Computer Networking

Slide Set 3

Andrew W. Moore
Andrew.Moore@cl.cam.ac.uk

Topic 5 – Transport

Our goals:

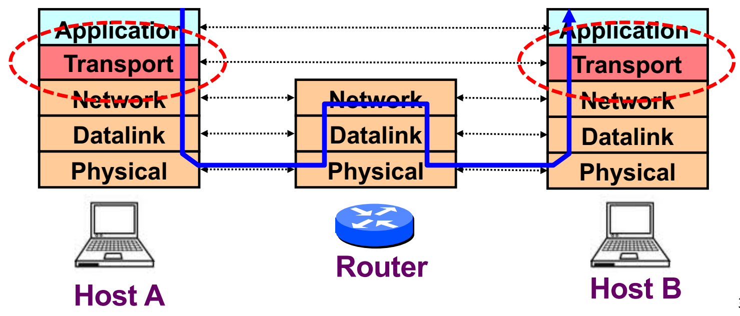
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
  - buffers
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control
  - TCP flow control

2

2

Transport Layer

- Commonly a layer at end-hosts, between the application and network layer



3

3

Why a transport layer?

- IP packets are addressed to a host but end-to-end communication is between application/ processes/tasks at hosts
  - Need a way to decide which packets go to which applications (*more multiplexing*)

4

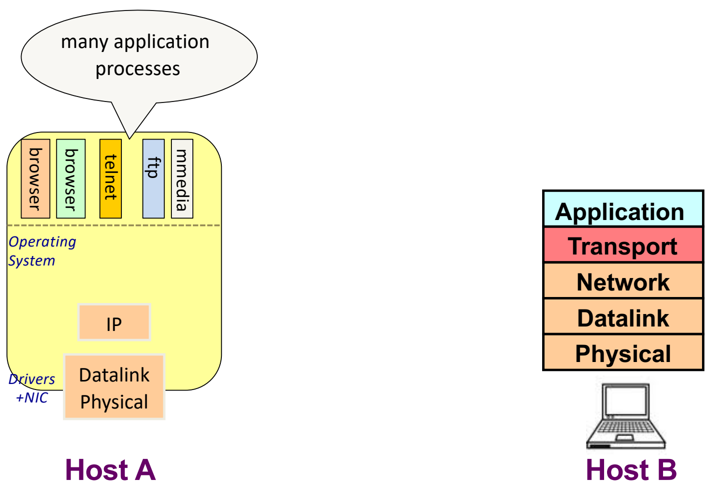
4

Why a transport layer?

5

5

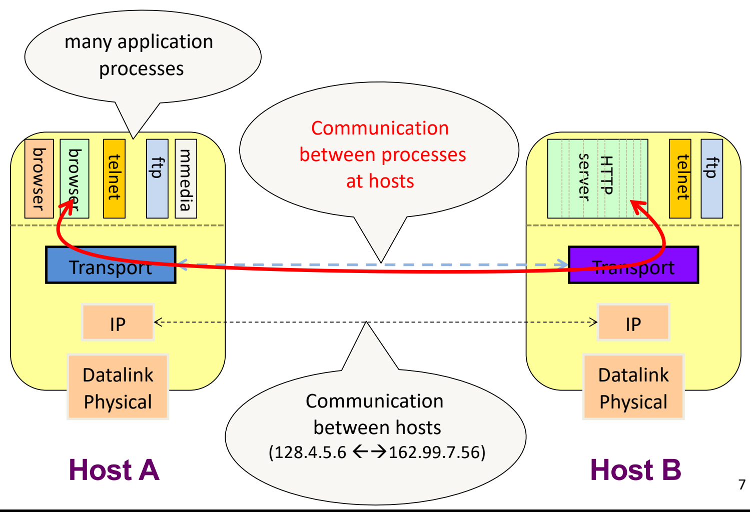
Why a transport layer?



6

6

Why a transport layer?



7

7

Why a transport layer?

- IP packets are addressed to a host but end-to-end communication is between application processes at hosts
  - Need a way to decide which packets go to which applications (mux/demux)
- IP provides a weak service model (*best-effort*)
  - Packets can be corrupted, delayed, dropped, reordered, duplicated
  - No guidance on how much traffic to send and when
  - Dealing with this is tedious for application developers

8

8

Role of the Transport Layer

- Communication between application processes
  - Multiplexing between application processes
  - Implemented using *ports*

9

9

Role of the Transport Layer

- Communication between application processes
- Provide common end-to-end services for app layer [optional]
  - Reliable, in-order data delivery
  - Paced data delivery: flow and congestion-control
    - too fast may overwhelm the network
    - too slow is not efficient

(Just Like Computer Networking Lectures….)

10

10

Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
  - also SCTP, MTCP, SST, RDP, DCCP, …

11

11

Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
- UDP is a minimalist, no-frills transport protocol
  - only provides mux/demux capabilities

12

12

Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
- UDP is a minimalist, no-frills transport protocol
- TCP is the *totus porcus* protocol
  - offers apps a reliable, in-order, byte-stream abstraction
  - with congestion control
  - but **no** performance (delay, bandwidth, ...) guarantees

13

13

Role of the Transport Layer

- Communication between processes
  - mux/demux from and to application processes
  - implemented using ports

14

14

Context: Applications and Sockets

- Socket: software abstraction by which an application process exchanges network messages with the (transport layer in the) operating system
  - socketID = socket(…, socket.TYPE)
  - socketID.sendto(message, …)
  - socketID.recvfrom(…)

- Two important types of sockets
  - UDP socket: TYPE is SOCK_DGRAM
  - TCP socket: TYPE is SOCK_STREAM

15

15

Ports

- Problem: deciding which app (socket) gets which packets

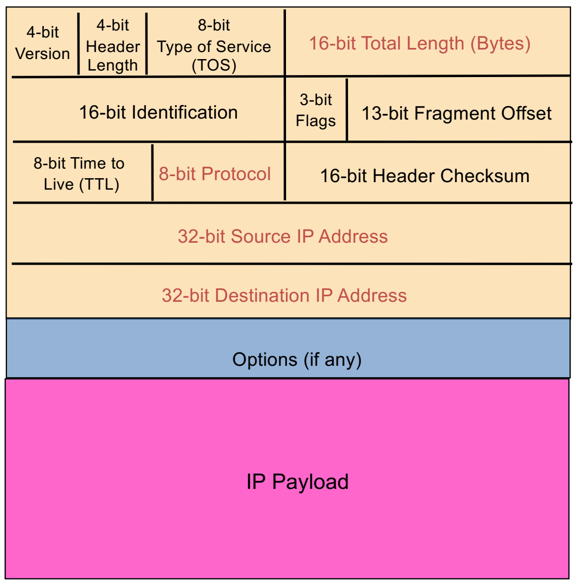
- Solution: ***port*** as a transport layer identifier
  - 16 bit identifier
  - OS stores mapping between sockets and *ports*
  - a packet carries a source and destination port number in its transport layer header

- For UDP ports (SOCK_DGRAM)
  - OS stores (local port, local IP address) ←→ socket

- For TCP ports (SOCK_STREAM)
  - OS stores (local port, local IP, remote port, remote IP) ←→ socket

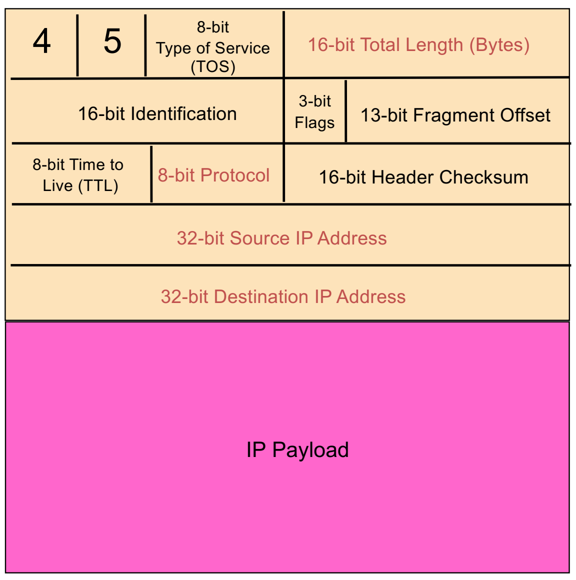
16

16



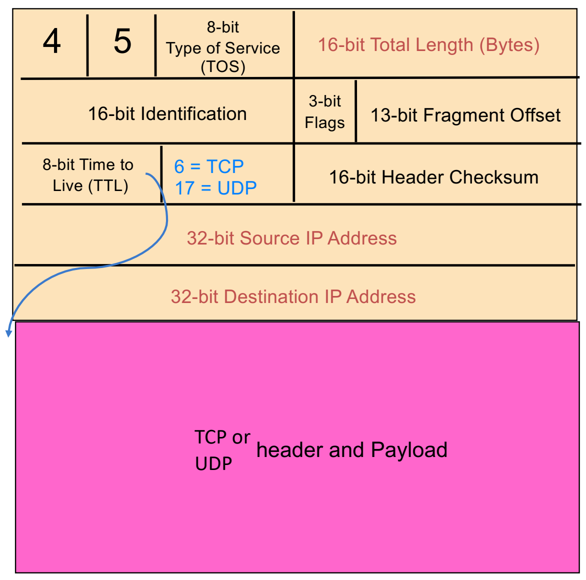
17

17



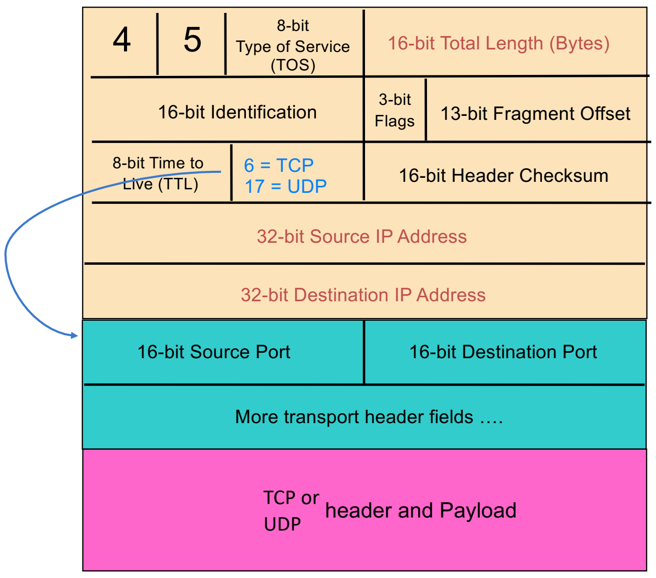
18

18



19

19



20

20

Recap: Multiplexing and Demultiplexing

- Host receives IP packets
  - Each IP header has source and destination IP address
  - Each Transport Layer header has source and destination port number
- Host uses IP addresses and port numbers to direct the message to appropriate socket

21

21

More on Ports

- Separate 16-bit port address space for UDP and TCP
- "Well known" ports (0-1023): everyone agrees which services run on these ports
  - e.g., ssh:22, http:80, https:443
  - helps client know server's port
- Ephemeral ports (most 1024-65535): dynamically selected: as the source port for a client process

22

22

UDP: User Datagram Protocol

- Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
- UDP described in RFC 768 – (1980!)
  - Destination IP address and port to support demultiplexing
  - Optional error checking on the packet contents
    - (checksum field of 0 means "don't verify checksum") ***not in IPv6!***
    - ((this idea of optional checksum is removed in IPv6))

SRC port	DST port
checksum	length
DATA	

23

23

Why a transport layer?

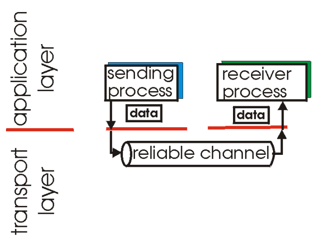
- IP packets are addressed to a host but end-to-end communication is between application processes at hosts
  - Need a way to decide which packets go to which applications (mux/demux)
- IP provides a weak service model (*best-effort*)
  - Packets can be corrupted, delayed, dropped, reordered, duplicated

24

24

Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!



(a) provided service

- In a perfect world, reliable transport is easy

But the Internet default is *best-effort*

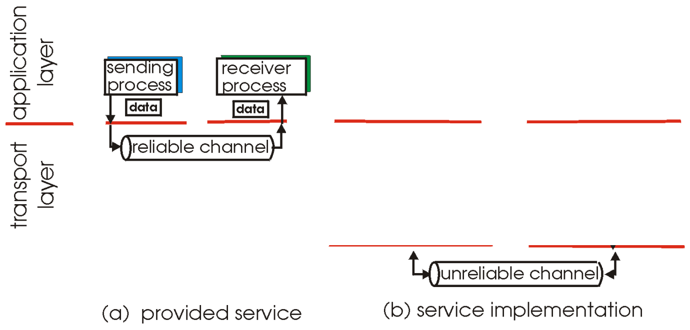
- All the bad things best-effort can do
  - a packet is corrupted (bit errors)
  - a packet is lost
  - a packet is delayed (*why?*)
  - packets are reordered (*why?*)
  - a packet is duplicated (*why?*)

25

25

Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!



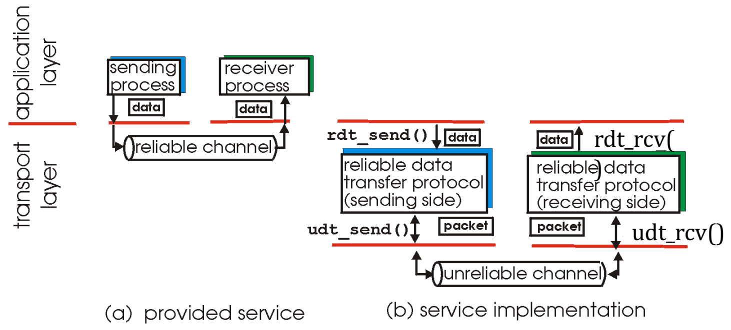
- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

26

26

Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!

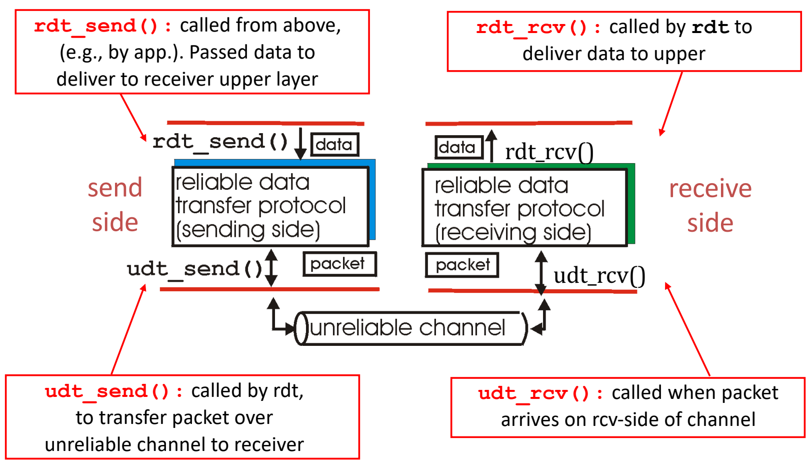


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

27

27

Reliable data transfer: getting started



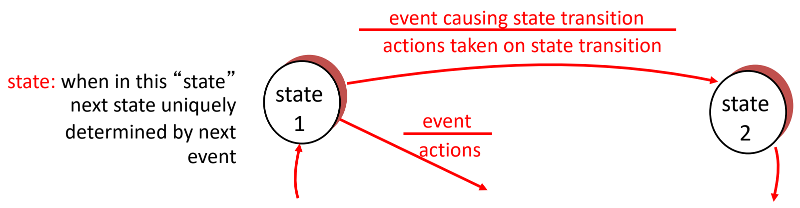
28

28

Reliable data transfer: getting started

We'll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver



29

29

KR state machines – a note.

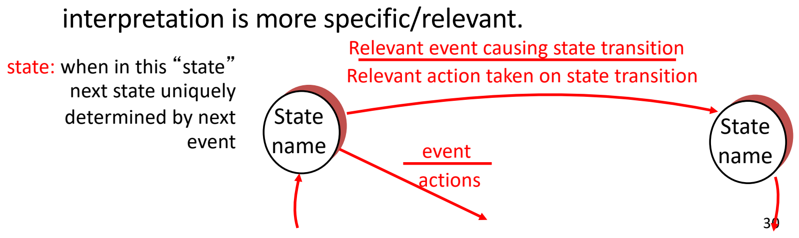
Beware

Kurose and Ross has a confusing/confused attitude to state-machines.

I've attempted to normalise the representation.

UPSHOT: these slides have differing information to the KR book (from which the RDT example is taken.)

in KR "actions taken" appear wide-ranging, my interpretation is more specific/relevant.

