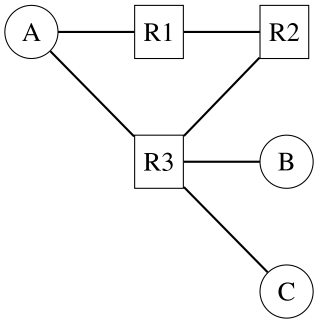


At this point, a new link is added between *R1* and *R3*. What happens for the forwarding of packets ?

2. The network below uses port forwarding tables. What happens if host *A* moves by removing its link with *R1* and replacing it with a link with *R3*? How should networks using port forwarding deal with such mobile hosts ?



3. Some hosts need to be multihomed, i.e. attached to two different network nodes as shown in the figure below.



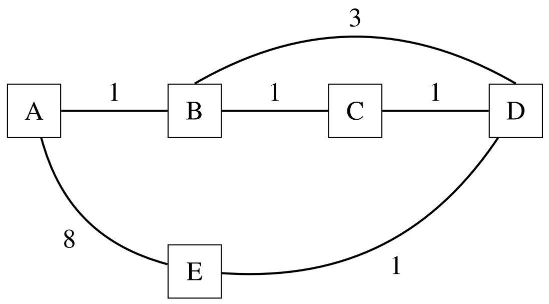
   Would this network work correctly with port-forwarding tables if :

   a. Host *A* uses the same flat address for both links.

   b. Host *A* uses a different flat address on each of its links

4. What are the advantages and drawbacks of flat addresses versus hierarchical addresses ?

5. Let us now consider the transient problems that mainly happen when the network topology changes. For this, consider the network topology shown in the figure below and assume that all routers use a distance vector protocol that uses split horizon.



If you compute the routing tables of all routers in this network, you would obtain a table such as the table below :

Destination	Routes on A	Routes on B	Routes on C	Routes on D	Routes on E
A	0	1 via A	2 via B	3 via C	4 via D
B	1 via B	0	1 via B	2 via C	3 via D
C	2 via B	1 via C	0	1 via C	2 via D
D	3 via B	2 via C	1 via D	0	1 via D
E	4 via B	3 via C	2 via D	1 via E	0

Distance vector protocols can operate in two different modes : *periodic updates* and *triggered updates*. *Periodic updates* is the default mode for a distance vector protocol. For example, each router could advertise its distance vector every thirty seconds. With the *triggered updates* a router sends its distance vector when its routing table changes (and periodically when there are no changes).

- Consider a distance vector protocol using split horizon and *periodic updates*. Assume that the link *B-C* fails. *B* and *C* update their local routing table but they will only advertise it at the end of their period. Select one ordering for the *periodic updates* and every time a router sends its distance vector, indicate the vector sent to each neighbor and update the table above. How many periods are required to allow the network to converge to a stable state ?
- Consider the same distance vector protocol, but now with *triggered updates*. When link *B-C* fails, assume that *B* updates its routing table immediately and sends its distance vector to *A* and *D*. Assume that both *A* and *D* process the received distance vector and that *A* sends its own distance vector, ... Indicate all the distance vectors that are exchanged and update the table above each time a distance vector is sent by a router (and received by other routers) until all routers have learned a new route to each destination. How many distance vector messages must be exchanged until the network converges to a stable state ?

6. Consider again the network shown above. After some time, link state routing converges and all routers compute the same routing tables as above.

   An important difference between OSPF and RIP is that OSPF routers flood link state packets that allow the other routers to recompute their own routing tables while RIP routers exchange distance vectors. Consider that link *B-C* fails and that router *B* is the first to detect the failure. At this point, *B* cannot reach anymore *C*, *D* and *E*. *C* cannot reach *B* and *A* anymore.

   Router *B* will flood its updated link state packet through the entire network and all routers will recompute their forwarding table. Upon reception of a link state packet, routers usually first flood the received link-state packet and then recompute their forwarding table. Assume that *B* is the first to recompute its forwarding table, followed by *D*, *A*, *C* and finally *E*.

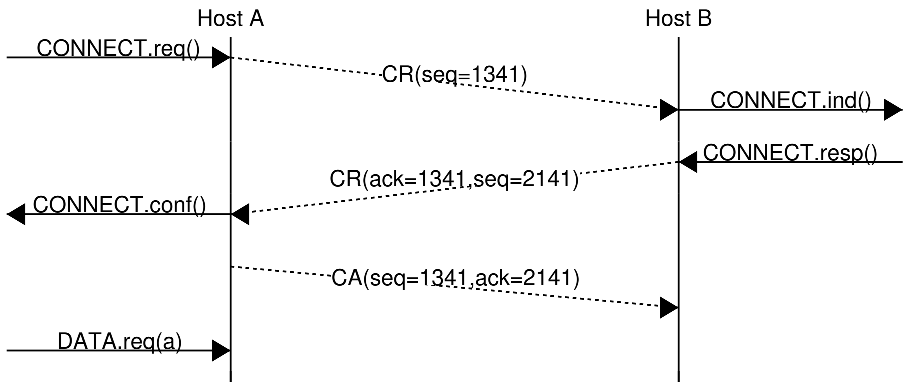
   After each update of a forwarding table, verify which pairs of routers are able to exchange packets. Provide your answer using a table similar to the one shown above.

4.5 Serving applications

4.5.1 Multiple choices questions

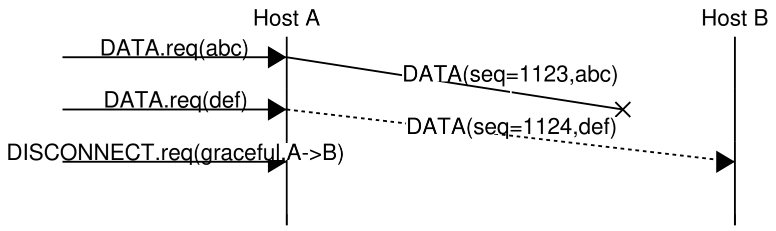
4.5.2 Open questions

1. Which are the mechanisms that should be included in a transport protocol providing an unreliable connectionless transport service that can detect transmission errors, but not correct them ?
2. A reliable connection oriented transport places a 32 bits sequence number inside the segment header to number the segments. This sequence number is incremented for each data segment. The connection starts as shown in the figure below :



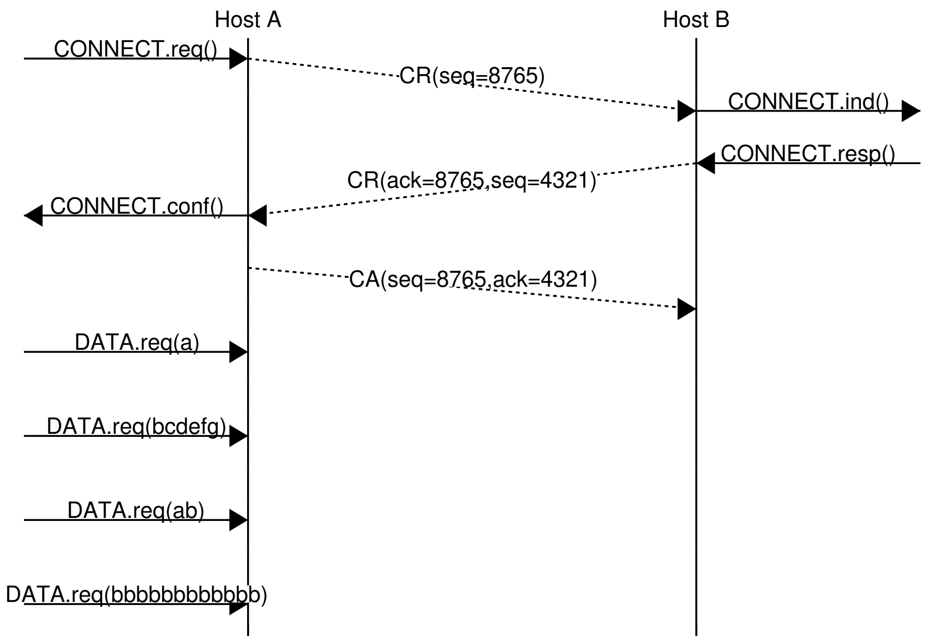
   Continue the connection so that *Host B* sends *Hello* as data and *Host A* replies by sending *Pleased to meet you*. After having received the response, *Host B* closes the connection gracefully and *Host A* does the same. Discuss on the state that needs to be maintained inside each host.

3. A transport connection that provides a message-mode service has been active for some time and all data has been exchanged and acknowledged in both directions. As in the exercise above, the sequence number is incremented after the transmission of each segment. At this time, *Host A* sends two DATA segments as shown in the figure below.

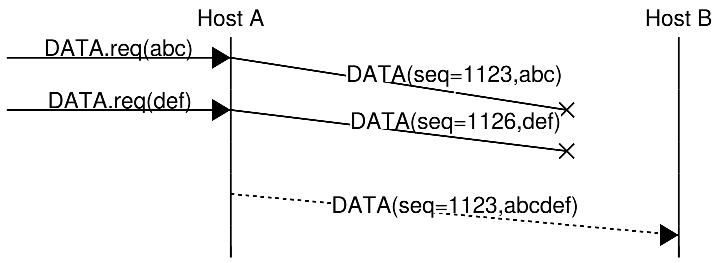


   What are the acknowledgments sent by *Host B*. How does *Host A* react and how does it terminate the connection ?

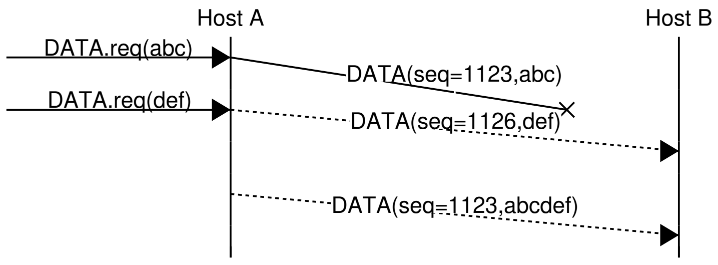
4. Consider a reliable connection-oriented transport protocol that provides the bytestream service. In this transport protocol, the sequence number that is placed inside each DATA segment reflects the position of the bytes in the bytestream. Considering the connection shown below, provide the DATA segments that are sent by *Host A* in response to the *DATA.request*, assuming that one segment is sent for each *DATA.request*.



5. Same question as above, but consider now that the transport protocol tries to send large DATA segments whenever possible. For this exercise, we consider that a DATA segment can contain up to 8 bytes of data in the payload. Do not forget to show the acknowledgments in your answer.
6. Consider a transport protocol that provides a reliable connection-oriented bytestream service. You observe the segments sent by a host that uses this protocol. Does the time-sequence diagram below reflects a valid implementation of this protocol ? Justify your answer.



7. In the above example, the two *DATA* segments were lost before arriving at the destination. Discuss the following scenario and explain how the receiver should react to the reception of the last *DATA* segment.



8. A network layer service guarantees that a packet will never live during more than 100 seconds inside the network. Consider a reliable connection-oriented transport protocol that places a 32 bits sequence number inside each segment. What is the maximum rate (in segments per second) at which is should sent data segments to prevent having two segments with the same sequence number inside the network ?

4.5.3 Practice

1. Amazon provides the S3 storage service where companies and researchers can store lots of information and perform computations on the stored information. Amazon allows users to send files through the Internet, but also by sending hard-disks. Assume that a 1 Terabyte hard-disk can be delivered within 24 hours to Amazon by courier service. What is the minimum bandwidth required to match the bandwidth of this courier service ?

4.5.4 Discussion questions

1. In the transport layer, the receive window advertised by a receiver can vary during the lifetime of the connection. What are the causes for these variations ?
2. A reliable connection-oriented protocol can provide a message-mode service or a byte stream service. Which of the following usages of the sequence numbers is the best suited for each of these services ?

a. DATA segments contain a sequence number that is incremented for each byte transmitted

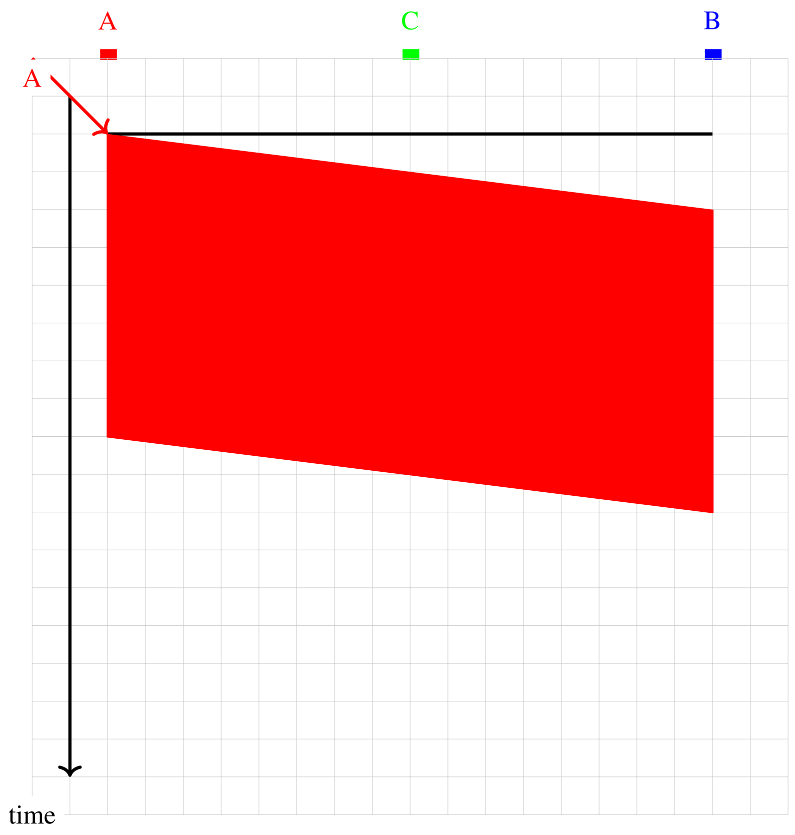
b. DATA segments contain a sequence number that is incremented for each DATA segment transmitted

3. Some transport protocols use 32 bits sequence numbers while others use 64 bits sequence number. What are the advantages and drawbacks of each approach ?
4. Consider a transport protocol that provides the bytestream service and uses 32 bits sequence number to represent the position of the first byte of the payload of DATA segments in the bytestream. How would you modify this protocol so that it can provide a message-mode service ? Consider first short messages that always fit inside a single segment. In a second step, discuss how you could support messages of unlimited size.
5. What is piggybacking and what are the benefits of this technique ?

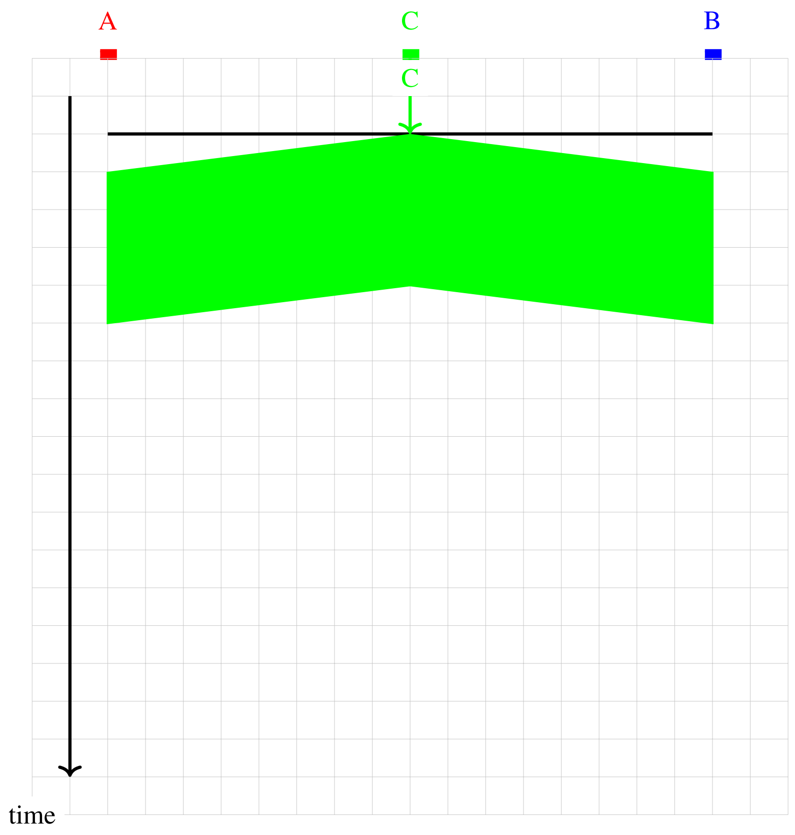
4.6 Sharing resources

4.6.1 Medium Access Control

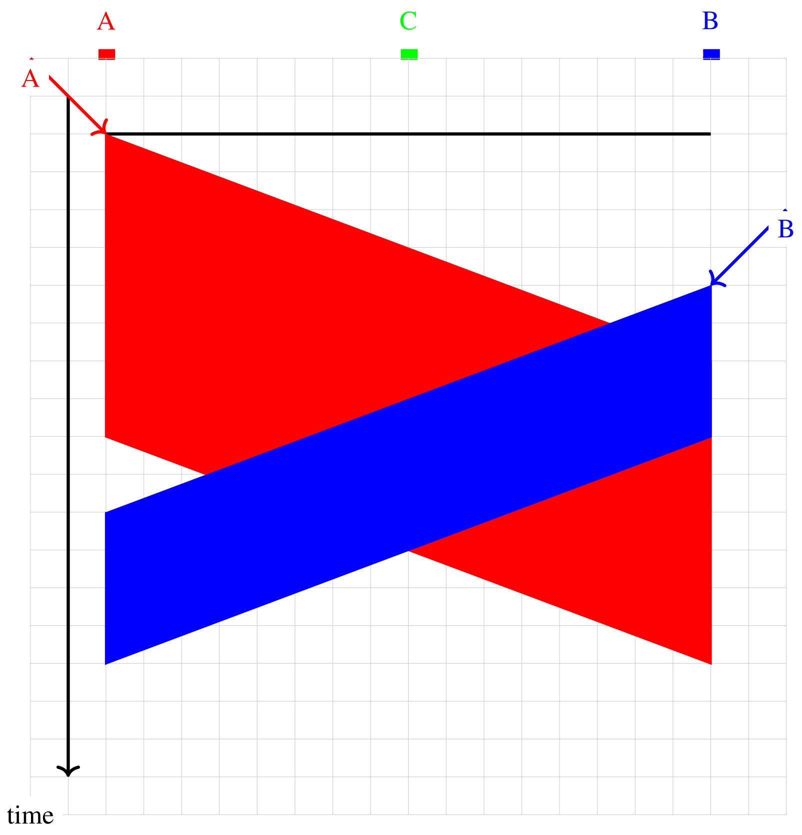
To understand the operation of Medium Access Control algorithms, it is often interesting to use a geometric representation of the transmission of frames on a shared medium. This representation is suitable if the communicating devices are attached to a single cable. Consider a simple scenario with a host connected at one end of a cable. For simplicity, let us consider a cable that has a length of one kilometer. Let us also consider that the propagation delay of the electrical signal is five microseconds per kilometer. The figure below shows the transmission of a 2000 bits frame at 100 Mbps by host A on the cable.



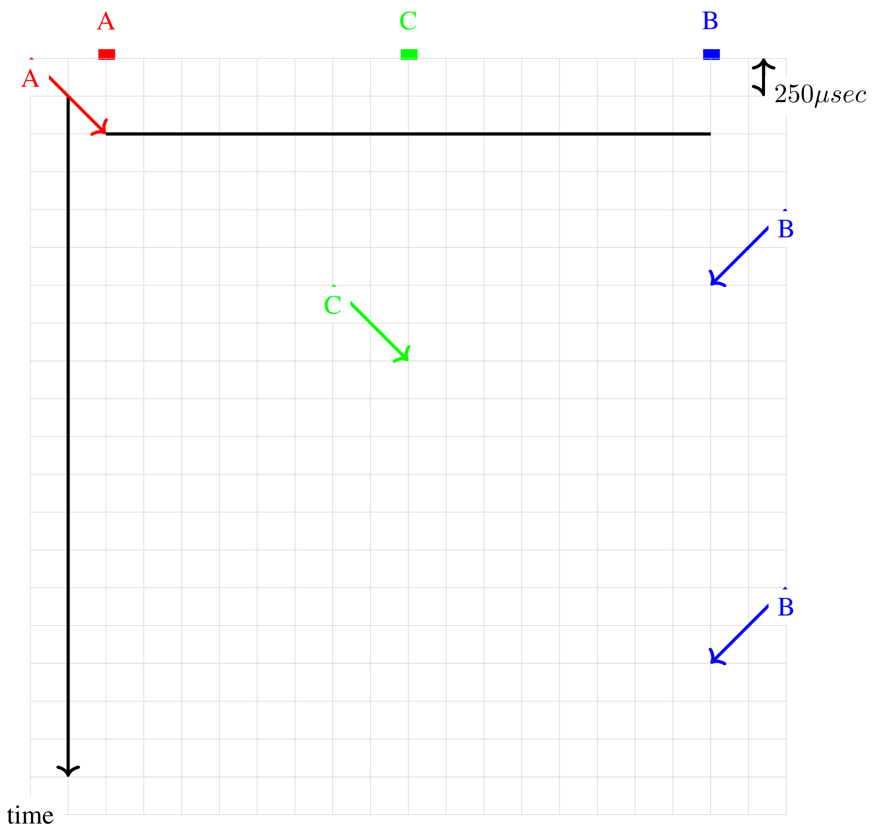
If the transmitting host is located at another position on the shared medium than one of the edges, then the geometrical pattern that represents the transmission of a frame is slightly different. If the transmitting host is placed in the middle of the cable, then the signal is transmitted in both directions on the cable. The figure below shows the transmission of one 100 bits frame at 100 Mbps by host C on the same cable.



In a shared medium, a collision may happen if two hosts transmit at almost the same time as shown in the example below.



1. Consider the following scenario for the ALOHA medium access control algorithm. Three hosts are attached to a one-kilometer long cable and transmit 1000 bits frames at 1 Mbps. Each arrow represents a request to transmit a frame on the corresponding host. Each square represents 250 microseconds in the figure. Represent all the transmitted frames and list the frames that collide.



2. Same question as above, but now consider that the hosts transmit 1000 bits frames at 100 Mbps. The cable has a length of 2 kilometers. C is in the middle of the cable. Each square in the figure below corresponds to 10 microseconds.

3. In ALOHA, the hosts rely on acknowledgments to detect whether their frame has been received correctly by the destination. Consider a network running at 100 Mbps where the host exchange 1000 bits frames and acknowledgments of 100 bits. Draw the frames sent by hosts A and B in the figure below. Assume that a square corresponds to 10 microseconds and that the cable has a length of 2 kilometers.

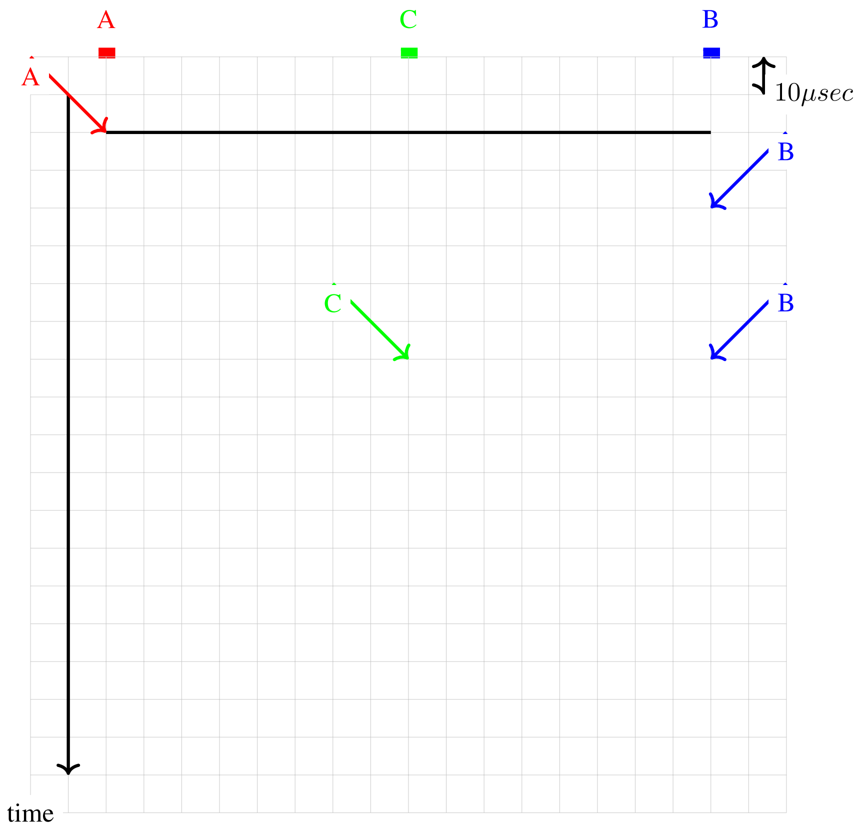
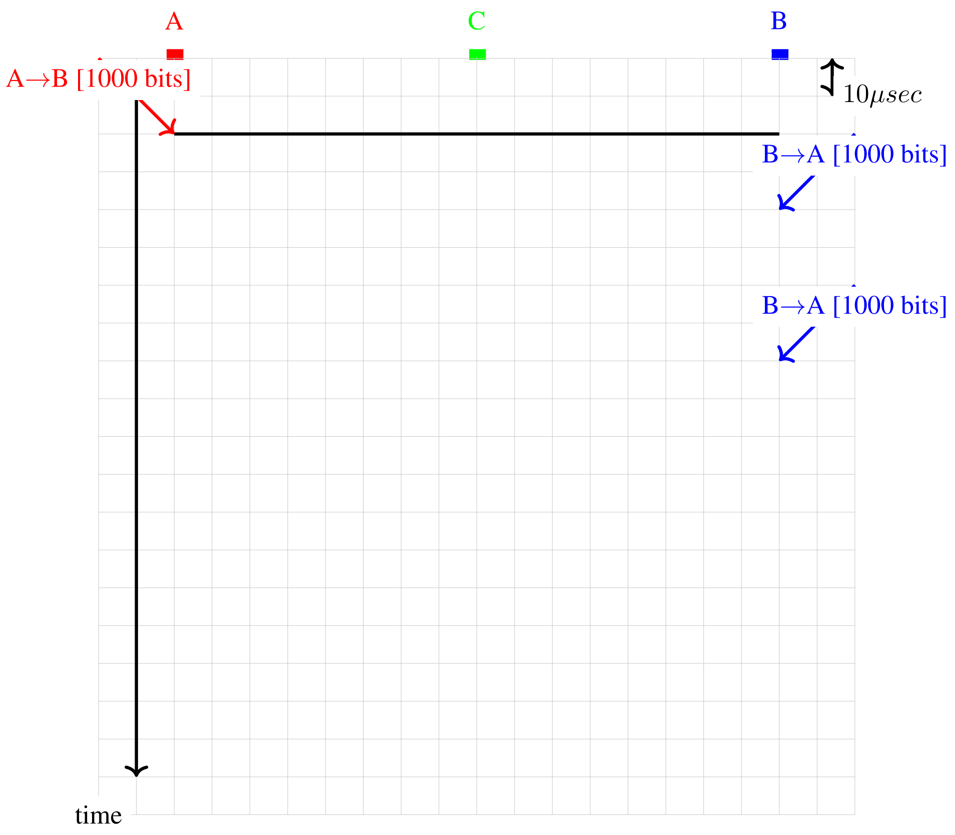


Fig. 5: :libs: positioning, matrix, arrows



4. Same question as above, but now assume that the retransmission timer of each host is set to 50 microseconds.

5. In practice, hosts transmit variable length frames. Consider a cable having a bandwidth of 100 Mbps and a length of 2 kilometers.

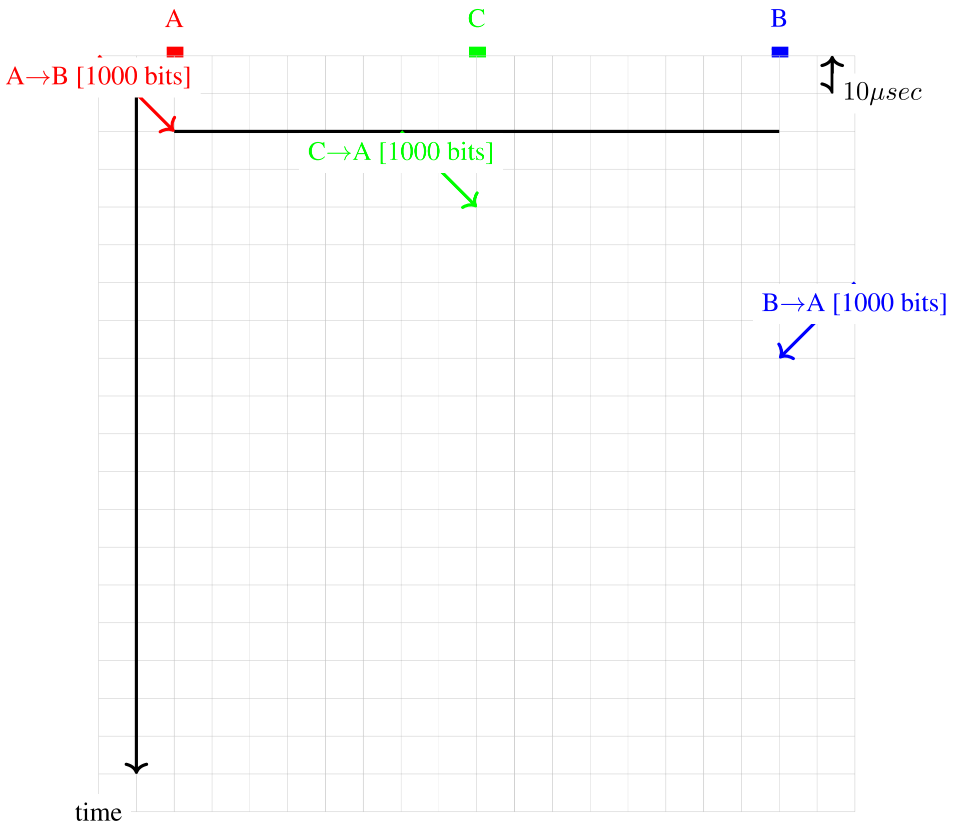
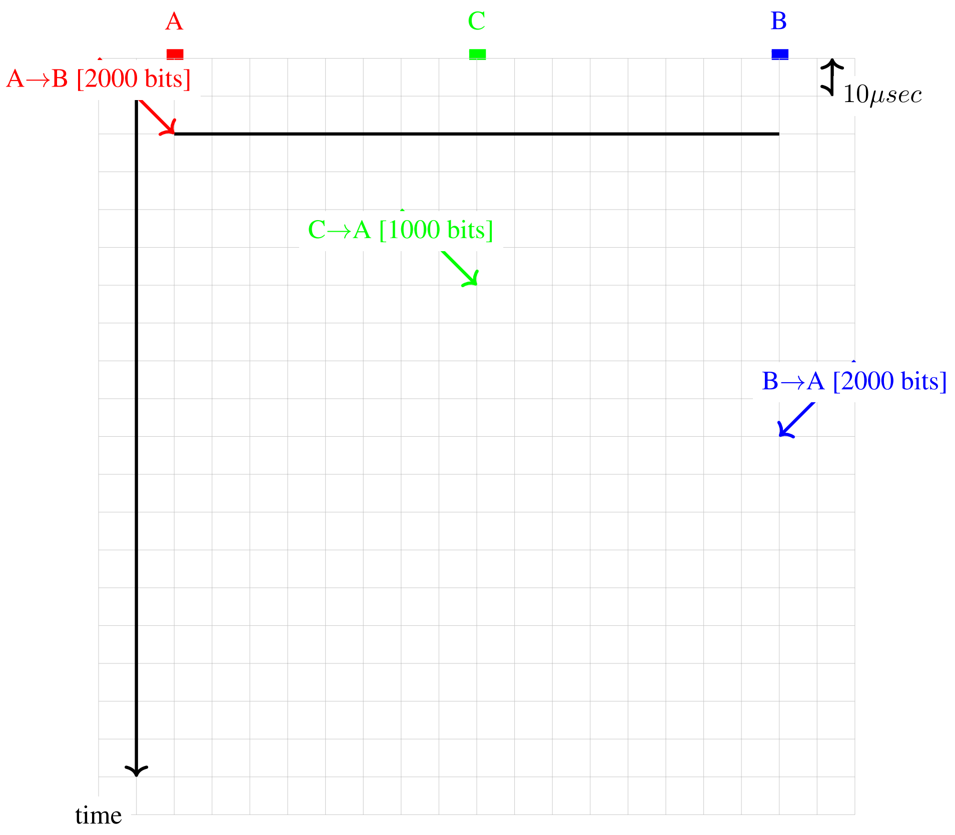
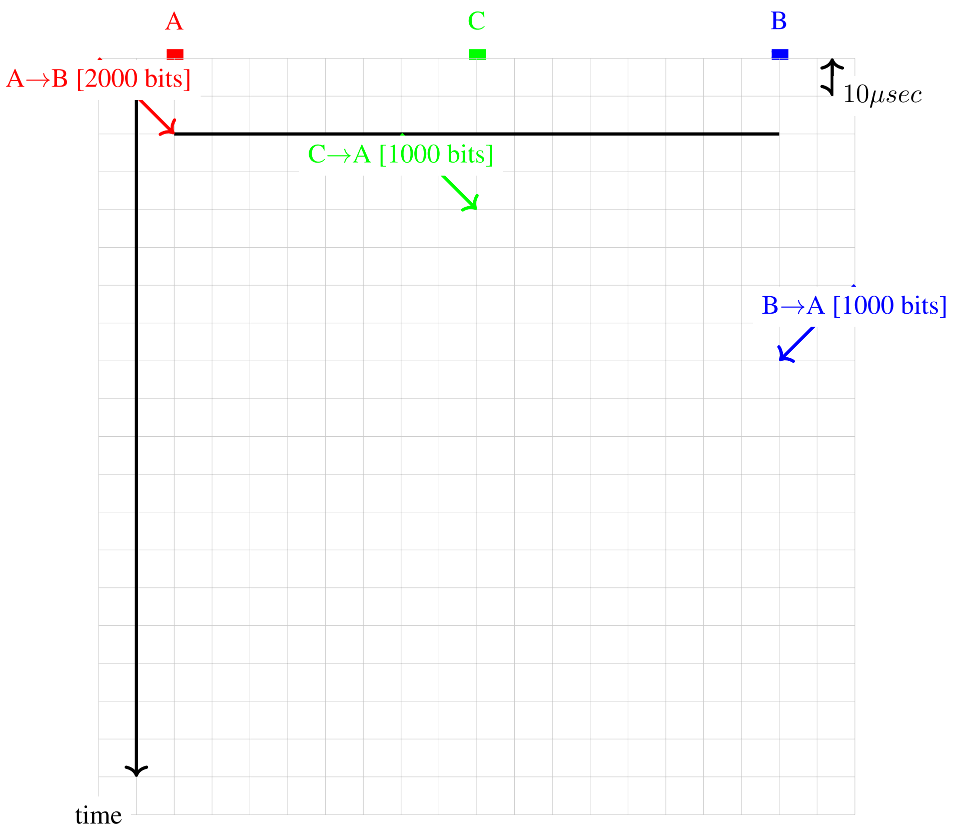


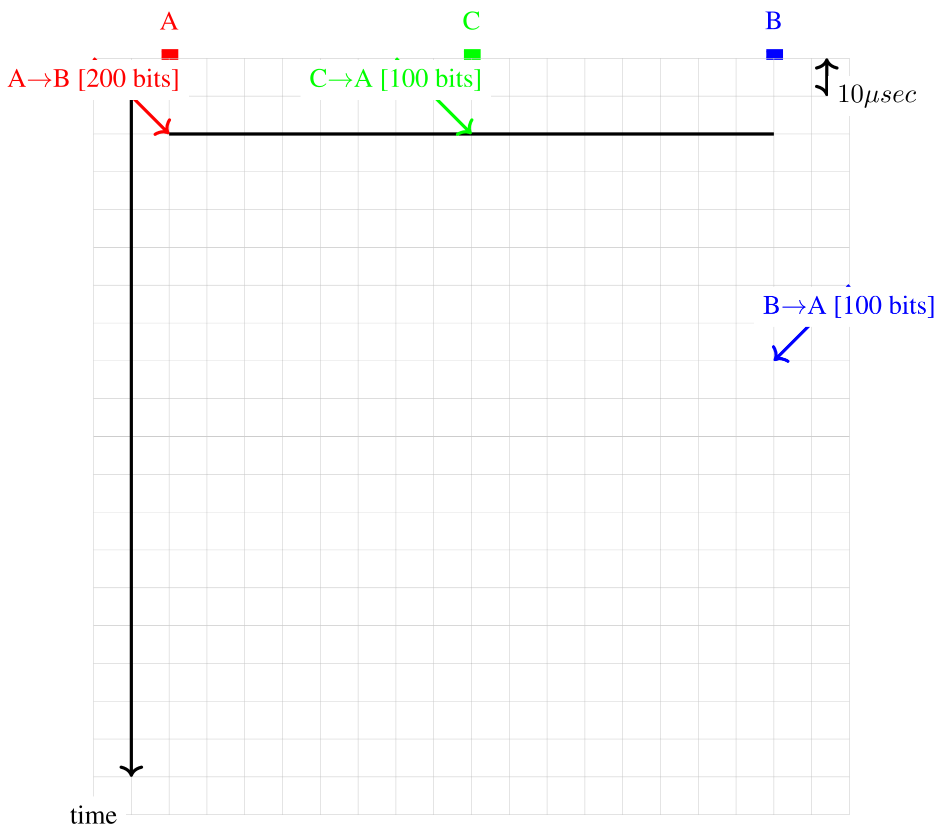
Fig. 6: :libs: positioning, matrix, arrows



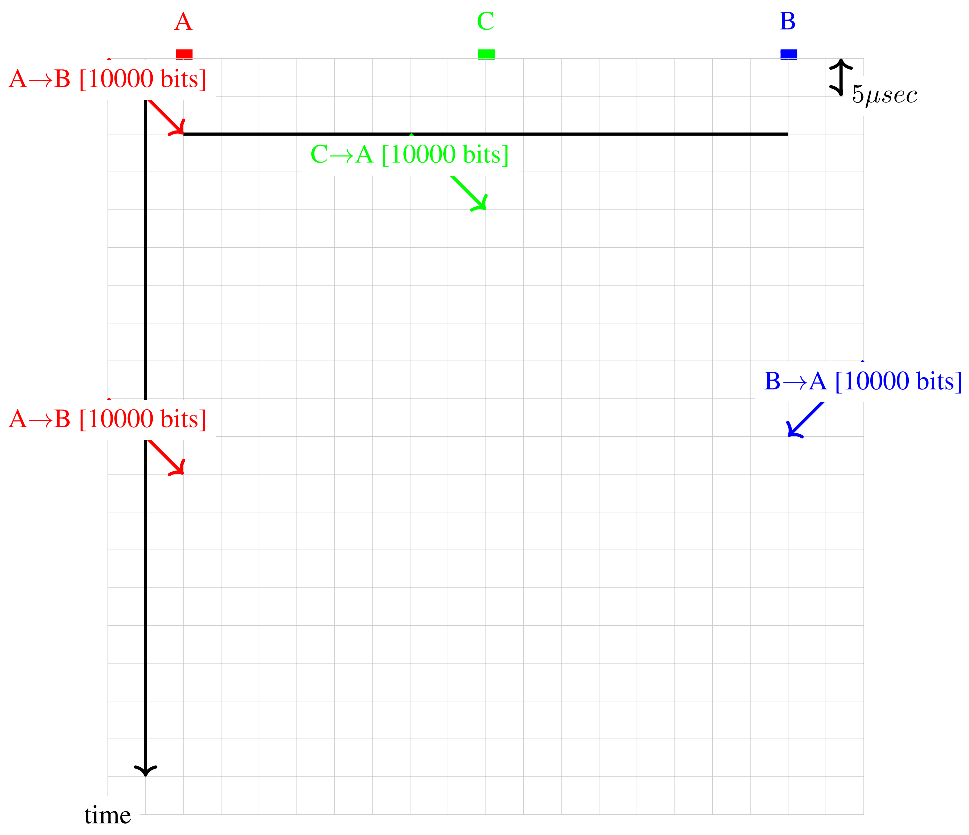
6. With CSMA, hosts need to listen to the communication channel before starting their transmission. Consider again a 2 kilometers long cable where hosts send frames at 100 Mbps. Show in the figure below the correct transmission of frames with CSMA.



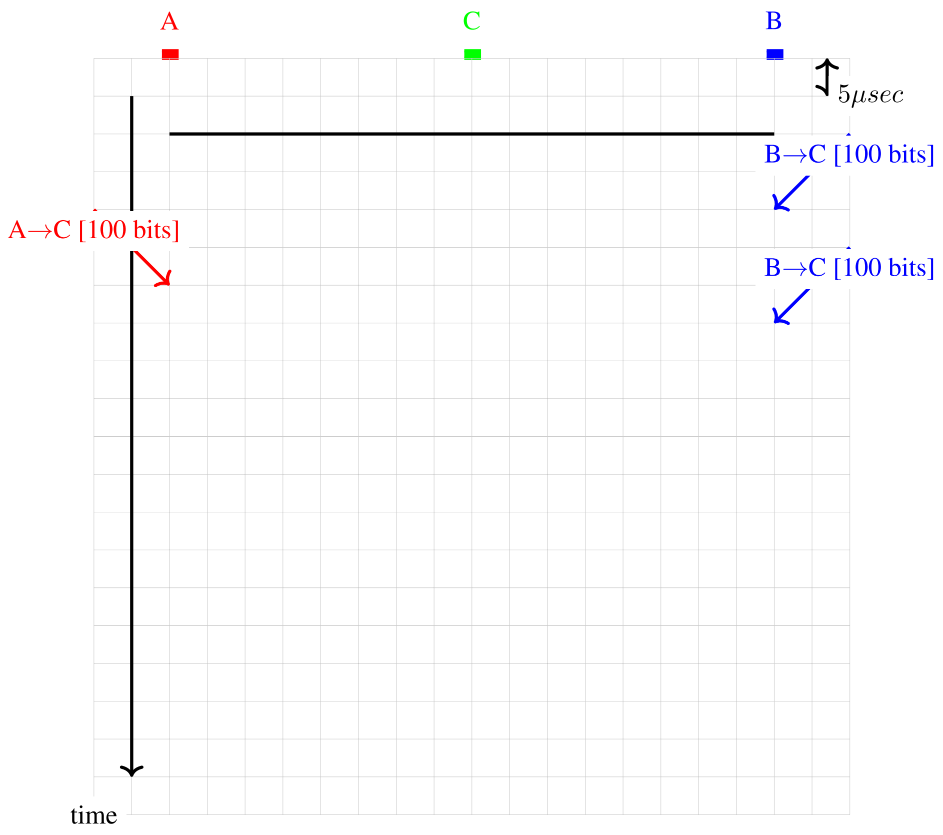
7. CSMA/CD does not use acknowledgments but instead assumes that each host can detect collisions by listening while transmitting. Consider a 2 kilometers long cable running at 10 Mbps. Show in the figure below the utilization of the communication channel and the collisions that would occur. For this exercise, do not attempt to retransmit the frames that have collided.



8. Consider again a network that uses CSMA/CD. This time, the bandwidth is set to 1 Gbps and the cable has a length of two kilometers. When a collision occurs, consider that the hosts B and C retransmit immediately while host A waits for the next slot.

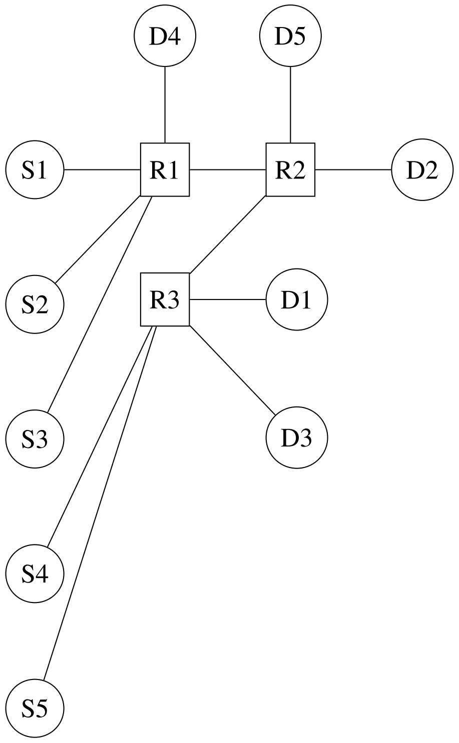


9. An important part of the CSMA/CD algorithm is the exponential backoff. To illustrate the operation of this algorithm, let us consider a cable that has a length of one kilometer. The bandwidth of the network is set to 10 Mbps. Assume that when a collision occurs, host A always selects the highest possible random delay according to the exponential backoff algorithm while host B always selects the shortest one. In this network, the slot time is equal to the time required to transmit 100 bits. We further assume that a host can detect collision immediately (i.e. as soon as the other frame arrives).

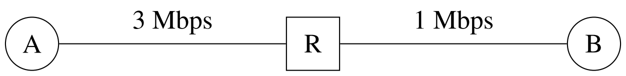


4.6.2 Fairness and congestion control

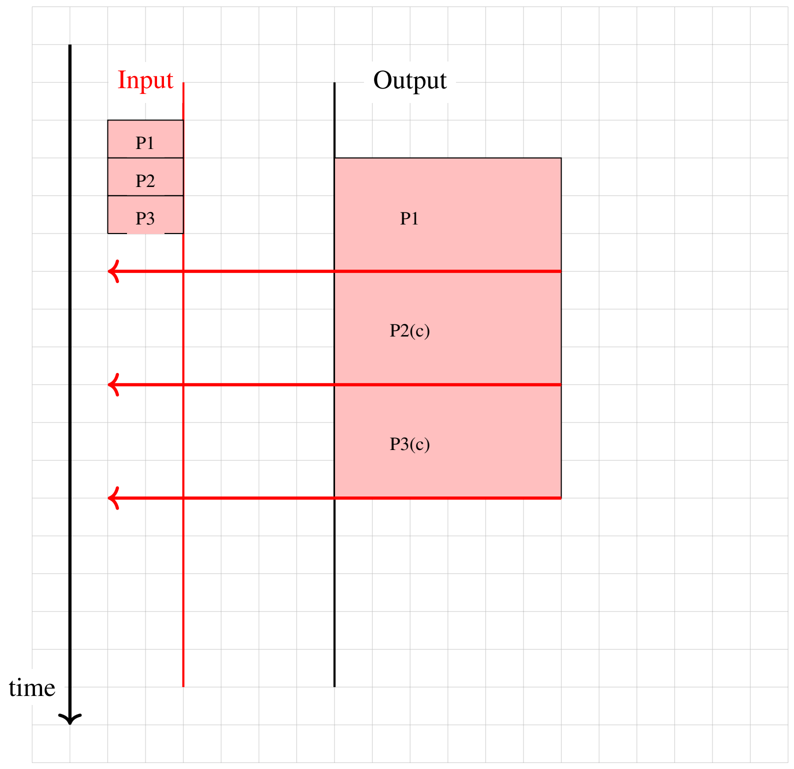
1. Consider the network below. Compute the max-min fair allocation for the hosts in this network assuming that nodes *Sx* always send traffic towards node *Dx*. Furthermore, link *R1*-*R2* has a bandwidth of 10 Mbps while link *R2*-*R3* has a bandwidth of 20 Mbps.



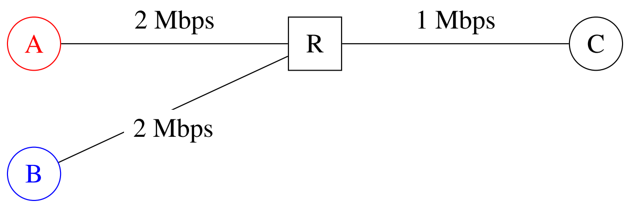
To understand congestion control algorithms, it can also be useful to represent the exchange of packets by using a graphical representation. As a first example, let us consider a very simple network composed of two hosts interconnected through a switch.