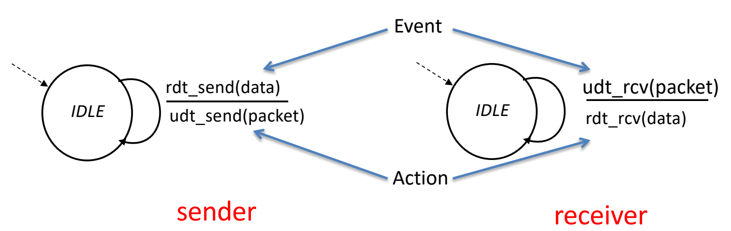


30

30

Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



31

31

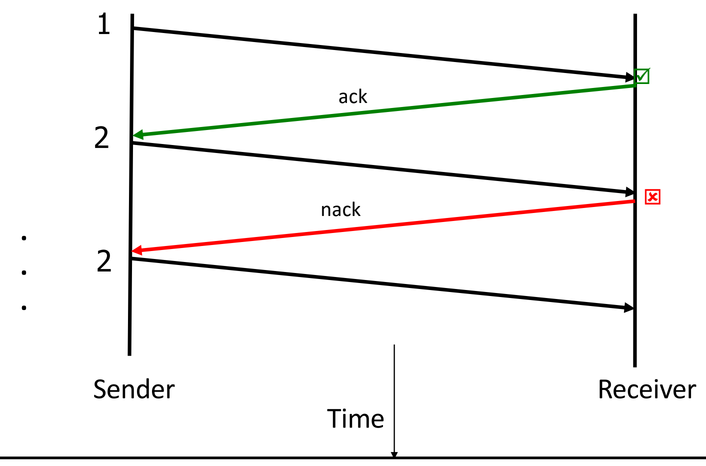
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that packet received is OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that packet had errors
  - sender retransmits packet on receipt of NAK
- new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) receiver->sender

32

32

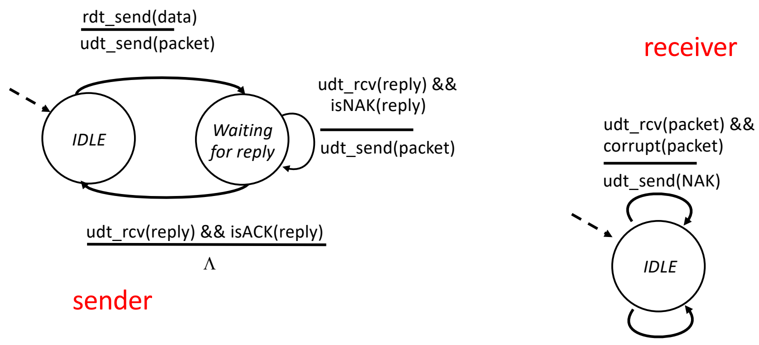
Dealing with Packet Corruption



33

33

rdt2.0: FSM specification

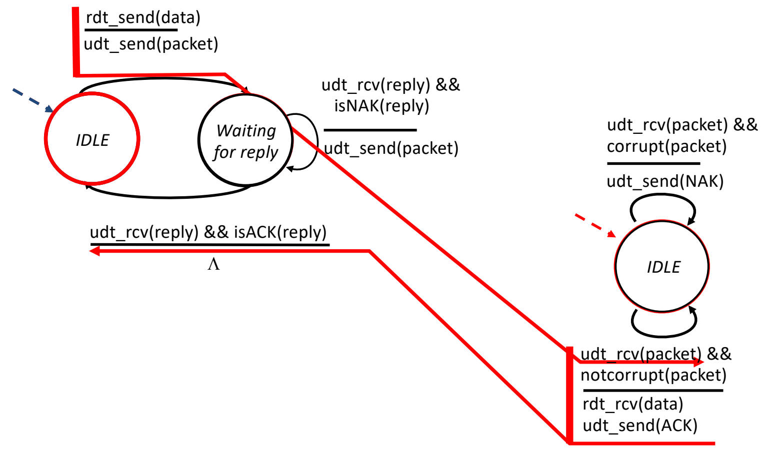


Note: the sender holds a copy of the packet being sent until the delivery is acknowledged.

34

34

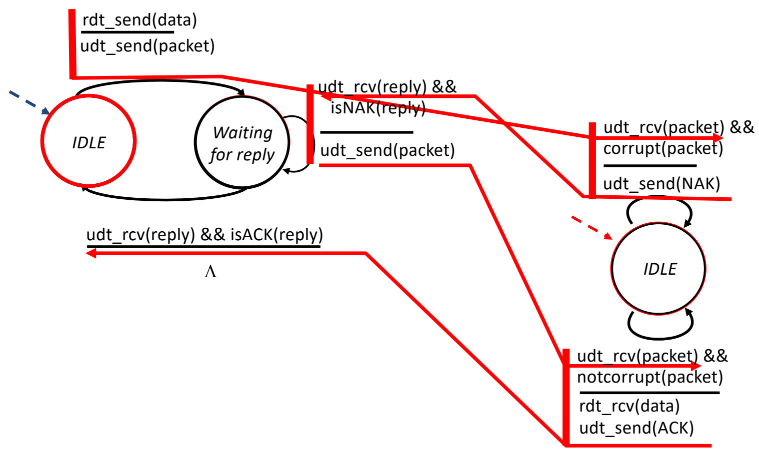
rdt2.0: operation with no errors



35

35

rdt2.0: error scenario



36

36

rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

Handling duplicates:

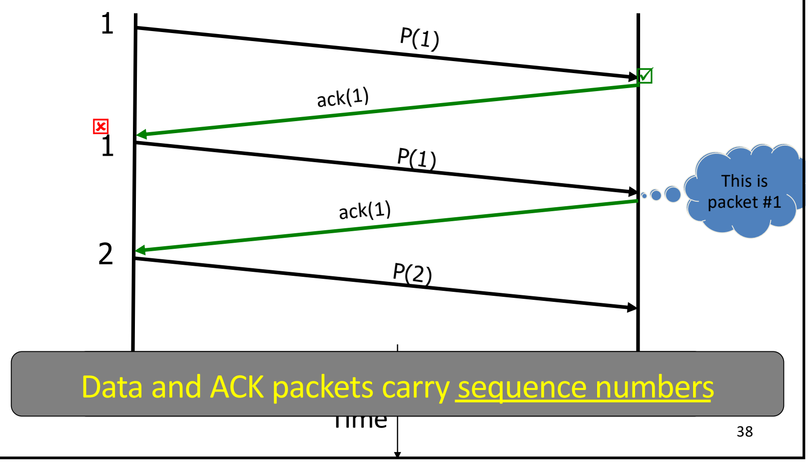
- sender retransmits current packet if ACK/NAK garbled
- sender adds *sequence number* to each packet
- receiver discards (doesn't deliver) duplicate packet

stop and wait
Sender sends one packet, then waits for receiver response

37

37

Dealing with Packet Corruption

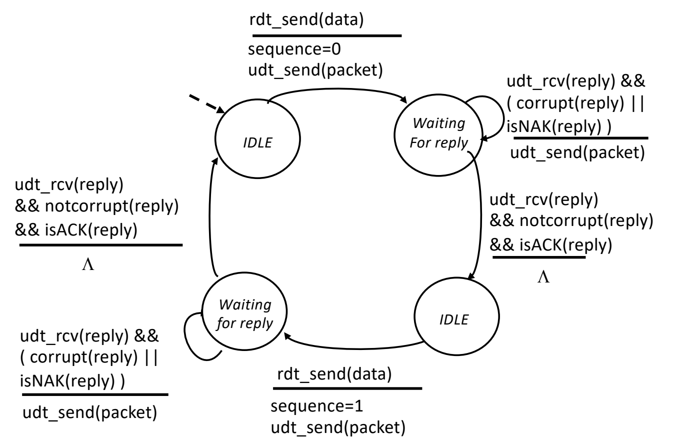


Data and ACK packets carry sequence numbers

38

38

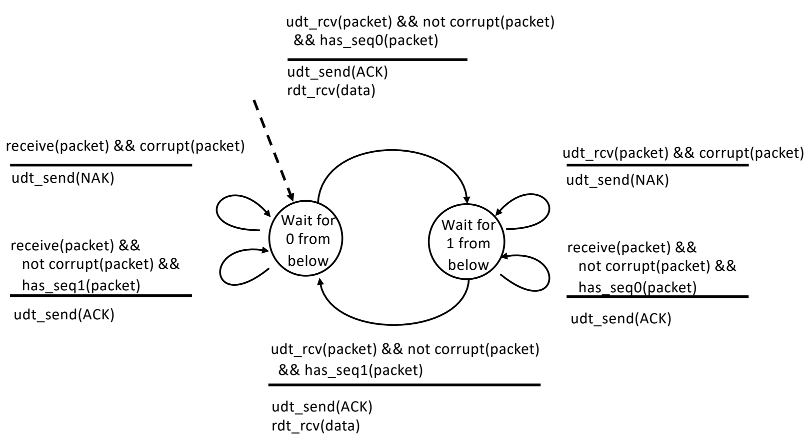
rdt2.1: sender, handles garbled ACK/NAKs



39

39

rdt2.1: receiver, handles garbled ACK/NAKs



40

40

rdt2.1: discussion

Sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "current" pkt has a 0 or 1 sequence number

Receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

41

41

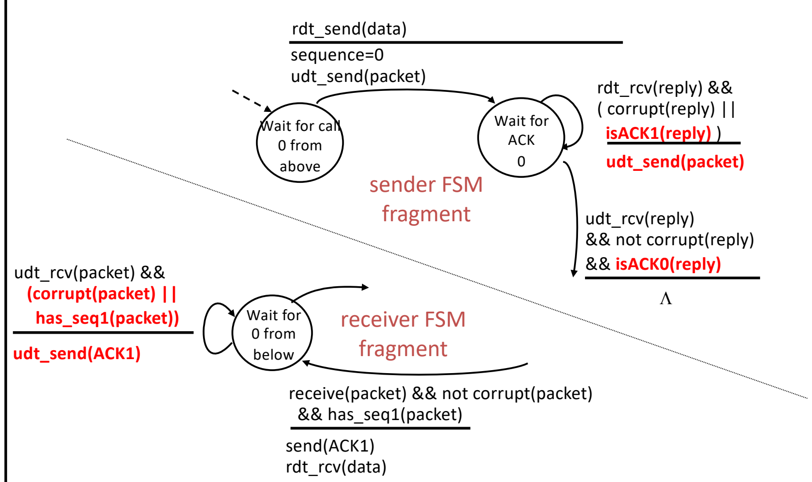
rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

42

42

rdt2.2: sender, receiver fragments



43

43

rdt3.0: channels with errors *and* loss

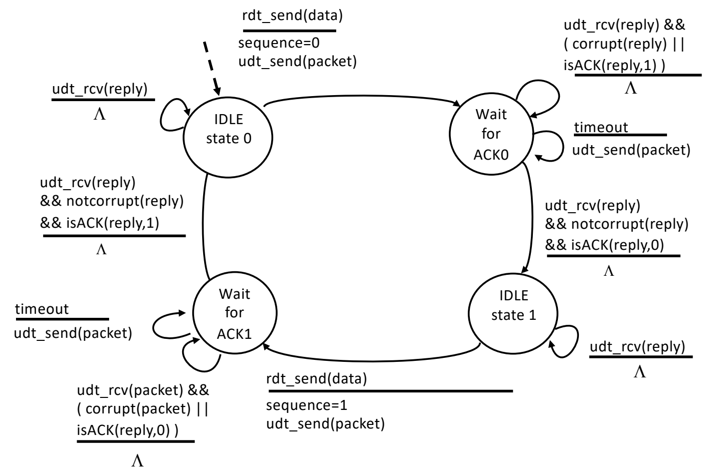
New assumption: underlying channel can also lose packets (data or ACKs)
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Approach: sender waits "reasonable" amount of time for ACK
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

44

44

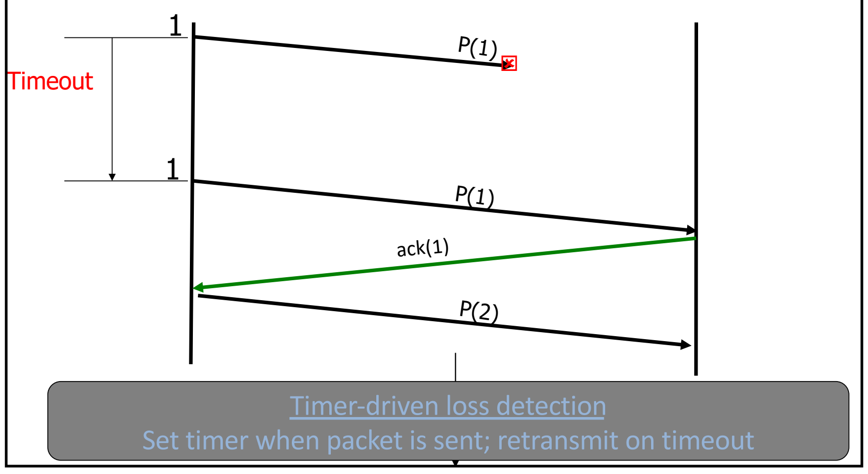
rdt3.0 sender



45

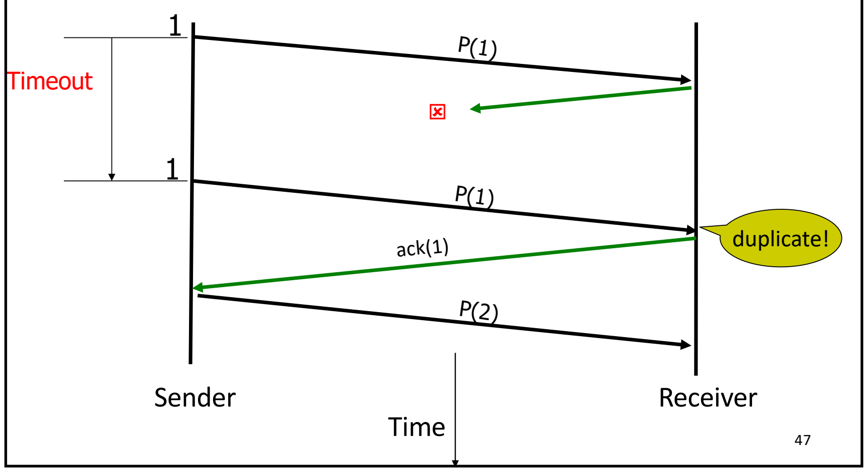
45

Dealing with Packet Loss



46

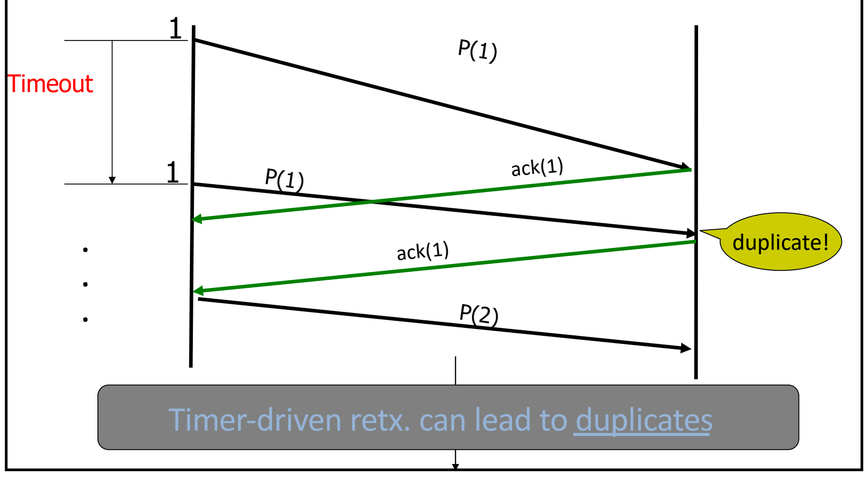
Dealing with Packet Loss



47

47

Dealing with Packet Loss



48

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\text{microseconds}$$

- U $_{sender}$: utilization – fraction of time sender busy sending

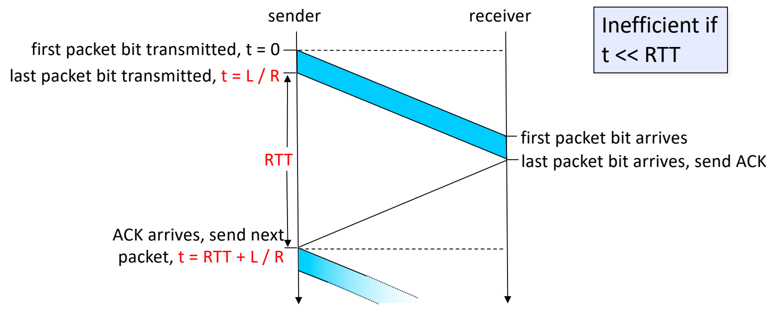
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

- 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!

49

49

rdt3.0: stop-and-wait operation



Inefficient if t << RTT

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

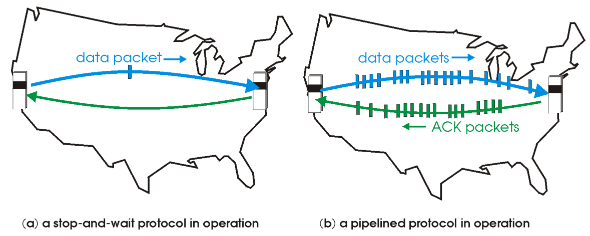
50

50

Pipelined (Packet-Window) protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver



51

51

A Sliding Packet Window

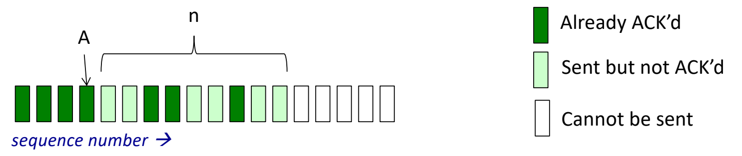
- window = set of adjacent sequence numbers
  - The size of the set is the window size; assume window size is *n*
- General idea: send up to *n* packets at a time
  - Sender can send packets in its window
  - Receiver can accept packets in its window
  - Window of acceptable packets "slides" on successful reception/acknowledgement

52

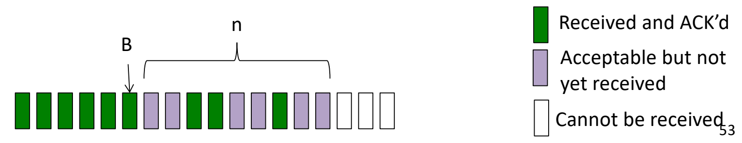
52

A Sliding Packet Window

- Let A be the last ack'd packet of sender without gap; then window of sender = {A+1, A+2, …, A+n}



- Let B be the last received packet without gap by receiver, then window of receiver = {B+1,…, B+n}



53

53

Acknowledgements w/ Sliding Window

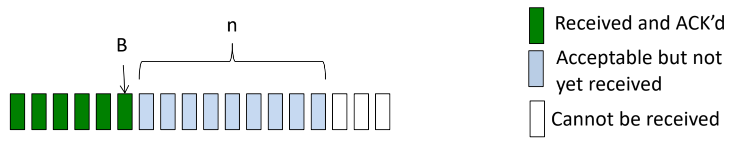
- Two common options
  - cumulative ACKs: ACK carries next in-order sequence number that the receiver expects

54

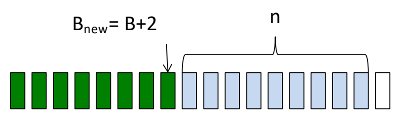
54

Cumulative Acknowledgements (1)

- At receiver



- After receiving B+1, B+2



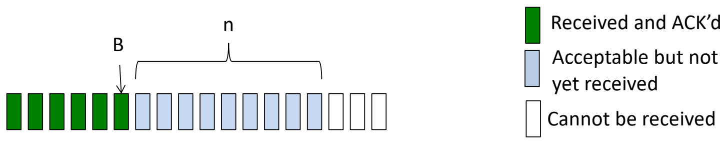
- Receiver sends ACK(B_{new}+1)

55

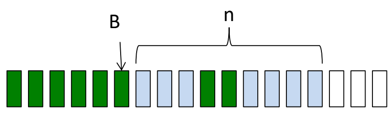
55

Cumulative Acknowledgements (2)

- At receiver



- After receiving B+4, B+5



How do we recover?

- Receiver sends ACK(B+1)

56

56

Go-Back-N (GBN)

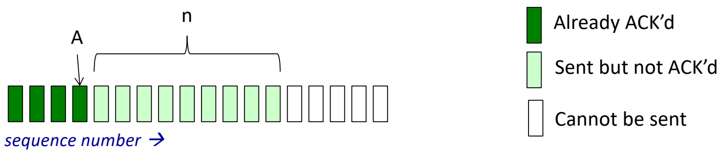
- Sender transmits up to *n* unacknowledged packets
- Receiver only accepts packets in order
  - discards out-of-order packets (i.e., packets other than *B+1*)
- Receiver uses cumulative acknowledgements
  - i.e., sequence# in ACK = next expected in-order sequence#
- Sender sets timer for 1st outstanding ack (A+1)
- If timeout, retransmit *A+1*, … , *A+n*

57

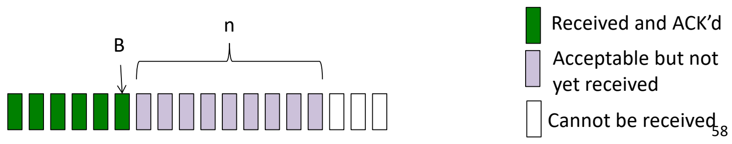
57

Sliding Window with GBN

- Let A be the last ack'd packet of sender without gap; then window of sender = {A+1, A+2, …, A+n}



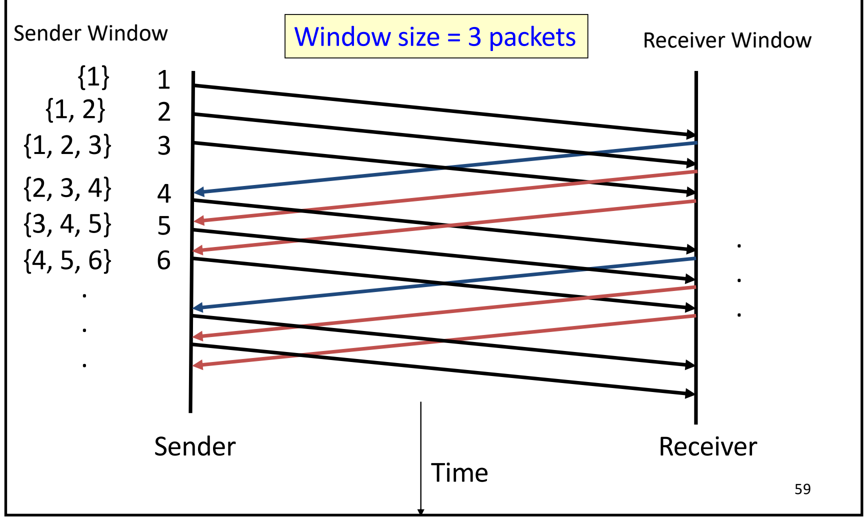
- Let B be the last received packet without gap by receiver, then window of receiver = {B+1,…, B+n}



58

58

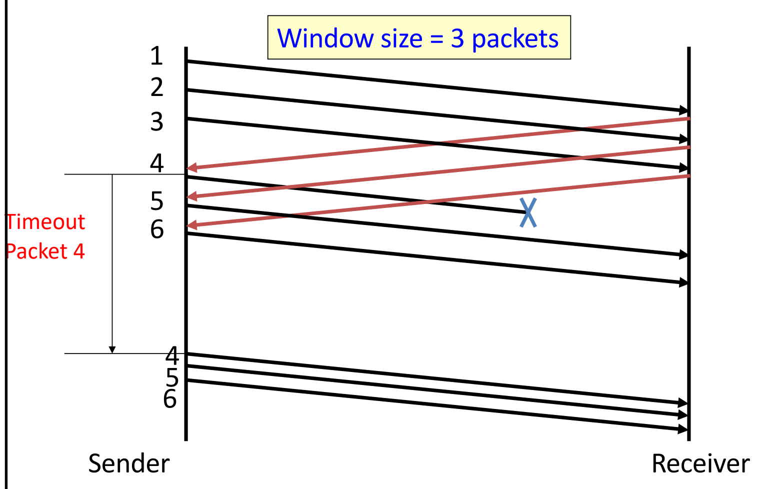
GBN Example w/o Errors



59

59

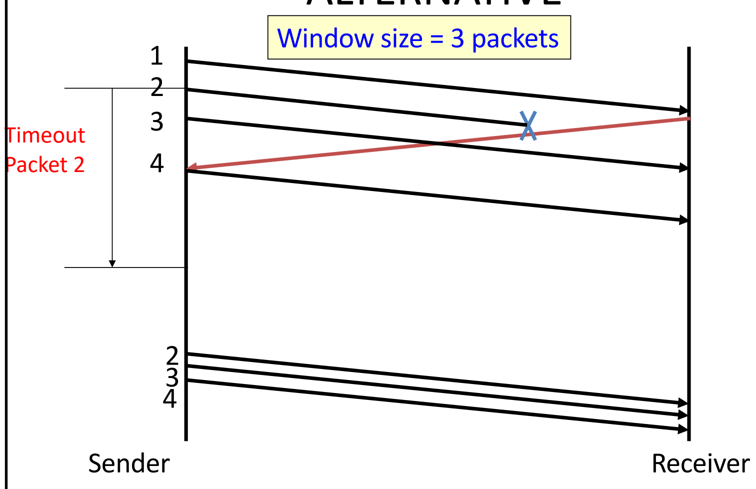
GBN Example with Errors



60

60

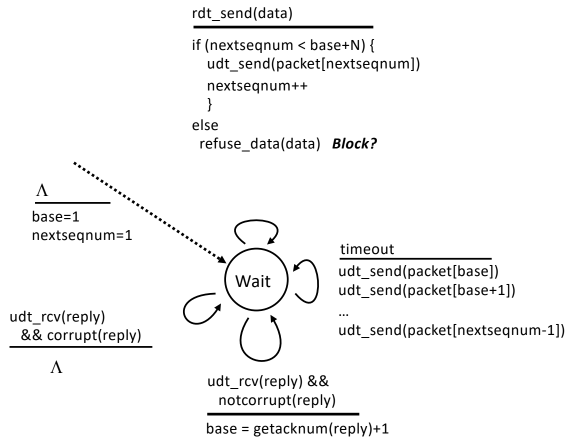
GBN Example with Errors - ALTERNATIVE



61

61

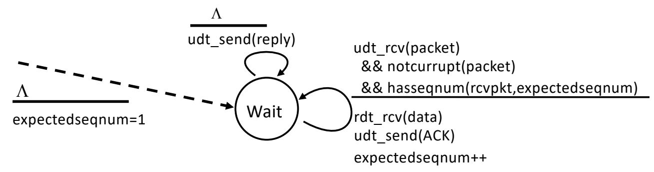
GBN: sender extended FSM



62

62

GBN: receiver extended FSM



ACK-only: always send an ACK for correctly-received packet with the highest *in-order* seq #
- may generate duplicate ACKs
- need only remember **expectedseqnum**

- out-of-order packet:
  - discard (don't buffer) -> no receiver buffering!
  - Re-ACK packet with highest in-order seq #

63

63

Acknowledgements w/ Sliding Window

- Two common options
  - cumulative ACKs: ACK carries next in-order sequence number the receiver expects
  - selective ACKs: ACK individually acknowledges correctly received packets
- Selective ACKs offer more precise information but require more complicated book-keeping
- Many variants that differ in implementation details

64

64

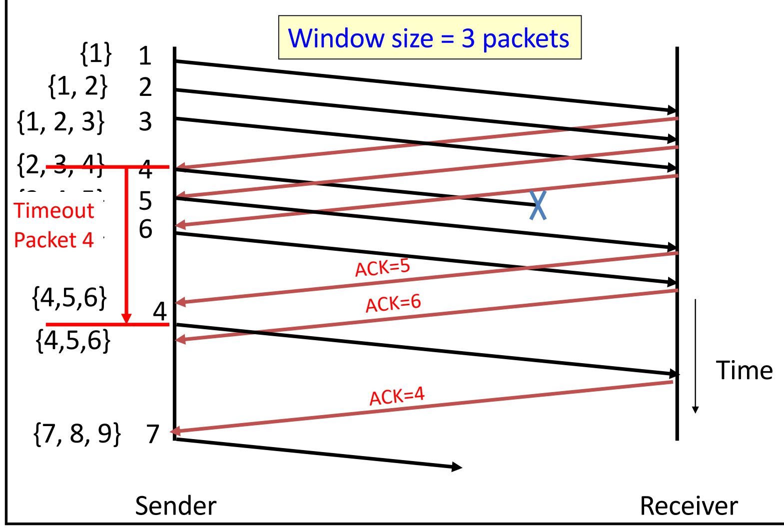
Selective Repeat (SR)

- Sender: transmit up to *n* unacknowledged packets
- Assume packet *k* is lost, *k+1* is not
- Receiver: indicates packet *k+1* correctly received
- Sender: retransmit only packet *k* on timeout
- Efficient in retransmissions but complex book-keeping
  - need a timer per packet

65

65

SR Example with Errors



66

66

Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size (n) is large enough. Throughput is ~ (n/RTT)
  - Stop & Wait is like n = 1.
- Sender has to buffer all unacknowledged packets, because they may require retransmission
- Receiver may be able to accept out-of-order packets, but only up to its buffer limits
- Implementation complexity depends on protocol details (GBN vs. SR)

67

67

Recap: components of a solution

- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - cumulative
  - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)

- Reliability protocols use the above to decide when and what to retransmit or acknowledge

68

68

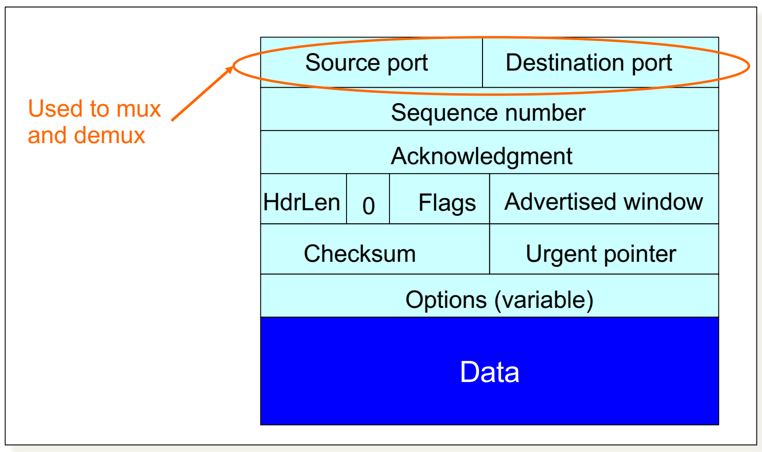
What does TCP do?

Most of our previous tricks + a few differences

- Sequence numbers are byte offsets
- Sender and receiver maintain a sliding window
- Receiver sends cumulative acknowledgements (like GBN)
- Sender maintains a single retx. timer
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit : optimization that uses duplicate ACKs to trigger early retx
- Introduces timeout estimation algorithms

69

TCP Header



71

71

What does TCP do?

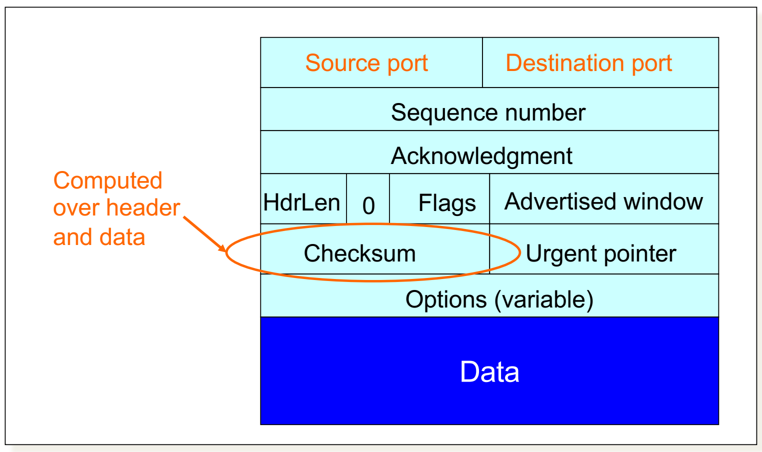
Many of our previous ideas, but some key differences

- Checksum

73

73

TCP Header



74

74

What does TCP do?

Many of our previous ideas, but some key differences

- Checksum
- **Sequence numbers are byte offsets**

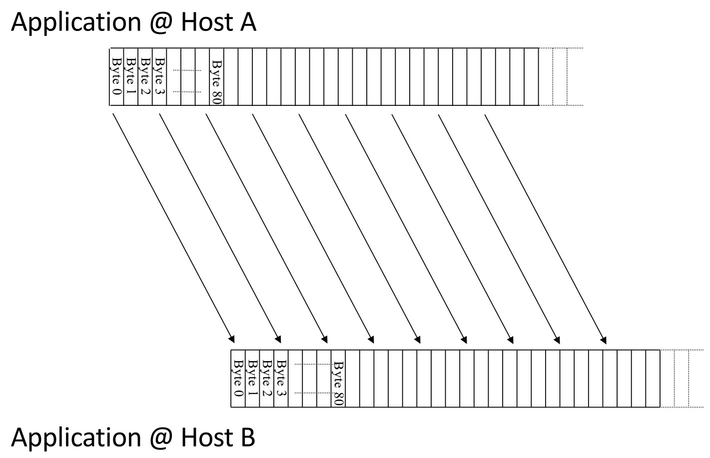
75

TCP: Segments and Sequence Numbers

76

76

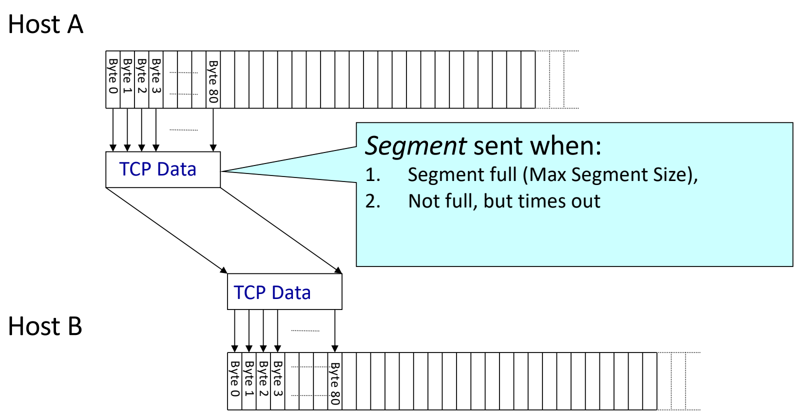
TCP "Stream of Bytes" Service…



77

77

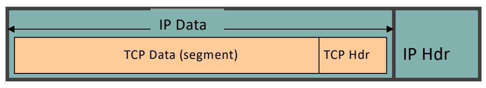
… Provided Using TCP "Segments"