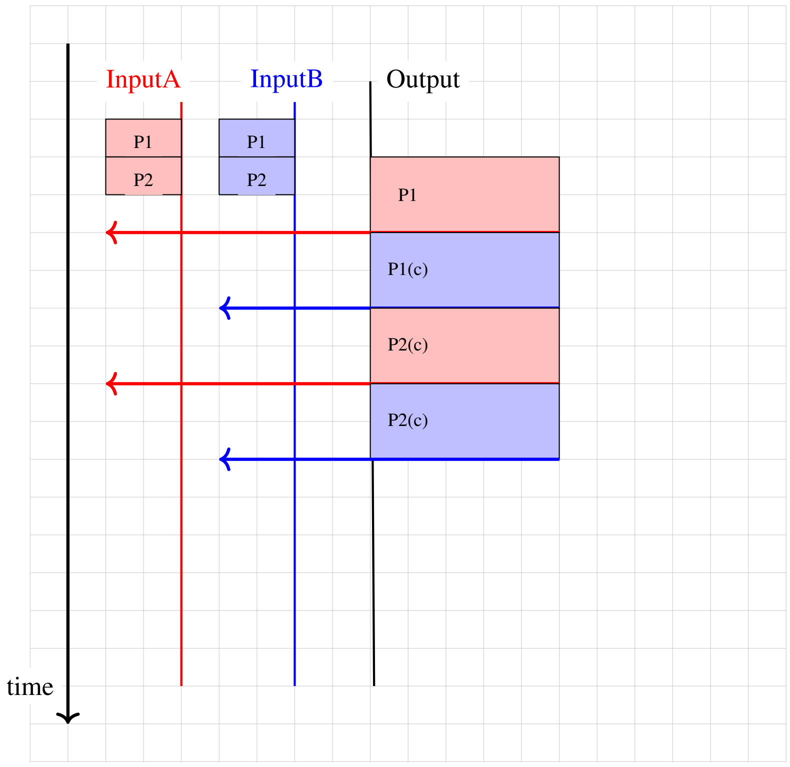


Suppose now that host A uses a window of three segments and sends these three segments immediately. The segments will be queued in the router before being transmitted on the output link and delivered to their destination. The destination will reply with a short acknowledgment segment. A possible visualization of this exchange of packets is represented in the figure below. We assume for this figure that the router marks the packets to indicate congestion as soon as its buffer is non-empty when its receives a packet on its input link. In the figure, a *(c)* sign is added to each packet to indicate that it has been explicitly marked.

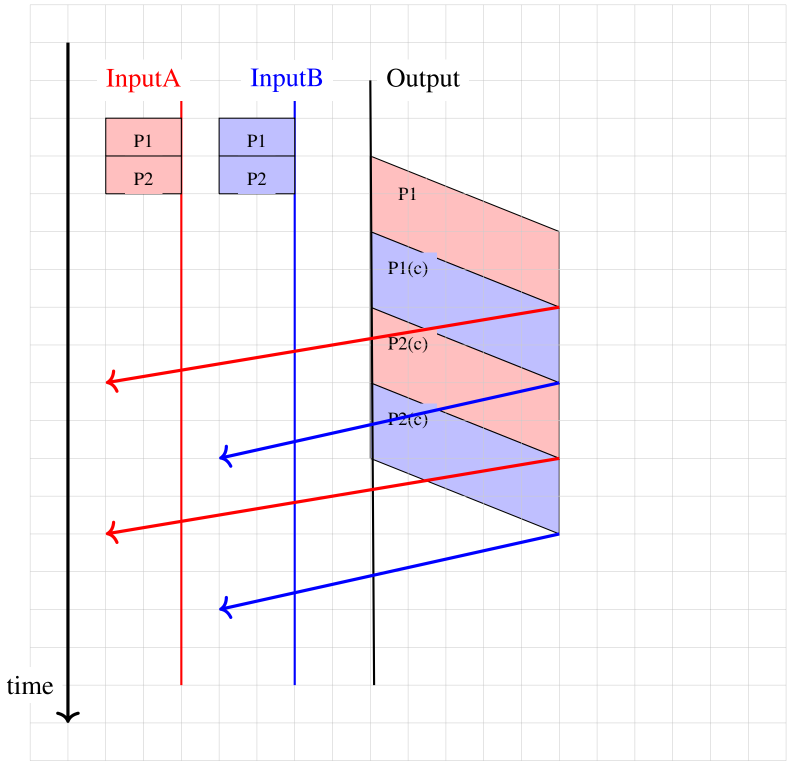


In practice, a router is connected to multiple input links. The figure below shows an example with two hosts.

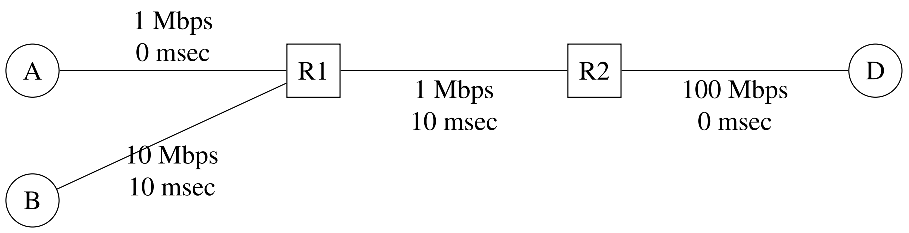




In general, the links have a non-zero delay. This is illustrated in the figure below where a delay has been added on the link between *R* and *C*.



2. Consider the network depicted in the figure below.



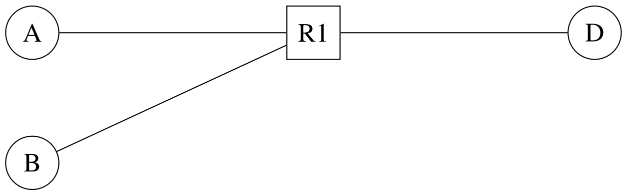
a. In this network, compute the minimum round-trip-time between *A* (resp. *B*) and *D*. Perform the computation if the hosts send segments containing 1000 bits.

b. How is the maximum round-trip-time influenced if the buffers of router *R1* store 10 packets ?

c. If hosts *A* and *B* send to *D* 1000 bits segments and use a sending window of four segments, what is the maximum throughput that they can achieve ?

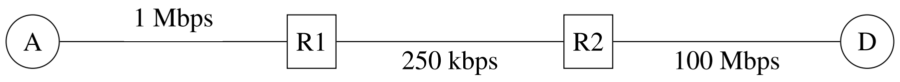
d. Assume now that *R1* is using round-robin scheduling instead of a FIFO buffer. One queue is used to store the packets sent by *A* and another for the packets sent by *B*. *A* sends one 1000 bits packet every second while *B* sends packets at 10 Mbps. What is the round-trip-time measured by each of these two hosts if each of the two queues of *R1* can store 5 packets ?

3. When analyzing the reaction of a network using round-robin schedulers, it is sometimes useful to consider that the packets sent by each source are equivalent to a fluid and that each scheduler acts as a tap. Using this analogy, consider the network below. In this network, all the links are 100 Mbps and host *B* is sending packets at 100 Mbps. If A sends at 1, 5, 10, 20, 30, 40, 50, 60, 80 and 100 Mbps, what is the throughput that destination *D* will receive from *A*. Use this data to plot a graph that shows the portion of the traffic sent by host *A* which is received by host *D*.



4. Compute the max-min fair bandwidth allocation in the network below.

5. Consider the simple network depicted in the figure below.



a. In this network, a 250 Kbps link is used between the routers. The propagation delays in the network are negligible. Host *A* sends 1000 bits long segments so that it takes one msec to transmit one segment on the *A*-*R1* link. Neglecting the transmission delays for the acknowledgments, what is the minimum round-trip time measured on host *A* with such segments ?

b. If host *A* uses a window of two segments and needs to transmit five segments of data. How long does the entire transfer lasts ?

c. Same question as above, but now host *A* uses the simple DECBIT congestion control mechanism and a maximum window size of four segments.

6. Consider the network depicted in the figure below.

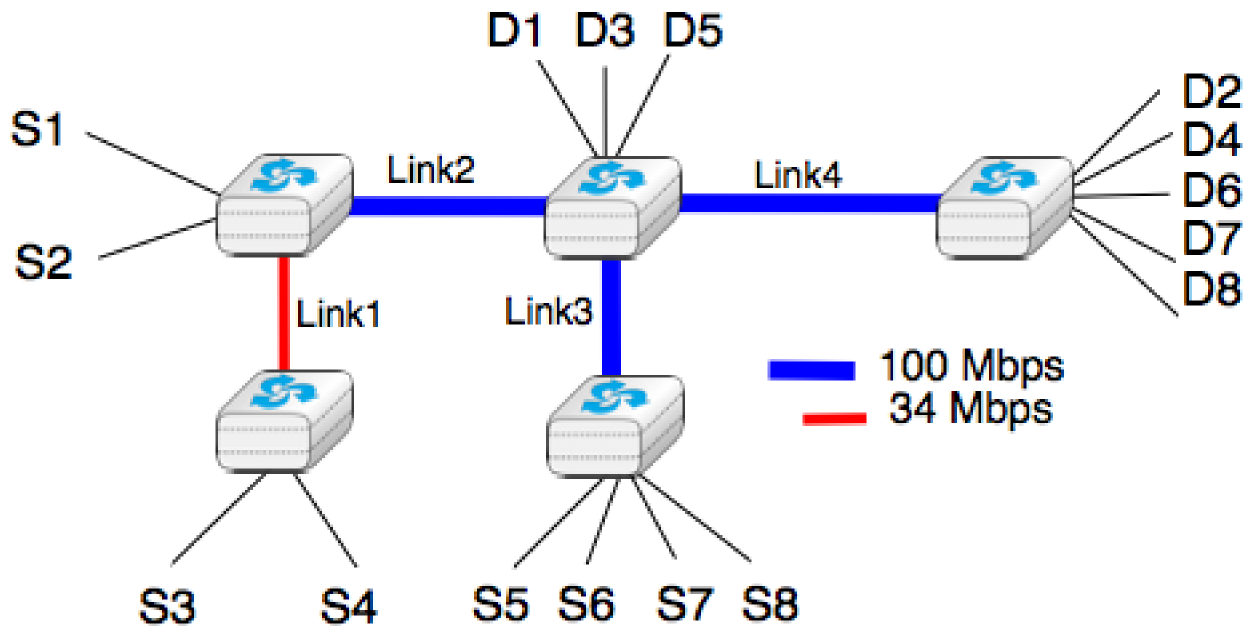
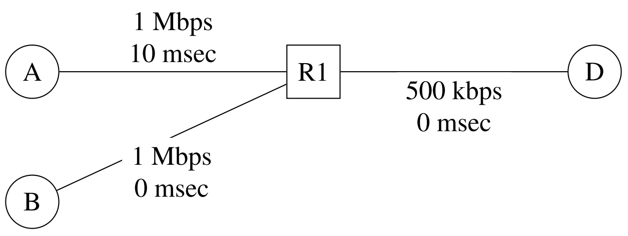


Fig. 7: Simple network topology



Hosts *A* and *B* use the simple congestion control scheme described in the book and router *R1* uses the DECBIT mechanism to mark packets as soon as its buffers contain one packet. Hosts *A* and *B* need to send five segments and start exactly at the same time. How long does each hosts needs to wait to receive the acknowledgment for its fifth segment ?

4.6.3 Discussion questions

1. In a deployed CSMA/CD network, would it be possible to increase or decrease the duration of the slotTime ? Justify your answer
2. Consider a CSMA/CD network that contains hosts that generate frames at a regular rate. When the transmission rate increases, the amount of collisions increases. For a given network load, measured in bits/sec, would the number of collisions be smaller, equal or larger with short frames than with long frames ?
3. Slotted ALOHA improves the performance of ALOHA by dividing the time in slots. However, this basic idea raises two interested questions. First how would you enforce the duration of these slots ? Second, should a slot include the time to transmit a data frame or the time to transmit a data frame and the corresponding acknowledgment ?
4. Like ALOHA, CSMA relies on acknowledgments to detect where a frame has been correctly received. When a host senses an idle channel, if should transmit its frame immediately. How should it react if it detects that another host is already transmitting ? Consider two options :

- the host continues to listen until the communication channel becomes free. It transmits as soon as the communication channel becomes free.
- the host stops to listen and waits for a random time before sensing again the communication channel to check whether it is free.

4.7 Application layer

4.7.1 The DNS

The Domain Name System (DNS) plays a key role in the Internet today as it allows applications to use fully qualified domain names (FQDN) instead of IPv4 or IPv6 addresses. When using the DNS, it is important to remember the role of the different types of DNS records.

Several software tools can be used to send queries to DNS servers. For this exercise, we use dig which is installed on most Unix/Linux systems.

A typical usage of dig is as follows:

```
dig @server -t type fqdn
```

where

- *server* is the IP address or the name of a DNS server or resolver
- *type* is the type of DNS record that is requested by the query such as *NS* for a nameserver, *A* for an IPv4 address, *AAAA* for an IPv6 address, *MX* for a mail relay, . . .
- *fqdn* is the fully qualified domain name being queried

dig also contains some additional parameters and flags that are described in the man page. Among these, the *+trace* flag allows to trace all requests that are sent when recursively contacting DNS servers.

1. What are the IP addresses of the resolvers that the *dig* implementation you are using relies on[1] ?
2. What are the nameservers that are responsible for the *info* top-level domain ? Is it possible to use IPv6 to query them ?
3. What is the IPv6 address that corresponds to *www.computer-networking.info* ? Which type of DNS query does *dig* send to obtain this information ?
4. When run without any parameter, *dig* queries one of the root DNS servers and retrieves the list of the names of all root DNS servers. For technical reasons, there are only 13 different root DNS servers. This information is also available as a text file from http://www.internic.net/zones/named.root. What are the IPv6 addresses of all these servers?
5. Assume now that you are residing in a network where there is no DNS resolver and that you need to perform your query manually starting from the DNS root.
    - Use *dig* to send a query to one of these root servers to find the IPv6 address of the DNS server(s) (NS record) responsible for the *org* top-level domain
    - Use *dig* to send a query to one of these DNS servers to find the IP address of the DNS server(s) (NS record) responsible for *root-servers.org*
    - Continue until you find the server responsible for *www.root-servers.org*
    - What is the lifetime associated to this IPv6 address ?

[1] On a Linux machine, the *Description* section of the *dig* man page tells you where *dig* finds the list of nameservers to query.

6. Perform the same analysis for a popular website such as *www.google.com*. What is the lifetime associated to the corresponding IPv6 address ? If you perform the same request several times, do you always receive the same answer ? Can you explain why a lifetime is associated to the DNS replies ?
7. Use *dig* to find the mail relays used by the *uclouvain.be* and *student.uclouvain.be* domains. What is the *TTL* of these records ? Can you explain the preferences used by the *MX* records. You can find more information about the MX records in **RFC 5321**.
8. When *dig* is run, the header section in its output indicates the *id* the DNS identifier used to send the query. Does your implementation of *dig* generates random identifiers ?

```
dig -t MX gmail.com

; <<>> DiG 9.4.3-P3 <<>> -t MX gmail.com
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 25718
```

9. A DNS implementation such as *dig*, and more importantly a name resolver such as bind or unbound, always checks that the received DNS reply contains the same identifier as the DNS request that it sent. Why is this so important ?
   - Imagine an attacker who is able to send forged DNS replies to, for example, associate *www.bigbank.com* to his own IP address. How could he attack a DNS implementation that
     - sends DNS requests containing always the same identifier
     - sends DNS requests containing identifiers that are incremented by one after each request
     - sends DNS requests containing random identifiers
10. The DNS protocol can run over UDP and over TCP. Most DNS servers prefer to use UDP because it consumes fewer resources on the server. However, TCP is useful when a large answer is expected. Compare *time dig +tcp* and *time dig* to query a root DNS server. Is it faster to receive an answer via TCP or via UDP ?

Besides *dig*, another way to analyze the DNS is to look at packet traces with tools such as wireshark or tcpdump These tools can capture packets in a network and also display and analyze their content. Wireshark provides a flexible Graphical User Interface that eases the analysis of the captured packets. The three questions below should help you to better understand the important fields of DNS messages.

The next three questions ask you to go one step further by predicting the values of specific fields in the DNS messages.

When a client requests the mapping of a domain name into an IP address to its local resolver, the resolver may need to query a large number of nameservers starting from the root nameserver. The three exercises below show packet traces collected while the resolver was resolving the following names: *www.example.com*, *www.google.com* and *www.computer-networking.info*. If you understand how the DNS operates, you should be able to correctly reorder those packet traces.

4.8 Internet email protocols

Many Internet protocols are ASCII-based protocols where the client sends requests as one line of ASCII text terminated by *CRLF* and the server replies with one of more lines of ASCII text. Using such ASCII messages has several advantages compared to protocols that rely on binary encoded messages

- the messages exchanged by the client and the server can be easily understood by a developer or network engineer by simply reading the messages
- it is often easy to write a small prototype that implements a part of the protocol

- it is possible to test a server manually by using telnet

Telnet is a protocol that allows to obtain a terminal on a remote server. For this, telnet opens a TCP connection with the remote server on port 23. However, most *telnet* implementations allow the user to specify an alternate port as *telnet hosts port* When used with a port number as parameter, *telnet* opens a TCP connection to the remote host on the specified port. *telnet* can thus be used to test any server using an ASCII-based protocol on top of TCP. Note that if you need to stop a running *telnet* session, `Ctrl-C` will not work as it will be sent by *telnet* to the remote host over the TCP connection. On many *telnet* implementations you can type `Ctrl-]` to freeze the TCP connection and return to the telnet interface.

1. Use your preferred email tool to send an email message to yourself containing a single line of text. Most email tools have the ability to show the *source* of the message, use this function to look at the message that you sent and the message that you received. Can you find an explanation for all the lines that have been added to your single line email ?
2. The TCP protocol supports 65536 different ports numbers. Many of these port numbers have been reserved for some applications. The official repository of the reserved port numbers is maintained by the Internet Assigned Numbers Authority (IANA) on http://www.iana.org/assignments/port-numbers[1]. Using this information, what is the default port number for the POP3 protocol ? Does it run on top of UDP or TCP ?
3. The Post Office Protocol (POP) is a rather simple protocol described in **RFC 1939**. POP operates in three phases. The first phase is the authorization phase where the client provides a username and a password. The second phase is the transaction phase where the client can retrieve emails. The last phase is the update phase where the client finalizes the transaction. What are the main POP commands and their parameters ? When a POP server returns an answer, how can you easily determine whether the answer is positive or negative ?
4. On smartphones, users often want to avoid downloading large emails over a slow wireless connection. How could a POP client only download emails that are smaller than 5 KBytes ?

4.9 The HyperText Transfer Protocol

An important difference between HTTP/1.0, HTTP/1.1 and HTTP/2.0 is their utilization of the underlying transport connections. Answer the three questions below to confirm that you understand the difference between these versions of the HTTP protocol.

1. System administrators who are responsible for web servers often want to monitor these servers and check that they are running correctly. As a HTTP server uses TCP on port 80, the simplest solution is to open a TCP connection on port 80 and check that the TCP connection is accepted by the remote host. However, as HTTP/1.x is an ASCII-based protocol, it is also very easy to write a small script that downloads a web page on the server and compares its content with the expected one. Use *telnet* or *ncat* to verify that a web server is running on host *www.computer-networking.info*[1].
2. Instead of using *telnet* on port 80, it is also possible to use a command-line tool such as curl. Use curl with the *–trace-ascii tracefile* option to store in *tracefile* all the information exchanged by curl when accessing the server.
   - What is the version of HTTP used by your version of curl ?
   - Can you explain the different headers placed by curl in the request ?
   - Can you explain the different headers found in the response ?
3. HTTP 1.1, specified in **RFC 2616**, forces the client to include the *Host:* header in all its requests. HTTP 1.0 does not define the *Host:* header, but most implementations support it. By using *telnet* and *curl* retrieve the first page of the https://www.computer-networking.info web server[2] by sending http requests with and without the *Host:* header. Explain the difference between the two.

[1] On Unix hosts, a subset of the port assignments is often placed in */etc/services*.
[1] The minimum command sent to a HTTP server is *GET / HTTP/1.0* followed by CRLF and a blank line.
[2] This syllabus is now hosted on a web server using HTTPS (port 443) instead of HTTP (port 80).

4. The headers sent in a HTTP request allow the client to provide additional information to the server. One of these headers is the *Accept-Language* header that allows indicating the preferred language of the client[3]. For example, *curl -HAccept-Language:en http://www.google.be* will send to *http://www.google.be* a HTTP request indicating English (*en*) as the preferred language. Does google provides a different page in French (*fr*) and Walloon (*wa*) ? Same question for *http://www.uclouvain.be* (given the size of the homepage, use `diff` to compare the different pages retrieved from *www.uclouvain.be*).
5. Compare the size of the http://www.yahoo.com and http://www.google.com web pages by downloading them with curl.
6. The ipvfoo extension on google chrome allows the user to visually detect whether a website is using IPv6 and IPv4, but also to see which web sites have been contacted when rendering a given web page. Some websites are distributed over several dozens of different servers. Can you find one ?
7. Some websites are reachable over both IPv4 and IPv6 while others are only reachable over IPv4[4]. You can use the *-6* (resp. *-4*) option to force curl to only use IPv6 (resp. IPv4). Verify that *www.computer-networking.info* is reachable over IPv6 and IPv4 and then check whether your university website already supports IPv6.
8. curl supports a huge number of options and parameters that are described in its man page Some of them allow you to force the utilization of a specific version of HTTP. These include *–http0.9*, *–http1.0*, *–http1.1* and *–http2*. Using the latter, verify whether your favorite website supports HTTP/2.0.

As for the DNS, besides using software tools that implement the HTTP protocols, it can also be useful to analyze packet traces with wireshark . The exercises below contain simple packet traces collected with different versions of the HTTP protocol.

4.10 TLS and ssh

One of the first motivations for the deployment of wide area networks such as the Internet was to enable researchers to connect to distant servers. For many years, these connections were carried out by using a simple application layer protocol such as telnet over a TCP connection. With telnet, all the characters typed by the user are sent in cleartext over the TCP connection. This implies that if someone is able to capture the packets transmitted over the network, he/she can collect sensitive information such as user names or passwords.

Fortunately, telnet is rarely used without TLS these days and system administrators usually prefer to deploy more secure protocols such as `ssh`.

The Transport Layer Security (TLS) protocol is now used by a wide range of applications, even if the most popular one is HTTPS. In the exercises below, you will analyze some of the features of TLS by looking at the packets that are exchanged over a TLS session.

[3] The list of available language tags can be found at http://www.iana.org/assignments/language-subtag-registry. Versions in other formats are available at http://www.langtag.net/registries.html. Additional information about the support of multiple languages in Internet protocols may be found in rfc5646.

[4] There are probably very few websites that only support IPv6 and not IPv4. If you find one, let us know by submitting a pull-request to change this exercise.

4.11 Analyzing packet traces

When debugging networking problems or to figure out performance problems, it is sometimes useful to capture the segments that are exchanged between two hosts and to analyze them.

Several packet trace analysis tools are available, either as commercial or open-source tools. These tools are able to capture all the packets exchanged on a link. Of course, capturing packets require administrator privileges. They can also analyze the content of the captured packets and display information about them. The captured packets can be stored in a file for offline analysis.

tcpdump is probably one of the most well known packet capture software. It is able to both capture packets and display their content. tcpdump is a text-based tool that can display the value of the most important fields of the captured packets. Additional information about tcpdump may be found in `tcpdump(1)`.

As an illustration, let us use tcpdump to analyze the packets exchanged while executing the following command on a Linux host:

```
curl -6 http://www.computer-networking.info
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="https://www.computer-networking.info/">here</a>.</
↪p>
</body></html>
```

The `-6` parameter passed to curl forces the utilization of IPv6. curl returns an HTML page that indicates that https must be used instead of http to access this web site.

A first solution to analyze this trace is to use tcpdump on the command line. The *-n* disables the reverse DNS lookups that tcpdump does by default for all IP addresses. The *-r* argument is the name of the file contained the captured packets. The trace starts with the DNS request. This request was sent over IPv4 which is the default on this host. tcpdump indicates the query and the response returned by the local DNS resolver.

```
tcpdump -n -r simple-trace.pcap
reading from file simple-trace.pcap, link-type LINUX_SLL (Linux cooked)
15:50:39.827908 IP 130.104.229.28.38417 > 130.104.230.68.53: 38133+ AAAA? www.
↪computer-networking.info. (46)
15:50:39.828793 IP 130.104.230.68.53 > 130.104.229.28.38417: 38133 3/4/8 CNAME␣
↪cnp3book.info.ucl.ac.be., CNAME cnp3.info.ucl.ac.be., AAAA␣
↪2001:6a8:308f:8:0:82ff:fe68:e48c (385)
```

The following three lines of the tcpdump output correspond to TCP's three-way handshake. There are several interesting points to note in this output. First, `Flags [S]` indicates that the *SYN* flag was set in the first and second segments. In this first segment, tcpdump indicates the initial sequence number (`2681184541`). In the second segment, tcpdump indicates both the initial sequence number (`3804204915`) and the acknowledgment number (`2681184542`). Both segments contain TCP options. Starting in the third segment, tcpdump shows relative sequence numbers. Thus, the acknowledgment that you observe in the third segment is an acknowledgment for the *SYN* returned by the server.

```
15:50:39.829353 IP6 2001:6a8:308f:9:0:82ff:fe68:e51c.34598 >␣
↪2001:6a8:308f:8:0:82ff:fe68:e48c.80: Flags [S], seq 2681184541, win 28800, options␣
↪[mss 1440,sackOK,TS val 609493767 ecr 0,nop,wscale 7], length 0
15:50:39.830043 IP6 2001:6a8:308f:8:0:82ff:fe68:e48c.80 >␣
↪2001:6a8:308f:9:0:82ff:fe68:e51c.34598: Flags [S.], seq 3804204915, ack 2681184542,␣
↪win 28560, options [mss 1440,sackOK,TS val 1646122290 ecr 609493767,nop,wscale 7],␣
↪length 0
```

(continues on next page)

(continued from previous page)

```
15:50:39.830074 IP6 2001:6a8:308f:9:0:82ff:fe68:e51c.34598 >␣
↪2001:6a8:308f:8:0:82ff:fe68:e48c.80: Flags [.], ack 1, win 225, options [nop,nop,TS␣
↪val 609493768 ecr 1646122290], length 0
```

```
15:50:39.830258 IP6 2001:6a8:308f:9:0:82ff:fe68:e51c.34598 >␣
↪2001:6a8:308f:8:0:82ff:fe68:e48c.80: Flags [P.], seq 1:93, ack 1, win 225, options␣
↪[nop,nop,TS val 609493768 ecr 1646122290], length 92: HTTP: GET / HTTP/1.1
15:50:39.830750 IP6 2001:6a8:308f:8:0:82ff:fe68:e48c.80 >␣
↪2001:6a8:308f:9:0:82ff:fe68:e51c.34598: Flags [.], ack 93, win 224, options [nop,
↪nop,TS val 1646122290 ecr 609493768], length 0
```

The two lines above correspond to the request sent by the client and the acknowledgment returned by the server. Note that the first byte sent by the client has *1* as relative sequence number. In this example, the HTTP request has a total length of 92 bytes. This request is immediately acknowledged by the server.

The server then sends its response, which fits inside a single segment. The client acknowledges the reception of this segment.

```
15:50:39.841255 IP6 2001:6a8:308f:8:0:82ff:fe68:e48c.80 >␣
↪2001:6a8:308f:9:0:82ff:fe68:e51c.34598: Flags [P.], seq 1:458, ack 93, win 224,␣
↪options [nop,nop,TS val 1646122301 ecr 609493768], length 457: HTTP: HTTP/1.1 302␣
↪Found
15:50:39.841270 IP6 2001:6a8:308f:9:0:82ff:fe68:e51c.34598 >␣
↪2001:6a8:308f:8:0:82ff:fe68:e48c.80: Flags [.], ack 458, win 234, options [nop,nop,
↪TS val 609493779 ecr 1646122301], length 0
```

The TCP connection is then closed by exchanging three segments, the first two having the *FIN* flag set.

```
15:50:39.843259 IP6 2001:6a8:308f:9:0:82ff:fe68:e51c.34598 >␣
↪2001:6a8:308f:8:0:82ff:fe68:e48c.80: Flags [F.], seq 93, ack 458, win 234, options␣
↪[nop,nop,TS val 609493781 ecr 1646122301], length 0
15:50:39.862246 IP6 2001:6a8:308f:8:0:82ff:fe68:e48c.80 >␣
↪2001:6a8:308f:9:0:82ff:fe68:e51c.34598: Flags [F.], seq 458, ack 94, win 224,␣
↪options [nop,nop,TS val 1646122317 ecr 609493781], length 0
15:50:39.862265 IP6 2001:6a8:308f:9:0:82ff:fe68:e51c.34598 >␣
↪2001:6a8:308f:8:0:82ff:fe68:e48c.80: Flags [.], ack 459, win 234, options [nop,nop,
↪TS val 609493800 ecr 1646122317], length 0
```

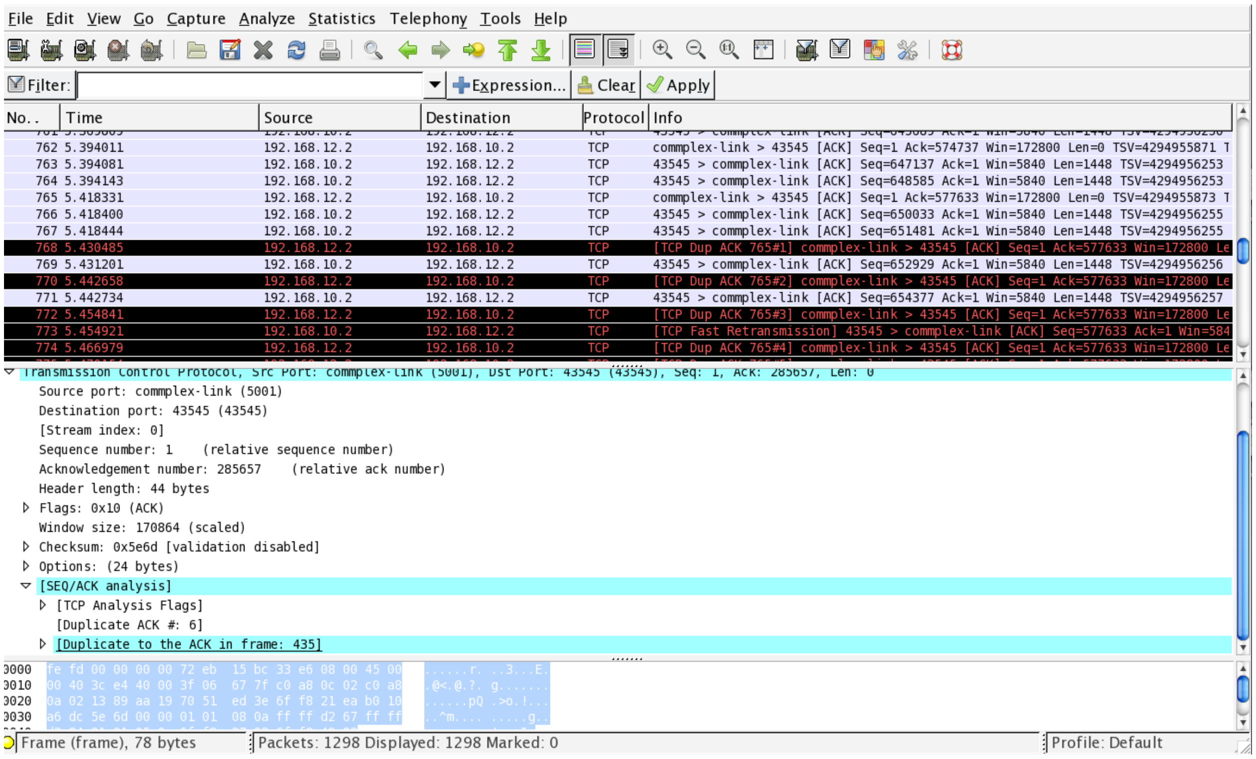
tcpdump can provide more detailed information about the packets by using the *-v* or *-vv* option.

wireshark is more recent than tcpdump. It evolved from the ethereal packet trace analysis software. It can be used as a text tool like tcpdump. For a TCP connection, wireshark can provide almost the same output as tcpdump. The main advantage of wireshark is that it also includes a graphical user interface that allows performing various types of analysis on a packet trace.

The wireshark window is divided in three parts. The top part of the window is a summary of the first packets from the trace. By clicking on one of the lines, you can show the detailed content of this packet in the middle part of the window. The middle of the window allows you to inspect all the fields of the captured packet. The bottom part of the window is the hexadecimal representation of the packet, with the field selected in the middle window being highlighted.

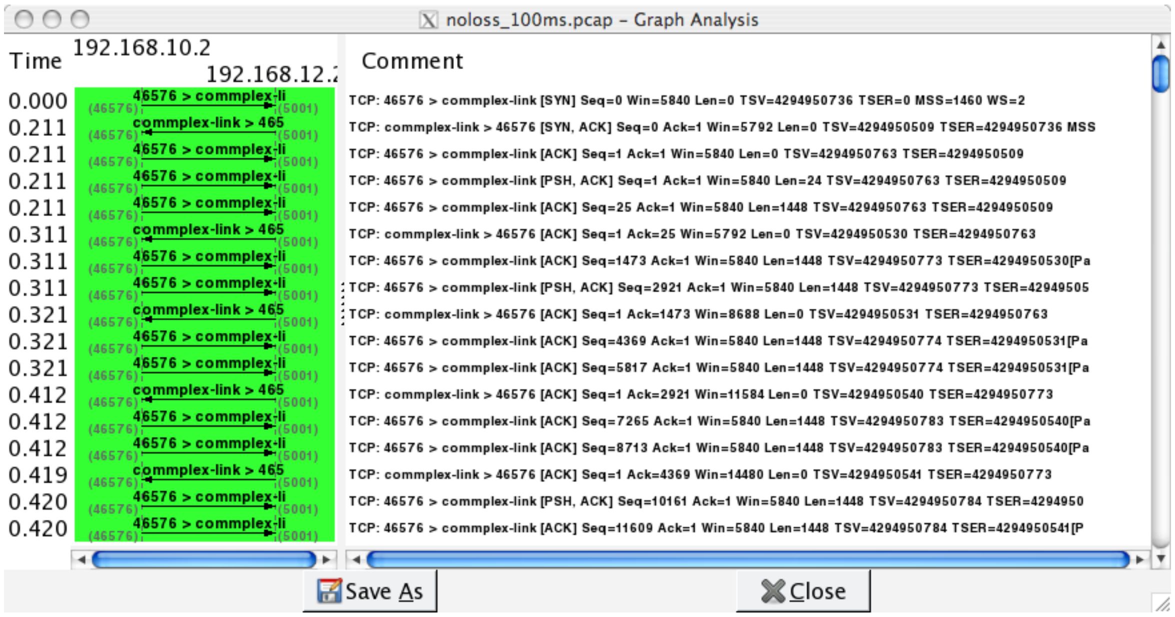
wireshark is very good at displaying packets, but it also contains several analysis tools that can be very useful. The first tool is *Follow TCP stream*. It is part of the *Analyze* menu and allows you to reassemble and display all the payload exchanged during a TCP connection. This tool can be useful if you need to analyze for example the commands exchanged during an HTTP or SMTP session.

The second tool is the flow graph that is part of the *Statistics* menu. It provides a time sequence diagram of the packets exchanged with some comments about the packet contents. See below for an example.


File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression...
Clear
Apply

```
No. . Time       Source         Destination    Protocol Info
  762 5.394011   192.168.12.2   192.168.10.2   TCP      commplex-link > 43545 [ACK] Seq=1 Ack=574737 Win=172800 Len=0 TSV=4294955871 T
  763 5.394081   192.168.10.2   192.168.12.2   TCP      43545 > commplex-link [ACK] Seq=647137 Ack=1 Win=5840 Len=1448 TSV=4294956253
  764 5.394143   192.168.10.2   192.168.12.2   TCP      43545 > commplex-link [ACK] Seq=648585 Ack=1 Win=5840 Len=1448 TSV=4294956253
  765 5.418331   192.168.12.2   192.168.10.2   TCP      commplex-link > 43545 [ACK] Seq=1 Ack=577633 Win=172800 Len=0 TSV=4294955873 T
  766 5.418400   192.168.10.2   192.168.12.2   TCP      43545 > commplex-link [ACK] Seq=650033 Ack=1 Win=5840 Len=1448 TSV=4294956255
  767 5.418444   192.168.10.2   192.168.12.2   TCP      43545 > commplex-link [ACK] Seq=651481 Ack=1 Win=5840 Len=1448 TSV=4294956255
  768 5.430485   192.168.12.2   192.168.10.2   TCP      [TCP Dup ACK 765#1] commplex-link > 43545 [ACK] Seq=1 Ack=577633 Win=172800 Le
  769 5.431201   192.168.10.2   192.168.12.2   TCP      43545 > commplex-link [ACK] Seq=652929 Ack=1 Win=5840 Len=1448 TSV=4294956256
  770 5.442658   192.168.12.2   192.168.10.2   TCP      [TCP Dup ACK 765#2] commplex-link > 43545 [ACK] Seq=1 Ack=577633 Win=172800 Le
  771 5.442734   192.168.10.2   192.168.12.2   TCP      43545 > commplex-link [ACK] Seq=654377 Ack=1 Win=5840 Len=1448 TSV=4294956257
  772 5.454841   192.168.12.2   192.168.10.2   TCP      [TCP Dup ACK 765#3] commplex-link > 43545 [ACK] Seq=1 Ack=577633 Win=172800 Le
  773 5.454921   192.168.10.2   192.168.12.2   TCP      [TCP Fast Retransmission] 43545 > commplex-link [ACK] Seq=577633 Ack=1 Win=584
  774 5.466979   192.168.12.2   192.168.10.2   TCP      [TCP Dup ACK 765#4] commplex-link > 43545 [ACK] Seq=1 Ack=577633 Win=172800 Le

Transmission Control Protocol, Src Port: commplex-link (5001), Dst Port: 43545 (43545), Seq: 1, Ack: 285657, Len: 0
    Source port: commplex-link (5001)
    Destination port: 43545 (43545)
    [Stream index: 0]
    Sequence number: 1    (relative sequence number)
    Acknowledgement number: 285657    (relative ack number)
    Header length: 44 bytes
  ▷ Flags: 0x10 (ACK)
    Window size: 170864 (scaled)
  ▷ Checksum: 0x5e6d [validation disabled]
  ▷ Options: (24 bytes)
  ▽ [SEQ/ACK analysis]
    ▷ [TCP Analysis Flags]
      [Duplicate ACK #: 6]
    ▷ [Duplicate to the ACK in frame: 435]

0000  fe fd 00 00 00 00 72 eb  15 bc 33 e6 08 00 45 00   ......r. ..3...E.
0010  00 40 3c e4 40 00 3f 06  67 7f c0 a8 0c 02 c0 a8   .@<.@.?. g.......
0020  0a 02 13 89 aa 19 70 51  ed 3e 6f f8 21 ea b0 10   ......pQ .>o.!...
0030  a6 dc 5e 6d 00 00 01 01  08 0a ff ff d2 67 ff ff   ..^m.... .....g..

Frame (frame), 78 bytes    Packets: 1298 Displayed: 1298 Marked: 0    Profile: Default
```

Fig. 8: Wireshark : default window



```
Time   Comment
0.000  TCP: 46576 > commplex-link [SYN] Seq=0 Win=5840 Len=0 TSV=4294950736 TSER=0 MSS=1460 WS=2
0.211  TCP: commplex-link > 46576 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 TSV=4294950509 TSER=4294950736 MSS
0.211  TCP: 46576 > commplex-link [ACK] Seq=1 Ack=1 Win=5840 Len=0 TSV=4294950763 TSER=4294950509
0.211  TCP: 46576 > commplex-link [PSH, ACK] Seq=1 Ack=1 Win=5840 Len=24 TSV=4294950763 TSER=4294950509
0.211  TCP: 46576 > commplex-link [ACK] Seq=25 Ack=1 Win=5840 Len=1448 TSV=4294950763 TSER=4294950509
0.311  TCP: commplex-link > 46576 [ACK] Seq=1 Ack=25 Win=5792 Len=0 TSV=4294950530 TSER=4294950763
0.311  TCP: 46576 > commplex-link [ACK] Seq=1473 Ack=1 Win=5840 Len=1448 TSV=4294950773 TSER=4294950530[Pa
0.311  TCP: 46576 > commplex-link [PSH, ACK] Seq=2921 Ack=1 Win=5840 Len=1448 TSV=4294950773 TSER=42949505
0.321  TCP: commplex-link > 46576 [ACK] Seq=1 Ack=1473 Win=8688 Len=0 TSV=4294950531 TSER=4294950763
0.321  TCP: 46576 > commplex-link [ACK] Seq=4369 Ack=1 Win=5840 Len=1448 TSV=4294950774 TSER=4294950531[Pa
0.321  TCP: 46576 > commplex-link [ACK] Seq=5817 Ack=1 Win=5840 Len=1448 TSV=4294950774 TSER=4294950531[Pa
0.412  TCP: commplex-link > 46576 [ACK] Seq=1 Ack=2921 Win=11584 Len=0 TSV=4294950540 TSER=4294950773
0.412  TCP: 46576 > commplex-link [ACK] Seq=7265 Ack=1 Win=5840 Len=1448 TSV=4294950783 TSER=4294950540[Pa
0.412  TCP: 46576 > commplex-link [ACK] Seq=8713 Ack=1 Win=5840 Len=1448 TSV=4294950783 TSER=4294950540[Pa
0.419  TCP: commplex-link > 46576 [ACK] Seq=1 Ack=4369 Win=14480 Len=0 TSV=4294950541 TSER=4294950773
0.420  TCP: 46576 > commplex-link [PSH, ACK] Seq=10161 Ack=1 Win=5840 Len=1448 TSV=4294950784 TSER=4294950
0.420  TCP: 46576 > commplex-link [ACK] Seq=11609 Ack=1 Win=5840 Len=1448 TSV=4294950784 TSER=4294950541[P
```

Fig. 9: Wireshark : flow graph

Use wireshark to analyze the packet traces described earlier `traces/simple-trace.pcap`.

When analyzing packet traces with wireshark, it is often very useful to use *Display filters* that only show the packets that match some specific criteria. There filters are described in several online documents:

- the *wireshark wiki <https://wiki.wireshark.org/>* page on Display filters
- a nice list of Wireshark Display Filters by Robert Allen

You can now use your understanding of wireshark and tcpdump to analyze a 2-minutes long packet trace.

4.12 TCP basics

TCP is one of the key protocols in today's Internet. A TCP connection always starts with a three-way handshake. The exercises below should help you to improve your understandings of this first phase of a TCP connection.

Once the connection is established, the client and the server can exchange data and acknowledgments.

A connection ends with the transmission of segments that include the *FIN* flag that marks the end of the data transfer.

TCP can be extended by using options that are negotiated during the three-way handshake.

With your knowledge of TCP, you should now be able to reorder all the segments exchanged over a TCP connection.

4.13 A closer look at TCP

In this series of exercises, you will explore in more details the operation of TCP and its congestion control scheme. TCP is a very important protocol in today's Internet since most applications use it to exchange data. We first look at TCP in more details by injecting segments in the Linux TCP stack and analyze how the stack reacts. Then we study the TCP congestion control scheme.

4.13.1 Injecting segments in the Linux TCP stack

Packet capture tools like tcpdump and Wireshark are very useful to observe the segments that transport protocols exchange. TCP is a complex protocol that has evolved a lot since its first specification **RFC 793**. TCP includes a large number of heuristics that influence the reaction of a TCP implementation to various types of events. A TCP implementation interacts with the application through the `socket` API.

packetdrill is a TCP test suite that was designed to develop unit tests to verify the correct operation of a TCP implementation. A more detailed description of packetdrill may be found in [CCB+2013]. packetdrill uses a syntax which is a mix between the C language and the tcpdump syntax. To understand the operation of packetdrill, we first discuss several examples. The TCP implementation in the Linux kernel supports all the recent TCP extensions to improve its performance. For pedagogical reasons, we disable[1] most of these extensions to use a simple TCP stack. packetdrill can be easily installed on recent Linux kernels[2].

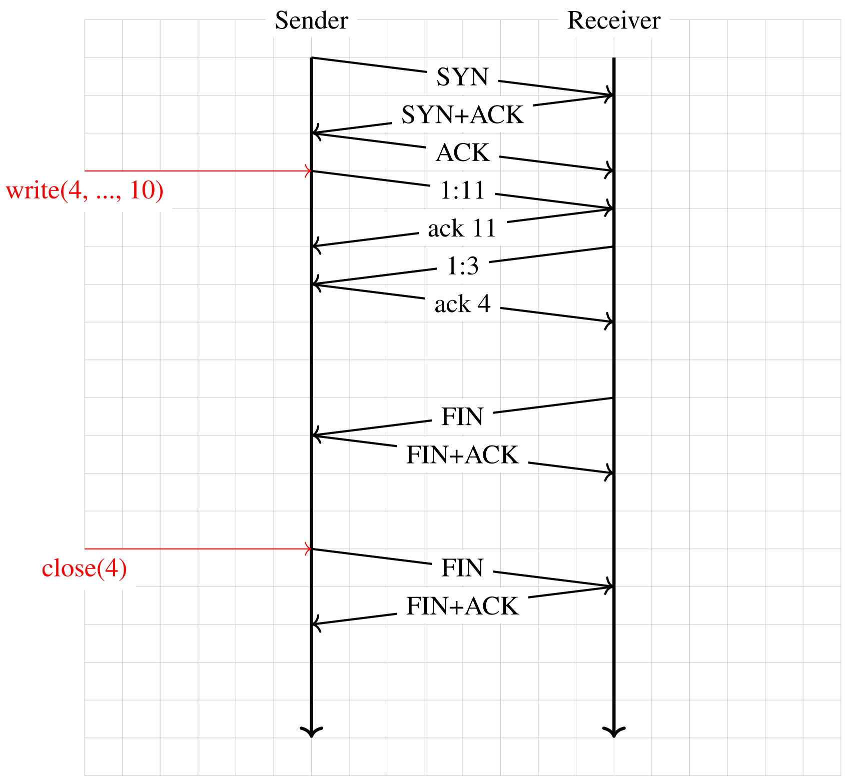
Let us start with a very simple example that uses packetdrill to open a TCP connection on a server running on the Linux kernel. A packetdrill script is a sequence of lines that are executed one after the other. There are three main types of lines in a packetdrill script.

[1] On Linux, most of the parameters to tune the TCP stack are accessible via *sysctl*. These parameters are briefly described in https://github.com/torvalds/linux/blob/master/Documentation/networking/ip-sysctl.txt and in the *tcp* manpage. Each script sets some of these configuration variables.

[2] packetdrill requires root privileges since it inject raw IP packets. The easiest way to install it is to use a virtualbox image with a Linux kernel 4.x or 5.x. You can clone its git repository from https://github.com/google/packetdrill and follow the instructions in https://github.com/google/packetdrill/tree/master/gtests/net/packetdrill. The packetdrill scripts used in this section are available from https://github.com/cnp3/ebook/tree/master/exercises/packetdrill_scripts

- packetdrill executes a system call and verifies its return value
- packetdrill injects[3] a packet in the instrumented Linux kernel as if it were received from the network
- packetdrill compares a packet transmitted by the instrumented Linux kernel with the packet that the script expects

For our first packetdrill script, we aim at reproducing the simple connection shown in the figure below.



Let us start with the execution of a system call. A simple example is shown below.

```
0   socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
```

The 0 indicates that the system call must be issued immediately. packetdrill then executes the system call and verifies that it returns 3`. If yes, the processing continues. Otherwise the script stops and indicates an error.

For this first example, we program packetdrill to inject the segments that a client would send. The first step is thus to prepare a *socket* that can be used to accept this connection. This socket can be created by using the four system calls below.

```
// create a TCP socket. Since stdin, stdout and stderr are already defined,
// the kernel will assign file descriptor 3 to this socket
// 0 is the absolute time at which the socket is created
0   socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
// Enable reuse of addresses
+0  setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
// binds the created socket to the available addresses
+0  bind(3, ..., ...) = 0
// configure the socket to accept incoming connections
+0  listen(3, 1) = 0
```

[3] By default, packetdrill uses port 8080 when creating TCP segments. You can thus capture the packets injected by packetdrill and the responses from the stack by using `tcpdump -i any -n port 8080`

At this point, the socket is ready to accept incoming TCP connections. packetdrill needs to inject a TCP segment in the instrumented Linux stack. This can be done with the line below.

```
+0  < S 0:0(0) win 1000 <mss 1000>
```

Each line of a packetdrill script starts with a *timing* parameter that indicates at what time the event specified on this line should happen. packetdrill supports absolute and relative timings. An absolute timing is simply a number that indicates the delay in seconds between the start of the script and the event. A relative timing is indicated by using + followed by a number. This number is then the delay in seconds between the previous event and the current line. Additional information may be found in [CCB+2013].