78

78

TCP Segment

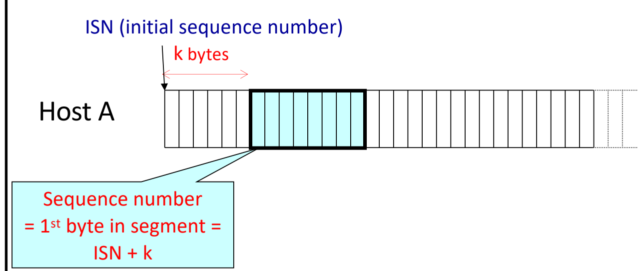


- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes with Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header ≥ 20 bytes long
- TCP **segment**
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream
  - MSS = MTU – (IP header) – (TCP header)

79

79

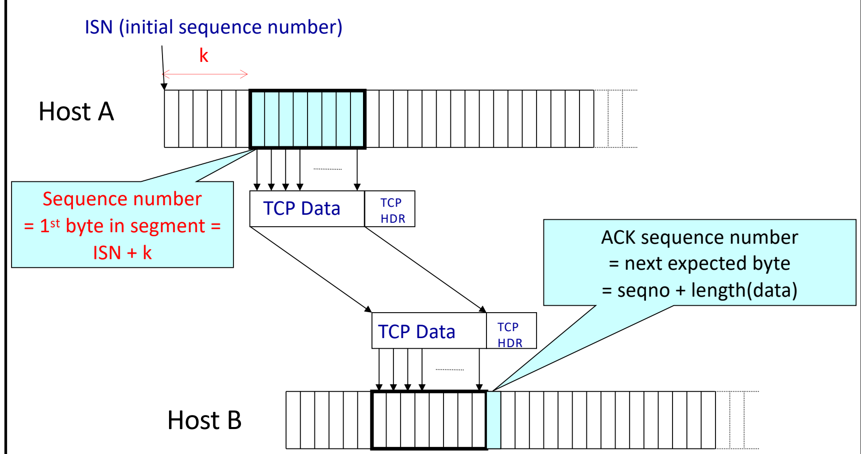
Sequence Numbers



80

80

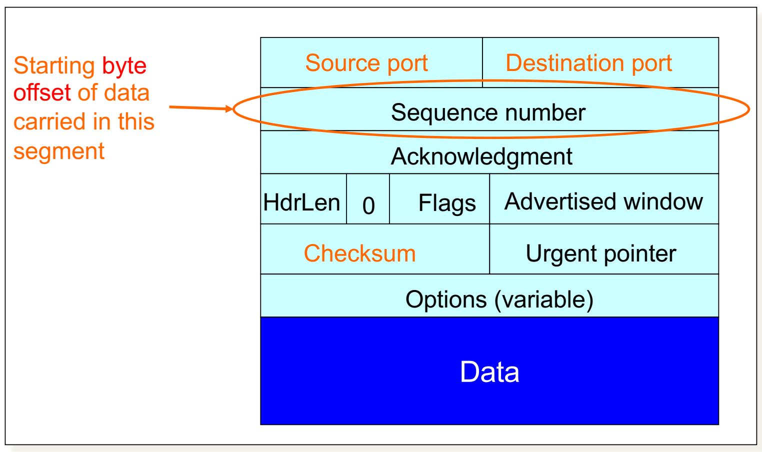
Sequence Numbers



81

81

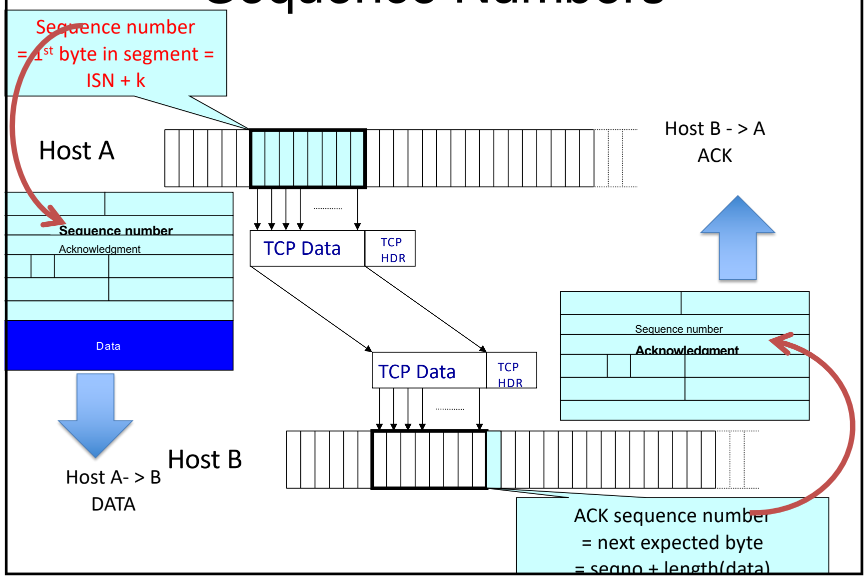
TCP Header



82

82

Sequence Numbers



83

TCP Sequences and ACKS

TCP is full duplex by default

- two independently flows of sequence numbers

Sequence acknowledgement is given in terms of BYTES (not packets); the window is in terms of bytes.

number of packets = window size (bytes) / Segment Size

Servers and Clients are not Source and Destination

Piggybacking increases efficiency but many flows may only have data moving in one direction

84

84

What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)

85

ACKing and Sequence Numbers

- Sender sends packet
  - Data starts with sequence number X
  - Packet contains B bytes [X, X+1, X+2, ...X+B-1]
- Upon receipt of packet, receiver sends an ACK
  - If all data prior to X already received:
    - ACK acknowledges X+B (because that is next expected byte)
  - If highest in-order byte received is Y s.t. (Y+1) < X
    - ACK acknowledges Y+1
    - Even if this has been ACKed before

86

86

Normal Pattern

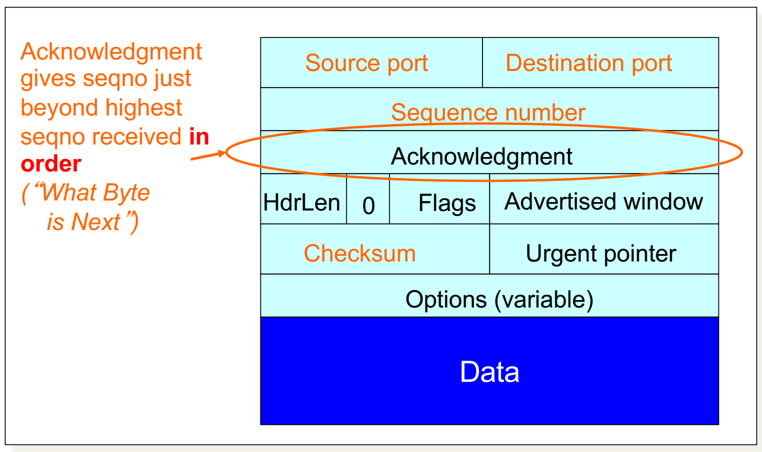
- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B
- Receiver: ACK=X+2B
- Sender: seqno=X+2B, length=B

- Seqno of next packet is same as last ACK field

87

87

TCP Header



88

88

What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers can buffer out-of-sequence packets (like SR)

89

89

Loss with cumulative ACKs

- Sender sends packets with 100B and seqnos.:
  - 100, 200, 300, 400, 500, 600, 700, 800, 900, …
- Assume the fifth packet (seqno 500) is lost, but no others
- Stream of ACKs will be:
  - 200, 300, 400, 500, 500, 500, 500,…

90

90

What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers may not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission

91

91

Loss with cumulative ACKs

- "Duplicate ACKs" are a sign of an isolated loss
  - The lack of ACK progress means 500 hasn't been delivered
  - Stream of ACKs means some packets are being delivered
- Therefore, could trigger resend upon receiving k duplicate ACKs
  - TCP uses k=3
- But response to loss is trickier….

92

92

Loss with cumulative ACKs

- Two choices:
  - Send missing packet and increase W by the number of dup ACKs
  - Send missing packet, and wait for ACK to increase W
- Which should TCP do?

93

93

What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout

94

94

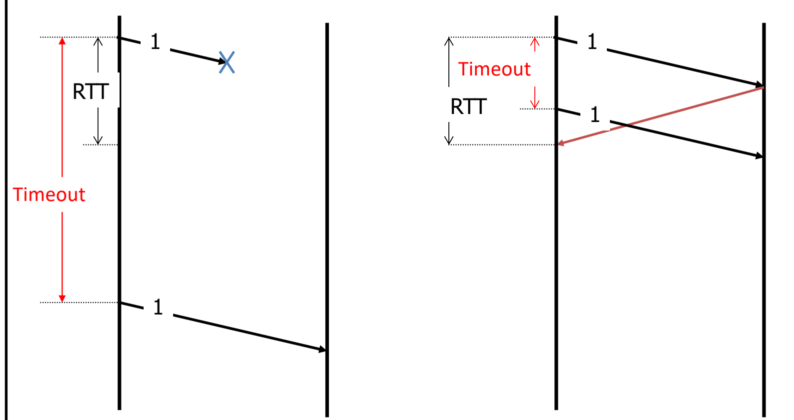
Retransmission Timeout

- If the sender hasn't received an ACK by timeout, retransmit the first packet in the window
- How do we pick a timeout value?

95

95

Timing Illustration



Timeout too long → inefficient

Timeout too short → duplicate packets

96

96

Retransmission Timeout

- If haven't received ack by timeout, retransmit the first packet in the window
- How to set timeout?
  - Too long: connection has low throughput
  - Too short: retransmit packet that was just delayed
- Solution: make timeout proportional to RTT
- But how do we measure RTT?

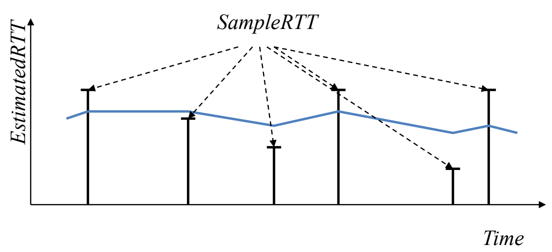
97

97

RTT Estimation

- Use exponential averaging of RTT samples

$$SampleRTT = AckRcvdTime - SendPacketTime$$
$$EstimatedRTT = \alpha \times EstimatedRTT + (1-\alpha) \times SampleRTT$$
$$0 < \alpha \leq 1$$



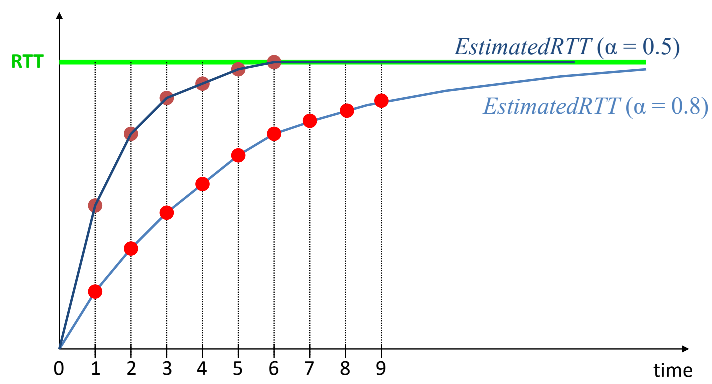
98

98

Exponential Averaging Example

$EstimatedRTT = \alpha * EstimatedRTT + (1 - \alpha) * SampleRTT$

Assume RTT is constant → $SampleRTT$ = RTT

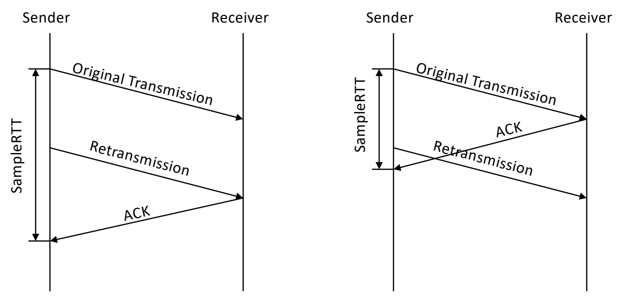


99

99

Problem: Ambiguous Measurements

- How do we differentiate between the real ACK, and ACK of the retransmitted packet?



100

100

Karn/Partridge Algorithm

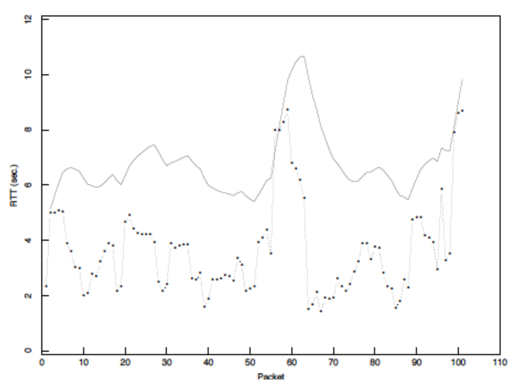
- Measure *SampleRTT* only for original transmissions
  - Once a segment has been retransmitted, do not use it for any further measurements
- Computes EstimatedRTT using $\alpha = 0.875$
- Timeout value (RTO) = 2 × EstimatedRTT
- Employs exponential backoff
  - Every time RTO timer expires, set RTO $\leftarrow$ 2·RTO
  - (Up to maximum ≥ 60 sec)
  - Every time new measurement comes in (= successful original transmission), collapse RTO back to 2 × EstimatedRTT

101

101

Karn/Partridge in action

Figure 5: Performance of an RFC793 retransmit timer



from Jacobson and Karels, SIGCOMM 1988

102

102

Jacobson/Karels Algorithm

- Problem: need to better capture variability in RTT
  - Directly measure deviation
- Deviation = | SampleRTT – EstimatedRTT |
- EstimatedDeviation: exponential average of Deviation
- RTO = EstimatedRTT + 4 x EstimatedDeviation

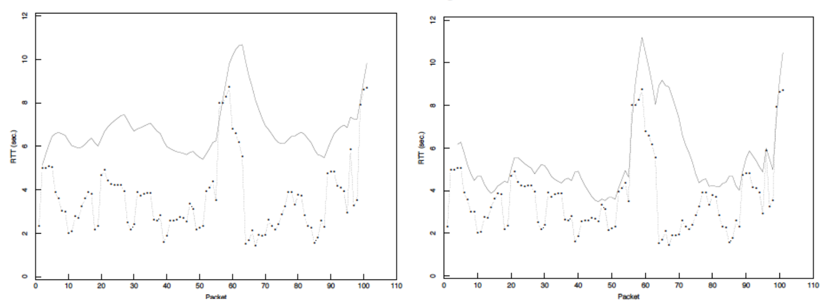
103

103

With Jacobson/Karels

Figure 5: Performance of an RFC793 retransmit timer

Figure 6: Performance of a Mean+Variance retransmit timer



104

104

What does TCP do?

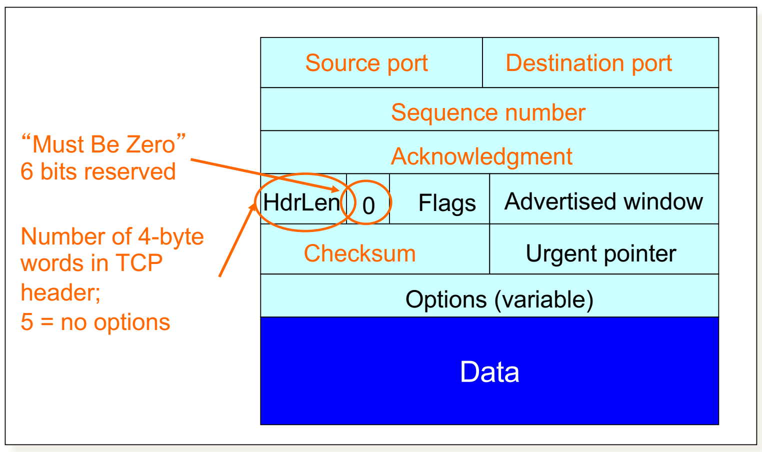
Most of our previous ideas, but some key differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout

105

105

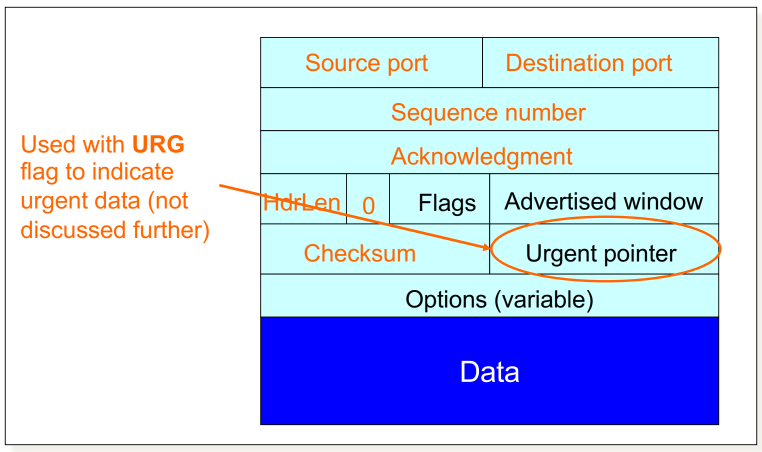
TCP Header: What's left?



106

106

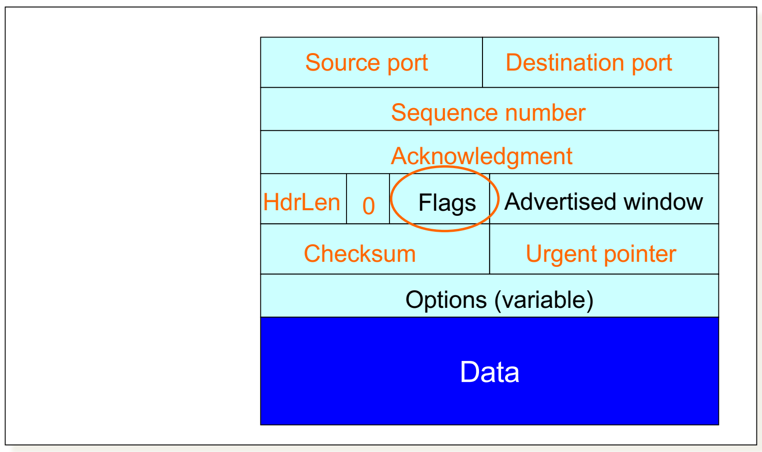
TCP Header: What's left?



107

107

TCP Header: What's left?



108

108

TCP Connection Establishment and Initial Sequence Numbers

109

109

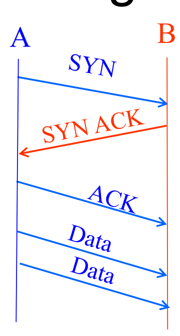
Initial Sequence Number (ISN)

- Sequence number for the very first byte
- Why not just use ISN = 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … small chance an old packet is still in flight
- TCP therefore requires changing ISN
- Hosts exchange ISNs when they establish a connection

110

110

Establishing a TCP Connection



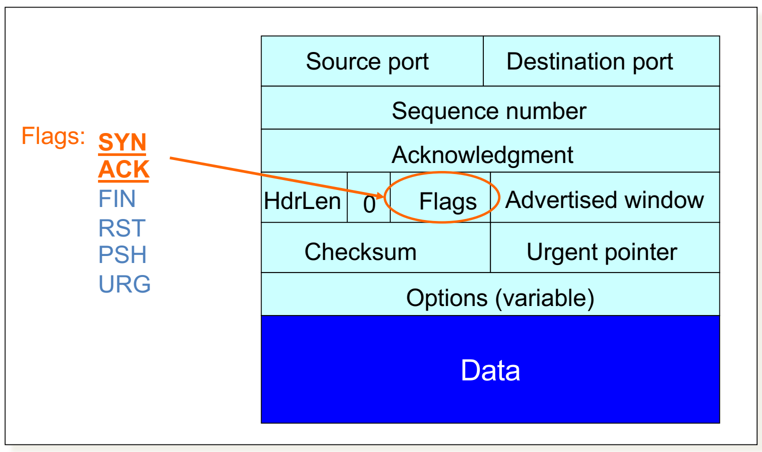
Each host tells its ISN to the other host.

- Three-way handshake to establish connection
  - Host A sends a **SYN** (open; "synchronize sequence numbers") to host B
  - Host B returns a SYN acknowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

111

111

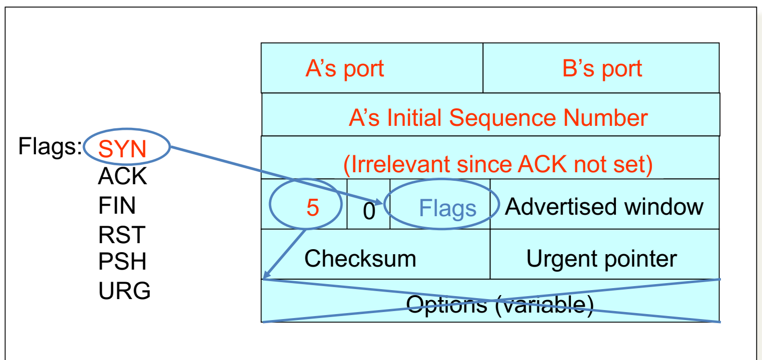
TCP Header



112

112

Step 1: A' s Initial SYN Packet

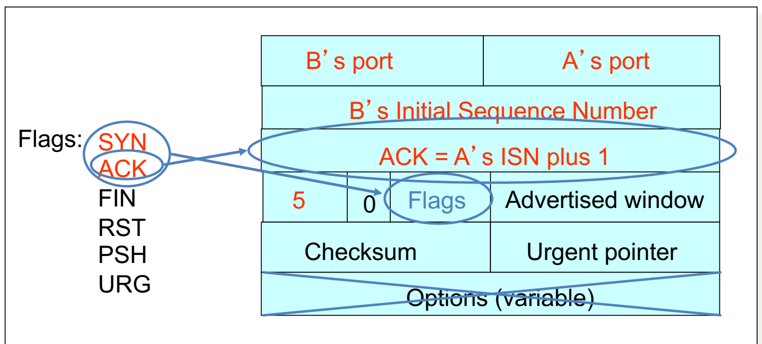


A tells B it wants to open a connection…

113

113

Step 2: B' s SYN-ACK Packet



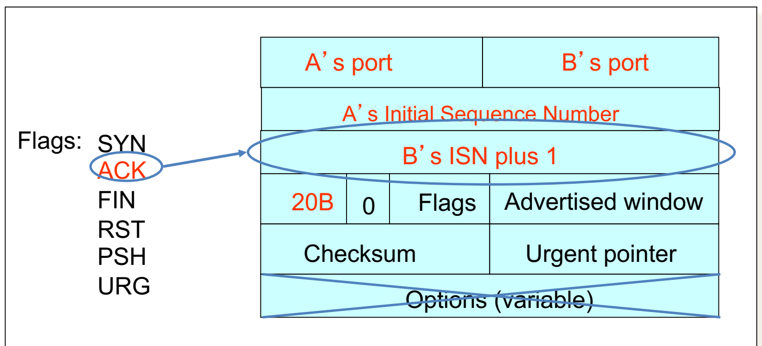
B tells A it accepts, and is ready to hear the next byte…

… upon receiving this packet, A can start sending data

114

114

Step 3: A' s ACK of the SYN-ACK



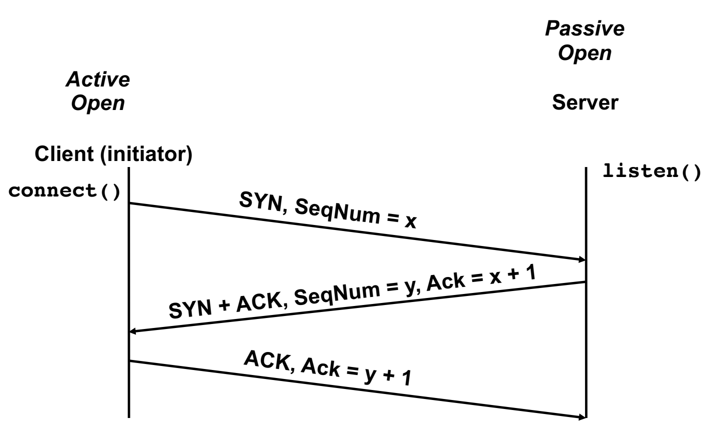
A tells B it' s likewise okay to start sending

… upon receiving this packet, B can start sending data

115

115

Timing Diagram: 3-Way Handshaking



116

116

What if the SYN Packet Gets Lost?

- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or:
  - Server discards the packet (e.g., it's too busy)
- Eventually, no SYN-ACK arrives
  - Sender sets a timer and waits for the SYN-ACK
  - … and retransmits the SYN if needed
- How should the TCP sender set the timer?
  - Sender has no idea how far away the receiver is
  - Hard to guess a reasonable length of time to wait
  - **SHOULD** (RFCs 1122 & 2988) use default of 3 seconds
    - Some implementations instead use 6 seconds

117

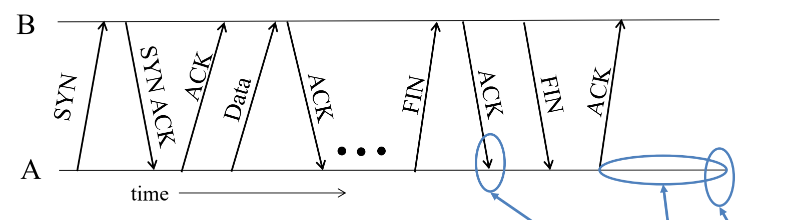
117

Tearing Down the Connection

118

118

Normal Termination, One Side At A Time



- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host acks the byte to confirm
- Closes A's side of the connection, but not B's
  - Until B likewise sends a **FIN**
  - Which A then acks

Connection now **closed**

Connection now **half-closed**

TIME_WAIT:

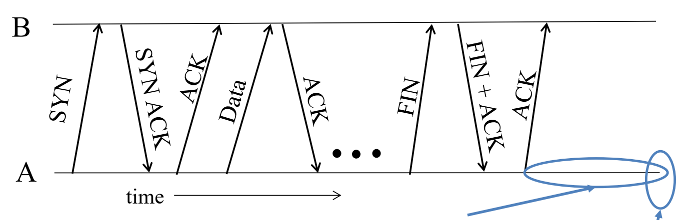
Avoid reincarnation

B will retransmit FIN if ACK is lost

119

119

Normal Termination, Both Together



TIME_WAIT:
Avoid reincarnation

Can retransmit FIN ACK if ACK lost

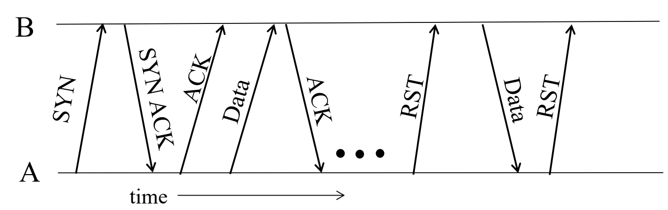
Connection now closed

- Same as before, but B sets **FIN** with their ack of A's **FIN**

120

120

Abrupt Termination

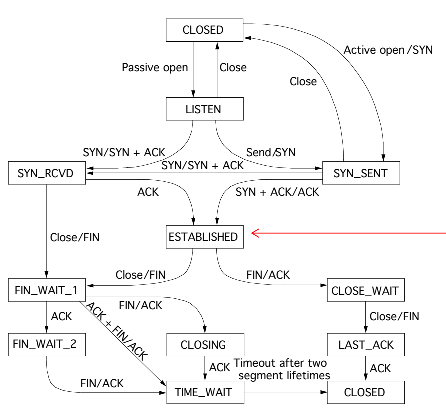


- A sends a RESET (**RST**) to B
  - E.g., because application process on A crashed
- That's it
  - B does not ack the **RST**
  - Thus, **RST** is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another **RST**

121

121

TCP State Transitions

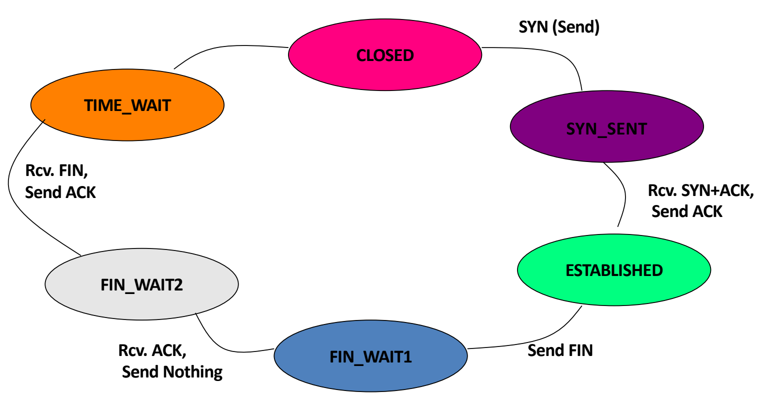


Data, ACK exchanges are in here

122

122

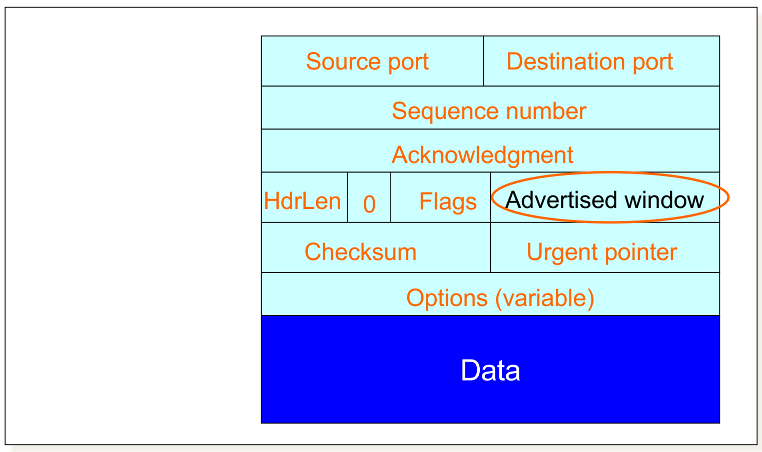
An Simpler View of the Client Side



123

123

TCP Header



124

124

- What does TCP do?
  – ARQ windowing, set-up, tear-down
- Flow Control in TCP

125

125

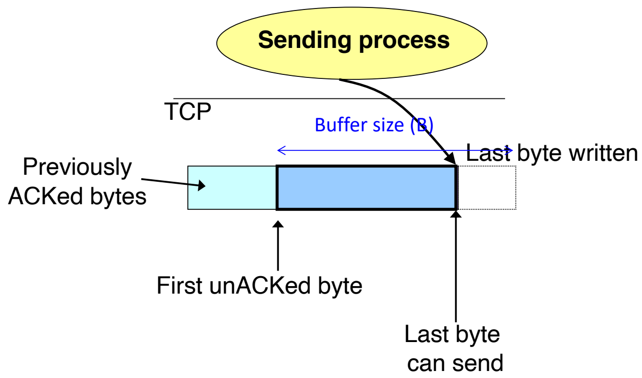
Recap: Sliding Window (so far)

- Both sender & receiver maintain a **window**
- Left edge of window:
  – Sender: beginning of unacknowledged data
  – Receiver: beginning of undelivered data
- Right edge: Left edge + *constant*
  – constant only limited by buffer size in the transport layer

126

126

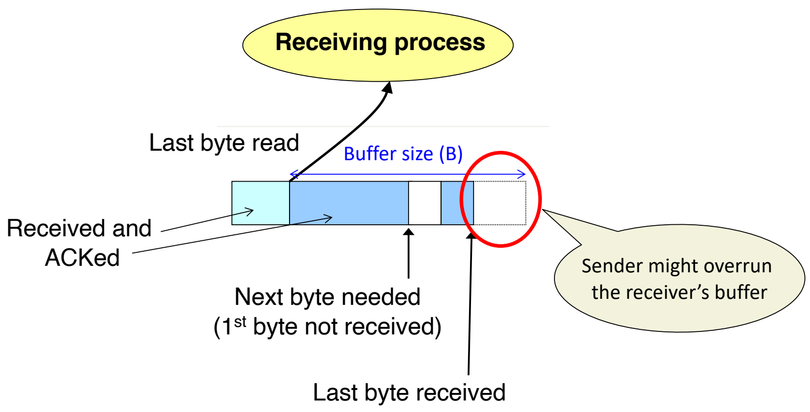
Sliding Window at Sender (so far)



127

127

Sliding Window at Receiver (so far)



128

128

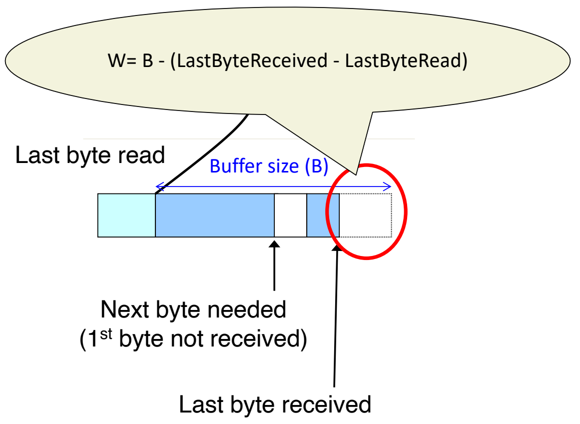
Solution: Advertised Window (Flow Control)

- Receiver uses an "Advertised Window" (W) to prevent sender from overflowing its window
  – Receiver indicates value of W in ACKs
  – Sender limits number of bytes it can have in flight <= W

129

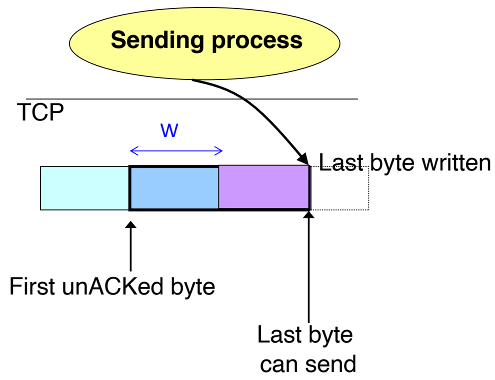
129

Sliding Window at Receiver



130

Sliding Window at Sender (so far)



131

Sliding Window w/ Flow Control

- Sender: window advances when new data ack'd
- Receiver: window advances as receiving process consumes data
- Receiver advertises to the sender where the receiver window currently ends ("righthand edge")
  - Sender agrees not to exceed this amount

132

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec
- Receiver only advertises more space when it has consumed old arriving data
- In original TCP design, that was the **sole** protocol mechanism controlling sender's rate
- What's missing?