The description of TCP packets in packetdrill uses a syntax that is very close to the tcpdump one. The +0 timing indicates that the line is executed immediately after the previous event. The < sign indicates that packetdrill injects a TCP segment and the S character indicates that the SYN flag must be set. Like tcpdump, packetdrill uses sequence numbers that are relative to initial sequence number. The three numbers that follow are the sequence number of the first byte of the payload of the segment (0), the sequence number of the last byte of the payload of the segment (0 after the semi-column) and the length of the payload (0 between brackets) of the SYN segment. This segment does not contain a valid acknowledgment but advertises a window of 1000 bytes. All SYN segments must also include the MSS option. In this case, we set the MSS to 1000 bytes. The next line of the packetdrill script verifies the reply sent by the instrumented Linux kernel.

```
+0  > S. 0:0(0) ack 1 <...>
```

This TCP segment is sent immediately by the stack. The SYN flag is set and the dot next to the S character indicates that the ACK flag is also set. The SYN+ACK segment does not contain any data but its acknowledgment number is set to 1 (relative to the initial sequence number). For outgoing packets, packetdrill does not verify the value of the advertised window. In this line, it also accepts any TCP options (<...>).

The third segment of the three-way handshake is sent by packetdrill after a delay of 0.1 seconds. The connection is now established and the accept system call will succeed.

```
+.1 < . 1:1(0) ack 1 win 1000
+0  accept(3, ..., ...) = 4
```

The *accept* system call returns a new file descriptor, in this case value 4. At this point, packetdrill can write data on the socket or inject packets.

```
+0 write(4, ..., 10)=10
+0 > P. 1:11(10) ack 1
+.1 < . 1:1(0) ack 11 win 1000
```

packetdrill writes 10 bytes of data through the *write* system call. The stack immediately sends these 10 bytes inside a segment whose Push flag is set[4]. The payload starts at sequence number 1 and ends at sequence number 10. packetdrill replies by injecting an acknowledgment for the entire data after 100 milliseconds.

packetdrill can also inject data that will be read by the stack as shown by the lines below.

```
+.1 < P. 1:3(2) ack 11 win 4000
+0 > . 11:11(0) ack 3
+.2 read(4,...,1000)=2
```

In the example above, packetdrill injects a segment containing two bytes. This segment is acknowledged and after that the *read* system call succeeds and reads the available data with a buffer of 1000 bytes. It returns the amount of read bytes, i.e. 2.

We can now close the connection gracefully. Let us first issue inject a segment with the FIN flag set.

[4] The *Push* flag is one of the TCP flags defined in **RFC 793**. TCP stacks usually set this flag when transmitting a segment that empties the send buffer. This is the reason why we observe this push flag in our example.

```
//Packetdrill closes connection gracefully
+0 < F. 3:3(0) ack 11 win 4000
+0 > . 11:11(0) ack 4
```

packetdrill injects the `FIN` segment and the instrumented kernel returns an acknowledgment. If packetdrill issues the *close* system call, the kernel will send a `FIN` segment to terminate the connection. packetdrill injects an acknowledgment to confirm the end of the connection.

```
+0 close(4) = 0
+0 > F. 11:11(0) ack 4
+0 < . 4:4(0) ack 12 win 4000
```

The complete packetdrill script is available from `/exercises/packetdrill_scripts/connect.pkt`.

Another interesting features of packetdrill is that it is possible to inspect the state maintained by the Linux kernel for the underlying connection using the `TCP_INFO` socket option. This makes it possible to retrieve the value of some variables of the TCP control block.

Let us first explore how a TCP connection can be established. In the previous script, we have injected the segments that a client would send to a server. We can also use the Linux stack as a client and inject the segments that a server would return. Such a client process would first create its *socket*` and then issue the *connect* system call. At this point, the stack sends a `SYN` segment. To simplify the scripts, we have configured the stack to use a `MSS` of 1000 bytes and disabled the TCP extensions (the details of this configuration may be found at the beginning of the script). The server replies with a `SYN+ACK` and the stack sends acknowledges it to finish the three-way-handshake.

```
0 socket(..., SOCK_STREAM, IPPROTO_TCP) = 4
+0 fcntl(4, F_SETFL, O_RDWR|O_NONBLOCK) = 0
+0 setsockopt(4, SOL_TCP, TCP_NODELAY, [1], 4) = 0

+0 connect(4, ..., ...) = -1 EINPROGRESS (Operation now in progress)
+0 > S 0:0(0) <mss 1000>
// TCP State is now SYN_SENT
+0 %{ print "State@1", tcpi_state  }%     // prints 2, i.e. TCP_SYN_SENT
+.1 < S. 0:0(0) ack 1 win 10000 <mss 1000>
+0 > . 1:1(0) ack 1
+0 %{ print "State@2", tcpi_state  }%   // prints 1, i.e. TCP_ESTABLISHED
// TCP State is now ESTABLISHED
```

The `tcpi_state` variable used in this script is returned by `TCP_INFO`[5]. It tracks the state of the TCP connection according to TCP's finite state machine[6]. This script is available from `/exercises/packetdrill_scripts/client.pkt`.

Another example is the simultaneous establishment of a TCP connection. The TCP stack sends a `SYN` and receives a `SYN` in response instead of a `SYN+ACK`. It then acknowledges the received `SYN` and retransmits its own `SYN`. The connection becomes established upon reception of the fourth segment. This script is available from `/exercises/packetdrill_scripts/dual.pkt`.

```
 0 socket(..., SOCK_STREAM, IPPROTO_TCP) = 4
+0 fcntl(4, F_SETFL, O_RDWR|O_NONBLOCK) = 0
+0 setsockopt(4, SOL_TCP, TCP_NODELAY, [1], 4) = 0

+0 connect(4, ..., ...) = -1 EINPROGRESS (Operation now in progress)
+0 > S 0:0(0) <mss 1000>
+0 %{ print "State@1", tcpi_state  }%  // prints 2, i.e. TCP_SYN_SENT
```

(continues on next page)

[5] The variables that are included in TCP_INFO are defined in https://github.com/torvalds/linux/blob/master/include/uapi/linux/tcp.h

[6] These states are defined in https://github.com/torvalds/linux/blob/master/include/net/tcp_states.h

(continued from previous page)

```
+.1 < S 0:0(0) win 5792 <mss 1000>
+0 %{ print "State@2", tcpi_state  }%  // prints 3, i.e. TCP_SYN_RECV
+0 > S. 0:0(0) ack 1 <mss 1000>
+0 %{ print "State@3", tcpi_state  }%  // prints 3, i.e. TCP_SYN_RECV
+.1 < . 1:1(0) ack 1 win 5792
+0 %{ print "State@4", tcpi_state  }%  // prints 1, i.e. TCP_ESTABLISHED
```

Another usage of packetdrill is to explore how a TCP connection ends. The scripts below show how a TCP stack closes a TCP connection. The first example shows a local host that connects to a remote host and then closes the connection. The remote host acknowledges the `FIN` and later closes its direction of data transfer. This script is available from `/exercises/packetdrill_scripts/local-close.pkt`.

```
0 socket(..., SOCK_STREAM, IPPROTO_TCP) = 4
+0 fcntl(4, F_SETFL, O_RDWR|O_NONBLOCK) = 0
+0 setsockopt(4, SOL_TCP, TCP_NODELAY, [1], 4) = 0

+0 connect(4, ..., ...) = -1 EINPROGRESS (Operation now in progress)
+0 > S 0:0(0) <mss 1000>
+.1 < S. 0:0(0) ack 1 win 10000 <mss 1000>
+0 > . 1:1(0) ack 1

+.1 close(4)=0
+0 > F. 1:1(0) ack 1
+0 < . 1:1(0) ack 2 win 10000
+.1 < F. 1:1(0) ack 2 win 10000
+0 > . 2:2(0) ack 2
```

As for the establishment of a connection, it is also possible for the two communicating hosts to close the connection at the same time. This is shown in the example below where the remote host sends its own `FIN` when acknowledging the first one. This script is available from `/exercises/packetdrill_scripts/local-close2.pkt`.

```
 0 socket(..., SOCK_STREAM, IPPROTO_TCP) = 4
 +0 fcntl(4, F_SETFL, O_RDWR|O_NONBLOCK) = 0
 +0 setsockopt(4, SOL_TCP, TCP_NODELAY, [1], 4) = 0

+0 connect(4, ..., ...) = -1 EINPROGRESS (Operation now in progress)
+0 > S 0:0(0) <mss 1000>
+.1 < S. 0:0(0) ack 1 win 10000 <mss 1000>
+0 > . 1:1(0) ack 1

+.1 close(4)=0
+0 > F. 1:1(0) ack 1
+.1 < F. 1:1(0) ack 2 win 10000
+0 > . 2:2(0) ack 2
```

A third scenario for the termination of a TCP connection is that the remote hosts sends its `FIN` first. This script is available from `/exercises/packetdrill_scripts/remote-close.pkt`.

```
0 socket(..., SOCK_STREAM, IPPROTO_TCP) = 4
+0 fcntl(4, F_SETFL, O_RDWR|O_NONBLOCK) = 0
+0 setsockopt(4, SOL_TCP, TCP_NODELAY, [1], 4) = 0

+0 connect(4, ..., ...) = -1 EINPROGRESS (Operation now in progress)
+0 > S 0:0(0) <mss 1000>
+.1 < S. 0:0(0) ack 1 win 10000 <mss 1000>
+0 > . 1:1(0) ack 1
```

(continues on next page)

(continued from previous page)

```
// remote closes
+.1 < F. 1:1(0) ack 1 win 10000
+0 > . 1:1(0) ack 2
// local host terminates the connection
+.1 close(4)=0
+0 > F. 1:1(0) ack 2
+0 < . 2:2(0) ack 2 win 10000
```

Another very interesting utilization of packetdrill is to explore how a TCP stack reacts to acknowledgments that would correspond to lost or reordered segments. For this analysis, we configure a very large initial congestion window to ensure that the connection does not start with a slow-start.

Let us first use packetdrill to explore the evolution of the TCP retransmission timeout. The value of this timeout is set based on the measured round-trip-time and its variance. When the retransmission timer expires, TCP doubles the retransmission timer. This exponential backoff mechanism is important to ensure that TCP slowdowns during very severe congestion periods. We use the `tcpi_rto` variable from `TCP_INFO` to track the evolution of the retransmission timer. This script is available from `/exercises/packetdrill_scripts/rto.pkt`.

```
+.1 accept(3, ..., ...) = 4
// initial congestion window is 16KBytes
// server sends 8000 bytes
+0 %{ print "RTO @1: ",tcpi_rto }% // prints 204000 (microseconds)
+.1 write(4, ..., 8000) = 8000
+0 > .  1:1001(1000) ack 1
+0 > .  1001:2001(1000) ack 1
+0 > .  2001:3001(1000) ack 1
+0 > .  3001:4001(1000) ack 1
+0 > .  4001:5001(1000) ack 1
+0 > .  5001:6001(1000) ack 1
+0 > .  6001:7001(1000) ack 1
+0 > P.  7001:8001(1000) ack 1

// client acks one segment
+.1 < .  1:1(0) ack 1001 win 50000
+0 %{ print "RTO @2: ",tcpi_rto }% // prints 216000 (microseconds)
// client did not receive any other segment
// first server retransmissions
+.216 > . 1001:2001(1000) ack 1
+0 %{ print "RTO @3: ",tcpi_rto }%  // prints 432000 (microseconds)
// second after doubling rto
+.432 > . 1001:2001(1000) ack 1
+0 %{ print "RTO @4: ",tcpi_rto }% // prints 864000 (microseconds)
+.864 > . 1001:2001(1000) ack 1
+0 %{ print "RTO @5: ",tcpi_rto }% // prints 1728000 (microseconds)
```

We can use a similar code to demonstrate that the TCP stack performs a fast retransmit after having received three duplicate acknowledgments. This script is available from `/exercises/packetdrill_scripts/frr.pkt`.

```
+.1 accept(3, ..., ...) = 4
// initial congestion window is 16KBytes
// server sends 8000 bytes
+0 %{ print "retransmissions @1: ",tcpi_bytes_retrans }% // prints 0
+.1 write(4, ..., 8000) = 8000
+0 > .  1:1001(1000) ack 1
+0 > .  1001:2001(1000) ack 1
+0 > .  2001:3001(1000) ack 1
```

(continues on next page)

(continued from previous page)

```
+0 > .  3001:4001(1000) ack 1
+0 > .  4001:5001(1000) ack 1
+0 > .  5001:6001(1000) ack 1
+0 > .  6001:7001(1000) ack 1
+0 > P.  7001:8001(1000) ack 1

// client acks two segments
+.1 < .  1:1(0) ack 1001 win 50000
+0 < .  1:1(0) ack 2001 win 50000
+0 < .  1:1(0) ack 2001 win 50000
+0 < .  1:1(0) ack 2001 win 50000
+0 < .  1:1(0) ack 2001 win 50000
// server retransmits after three duplicate acks
+0 > .  2001:3001(1000) ack 1
+0 < .  1:1(0) ack 2001 win 50000
+0 < .  1:1(0) ack 2001 win 50000
// client acks everything
+0 %{ print "retransmissions @2: ",tcpi_bytes_retrans }% // prints 1000
+.1 < .  1:1(0) ack 8001 win 50000
```

A TCP stack uses both the fast retransmit technique and retransmission timers. A retransmission timer can fire after a fast retransmit when several segments are lost. The example below shows a loss of two consecutive segments. This script is available from `/exercises/packetdrill_scripts/frr-rto.pkt`.

```
+.1 accept(3, ..., ...) = 4
// initial congestion window is 16KBytes
// server sends 8000 bytes
+0 %{ print "retransmissions @1: ",tcpi_bytes_retrans }%
+.1 write(4, ..., 8000) = 8000
 +0 > .  1:1001(1000) ack 1
 +0 > .  1001:2001(1000) ack 1
 +0 > .  2001:3001(1000) ack 1 // lost
 +0 > .  3001:4001(1000) ack 1 // lost
 +0 > .  4001:5001(1000) ack 1
 +0 > .  5001:6001(1000) ack 1
 +0 > .  6001:7001(1000) ack 1
 +0 > P.  7001:8001(1000) ack 1

 // client acks one segments
 +.1 < .  1:1(0) ack 1001 win 50000
 +0 < .  1:1(0) ack 2001 win 50000
 +0 < .  1:1(0) ack 2001 win 50000
 +0 < .  1:1(0) ack 2001 win 50000
 +0 < .  1:1(0) ack 2001 win 50000
 // server retransmits after three duplicate acks
 +0 > .  2001:3001(1000) ack 1
 +0 < .  1:1(0) ack 2001 win 50000
 +0 < .  1:1(0) ack 2001 win 50000
 // client acks retransmission
 +0 %{ print "retransmissions @2: ",tcpi_bytes_retrans }% // prints 1000
 +0 %{ print "RTO @2: ",tcpi_rto }%   // prints 224000
 +.1 < .  1:1(0) ack 3001 win 50000

 +.024 > .  3001:4001(1000) ack 1
 // client acks everything
 +.1 < .  1:1(0) ack 8001 win 50000
```

More complex scenarios can be written. The script below demonstrates how the TCP stack behaves when three segments are lost. This script is available from /exercises/packetdrill_scripts/frr-rto2.pkt.

```
+.1 accept(3, ..., ...) = 4
// initial congestion window is 16KBytes
// server sends 8000 bytes
+0 %{ print "retransmissions @1: ",tcpi_bytes_retrans }%
+.1 write(4, ..., 8000) = 8000
+0 > .  1:1001(1000) ack 1
+0 > .  1001:2001(1000) ack 1
+0 > .  2001:3001(1000) ack 1 // lost
+0 > .  3001:4001(1000) ack 1
+0 > .  4001:5001(1000) ack 1 // lost
+0 > .  5001:6001(1000) ack 1
+0 > .  6001:7001(1000) ack 1 // lost
+0 > P.  7001:8001(1000) ack 1

// client acks first two segments
+.1 < .  1:1(0) ack 1001 win 50000
+0 < .  1:1(0) ack 2001 win 50000
+0 < .  1:1(0) ack 2001 win 50000
+0 < .  1:1(0) ack 2001 win 50000
+0 < .  1:1(0) ack 2001 win 50000
// server retransmits after three duplicate acks
+0 > .  2001:3001(1000) ack 1
+0 < .  1:1(0) ack 2001 win 50000
+0 < .  1:1(0) ack 2001 win 50000
// client acks retransmission
+0 %{ print "retransmissions @2: ",tcpi_bytes_retrans }% // prints 1000
+0 %{ print "RTO @2: ",tcpi_rto }%   // prints 224000
+.1 < .  1:1(0) ack 4001 win 50000

+.024 > .  4001:5001(1000) ack 1
// client acks block
+0 %{ print "retransmissions @3: ",tcpi_bytes_retrans }% // prints 2000
+0 %{ print "RTO @3: ",tcpi_rto }%   // prints 224000
+.1 < .  1:1(0) ack 6001 win 50000
+.024 > .  6001:7001(1000) ack 1
// client acks block
+0 %{ print "retransmissions @3: ",tcpi_bytes_retrans }% // prints 3000
+0 %{ print "RTO @3: ",tcpi_rto }%   // prints 224000
+.1 < .  1:1(0) ack 8001 win 50000
```

The examples above have demonstrated how TCP retransmits lost segments. However, they did not consider the interactions with the congestion control scheme since the use a large initial congestion window. We now set the initial congestion window to two MSS-sized segments and use the tcpi_snd_cwnd and tcpi_snd_ssthresh variables from TCP_INFO to explore the evolution of the TCP congestion control scheme. Our first script looks at the evolution of the congestion window during a slow-start when there are no losses. This script is available from /exercises/packetdrill_scripts/slow-start.pkt.

```
+.1 accept(3, ..., ...) = 4
// initial congestion window is 2 KBytes
+0 %{ print "cwnd @1: ",tcpi_snd_cwnd }% // prints 2
+0 %{ print "ssthresh @1: ",tcpi_snd_ssthresh }% // prints 2147483647
// server sends 16000 bytes
+.1 write(4, ..., 16000) = 16000
+0 > .  1:1001(1000) ack 1
```

(continues on next page)

(continued from previous page)

```
+0 > .  1001:2001(1000) ack 1

+.1 < .  1:1(0) ack 1001 win 50000
+0 %{ print "cwnd @2: ",tcpi_snd_cwnd }% // prints 3
+0 > .  2001:3001(1000) ack 1
+0 > .  3001:4001(1000) ack 1
+0.01  < .  1:1(0) ack 2001 win 50000
+0 %{ print "cwnd @3: ",tcpi_snd_cwnd }%  // prints 4
+0 > .  4001:5001(1000) ack 1
+0 > .  5001:6001(1000) ack 1

+.1 < .  1:1(0) ack 3001 win 50000
+0 %{ print "cwnd @4: ",tcpi_snd_cwnd }%  // prints 5
+0 > .  6001:7001(1000) ack 1
+0 > .  7001:8001(1000) ack 1
+0.01  < .  1:1(0) ack 4001 win 50000
+0 %{ print "cwnd @5: ",tcpi_snd_cwnd }% // prints 6
+0 > .  8001:9001(1000) ack 1
+0 > .  9001:10001(1000) ack 1
+0.01 < .  1:1(0) ack 5001 win 50000
+0 %{ print "cwnd @6: ",tcpi_snd_cwnd }% // prints 7
+0 > .  10001:11001(1000) ack 1
+0 > .  11001:12001(1000) ack 1
+0.01  < .  1:1(0) ack 6001 win 50000
+0 %{ print "cwnd @7: ",tcpi_snd_cwnd }% // prints 8
+0 > .  12001:13001(1000) ack 1
+0 > .  13001:14001(1000) ack 1

+.1 < .  1:1(0) ack 7001 win 50000
+0 %{ print "cwnd @8: ",tcpi_snd_cwnd }%     // prints 9
+0 > .  14001:15001(1000) ack 1
+0 > P.  15001:16001(1000) ack 1

// client acks everything
+.1 < .  1:1(0) ack 16001 win 50000
+0 %{ print "cwnd @9: ",tcpi_snd_cwnd }% // prints 18
```

Some TCP clients use delayed acknowledgments and send a TCP acknowledgment after after second in-sequence segment. This behavior is illustrated in the script below. This script is available from `/exercises/packetdrill_scripts/slow-start-delayed.pkt`.

```
+.1 accept(3, ..., ...) = 4
// initial congestion window is 2 KBytes
+0 %{ print "cwnd @1: ",tcpi_snd_cwnd }% // prints 2
+0 %{ print "ssthresh @1: ",tcpi_snd_ssthresh }% // prints 2147483647
// server sends 16000 bytes
+.1 write(4, ..., 16000) = 16000
+0 > .  1:1001(1000) ack 1
+0 > .  1001:2001(1000) ack 1

+.1 < .  1:1(0) ack 2001 win 50000
+0 %{ print "cwnd @2: ",tcpi_snd_cwnd }% // prints 4
+0 > .  2001:3001(1000) ack 1
+0 > .  3001:4001(1000) ack 1
+0 > .  4001:5001(1000) ack 1
+0 > .  5001:6001(1000) ack 1
```

(continues on next page)

(continued from previous page)

```
+.1 < .  1:1(0) ack 4001 win 50000
+0 %{ print "cwnd @4: ",tcpi_snd_cwnd }%  // prints 6
+0 > .  6001:7001(1000) ack 1
+0 > .  7001:8001(1000) ack 1
+0 > .  8001:9001(1000) ack 1
+0 > .  9001:10001(1000) ack 1
+0.01 < .  1:1(0) ack 6001 win 50000
+0 %{ print "cwnd @6: ",tcpi_snd_cwnd }% // prints 8
+0 > .  10001:11001(1000) ack 1
+0 > .  11001:12001(1000) ack 1
+0 > .  12001:13001(1000) ack 1
+0 > .  13001:14001(1000) ack 1

+.1 < .  1:1(0) ack 8001 win 50000
+0 %{ print "cwnd @8: ",tcpi_snd_cwnd }% // prints 10
+0 > .  14001:15001(1000) ack 1
+0 > P.  15001:16001(1000) ack 1

// client acks everything
+.1 < .  1:1(0) ack 16001 win 50000
+0 %{ print "cwnd @9: ",tcpi_snd_cwnd }% // prints 18
```

We can now explore how TCP's retransmission techniques interact with the congestion control scheme. The Linux TCP code that combines these two techniques contains several heuristics to improve their performance. We start with a transfer of 8KBytes where the penultimate segment is not received by the remote host. In this case, TCP does not receive enough acknowledgments to trigger the fast retransmit and it must wait for the expiration of the retransmission timer. This script is available from `/exercises/packetdrill_scripts/slow-start-rto2.pkt`.

```
+.1 accept(3, ..., ...) = 4
// initial congestion window is 2 KBytes
+0 %{ print "cwnd @1: ",tcpi_snd_cwnd }% // prints 2
+0 %{ print "ssthresh @1: ",tcpi_snd_ssthresh }% // prints 2147483647
// server sends 8000 bytes
+.1 write(4, ..., 8000) = 8000
+0 > .  1:1001(1000) ack 1
+0 > .  1001:2001(1000) ack 1

+.1 < .  1:1(0) ack 1001 win 50000
+0 %{ print "cwnd @2: ",tcpi_snd_cwnd }% // prints 3
+0 > .  2001:3001(1000) ack 1
+0 > .  3001:4001(1000) ack 1
+0.01  < .  1:1(0) ack 2001 win 50000
+0 %{ print "cwnd @3: ",tcpi_snd_cwnd }%  // prints 4
+0 > .  4001:5001(1000) ack 1
+0 > .  5001:6001(1000) ack 1

+.1 < .  1:1(0) ack 3001 win 50000
+0 %{ print "cwnd @4: ",tcpi_snd_cwnd }%  // prints 5
+0 %{ print "RTO @4: ",tcpi_rto }%  // prints 252000
+0 > .  6001:7001(1000) ack 1 // lost
+0 > P.  7001:8001(1000) ack 1
+0.01  < .  1:1(0) ack 4001 win 50000
+0 %{ print "cwnd @5: ",tcpi_snd_cwnd }% // prints 6
+0.01 < .  1:1(0) ack 5001 win 50000
+0 %{ print "cwnd @6: ",tcpi_snd_cwnd }% // prints 7
```

(continues on next page)

(continued from previous page)

```
+0.01  < .  1:1(0) ack 6001 win 50000
+0 %{ print "cwnd @7: ",tcpi_snd_cwnd }% // prints 8

+.1 < .  1:1(0) ack 6001 win 50000
+0 %{ print "cwnd @8: ",tcpi_snd_cwnd }% // prints 8

+.25 > .  6001:7001(1000) ack 1 // retransmission
+0 %{ print "cwnd @9: ",tcpi_snd_cwnd }% // prints 1

+.1 < .  1:1(0) ack 8001 win 50000
+0 %{ print "cwnd @10: ",tcpi_snd_cwnd }% // prints 3
```

Another interesting scenario is when the loss happens early in the data transfer. This is shown in the script below where the second segment is lost. We observe that by triggering transmissions of unacknowledged data, the RFC 3042 rule speeds up the recovery since a fast retransmit happens. This script is available from `/exercises/packetdrill_scripts/slow-start-frr2.pkt`.

```
   +.1 accept(3, ..., ...) = 4
// initial congestion window is 2 KBytes
+0 %{ print "cwnd @1: ",tcpi_snd_cwnd }% // prints 2
+0 %{ print "ssthresh @1: ",tcpi_snd_ssthresh }% // prints 2147483647
// server sends 8000 bytes
+.1 write(4, ..., 8000) = 8000
+0 > .  1:1001(1000) ack 1
+0 > .  1001:2001(1000) ack 1 // lost

+.01 < .  1:1(0) ack 1001 win 50000
+0 %{ print "cwnd @2: ",tcpi_snd_cwnd }% // prints 3
+0 %{ print "RTO @2: ",tcpi_rto }% // prints 204000
+0 > .  2001:3001(1000) ack 1
+0 > .  3001:4001(1000) ack 1

+0.01  < .  1:1(0) ack 1001 win 50000 // reception of 2001:3001
+0 %{ print "cwnd @3: ",tcpi_snd_cwnd }%  // prints 3
+0 > .  4001:5001(1000) ack 1  // rfc 3042
+0.001  < .  1:1(0) ack 1001 win 50000 // reception of 3001:4001
+0 %{ print "cwnd @4: ",tcpi_snd_cwnd }%  // prints 3
+0 > .  5001:6001(1000) ack 1 // rfc 3042

+0.01  < .  1:1(0) ack 1001 win 50000 // reception of 4001:5001
+0 %{ print "cwnd @5: ",tcpi_snd_cwnd }%  // prints 2
+0 %{ print "ssthresh @5: ",tcpi_snd_ssthresh }% // prints 2
// fast retransmit
+0 > .  1001:2001(1000) ack 1

+0.001  < .  1:1(0) ack 1001 win 50000 // reception of 5001:6001
+0 %{ print "cwnd @6: ",tcpi_snd_cwnd }%  // prints 2
+0 > .  6001:7001(1000) ack 1 // rfc 3042
+0.01  < .  1:1(0) ack 6001 win 50000 // reception of 1001:2001
+0 %{ print "cwnd @7: ",tcpi_snd_cwnd }%  // prints 2
+0 %{ print "ssthresh @7: ",tcpi_snd_ssthresh }% // prints 2

+0 > P.  7001:8001(1000) ack 1

+0.01  < .  1:1(0) ack 7001 win 50000
```

(continues on next page)

(continued from previous page)

```
+0 %{ print "cwnd @7: ",tcpi_snd_cwnd }%  // prints 4
+0.001  < .  1:1(0) ack 8001 win 50000
+0 %{ print "cwnd @8: ",tcpi_snd_cwnd }%  // prints 5
```

Our last scenario is when the first segment sent is lost. In this case, two round-trip-times are required to retransmit the missing segment and recover from the loss. This script is available from `/exercises/packetdrill_scripts/slow-start-frr3.pkt`.

```
+.1 accept(3, ..., ...) = 4
// initial congestion window is 2 KBytes
+0 %{ print "cwnd @1: ",tcpi_snd_cwnd }% // prints 2
+0 %{ print "ssthresh @1: ",tcpi_snd_ssthresh }% // prints 2147483647
+0 %{ print "RTO @1: ",tcpi_rto }% // prints 204000
// server sends 8000 bytes
+.1 write(4, ..., 8000) = 8000
+0 > .  1:1001(1000) ack 1 // lost
+0 > .  1001:2001(1000) ack 1

+.01 < .  1:1(0) ack 1 win 50000  // duplicate ack
+0 %{ print "cwnd @2: ",tcpi_snd_cwnd }% // prints 2
+0 > .  2001:3001(1000) ack 1 // rfc3042

+0.01  < .  1:1(0) ack 1 win 50000 // reception of 2001:3001
+0 %{ print "cwnd @3: ",tcpi_snd_cwnd }%  // prints 2
+0 > .  3001:4001(1000) ack 1 // rfc3042

+0.01  < .  1:1(0) ack 1 win 50000 // reception of 3001:4001
+0 %{ print "cwnd @4: ",tcpi_snd_cwnd }%  // prints 1
// fast retransmit
+0 > .  1:1001(1000) ack 1


+0.01  < .  1:1(0) ack 4001 win 50000 // reception of 1:1001
+0 %{ print "cwnd @5: ",tcpi_snd_cwnd }%  // prints 2
+0 > .  4001:5001(1000) ack 1
+0 > .  5001:6001(1000) ack 1

+0.01  < .  1:1(0) ack 6001 win 50000
+0 %{ print "cwnd @6: ",tcpi_snd_cwnd }%  // prints 4
+0 > .  6001:7001(1000) ack 1
+0 > P.  7001:8001(1000) ack 1

+0.01  < .  1:1(0) ack 8001 win 50000
+0 %{ print "cwnd @7: ",tcpi_snd_cwnd }%  // prints 4
```

4.13.2 Open questions

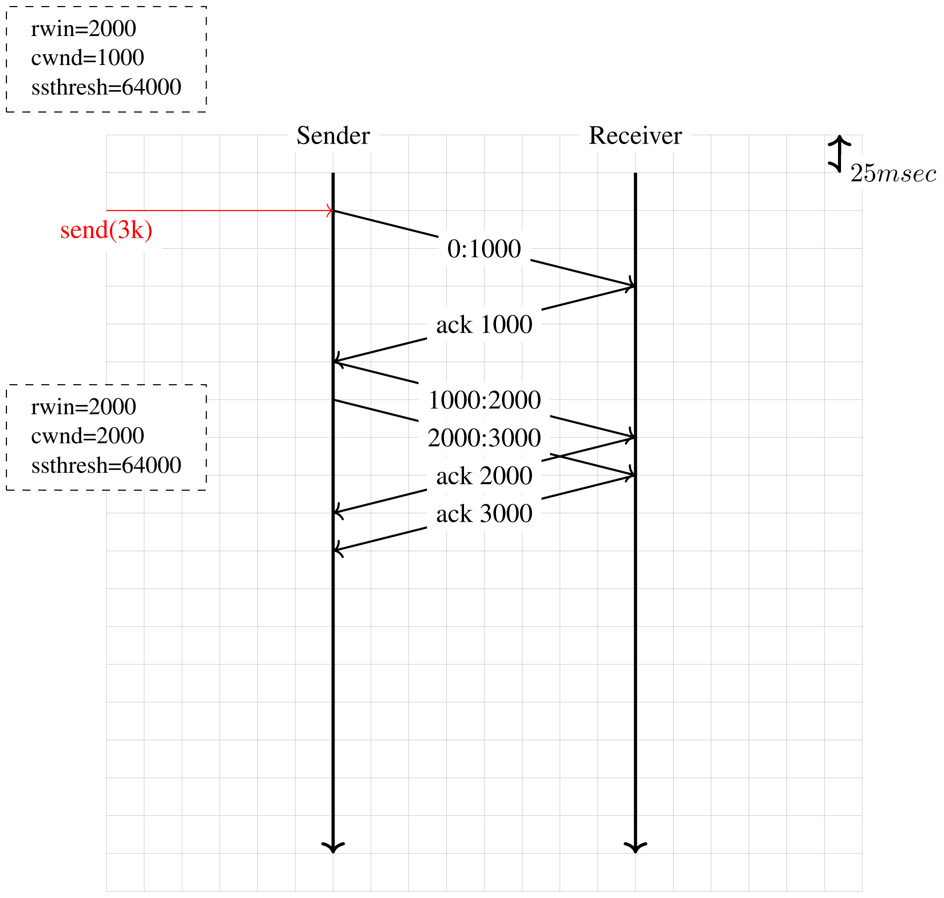
Unless otherwise noted, we assume for the questions in this section that the following conditions hold.

- the sender/receiver performs a single *send(3)* of *x* bytes
- the round-trip-time is fixed and does not change during the lifetime of the TCP connection. We assume a fixed value of 100 milliseconds for the round-trip-time and a fixed value of 200 milliseconds for the retransmission timer.
- the delay required to transmit a single TCP segment containing MSS bytes is small and set to 1 milliseconds,

independently of the MSS size

- the transmission delay for a TCP acknowledgment is negligible
- the initial value of the congestion window is one MSS-sized segment
- the value of the duplicate acknowledgment threshold is fixed and set to 3
- TCP always acknowledges each received segment

1. To understand the operation of the TCP congestion control, it is often useful to write time-sequence diagrams for different scenarios. The example below shows the operation of the TCP congestion control scheme in a very simple scenario. The initial congestion window (`cwnd`) is set to 1000 bytes and the receive window (`rwin`) advertised by the receiver (supposed constant for the entire connection) is set to 2000 bytes. The slow-start threshold (`ssthresh`) is set to 64000 bytes.

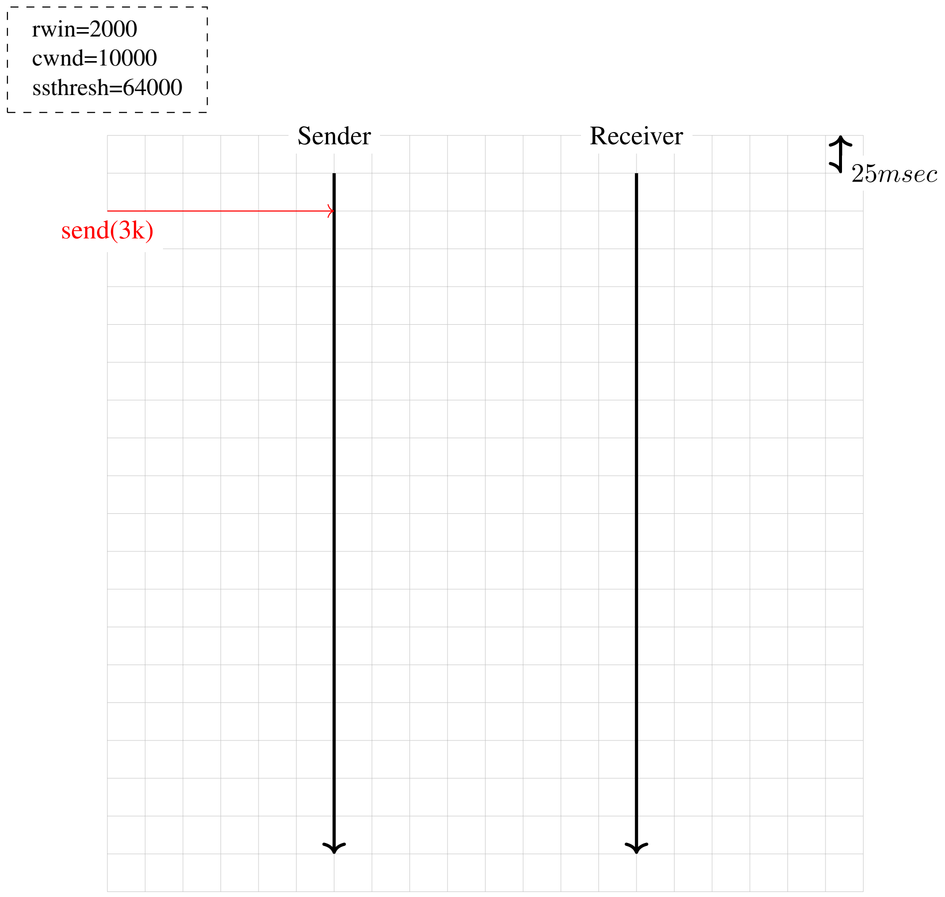


   a. Can you explain why the sender only sends one segment first and then two successive segments (the delay between the two segments on the figure is due to graphical reasons) ?

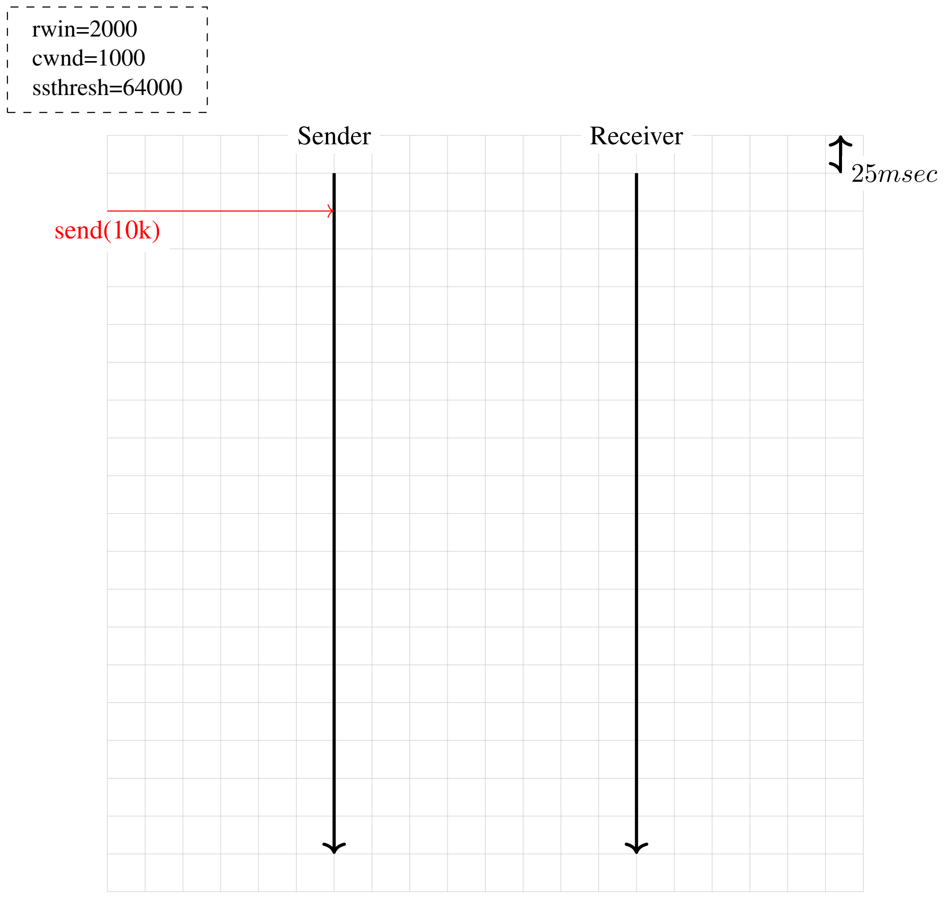
   b. Can you explain why the congestion window is increased after the reception of the first acknowledgment ?

   c. How long does it take for the sender to deliver 3 KBytes to the receiver ?

2. Same question as above but now with a small variation. Recent TCP implementations use a large initial value for the congestion window. Draw the time-sequence diagram that corresponds to an initial value of 10000 bytes for this congestion window.

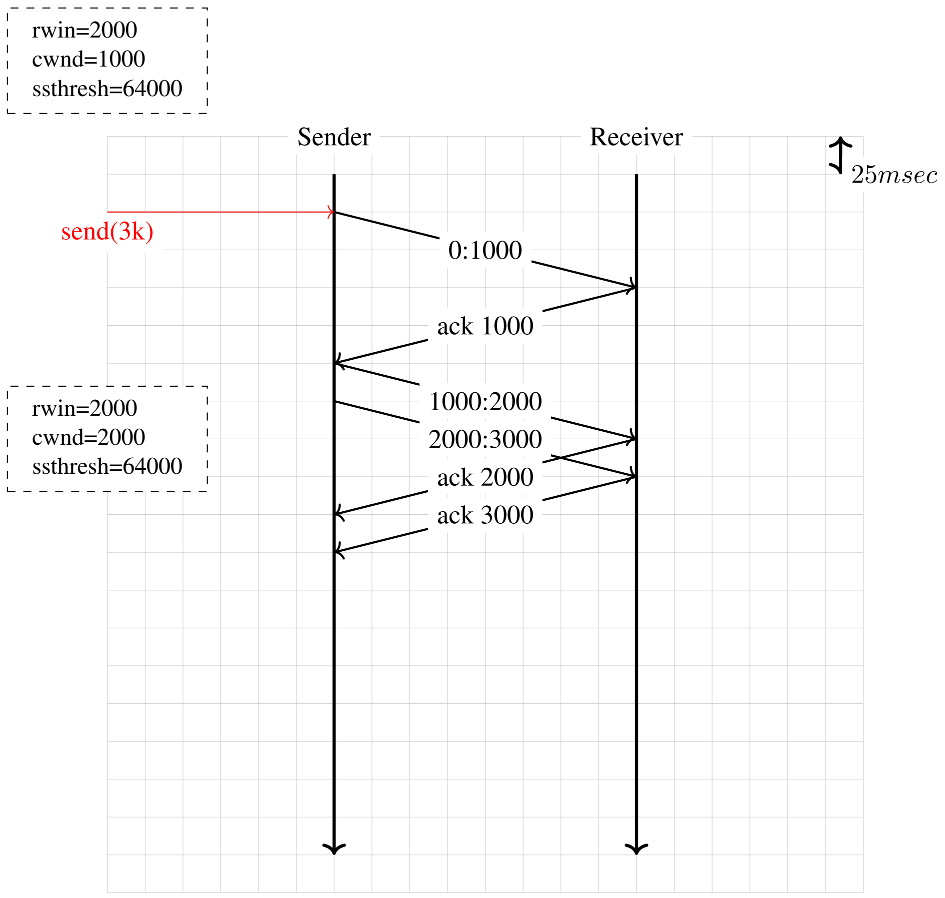


3. Same question as the first one, but consider that the MSS on the sender is set to 500 bytes. How does this modification affect the entire delay ?

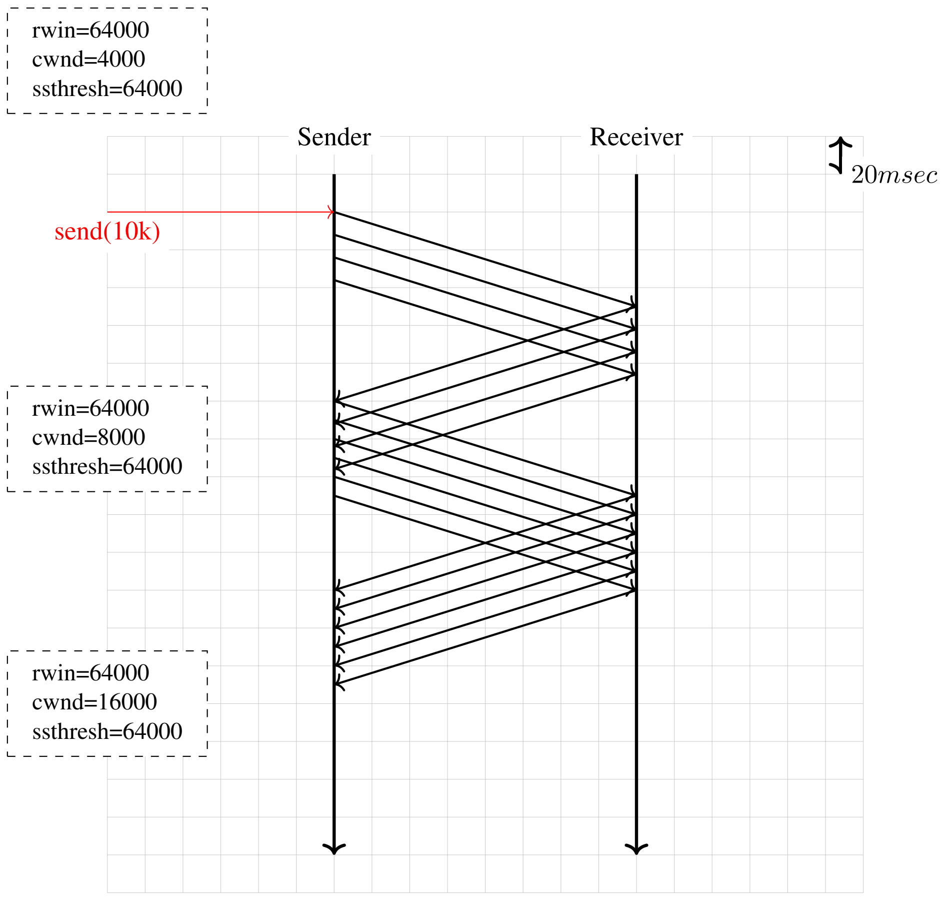


4. Assuming that there are no losses and that there is no congestion in the network. If the sender writes x bytes on a newly established TCP connection, derive a formula that computes the minimum time required to deliver all these x bytes to the receiver. For the derivation of this formula, assume that x is a multiple of the maximum segment size and that the receive window and the slow-start threshold are larger than x.

5. In question 1, we assumed that the receiver acknowledged every segment received from the sender. In practice, many deployed TCP implementations use delayed acknowledgments. Assuming a delayed acknowledgment timer of 50 milliseconds, modify the time-sequence diagram below to reflect the impact of these delayed acknowledgment. Does their usage decreases or increased the transmission delay ?



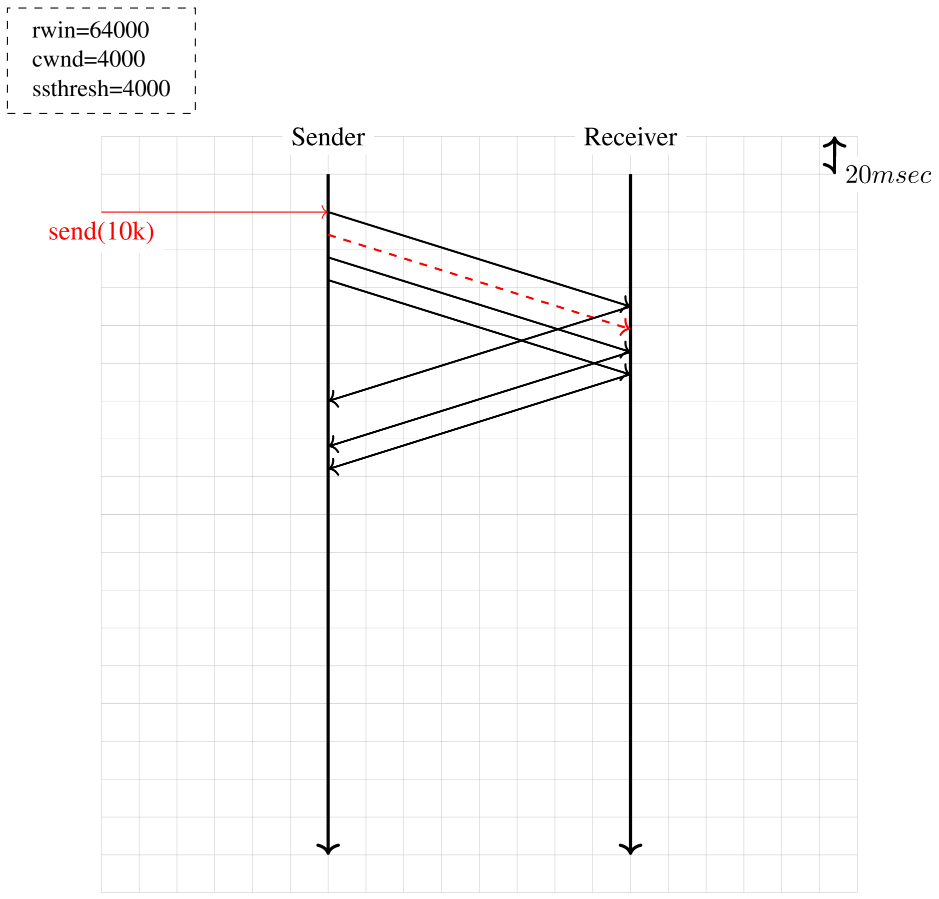
6. Let us now explore the impact of congestion on the slow-start and congestion avoidance mechanisms. Consider the scenario below. For graphical reasons, it is not possible anymore to show information about the segments on the graph, but you can easily infer them.



a. Redraw the same figure assuming that the second segment that was delivered by the sender in the figure experienced congestion. In a network that uses Explicit Congestion Notification, this segment would be marked by routers and the receiver would return the congestion mark in the corresponding acknowledgment.

b. Same question, but assume now that the fourth segment delivered by the sender experienced congestion (but was not discarded).