133

TCP

- The concepts underlying TCP are simple
  - acknowledgments (feedback)
  - timers
  - sliding windows
  - buffer management
  - sequence numbers

134

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP

136

We have seen:

– Flow control: adjusting the sending rate to keep from overwhelming a slow *receiver*

Now lets attend…

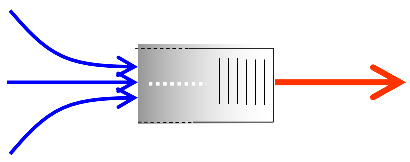
– Congestion control: adjusting the sending rate to keep from overloading the *network*

137

137

Statistical Multiplexing → Congestion

- If two packets arrive at the same time
  – A router can only transmit one
  – … and either buffers or drops the other
- If many packets arrive in a short period of time
  – The router cannot keep up with the arriving traffic
  – … delays traffic, and the buffer may eventually overflow
- Internet traffic is bursty

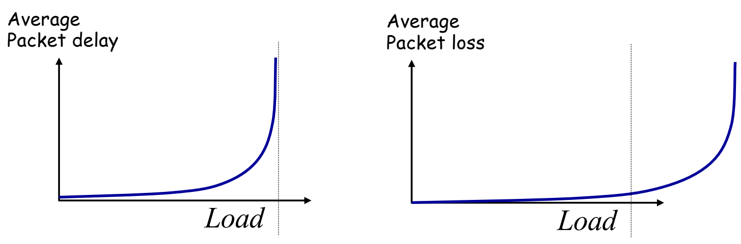


138

138

Congestion is undesirable

Typical queuing system with bursty arrivals



Must balance utilization versus delay and loss

139

139

Who Takes Care of Congestion?

- Network? End hosts? Both?

- TCP's approach:
  – **End hosts** adjust sending rate
  – Based on **implicit feedback** from network

- Not the only approach
  – A consequence of history rather than planning

140

140

Some History: TCP in the 1980s

- Sending rate only limited by flow control
  – Packet drops → senders (repeatedly!) retransmit a full window's worth of packets

- Led to "congestion collapse" starting Oct. 1986
  – Throughput on the NSF network dropped from 32Kbits/s to 40bits/sec

- "Fixed" by Van Jacobson's development of TCP's congestion control (CC) algorithms

141

141

Jacobson's Approach

- Extend TCP's existing window-based protocol but adapt the window size in response to congestion
  – required no upgrades to routers or applications!
  – patch of a few lines of code to TCP implementations

- A pragmatic and effective solution
  – but many other approaches exist

- Extensively improved on since
  – topic now sees less activity in ISP contexts
  – but is making a comeback in datacenter environments

142

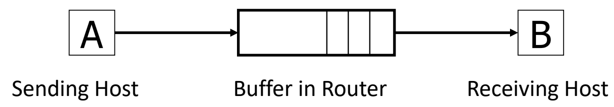
142

Three Issues to Consider

- Discovering the available (bottleneck) bandwidth
- Adjusting to variations in bandwidth
- Sharing bandwidth between flows

143

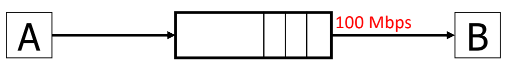
Abstract View



- Ignore internal structure of router and model it as having a single queue for a particular input-output pair

144

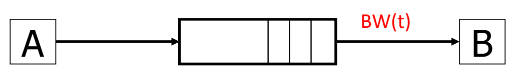
Discovering available bandwidth



- Pick sending rate to match bottleneck bandwidth
  - Without any *a priori* knowledge
  - Could be gigabit link, could be a modem

145

Adjusting to variations in bandwidth



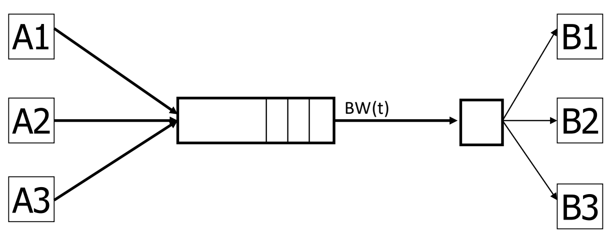
- Adjust rate to match instantaneous bandwidth
  - Assuming you have rough idea of bandwidth

146

Multiple flows and sharing bandwidth

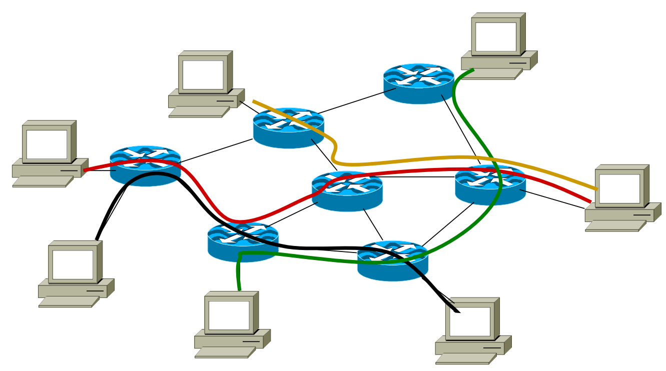
Two Issues:

- Adjust total sending rate to match bandwidth
- Allocation of bandwidth between flows



147

Reality

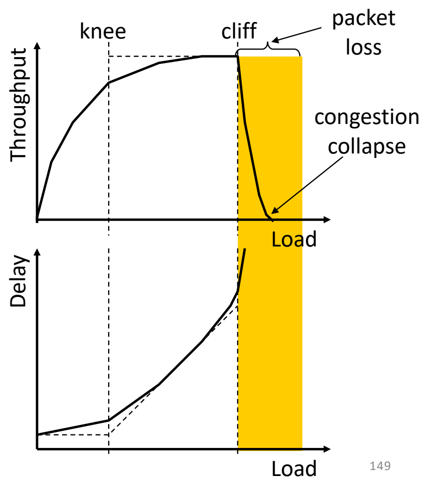


Congestion control is a resource allocation problem involving many flows, many links, and complicated global dynamics

148

View from a single flow

- Knee – point after which
  - Throughput increases slowly
  - Delay increases fast
- Cliff – point after which
  - Throughput starts to drop to zero (congestion collapse)
  - Delay approaches infinity



149

General Approaches

(0) Send without care

– Many packet drops

150

General Approaches

(0) Send without care

(1) Reservations

– Pre-arrange bandwidth allocations

– Requires negotiation before sending packets

– Low utilization

151

General Approaches

(0) Send without care

(1) Reservations

(2) Pricing

– Don't drop packets for the high-bidders

– Requires payment model

152

General Approaches

(0) Send without care

(1) Reservations

(2) Pricing

(3) Dynamic Adjustment

– Hosts probe network; infer level of congestion; adjust

– Network reports congestion level to hosts; hosts adjust

– Combinations of the above

– Simple to implement but suboptimal, messy dynamics

153

General Approaches

(0) Send without care

(1) Reservations

(2) Pricing

(3) Dynamic Adjustment

All three techniques have their place

- *Generality* of dynamic adjustment has proven powerful
- Doesn't presume business model, traffic characteristics, application requirements; does assume good citizenship

154

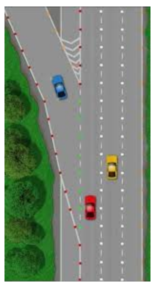
TCP's Approach in a Nutshell

- TCP connection has window
  - Controls number of packets in flight
- Sending rate: ~Window/RTT
- Vary window size to control sending rate

155

155

Windows, Buffers, and TCP




156

156

Windows, Buffers, and TCP

- TCP connection has a window
  - Controls number of packets in flight;
    filling a channel to improve throughput, and
    vary window size to control sending rate
- Buffers adapt mis-matched channels
  - Buffers smooth bursts
  - Adapt (re-time) arrivals for multiplexing

157

157

Windows, Buffers, and TCP

Buffers & TCP can make link utilization 100%

but

Buffers add delay, **variable** delay

158

158

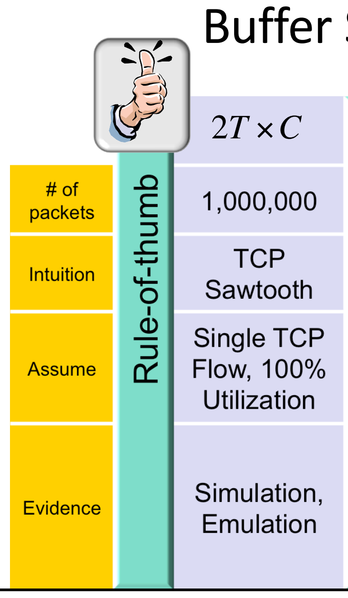
Sizing Buffers in Routers

- Packet loss
  - Queue overload, and subsequent packet loss
- End-to-end delay
  - Transmission, propagation, and queueing delay
  - The only variable part is queueing delay
- Router architecture
  - Board space, power consumption, and cost
  - On chip buffers: higher density, higher capacity

159

159

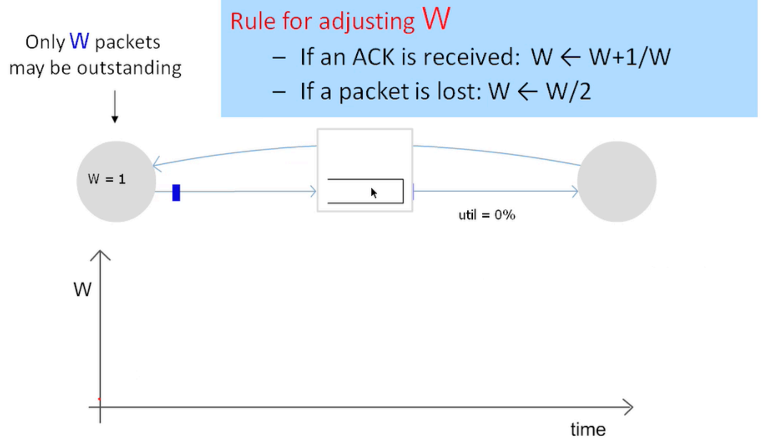
Buffer Sizing Story



160

Continuous ARQ (TCP) adapting to congestion

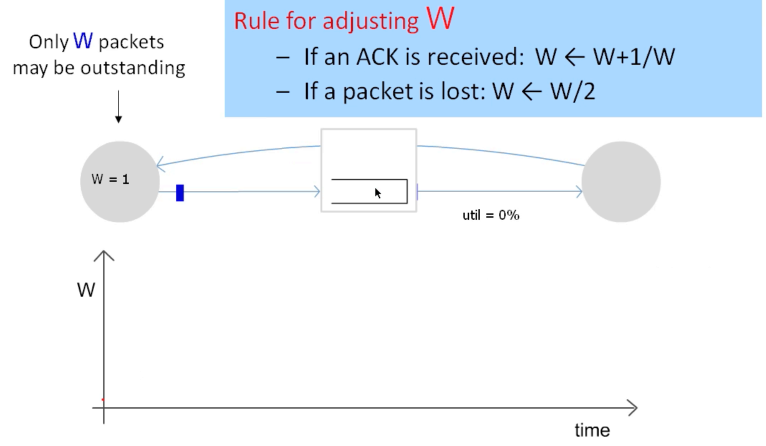
Rule for adjusting W
- If an ACK is received: W ← W+1/W
- If a packet is lost: W ← W/2



161

Continuous ARQ (TCP) adapting to congestion

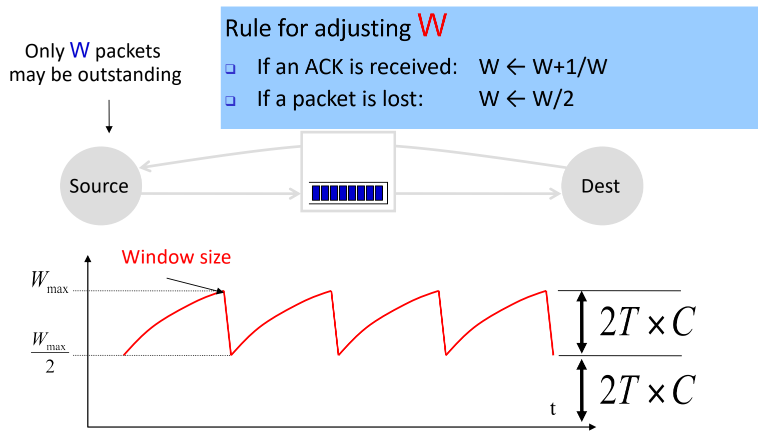
Rule for adjusting W
- If an ACK is received: W ← W+1/W
- If a packet is lost: W ← W/2



162

Rule-of-thumb – Intuition

Rule for adjusting W
- If an ACK is received: W ← W+1/W
- If a packet is lost: W ← W/2



163

Buffers in Routers

So how large should the buffers be?

Buffer size matters
- Packet loss
  - Queue overload, and subsequent packet loss
- End-to-end delay
  - Transmission, propagation, and queueing delay
  - The only variable part is queueing delay

164

164

Buffer Sizing Story

	Rule-of-thumb $2T \times C$	Small Buffers $\frac{2T \times C}{\sqrt{n}}$
# of packets	1,000,000	10,000
Intuition	TCP Sawtooth	Sawtooth Smoothing
Assume	Single TCP Flow, 100% Utilization	Many Flows, 100% Utilization
Evidence	Simulation, Emulation	Simulations, Test-bed and Real Network Experiments

165

Buffers in Routers

So how large should the buffers be?

Buffer size matters
- Packet loss
  - Queue overload, and subsequent packet loss
- End-to-end delay
  - Transmission, propagation, and queueing delay
  - The only variable part is queueing delay
- Router architecture
  - Board space, power consumption, and cost
  - On chip buffers: higher density, higher capacity

166

166

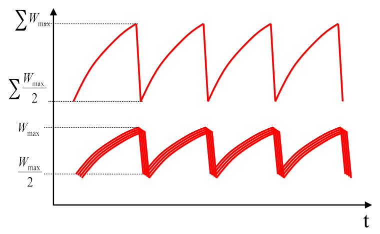
Small Buffers – Intuition

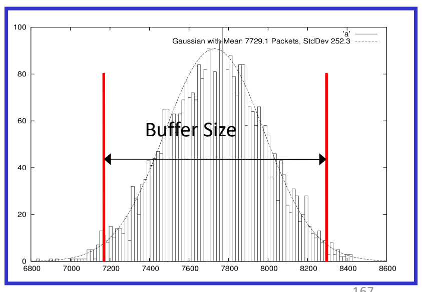
Synchronized Flows

- Aggregate window has same dynamics
- Therefore buffer occupancy has same dynamics
- Rule-of-thumb still holds.

Many TCP Flows

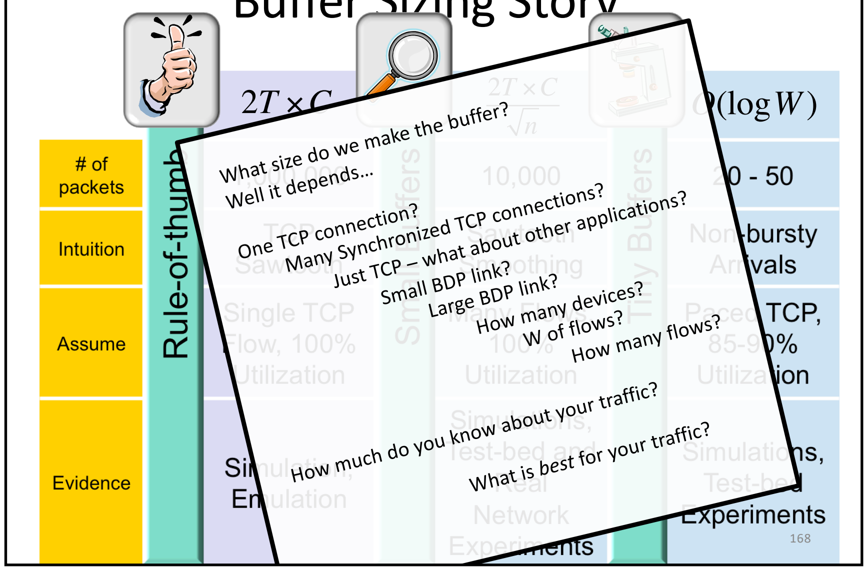
- Independent, desynchronized
- Central limit theorem says the aggregate becomes Gaussian
- Variance (buffer size) decreases as N increases





167

Buffer Sizing Story



168

TCP's Approach in a Nutshell

- TCP connection has window
  - Controls number of packets in flight
- Sending rate: ~Window/RTT
- Vary window size to control sending rate

169

All These Windows…

- Congestion Window: CWND
  - How many bytes can be sent without overflowing routers
  - Computed by the sender using congestion control algorithm
- Flow control window: AdvertisedWindow (RWND)
  - How many bytes can be sent without overflowing receiver's buffers
  - Determined by the receiver and reported to the sender
- Sender-side window = minimum{CWND,RWND}
  - Assume for this material that RWND >> CWND

170

Note

- This lecture will talk about CWND in units of MSS
  - (Recall MSS: Maximum Segment Size, the amount of payload data in a TCP packet)
  - This is only for pedagogical purposes
- **In reality this is a LIE:** Real implementations maintain CWND in bytes

171

Two Basic Questions

- How does the sender detect congestion?
- How does the sender adjust its sending rate?
  - To address three issues
    - Finding available bottleneck bandwidth
    - Adjusting to bandwidth variations
    - Sharing bandwidth

172

Detecting Congestion

- Packet delays
  - Tricky: noisy signal (delay often varies considerably)
- Router tell end-hosts they're congested
- Packet loss
  - Fail-safe signal that TCP already has to detect
  - Complication: non-congestive loss (checksum errors)
- Two indicators of packet loss
  - No ACK after certain time interval: timeout
  - Multiple duplicate ACKs

173

Not All Losses the Same

- Duplicate ACKs: isolated loss
  - Still getting ACKs
- Timeout: much more serious
  - Not enough packets in progress to trigger duplicate-acks, OR
  - Suffered several losses
- We will adjust rate differently for each case

174

Rate Adjustment

- Basic structure:
  - Upon receipt of ACK (of new data): increase rate
  - Upon detection of loss: decrease rate
- How we increase/decrease the rate depends on the phase of congestion control we're in:
  - Discovering available bottleneck bandwidth *vs.*
  - Adjusting to bandwidth variations

175

Bandwidth Discovery with Slow Start

- Goal: estimate available bandwidth
  - start slow (for safety)
  - but ramp up quickly (for efficiency)
- Consider
  - RTT = 100ms, MSS=1000bytes
  - Window size to fill 1Mbps of BW = 12.5 packets
  - Window size to fill 1Gbps = 12,500 packets
  - Either is possible!

176

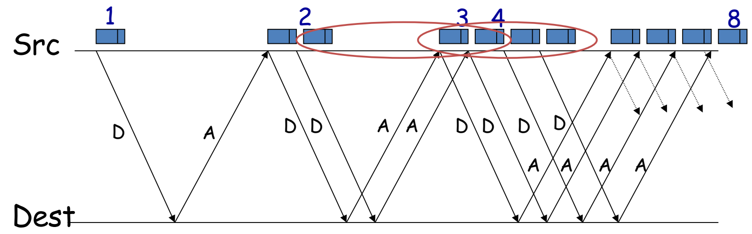
"Slow Start" Phase

- Sender starts at a slow rate but increases **exponentially** until first loss
- Start with a small congestion window
  - Initially, CWND = 1
  - So, initial sending rate is MSS/RTT
- Double the CWND for each RTT with no loss

177

Slow Start in Action

- For each RTT: double CWND
- Simpler implementation: for each ACK, CWND += 1



178

Adjusting to Varying Bandwidth

- Slow start gave an estimate of available bandwidth
- Now, want to track variations in this available bandwidth, oscillating around its current value
  - Repeated probing (rate increase) and backoff (rate decrease)
- TCP uses: "Additive Increase Multiplicative Decrease" (AIMD)
  - We'll see why shortly…

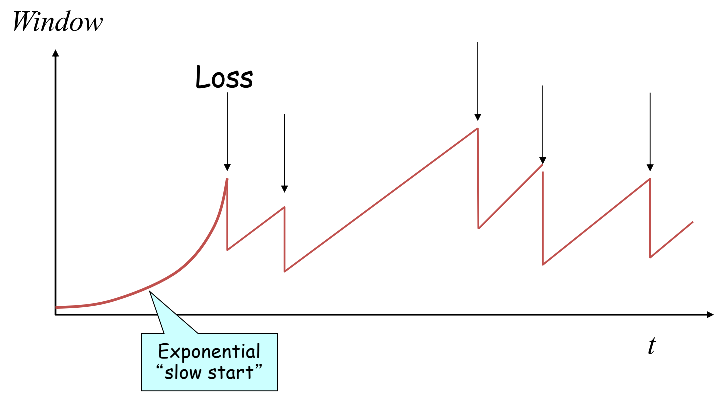
179

AIMD

- Additive increase
  - Window grows by one MSS for every RTT with no loss
  - For each successful RTT, CWND = CWND + 1
  - Simple implementation:
    - for each ACK, CWND = CWND+ 1/CWND
- Multiplicative decrease
  - On loss of packet, divide congestion window in **half**
  - On loss, CWND = CWND/2

180

Leads to the TCP "Sawtooth"



181

Slow-Start vs. AIMD

- When does a sender stop Slow-Start and start Additive Increase?
- Introduce a "slow start threshold" (ssthresh)
  - Initialized to a large value
  - On timeout, ssthresh = CWND/2
- When CWND = ssthresh, sender switches from slow-start to AIMD-style increase

182

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD

183

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD, Fast-Recovery

184

One Final Phase: Fast Recovery

- The problem: congestion avoidance too slow in recovering from an isolated loss

185

185

Example (in units of MSS, not bytes)

- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101
- 10 packets [101, 102, 103,..., 110] are in flight
  - Packet 101 is dropped
  - What ACKs do they generate?
  - And how does the sender respond?

186

186

The problem – A timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)
- ACK 111 (due to 101) ← only now can we transmit new packets
- Plus no packets in flight so ACK "clocking" (to increase CWND) stalls for another RTT

187

187

Solution: Fast Recovery

Idea: Grant the sender temporary "credit" for each dupACK so as to keep packets in flight

- If dupACKcount = 3
  - ssthresh = cwnd/2
  - cwnd = ssthresh + 3
- While in fast recovery
  - cwnd = cwnd + 1 for each additional duplicate ACK
- Exit fast recovery after receiving new ACK
  - set cwnd = ssthresh

188

188

Example

- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101
- 10 packets [101, 102, 103,..., 110] are in flight
  - Packet 101 is dropped

189

189

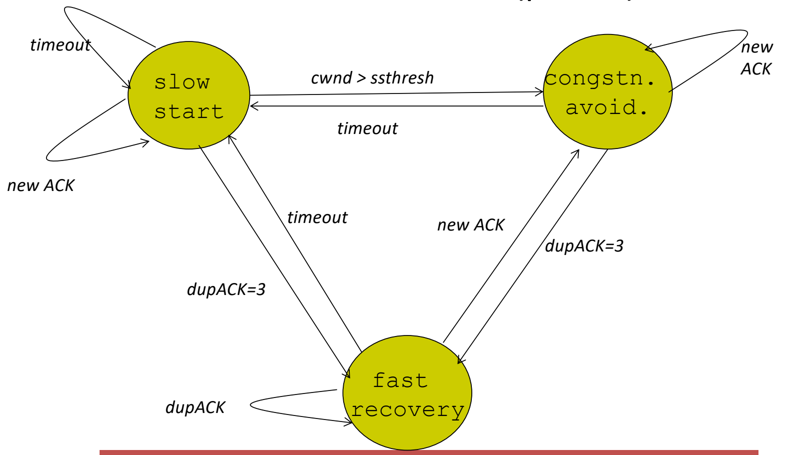
Timeline

- ACK 101 (due to 102) cwnd=10 dup#1
- ACK 101 (due to 103) cwnd=10 dup#2
- ACK 101 (due to 104) cwnd=10 dup#3
- REXMIT 101 ssthresh=5 cwnd= 8 (5+3)
- ACK 101 (due to 105) cwnd= 9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)
- ACK 101 (due to 109) cwnd=13 (xmit 113)
- ACK 101 (due to 110) cwnd=14 (xmit 114)
- ACK 111 (due to 101) cwnd = 5 (xmit 115) ← exiting fast recovery
- Packets 111-114 already in flight
- ACK 112 (due to 111) cwnd = 5 + 1/5 ← back in congestion avoidance

190

190

Putting it all together: The TCP State Machine (partial)



- How are ssthresh, CWND and dupACKcount updated for each event that causes a state transition?

191

TCP Flavors

- TCP-Tahoe
  - cwnd =1 on triple dupACK
- TCP-Reno
  - cwnd =1 on timeout
  - cwnd = cwnd/2 on triple dupack
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - incorporates selective acknowledgements

192

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD, Fast-Recovery, Throughput

193

193

TCP Flavors

- TCP-Tahoe
  - CWND =1 on triple dupACK
- TCP-Reno
  - CWND =1 on timeout
  - CWND = CWND/2 on triple dupack
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - incorporates selective acknowledgements

Our default assumption

194

194

Interoperability

- How can all these algorithms coexist? Don't we need a single, uniform standard?
- What happens if I'm using Reno and you are using Tahoe, and we try to communicate?

195

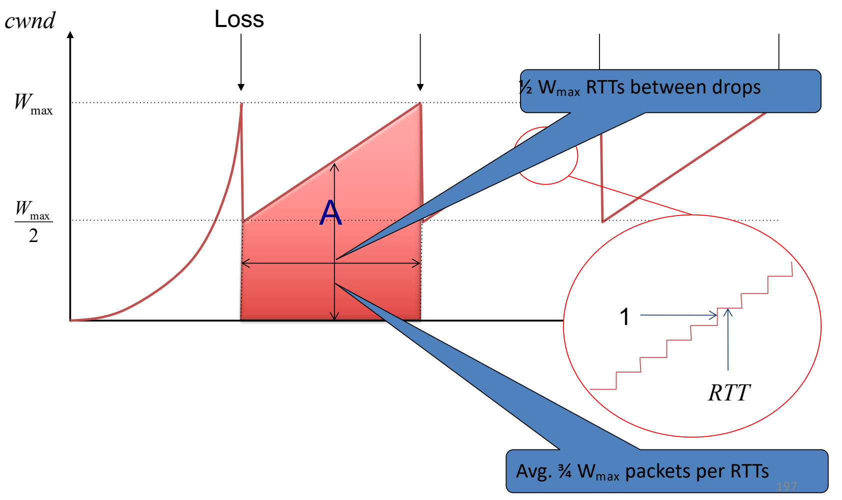
195

TCP Throughput Equation

196

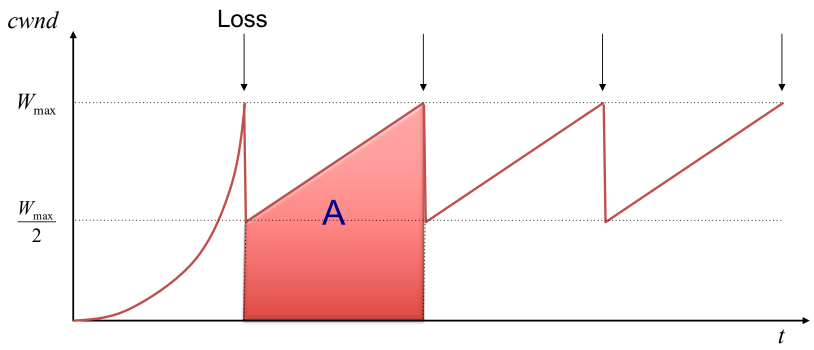
196

A Simple Model for TCP Throughput



197

A Simple Model for TCP Throughput



Packet drop rate, $p = 1/A$, where $A = \frac{3}{8}W_{max}^2$

Throughput, $B = \frac{A}{\left(\frac{W_{max}}{2}\right)RTT} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$

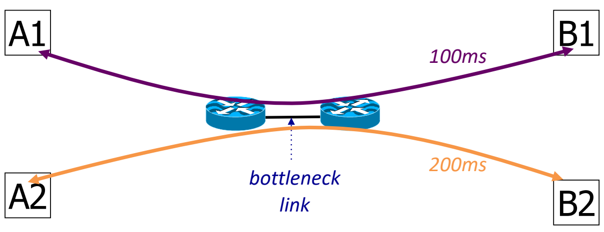
198

198

Implications (1): Different RTTs

$$\text{Throughput} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- Flows get throughput inversely proportional to RTT
- TCP unfair in the face of heterogeneous RTTs!



199

199

Implications (2): High Speed TCP

$$\text{Throughput} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- Assume RTT = 100ms, MSS=1500bytes
- What value of *p* is required to reach 100Gbps throughput
  - ~ 2×10^{-12}
- How long between drops?
  - ~ 16.6 hours
- How much data has been sent in this time?
  - ~ 6 petabits
- These are not practical numbers!

200

200

Adapting TCP to High Speed

- Once past a threshold speed, increase CWND faster
  - A proposed standard [Floyd'03]: once speed is past some threshold, change equation to $p^{-.8}$ rather than $p^{-.5}$
  - Let the additive constant in AIMD depend on CWND
- Other approaches?
  - Multiple simultaneous connections (*hacky* but works today)
  - Router-assisted approaches (will see shortly)

201

201

Implications (3): *Rate*-based CC

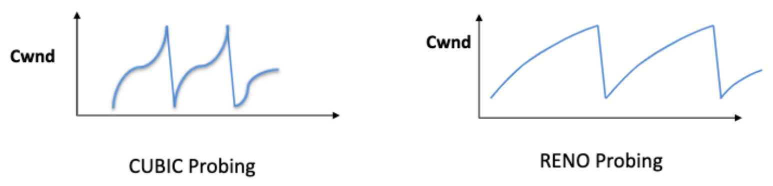
$$\text{Throughput} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- TCP throughput is "choppy"
  - repeated swings between W/2 to W
- Some apps would prefer sending at a steady rate
  - e.g., streaming apps
- A solution: "Equation-Based Congestion Control"
  - ditch TCP's increase/decrease rules and just follow the equation
  - measure drop percentage *p*, and set rate accordingly
- Following the TCP equation ensures we're "TCP friendly"
  - i.e., use no more than TCP does in similar setting

202

202

TCP Cubic V TCP Reno



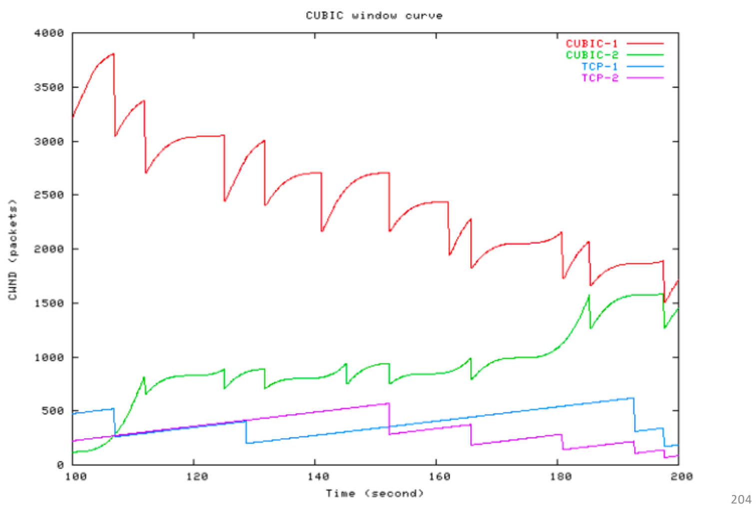
Faster Increase
Sharp decrease

Slow Increase
Sharp decrease

203

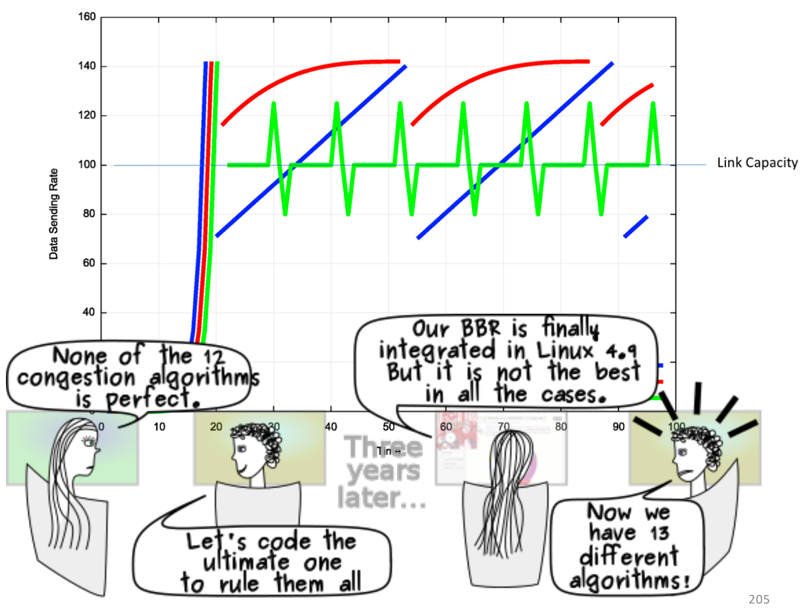
203

New world of fairness….



204

204



205

205

Recap: TCP problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows complete before discovering available capacity
- AIMD impractical for high speed links
- Sawtooth discovery too choppy for some apps
- Unfair under heterogeneous RTTs
- Tight coupling with reliability mechanisms
- Endhosts can cheat

Routers tell endpoints if they're congested

Routers tell endpoints what rate to send at

Routers enforce fair sharing

Could fix many of these with some help from routers!

206

206

Router-Assisted Congestion Control

- Three tasks for CC:
  – Isolation/fairness
  – Adjustment*
  – Detecting congestion

* This may be *automatic* eg loss-response of TCP

207

207

How can routers ensure each flow gets its "fair share"?

208

208

Fairness: General Approach

- Routers classify packets into "flows"
  - (For now) flows are packets between same source/destination
- Each flow has its own FIFO queue in router
- Router services flows in a fair fashion
  - When line becomes free, take packet from next flow in a fair order
- What does "fair" mean exactly?