7. A TCP connection has been active for some time and has reached a congestion window of 4000 bytes. Four segments are sent, but the second (shown in red in the figure) is corrupted. Complete the time-sequence diagram.



4.14 IPv6 Networks

4.14.1 Basic questions on IPv6 Networks

Before starting to determine the paths that packets will follow in an IPv6 network, it is important to remember how to convert IPv6 addresses in binary numbers.

An IPv6 forwarding table contains a list of IPv6 prefixes with their associated nexthop or outgoing interface. When an IPv6 router receives a packet, it forwards it according to its forwarding table. Note that IPv6 routers forward packets along the *longest match* between the destination address of the packet and the routes in the forwarding table.

Now that you master the basics, you can determine the paths followed by IPv6 packets in simple networks.

4.14.2 Design questions

1. Consider the network shown in the figure below. In this network, the following addresses are used.

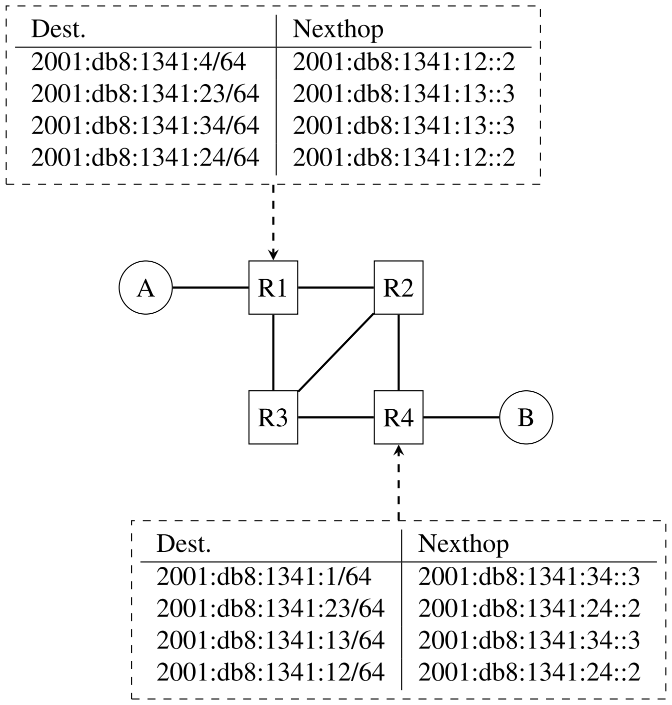
- host `A` : `2001:db8:1341:1::A` and its default route points to `2001:db8:1341:1::1`
- host `B` : `2001:db8:1341:4::B` and its default route points to `2001:db8:1341:4::4`

The routers have one address inside each network :

- router `R1` uses address `2001:db8:1341:1::1` on its West interface, address `2001:db8:1341:12::1` on its East interface and address `2001:db8:1341:13::1` on its South interface

- router `R2` uses address `2001:db8:1341:12::2` on its West interface, address `2001:db8:1341:23::2` on its South-West interface and address `2001:db8:1341:24::2` on its South interface.
- router `R3` uses address `2001:db8:1341:34::3` on its East interface, address `2001:db8:1341:23::3` on its North-East interface and address `2001:db8:1341:13::3` on its North interface
- router `R4` uses address `2001:db8:1341:34::4` on its West interface, address `2001:db8:1341:24::4` on its North interface and address `2001:db8:1341:4::4` on its East interface

The forwarding paths used in a network depend on the forwarding tables installed in the network nodes. Sometimes, these forwarding tables must be configured manually.

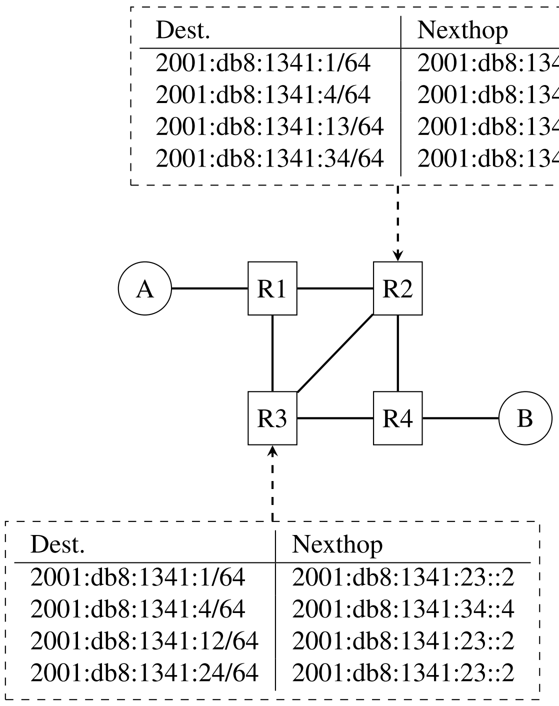


Dest.	Nexthop
2001:db8:1341:4/64	2001:db8:1341:12::2
2001:db8:1341:23/64	2001:db8:1341:13::3
2001:db8:1341:34/64	2001:db8:1341:13::3
2001:db8:1341:24/64	2001:db8:1341:12::2

Dest.	Nexthop
2001:db8:1341:1/64	2001:db8:1341:34::3
2001:db8:1341:23/64	2001:db8:1341:24::2
2001:db8:1341:13/64	2001:db8:1341:34::3
2001:db8:1341:12/64	2001:db8:1341:24::2

In this network, propose the forwarding tables of `R2` and `R3` that ensure that hosts `A` and `B` can exchange packets in both directions.

2. Consider the same network as in the previous question, but now the forwarding tables of `R2` and `R3` are configured as shown below :



Dest.	Nexthop
2001:db8:1341:1/64	2001:db8:1341:12::1
2001:db8:1341:4/64	2001:db8:1341:23::3
2001:db8:1341:13/64	2001:db8:1341:23::3
2001:db8:1341:34/64	2001:db8:1341:23::3

Dest.	Nexthop
2001:db8:1341:1/64	2001:db8:1341:23::2
2001:db8:1341:4/64	2001:db8:1341:34::4
2001:db8:1341:12/64	2001:db8:1341:23::2
2001:db8:1341:24/64	2001:db8:1341:23::2

In this network, select *all* the rules in the shown forwarding tables that ensure that the packets sent from `A` to `B` follow the reverse path of the packets sent by `B` to `A`.

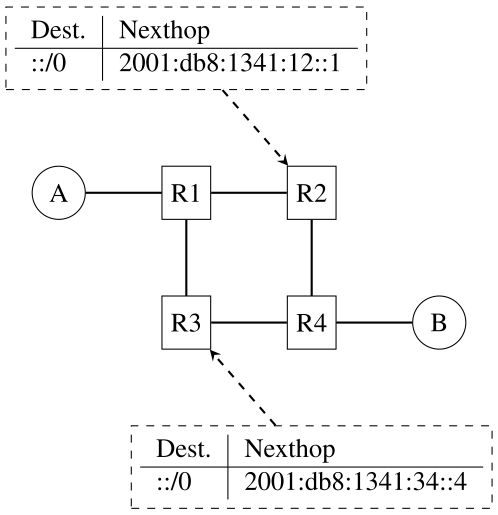
3. Consider the network shown in the figure below. In this network, the following addresses are used.

- host `A` : `2001:db8:1341:1::A` and its default route points to `2001:db8:1341:1::1`
- host `B` : `2001:db8:1341:4::B` and its default route points to `2001:db8:1341:4::4`

The routers have one address inside each network :

- router `R1` uses address `2001:db8:1341:1::1` on its West interface, address `2001:db8:1341:12::1` on its East interface and address `2001:db8:1341:13::1` on its South interface
- router `R2` uses address `2001:db8:1341:12::2` on its West interface, and address `2001:db8:1341:24::2` on its South interface
- router `R3` uses address `2001:db8:1341:34::3` on its East interface and address `2001:db8:1341:13::3` on its North interface
- router `R4` uses address `2001:db8:1341:34::4` on its West interface, address `2001:db8:1341:24::4` on its North interface and address `2001:db8:1341:4::4` on its East interface

Routers `R2` and `R3` are buggy in this network. Besides the routes for their local interfaces (not shown in the figure), they only have a default route which is shown in the figure below.

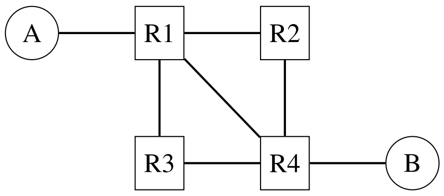


Dest.	Nexthop
::/0	2001:db8:1341:12::1

Dest.	Nexthop
::/0	2001:db8:1341:34::4

How do you configure the forwarding tables on `R1` and `R4` so that `A` can reach `B` and the reverse ?

4. Consider a slightly different network than in the previous question.



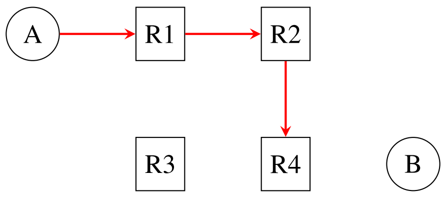
Assuming that the following IPv6 addresses are used :

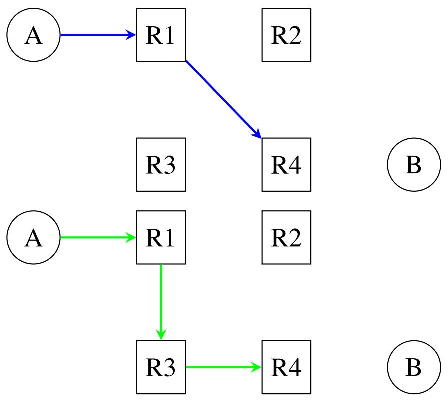
- host `A` : `2001:db8:1341:1::A` and its default route points to `2001:db8:1341:1::1`
- host `B` : `2001:db8:1341:4::B` and its default route points to `2001:db8:1341:4::4`

The routers have one address inside each network :

- router `R1` uses address `2001:db8:1341:1::1` on its West interface, address `2001:db8:1341:12::1` on its East interface, address `2001:db8:1341:14::1` on its South-East interface and address `2001:db8:1341:13::1` on its South interface
- router `R2` uses address `2001:db8:1341:12::2` on its West interface, and address `2001:db8:1341:24::2` on its South interface
- router `R3` uses address `2001:db8:1341:34::3` on its East interface and address `2001:db8:1341:13::3` on its North interface
- router `R4` uses address `2001:db8:1341:34::4` on its West interface, address `2001:db8:1341:24::4` on its North interface, address `2001:db8:1341:14::4` on its North-West interface and address `2001:db8:1341:4::4` on its East interface

Can you configure the forwarding tables so that the following paths are used by packets sent by host `A` to reach one of the four addresses of router `R4`?

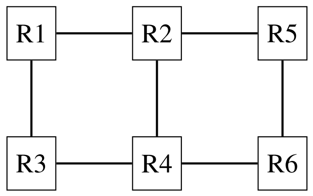




Do your forwarding tables impose the path used to reach host B which is attached to router R4 or do you need to configure an additional entry in these tables ?

5. Consider the network below that contains only routers. This network has been configured by a group of students and you must verify whether the configuration is correct. All the IPv6 addresses are part of the same /48 prefix that we name p. The following subnets are defined in this /48 prefix.

- p:12/64 for the link between R1 and R2. On this subnet, R1 uses address p:12::1 while router R2 uses address p:12::2
- p:13/64 for the link between R1 and R3. On this subnet, R1 uses address p:13::1 while router R3 uses address p:13::3
- p:24/64 for the link between R2 and R4. On this subnet, R2 uses address p:24::2 while router R4 uses address p:24::4
- ...



The students have configured the following forwarding tables on these six routers.

- on router R1

Dest.	Nexthop/Interface
::/0	p:12::2
p:12::/64	East
p:13::/64	South
p:25::/64	p:12::2
p:34::/64	p:12::2

- on router R2

Dest.	Nexthop/Interface
::/0	p:12::1
p:12::/64	West
p:13::/64	p:24::4
p:24::/64	South
p:25::/64	East
p:56::/64	p:24::4

- on router R3

Dest.	Nexthop/Interface
::/0	p:13::1
p:13::/64	North
p:34::/64	East
p:56::/64	p:34::4

- on router R5

Dest.	Nexthop/Interface
::/0	p:56::6
p:12::/64	p:25::2
p:25::/64	West
p:56::/64	South

- on router R4

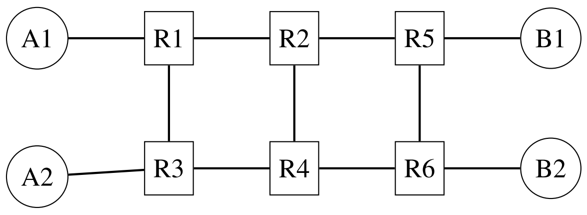
Dest.	Nexthop/Interface
p:12::/63	p:24::2
p:24::/64	North
p:25::/64	p:46::6
p:34::/64	West
p:46::/64	East

- on router R6

Dest.	Nexthop/Interface
::/0	p:56::5
p:13::/64	p:46::4
p:24::/63	p:46::4
p:34::/64	p:46::4
p:46::/64	West
p:56::/64	North

What do you think about the proposed configuration?

6. Sometimes, static routes must be configured on networks to enforce certain paths. Consider the six routers network shown in the figure below.



In this network, we will focus on four IPv6 prefixes :

- `p:0000::/64` used on the link `A1-R1`. `A1` uses address `p:0000::A1/64`
- `p:0001::/64` used on the link `A2-R3`. `A2` uses address `p:0001::A2/64`
- `p:0002::/64` used on the link `B1-R5`. `B1` uses address `p:0002::B1/64`
- `p:0003::/64` used on the link `B2-R6`. `B2` uses address `p:0003::B2/64`

Can you configure the forwarding tables of the six routers to achieve the following network objectives :

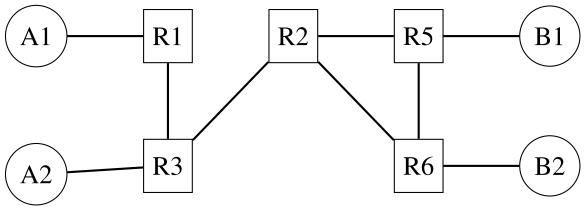
a. All packets sent by B1 and B2 to A1 and A2 are always forwarded via R2 while all packets from A1 and A2 are always forwarded via R4

b. The packets whose destinations are A1, A2, B1 or B2 are never forwarded via router R4

c. The packets sent by A1 or A2 towards B1 are always forwarded via R2 while the packets towards B2 are always forwarded via R4.

When creating these forwarding tables, try to minimize the number of entries that you install on each router.

7. When a network is designed, an important element of the design is the IP address allocation plan. A good allocation plan can provide flexibility and help to reduce the size of the forwarding tables.



Assign IP subnets to all links in this network so that you can reduce the number of entries in the forwarding tables of all routers. Assume that you have received a /56 prefix that you can use as you want. Each subnet containing a host must be allocated a /64 subnet.

4.14.3 Configuring IPv6 Networks

With the previous exercises, you have learned how to reason about IPv6 networks "on paper". Given the availability of IPv6 implementations, it is also possible to carry out experiments in real and virtual labs. Several virtual environments are possible. In this section, we focus on mininet. mininet is an emulation framework developed at Stanford University that leverages the namespaces features of recent Linux kernels. With those namespaces, a single Linux kernel can support a variety of routers and hosts interconnected by virtual links. mininet has been used by several universities as an educational tool, but unfortunately it was designed without IPv6 support.

During the last years, Olivier Tilmans and Mathieu Jadin have developed the missing piece to enable students to use mininet to experiment with IPv6: ipmininet. ipmininet is a python module that provides the classes that are required to automatically configure IPv6 networks with different routing protocols. It is available from PyPi from https://pypi.python.org/ipmininet.

The syntax of IPMininet is relatively simple and can be learned by looking at a few examples.

Let us start our exploration of IPv6 routing with a simple network topology that contains two hosts and three routers and uses static routes. IPMininet simplifies the creation of the network topology by providing a simple API. For this, you simply need to declare a class that extends the IPTopo class.

```python
from ipmininet.iptopo import IPTopo
from ipmininet.router.config import RouterConfig, STATIC, StaticRoute
from ipmininet.ipnet import IPNet
from ipmininet.cli import IPCLI

class MyTopology(IPTopo):
    pass
```

Then, you need to extend the build method that creates routers and hosts.

```python
def build(self, *args, **kwargs):

    # The routers using static routes
```

(continues on next page)

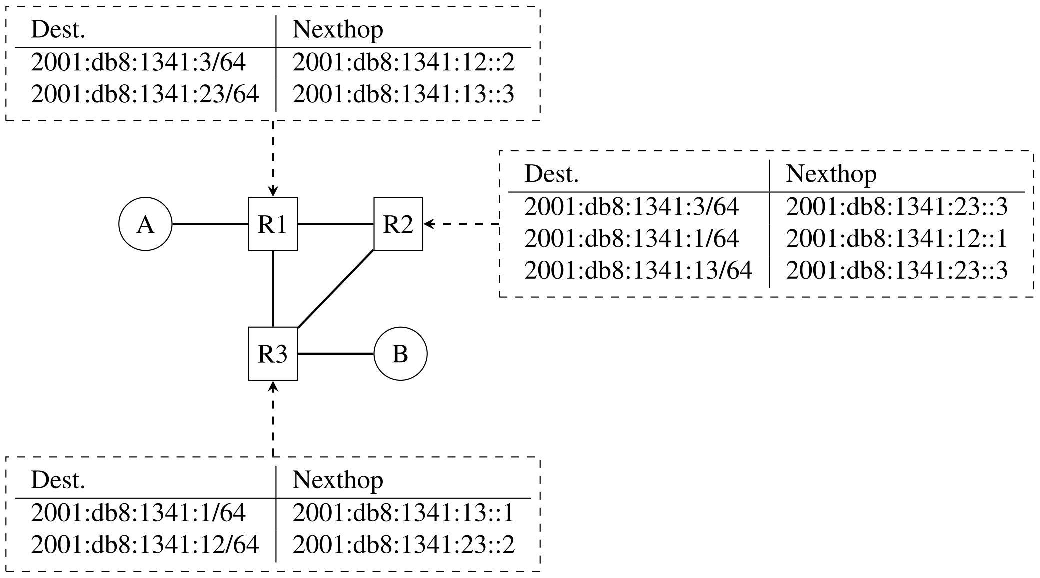


Fig. 10: A simple network

(continued from previous page)

```
    r1 = self.addRouter("r1", config=RouterConfig)
    r2 = self.addRouter("r2", config=RouterConfig)
    r3 = self.addRouter("r3", config=RouterConfig)
    # The hosts
    a = self.addHost("a")
    b = self.addHost("b")
```

Although IPMininet can assign prefixes and addresses automatically, we use manually assigned addresses in this example.

We use five /64 IPv6 prefixes in this network topology:

- `2001:db8:1341:1::/64` on the link between `a` and `r1`
- `2001:db8:1341:12::/64` on the link between `r1` and `r2`
- `2001:db8:1341:13::/64` on the link between `r1` and `r3`
- `2001:db8:1341:23::/64` on the link between `r2` and `r3`
- `2001:db8:1341:1::/64` on the link between `b` and `r3`

We can then manually configure the IPv6 addresses of each host/router on each link. Let us start with the links attached to the two hosts.

```
# link between r1 and a
lr1a = self.addLink(r1, a)
lr1a[r1].addParams(ip=("2001:db8:1341:1::1/64"))
lr1a[a].addParams(ip=("2001:db8:1341:1::A/64"))
# link between r3 and b
lr3b = self.addLink(r3, b)
lr3b[r3].addParams(ip=("2001:db8:1341:3::3/64"))
lr3b[b].addParams(ip=("2001:db8:1341:3::B/64"))
```

The same can be done for the three links between the different routers.

```
lr1r2 = self.addLink(r1, r2)
lr1r2[r1].addParams(ip=("2001:db8:1341:12::1/64"))
lr1r2[r2].addParams(ip=("2001:db8:1341:12::2/64"))

lr1r3 = self.addLink(r1, r3)
lr1r3[r1].addParams(ip=("2001:db8:1341:13::1/64"))
lr1r3[r3].addParams(ip=("2001:db8:1341:13::3/64"))

lr2r3 = self.addLink(r2, r3)
lr2r3[r2].addParams(ip=("2001:db8:1341:23::2/64"))
lr2r3[r3].addParams(ip=("2001:db8:1341:23::3/64"))
```

With these IP prefixes and the network topology, we can now use IPMininet to create the topology and assign the addresses.

We start by creating the objects that correspond to the static routes on the three routers. The second argument of the `addDaemon` method is a list of `StaticRoute` objects. Each of these objects is created by specifying an IP prefix and a nexthop.

```
# Add static routes
r1.addDaemon(STATIC,
             static_routes=[StaticRoute("2001:db8:1341:3::/64","2001:db8:1341:12::2"),
                            StaticRoute("2001:db8:1341:23::/64","2001:db8:1341:13::3
↪")])

r2.addDaemon(STATIC,
             static_routes=[StaticRoute("2001:db8:1341:3::/64", "2001:db8:1341:23::3
↪"),
                            StaticRoute("2001:db8:1341:1::/64", "2001:db8:1341:12::1
↪"),
                            StaticRoute("2001:db8:1341:13::/64", "2001:db8:1341:23::3
↪")])

r3.addDaemon(STATIC,
             static_routes=[StaticRoute("2001:db8:1341:1::/64", "2001:db8:1341:13::1
↪"),
                            StaticRoute("2001:db8:1341:12::/64","2001:db8:1341:23::2
↪")])
```

We can now create the hosts and the routers

```
super(MyTopology, self).build(*args, **kwargs)
```

With this `build` method, we can now launch the network by using the python code below.

```
net = IPNet(topo=MyTopology(), allocate_IPs=False)  # Disable IP auto-allocation
try:
   net.start()
   IPCLI(net)
finally:
   net.stop()
```

The entire script is available from `/exercises/ipmininet_scripts/static-1.py`.

To help students to start using IPMininet, Mathieu Jadin has created a Vagrant box that launches a Ubuntu virtual machine with all the required software. See https://ipmininet.readthedocs.io/en/latest/install.html for additional information.

Here is a simple example of the utilization of this Vagrant box.

We start the network topology shown above with the `sudo python script.py` command. It launches the mininet interactive shell that provides several useful commands:

```
mininet> help

Documented commands (type help <topic>):
========================================
EOF    gterm  iperf     links   pingall       ports  route   time
dpctl  help   iperfudp  net     pingallfull   px     sh      x
dump   intfs  ips       nodes   pingpair      py     source  xterm
exit   ip     link      noecho  pingpairfull  quit   switch

You may also send a command to a node using:
   <node> command {args}
For example:
   mininet> h1 ifconfig

The interpreter automatically substitutes IP addresses
for node names when a node is the first arg, so commands
like
    mininet> h2 ping h3
should work.

Some character-oriented interactive commands require
noecho:
   mininet> noecho h2 vi foo.py
However, starting up an xterm/gterm is generally better:
   mininet> xterm h2

mininet>
```

Some of the standard mininet commands assume the utilisation of IPv4 and do not have a direct IPv6 equivalent. Here are some useful commands.

The `nodes` command lists the routers and hosts that have been created in the mininet topology.

```
mininet> nodes
available nodes are:
a b r1 r2 r3
```

The `links` command lists the links that have been instantiated and shows that mapping between the named interfaces on each node.

```
mininet> links
r1-eth2<->a-eth0 (OK OK)
r1-eth0<->r2-eth0 (OK OK)
r1-eth1<->r3-eth0 (OK OK)
r2-eth1<->r3-eth1 (OK OK)
r3-eth2<->b-eth0 (OK OK)
mininet>
```

It is possible to execute any of the standard Linux commands to configure the network stack on any of the hosts by prefixing the command with the corresponding host. Remember to always specify `inet6` as the address family to retrieve the IPv6 information.

```
mininet> a ip -f inet6 link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT␣
↪group default qlen 1
```

(continues on next page)

(continued from previous page)

```
 link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: a-eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode␣
↪DEFAULT group default qlen
 link/ether c6:4e:26:d9:de:6d brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Host a has two interfaces: the standard loopback interface and a network interface named a-eth0 that is attached to router r1. We can also verify how the IPv6 addresses have been configured:

```
mininet> a ip -f inet6 address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 state UNKNOWN qlen 1
   inet6 ::1/128 scope host
     valid_lft forever preferred_lft forever
2: a-eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP qlen 1000
   inet6 2001:db8:1341:1::a/64 scope global
     valid_lft forever preferred_lft forever
   inet6 fe80::c44e:26ff:fed9:de6d/64 scope link
     valid_lft forever preferred_lft forever
```

On its a-eth0 interface, host a uses IPv6 address 2001:db8:1341:1::a/64. The link local address (fe80::c44e:26ff:fed9:de6d/64) will be described in another chapter. Finally, we can check the forwarding table of host a.

```
mininet> a ip -f inet6 route
2001:db8:1341:1::/64 dev a-eth0  proto kernel  metric 256  pref medium
fe80::/64 dev a-eth0  proto kernel  metric 256  pref medium
default via 2001:db8:1341:1::1 dev a-eth0  metric 1024  pref medium
```

There are three routes in this table. The first two correspond to the two prefixes that are used over the a-eth0 interface. These routes are automatically created when an IPv6 address is configured on an interface. The last route is the default route (::/0) which points towards 2001:db8:1341:1::1, i.e. router r1.

Another useful command is xterm 'node' that allows to launch a terminal on the specified node. This gives you a interactive shell on any node. You can use it to capture packets with tcpdump. As an example, let us use *traceroute6(8)* to trace the path followed by packets from host a towards the IPv6 address of host b i.e. 2001:db8:1341:3::b. The output of this command shows that the path passes through routers r1, r2 and r3.

```
mininet> a traceroute6 -q 1 2001:7ab:3::c
traceroute to 2001:7ab:3::c (2001:7ab:3::c) from 2001:7ab:1::a, 30 hops max, 16 byte␣
↪packets
1  2001:7ab:1::1 (2001:7ab:1::1)  0.105 ms
2  2001:89ab:12::2 (2001:89ab:12::2)  1.131 ms
3  2001:89ab:23::2 (2001:89ab:23::2)  0.845 ms
4  2001:7ab:3::c (2001:7ab:3::c)  0.254 ms
```

Another interesting mininet command is pingall it allows to check that any host can reach any other host inside the network. It executes a ping from any host to any other host inside the network topology.

```
mininet> pingall
*** Ping: testing reachability over IPv4 and IPv6
a --IPv6--> b
b --IPv6--> a
*** Results: 0% dropped (2/2 received)
```

When debugging a network, it can be interesting to capture packets using tcpdump on specific links to check that they follow the expect. If you use tcpdump without any filter, you will capture the packets generated by xterm. To capture packets, you need to specify precise filters that will match the packets of interest. For traceroute6, you need to match the IPv6 packets that contain UDP segments and some ICMPv6 packets. The script below provides a simple filter that you can reuse. It takes one argument: the name of the interface on which tcpdump needs to run.

```
#!/bin/bash

tcpdump -v -i $1 -n '(ip6 && udp) || (icmp6  && (ip6[40] == 1 || ip6[40]==3))'
```

Starting from the `/exercises/ipmininet_scripts/static-1.py` IPMininet script, we can explore classical problems when networks are configured with static routes. A first problem is when a router has an incomplete forwarding table. We configure the static routes as shown below. The entire script is available from `/exercises/ipmininet_scripts/static-1-hole.py`.

```
  # Add static routes
  r1.addDaemon(STATIC, static_routes=[StaticRoute("2001:db8:1341:3::/64",
"2001:db8:1341:12::2")])

  r2.addDaemon(STATIC, static_routes=[StaticRoute("2001:db8:1341:1::/64",
"2001:db8:1341:12::1")])

  r3.addDaemon(STATIC, static_routes=[StaticRoute("2001:db8:1341:1::/64",
"2001:db8:1341:13::1")])
```

We first check with `pingall` whether the network works correctly.

```
mininet> pingall
*** Ping: testing reachability over IPv4 and IPv6
a --IPv6--> X
b --IPv6--> X
*** Results: 100% dropped (0/2 received)
```

The problem can be detected by using *traceroute6(8)*.

```
mininet> a traceroute6 -q 1 -n 2001:db8:1341:3::b
traceroute to 2001:db8:1341:3::b (2001:db8:1341:3::b) from 2001:db8:1341:1::a, 30␣
↪hops max, 24 byte packets
1  2001:db8:1341:1::1  0.074 ms
2  2001:db8:1341:12::2  0.042 ms !N
```

In the output of *traceroute6(8)*, a `!N` indicates that host a received from `2001:db8:1341:12::2`, i.e. router `r2`, a Network unreachable ICMPv6 message. The forwarding table of `r2` confirms the root cause of this problem.

```
mininet> r2 ip -f inet6 route show
2001:db8:1341:1::/64 via 2001:db8:1341:12::1 dev r2-eth0  proto 196  metric 20  pref␣
↪medium
2001:db8:1341:12::/64 dev r2-eth0  proto kernel  metric 256  pref medium
2001:db8:1341:23::/64 dev r2-eth1  proto kernel  metric 256  pref medium
fe80::/64 dev r2-eth0  proto kernel  metric 256  pref medium
fe80::/64 dev r2-eth1  proto kernel  metric 256  pref medium
```

A second problem is when there is a forwarding loop inside the network, i.e. packets sent to a specific destination loop through several routers. With the static routes shown below, router `r2` forwards the packets towards `2001:db8:1341:3::b` via router `r1`. The entire script is available from `/exercises/ipmininet_scripts/static-1-loop.py`.

```
# Add static routes
r1.addDaemon(STATIC,
```

(continues on next page)

(continued from previous page)

```
             static_routes=[StaticRoute("2001:db8:1341:3::/64","2001:db8:1341:12::2
↪")])

r2.addDaemon(STATIC,
             static_routes=[StaticRoute("2001:db8:1341::/60", "2001:db8:1341:12::1"),
                            StaticRoute("2001:db8:1341:1::/64","2001:db8:1341:12::1
↪")])

r3.addDaemon(STATIC,
             static_routes=[StaticRoute("2001:db8:1341:1::/64", "2001:db8:1341:13::1
↪")])
```

The `pingall` command reveals that there is a problem in this network.

```
mininet>pingall
*** Ping: testing reachability over IPv4 and IPv6
a --IPv6--> X
b --IPv6--> X
*** Results: 100% dropped (0/2 received)
```

We can analyze this configuration problem in more details by using `traceroute6`. The loop appears clearly.

```
mininet>a traceroute6 -q 1 -n 2001:db8:1341:3::b
traceroute to 2001:db8:1341:3::b (2001:db8:1341:3::b) from 2001:db8:1341:1::a, 30␣
↪hops max, 24 byte packets
1  2001:db8:1341:1::1  0.102 ms
2  2001:db8:1341:12::2  0.225 ms
3  2001:db8:1341:1::1  0.201 ms
4  2001:db8:1341:12::2  0.075 ms
5  2001:db8:1341:1::1  0.057 ms
6  2001:db8:1341:12::2  0.041 ms
7  2001:db8:1341:1::1  0.051 ms
8  2001:db8:1341:12::2  0.043 ms
9  2001:db8:1341:1::1  0.122 ms
10  2001:db8:1341:12::2  0.058 ms
11  2001:db8:1341:1::1  0.033 ms
12  2001:db8:1341:12::2  0.043 ms
^C
mininet>
```

On host `b`, the problem is different. The packets that it sends towards host `a` do not seem to go beyond router `r3`.

```
mininet> b traceroute6 -q 1 -n 2001:db8:1341:1::a
traceroute to 2001:db8:1341:1::a (2001:db8:1341:1::a) from 2001:db8:1341:3::b, 30␣
↪hops max, 24 byte packets
1  2001:db8:1341:3::3  0.091 ms
2  *
3  *
4  *
^C
mininet>
```

To debug this problem, let us look at the forwarding table of `r3`. This router forwards the packets sent to host `a` to router `r1` that is directly connected to host `a`.

```
mininet> r3 ip -f inet6 route show
2001:db8:1341:1::/64 via 2001:db8:1341:13::1 dev r3-eth0  proto 196  metric 20  pref␣
↪medium
2001:db8:1341:3::/64 dev r3-eth2  proto kernel  metric 256  pref medium
2001:db8:1341:13::/64 dev r3-eth0  proto kernel  metric 256  pref medium
2001:db8:1341:23::/64 dev r3-eth1  proto kernel  metric 256  pref medium
fe80::/64 dev r3-eth0  proto kernel  metric 256  pref medium
fe80::/64 dev r3-eth1  proto kernel  metric 256  pref medium
fe80::/64 dev r3-eth2  proto kernel  metric 256  pref medium
```

Unfortunately, when router `r1` sends its ICMP HopLimit exceeded message, the destination of this IP packet is `2001:db8:1341:3::b`. This packet is forward to router `r2` that returns the packet back to router `r1`. The packet loops between the two routers until their HopLimit reaches zero.

```
mininet> r1 ip -f inet6 route show
2001:db8:1341:1::/64 dev r1-eth2  proto kernel  metric 256  pref medium
2001:db8:1341:3::/64 via 2001:db8:1341:12::2 dev r1-eth0  proto 196  metric 20  pref␣
↪medium
2001:db8:1341:12::/64 dev r1-eth0  proto kernel  metric 256  pref medium
2001:db8:1341:13::/64 dev r1-eth1  proto kernel  metric 256  pref medium
fe80::/64 dev r1-eth2  proto kernel  metric 256  pref medium
fe80::/64 dev r1-eth0  proto kernel  metric 256  pref medium
fe80::/64 dev r1-eth1  proto kernel  metric 256  pref medium
mininet>
```

4.14.4 IPv6 packets

To correctly understand the operation of IPv6, it is sometimes important to remember the packet format and how the different fields are used.

The *Next Header* of the IPv6 packet indicates the type of the header that follows the IPv6 packet. IANA maintains a list of all the assigned values of this header at https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

When an IPv6 router receives a packet that is larger than the Maximum Transmission Unit (MTU) on its outgoing interface, it drops the packet and returns an ICMPv6 message back to the source. Upon reception of this ICMPv6 message, the source will either adjust the size of the packets that it transmits or use IPv6 packet fragmentation. The exercises below show a few examples of the utilization of IPv6 fragmentation.

Network engineers often rely on *ping6(8)* to verify the reachability of a remote host or router. *ping6(8)* sends ICMPv6 echo request messages and analyzes the received ICMPv6 echo responses. Each echo request message contains an identifier and a sequence number that is returned in the response.

When the *ping6(8)* is executed, it sends ICMPv6 echo request messages with increasing sequence numbers.

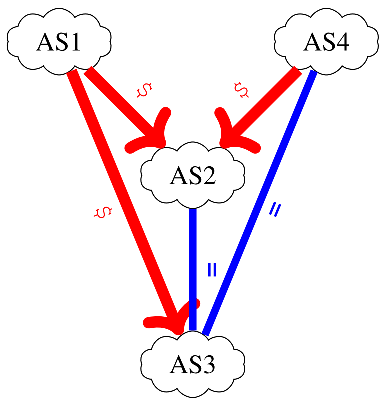
The *traceroute6(8)* software is very useful to debug network problems. It sends a series of UDP segments encapsulated inside IP packets with increasing values of the HopLimit. The first packet has a HotLimit and the first router on the path returns an ICMPv6 HopLimit exceeded message.

When *traceroute6(8)* sends UDP segments, it uses the UDP source port as a way to remember the target hop for this specific UDP segment.

4.15 Inter-domain routing

4.15.1 Exercises

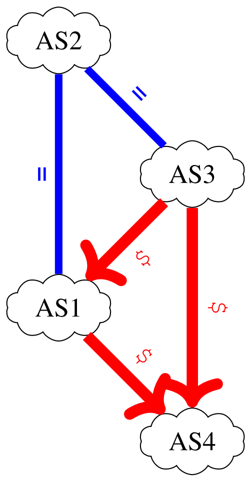
1. Consider the interdomain topology shown in the figure below.



In this network, what are the paths :

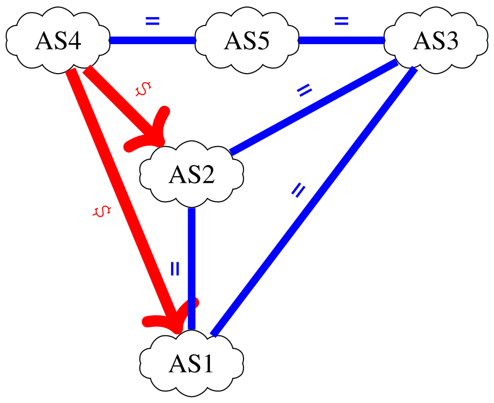
- from *AS1* to *AS4*
- from *AS4* to *AS2*
- from *AS4* to *AS1*

2. Consider the interdomain topology shown in the figure below. Assuming, that *AS1* advertises prefix `2001:db8:1::/48`, *AS2* prefix `2001:db8:2::/48`, ... compute the routing tables of the different ASes.



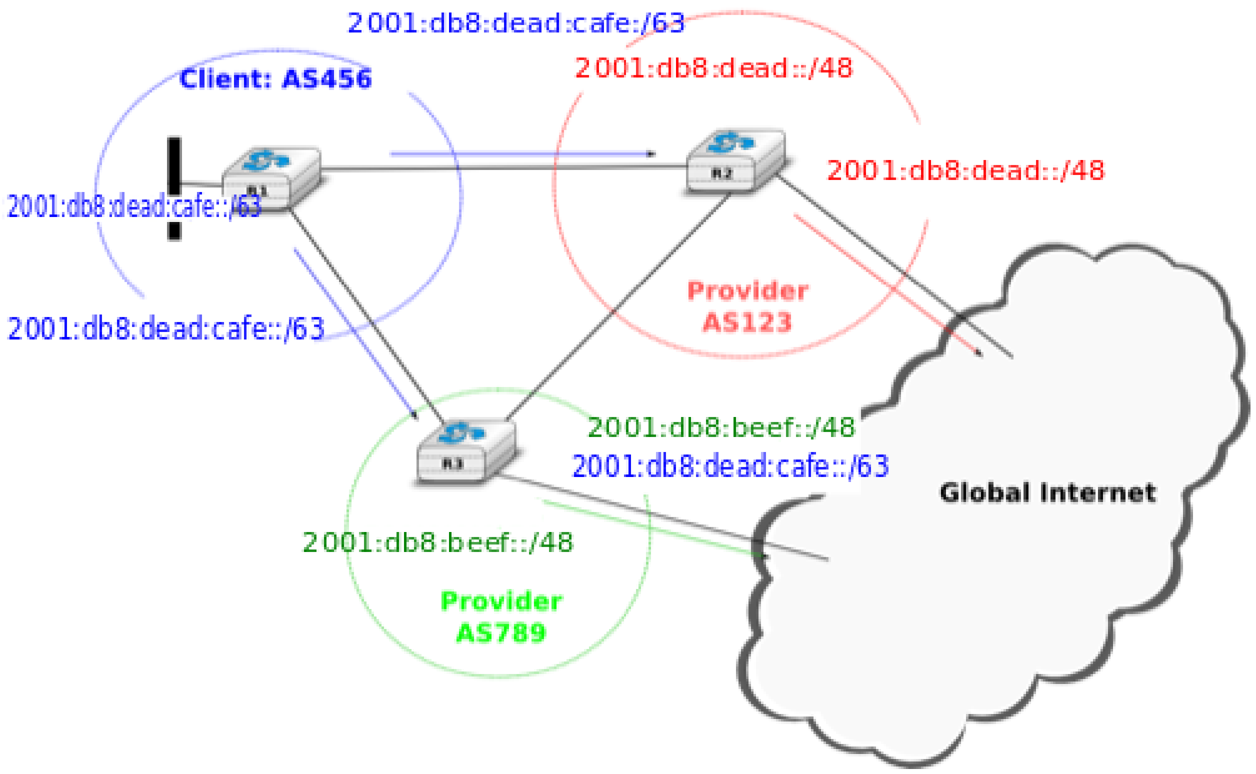
Are all ASes capable of reaching all the other ASes in this simple Internet ?

3. Consider the interdomain topology shown in the figure below. Assuming, that *AS1* advertises prefix `2001:db8:1::/48`, *AS2* prefix `2001:db8:2::/48`, ... compute the routing tables of the different ASes.



In this internet, some ASes cannot reach all other ASes. Can you fix the problem by adding one shared-cost peering link or one customer-provider peering link ?

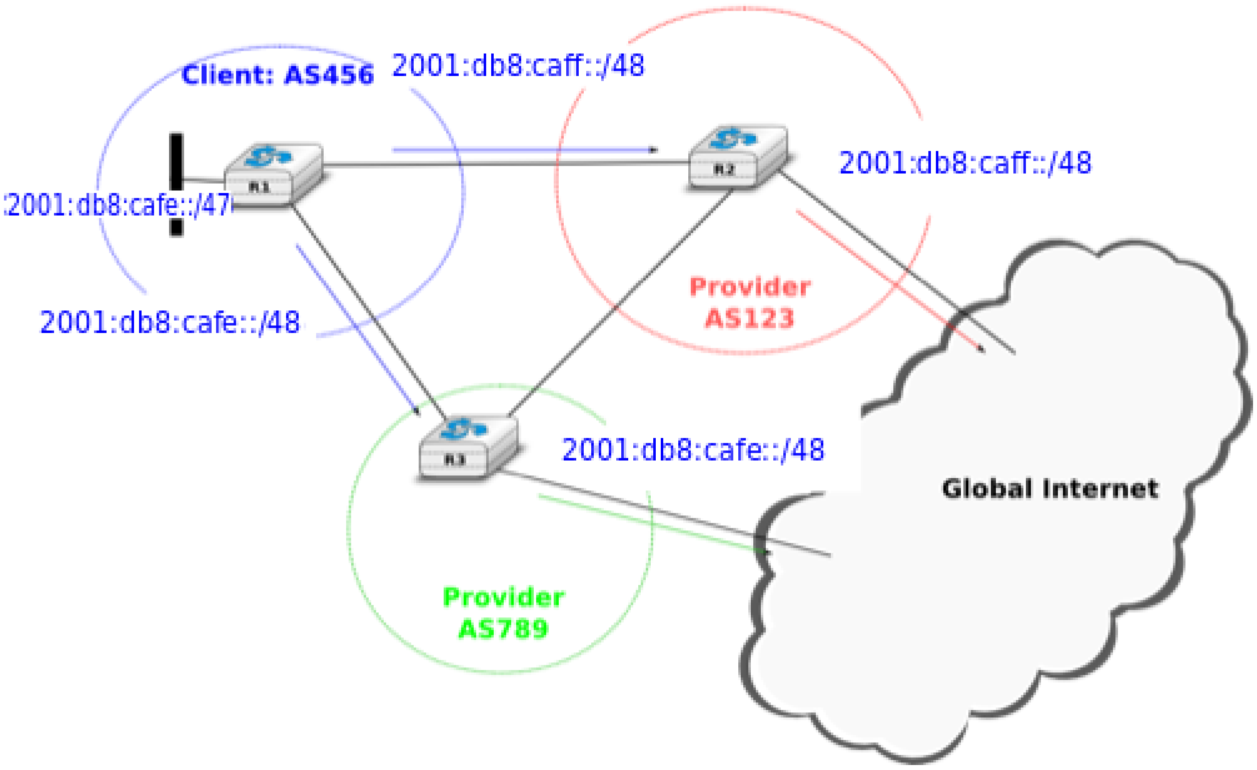
4. Consider the network below in which a stub domain, *AS456*, is connected to two providers *AS123* and *AS789*. *AS456* advertises its prefix to both its providers. On the other hand, *AS123* advertises `2001:db8:dead::/48` while *AS789* advertises `2001:db8:beef::/48` and `2001:db8:dead:cafe::/63`. Via which provider will the packets destined to `2001:db8:dead:cafe::1` will be received by *AS456* ?



Should *AS123* change its configuration ?

5. Consider that the AS stub (*AS456*) shown in the figure below decides to advertise two `/48` prefixes instead of its allocated `/47` prefix.

   - Via which provider does *AS456* receive the packets destined to `2001:db8:caff::bb` and `2001:db8:cafe::aa` ?



- How is the reachability of these addresses affected when link *R1-R3* fails ?
- Propose a configuration on R1 that achieves the same objective as the one shown in the figure but also preserves the reachability of all IP addresses inside *AS456* if one of *AS456*'s interdomain links fails.

6. Consider the network shown below. In this network, the metric of each link is set to *1* except link *A-B* whose metric is set to *4* in both directions. In this network, there are two paths with the same cost between *D* and *C*. Old routers would randomly select one of these equal cost paths and install it in their forwarding table. Recent routers are able to use up to *N* equal cost paths towards the same destination.

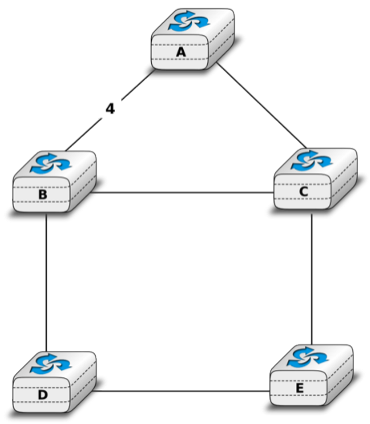


Fig. 11: A simple network

On recent routers, a lookup in the forwarding table for a destination address returns a set of outgoing interfaces. How would you design an algorithm that selects the outgoing interface used for each packet, knowing that to avoid reordering, all segments of a given TCP connection should follow the same path ?

7. A `traceroute6` towards `ipv6.google.com` provides the following output :

```
traceroute6 to ipv6.l.google.com (2a00:1450:4009:800::1001) from␣
↪2a02:2788:2c4:16f:5099:ccba:671d:e085, 64 hops max, 12 byte packets
1  2a02:2788:2c4:16f:a221:b7ff:fed8:aa90  4.777 ms  1.189 ms  1.023 ms
2  2a02:2788:2c0::1  8.746 ms  7.934 ms  10.024 ms
3  2a02:2788:2c0:3::1  10.039 ms  14.967 ms  8.943 ms
4  2a02:2788:ffff:12::1  9.808 ms  11.076 ms  13.658 ms
5  xe-4-2.r00.brslbe01.be.bb.gin.ntt.net  10.043 ms  10.408 ms  9.551 ms
6  ae-10.r02.amstnl02.nl.bb.gin.ntt.net  15.591 ms  18.416 ms  15.665 ms
7  core1.ams.net.google.com  21.259 ms  24.261 ms  20.826 ms
8  2001:4860::1:0:4b3  19.134 ms
   2001:4860::1:0:8  22.208 ms
   2001:4860::1:0:4b3  19.713 ms
9  2001:4860::8:0:519f  26.712 ms
   2001:4860::8:0:51a0  25.313 ms  19.392 ms
10  2001:4860::8:0:5bb8  24.197 ms
    2001:4860::8:0:5bb9  25.337 ms  26.264 ms
11  2001:4860::1:0:3067  29.431 ms  31.585 ms  29.260 ms
12  2001:4860:0:1::9  24.806 ms  24.297 ms  23.819 ms
13  lhr14s23-in-x01.1e100.net  29.406 ms  25.729 ms  29.160 ms
```

Can you explain why at the eighth, ninth and tenth hopes several IPv6 addresses are reported in the `traceroute6` output ?

8. Section 3.3 of **RFC 4443** explains two different reasons why an IPv6 enabled device could generate an ICMPv6 Time Exceeded message. Explain when a router could generate such a message with `Code==0` and when a host could generate such a message with `Code==1`.

9. Section 3.1 of **RFC 4443** seven different Codes for the ICMPv6 Destination Unreachable Message. Under which circumstances would a router generate such an ICMPv6 message with :

   - `Code==0`

10. An ICMPv6 error message includes in its message body the beginning of the IPv6 packet that triggered this error. How many bytes of the original packet must be returned to allow the host to recover the original source and destination addresses and source and destination ports of the packet that caused the error ?

4.16 Exploring routing protocols

Routing protocols play a key role in the Internet since they ensure that all the routers have valid routing tables. In this section, we use IPMininet to explore how intradomain and interdomain routing protocols work in practice. IPMininet adds an abstraction layer above the actual configuration of the FRRouting daemons that implement these routing protocols.

4.16.1 Exploring OSPF

We first use IPMininet to explore the operation of OSPFv3, the version of OSPF that supports IPv6. We create a simple network with three routers and two hosts as shown in the figure below.

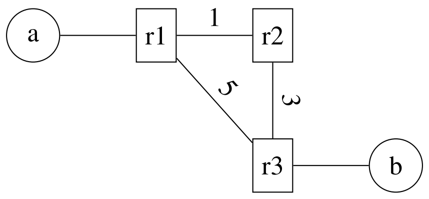


Fig. 12: A simple network

```
import shlex
from ipmininet.iptopo import IPTopo
from ipmininet.ipnet import IPNet
from ipmininet.cli import IPCLI

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):
        # Add routers (OSPF daemon is added by default with the default config)

        r1 = self.addRouter("r1")
        r2 = self.addRouter("r2")
        r3 = self.addRouter("r3")
        a = self.addHost("a")
        b = self.addHost("b")

        lr1r2 = self.addLink(r1, r2, igp_cost=1)
        lr1r2[r1].addParams(ip=("2001:db8:1341:12::1/64"))
        lr1r2[r2].addParams(ip=("2001:db8:1341:12::2/64"))

        lr1r3 = self.addLink(r1, r3, igp_cost=5)
        lr1r3[r1].addParams(ip=("2001:db8:1341:13::1/64"))
        lr1r3[r3].addParams(ip=("2001:db8:1341:13::3/64"))

        lr2r3 = self.addLink(r2, r3, igp_cost=3)
        lr2r3[r2].addParams(ip=("2001:db8:1341:23::2/64"))
        lr2r3[r3].addParams(ip=("2001:db8:1341:23::3/64"))

        lr1a = self.addLink(r1, a, igp_passive=True)
        lr1a[r1].addParams(ip=("2001:db8:1341:1::1/64"))
        lr1a[a].addParams(ip=("2001:db8:1341:1::A/64"))

        lr3b = self.addLink(r3, b, igp_passive=True)
        lr3b[r3].addParams(ip=("2001:db8:1341:3::3/64"))
        lr3b[b].addParams(ip=("2001:db8:1341:3::B/64"))

        super(MyTopology, self).build(*args, **kwargs)
```

The code is very simple as by default IPMininet enables OSPF on routers. We introduce two specific parameters. First, the interface that connects a router to a host is flagged as a passive interface (`igp_passive=True`). This indicates to the router that there are no other routers attached to this interface and that it should not send or accept OSPF Hello messages on this interface. The second configuration parameter is that we set the IGP cost on each link.

We use this mininet topology to collect packet traces that show the packets that OSPF routers exchange. For this, we add the following `post_build` method that IPMininet starts after having constructed the network and before launching the daemons. It simply starts tcpdump on each router to collect the first 100 OSPF packets that they send/receive.

```
def post_build(self, net):
    for r in self.routers():
        command="/usr/sbin/tcpdump --immediate-mode -c 100 -w ./"+r+"-trace.pcap␣
↪proto ospf"
        p = net[r].popen(shlex.split(command))

    super(MyTopology, self).post_build(net)
```

Finally, we can start the IPMininet topology and launch the daemons. The entire script is available from `/exercises/ipmininet_scripts/ospf6.py`.

```
net = IPNet(topo=MyTopology(), allocate_IPs=False)  # Disable IP auto-allocation
try:
    net.start()
    IPCLI(net)
finally:
    net.stop()
```

The script starts the routers and hosts.

```
mininet> nodes
available nodes are:
a b r1 r2 r3
mininet> links
r1-eth2<->a-eth0 (OK OK)
r1-eth0<->r2-eth0 (OK OK)
r1-eth1<->r3-eth0 (OK OK)
r2-eth1<->r3-eth1 (OK OK)
r3-eth2<->b-eth0 (OK OK)
```

We can easily verify that the paths used to forward packets are the expected ones according to the configured IGP weights.

```
mininet> a traceroute6 2001:db8:1341:3::B
traceroute to 2001:db8:1341:3::B (2001:db8:1341:3::b) from 2001:db8:1341:1::a, 30␣
↪hops max, 24 byte packets
1  2001:db8:1341:1::1 (2001:db8:1341:1::1)  0.203 ms  0.061 ms  0.045 ms
2  2001:db8:1341:13::3 (2001:db8:1341:13::3)  0.064 ms  0.055 ms  0.049 ms
3  2001:db8:1341:3::b (2001:db8:1341:3::b)  0.231 ms  0.057 ms  0.05 ms
mininet> b traceroute6 2001:db8:1341:1::A
traceroute to 2001:db8:1341:1::A (2001:db8:1341:1::a) from 2001:db8:1341:3::b, 30␣
↪hops max, 24 byte packets
1  2001:db8:1341:3::3 (2001:db8:1341:3::3)  0.088 ms  0.19 ms  0.053 ms
2  2001:db8:1341:13::1 (2001:db8:1341:13::1)  0.066 ms  0.059 ms  0.047 ms
3  2001:db8:1341:1::a (2001:db8:1341:1::a)  0.054 ms  0.059 ms  0.047 ms
```

We can also connect to the OSPFv3 daemon running on the routers to observe its state. For this, we use the `noecho r1 telnet localhost ospf6d` command.

```
mininet> noecho r1 telnet localhost ospf6d
Trying ::1...
```

(continues on next page)

(continued from previous page)

```
Connected to localhost.
Escape character is '^]'.

Hello, this is FRRouting (version 7.1).
Copyright 1996-2005 Kunihiro Ishiguro, et al.


User Access Verification

Password:
```

The password to access this daemon is *zebra*. It supports various commands that are described in the FRRouting documentation We briefly illustrate some of them below.

```
r1> show ipv6 ospf6
OSPFv3 Routing Process (0) with Router-ID 0.0.0.2
Running 14:26:45
LSA minimum arrival 1000 msecs
Initial SPF scheduling delay 0 millisec(s)
Minimum hold time between consecutive SPFs 50 millsecond(s)
Maximum hold time between consecutive SPFs 5000 millsecond(s)
Hold time multiplier is currently 1
SPF algorithm last executed 14:26:26 ago, reason R+, R-
Last SPF duration 0 sec 239 usec
SPF timer is inactive
Number of AS scoped LSAs is 0
Number of areas in this router is 1

Area 0.0.0.0
     Number of Area scoped LSAs is 11
             Interface attached to this area: lo r1-eth0 r1-eth1 r1-eth2
SPF last executed 51986.405587s ago
```

The `show ipv6 ospf6` command reports the general state of the OSPFv3 daemon. The `show ipv6 ospf6 neighbor` command reports the state of the connected neighbors.

```
r1> show ipv6 ospf6 neighbor
Neighbor ID     Pri    DeadTime    State/IfState         Duration I/F[State]
0.0.0.3          10    00:00:02     Full/DR              14:44:44 r1-eth0[BDR]
0.0.0.4          10    00:00:02     Full/DR              14:44:48 r1-eth1[BDR]
```

In its output, we see that `r1` is attached to two different routers. Finally, the `show ipv6 ospf6 database` returns the full OSPFv3 database with all the link state information that was distributed by OSPFv3.

```
r1> show ipv6 ospf6 database

     Area Scoped Link State Database (Area 0.0.0.0)

Type LSId           AdvRouter       Age   SeqNum                        Payload
Rtr  0.0.0.0        0.0.0.2        1037 80000020                0.0.0.3/0.0.0.2
Rtr  0.0.0.0        0.0.0.2        1037 80000020                0.0.0.4/0.0.0.2
Rtr  0.0.0.0        0.0.0.3        1033 80000020                0.0.0.3/0.0.0.2
Rtr  0.0.0.0        0.0.0.3        1033 80000020                0.0.0.4/0.0.0.3
Rtr  0.0.0.0        0.0.0.4        1033 80000020                0.0.0.4/0.0.0.2
Rtr  0.0.0.0        0.0.0.4        1033 80000020                0.0.0.4/0.0.0.3
Net  0.0.0.2        0.0.0.3        1038 8000001e                        0.0.0.3
```

(continues on next page)

(continued from previous page)

```
Net  0.0.0.2        0.0.0.3        1038 8000001e                        0.0.0.2
Net  0.0.0.2        0.0.0.4        1043 8000001e                        0.0.0.4
Net  0.0.0.2        0.0.0.4        1043 8000001e                        0.0.0.2
Net  0.0.0.3        0.0.0.4        1033 8000001e                        0.0.0.4
Net  0.0.0.3        0.0.0.4        1033 8000001e                        0.0.0.3
INP  0.0.0.0        0.0.0.2        1037 80000022           2001:db8:1341:1::/64
INP  0.0.0.2        0.0.0.3        1038 8000001e          2001:db8:1341:12::/64
INP  0.0.0.0        0.0.0.4        1033 80000022           2001:db8:1341:3::/64
INP  0.0.0.2        0.0.0.4        1043 8000001e          2001:db8:1341:13::/64
INP  0.0.0.3        0.0.0.4        1033 8000001e          2001:db8:1341:23::/64

     I/F Scoped Link State Database (I/F lo in Area 0.0.0.0)

Type LSId           AdvRouter       Age   SeqNum                        Payload

     I/F Scoped Link State Database (I/F r1-eth0 in Area 0.0.0.0)

Type LSId           AdvRouter       Age   SeqNum                        Payload
Lnk  0.0.0.3        0.0.0.2        1044 8000001e      fe80::5825:b0ff:fe60:abaa
Lnk  0.0.0.2        0.0.0.3        1045 8000001f      fe80::68bc:b4ff:fe19:42b7

     I/F Scoped Link State Database (I/F r1-eth1 in Area 0.0.0.0)

Type LSId           AdvRouter       Age   SeqNum                        Payload
Lnk  0.0.0.4        0.0.0.2        1044 8000001f      fe80::84eb:5dff:fe5b:dc9d
Lnk  0.0.0.2        0.0.0.4        1046 8000001e      fe80::c088:51ff:fee7:1def

     I/F Scoped Link State Database (I/F r1-eth2 in Area 0.0.0.0)

Type LSId           AdvRouter       Age   SeqNum                        Payload
Lnk  0.0.0.2        0.0.0.2        1044 8000001e      fe80::40d0:61ff:fed9:bccf

     AS Scoped Link State Database

Type LSId           AdvRouter       Age   SeqNum                        Payload
```

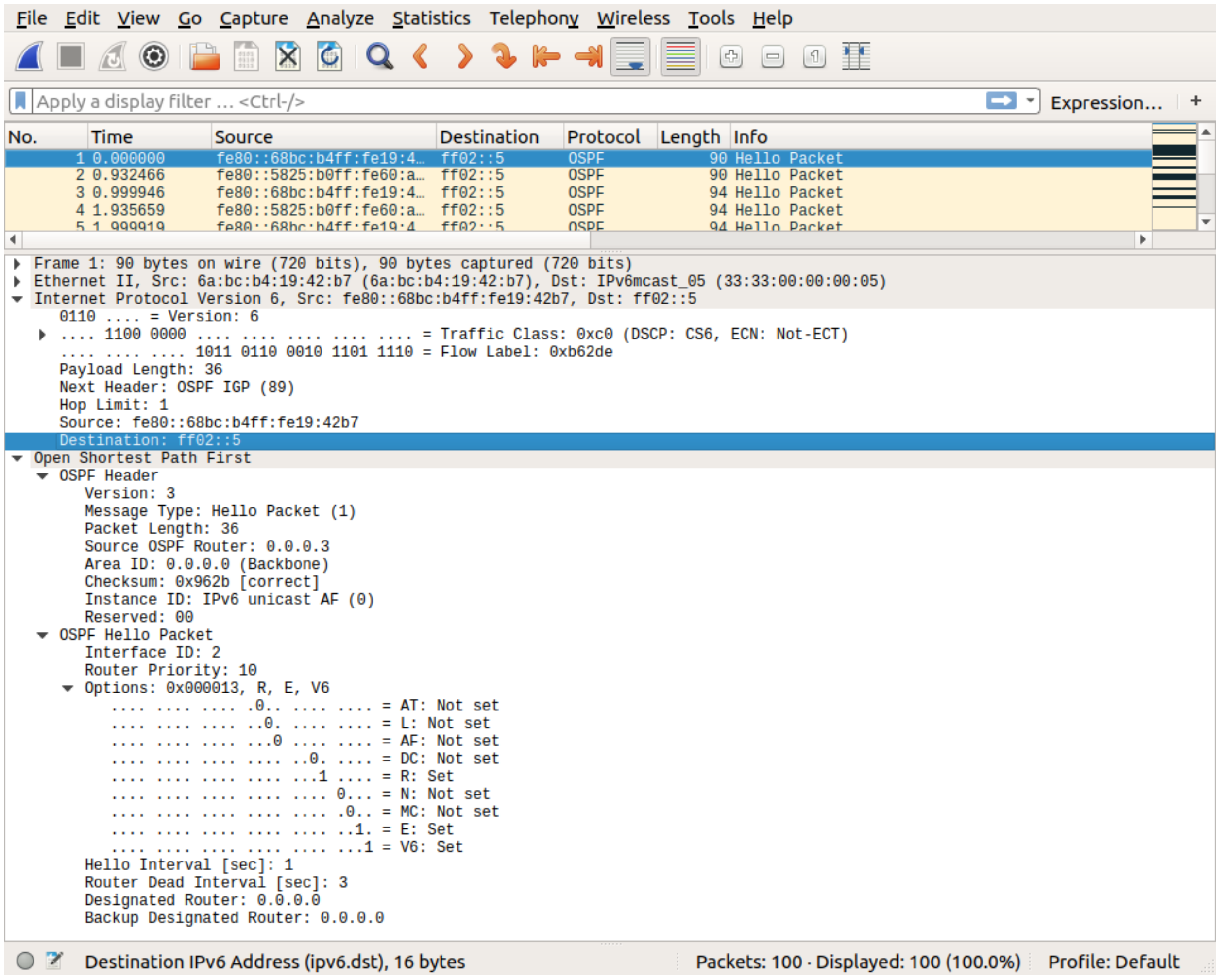
We can also use the packet traces that were collected by tcpdump to observe the packets that the OSPFv3 daemons exchange. OSPFv3 is a more complex protocol that the basic link state protocol that we have described in this book, but you should be able to understand some of these packets. The packet traces are available as `/exercises/traces/ospf6-r1-trace.pcap`, `/exercises/traces/ospf6-r2-trace.pcap` and `/exercises/traces/ospf6-r3-trace.pcap`. Here are a few interesting packets collected on router `r1`.

The first packet that this router received his a Hello packet that was sent by router `r2`. There are several interesting points to note about this packet. First, its source address is the link-local address (`fe80::68bc:b4ff:fe19:42b7`) of router `r2` on this interface. The destination address of the packet is reserved IPv6 multicast address for OSPFv3, i.e. `ff02::5`. The Hop Limit of the packet is set to 1 and OSFPv3 uses a next header of type 89.

The Hello packet contains some parameters such as the *Hello interval* that is set to 1 second. This interval is the delay between the transmission of successive Hello packets. Since the *Router Dead Interval* is set to 3 seconds, the router will consider the link as down if it does not receive Hello packets during a period of 3 seconds. The second packet of the trace is sent by router `r1`.

We can then observe the Database description packet that is sent by routers to announce the state of their OSPFv3 database. The details of this packet are beyond the scope of this simple exercise.

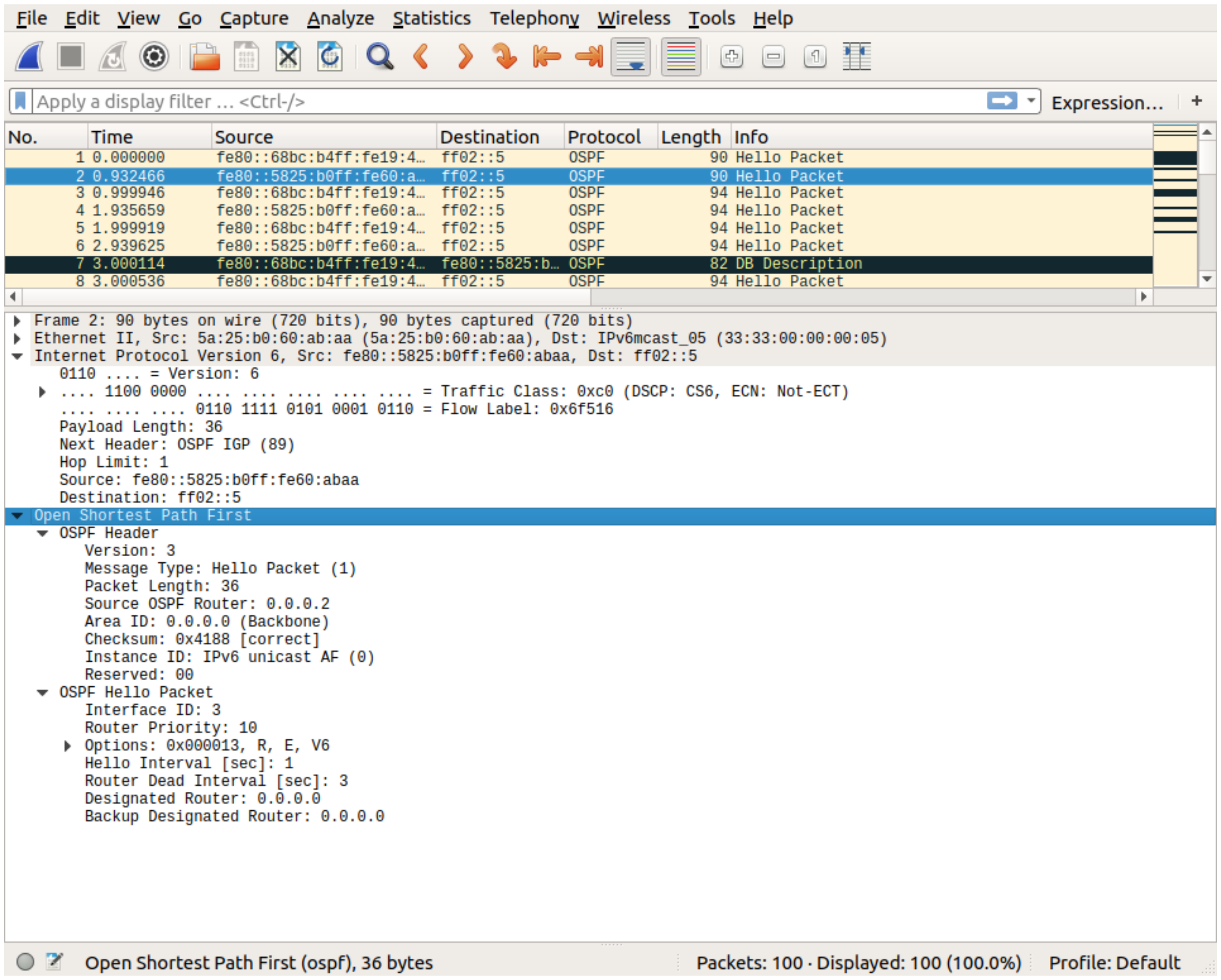
This packet is updated when new information is added in the router's OSPFv3 database. A few seconds router, this router sends another Database description packet that announces more information.


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Expression…
+

```
No.  Time        Source                     Destination  Protocol  Length  Info
   1 0.000000    fe80::68bc:b4ff:fe19:4…    ff02::5      OSPF          90  Hello Packet
   2 0.932466    fe80::5825:b0ff:fe60:a…    ff02::5      OSPF          90  Hello Packet
   3 0.999946    fe80::68bc:b4ff:fe19:4…    ff02::5      OSPF          94  Hello Packet
   4 1.935659    fe80::5825:b0ff:fe60:a…    ff02::5      OSPF          94  Hello Packet
   5 1.999919    fe80::68bc:b4ff:fe19:4     ff02::5      OSPF          94  Hello Packet
```

```
▸ Frame 1: 90 bytes on wire (720 bits), 90 bytes captured (720 bits)
▸ Ethernet II, Src: 6a:bc:b4:19:42:b7 (6a:bc:b4:19:42:b7), Dst: IPv6mcast_05 (33:33:00:00:00:05)
▾ Internet Protocol Version 6, Src: fe80::68bc:b4ff:fe19:42b7, Dst: ff02::5
    0110 .... = Version: 6
  ▸ .... 1100 0000 .... .... .... .... .... = Traffic Class: 0xc0 (DSCP: CS6, ECN: Not-ECT)
    .... .... .... 1011 0110 0010 1101 1110 = Flow Label: 0xb62de
    Payload Length: 36
    Next Header: OSPF IGP (89)
    Hop Limit: 1
    Source: fe80::68bc:b4ff:fe19:42b7
    Destination: ff02::5
▾ Open Shortest Path First
  ▾ OSPF Header
      Version: 3
      Message Type: Hello Packet (1)
      Packet Length: 36
      Source OSPF Router: 0.0.0.3
      Area ID: 0.0.0.0 (Backbone)
      Checksum: 0x962b [correct]
      Instance ID: IPv6 unicast AF (0)
      Reserved: 00
  ▾ OSPF Hello Packet
      Interface ID: 2
      Router Priority: 10
    ▾ Options: 0x000013, R, E, V6
        .... .... .... .0.. .... .... = AT: Not set
        .... .... .... ..0. .... .... = L: Not set
        .... .... .... ...0 .... .... = AF: Not set
        .... .... .... .... ..0. .... = DC: Not set
        .... .... .... .... ...1 .... = R: Set
        .... .... .... .... .... 0... = N: Not set
        .... .... .... .... .... .0.. = MC: Not set
        .... .... .... .... .... ..1. = E: Set
        .... .... .... .... .... ...1 = V6: Set
      Hello Interval [sec]: 1
      Router Dead Interval [sec]: 3
      Designated Router: 0.0.0.0
      Backup Designated Router: 0.0.0.0
```

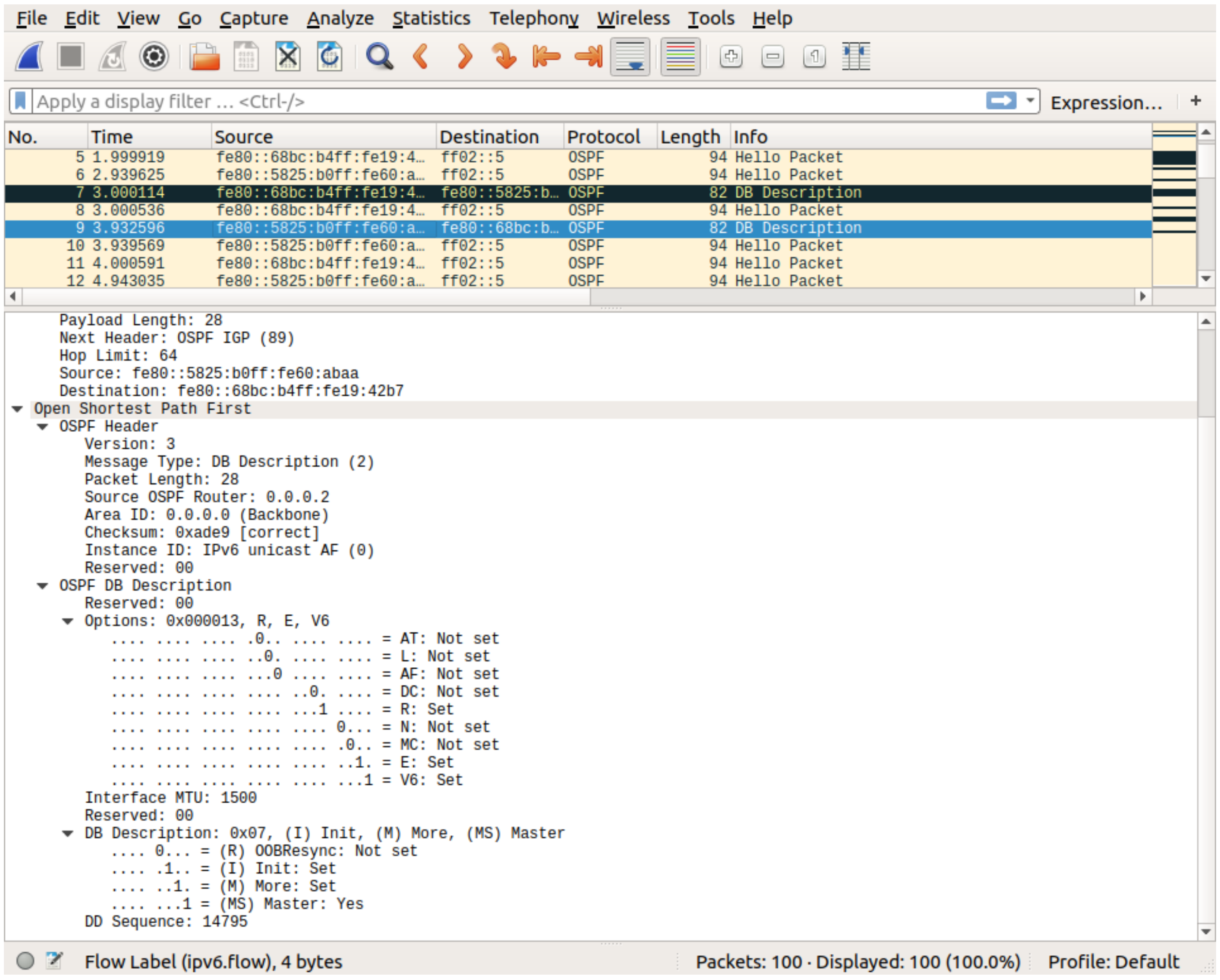
```
Destination IPv6 Address (ipv6.dst), 16 bytes      Packets: 100 · Displayed: 100 (100.0%)      Profile: Default
```


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Expression…
+

```
No.  Time       Source                     Destination       Protocol  Length  Info
  1 0.000000    fe80::68bc:b4ff:fe19:4…   ff02::5           OSPF          90 Hello Packet
  2 0.932466    fe80::5825:b0ff:fe60:a…   ff02::5           OSPF          90 Hello Packet
  3 0.999946    fe80::68bc:b4ff:fe19:4…   ff02::5           OSPF          94 Hello Packet
  4 1.935659    fe80::5825:b0ff:fe60:a…   ff02::5           OSPF          94 Hello Packet
  5 1.999919    fe80::68bc:b4ff:fe19:4…   ff02::5           OSPF          94 Hello Packet
  6 2.939625    fe80::5825:b0ff:fe60:a…   ff02::5           OSPF          94 Hello Packet
  7 3.000114    fe80::68bc:b4ff:fe19:4…   fe80::5825:b…     OSPF          82 DB Description
  8 3.000536    fe80::68bc:b4ff:fe19:4…   ff02::5           OSPF          94 Hello Packet
```

```
▸ Frame 2: 90 bytes on wire (720 bits), 90 bytes captured (720 bits)
▸ Ethernet II, Src: 5a:25:b0:60:ab:aa (5a:25:b0:60:ab:aa), Dst: IPv6mcast_05 (33:33:00:00:00:05)
▾ Internet Protocol Version 6, Src: fe80::5825:b0ff:fe60:abaa, Dst: ff02::5
    0110 .... = Version: 6
  ▸ .... 1100 0000 .... .... .... .... .... = Traffic Class: 0xc0 (DSCP: CS6, ECN: Not-ECT)
    .... .... .... 0110 1111 0101 0001 0110 = Flow Label: 0x6f516
    Payload Length: 36
    Next Header: OSPF IGP (89)
    Hop Limit: 1
    Source: fe80::5825:b0ff:fe60:abaa
    Destination: ff02::5
▾ Open Shortest Path First
  ▾ OSPF Header
      Version: 3
      Message Type: Hello Packet (1)
      Packet Length: 36
      Source OSPF Router: 0.0.0.2
      Area ID: 0.0.0.0 (Backbone)
      Checksum: 0x4188 [correct]
      Instance ID: IPv6 unicast AF (0)
      Reserved: 00
  ▾ OSPF Hello Packet
      Interface ID: 3
      Router Priority: 10
    ▸ Options: 0x000013, R, E, V6
      Hello Interval [sec]: 1
      Router Dead Interval [sec]: 3
      Designated Router: 0.0.0.0
      Backup Designated Router: 0.0.0.0
```

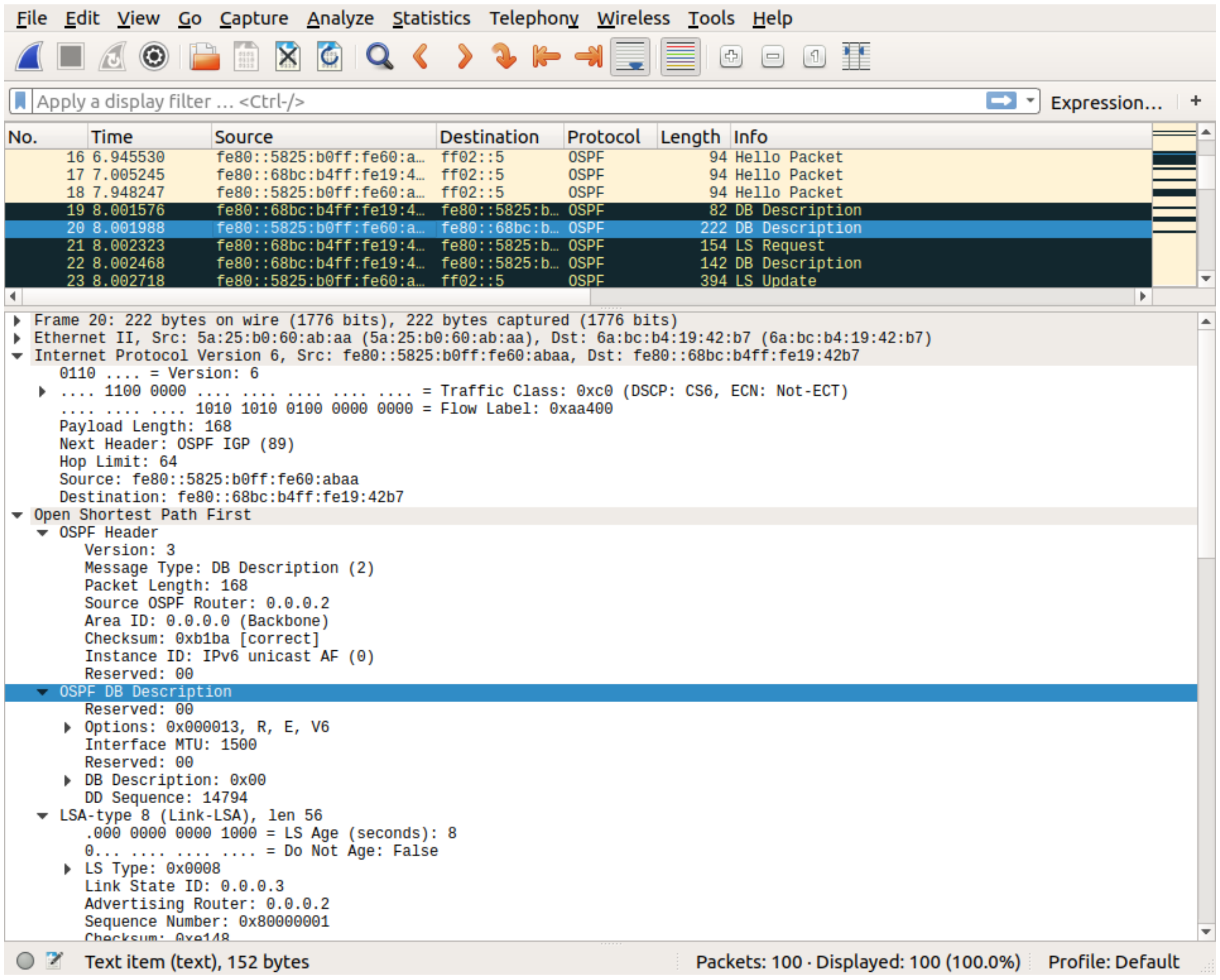
```
Open Shortest Path First (ospf), 36 bytes    Packets: 100 · Displayed: 100 (100.0%)    Profile: Default
```


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Expression…
+

```
No.  Time        Source                       Destination      Protocol Length Info
   5 1.999919    fe80::68bc:b4ff:fe19:4…      ff02::5          OSPF         94 Hello Packet
   6 2.939625    fe80::5825:b0ff:fe60:a…      ff02::5          OSPF         94 Hello Packet
   7 3.000114    fe80::68bc:b4ff:fe19:4…      fe80::5825:b…    OSPF         82 DB Description
   8 3.000536    fe80::68bc:b4ff:fe19:4…      ff02::5          OSPF         94 Hello Packet
   9 3.932596    fe80::5825:b0ff:fe60:a…      fe80::68bc:b…    OSPF         82 DB Description
  10 3.939569    fe80::5825:b0ff:fe60:a…      ff02::5          OSPF         94 Hello Packet
  11 4.000591    fe80::68bc:b4ff:fe19:4…      ff02::5          OSPF         94 Hello Packet
  12 4.943035    fe80::5825:b0ff:fe60:a…      ff02::5          OSPF         94 Hello Packet
```

```
    Payload Length: 28
    Next Header: OSPF IGP (89)
    Hop Limit: 64
    Source: fe80::5825:b0ff:fe60:abaa
    Destination: fe80::68bc:b4ff:fe19:42b7
▼ Open Shortest Path First
  ▼ OSPF Header
      Version: 3
      Message Type: DB Description (2)
      Packet Length: 28
      Source OSPF Router: 0.0.0.2
      Area ID: 0.0.0.0 (Backbone)
      Checksum: 0xade9 [correct]
      Instance ID: IPv6 unicast AF (0)
      Reserved: 00
  ▼ OSPF DB Description
      Reserved: 00
    ▼ Options: 0x000013, R, E, V6
        .... .... .... .0.. .... .... = AT: Not set
        .... .... .... ..0. .... .... = L: Not set
        .... .... .... ...0 .... .... = AF: Not set
        .... .... .... .... ..0. .... = DC: Not set
        .... .... .... .... ...1 .... = R: Set
        .... .... .... .... .... 0... = N: Not set
        .... .... .... .... .... .0.. = MC: Not set
        .... .... .... .... .... ..1. = E: Set
        .... .... .... .... .... ...1 = V6: Set
      Interface MTU: 1500
      Reserved: 00
    ▼ DB Description: 0x07, (I) Init, (M) More, (MS) Master
        .... 0... = (R) OOBResync: Not set
        .... .1.. = (I) Init: Set
        .... ..1. = (M) More: Set
        .... ...1 = (MS) Master: Yes
      DD Sequence: 14795
```

Flow Label (ipv6.flow), 4 bytes   Packets: 100 · Displayed: 100 (100.0%)   Profile: Default

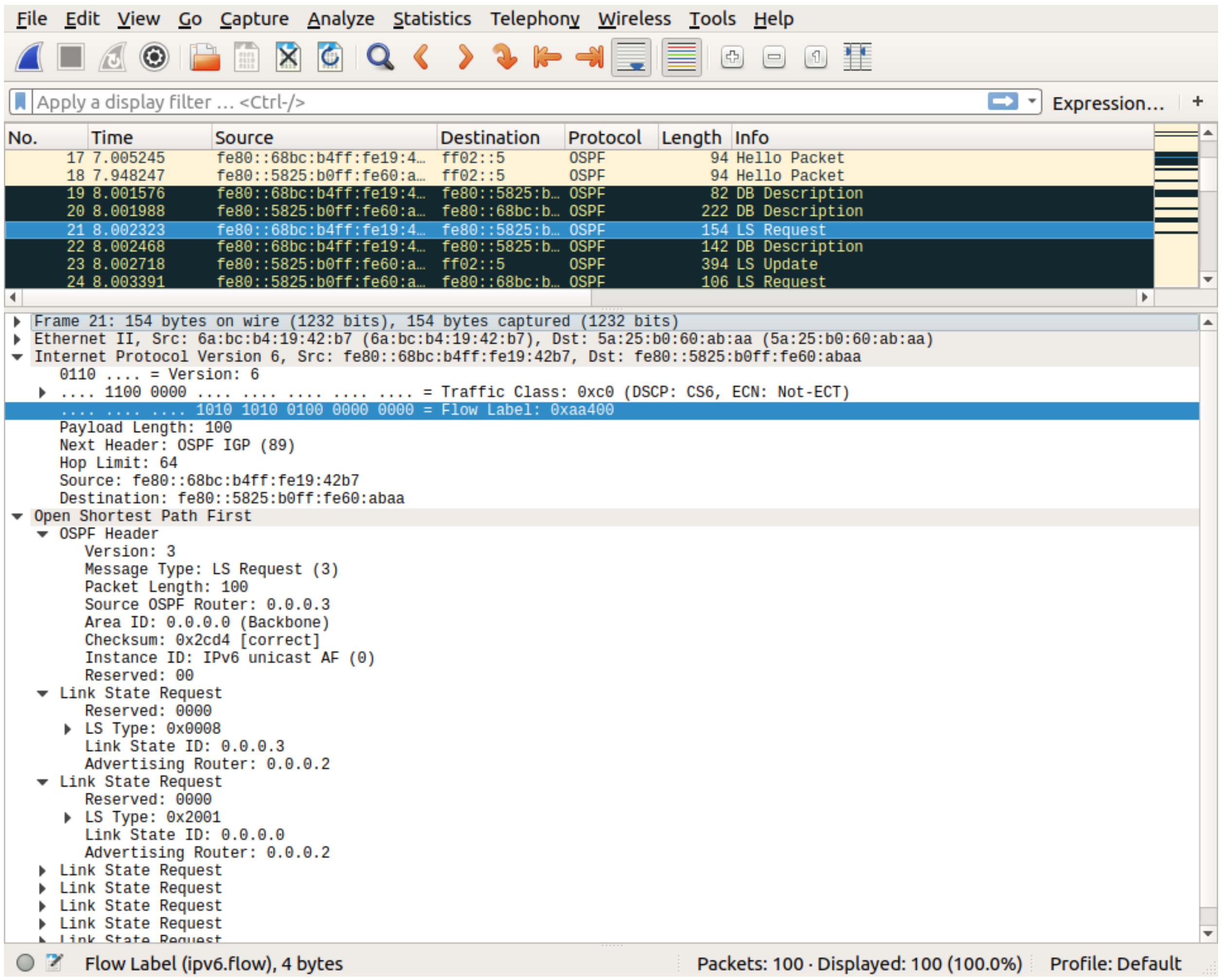

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Expression…
+

```
No.   Time        Source                  Destination        Protocol  Length  Info
   16 6.945530    fe80::5825:b0ff:fe60:a…  ff02::5           OSPF        94 Hello Packet
   17 7.005245    fe80::68bc:b4ff:fe19:4…  ff02::5           OSPF        94 Hello Packet
   18 7.948247    fe80::5825:b0ff:fe60:a…  ff02::5           OSPF        94 Hello Packet
   19 8.001576    fe80::68bc:b4ff:fe19:4…  fe80::5825:b…     OSPF        82 DB Description
   20 8.001988    fe80::5825:b0ff:fe60:a…  fe80::68bc:b…     OSPF       222 DB Description
   21 8.002323    fe80::68bc:b4ff:fe19:4…  fe80::5825:b…     OSPF       154 LS Request
   22 8.002468    fe80::68bc:b4ff:fe19:4…  fe80::5825:b…     OSPF       142 DB Description
   23 8.002718    fe80::5825:b0ff:fe60:a…  ff02::5           OSPF       394 LS Update
```

```
▸ Frame 20: 222 bytes on wire (1776 bits), 222 bytes captured (1776 bits)
▸ Ethernet II, Src: 5a:25:b0:60:ab:aa (5a:25:b0:60:ab:aa), Dst: 6a:bc:b4:19:42:b7 (6a:bc:b4:19:42:b7)
▾ Internet Protocol Version 6, Src: fe80::5825:b0ff:fe60:abaa, Dst: fe80::68bc:b4ff:fe19:42b7
    0110 .... = Version: 6
  ▸ .... 1100 0000 .... .... .... .... .... = Traffic Class: 0xc0 (DSCP: CS6, ECN: Not-ECT)
    .... .... .... 1010 1010 0100 0000 0000 = Flow Label: 0xaa400
    Payload Length: 168
    Next Header: OSPF IGP (89)
    Hop Limit: 64
    Source: fe80::5825:b0ff:fe60:abaa
    Destination: fe80::68bc:b4ff:fe19:42b7
▾ Open Shortest Path First
  ▾ OSPF Header
      Version: 3
      Message Type: DB Description (2)
      Packet Length: 168
      Source OSPF Router: 0.0.0.2
      Area ID: 0.0.0.0 (Backbone)
      Checksum: 0xb1ba [correct]
      Instance ID: IPv6 unicast AF (0)
      Reserved: 00
  ▾ OSPF DB Description
      Reserved: 00
    ▸ Options: 0x000013, R, E, V6
      Interface MTU: 1500
      Reserved: 00
    ▸ DB Description: 0x00
      DD Sequence: 14794
  ▾ LSA-type 8 (Link-LSA), len 56
      .000 0000 0000 1000 = LS Age (seconds): 8
      0... .... .... .... = Do Not Age: False
    ▸ LS Type: 0x0008
      Link State ID: 0.0.0.3
      Advertising Router: 0.0.0.2
      Sequence Number: 0x80000001
      Checksum: 0xe148
```

Text item (text), 152 bytes   Packets: 100 · Displayed: 100 (100.0%)   Profile: Default

Router `r2` reacts to this updated Database description packet by requesting the link state information that it does not already know. For this, it sends a *LS Request* packet.


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
+

```
No.  Time      Source                Destination      Protocol Length Info
  17 7.005245  fe80::68bc:b4ff:fe19:4… ff02::5        OSPF       94 Hello Packet
  18 7.948247  fe80::5825:b0ff:fe60:a… ff02::5        OSPF       94 Hello Packet
  19 8.001576  fe80::68bc:b4ff:fe19:4… fe80::5825:b… OSPF       82 DB Description
  20 8.001988  fe80::5825:b0ff:fe60:a… fe80::68bc:b… OSPF      222 DB Description
  21 8.002323  fe80::68bc:b4ff:fe19:4… fe80::5825:b… OSPF      154 LS Request
  22 8.002468  fe80::68bc:b4ff:fe19:4… fe80::5825:b… OSPF      142 DB Description
  23 8.002718  fe80::5825:b0ff:fe60:a… ff02::5        OSPF      394 LS Update
  24 8.003391  fe80::5825:b0ff:fe60:a… fe80::68bc:b… OSPF      106 LS Request
```

```
▶ Frame 21: 154 bytes on wire (1232 bits), 154 bytes captured (1232 bits)
▶ Ethernet II, Src: 6a:bc:b4:19:42:b7 (6a:bc:b4:19:42:b7), Dst: 5a:25:b0:60:ab:aa (5a:25:b0:60:ab:aa)
▼ Internet Protocol Version 6, Src: fe80::68bc:b4ff:fe19:42b7, Dst: fe80::5825:b0ff:fe60:abaa
    0110 .... = Version: 6
  ▶ .... 1100 0000 .... .... .... .... .... = Traffic Class: 0xc0 (DSCP: CS6, ECN: Not-ECT)
    .... .... .... 1010 1010 0100 0000 0000 = Flow Label: 0xaa400
    Payload Length: 100
    Next Header: OSPF IGP (89)
    Hop Limit: 64
    Source: fe80::68bc:b4ff:fe19:42b7
    Destination: fe80::5825:b0ff:fe60:abaa
▼ Open Shortest Path First
  ▼ OSPF Header
      Version: 3
      Message Type: LS Request (3)
      Packet Length: 100
      Source OSPF Router: 0.0.0.3
      Area ID: 0.0.0.0 (Backbone)
      Checksum: 0x2cd4 [correct]
      Instance ID: IPv6 unicast AF (0)
      Reserved: 00
  ▼ Link State Request
      Reserved: 0000
    ▶ LS Type: 0x0008
      Link State ID: 0.0.0.3
      Advertising Router: 0.0.0.2
  ▼ Link State Request
      Reserved: 0000
    ▶ LS Type: 0x2001
      Link State ID: 0.0.0.0
      Advertising Router: 0.0.0.2
  ▶ Link State Request
  ▶ Link State Request
  ▶ Link State Request
  ▶ Link State Request
  ▶ Link State Request
```

```
Flow Label (ipv6.flow), 4 bytes          Packets: 100 · Displayed: 100 (100.0%)   Profile: Default
```

The requested information is sent in a *LS Update* packet shortly after that.

OSPFv3 also includes *LS Acknowledge* packets that acknowledge the correct reception of link state information.

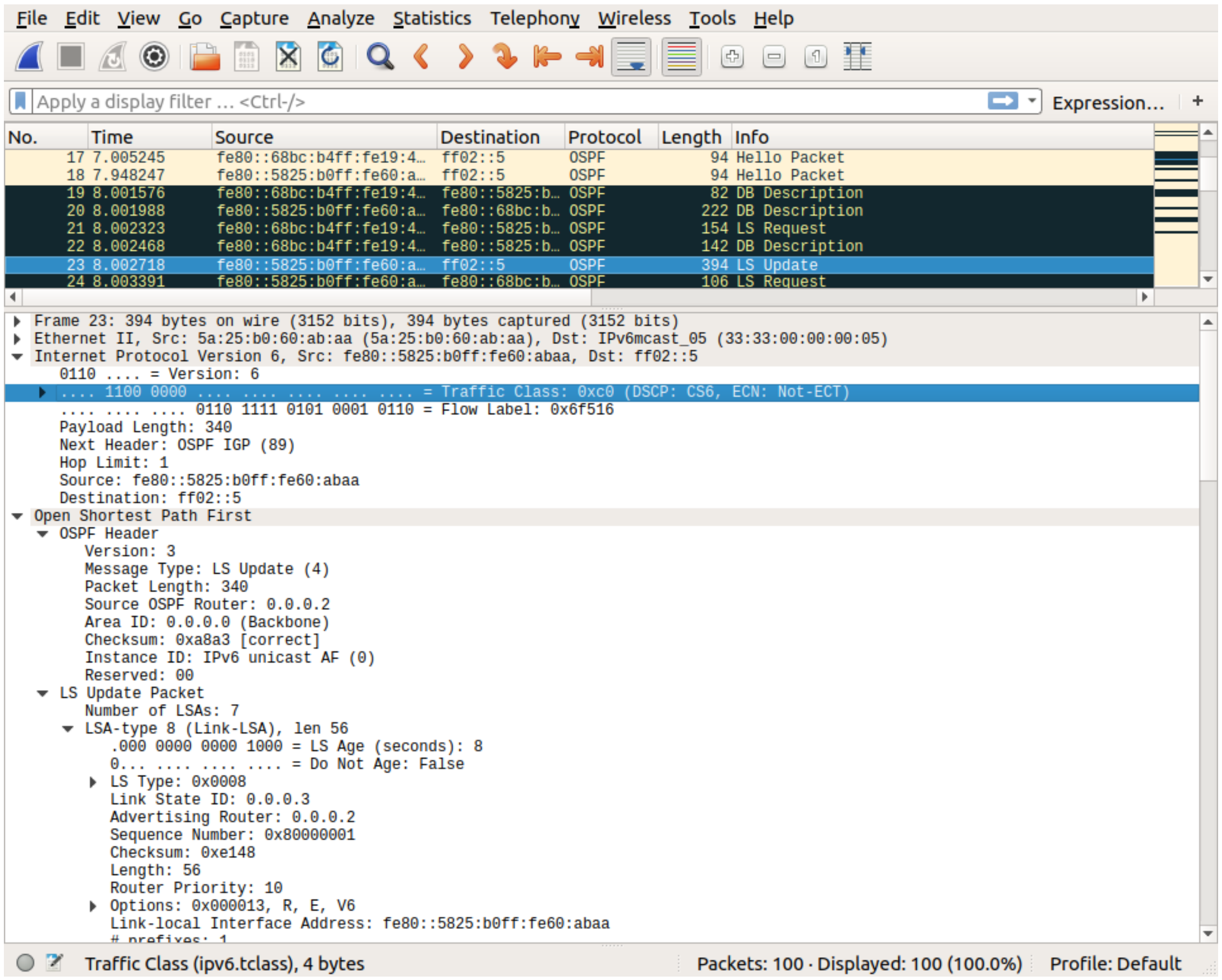
A more detailed discussion of the packets that routing protocols exchange may be found in [Goralski2009].

4.16.2 Exploring RIP

IPMininet can also be used to perform experiments with RIP. A simple script that uses RIPng is provided below.

```
import shlex
from ipmininet.iptopo import IPTopo
from ipmininet.router.config import RIPng, RouterConfig
from ipmininet.ipnet import IPNet
from ipmininet.cli import IPCLI

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):
```

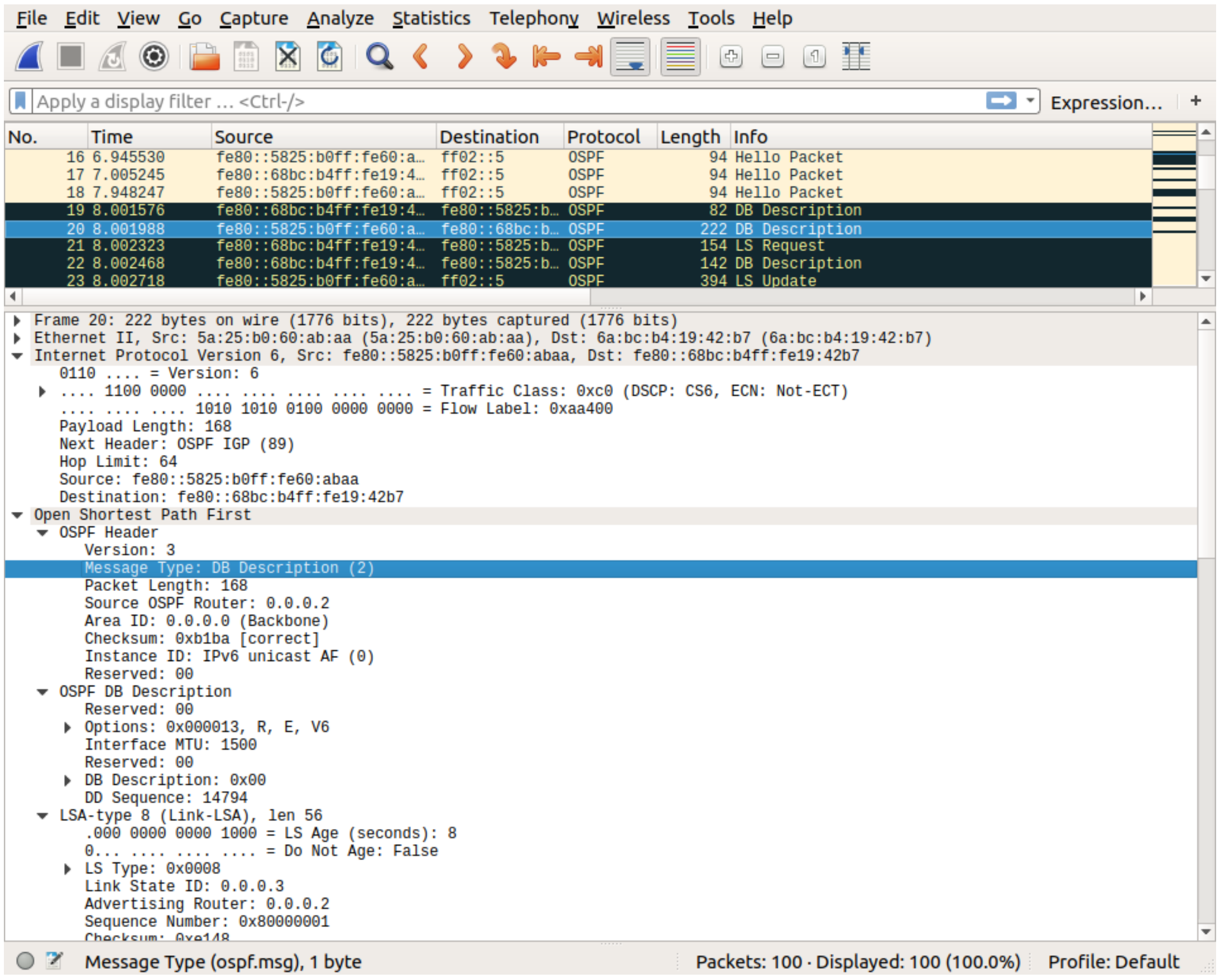
(continues on next page)


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Expression…
+

```
No.  Time        Source                   Destination       Protocol  Length Info
  17 7.005245    fe80::68bc:b4ff:fe19:4…  ff02::5           OSPF          94 Hello Packet
  18 7.948247    fe80::5825:b0ff:fe60:a…  ff02::5           OSPF          94 Hello Packet
  19 8.001576    fe80::68bc:b4ff:fe19:4…  fe80::5825:b…     OSPF          82 DB Description
  20 8.001988    fe80::5825:b0ff:fe60:a…  fe80::68bc:b…     OSPF         222 DB Description
  21 8.002323    fe80::68bc:b4ff:fe19:4…  fe80::5825:b…     OSPF         154 LS Request
  22 8.002468    fe80::68bc:b4ff:fe19:4…  fe80::5825:b…     OSPF         142 DB Description
  23 8.002718    fe80::5825:b0ff:fe60:a…  ff02::5           OSPF         394 LS Update
  24 8.003391    fe80::5825:b0ff:fe60:a…  fe80::68bc:b…     OSPF         106 LS Request
```

```
▸ Frame 23: 394 bytes on wire (3152 bits), 394 bytes captured (3152 bits)
▸ Ethernet II, Src: 5a:25:b0:60:ab:aa (5a:25:b0:60:ab:aa), Dst: IPv6mcast_05 (33:33:00:00:00:05)
▾ Internet Protocol Version 6, Src: fe80::5825:b0ff:fe60:abaa, Dst: ff02::5
    0110 .... = Version: 6
  ▸ .... 1100 0000 .... .... .... .... .... = Traffic Class: 0xc0 (DSCP: CS6, ECN: Not-ECT)
    .... .... .... 0110 1111 0101 0001 0110 = Flow Label: 0x6f516
    Payload Length: 340
    Next Header: OSPF IGP (89)
    Hop Limit: 1
    Source: fe80::5825:b0ff:fe60:abaa
    Destination: ff02::5
▾ Open Shortest Path First
  ▾ OSPF Header
      Version: 3
      Message Type: LS Update (4)
      Packet Length: 340
      Source OSPF Router: 0.0.0.2
      Area ID: 0.0.0.0 (Backbone)
      Checksum: 0xa8a3 [correct]
      Instance ID: IPv6 unicast AF (0)
      Reserved: 00
  ▾ LS Update Packet
      Number of LSAs: 7
    ▾ LSA-type 8 (Link-LSA), len 56
        .000 0000 0000 1000 = LS Age (seconds): 8
        0... .... .... .... = Do Not Age: False
      ▸ LS Type: 0x0008
        Link State ID: 0.0.0.3
        Advertising Router: 0.0.0.2
        Sequence Number: 0x80000001
        Checksum: 0xe148
        Length: 56
        Router Priority: 10
      ▸ Options: 0x000013, R, E, V6
        Link-local Interface Address: fe80::5825:b0ff:fe60:abaa
        # prefixes: 1
```

Traffic Class (ipv6.tclass), 4 bytes   Packets: 100 · Displayed: 100 (100.0%)   Profile: Default


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Expression…
+

```
No.   Time      Source                   Destination         Protocol  Length Info
   16 6.945530  fe80::5825:b0ff:fe60:a…  ff02::5             OSPF          94 Hello Packet
   17 7.005245  fe80::68bc:b4ff:fe19:4…  ff02::5             OSPF          94 Hello Packet
   18 7.948247  fe80::5825:b0ff:fe60:a…  ff02::5             OSPF          94 Hello Packet
   19 8.001576  fe80::68bc:b4ff:fe19:4…  fe80::5825:b…       OSPF          82 DB Description
   20 8.001988  fe80::5825:b0ff:fe60:a…  fe80::68bc:b…       OSPF         222 DB Description
   21 8.002323  fe80::68bc:b4ff:fe19:4…  fe80::5825:b…       OSPF         154 LS Request
   22 8.002468  fe80::68bc:b4ff:fe19:4…  fe80::5825:b…       OSPF         142 DB Description
   23 8.002718  fe80::5825:b0ff:fe60:a…  ff02::5             OSPF         394 LS Update
```

```
▸ Frame 20: 222 bytes on wire (1776 bits), 222 bytes captured (1776 bits)
▸ Ethernet II, Src: 5a:25:b0:60:ab:aa (5a:25:b0:60:ab:aa), Dst: 6a:bc:b4:19:42:b7 (6a:bc:b4:19:42:b7)
▾ Internet Protocol Version 6, Src: fe80::5825:b0ff:fe60:abaa, Dst: fe80::68bc:b4ff:fe19:42b7
    0110 .... = Version: 6
  ▸ .... 1100 0000 .... .... .... .... .... = Traffic Class: 0xc0 (DSCP: CS6, ECN: Not-ECT)
    .... .... .... 1010 1010 0100 0000 0000 = Flow Label: 0xaa400
    Payload Length: 168
    Next Header: OSPF IGP (89)
    Hop Limit: 64
    Source: fe80::5825:b0ff:fe60:abaa
    Destination: fe80::68bc:b4ff:fe19:42b7
▾ Open Shortest Path First
  ▾ OSPF Header
      Version: 3
      Message Type: DB Description (2)
      Packet Length: 168
      Source OSPF Router: 0.0.0.2
      Area ID: 0.0.0.0 (Backbone)
      Checksum: 0xb1ba [correct]
      Instance ID: IPv6 unicast AF (0)
      Reserved: 00
  ▾ OSPF DB Description
      Reserved: 00
    ▸ Options: 0x000013, R, E, V6
      Interface MTU: 1500
      Reserved: 00
    ▸ DB Description: 0x00
      DD Sequence: 14794
  ▾ LSA-type 8 (Link-LSA), len 56
      .000 0000 0000 1000 = LS Age (seconds): 8
      0... .... .... .... = Do Not Age: False
    ▸ LS Type: 0x0008
      Link State ID: 0.0.0.3
      Advertising Router: 0.0.0.2
      Sequence Number: 0x80000001
      Checksum: 0xe148
```

```
Message Type (ospf.msg), 1 byte        Packets: 100 · Displayed: 100 (100.0%)        Profile: Default
```

(continued from previous page)

```
        # RouterConfig ensures that OSPF is not automatically started
        r1 = self.addRouter("r1", config=RouterConfig)
        r2 = self.addRouter("r2", config=RouterConfig)
        r3 = self.addRouter("r3", config=RouterConfig)
        a = self.addHost("a")
        b = self.addHost("b")

        lr1r2 = self.addLink(r1, r2, igp_cost=1)
        lr1r2[r1].addParams(ip=("2001:db8:1341:12::1/64"))
        lr1r2[r2].addParams(ip=("2001:db8:1341:12::2/64"))

        lr1r3 = self.addLink(r1, r3, igp_cost=5)
        lr1r3[r1].addParams(ip=("2001:db8:1341:13::1/64"))
        lr1r3[r3].addParams(ip=("2001:db8:1341:13::3/64"))

        lr2r3 = self.addLink(r2, r3, igp_cost=3)
        lr2r3[r2].addParams(ip=("2001:db8:1341:23::2/64"))
        lr2r3[r3].addParams(ip=("2001:db8:1341:23::3/64"))

        lr1a = self.addLink(r1, a)
        lr1a[r1].addParams(ip=("2001:db8:1341:1::1/64"))
        lr1a[a].addParams(ip=("2001:db8:1341:1::A/64"))

        lr3b = self.addLink(r3, b)
        lr3b[r3].addParams(ip=("2001:db8:1341:3::3/64"))
        lr3b[b].addParams(ip=("2001:db8:1341:3::B/64"))


        r1.addDaemon(RIPng)
        r2.addDaemon(RIPng)
        r3.addDaemon(RIPng)

        super(MyTopology, self).build(*args, **kwargs)

    def post_build(self, net):
        for r in self.routers():
            command = "/usr/sbin/tcpdump --immediate-mode -c 10 -w ./ripng-"+r+"-
↪trace.pcap udp port 521"
            p = net[r].popen(shlex.split(command))

    super(MyTopology, self).post_build(net)

net = IPNet(topo=MyTopology(), allocate_IPs=False)  # Disable IP auto-allocation
try:
    net.start()
    IPCLI(net)
finally:
    net.stop()
```

As RIP messages are exchanged using UDP on port 521, we filter this port in the tcpdump trace. RIPng distributes the routes and our two hosts can exchange packets. The entire script is available from `/exercises/ipmininet_scripts/ripng.py`.

```
mininet> ping6all
*** Ping: testing reachability over IPv6
```

(continues on next page)

(continued from previous page)

```
a --IPv6--> b
b --IPv6--> a
*** Results: 0% dropped (2/2 received)
mininet> a traceroute6 -n 2001:db8:1341:3::b
traceroute to 2001:db8:1341:3::b (2001:db8:1341:3::b) from 2001:db8:1341:1::a, 30␣
↪hops max, 24 byte packets
1  2001:db8:1341:1::1  0.078 ms  0.074 ms  0.051 ms
2  2001:db8:1341:13::3  0.071 ms  0.072 ms  0.212 ms
3  2001:db8:1341:3::b  0.199 ms  0.08 ms  0.835 ms
mininet> b traceroute6 -n 2001:db8:1341:1::a
traceroute to 2001:db8:1341:1::a (2001:db8:1341:1::a) from 2001:db8:1341:3::b, 30␣
↪hops max, 24 byte packets
1  2001:db8:1341:3::3  0.06 ms  0.022 ms  0.018 ms
2  2001:db8:1341:13::1  0.038 ms  0.024 ms  0.022 ms
3  2001:db8:1341:1::a  0.03 ms  0.023 ms  0.022 ms
mininet>
```

We can observe the RIPng messages that are exchanged over the network. **RFC 2080** defines two types of RIPng messages:

- the requests
- the responses that contain the router's routing table

When a router starts, it sends a request message. This is illustrated in the figure below with the first message sent by router `r2`. This message is sent inside an IPv6 packet whose source address is the link-local address of the router and the destination address is `ff02::9` which is the reserved multicast address for RIPng.

Router `r2` receives a similar request from `fe80::481a:48ff:fed7:292e` and replies by sending its routing table in a response message. Note that this message is sent to the link-local address of the requesting router.

Later, router `r2` will regularly transmit its distance vector inside an unsolicited response message that is sent towards the IPv7 multicast address `ff02::9`.

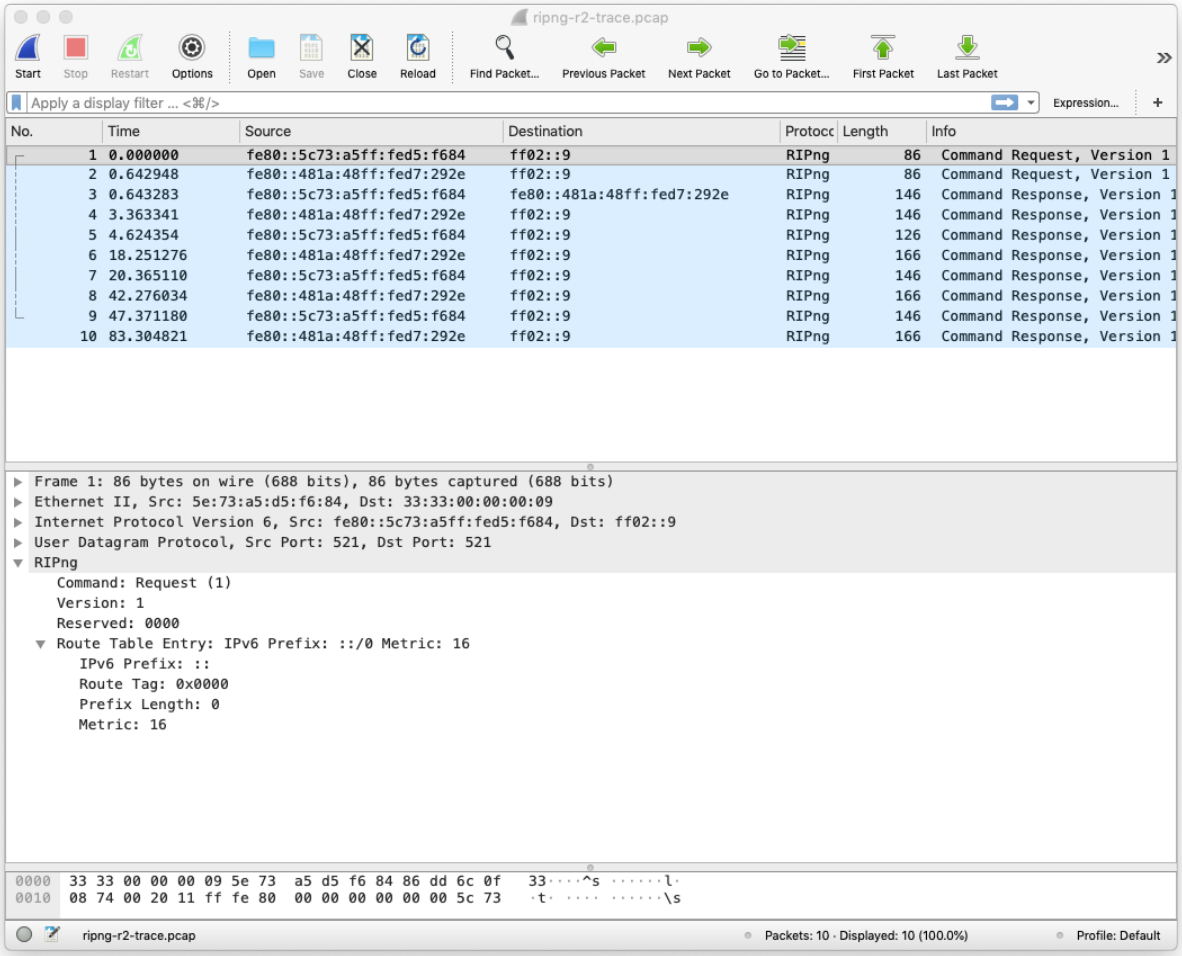
The packet traces collected on the three routers of this example are available from `/exercises/traces/ripng-r1-trace.pcap`, `/exercises/traces/ripng-r2-trace.pcap` and `/exercises/traces/ripng-r3-trace.pcap`.

4.16.3 Exploring BGP

To explore the configuration of BGP, let us consider a network that contains three ASes: `AS1`, `AS2` and `AS3`. To simplify the tests, we identify one host inside each of these ASes.

```
import ipmininet.router.config.bgp as _bgp
from ipmininet.iptopo import IPTopo
from ipmininet.router.config import BGP, ebgp_session, AF_INET6, CLIENT_PROVIDER,␣
↪SHARE

from ipmininet.ipnet import IPNet
from ipmininet.cli import IPCLI

class MyTopology(IPTopo):
    """Creates a very simple interdomain topology"""
    def build(self, *args, **kwargs):
        """
```

(continues on next page)

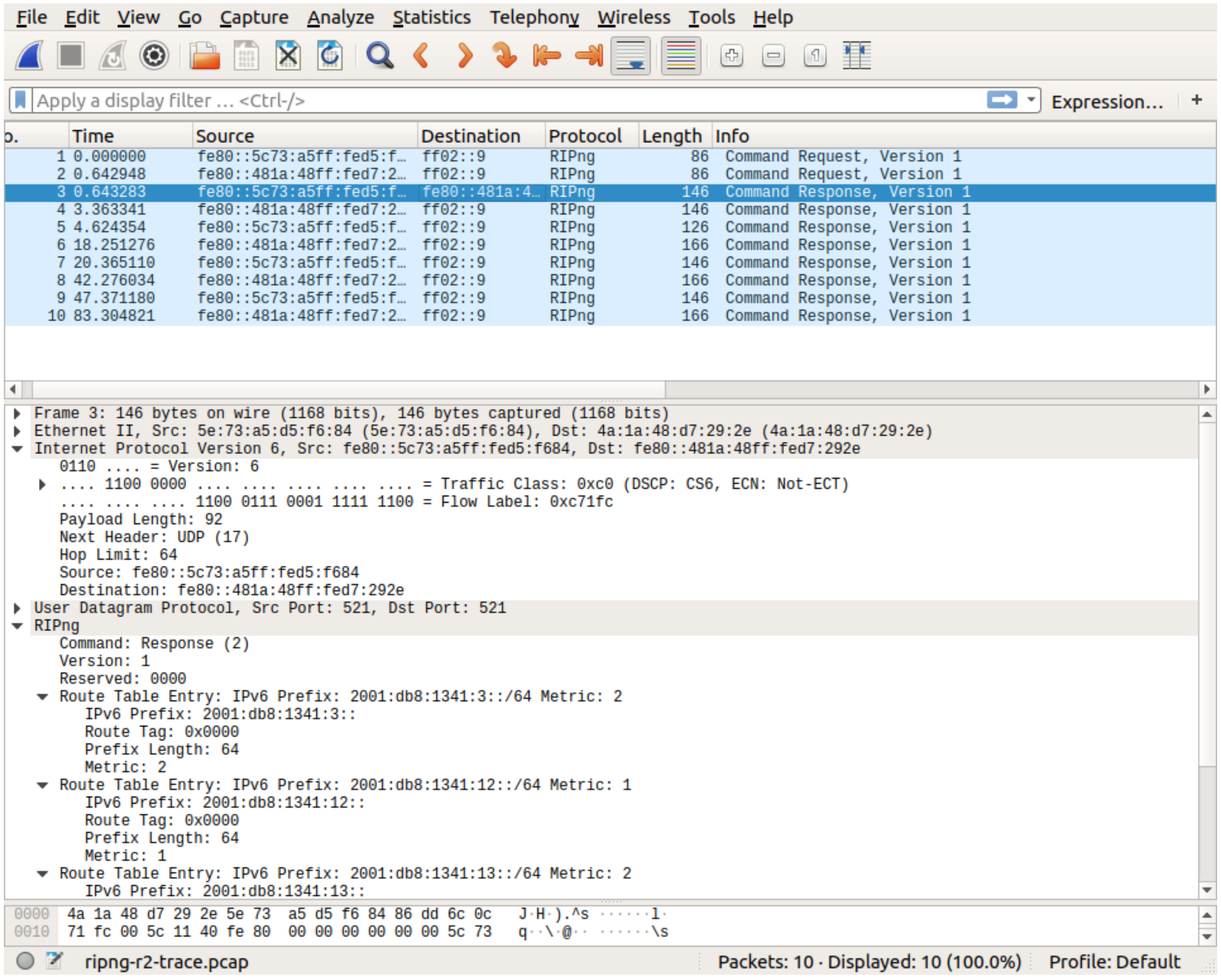


```
No.   Time        Source                      Destination                 Protocc Length  Info
    1 0.000000    fe80::5c73:a5ff:fed5:f684   ff02::9                     RIPng       86  Command Request, Version 1
    2 0.642948    fe80::481a:48ff:fed7:292e   ff02::9                     RIPng       86  Command Request, Version 1
    3 0.643283    fe80::5c73:a5ff:fed5:f684   fe80::481a:48ff:fed7:292e   RIPng      146  Command Response, Version 1
    4 3.363341    fe80::481a:48ff:fed7:292e   ff02::9                     RIPng      146  Command Response, Version 1
    5 4.624354    fe80::5c73:a5ff:fed5:f684   ff02::9                     RIPng      126  Command Response, Version 1
    6 18.251276   fe80::481a:48ff:fed7:292e   ff02::9                     RIPng      166  Command Response, Version 1
    7 20.365110   fe80::5c73:a5ff:fed5:f684   ff02::9                     RIPng      146  Command Response, Version 1
    8 42.276034   fe80::481a:48ff:fed7:292e   ff02::9                     RIPng      166  Command Response, Version 1
    9 47.371180   fe80::5c73:a5ff:fed5:f684   ff02::9                     RIPng      146  Command Response, Version 1
   10 83.304821   fe80::481a:48ff:fed7:292e   ff02::9                     RIPng      166  Command Response, Version 1
```

```
▶ Frame 1: 86 bytes on wire (688 bits), 86 bytes captured (688 bits)
▶ Ethernet II, Src: 5e:73:a5:d5:f6:84, Dst: 33:33:00:00:00:09
▶ Internet Protocol Version 6, Src: fe80::5c73:a5ff:fed5:f684, Dst: ff02::9
▶ User Datagram Protocol, Src Port: 521, Dst Port: 521
▼ RIPng
    Command: Request (1)
    Version: 1
    Reserved: 0000
  ▼ Route Table Entry: IPv6 Prefix: ::/0 Metric: 16
      IPv6 Prefix: ::
      Route Tag: 0x0000
      Prefix Length: 0
      Metric: 16
```

```
0000  33 33 00 00 00 09 5e 73  a5 d5 f6 84 86 dd 6c 0f   33····^s ······l·
0010  08 74 00 20 11 ff fe 80  00 00 00 00 00 00 5c 73   ·t· ···· ······\s
```

ripng-r2-trace.pcap   Packets: 10 · Displayed: 10 (100.0%)   Profile: Default

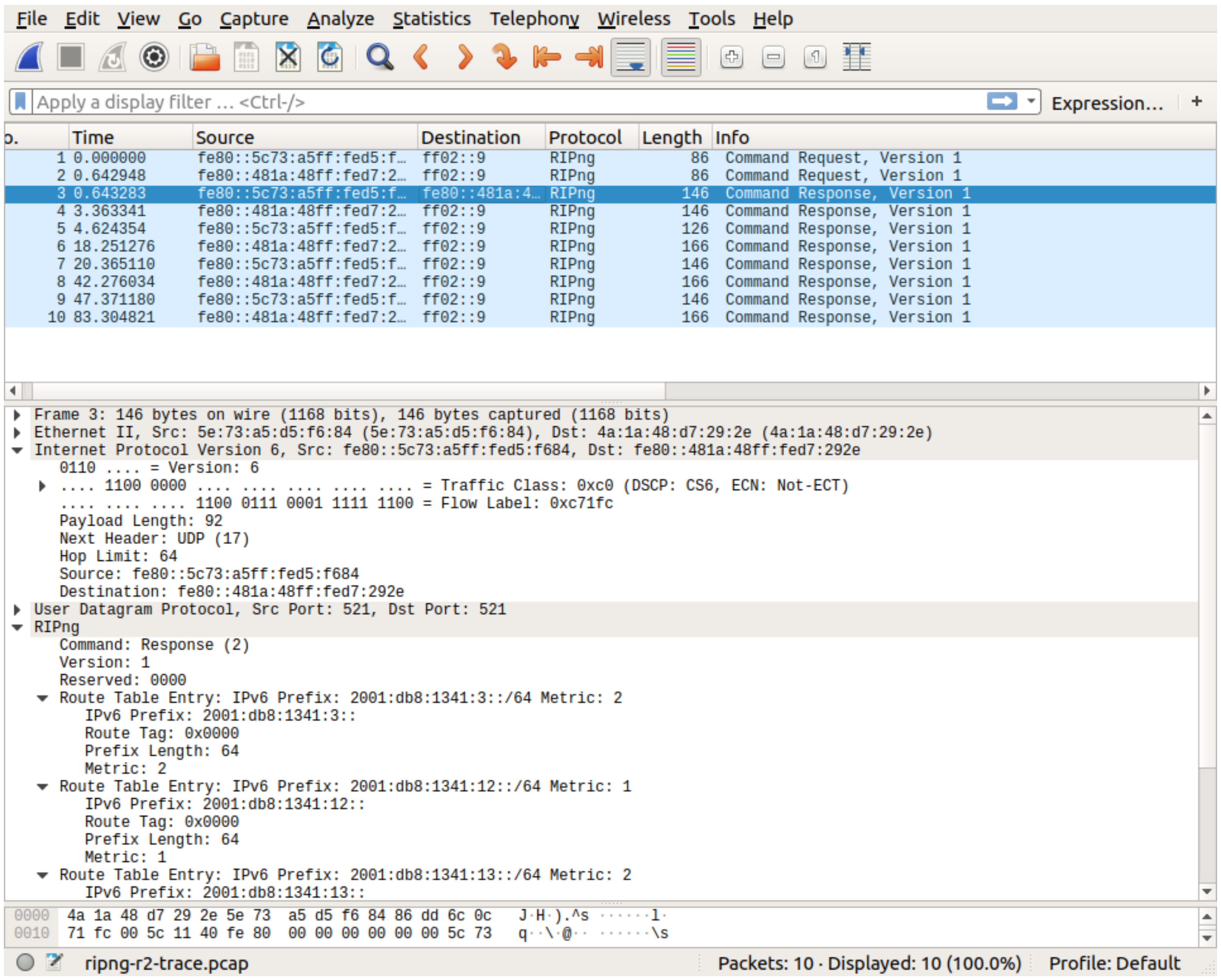

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Expression…
+

```
o.   Time        Source                   Destination      Protocol  Length  Info
   1 0.000000    fe80::5c73:a5ff:fed5:f…  ff02::9          RIPng         86  Command Request, Version 1
   2 0.642948    fe80::481a:48ff:fed7:2…  ff02::9          RIPng         86  Command Request, Version 1
   3 0.643283    fe80::5c73:a5ff:fed5:f…  fe80::481a:4…    RIPng        146  Command Response, Version 1
   4 3.363341    fe80::481a:48ff:fed7:2…  ff02::9          RIPng        146  Command Response, Version 1
   5 4.624354    fe80::5c73:a5ff:fed5:f…  ff02::9          RIPng        126  Command Response, Version 1
   6 18.251276   fe80::481a:48ff:fed7:2…  ff02::9          RIPng        166  Command Response, Version 1
   7 20.365110   fe80::5c73:a5ff:fed5:f…  ff02::9          RIPng        146  Command Response, Version 1
   8 42.276034   fe80::481a:48ff:fed7:2…  ff02::9          RIPng        166  Command Response, Version 1
   9 47.371180   fe80::5c73:a5ff:fed5:f…  ff02::9          RIPng        146  Command Response, Version 1
  10 83.304821   fe80::481a:48ff:fed7:2…  ff02::9          RIPng        166  Command Response, Version 1
```

```
▸ Frame 3: 146 bytes on wire (1168 bits), 146 bytes captured (1168 bits)
▸ Ethernet II, Src: 5e:73:a5:d5:f6:84 (5e:73:a5:d5:f6:84), Dst: 4a:1a:48:d7:29:2e (4a:1a:48:d7:29:2e)
▾ Internet Protocol Version 6, Src: fe80::5c73:a5ff:fed5:f684, Dst: fe80::481a:48ff:fed7:292e
    0110 .... = Version: 6
  ▸ .... 1100 0000 .... .... .... .... .... = Traffic Class: 0xc0 (DSCP: CS6, ECN: Not-ECT)
    .... .... .... 1100 0111 0001 1111 1100 = Flow Label: 0xc71fc
    Payload Length: 92
    Next Header: UDP (17)
    Hop Limit: 64
    Source: fe80::5c73:a5ff:fed5:f684
    Destination: fe80::481a:48ff:fed7:292e
▸ User Datagram Protocol, Src Port: 521, Dst Port: 521
▾ RIPng
    Command: Response (2)
    Version: 1
    Reserved: 0000
  ▾ Route Table Entry: IPv6 Prefix: 2001:db8:1341:3::/64 Metric: 2
      IPv6 Prefix: 2001:db8:1341:3::
      Route Tag: 0x0000
      Prefix Length: 64
      Metric: 2
  ▾ Route Table Entry: IPv6 Prefix: 2001:db8:1341:12::/64 Metric: 1
      IPv6 Prefix: 2001:db8:1341:12::
      Route Tag: 0x0000
      Prefix Length: 64
      Metric: 1
  ▾ Route Table Entry: IPv6 Prefix: 2001:db8:1341:13::/64 Metric: 2
      IPv6 Prefix: 2001:db8:1341:13::
```

```
0000  4a 1a 48 d7 29 2e 5e 73  a5 d5 f6 84 86 dd 6c 0c   J·H·).^s ······l·
0010  71 fc 00 5c 11 40 fe 80  00 00 00 00 00 00 5c 73   q··\·@·· ······\s
```

ripng-r2-trace.pcap   Packets: 10 · Displayed: 10 (100.0%)   Profile: Default


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Expression…
+

```
o.      Time        Source                     Destination    Protocol  Length  Info
     1 0.000000    fe80::5c73:a5ff:fed5:f…    ff02::9        RIPng         86  Command Request, Version 1
     2 0.642948    fe80::481a:48ff:fed7:2…    ff02::9        RIPng         86  Command Request, Version 1
     3 0.643283    fe80::5c73:a5ff:fed5:f…    fe80::481a:4…  RIPng        146  Command Response, Version 1
     4 3.363341    fe80::481a:48ff:fed7:2…    ff02::9        RIPng        146  Command Response, Version 1
     5 4.624354    fe80::5c73:a5ff:fed5:f…    ff02::9        RIPng        126  Command Response, Version 1
     6 18.251276   fe80::481a:48ff:fed7:2…    ff02::9        RIPng        166  Command Response, Version 1
     7 20.365110   fe80::5c73:a5ff:fed5:f…    ff02::9        RIPng        146  Command Response, Version 1
     8 42.276034   fe80::481a:48ff:fed7:2…    ff02::9        RIPng        166  Command Response, Version 1
     9 47.371180   fe80::5c73:a5ff:fed5:f…    ff02::9        RIPng        146  Command Response, Version 1
    10 83.304821   fe80::481a:48ff:fed7:2…    ff02::9        RIPng        166  Command Response, Version 1
```

```
▸ Frame 3: 146 bytes on wire (1168 bits), 146 bytes captured (1168 bits)
▸ Ethernet II, Src: 5e:73:a5:d5:f6:84 (5e:73:a5:d5:f6:84), Dst: 4a:1a:48:d7:29:2e (4a:1a:48:d7:29:2e)
▾ Internet Protocol Version 6, Src: fe80::5c73:a5ff:fed5:f684, Dst: fe80::481a:48ff:fed7:292e
    0110 .... = Version: 6
  ▸ .... 1100 0000 .... .... .... .... .... = Traffic Class: 0xc0 (DSCP: CS6, ECN: Not-ECT)
    .... .... .... 1100 0111 0001 1111 1100 = Flow Label: 0xc71fc
    Payload Length: 92
    Next Header: UDP (17)
    Hop Limit: 64
    Source: fe80::5c73:a5ff:fed5:f684
    Destination: fe80::481a:48ff:fed7:292e
▸ User Datagram Protocol, Src Port: 521, Dst Port: 521
▾ RIPng
    Command: Response (2)
    Version: 1
    Reserved: 0000
  ▾ Route Table Entry: IPv6 Prefix: 2001:db8:1341:3::/64 Metric: 2
      IPv6 Prefix: 2001:db8:1341:3::
      Route Tag: 0x0000
      Prefix Length: 64
      Metric: 2
  ▾ Route Table Entry: IPv6 Prefix: 2001:db8:1341:12::/64 Metric: 1
      IPv6 Prefix: 2001:db8:1341:12::
      Route Tag: 0x0000
      Prefix Length: 64
      Metric: 1
  ▾ Route Table Entry: IPv6 Prefix: 2001:db8:1341:13::/64 Metric: 2
      IPv6 Prefix: 2001:db8:1341:13::
```

```
0000  4a 1a 48 d7 29 2e 5e 73  a5 d5 f6 84 86 dd 6c 0c   J·H·).^s ······l·
0010  71 fc 00 5c 11 40 fe 80  00 00 00 00 00 00 5c 73   q··\·@·· ······\s
```

```
ripng-r2-trace.pcap          Packets: 10 · Displayed: 10 (100.0%)     Profile: Default
```

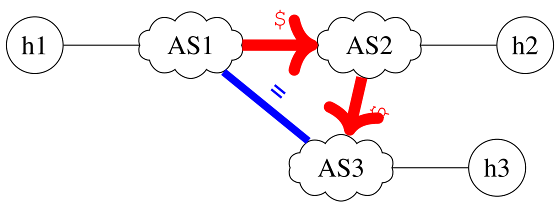


Fig. 13: A simple Internet

(continued from previous page)

```
        AS1 --$--> AS2 --$--> AS3
         |                    |
         +-----------=---------+
        """
        # The remaining code snippets go here
```

As in the previous examples, we create the routers and associate one IPv6 prefix to each AS:

- `AS1` is assigned `2001:cafe:1::/48`
- `AS2` is assigned `2001:cafe:2::/48`
- `AS3` is assigned `2001:cafe:3::/48`

```
# Add all routers
as1 = self.addRouter('as1')
as2 = self.addRouter('as2')
as3 = self.addRouter('as3')

routers=self.routers()
prefix = {routers[i]: '2001:cafe:%04x::/48' % (i+1) for i in range(len(routers ))}

as1.addDaemon(BGP, address_families=(AF_INET6(networks=(prefix[as1],)),))
as2.addDaemon(BGP, address_families=(AF_INET6(networks=(prefix[as2],)),))
as3.addDaemon(BGP, address_families=(AF_INET6(networks=(prefix[as3],)),))
```

The *addDaemon* method adds a BGP daemon on each router and configures it to advertise the IPv6 prefix allocated to this AS. We then create all the links and manually assign one IPv6 subnet to each link and one IPv6 address to each interface. For the interdomain links, we use an IPv6 prefix that belongs to one of the attached ASes.

```
# Add hosts

h1= self.addHost("h1")
h2= self.addHost("h2")
h3= self.addHost("h3")


# Add all links
l12=self.addLink(as1, as2)
l12[as1].addParams(ip="2001:cafe:1:12::1/64")
l12[as2].addParams(ip="2001:cafe:1:12::2/64")

l13=self.addLink(as1, as3)
l13[as1].addParams(ip="2001:cafe:1:13::1/64")
l13[as3].addParams(ip="2001:cafe:1:13::3/64")
l23=self.addLink(as2, as3)
l23[as2].addParams(ip="2001:cafe:2:23::2/64")
l23[as3].addParams(ip="2001:cafe:2:23::3/64")

# Links to the hosts
las1h1 = self.addLink(as1, h1)
las1h1[as1].addParams(ip=("2001:cafe:1:1::1/64"))
las1h1[h1].addParams(ip=("2001:cafe:1:1::11/64"))

las2h2 = self.addLink(as2, h2)
las2h2[as2].addParams(ip=("2001:cafe:2:1::2/64"))
las2h2[h2].addParams(ip=("2001:cafe:2:1::12/64"))
```

(continues on next page)

(continued from previous page)

```

las3h3 = self.addLink(as3, h3)
las3h3[as3].addParams(ip=("2001:cafe:3:1::3/64"))
las3h3[h3].addParams(ip=("2001:cafe:3:1::13/64"))
```

The last step is to specify to which AS each router belongs and to configure the eBGP sessions and their routing policies. IPMininet abstracts most of the complexity of the configuration of these policies by supporting two policies

```
# Set AS-ownerships
self.addAS(1, (as1,))
self.addAS(2, (as2,))
self.addAS(3, (as3,))

# Add eBGP sessions
# AS1 is a client of AS2
ebgp_session(self, as1, as2, link_type=CLIENT_PROVIDER)
# AS2 is a client of AS3
ebgp_session(self, as2, as3, link_type=CLIENT_PROVIDER)
# AS1 and AS3 are shared cost peers
ebgp_session(self, as1, as3, link_type=SHARE)

super(MyTopology, self).build(*args, **kwargs)
```

The script ends by launching the full topology. The entire script is available from `/exercises/ipmininet_scripts/ebgp-simple.py`.

We can now run this simple network.

```
sudo python3 ebgp-simple.py
```

If you launch the script and immediately type `ping6all` to check the connectivity, you might obtained the following result.

```
mininet> ping6all
*** Ping: testing reachability over IPv6
h1 --IPv6--> X X
h2 --IPv6--> X X
h3 --IPv6--> X X
*** Results: 100% dropped (0/6 received)
```

Remember that BGP is a distributed protocol and that it takes some time to launch the daemons and exchange the messages. After some time, the same command will confirm that everything works as expected.

```
mininet> ping6all
*** Ping: testing reachability over IPv6
h1 --IPv6--> h2 h3
h2 --IPv6--> h1 h3
h3 --IPv6--> h2 h1
*** Results: 0% dropped (6/6 received)
```

We can also use *traceroute6(8)* to check the path followed by the packets. Before doing that, think about the configuration of the BGP routing policies and try to predict the output of *traceroute6(8)*. This is a good exercise to check your understanding of BGP.

We have configured the following addresses on the hosts.

```
mininet> h1 ip -6 -o addr show
1: lo    inet6 ::1/128 scope host \       valid_lft forever preferred_lft forever
2: h1-eth0    inet6 2001:cafe:1:1::11/64 scope global \       valid_lft forever␣
↪preferred_lft forever
2: h1-eth0    inet6 fe80::8ae:b0ff:fe9d:aefa/64 scope link \       valid_lft forever␣
↪preferred_lft forever
mininet> h2 ip -6 -o addr show
1: lo    inet6 ::1/128 scope host \       valid_lft forever preferred_lft forever
2: h2-eth0    inet6 2001:cafe:2:1::12/64 scope global \       valid_lft forever␣
↪preferred_lft forever
2: h2-eth0    inet6 fe80::d8a3:cdff:fed6:14ad/64 scope link \       valid_lft forever␣
↪preferred_lft forever
mininet> h3 ip -6 -o addr show
1: lo    inet6 ::1/128 scope host \       valid_lft forever preferred_lft forever
2: h3-eth0    inet6 2001:cafe:3:1::13/64 scope global \       valid_lft forever␣
↪preferred_lft forever
2: h3-eth0    inet6 fe80::101d:e1ff:fe4e:a3a9/64 scope link \       valid_lft forever␣
↪preferred_lft forever
```

We can now explore the routes in this small Internet. Host `h1` can reach directly host `h3`.

```
mininet> h1 traceroute6 -n 2001:cafe:3:1::13
traceroute to 2001:cafe:3:1::13 (2001:cafe:3:1::13) from 2001:cafe:1:1::11, 30 hops␣
↪max, 24 byte packets
1  2001:cafe:1:1::1  0.099 ms  0.038 ms  0.056 ms
2  2001:cafe:2:23::3  2.3 ms  0.135 ms  0.161 ms
3  2001:cafe:3:1::13  0.216 ms  0.182 ms  0.187 ms
```

Note that the path preferred by `AS3` to reach `AS1` is different.

```
mininet> h3 traceroute6 -n 2001:cafe:1:1::11
traceroute to 2001:cafe:1:1::11 (2001:cafe:1:1::11) from 2001:cafe:3:1::13, 30 hops␣
↪max, 24 byte packets
1  2001:cafe:3:1::3  0.133 ms  0.088 ms  0.078 ms
2  2001:cafe:2:23::2  0.099 ms  0.085 ms  0.086 ms
3  2001:cafe:1:13::1  0.103 ms  0.097 ms  0.083 ms
4  2001:cafe:1:1::11  0.075 ms  0.037 ms  0.062 ms
```

The same applies for the paths between `h1` and `h2`

```
mininet> h1 traceroute6 -n 2001:cafe:2:1::12
traceroute to 2001:cafe:2:1::12 (2001:cafe:2:1::12) from 2001:cafe:1:1::11, 30 hops␣
↪max, 24 byte packets
1  2001:cafe:1:1::1  0.102 ms  0.03 ms  0.026 ms
2  2001:cafe:2:23::3  0.051 ms  0.034 ms  0.033 ms
3  2001:cafe:1:12::2  0.036 ms  0.034 ms  0.031 ms
4  2001:cafe:2:1::12  0.043 ms  0.207 ms  0.17 ms
mininet> h2 traceroute6 -n 2001:cafe:1:1::11
traceroute to 2001:cafe:1:1::11 (2001:cafe:1:1::11) from 2001:cafe:2:1::12, 30 hops␣
↪max, 24 byte packets
1  2001:cafe:2:1::2  0.075 ms  0.088 ms  0.029 ms
2  2001:cafe:1:13::1  0.059 ms  0.052 ms  0.034 ms
3  2001:cafe:1:1::11  0.05 ms  0.036 ms  0.031 ms
```

Besides *`ping6(8)`* and *`traceroute6(8)`*, it is also useful to interact with the BGP daemon that runs on each of our routers. This is done by connecting on the Command Line Interface of the BGP router using telnet.

```
mininet> noecho as1 telnet localhost bgpd
Trying ::1...
Connected to localhost.
Escape character is '^]'.

Hello, this is FRRouting (version 7.1).
Copyright 1996-2005 Kunihiro Ishiguro, et al.


User Access Verification

Password:
```

The password for the BGP daemon is *zebra*. The *noecho* command indicates that mininet does not need to echo the characters that you type. You then enter the Quagga VTY that enables you to type commands. The *help* commands gives you some information about the available commands as well as *?*.

```
as1> help
Quagga VTY provides advanced help feature.  When you need help,
anytime at the command line please press '?'.

If nothing matches, the help list will be empty and you must backup
until entering a '?' shows the available options.
Two styles of help are provided:
1. Full help is available when you are ready to enter a
   command argument (e.g. 'show ?') and describes each possible
   argument.
2. Partial help is provided when an abbreviated argument is entered
   and you want to know what arguments match the input
   (e.g. 'show me?'.)

as1>
enable    Turn on privileged mode command
exit      Exit current mode and down to previous mode
find      Find CLI command containing text
help      Description of the interactive help system
list      Print command list
quit      Exit current mode and down to previous mode
show      Show running system information
terminal  Set terminal line parameters
who       Display who is on vty
as1>
```

In these exercises, we mainly consider the `show` that extracts information from the BGP daemon. We type *show bgp* and press the *tabulation* key to see the available commands in the *show bgp*.

```
as1> show bgp
as-path-access-list attribute-info cidr-only  community  community-info community-list
dampening  detail     extcommunity-list filter-list import-check-table ipv4
ipv6       json       l2vpn      large-community large-community-list mac
martian    memory     multicast  neighbors  nexthop    paths
peer-group peerhash   prefix-list regexp     route-leak route-map
statistics summary    unicast    update-groups view       views
vpn        vrf        vrfs
```

A useful command to start is *show bgp summary* which provides a summary of the state of the BGP daemon.

```
as1> show bgp summary

IPv6 Unicast Summary:
BGP router identifier 192.168.8.2, local AS number 1 vrf-id 0
BGP table version 10
RIB entries 5, using 800 bytes of memory
Peers 2, using 41 KiB of memory

Neighbor          V         AS MsgRcvd MsgSent   TblVer  InQ OutQ  Up/Down State/
↪PfxRcd
2001:cafe:1:12::2 4          2      52      51        0    0    0 00:15:21            ␣
↪2
2001:cafe:1:13::3 4          3      51      50        0    0    0 00:15:21            ␣
↪2

Total number of neighbors 2
```

This router (*as1*) has two BGP neighbors: `2001:cafe:1:12::2` and `2001:cafe:1:13::3`. Both BGP sessions are established using the current version of the protocol (version 4). About 50 messages were sent/received over each session. These messages are mainly the BGP Keepalive messages that are exchanged every 30 seconds. The last column indicates that two prefixes were received over each session. We can see more details about these two eBGP sessions with the *show bgp neighbors* command.

```
as1> show bgp ipv6 neighbors

BGP neighbor is 2001:cafe:1:12::2, remote AS 2, local AS 1, external link
  Description: as2 (eBGP)
Hostname: as2
BGP version 4, remote router ID 192.168.3.1, local router ID 192.168.8.2
BGP state = Established, up for 00:41:48
Last read 00:00:48, Last write 00:00:48
Hold time is 180, keepalive interval is 60 seconds
Neighbor capabilities:
 4 Byte AS: advertised and received
 AddPath:
   IPv6 Unicast: RX advertised IPv6 Unicast and received
 Route refresh: advertised and received(old & new)
 Address Family IPv6 Unicast: advertised and received
 Hostname Capability: advertised (name: as1,domain name: n/a) received (name: as2,
↪domain name: n/a)
 Graceful Restart Capabilty: advertised and received
   Remote Restart timer is 120 seconds
   Address families by peer:
     none
Graceful restart information:
 End-of-RIB send: IPv6 Unicast
 End-of-RIB received: IPv6 Unicast
Message statistics:
 Inq depth is 0
 Outq depth is 0
                      Sent       Rcvd
 Opens:                  2          2
 Notifications:          2          0
 Updates:                5          9
 Keepalives:            67         66
 Route Refresh:          1          1
 Capability:             0          0
```

(continues on next page)

(continued from previous page)

```
 Total:                 77         78
Minimum time between advertisement runs is 0 seconds

For address family: IPv6 Unicast
 Update group 4, subgroup 4
 Packet Queue length 0
 NEXT_HOP is always this router
 Community attribute sent to this neighbor(all)
 Inbound path policy configured
 Outbound path policy configured
 Route map for incoming advertisements is *rm3-ipv6
 Route map for outgoing advertisements is *export-to-up-as2-ipv6
 2 accepted prefixes

 Connections established 2; dropped 1
 Last reset 00:41:49, due to NOTIFICATION sent (Hold Timer Expired)
 External BGP neighbor may be up to 255 hops away.
 Local host: 2001:cafe:1:12::1, Local port: 179
 Foreign host: 2001:cafe:1:12::2, Foreign port: 51406
 Nexthop: 192.168.0.1
 Nexthop global: 2001:cafe:1:12::1
 Nexthop local: fe80::fca2:adff:fe20:4cb1
 BGP connection: shared network
 BGP Connect Retry Timer in Seconds: 120
 Read thread: on  Write thread: on

 BGP neighbor is 2001:cafe:1:13::3, remote AS 3, local AS 1, external link
 Description: as3 (eBGP)
  Hostname: as3
  BGP version 4, remote router ID 192.168.6.1, local router ID 192.168.8.2
  BGP state = Established, up for 00:41:48
  Last read 00:00:48, Last write 00:00:48
  Hold time is 180, keepalive interval is 60 seconds
  Neighbor capabilities:
   4 Byte AS: advertised and received
  AddPath:
   IPv6 Unicast: RX advertised IPv6 Unicast and received
 Route refresh: advertised and received(old & new)
 Address Family IPv6 Unicast: advertised and received
 Hostname Capability: advertised (name: as1,domain name: n/a) received (name: as3,
↪domain name: n/a)
 Graceful Restart Capabilty: advertised and received
   Remote Restart timer is 120 seconds
   Address families by peer:
     none
 Graceful restart information:
  End-of-RIB send: IPv6 Unicast
  End-of-RIB received: IPv6 Unicast
 Message statistics:
  Inq depth is 0
  Outq depth is 0
                     Sent       Rcvd
 Opens:                 2          2
 Notifications:         2          0
 Updates:               5          8
 Keepalives:           66         66
 Route Refresh:         1          1
```

(continues on next page)

(continued from previous page)

```
 Capability:             0          0
 Total:                 76         77
 Minimum time between advertisement runs is 0 seconds

 For address family: IPv6 Unicast
  Update group 3, subgroup 3
  Packet Queue length 0
  NEXT_HOP is always this router
  Community attribute sent to this neighbor(all)
  Inbound path policy configured
  Outbound path policy configured
  Route map for incoming advertisements is *rm13-ipv6
  Route map for outgoing advertisements is *export-to-peer-as3-ipv6
  2 accepted prefixes

  Connections established 2; dropped 1
  Last reset 00:41:49, due to Peer closed the session
  External BGP neighbor may be up to 255 hops away.
  Local host: 2001:cafe:1:13::1, Local port: 179
  Foreign host: 2001:cafe:1:13::3, Foreign port: 41630
  Nexthop: 192.168.4.2
  Nexthop global: 2001:cafe:1:13::1
  Nexthop local: fe80::f823:70ff:fe80:37c4
  BGP connection: shared network
  BGP Connect Retry Timer in Seconds: 120
  Read thread: on  Write thread: on
```

We can now observe the BGP-Loc-RIB of the router with the `show bgp ipv6 command` command.

```
as1> show bgp ipv6
BGP table version is 10, local router ID is 192.168.8.2, vrf id 0
Default local pref 100, local AS 1
Status codes:  s suppressed, d damped, h history, * valid, > best, = multipath,
               i internal, r RIB-failure, S Stale, R Removed
Nexthop codes: @NNN nexthop's vrf id, < announce-nh-self
Origin codes:  i - IGP, e - EGP, ? - incomplete

Network          Next Hop            Metric LocPrf Weight Path
*> 2001:cafe:1::/48 ::                       0         32768 i
*> 2001:cafe:2::/48 fe80::3c81:2eff:fe19:465d
                                              150      0 3 2 i
*                   fe80::c001:dcff:fe49:a512
                                          0    100      0 2 i
*  2001:cafe:3::/48 fe80::c001:dcff:fe49:a512
                                              100      0 2 3 i
*>                  fe80::3c81:2eff:fe19:465d
                                          0    150      0 3 i

Displayed  3 routes and 5 total paths
```

It is interesting to look at the output of this command in details. Router *as1* has routes for three different IPv6 prefixes. The first prefix is its own prefix, `2001:cafe:1::/48`. It has no nexthop since this prefix is originated by the router. Then, *as1* has received two paths for `2001:cafe:2::/48`. In the BGP Loc-RIB, the > character indicates the best route according to the BGP decision process. `2001:cafe:2::/48` was learned over two different BGP sessions:

- the eBGP session with `fe80::3c81:2eff:fe19:465d` with an AS-Path of `AS3:AS2` (see last column)
- the eBGP session with `fe80::c001:dcff:fe49:a512` with an AS-Path of `AS2` (see last column)

The first of these two routes is preferred as indicated by the > character because it has a higher `local-preference` (150) than the second one (100). For prefix ``2001:cafe:3::/48``, the route learned via `fe80::3c81:2eff:fe19:465d` is also preferred for the same reason.

IPMininet also allows to explore the dynamics of BGP by looking at the packets that the routers exchange. For this, we slightly modify the example above and add delays to the interdomain links as follows.

```
l12=self.addLink(as1, as2, delay='10ms')
l13=self.addLink(as1, as3, delay='10ms')
l23=self.addLink(as2, as3, delay='200ms')
```

We also add a `post_build` method to launch tcpdump and capture the BGP packets exchanged by the routers. A BGP session runs over a TCP connection. Let us examine a few of the BGP messages exchanged on the BGP session between `AS1` and `AS2`. The traces collected on the three routers are available from `/exercises/traces/bgp-as1-trace.pcap`, `/exercises/traces/bgp-as2-trace.pcap` and `/exercises/traces/bgp-as2-trace.pcap`.

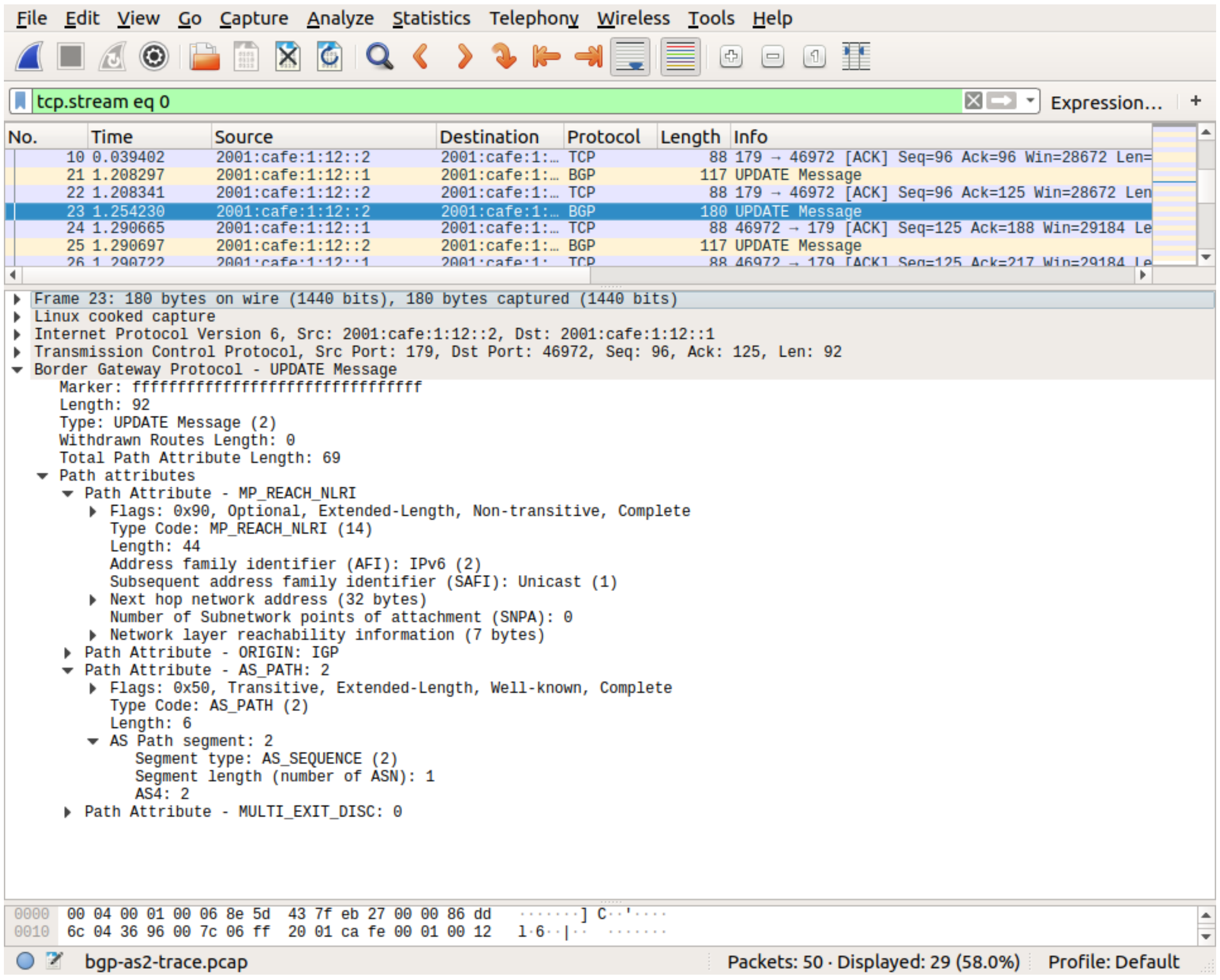
The BGP session starts with a TCP three-way handshake. Once the session has been established, both BGP daemons send an `OPEN` message describing their capabilities and the BGP extensions that it supports. The details of these extensions go beyond the scope of this book. However, it is important to note that the `OPEN` message contains the `AS` number of the router that sends the message and its identifier as a 32 bits IPv4 address. This router identifier uniquely identifies the router. The last mandatory parameter of the `OPEN` message is the *Hold Time*, i.e. the maximum delay between two successive messages over this BGP session. A BGP router should send `KEEPALIVE` messages every one third of the *Hold Time* to keep the session up.

The `UPDATE` message can be used to withdraw and advertise routes. The packet below is sent by `AS2` to advertise its route towards *2001:cafe:2::/48* on the BGP session with `AS1`.

Another interesting utilization of IPMininet is to explore how routers react to link failures. We start from the same network as with the previous example and disable the link between `AS2` and `AS3`. For this, we log on one of the two routers and issue the following commands.

```
mininet> noecho as2 telnet localhost bgpd
Trying ::1...
Connected to localhost.
Escape character is '^]'.

Hello, this is FRRouting (version 7.1).
Copyright 1996-2005 Kunihiro Ishiguro, et al.


User Access Verification

Password:
as2> enable
as2# show bgp summary

IPv6 Unicast Summary:
BGP router identifier 192.168.8.1, local AS number 2 vrf-id 0
BGP table version 5
RIB entries 5, using 800 bytes of memory
Peers 2, using 41 KiB of memory

Neighbor           V         AS  MsgRcvd MsgSent   TblVer  InQ OutQ  Up/Down State/
↪PfxRcd
2001:cafe:1:12::1 4          1      13      17        0    0    0 00:07:28          ␣
↪1
2001:cafe:2:23::3 4          3      20      20        0    0    0 00:00:55          ␣
↪1
```

(continues on next page)

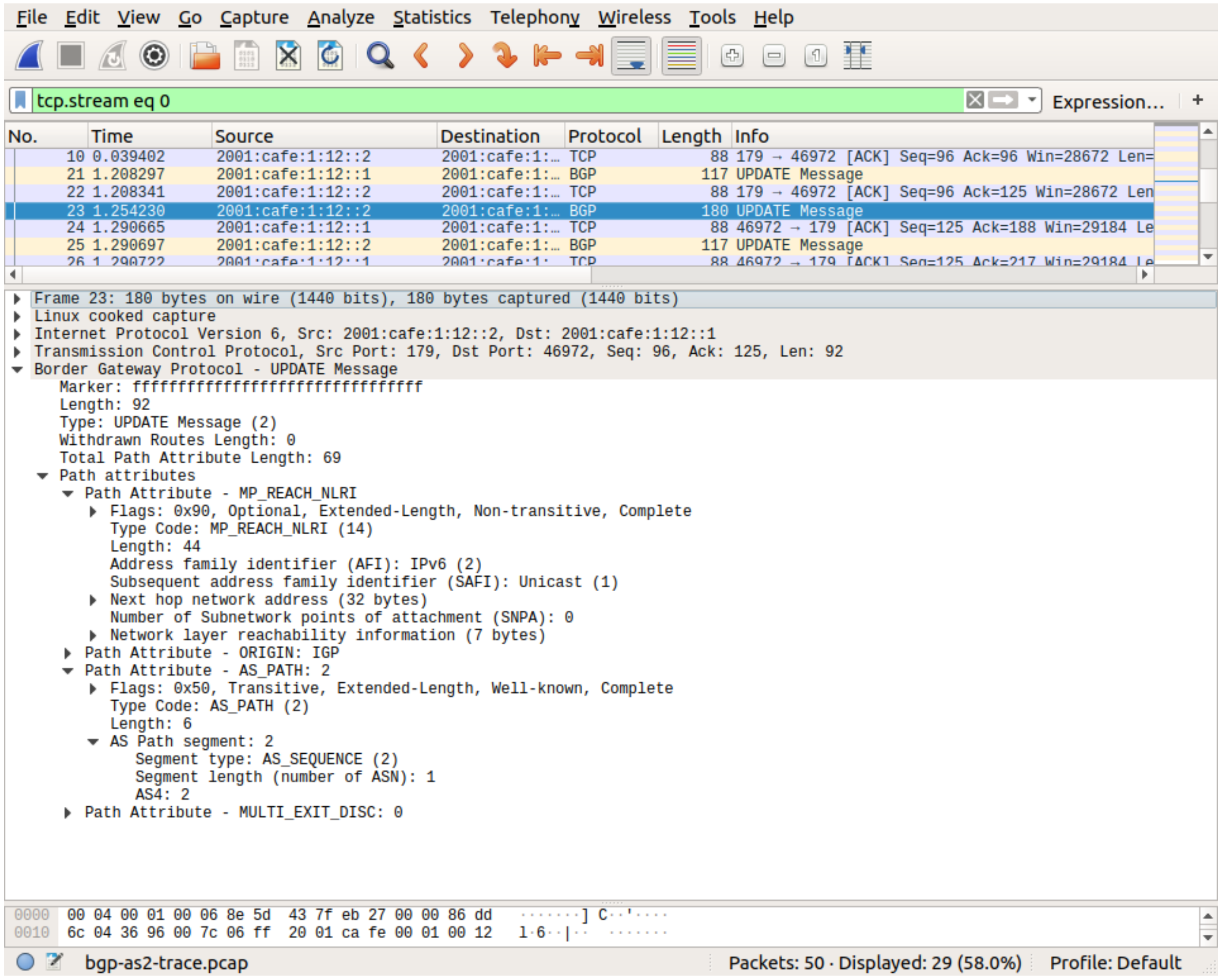

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp.stream eq 0
Expression…
+

```
No.  Time       Source             Destination    Protocol  Length Info
  10 0.039402   2001:cafe:1:12::2  2001:cafe:1:…  TCP        88 179 → 46972 [ACK] Seq=96 Ack=96 Win=28672 Len=
  21 1.208297   2001:cafe:1:12::1  2001:cafe:1:…  BGP       117 UPDATE Message
  22 1.208341   2001:cafe:1:12::2  2001:cafe:1:…  TCP        88 179 → 46972 [ACK] Seq=96 Ack=125 Win=28672 Len
  23 1.254230   2001:cafe:1:12::2  2001:cafe:1:…  BGP       180 UPDATE Message
  24 1.290665   2001:cafe:1:12::1  2001:cafe:1:…  TCP        88 46972 → 179 [ACK] Seq=125 Ack=188 Win=29184 Le
  25 1.290697   2001:cafe:1:12::2  2001:cafe:1:…  BGP       117 UPDATE Message
  26 1.290722   2001:cafe:1:12::1  2001:cafe:1:…  TCP        88 46972 → 179 [ACK] Seq=125 Ack=217 Win=29184 Le
```

```
▸ Frame 23: 180 bytes on wire (1440 bits), 180 bytes captured (1440 bits)
▸ Linux cooked capture
▸ Internet Protocol Version 6, Src: 2001:cafe:1:12::2, Dst: 2001:cafe:1:12::1
▸ Transmission Control Protocol, Src Port: 179, Dst Port: 46972, Seq: 96, Ack: 125, Len: 92
▾ Border Gateway Protocol - UPDATE Message
    Marker: ffffffffffffffffffffffffffffffff
    Length: 92
    Type: UPDATE Message (2)
    Withdrawn Routes Length: 0
    Total Path Attribute Length: 69
  ▾ Path attributes
    ▾ Path Attribute - MP_REACH_NLRI
      ▸ Flags: 0x90, Optional, Extended-Length, Non-transitive, Complete
        Type Code: MP_REACH_NLRI (14)
        Length: 44
        Address family identifier (AFI): IPv6 (2)
        Subsequent address family identifier (SAFI): Unicast (1)
      ▸ Next hop network address (32 bytes)
        Number of Subnetwork points of attachment (SNPA): 0
      ▸ Network layer reachability information (7 bytes)
    ▸ Path Attribute - ORIGIN: IGP
    ▾ Path Attribute - AS_PATH: 2
      ▸ Flags: 0x50, Transitive, Extended-Length, Well-known, Complete
        Type Code: AS_PATH (2)
        Length: 6
      ▾ AS Path segment: 2
          Segment type: AS_SEQUENCE (2)
          Segment length (number of ASN): 1
          AS4: 2
    ▸ Path Attribute - MULTI_EXIT_DISC: 0
```

```
0000  00 04 00 01 00 06 8e 5d  43 7f eb 27 00 00 86 dd   ·······] C··'····
0010  6c 04 36 96 00 7c 06 ff  20 01 ca fe 00 01 00 12   l·6··|··  ·······
```

```
bgp-as2-trace.pcap        Packets: 50 · Displayed: 29 (58.0%)    Profile: Default
```


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp.stream eq 0
Expression…
+

```
No.  Time       Source              Destination     Protocol  Length Info
  10 0.039402   2001:cafe:1:12::2   2001:cafe:1:…   TCP           88 179 → 46972 [ACK] Seq=96 Ack=96 Win=28672 Len=
  21 1.208297   2001:cafe:1:12::1   2001:cafe:1:…   BGP          117 UPDATE Message
  22 1.208341   2001:cafe:1:12::2   2001:cafe:1:…   TCP           88 179 → 46972 [ACK] Seq=96 Ack=125 Win=28672 Len
  23 1.254230   2001:cafe:1:12::2   2001:cafe:1:…   BGP          180 UPDATE Message
  24 1.290665   2001:cafe:1:12::1   2001:cafe:1:…   TCP           88 46972 → 179 [ACK] Seq=125 Ack=188 Win=29184 Le
  25 1.290697   2001:cafe:1:12::2   2001:cafe:1:…   BGP          117 UPDATE Message
  26 1.290722   2001:cafe:1:12::1   2001:cafe:1:…   TCP           88 46972 → 179 [ACK] Seq=125 Ack=217 Win=29184 Le
```

```
▸ Frame 23: 180 bytes on wire (1440 bits), 180 bytes captured (1440 bits)
▸ Linux cooked capture
▸ Internet Protocol Version 6, Src: 2001:cafe:1:12::2, Dst: 2001:cafe:1:12::1
▸ Transmission Control Protocol, Src Port: 179, Dst Port: 46972, Seq: 96, Ack: 125, Len: 92
▾ Border Gateway Protocol - UPDATE Message
    Marker: ffffffffffffffffffffffffffffffff
    Length: 92
    Type: UPDATE Message (2)
    Withdrawn Routes Length: 0
    Total Path Attribute Length: 69
  ▾ Path attributes
    ▾ Path Attribute - MP_REACH_NLRI
      ▸ Flags: 0x90, Optional, Extended-Length, Non-transitive, Complete
        Type Code: MP_REACH_NLRI (14)
        Length: 44
        Address family identifier (AFI): IPv6 (2)
        Subsequent address family identifier (SAFI): Unicast (1)
      ▸ Next hop network address (32 bytes)
        Number of Subnetwork points of attachment (SNPA): 0
      ▸ Network layer reachability information (7 bytes)
    ▸ Path Attribute - ORIGIN: IGP
    ▾ Path Attribute - AS_PATH: 2
      ▸ Flags: 0x50, Transitive, Extended-Length, Well-known, Complete
        Type Code: AS_PATH (2)
        Length: 6
      ▾ AS Path segment: 2
          Segment type: AS_SEQUENCE (2)
          Segment length (number of ASN): 1
          AS4: 2
    ▸ Path Attribute - MULTI_EXIT_DISC: 0
```

```
0000  00 04 00 01 00 06 8e 5d  43 7f eb 27 00 00 86 dd   ·······] C··'····
0010  6c 04 36 96 00 7c 06 ff  20 01 ca fe 00 01 00 12   l·6··|··  ·······
```

```
bgp-as2-trace.pcap        Packets: 50 · Displayed: 29 (58.0%)    Profile: Default
```

(continued from previous page)

```

Total number of neighbors 2
```

We first connect to the BGP daemon on router *as2*. In addition to the *show* commands that have been described earlier, the router also supports privileged commands that can change its configuration. Before executing these commands, we must enter the privileged mode with the *enable* command. On production routers, this command requires a password to verify the credentials of the network administrator. The # prompt indicates that we are allowed to execute privileged commands. We first check the state of the BGP sessions with the *show bgp summary* commands. There are two BGP sessions configured on this router.

We can now disable one of the BGP sessions on router *as2* as follows.

```
as2# configure terminal
as2(config)# router bgp 2
as2(config-router)# neighbor 2001:cafe:2:23::3 shutdown
as2(config-router)# exit
as2(config)# exit
```

We start indicate that we will use the terminal to change the router configuration with *configure terminal*. We then enter the BGP part of the configuration with *router bgp 2* (*2* is the AS number of *as2*). Then we use the *neighbor 2001:cafe:2:23::3 shutdown* that takes as parameter the IP address of the peer of the session that we want to stop. We then leave the BGP part of the configuration (first *exit*) and the configuration menu (second *exit* command). At this point, the BGP session between `AS2` and `AS3` is down.

```
as2# show bgp summary

IPv6 Unicast Summary:
BGP router identifier 192.168.8.1, local AS number 2 vrf-id 0
BGP table version 6
RIB entries 3, using 480 bytes of memory
Peers 2, using 41 KiB of memory

Neighbor          V         AS MsgRcvd MsgSent   TblVer  InQ OutQ  Up/Down State/
↪PfxRcd
2001:cafe:1:12::1 4          1      14      19        0    0    0 00:08:02          ␣
↪1
2001:cafe:2:23::3 4          3      21      23        0    0    0 00:00:10 Idle␣
↪(Admin)

Total number of neighbors 2
as2# exit
```

Without a BGP session between `AS2` and `AS3`, there are reachability problems in this simple Internet.

```
mininet> ping6all
*** Ping: testing reachability over IPv6
h1 --IPv6--> h2 h3
h2 --IPv6--> X h1
h3 --IPv6--> X h1
*** Results: 33% dropped (4/6 received)
```

We can fix them by enabling again the BGP session with the *no neighbor 2001:cafe:2:23::3 shutdown* command.

```
mininet> noecho as2 telnet localhost bgpd
Trying ::1...
Connected to localhost.
```

(continues on next page)

(continued from previous page)

```
Escape character is '^]'.

Hello, this is FRRouting (version 7.1).
Copyright 1996-2005 Kunihiro Ishiguro, et al.


User Access Verification

Password:
as2> enable
as2# configure terminal
as2(config)# router bgp 2
as2(config-router)# no neighbor 2001:cafe:2:23::3 shutdown
as2(config-router)# exi
as2(config)# exit
as2# show bgp summary

IPv6 Unicast Summary:
BGP router identifier 192.168.8.1, local AS number 2 vrf-id 0
BGP table version 7
RIB entries 5, using 800 bytes of memory
Peers 2, using 41 KiB of memory

Neighbor          V         AS MsgRcvd MsgSent   TblVer  InQ OutQ  Up/Down State/
↪PfxRcd
2001:cafe:1:12::1 4          1      15      21        0    0    0 00:09:30          ␣
↪1
2001:cafe:2:23::3 4          3      28      28        0    0    0 00:00:07          ␣
↪1

Total number of neighbors 2
as2# exit
Connection closed by foreign host.
mininet> ping6all
*** Ping: testing reachability over IPv6
h1 --IPv6--> h2 h3
h2 --IPv6--> h3 h1
h3 --IPv6--> h2 h1
*** Results: 0% dropped (6/6 received)
```

4.16.4 Exercises

We can use IPMininet to prepare some networks with problems that need to be analyzed and corrected.

1. Our first example is a small Internet with 5 ASes. A subset of the script that configures this network is shown below. There is one host attached to each AS and this host has the same number as its AS. The entire script is available from `/exercises/ipmininet_scripts/ebgp-bug.py`.

```
l12=self.addLink(as1, as2, delay='10ms')
l13=self.addLink(as1, as3, delay='10ms')
l23=self.addLink(as2, as3, delay='10ms')
l15=self.addLink(as1, as5, delay='10ms')
l24=self.addLink(as2, as4, delay='10ms')
l34=self.addLink(as3, as4, delay='10ms')
l45=self.addLink(as4, as5, delay='10ms')
```

(continues on next page)

(continued from previous page)

```
# Add eBGP sessions
ebgp_session(self, as2, as1, link_type=CLIENT_PROVIDER)
ebgp_session(self, as3, as1, link_type=CLIENT_PROVIDER)
ebgp_session(self, as5, as1, link_type=CLIENT_PROVIDER)
ebgp_session(self, as3, as4, link_type=CLIENT_PROVIDER)

ebgp_session(self, as2, as3, link_type=SHARE)
ebgp_session(self, as2, as4, link_type=SHARE)
ebgp_session(self, as4, as5, link_type=SHARE)
```

When this network is launched, *ping6all* reports connectivity problems. Hosts *h1* and *h4* cannot exchange packets. Can you fix the problem by changing the routing policy used on only one interdomain link ? Justify your answer

```
mininet> ping6all
*** Ping: testing reachability over IPv6
h1 --IPv6--> h2 h5 X h3
h2 --IPv6--> h3 h1 h4 h5
h3 --IPv6--> h2 h1 h4 h5
h4 --IPv6--> h2 X h5 h3
h5 --IPv6--> h2 h1 h4 h3
*** Results: 10% dropped (18/20 received)
```

2. Another interesting utilization IPMininet is to explore the impact of a link failure. We start from a small variant of the above topology.

```
l12=self.addLink(as1, as2, delay='10ms')
l13=self.addLink(as1, as3, delay='10ms')
l23=self.addLink(as2, as3, delay='10ms')
l15=self.addLink(as1, as5, delay='10ms')
l24=self.addLink(as2, as4, delay='10ms')
l34=self.addLink(as3, as4, delay='10ms')
l45=self.addLink(as4, as5, delay='10ms')
l25=self.addLink(as2, as5, delay='10ms')
# Add eBGP sessions
ebgp_session(self, as2, as1, link_type=CLIENT_PROVIDER)
ebgp_session(self, as3, as1, link_type=CLIENT_PROVIDER)
ebgp_session(self, as5, as1, link_type=CLIENT_PROVIDER)
ebgp_session(self, as4, as3, link_type=CLIENT_PROVIDER)

ebgp_session(self, as2, as3, link_type=SHARE)
ebgp_session(self, as2, as4, link_type=SHARE)
ebgp_session(self, as4, as5, link_type=SHARE)
ebgp_session(self, as2, as5, link_type=SHARE)
```

When this network starts, all hosts can reach all other hosts.

```
mininet> ping6all
*** Ping: testing reachability over IPv6
h1 --IPv6--> h4 h3 h2 h5
h2 --IPv6--> h4 h1 h3 h5
h3 --IPv6--> h4 h1 h2 h5
h4 --IPv6--> h1 h3 h2 h5
h5 --IPv6--> h4 h1 h3 h2
*** Results: 0% dropped (20/20 received)
```

Draw the network and try to predict how it will react to a shutdown of any of the customer-provider links ?

a. What are the BGP messages that will be exchanged when the link between `AS1` and `AS2` fails ? How does this affect the reachability of the different hosts ?

b. What are the BGP messages that will be exchanged when the link between `AS1` and `AS3` fails ? How does this affect the reachability of the different hosts ?

c. What are the BGP messages that will be exchanged when the link between `AS1` and `AS5` fails ? How does this affect the reachability of the different hosts ?

d. What are the BGP messages that will be exchanged when the link between `AS3` and `AS4` fails ? How does this affect the reachability of the different hosts ?

3. Let us now consider another example. The network contains nine ASes with one host per AS. Assuming that `AS9` announces prefix *p9* and that `AS2` announces prefix *p2*.

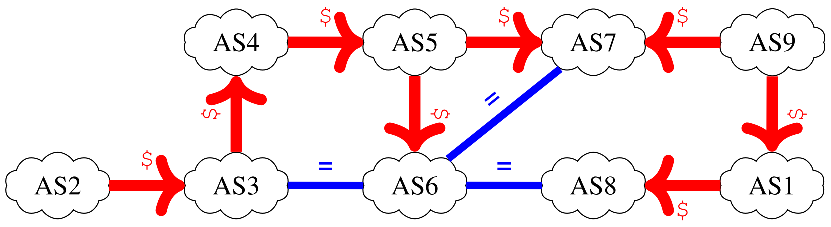


Fig. 14: A simple Internet

a. What is the Loc-RIB of `AS6` for prefix *p9* ? Indicate which is the best route towards this prefix.

b. What is the Loc-RIB of `AS9` for prefix *p2* ? Indicate which is the best route towards this prefix.

4. The network below contains nine ASes with one host per AS. Assuming that `AS1` announces prefix *p1* and that `AS2` announces prefix *p2*.

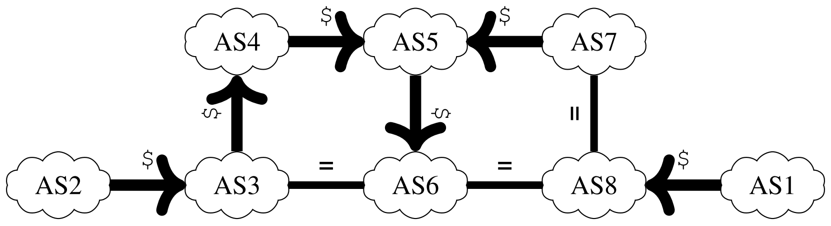


Fig. 15: A simple Internet

a. What is the Loc-RIB of `AS6` for prefix *p1* ? Indicate which is the best route towards this prefix.

b. What is the Loc-RIB of `AS8` for prefix *p2* ? Indicate which is the best route towards this prefix.

5. Let us now consider another example, also implemented using an IPMininet script. The network contains eight ASes with one host per AS. This small Internet is shown below and the script is available from `/exercises/ipmininet_scripts/ebgp-bug-3.py`.

a. The network does not provide a full connectivity. The hosts attached to `AS5` cannot ping the hosts attached to `AS8`.

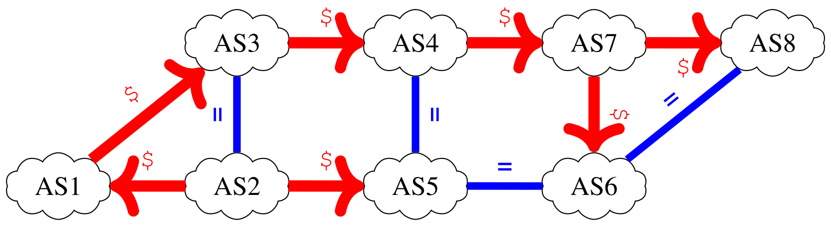


Fig. 16: A simple Internet

b. What is the path that packets follow from a host attached to `AS1` to a host attached to `AS8` ?
c. What is the path that packets follow from a host attached to `AS8` to a host attached to `AS1` ?

d. What is the path that packets follow from a host attached to `AS8` to a host attached to `AS2` ?

e. What is the path that packets follow from a host attached to `AS2` to a host attached to `AS7` ?

f. We now disable the interdomain link between `AS3` and `AS4`. What are the hosts that `AS1`, `AS5` and `AS6` are still able to ping ?

4.17 Local Area Networks: The Spanning Tree Protocol and Virtual LANs

4.17.1 Exercises

1. Consider the switched network shown in Fig. 1. What is the spanning tree that will be computed by 802.1d in this network assuming that all links have a unit cost ? Indicate the state of each port.

2. Consider the switched network shown in Fig. 1. In this network, assume that the LAN between switches S3 and S12 fails. How should the switches update their port/address tables after the link failure ?
3. Consider the switched network shown in the figure below. Compute the Spanning Tree of this network.

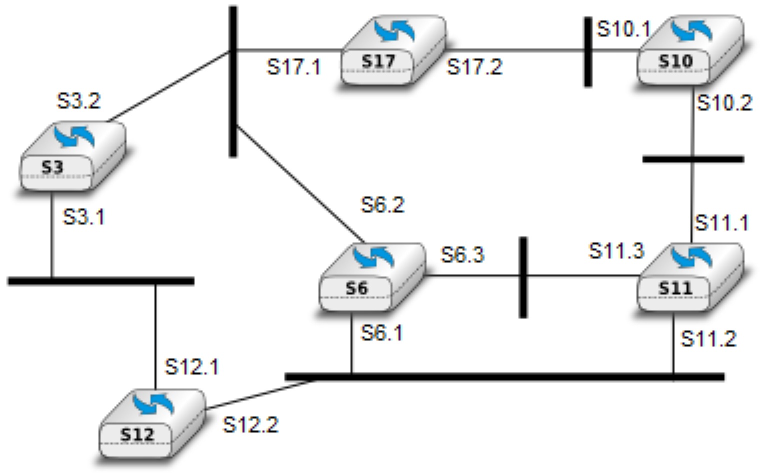
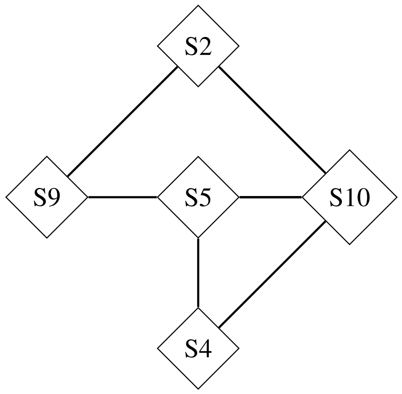


Fig. 17: Fig. 1. A small network composed of Ethernet switches



4. Many enterprise networks are organized with a set of backbone devices interconnected by using a full mesh of links as shown in Fig.2. In this network, what are the benefits and drawbacks of using Ethernet switches and IP routers running OSPF ?

5. In the network depicted in Fig. 3, the host *H0* performs a traceroute toward its peer *H1* (designated by its name) through a network composed of switches and routers. Explain precisely the frames, packets, and segments exchanged since the network was turned on. You may assign addresses if you need to.

6. In the network represented in Fig. 4, can the host *H0* communicate with *H1* and vice-versa? Explain. Add whatever you need in the network to allow them to communicate.

7. Consider the network depicted in Fig. 5. Both of the hosts *H0* and *H1* have two interfaces: one connected to the switch *S0* and the other one to the switch *S1*. Will the link between *S0* and *S1* ever be used? If so, under which assumptions? Provide a comprehensive answer.

8. Most commercial Ethernet switches are able to run the Spanning tree protocol independently on each VLAN. What are the benefits of using per-VLAN spanning trees ?

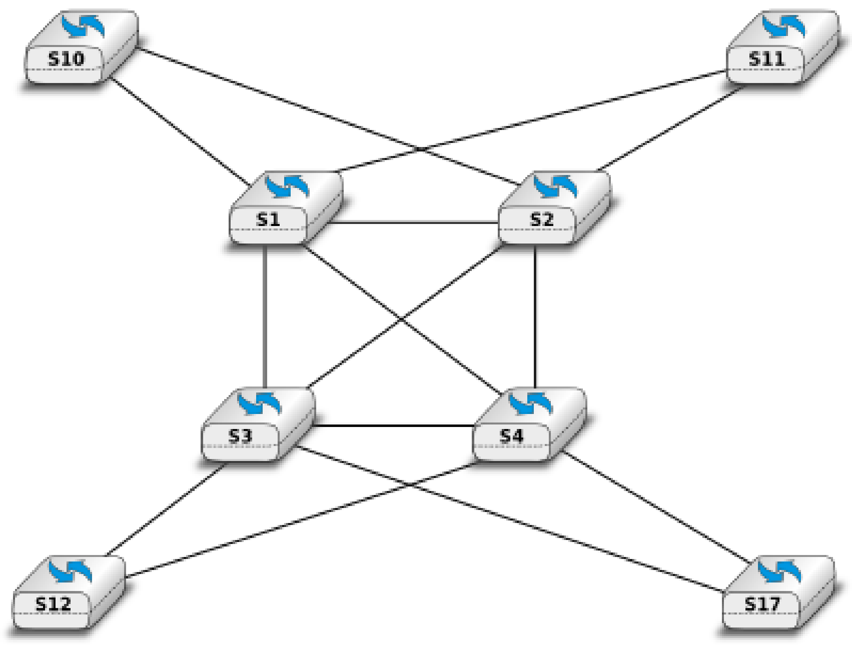


Fig. 18: Fig. 2. A typical enterprise backbone network

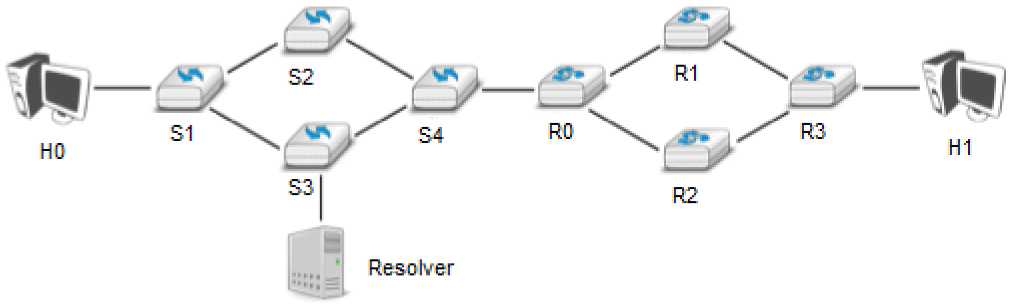


Fig. 19: Fig. 3. Host *H0* performs a traceroute towards its peer *H1* through a network composed of switches and routers

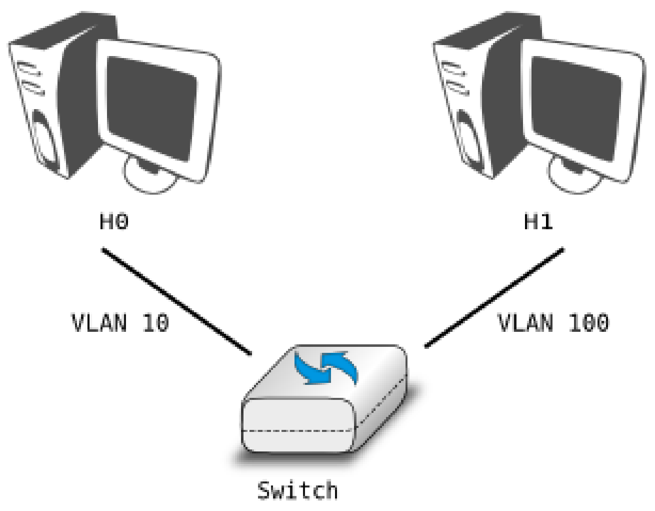


Fig. 20: Fig. 4. Can *H0* and *H1* communicate ?

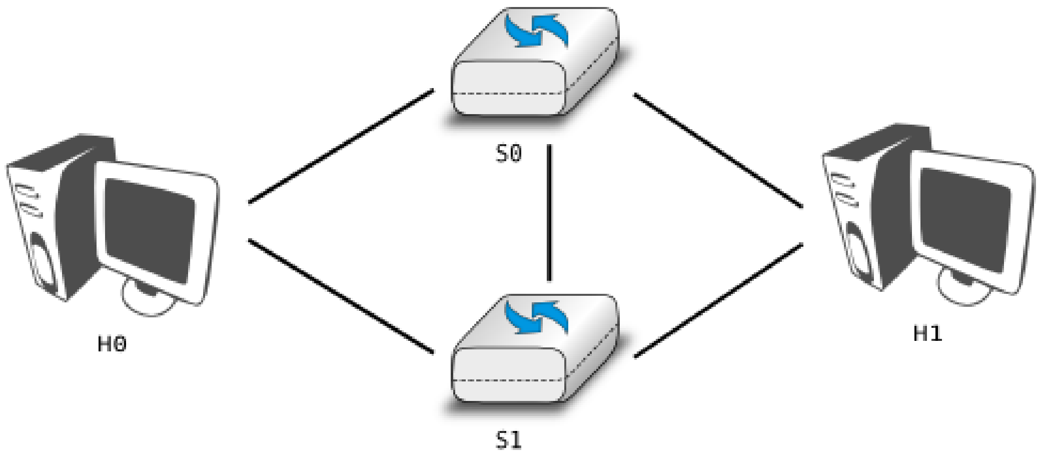


Fig. 21: Fig. 5. Will the link between *S0* and *S1* ever be used?

4.17.2 Testing the Spanning Tree with IPMininet

IPMininet can also be used to configure the Spanning Tree protocol on Linux hosts that act as Ethernet switches. Let us consider the simple Ethernet network shown in the figure below.

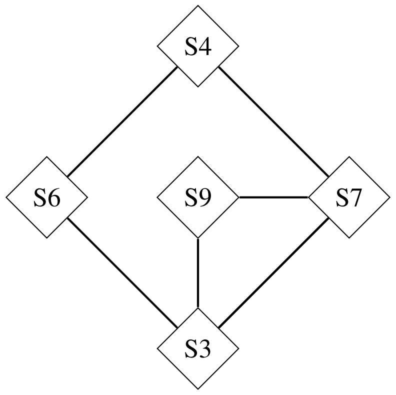


Fig. 22: A simple Ethernet network

This network can be launched with the IPMininet script shown below. The entire script is available from `/exercises/ipmininet_scripts/stp.py`.

```
import shlex
from ipmininet.iptopo import IPTopo

from ipmininet.ipnet import IPNet
from ipmininet.cli import IPCLI


class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        # Switches with manually set STP priority
```

(continues on next page)

(continued from previous page)

```
        s3 = self.addSwitch("s3", prio=3, lo_addresses=["2001:1::4/64"])
        s4 = self.addSwitch("s4", prio=4, lo_addresses=["2001:1::4/64"])
        s6 = self.addSwitch("s6", prio=6, lo_addresses=["2001:1::6/64"])
        s7 = self.addSwitch("s7", prio=7, lo_addresses=["2001:1::7/64"])
        s9 = self.addSwitch("s9", prio=9, lo_addresses=["2001:1::9/64"])

        # Hub
        # hub1 = self.addHub("hub1")

        # Links
        self.addLink(s3, s9, stp_cost=1)  # Cost changed for both interfaces
        l37 = self.addLink(s3, s7)
        l37[s3].addParams(stp_cost=1) # cost changed for s3->s7
        l37[s7].addParams(stp_cost=1) # cost changed for s7->s3
        self.addLink(s9, s7) # default cost of 1
        self.addLink(s6, s9)
        self.addLink(s6, s4)
        self.addLink(s7, s4)

        super(MyTopology, self).build(*args, **kwargs)

    def post_build(self, net):
        for s in self.switches():
            command="/usr/sbin/tcpdump -i any --immediate-mode -c 50 -w ./stp-"+s+"-
↪trace.pcap stp"
            p = net[s].popen(shlex.split(command))

        super(MyTopology, self).post_build(net)


net = IPNet(topo=MyTopology())
try:
    net.start()
    IPCLI(net)
finally:
    net.stop()
```

The addSwitch method creates an Ethernet switch. It assigns a random MAC address to each switch and we can configure it with a priority that is used in the high order bits of the switch identifier. We add one IP address to each switch so that we can connect to them on mininet. In practice, IPMininet configures the *brtcl(8)* software that implements the Spanning Tree protocol on Linux. We can then create the links, configure their cost if required and launch tcpdump to capture the Ethernet frames that contain the messages of the Spanning Tree protocol.

The network contains five nodes and six links.

```
mininet> nodes
available nodes are:
s3 s4 s6 s7 s9
mininet> links
s3-eth2<->s7-eth1 (OK OK)
s3-eth1<->s9-eth1 (OK OK)
s6-eth2<->s4-eth1 (OK OK)
s6-eth1<->s9-eth3 (OK OK)
s7-eth3<->s4-eth2 (OK OK)
s9-eth2<->s7-eth2 (OK OK)
```

By using *brtcl(8)*, we can easily observe the state of the Spanning Tree protocol on the different switches. Let us start with s3, i.e. the root of the Spanning Tree.

```
mininet> s3 brctl showstp s3
s3
  bridge id          0003.f63545ab5f79
  designated root    0003.f63545ab5f79
  root port             0                    path cost                  0
  max age              20.00                 bridge max age            20.00
  hello time            2.00                 bridge hello time          2.00
  forward delay                15.00                 bridge forward delay      15.00
  ageing time                 300.00
  hello timer                   1.03                 tcn timer                  0.00
  topology change timer         0.00                 gc timer                  77.90
  flags


s3-eth1 (1)
  port id            8001                    state                forwarding
  designated root    0003.f63545ab5f79       path cost                  1
  designated bridge  0003.f63545ab5f79       message age timer          0.00
  designated port    8001                    forward delay timer        0.00
  designated cost       0                    hold timer                 0.02
  flags

s3-eth2 (2)
  port id            8002                    state                forwarding
  designated root    0003.f63545ab5f79       path cost                  1
  designated bridge  0003.f63545ab5f79       message age timer          0.00
  designated port    8002                    forward delay timer        0.00
  designated cost       0                    hold timer                 0.02
  flags
```

The first part of the output of the *brctl(8)* command shows the state of the Spanning Tree software on the switch. The identifier of this switch is 0003.f63545ab5f79 and the root switch is itself. There is no root port on this switch since it is the root. The path cost is the cost of the path to reach the root switch, i.e. 0 on the root. Then the switch reports the different timers.

The second part of the output provides the state of each switch port. Port s3-eth1 is active and forwards data frames (state is set to *forwarding*). This port is a *designated* port. The cost of 1 is the cost associated to this interface. The same information is found for port s3-eth2.

The state of switch s9 is different. The output of *brctl(8)* indicates that the root identifier is 0003.f63545ab5f79 which is at a distance of 1 from switch s9. The root port on s9 is port *1*, i.e. s9-eth1. Two of the ports of this switch forward data packets, the root port and the s9-eth3 which is a designated port. The s9-eth2 port is a blocked port.

```
mininet> s9 brctl showstp s9
s9
  bridge id          0009.7ecc45e18e5b
  designated root    0003.f63545ab5f79
  root port             1                    path cost                  1
  max age              20.00                 bridge max age            20.00
  hello time            2.00                 bridge hello time          2.00
  forward delay                15.00                 bridge forward delay      15.00
  ageing time                 300.00
  hello timer                   0.00                 tcn timer                  0.00
  topology change timer         0.00                 gc timer                 167.22
  flags
```

(continues on next page)

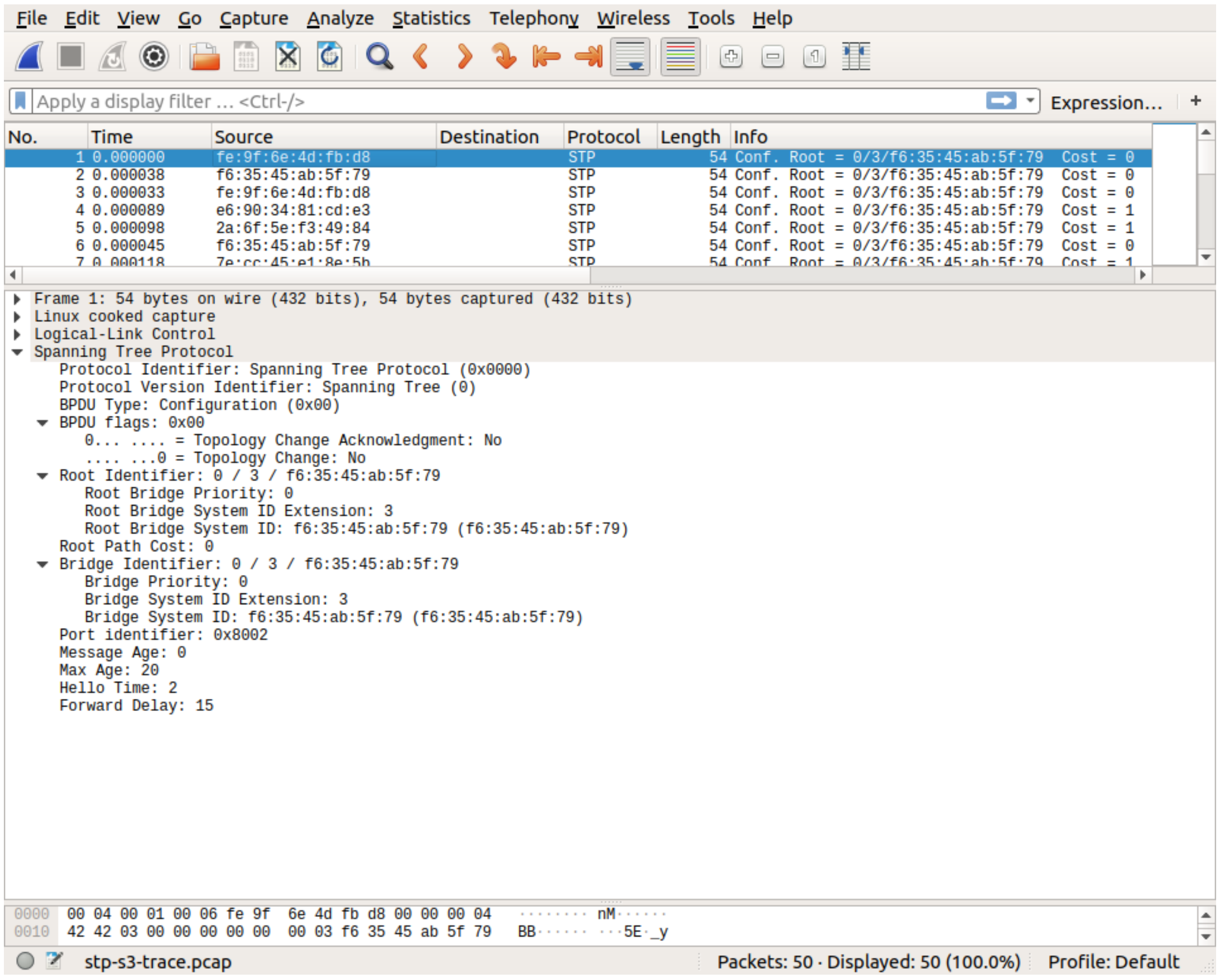
(continued from previous page)

```


s9-eth1 (1)
  port id              8001                     state                forwarding
  designated root      0003.f63545ab5f79        path cost                  1
  designated bridge    0003.f63545ab5f79        message age timer         20.00
  designated port      8001                     forward delay timer        0.00
  designated cost         0                     hold timer                 0.00
  flags

s9-eth2 (2)
  port id              8002                     state                  blocking
  designated root      0003.f63545ab5f79        path cost                  1
  designated bridge    0007.2a6f5ef34984        message age timer         19.98
  designated port      8002                     forward delay timer        0.00
  designated cost         1                     hold timer                 0.00
  flags

s9-eth3 (3)
  port id              8003                     state                forwarding
  designated root      0003.f63545ab5f79        path cost                  1
  designated bridge    0009.7ecc45e18e5b        message age timer          0.00
  designated port      8003                     forward delay timer        0.00
  designated cost         1                     hold timer                 0.97
  flags
```

`brctl(8)` also maintains a MAC address table that contains the Ethernet addresses that have been learned on each switch port.

```
mininet> s9 brctl showmacs s9
port no      mac addr                is local?       ageing timer
1    2a:6f:5e:f3:49:84       no               257.92
1    62:60:d3:46:2f:12       no               257.92
3    7e:cc:45:e1:8e:5b       yes                0.00
3    7e:cc:45:e1:8e:5b       yes                0.00
2    a2:07:cb:02:90:4a       yes                0.00
2    a2:07:cb:02:90:4a       yes                0.00
1    d6:a1:b4:c8:de:72       yes                0.00
1    d6:a1:b4:c8:de:72       yes                0.00
1    f6:35:45:ab:5f:79       no                 0.45
```

Thanks to the traces collected by tcpdump, we can easily analyze the messages exchanged by the switches. Here is the fist message sent by switch `s3`.


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Expression…
+

```
No.  Time      Source             Destination  Protocol  Length Info
   1 0.000000  fe:9f:6e:4d:fb:d8               STP           54 Conf. Root = 0/3/f6:35:45:ab:5f:79  Cost = 0
   2 0.000038  f6:35:45:ab:5f:79               STP           54 Conf. Root = 0/3/f6:35:45:ab:5f:79  Cost = 0
   3 0.000033  fe:9f:6e:4d:fb:d8               STP           54 Conf. Root = 0/3/f6:35:45:ab:5f:79  Cost = 0
   4 0.000089  e6:90:34:81:cd:e3               STP           54 Conf. Root = 0/3/f6:35:45:ab:5f:79  Cost = 1
   5 0.000098  2a:6f:5e:f3:49:84               STP           54 Conf. Root = 0/3/f6:35:45:ab:5f:79  Cost = 1
   6 0.000045  f6:35:45:ab:5f:79               STP           54 Conf. Root = 0/3/f6:35:45:ab:5f:79  Cost = 0
   7 0.000118  7e:cc:45:e1:8e:5b               STP           54 Conf. Root = 0/3/f6:35:45:ab:5f:79  Cost = 1
```

```
▸ Frame 1: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
▸ Linux cooked capture
▸ Logical-Link Control
▾ Spanning Tree Protocol
    Protocol Identifier: Spanning Tree Protocol (0x0000)
    Protocol Version Identifier: Spanning Tree (0)
    BPDU Type: Configuration (0x00)
  ▾ BPDU flags: 0x00
      0... .... = Topology Change Acknowledgment: No
      .... ...0 = Topology Change: No
  ▾ Root Identifier: 0 / 3 / f6:35:45:ab:5f:79
      Root Bridge Priority: 0
      Root Bridge System ID Extension: 3
      Root Bridge System ID: f6:35:45:ab:5f:79 (f6:35:45:ab:5f:79)
    Root Path Cost: 0
  ▾ Bridge Identifier: 0 / 3 / f6:35:45:ab:5f:79
      Bridge Priority: 0
      Bridge System ID Extension: 3
      Bridge System ID: f6:35:45:ab:5f:79 (f6:35:45:ab:5f:79)
    Port identifier: 0x8002
    Message Age: 0
    Max Age: 20
    Hello Time: 2
    Forward Delay: 15
```

```
0000  00 04 00 01 00 06 fe 9f  6e 4d fb d8 00 00 00 04   ········ nM······
0010  42 42 03 00 00 00 00 00  00 03 f6 35 45 ab 5f 79   BB······ ···5E·_y
```

stp-s3-trace.pcap   Packets: 50 · Displayed: 50 (100.0%)   Profile: Default

CHAPTER

FIVE

APPENDICES

5.1 Glossary

address A string of bits that identifies a network interface in the network layer or the datalink layer. Most addresses have a fixed length, e.g. 32 bits for *IPv4*, 128 bits for *IPv6* or 48 bits for *Ethernet* and other related Local Area Networks.

AIMD Additive Increase, Multiplicative Decrease. A rate adaption algorithm used notably by TCP where a host additively increases its transmission rate when the network is not congested and multiplicatively decreases when congested is detected.

anycast a transmission mode where an information is sent from one source to *one* receiver that belongs to a specified group

API Application Programming Interface

ARP The Address Resolution Protocol is a protocol used by IPv4 devices to obtain the datalink layer address that corresponds to an IPv4 address on the local area network. ARP is defined in **RFC 826**

ARPANET The Advanced Research Project Agency (ARPA) Network is a network that was built by network scientists in USA with funding from the ARPA of the US Ministry of Defense. ARPANET is considered as the grandfather of today's Internet.

ASCII The American Standard Code for Information Interchange (ASCII) is a character-encoding scheme that defines a binary representation for characters. The ASCII table contains both printable characters and control characters. ASCII characters were encoded in 7 bits and only contained the characters required to write text in English. Other character sets such as Unicode have been developed later to support all written languages.

ASN.1 The Abstract Syntax Notation One (ASN.1) was designed by ISO and ITU-T. It is a standard and flexible notation that can be used to describe data structures for representing, encoding, transmitting, and decoding data between applications. It was designed to be used in the Presentation layer of the OSI reference model but is now used in other protocols such as *SNMP*.

ATM Asynchronous Transfer Mode

BGP The Border Gateway Protocol is the interdomain routing protocol used in the global Internet.

BNF A Backus-Naur Form (BNF) is a formal way to describe a language by using syntactic and lexical rules. BNFs are frequently used to define programming languages, but also to define the messages exchanged between networked applications. **RFC 5234** explains how a BNF must be written to specify an Internet protocol.

broadcast a transmission mode where is same information is sent to all nodes in the network

CIDR Classless Inter Domain Routing is the current address allocation architecture for IP version 4. It was defined in **RFC 1518** and **RFC 4632**.

dial-up line A synonym for a regular telephone line, i.e. a line that can be used to dial any telephone number.

DNS The Domain Name System is a distributed database that can be queried by hosts to map names onto IP addresses. It is defined in **RFC 1035**

eBGP An eBGP session is a BGP session between two directly connected routers that belong to two different Autonomous Systems. Also called an external BGP session.

EGP Exterior Gateway Protocol. Synonym of interdomain routing protocol

EIGRP The Enhanced Interior Gateway Routing Protocol (EIGRP) is a proprietary intradomain routing protocol that is often used in enterprise networks. EIGRP uses the DUAL algorithm described in [Garcia1993].

Ethernet The most widely used LAN technology.

file transfer A service that enables a user to send or receive a file from a distant server over the network. The File Transfer Protocol *FTP* was a popular service. It has now been replaced by HTTP/HTTPs or more secure protocols such as the SSH File Transfer Protocol.

frame a frame is the unit of information transfer in the datalink layer

Frame-Relay A wide area networking technology using virtual circuits that is deployed by telecom operators.

ftp The File Transfer Protocol defined in **RFC 959** has been the *de facto* protocol to exchange files over the Internet before the widespread adoption of HTTP **RFC 2616**.

FTP The File Transfer Protocol is defined in **RFC 959**

hosts.txt The original file containing the list of all Internet hosts. This file has been deprecated, but Unix variants still maintain a local `/etc/hosts` containing mappings between names and IP addresses. See http://man7.org/linux/man-pages/man5/hosts.5.html for a description of the format of this file on Linux.

HTML The HyperText Markup Language specifies the structure and the syntax of the documents that are exchanged on the world wide web. HTML is maintained by the HTML working group of the *W3C*

HTTP The HyperText Transport Protocol is defined in **RFC 2616**

hub A relay operating in the physical layer.

IANA The Internet Assigned Numbers Authority (IANA) is responsible for the coordination of the DNS Root, IP addressing, and other Internet protocol resources

iBGP An iBGP session is a BGP between two routers belonging to the same Autonomous System. Also called an internal BGP session.

ICANN The Internet Corporation for Assigned Names and Numbers (ICANN) coordinates the allocation of domain names, IP addresses and AS numbers as well protocol parameters. It also coordinates the operation and the evolution of the DNS root name servers.

IETF The Internet Engineering Task Force is a non-profit organization that develops the standards for the protocols used in the Internet. The IETF mainly covers the transport and network layers. Several application layer protocols are also standardized within the IETF. The work in the IETF is organized in working groups. Most of the work is performed by exchanging emails and there are three IETF meetings every year. Participation is open to anyone. See https://www.ietf.org

IGP Interior Gateway Protocol. Synonym of intradomain routing protocol

IGRP The Interior Gateway Routing Protocol (IGRP) is a proprietary intradomain routing protocol that uses distance vector. IGRP supports multiple metrics for each route but has been replaced by *EIGRP*

IMAP The Internet Message Access Protocol (IMAP), defined in **RFC 3501**, is an application-level protocol that allows a client to access and manipulate the emails stored on a server. With IMAP, the email messages remain on the server and are not downloaded on the client.

Internet a public internet, i.e. a network composed of different networks that are running *IPv4* or *IPv6*

internet an internet is an internetwork, i.e. a network composed of different networks. The *Internet*, with a capital *I* corresponds to the global network that we use today, but other internetworks have been used in the path.

inverse query For DNS servers and resolvers, an inverse query is a query for the domain name that corresponds to a given IP address.

IP Internet Protocol is the generic term for the network layer protocol in the TCP/IP protocol suite. IP version 4 is widely used but IP version 6 is being deployed globally.

IPv4 is the version 4 of the Internet Protocol, the connectionless network layer protocol used in most of the Internet today. IPv4 addresses are encoded as a 32 bits field.

IPv6 is the version 6 of the Internet Protocol, the connectionless network layer protocol which is intended to replace IPv4. IP version 6 addresses are encoded as a 128 bits field.

IS-IS Intermediate System- Intermediate System. A link-state intradomain routing that was initially defined for the ISO CLNP protocol but was extended to support IP v4 and IP v6. IS-IS is often used in ISP networks. It is defined in [ISO10589]

ISN The Initial Sequence Number of a TCP connection is the sequence number chosen by the client (resp. server) that is placed in the *SYN* (resp. *SYN+ACK*) segment during the establishment of the TCP connection.

ISO The International Standardization Organization is an agency of the United Nations that is based in Geneva and develop standards on various topics. Within ISO, country representatives vote to approve or reject standards. Most of the work on the development of ISO standards is done in expert working groups. Additional information about ISO may be obtained from https://www.iso.org

ISO-3166 An *ISO* standard that defines codes to represent countries and their subdivisions. See http://www.iso.org/iso/country_codes.htm

ISP An Internet Service Provider, i.e. a network that provides Internet access to its clients.

ITU The International Telecommunication Union is a United Nation's agency whose purpose is to develop standards for the telecommunication industry. It was initially created to standardize the basic telephone system but expanded later towards data networks. The work within ITU is mainly done by network specialists from the telecommunication industry (operators and vendors). See https://www.itu.int for more information

IXP Internet eXchange Point. A location where routers belonging to different domains are attached to the same Local Area Network to establish peering sessions and exchange packets. See http://www.euro-ix.net/ or https://en.wikipedia.org/wiki/List_of_Internet_exchange_points_by_size for a partial list of IXPs.

LAN Local Area Network

leased line A telephone line that is permanently available between two endpoints.

MAN Metropolitan Area Network

MIME The Multipurpose Internet Mail Extensions (MIME) defined in **RFC 2045** are a set of extensions to the format of email messages that allow to use non-ASCII characters inside mail messages. A MIME message can be composed of several different parts each having a different format.

MIME document A MIME document is a document, encoded by using the *MIME* format.

minicomputer A minicomputer is a multi-user system that was typically used in the 1960s/1970s to serve departments. See the corresponding Wikipedia article for additional information : https://en.wikipedia.org/wiki/Minicomputer

modem A modem (modulator-demodulator) is a device that encodes (resp. decodes) digital information by modulating (resp. demodulating) an analog signal. Modems are frequently used to transmit digital information over telephone lines and radio links. See https://en.wikipedia.org/wiki/Modem for a survey of various types of modems

MSS A TCP option used by a TCP entity in SYN segments to indicate the Maximum Segment Size that it is able to receive.

multicast a transmission mode where an information is sent efficiently to *all* the receivers that belong to a given group

nameserver A server that implements the DNS protocol and can answer queries for names inside its own domain.

NAT A Network Address Translator is a middlebox that translates IP packets.

NBMA A Non Broadcast Mode Multiple Access Network is a subnetwork that supports multiple hosts/routers but does not provide an efficient way of sending broadcast frames to all devices attached to the subnetwork. ATM subnetworks are an example of NBMA networks.

network-byte order Internet protocol allow to transport sequences of bytes. These sequences of bytes are sufficient to carry ASCII characters. The network-byte order refers to the Big-Endian encoding for 16 and 32 bits integer. See https://en.wikipedia.org/wiki/Endianness

NFS The Network File System is defined in **RFC 1094**

NTP The Network Time Protocol is defined in **RFC 1305**

OSI Open Systems Interconnection. A set of networking standards developed by *ISO* including the 7 layers OSI reference model.

OSPF Open Shortest Path First. A link-state intradomain routing protocol that is often used in enterprise and ISP networks. OSPF is defined in and **RFC 2328** and **RFC 5340**

packet a packet is the unit of information transfer in the network layer

PBL Problem-based learning is a teaching approach that relies on problems.

POP The Post Office Protocol (POP), defined **RFC 1939**, is an application-level protocol that allows a client to download email messages stored on a server.

remote login A service that enables a user to connect to a distant server over the network. Telnet, defined in **RFC 854** and the BSD rlogin services defined in **RFC 1282** were popular in the past. They have been deprecated for security reasons and are now replaced by *ssh*.

resolver A server that implements the DNS protocol and can resolve queries. A resolver usually serves a set of clients (e.g. all hosts in campus or all clients of a given ISP). It sends DNS queries to nameservers everywhere on behalf of its clients and stores the received answers in its cache. A resolver must know the IP addresses of the root nameservers.

RIP Routing Information Protocol. An intradomain routing protocol based on distance vectors that is sometimes used in enterprise networks. RIP is defined in **RFC 2453**.

RIR Regional Internet Registry. An organization that manages IP addresses and AS numbers on behalf of *IANA*.

root nameserver A name server that is responsible for the root of the domain names hierarchy. There are currently a dozen root nameservers and each DNS resolver See http://www.root-servers.org/ for more information about the operation of these root servers.

round-trip-time The round-trip-time (RTT) is the delay between the transmission of a segment and the reception of the corresponding acknowledgment in a transport protocol.

router A relay operating in the network layer.

RPC Several types of remote procedure calls have been defined. The RPC mechanism defined in **RFC 5531** is used by applications such as NFS

SDU (Service Data Unit) a Service Data Unit is the unit information transferred between applications

segment a segment is the unit of information transfer in the transport layer

SMTP The Simple Mail Transfer Protocol is defined in **RFC 821**

SNMP The Simple Network Management Protocol is a management protocol defined for TCP/IP networks.

socket A low-level API originally defined on Berkeley Unix to allow programmers to develop clients and servers.

spoofed packet A packet is said to be spoofed when the sender of the packet has used as source address a different address than its own.

ssh The Secure Shell (SSH) Transport Layer Protocol is defined in **RFC 4253**

standard query For DNS servers and resolvers, a standard query is a query for a *A* or a *AAAA* record. Such a query typically returns an IP address.

switch A relay operating in the datalink layer.

SYN cookie The SYN cookies is a technique used to compute the initial sequence number (ISN)

TCB The Transmission Control Block is the set of variables that are maintained for each established TCP connection by a TCP implementation.

TCP The Transmission Control Protocol is a protocol of the transport layer in the TCP/IP protocol suite that provides a reliable bytestream connection-oriented service on top of IP

TCP/IP refers to the *TCP* and *IP* protocols

telnet The telnet protocol is defined in **RFC 854**

TLD A Top-level domain name. There are two types of TLDs. The ccTLD are the TLD that correspond to a two letters *ISO-3166* country code. The gTLD are the generic TLDs that are not assigned to a country.

TLS Transport Layer Security, defined in **RFC 5246** is a cryptographic protocol that is used to provide communication security for Internet applications. This protocol is used on top of the transport service but a detailed description is outside the scope of this book.

UDP User Datagram Protocol is a protocol of the transport layer in the TCP/IP protocol suite that provides an unreliable connectionless service that includes a mechanism to detect corruption

unicast a transmission mode where an information is sent from one source to one recipient

VNC

vnc A networked application that allows to remotely access a computer's Graphical User Interface. See https://en.wikipedia.org/wiki/Virtual_Network_Computing

W3C The world wide web consortium was created to standardize the protocols and mechanisms used in the world wide web. It is thus focused on a subset of the application layer. See https://www.w3.org

WAN Wide Area Network

X.25 A wide area networking technology using virtual circuits that was deployed by telecommunication operators.

X11 The XWindow system and the associated protocols are defined in [SG1990]

XML The eXtensible Markup Language (XML) is a flexible text format derived from SGML. It was originally designed for the electronic publishing industry but is now used by a wide variety of applications that need to exchange structured data. The XML specifications are maintained by several working groups of the *W3C*

5.2 Bibliography

Whenever possible, the bibliography includes stable hypertext links to the references cited.

5.3 Indices and tables

- genindex
- search

BIBLIOGRAPHY

[IEEE802.11] LAN/MAN Standards Committee of the IEEE Computer Society. IEEE Standard for Information Technology - Telecommunications and information exchange between systems - local and metropolitan area networks - specific requirements - Part 11 : Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE, 1999.

[IEEE802.1d] LAN/MAN Standards Committee of the IEEE Computer Society, IEEE Standard for Local and metropolitan area networks Media Access Control (MAC) Bridges , IEEE Std 802.1DTM-2004, 2004,

[IEEE802.1q] LAN/MAN Standards Committee of the IEEE Computer Society, IEEE Standard for Local and metropolitan area networks— Virtual Bridged Local Area Networks, 2005,

[IEEE802.2] IEEE 802.2-1998 (ISO/IEC 8802-2:1998), IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements–Part 2: Logical Link Control. Available from http://standards.ieee.org/getieee802/802.2.html

[IEEE802.3] LAN/MAN Standards Committee of the IEEE Computer Society. IEEE Standard for Information Technology - Telecommunications and information exchange between systems - local and metropolitan area networks - specific requirements - Part 3 : Carrier Sense multiple access with collision detection (CSMA/CD) access method and physical layer specification. IEEE, 2000. Available from http://standards.ieee.org/getieee802/802.3.html

[IEEE802.5] LAN/MAN Standards Committee of the IEEE Computer Society. IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements–Part 5: Token Ring Access Method and Physical Layer Specification. IEEE, 1998. available from http://standards.ieee.org/getieee802

[IEEE802] IEEE, Std 802-2001 : IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture, Available from http://standards.ieee.org/getieee802/download/802-2001.pdf

[ACO+2006] Augustin, B., Cuvellier, X., Orgogozo, B., Viger, F., Friedman, T., Latapy, M., Magnien, C., Teixeira, R., Avoiding traceroute anomalies with Paris traceroute, Internet Measurement Conference, October 2006, See also http://www.paris-traceroute.net/

[AM2019] Anderson, B. and McGrew, D., 2019, October. TLS Beyond the Browser: Combining End Host and Network Data to Understand Application Behavior. In Proceedings of the Internet Measurement Conference (pp. 379-392). ACM.

[AS2004] Androutsellis-Theotokis, S. and Spinellis, D.. .. 2004. A survey of peer-to-peer content distribution technologies. ACM Computing Surveys 36, 4 (December 2004), 335-371.

[ATLAS2009] Labovitz, C., Iekel-Johnson, S., McPherson, D., Oberheide, J. and Jahanian, F., Internet inter-domain traffic. In Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM (SIGCOMM '10). ACM, New York, NY, USA, 75-86.

[AW05] Arlitt, M. and Williamson, C. 2005. An analysis of TCP reset behaviour on the internet. SIGCOMM Computer Communication Review 35, 1 (Jan. 2005), 37-44.

[Abramson1970] Abramson, N., THE ALOHA SYSTEM: another alternative for computer communications. In Proceedings of the November 17-19, 1970, Fall Joint Computer Conference (Houston, Texas, November 17 - 19, 1970). AFIPS '70 (Fall). ACM, New York, NY, 281-285.

[B1989] Berners-Lee, T., Information Management: A Proposal, March 1989

[BE2007] Biondi, P. and A. Ebalard, IPv6 Routing Header Security, CanSecWest Security Conference 2007, April 2007.

[BF1995] Bonomi, F. and Fendick, K.W., The rate-based flow control framework for the available bit rate ATM service, IEEE Network, Mar/Apr 1995, Volume: 9, Issue: 2, pages : 25-39

[BMO2006] Bhatia, M., Manral, V., Ohara, Y., IS-IS and OSPF Difference Discussions, work in progress, Internet draft, Jan. 2006

[BNT1997] Beech, W., Nielsen, D., Taylor, J., AX.25 Link Access Protocol for Amateur Packet Radio, version 2.2, Revision: July 1998

[BOP1994] Brakmo, L. S., O'Malley, S. W., and Peterson, L. L., TCP Vegas: new techniques for congestion detection and avoidance. In Proceedings of the Conference on Communications Architectures, Protocols and Applications (London, United Kingdom, August 31 - September 02, 1994). SIGCOMM '94. ACM, New York, NY, 24-35.

[BH2013] Bormann, C., Hoffman, P., Concise Binary Object Representation (CBOR), RFC7049 2013. See also https://cbor.io

[BS2005] D. Barrett, R. Silverman, R. Byrnes, SSH: The Secure Shell (The Definitive Guide), O'Reilly 2005 (2nd edition).

[Bush1945] Bush, V. As we may think The Atlantic Monthly 176 (July 1945), pp. 101–108

[Bush1993] Bush, R., FidoNet: technology, tools, and history. Communications ACM 36, 8 (Aug. 1993), 31-35.

[Bux1989] Bux, W., Token-ring local-area networks and their performance, Proceedings of the IEEE, Vol 77, No 2, p. 238-259, Feb. 1989

[BYL2008] Buford, J., Yu, H., Lua, E.K., P2P Networking and Applications, Morgan Kaufmann, 2008

[CCB+2013] Cardwell, N., Cheng, Y., Brakmo, L., Mathis, M., Raghavan, B., Dukkipati, N., Chu, H., Terzis, A., and Herbert, T., packetdrill: scriptable network stack testing, from sockets to packets. In Proceedings of the 2013 USENIX conference on Annual Technical Conference (USENIX ATC'13). USENIX Association, Berkeley, CA, USA, 213-218.

[CCG+2016] Cardwell, N., Cheng, Y., Gunn, S., Hassas Yeganeh, S. and Jacobson, J., BBR: Congestion-Based Congestion Control. Queue 14, 5, Pages 50 (October 2016)

[CD2008] Calvert, K., Donahoo, M., TCP/IP Sockets in Java : Practical Guide for Programmers, Morgan Kaufman, 2008

[CJ1989] Chiu, D., Jain, R., Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks, Computer Networks and ISDN Systems Vol 17, pp 1-14, 1989

[CNPI09] Gont, F., Security Assessment of the Transmission Control Protocol (TCP),Security Assessment of the Transmission Control Protocol (TCP), Internet draft, work in progress, Jan. 2011

[COZ2008] Chi, Y., Oliveira, R., Zhang, L., Cyclops: The Internet AS-level Observatory, ACM SIGCOMM Computer Communication Review (CCR), October 2008

[CSP2009] Carr, B., Sury, O., Palet Martinez, J., Davidson, A., Evans, R., Yilmaz, F., Wijte, Y., IPv6 Address Allocation and Assignment Policy, RIPE document ripe-481, September 2009

[Clark88] Clark D., The Design Philosophy of the DARPA Internet Protocols, Computer Communications Review 18:4, August 1988, pp. 106-114

[Comer1988] Comer, D., *Internetworking with TCP/IP : principles, protocols & architecture*, Prentice Hall, 1988

[Cohen1980] Cohen, D., *On Holy Wars and a Plea for Peace*, IEN 137, April 1980, http://www.ietf.org/rfc/ien/ien137.txt

[CWE444] CWE-444, Inconsistent Interpretation of HTTP Requests ('HTTP Request Smuggling'), https://cwe.mitre.org/data/definitions/444.html

[DC2009] Donahoo, M., Calvert, K., TCP/IP Sockets in C: Practical Guide for Programmers , Morgan Kaufman, 2009

[DH1976] Diffie, W., Hellman, M., *New directions in cryptography*, in Information Theory, IEEE Transactions on , vol.22, no.6, pp.644-654, Nov 1976, https://dx.doi.org/10.1109/TIT.1976.1055638

[DIX] Digital, Intel, Xerox, The Ethernet: a local area network: data link layer and physical layer specifications. SIGCOMM Computer Communication Review 11, 3 (Jul. 1981), 20-66.

[DKF+2007] Dimitropoulos, X., Krioukov, D., Fomenkov, M., Huffaker, B., Hyun, Y., Claffy, K., Riley, G., AS Relationships: Inference and Validation, ACM SIGCOMM Computer Communication Review (CCR), Jan. 2007

[DP1981] Dalal, Y. K. and Printis, R. S., 48-bit absolute internet and Ethernet host numbers. In Proceedings of the Seventh Symposium on Data Communications (Mexico City, Mexico, October 27 - 29, 1981). SIGCOMM '81. ACM, New York, NY, 240-245.

[DRC+2010] Dukkipati, N., Refice, T., Cheng, Y., Chu, J., Herbert, T., Agarwal, A., Jain, A., Sutin, N., An Argument for Increasing TCP's Initial Congestion Window, ACM SIGCOMM Computer Communications Review, vol. 40 (2010), pp. 27-33

[Dubuisson2000] O. Dubuisson, *ASN.1 : Communication between Heterogeneous Systems <http://www.oss.com/asn1/resources/books-whitepapers-pubs/asn1-books.html#dubuisson>*, Morgan Kauffman, 2000

[DR2002] Daemen, J., Rijmen, V., The Design of Rijndael: AES – The Advanced Encryption Standard Springer, 2002. ISBN 3-540-42580-2.

[DYGU2004] Davik, F. Yilmaz, M. Gjessing, S. Uzun, N., IEEE 802.17 resilient packet ring tutorial, IEEE Communications Magazine, Mar 2004, Vol 42, N 3, p. 112-118

[Dijkstra1959] Dijkstra, E., A Note on Two Problems in Connection with Graphs. Numerische Mathematik, 1:269-271, 1959

[Wikipedia:Dijkstra] Wikipedia, Dijkstra's algorithm

[eTLS2018] ETSI, TS 103 533-3, v 1.1.1, CYBER: Middlebox Security Protocol; Part 3: Profile for enterprise network and data center access control, Oct. 2018

[Fletcher1982] Fletcher, J., An Arithmetic Checksum for Serial Transmissions, Communications, IEEE Transactions on, Jan. 1982, Vol. 30, N. 1, pp. 247-252

[FFEB2005] Francois, P., Filsfils, C., Evans, J., and Bonaventure, O., Achieving sub-second IGP convergence in large IP networks. SIGCOMM Computer Communication Review 35, 3 (Jul. 2005), 35-44.

[FJ1993] Sally Floyd and Van Jacobson. 1993. Random early detection gateways for congestion avoidance. IEEE/ACM Transactions Networking 1, 4 (August 1993), 397-413.

[FJ1994] Floyd, S., and Jacobson, V., The Synchronization of Periodic Routing Messages, IEEE/ACM Transactions on Networking, V.2 N.2, p. 122-136, April 1994

[FKC1996] Freier, A., Karlton, P., Kocher, C., *The SSL Protocol Version 3.0*, Internet draft, November 1996, https://tools.ietf.org/html/draft-ietf-tls-ssl-version3-00

[FRT2002] Fortz, B. Rexford, J. ,Thorup, M., Traffic engineering with traditional IP routing protocols, IEEE Communication Magazine, October 2002

[FTY99] Theodore Faber, Joe Touch, and Wei Yue, The TIME-WAIT state in TCP and Its Effect on Busy Servers, Proceedings INFOCOM'99, pp. 1573

[Feldmeier95] Feldmeier, D. C., Fast software implementation of error detection codes. IEEE/ACM Transactions Networking 3, 6 (Dec. 1995), 640-651.

[GAVE1999] Govindan, R., Alaettinoglu, C., Varadhan, K., Estrin, D., An Architecture for Stable, Analyzable Internet Routing, IEEE Network Magazine, Vol. 13, No. 1, pp. 29–35, January 1999

[GC2000] Grier, D., Campbell, M., A social history of Bitnet and Listserv, 1985-1991, Annals of the History of Computing, IEEE, Volume 22, Issue 2, Apr-Jun 2000, pp. 32 - 41

[Genilloud1990] Genilloud, G., X.400 MHS: first steps towards an EDI communication standard. SIGCOMM Computer Communication Review 20, 2 (Apr. 1990), 72-86.

[Greenwald2014] G. Greenwald, No Place to Hide: Edward Snowden, the NSA, and the U.S. Surveillance State, Metropolitan books, 2014

[GGR2001] Gao, L., Griffin, T., Rexford, J., Inherently safe backup routing with BGP, Proceedings IEEE INFOCOM, April 2001

[GN2011] Gettys, J., Nichols, K., Bufferbloat: dark buffers in the internet. Communications of the ACM 55, no. 1 (2012): 57-65.

[GR2001] Gao, L., Rexford, J., Stable Internet routing without global coordination, IEEE/ACM Transactions on Networking, December 2001, pp. 681-692

[GSW2002] Griffin, T. G., Shepherd, F. B., and Wilfong, G., The stable paths problem and interdomain routing. IEEE/ACM Transactions Networking 10, 2 (Apr. 2002), 232-243

[GW1999] Griffin, T. G. and Wilfong, G., An analysis of BGP convergence properties. SIGCOMM Computer Communication Review 29, 4 (Oct. 1999), 277-288.

[Garcia1993] Garcia-Lunes-Aceves, J., Loop-Free Routing Using Diffusing Computations, IEEE/ACM Transactions on Networking, Vol. 1, No, 1, Feb. 1993

[Gast2002] Gast, M., 802.11 Wireless Networks : The Definitive Guide, O'Reilly, 2002

[Gill2004] Gill, V. , Lack of Priority Queuing Considered Harmful, ACM Queue, December 2004

[Goralski2009] Goralski, W., The Illustrated network : How TCP/IP works in a modern network, Morgan Kaufmann, 2009

[Helme2019] Helme, S., Top 1 Million Analysis - September 2019, 2019, https://scotthelme.co.uk/top-1-million-analysis-september-2019/

[HRX2008] Ha, S., Rhee, I., and Xu, L., CUBIC: a new TCP-friendly high-speed TCP variant. SIGOPS Operating Systems Review 42, 5 (Jul. 2008), 64-74.

[HV2008] Hogg, S. Vyncke, E., IPv6 Security, Cisco Press, 2008

[IMHM2013] Ishihara, K., Mukai, M., Hiromi, R., Mawatari, M., Packet Filter and Route Filter Recommendation for IPv6 at xSP routers, 2013

[ISO10589] ISO, Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473) , 2002

[Jacobson1988] Jacobson, V., Congestion avoidance and control. In Symposium Proceedings on Communications Architectures and Protocols (Stanford, California, United States, August 16 - 18, 1988). V. Cerf, Ed. SIGCOMM '88. ACM, New York, NY, 314-329.

[Jain1990] Jain, R., Congestion control in computer networks : Issues and trends, IEEE Network Magazine, May 1990, pp. 24-30

[JSBM2002] Jung, J., Sit, E., Balakrishnan, H., and Morris, R. 2002. DNS performance and the effectiveness of caching. IEEE/ACM Transactions Networking 10, 5 (Oct. 2002), 589-603.

[JSON-RPC2] JSON-RPC Working group, JSON-RPC 2.0 Specification, available on http://www.jsonrpc.org, 2010

[Kerrisk2010] Kerrisk, M., The Linux Programming Interface, No Starch Press, 2010

[KM1995] Kent, C. A. and Mogul, J. C., Fragmentation considered harmful. SIGCOMM Computer Communication Review 25, 1 (Jan. 1995), 75-87.

[KMS2017] Krombholz, K., Mayer, W., Schmiedecker, M. and Weippl, E., 2017. " I Have No Idea What I'm Doing" - On the Usability of Deploying HTTPS. In 26th {USENIX} Security Symposium ({USENIX} Security 17) (pp. 1339-1356).

[KNT2013] Kühlewind, M., Neuner, S., Trammell, B., On the state of ECN and TCP Options on the Internet. Proceedings of the 14th Passive and Active Measurement conference (PAM 2013), Hong Kong, March 2013

[KP91] Karn, P. and Partridge, C., Improving round-trip time estimates in reliable transport protocols. ACM Transactions Computer Systems 9, 4 (Nov. 1991), 364-373.

[KPD1985] Karn, P., Price, H., Diersing, R., Packet radio in amateur service, IEEE Journal on Selected Areas in Communications, 3, May, 1985

[KPS2003] Kaufman, C., Perlman, R., and Sommerfeld, B. DoS protection for UDP-based protocols. In Proceedings of the 10th ACM Conference on Computer and Communications Security (Washington D.C., USA, October 27 - 30, 2003). CCS '03. ACM, New York, NY, 2-7.

[KPS2002] Kaufman, C., Perlman, R., Speciner, M., Network Security : Private communication in a public world, 2nd edition, Prentice Hall, 2002

[KR1995] Kung, N.T. Morris, R., Credit-based flow control for ATM networks, IEEE Network, Mar/Apr 1995, Volume: 9, Issue: 2, pages: 40-48

[KR2001] Krishnamurthy, B. and Rexford, J., Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement, Addison Wesley, 2001

[KRA2018] Kotzias, P., Razaghpanah, A., Amann, J., Paterson, K.G., Vallina-Rodriguez, N. and Caballero, J., 2018, October. Coming of age: A longitudinal study of TLS deployment. In Proceedings of the Internet Measurement Conference 2018 (pp. 415-428). ACM.

[KT1975] Kleinrock, L., Tobagi, F., Packet Switching in Radio Channels: Part I–Carrier Sense Multiple-Access Modes and their Throughput-Delay Characteristics, IEEE Transactions on Communications, Vol. COM-23, No. 12, pp. 1400-1416, December 1975.

[KW2009] Katz, D., Ward, D., *Bidirectional Forwarding Detection*, **RFC 5880**, June 2010

[KZ1989] Khanna, A. and Zinky, J. 1989. The revised ARPANET routing metric. SIGCOMM Computer Communication Review 19, 4 (Aug. 1989), 45-56.

[KuroseRoss09] Kurose J. and Ross K., Computer networking : a top-down approach featuring the Internet, Addison-Wesley, 2009

[Lamport1981] Lamport, L., Password authentication with insecure communication. Communications ACM 24, 11 (November 1981), 770-772.

[LCP2005] Eng Keong Lua, Crowcroft, J., Pias, M., Sharma, R., Lim, S., A survey and comparison of peer-to-peer overlay network schemes, Communications Surveys & Tutorials, IEEE, Volume: 7 , Issue: 2, 2005, pp. 72-93

[LeB2009] Leroy, D. and O. Bonaventure, Preparing network configurations for IPv6 renumbering, International Journal of Network Management, 2009

[LFJLMT] Leffler, S., Fabry, R., Joy, W., Lapsley, P., Miller, S., Torek, C., An Advanced 4.4BSD Interprocess Communication Tutorial, 4.4 BSD Programmer's Supplementary Documentation

[Leboudec2008] Leboudec, J.-Y., Rate Adaptation Congestion Control and Fairness : a tutorial, Dec. 2008

[MAB2008] McKeown N, Anderson T, Balakrishnan H, Parulkar G, Peterson L, Rexford J, Shenker S, Turner J., OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review. 2008 Mar 31;38(2):69-74.

[MRR1979] McQuillan, J. M., Richer, I., and Rosen, E. C., An overview of the new routing algorithm for the ARPANET. In Proceedings of the Sixth Symposium on Data Communications (Pacific Grove, California, United States, November 27 - 29, 1979). SIGCOMM '79. ACM, New York, NY, 63-68.

[MRR1980] McQuillan, J.M., Richer, I., Rosen, E., The New Routing Algorithm for the ARPANET Communications, IEEE Transactions on , vol.28, no.5, pp.711,719, May 1980

[MSMO1997] Mathis, M., Semke, J., Mahdavi, J., and Ott, T. 1997. The macroscopic behavior of the TCP congestion avoidance algorithm. SIGCOMM Computer Communication Review 27, 3 (Jul. 1997), 67-82.

[MUF+2007] Mühlbauer, W., Uhlig, S., Fu, B., Meulle, M., and Maennel, O., In search for an appropriate granularity to model routing policies. In Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications (Kyoto, Japan, August 27 - 31, 2007). SIGCOMM '07. ACM, New York, NY, 145-156.

[Malkin1999] Malkin, G., RIP: An Intra-Domain Routing Protocol, Addison Wesley, 1999

[Metcalfe1976] Metcalfe R., Boggs, D., Ethernet: Distributed packet-switching for local computer networks. Communications of the ACM, 19(7):395–404, 1976.

[Mills2006] Mills, D.L., Computer Network Time Synchronization: the Network Time Protocol. CRC Press, March 2006, 304 pp.

[Mogul1995] Mogul, J. , The case for persistent-connection HTTP. In Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols For Computer Communication (Cambridge, Massachusetts, United States, August 28 - September 01, 1995). D. Oran, Ed. SIGCOMM '95. ACM, New York, NY, 299-313.

[MoR2004] Modadugu, N. and Rescorla, E., 2004, February. The Design and Implementation of Datagram TLS. In NDSS.

[Moy1998] Moy, J., OSPF: Anatomy of an Internet Routing Protocol, Addison Wesley, 1998

[MVV2011] Menezes, A., van Oorschot, P. and Vanstone, S. , Handbook of Applied Cryptography , CRC Press, 2011

[Myers1998] Myers, B. A., A brief history of human-computer interaction technology. interactions 5, 2 (Mar. 1998), 44-54.

[Nelson1965] Nelson, T. H., Complex information processing: a file structure for the complex, the changing and the indeterminate. In Proceedings of the 1965 20th National Conference (Cleveland, Ohio, United States, August 24 - 26, 1965). L. Winner, Ed. ACM '65. ACM, New York, NY, 84-100.

[NSS2010] Nygren E, Sitaraman RK, Sun J., The Akamai network: a platform for high-performance Internet applications. ACM SIGOPS Operating Systems Review. 2010 Aug 17;44(3):2-19.

[Paxson99] Paxson, V. , End-to-end Internet packet dynamics. SIGCOMM Computer Communication Review 27, 4 (Oct. 1997), 139-152.

[Perlman1985] Perlman, R., An algorithm for distributed computation of a spanning tree in an extended LAN. SIGCOMM Computer Communication Review 15, 4 (September 1985), 44-53.

[Perlman2000] Perlman, R., Interconnections : Bridges, routers, switches and internetworking protocols, 2nd edition, Addison Wesley, 2000

[PHG2013] Prado, A., Harris, N., and Y. Gluck, The BREACH Attack , 2013, <http://breachattack.com/>.

[Rago1993] Rago, S., UNIX System V network programming, Addison Wesley, 1993

[Rescorla2015] Rescorla, E., Stanford Seminar - The TLS 1.3 protocol, Nov. 2015

[Ristic2015] Ristic, I., Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Web Servers and Applications, Feisty Duck, 2015

[RJ1995] Ramakrishnan, K. K. and Jain, R., A binary feedback scheme for congestion avoidance in computer networks with a connectionless network layer. SIGCOMM Computer Communication Review 25, 1 (Jan. 1995), 138-156.

[RSA1978] Rivest, R., Shamir, A. and Adleman, L., A method for obtaining digital signatures and public-key cryptosystems. Communications ACM 21, 2 (February 1978), 120-126

[RY1994] Ramakrishnan, K.K. and Henry Yang, The Ethernet Capture Effect: Analysis and Solution, Proceedings of IEEE 19th Conference on Local Computer Networks, MN, Oct. 1994.

[Roberts1975] Roberts, L., ALOHA packet system with and without slots and capture. SIGCOMM Computer Communication Review 5, 2 (Apr. 1975), 28-42.

[Ross1989] Ross, F., An overview of FDDI: The fiber distributed data interface, IEEE J. Selected Areas in Comm., vol. 7, no. 7, pp. 1043-1051, Sept. 1989

[Russel06] Russell A., Rough Consensus and Running Code and the Internet-OSI Standards War, IEEE Annals of the History of Computing, July-September 2006

[SARK2002] Subramanian, L., Agarwal, S., Rexford, J., Katz, R.. .. Characterizing the Internet hierarchy from multiple vantage points. In IEEE INFOCOM, 2002

[Sechrest] Sechrest, S., An Introductory 4.4BSD Interprocess Communication Tutorial, 4.4BSD Programmer's Supplementary Documentation

[SG1990] Scheifler, R., Gettys, J., X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD, X Version 11, Release 4, Digital Press

[SGP98] Stone, J., Greenwald, M., Partridge, C., and Hughes, J., Performance of checksums and CRC's over real data. IEEE/ACM Transactions Networking 6, 5 (Oct. 1998), 529-543.

[SH1980] Shoch, J. F. and Hupp, J. A., Measured performance of an Ethernet local network. Communications ACM 23, 12 (Dec. 1980), 711-721.

[SH2004] Senapathi, S., Hernandez, R., Introduction to TCP Offload Engines, March 2004

[SMM1998] Semke, J., Mahdavi, J., and Mathis, M., Automatic TCP buffer tuning. SIGCOMM Computer Communication Review 28, 4 (Oct. 1998), 315-323.

[SFR2004] Stevens R. and Fenner, and Rudoff, A., UNIX Network Programming: The sockets networking API, Addison Wesley, 2004

[Sklower89] Sklower, K. 1989. Improving the efficiency of the OSI checksum calculation. SIGCOMM Computer Communication Review 19, 5 (Oct. 1989), 32-43.

[SMASU2012] Sarrar, N., Maier, G., Ager, B., Sommer, R. and Uhlig, S., Investigating IPv6 traffic, Passive and Active Measurements, Lecture Notes in Computer Science vol 7192, 2012, pp.11-20

[Stallings2009] Stallings, W., Protocol Basics: Secure Shell Protocol, Internet Protocol Journal, vol 12, n 4, Dec. 2009

[Stevens1994] Stevens, R., TCP/IP Illustrated : the Protocols, Addison-Wesley, 1994

[Stevens1998] Stevens, R., UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI, Prentice Hall, 1998

[SV1995] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin SIGCOMM Computer Communication Review 25, 4 (October 1995), 231-242.

[TKU2019] Turkovic, B., Kuipers, F., Uhlig, S., Fifty Shades of Congestion Control: A Performance and Interactions Evaluation CoRR abs/1903.03852 (2019)

[Thomborson1992] Thomborson, C., The V.42bis Standard for Data-Compressing Modems, IEEE Micro, September/October 1992 (vol. 12 no. 5), pp. 41-53

[Unicode] The Unicode Consortium. The Unicode Standard, Version 5.0.0, defined by: The Unicode Standard, Version 5.0 (Boston, MA, Addison-Wesley, 2007

[VPD2004] Vasseur, J., Pickavet, M., and Demeester, P., Network Recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS. Morgan Kaufmann Publishers Inc., 2004

[Varghese2005] Varghese, G., Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices, Morgan Kaufmann, 2005

[Vyncke2007] Vyncke, E., Paggen, C., LAN Switch Security: What Hackers Know About Your Switches, Cisco Press, 2007

[WBK2014] Wang XS, Balasubramanian A, Krishnamurthy A, Wetherall D. How Speedy is SPDY ? . In 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14) 2014 (pp. 387-399).

[WMS2004] White, R., Mc Pherson, D., Srihari, S., Practical BGP, Addison-Wesley, 2004

[WF2003] Wessels, D., Fomenkov, M., Wow, That's a lot of packets, Passive and Active Network Measurement Workshop (PAM), Apr 2003

[Williams1993] Williams, R. *A painless guide to CRC error detection algorithms*, August 1993, unpublished manuscript, https://web.archive.org/web/20060101004751/http://www.ross.net/crc/download/crc_v3.txt

[WMSS2019] Ware, R., Mukerjee, M., Seshan, S. and Sherry, J. Modeling BBR's Interactions with Loss-Based Congestion Control. In Proceedings of the Internet Measurement Conference (IMC '19). ACM, New York, NY, USA, 137-143.

[X200] ITU-T, recommendation X.200, Open Systems Interconnection - Model and Notation, 1994

[Ylonen1996] Ylonen, T., SSH — Secure Login Connections over the Internet, Usenix Security 1996

[ZWH2018] Zimmermann T., Wolters B., Hohlfeld O., Wehrle K. Is the web ready for HTTP/2 server push? , Proceedings of the 14th International Conference on emerging Networking Experiments and Technologies 2018 Dec 4 (pp. 13-19). ACM.

[Zimmermann80] Zimmermann, H., OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection, IEEE Transactions on Communications, vol. 28, no. 4, April 1980, pp. 425 - 432.

[Zakon] Zakon, R., Hobbes Internet Timeline, online, https://www.zakon.org/robert/internet/timeline/

[Zhe2017] Zheng, X., Phishing with Unicode Domains, April 14, 2017, https://www.xudongz.com/blog/2017/idn-phishing/

INDEX