209

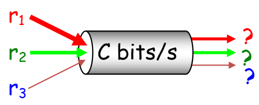
209

Max-Min Fairness

- Given set of bandwidth demands r_i and total bandwidth C, max-min bandwidth allocations are:

  $a_i = \min(f, r_i)$

  where f is the unique value such that $\mathrm{Sum}(a_i) = C$

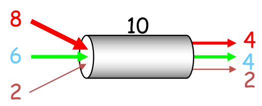


210

210

Example

- $C = 10$; $r_1 = 8$, $r_2 = 6$, $r_3 = 2$; $N = 3$
- $C/3 = 3.33 \rightarrow$
  - Can service all of r_3
  - Remove r_3 from the accounting: $C = C - r_3 = 8$; $N = 2$
- $C/2 = 4 \rightarrow$
  - Can't service all of r_1 or r_2
  - So hold them to the remaining fair share: $f = 4$



$f = 4$:
min(8, 4) = 4
min(6, 4) = 4
min(2, 4) = 2

211

211

Max-Min Fairness

- Given set of bandwidth demands r_i and total bandwidth C, max-min bandwidth allocations are:

  $a_i = \min(f, r_i)$
- where f is the unique value such that $\mathrm{Sum}(a_i) = C$
- Property:
  - If you don't get full demand, no one gets more than you
- This is what round-robin service gives if all packets are the same size

212

212

How do we deal with packets of different sizes?

- Mental model: Bit-by-bit round robin ("fluid flow")
- Can you do this in practice?
- No, packets cannot be preempted
- But we can approximate it
  - This is what "fair queuing" routers do

213

213

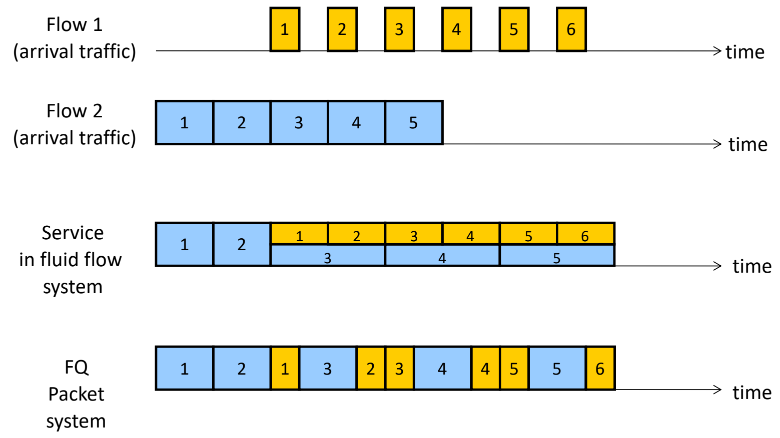
Fair Queuing (FQ)

- For each packet, compute the time at which the last bit of a packet would have left the router *if* flows are served bit-by-bit
- Then serve packets in the increasing order of their deadlines

214

214

Example



215

Fair Queuing (FQ)

- Think of it as an implementation of round-robin generalized to the case where not all packets are equal sized
- Weighted fair queuing (WFQ): assign different flows different shares
- Today, some form of WFQ implemented in almost all routers
  - Not the case in the 1980-90s, when CC was being developed
  - Mostly used to isolate traffic at larger granularities (e.g., per-prefix)

216

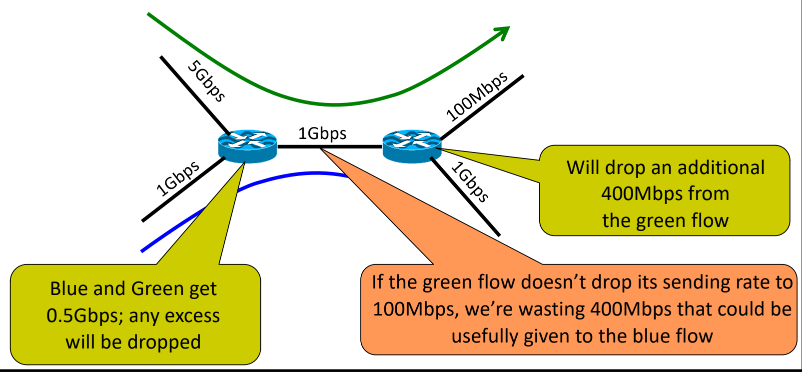
FQ vs. FIFO

- FQ advantages:
  - Isolation: cheating flows don't benefit
  - Bandwidth share does not depend on RTT
  - Flows can pick any rate adjustment scheme they want
- Disadvantages:
  - More complex than FIFO: per flow queue/state, additional per-packet book-keeping

217

FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion



218

FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion
  - robust to cheating, variations in RTT, details of delay, reordering, retransmission, *etc.*
- But congestion (and packet drops) still occurs
- And we still want end-hosts to discover/adapt to their fair share!
- What would the end-to-end argument say w.r.t. congestion control?

219

Fairness is a controversial goal

- What if you have 8 flows, and I have 4?
  - Why should you get twice the bandwidth
- What if your flow goes over 4 congested hops, and mine only goes over 1?
  - Why shouldn't you be penalized for using more scarce bandwidth?
- And what is a flow anyway?
  - TCP connection
  - Source-Destination pair?
  - Source?

220

Explicit Congestion Notification (ECN)

- Single bit in packet header; set by congested routers
  - If data packet has bit set, then ACK has ECN bit set
- Many options for when routers set the bit
  - tradeoff between (link) utilization and (packet) delay
- Congestion semantics can be exactly like that of drop
  - I.e., endhost reacts as though it saw a drop
- Advantages:
  - Don't confuse corruption with congestion; recovery w/ rate adjustment
  - Can serve as an early indicator of congestion to avoid delays
  - Easy (easier) to incrementally deploy
    - defined as extension to TCP/IP in RFC 3168 (uses diffserv bits in the IP header)

221

221

TCP in detail

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD, Fast-Recovery, Throughput
- Limitations of TCP Congestion Control
- Router-assisted Congestion Control (eg ECN)

222

222

Transport Recap

A *"big bag"*:
Multiplexing, reliability, error-detection, error-recovery, flow and congestion control, ….

- UDP:
  - Minimalist - multiplexing and error detection
- TCP:
  - somewhat hacky
  - but practical/deployable
  - good enough to have raised the bar for the deployment of new, more optimal, approaches
  - though the needs of datacenters might change the status quos
- Beyond TCP (discussed in Topic 6):
  - QUIC / application-aware transport layers

223

223

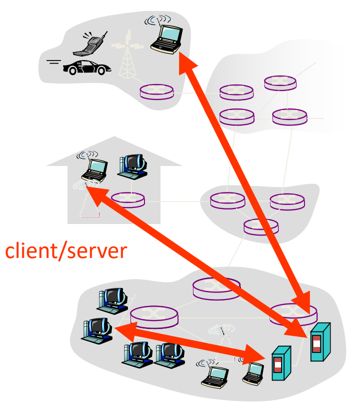
Topic 6 – Applications

- Infrastructure Services (DNS)
  - Now with added security…
- Traditional Applications (web)
  - Now with added QUIC
- Multimedia Applications (SIP)
  - One day (more…)…
- P2P Networks
  - Every device serves

1

1

Client-server paradigm reminder



server:
- always-on host
- permanent IP address
- server farms for scaling

clients:
- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

2

2

Relationship Between Names&Addresses

- Addresses can change underneath
  - Move www.bbc.co.uk to 212.58.246.92
  - Humans/Apps should be unaffected
- Name could map to multiple IP addresses
  - www.bbc.co.uk to multiple replicas of the Web site
  - Enables
    - Load-balancing
    - Reducing latency by picking nearby servers
- Multiple names for the same address
  - E.g., aliases like www.bbc.co.uk and bbc.co.uk
  - Mnemonic stable name, and dynamic canonical name
    - Canonical name = actual name of host

3

3

Mapping from Names to Addresses

- Originally: per-host file /etc/hosts*
  - SRI (Menlo Park) kept master copy
  - Downloaded regularly
  - Flat namespace
- Single server not resilient, doesn't scale
  - Adopted a distributed hierarchical system
- Two intertwined hierarchies:
  - Infrastructure: hierarchy of DNS servers
  - Naming structure: www.bbc.co.uk

**C:\Windows\System32\drivers\etc\hosts for recent windows*

4

4

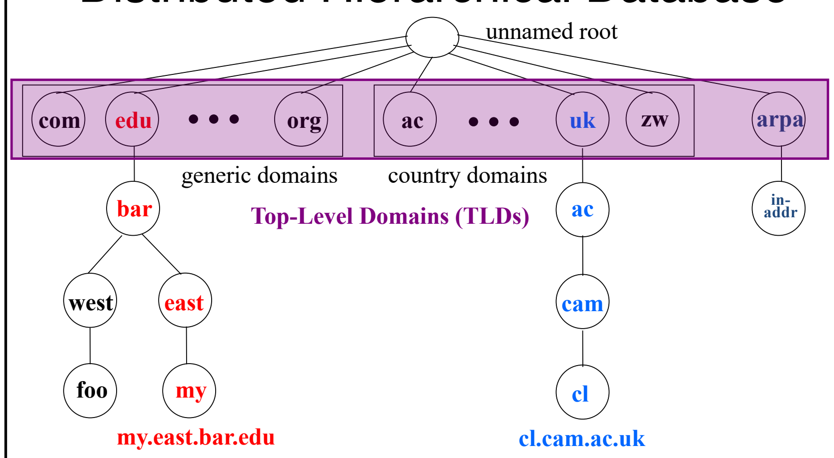
Domain Name System (DNS)

- Top of hierarchy: Root
  - Location hardwired into other servers
- Next Level: Top-level domain (TLD) servers
  - .com, .edu, etc.
  - .uk, .au, .to, etc.
  - Managed professionally
- Bottom Level: Authoritative DNS servers
  - Actually do the mapping
  - Can be maintained locally or by a service provider

5

5

Distributed Hierarchical Database

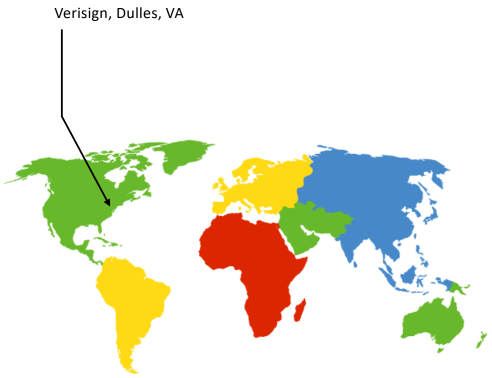


6

6

DNS Root

- Located in Virginia, USA
- How do we make the root scale?

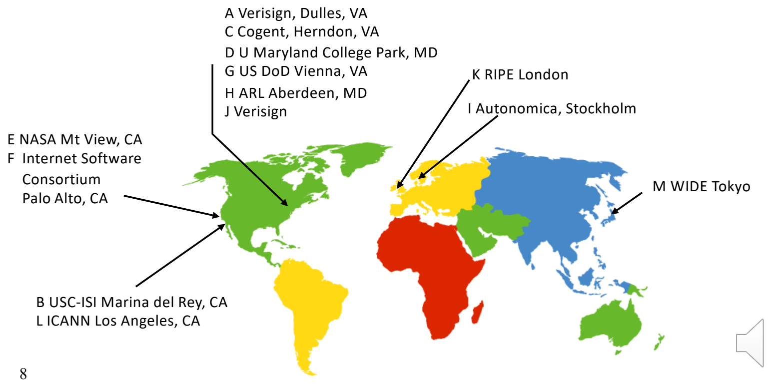


7

7

DNS Root Servers

- 13 root servers (see http://www.root-servers.org/)
  - Labeled A through M
- Does this scale?



8

8

DNS Root Servers

- 13 root servers (see http://www.root-servers.org/)
  - Labeled A through M
- Replication via any-casting (localized routing for addresses)

9

9

Using DNS

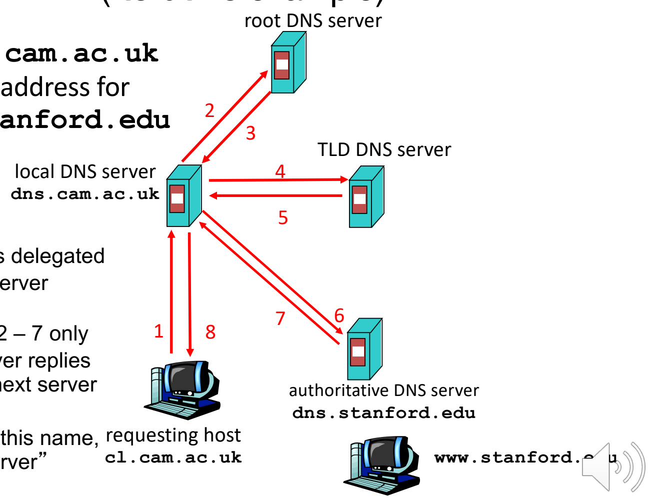
- Two components
  - Local DNS servers
  - Resolver software on hosts
- Local DNS server ("default name server")
  - Usually near the endhosts that use it
  - Local hosts configured with local server (e.g., /etc/resolv.conf) or learn server via DHCP
- Client application
  - Extract server name (e.g., from the URL)
  - Do gethostbyname() to trigger resolver code

10

10

How Does Resolution Happen? (**Iterative** example)

Host at **cl.cam.ac.uk** wants IP address for **www.stanford.edu**



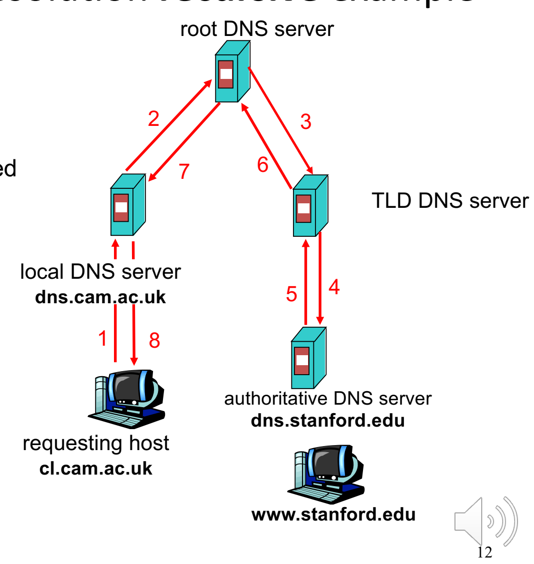
iterated query:
- Host enquiry is delegated to local DNS server
- Consider transactions 2 – 7 only
- contacted server replies with name of next server to contact
- "I don't know this name, but ask this server"

11

11

DNS name resolution **recursive** example

recursive query:
- puts burden of name resolution on contacted name server
- heavy load?

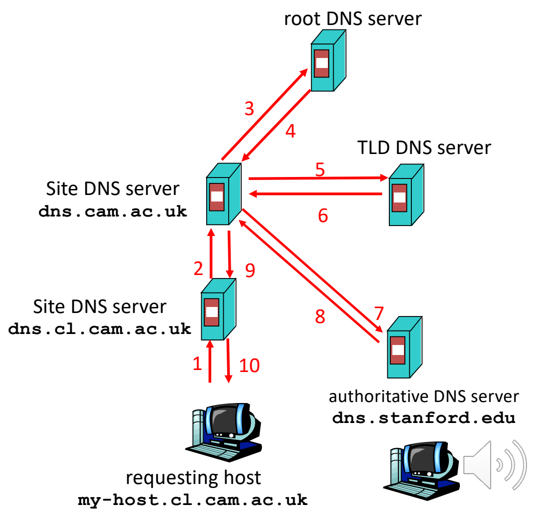


12

12

Recursive and Iterative Queries - **Hybrid** case

- Recursive query
  - Ask server to get answer for you
  - E.g., requests 1,2 and responses 9,10
- Iterative query
  - Ask server who to ask next
  - E.g., all other request-response pairs



13

13

DNS Caching

- Performing all these queries takes time
  - And all this before actual communication takes place
  - E.g., 1-second latency before starting Web download
- Caching can greatly reduce overhead
  - The top-level servers very rarely change
  - Popular sites (e.g., www.bbc.co.uk) visited often
  - Local DNS server often has the information cached
- How DNS caching works
  - DNS servers cache responses to queries
  - Responses include a "time to live" (TTL) field
  - Server deletes cached entry after TTL expires

14

14

Negative Caching

- Remember things that don't work
  - Misspellings like *bbcc.co.uk* and *www.bbc.com.uk*
  - These can take a long time to fail the first time
  - Good to remember that they don't work
  - … so the failure takes less time the next time around
- But: negative caching is optional
  - And not widely implemented

15

15

Reliability

- DNS servers are replicated (primary/secondary)
  - Name service available if at least one replica is up
  - Queries can be load-balanced between replicas
- Usually, UDP used for queries
  - Need reliability: must implement this on top of UDP
  - Spec supports TCP too, but not always implemented
- Try alternate servers on timeout
  - Exponential backoff when retrying same server
- Same identifier for all queries
  - Don't care which server responds

16

16

Invalid queries categories

From https://www.caida.org/publications/presentations/2008/wide_castro_root_servers/wide_castro_root_servers.pdf

- Unused query class:
  - Any class not in IN, CHAOS, HESIOD, NONE or ANY
- A-for-A: A-type query for a name is already a IPv4 Address
  - <IN, A, 192.16.3.0>
- Invalid TLD: a query for a name with an invalid TLD
  - <IN, MX, localhost.lan>
- Non-printable characters:
  - <IN, A, www.ra^B.us.>
- Queries with '_':
  - <IN, SRV, _ldap._tcp.dc._msdcs.SK0530-K32-1.>
- RFC 1918 PTR:
  - <IN, PTR, 171.144.144.10.in-addr.arpa.>
- Identical queries:
  - a query with the same class, type, name and id (during the whole period)
- Repeated queries:
  - a query with the same class, type and name
- Referral-not-cached:
  - a query seen with a referral previously given.

17 11

17

Invalid TLD

From https://www.caida.org/publications/presentations/2008/wide_castro_root_servers/wide_castro_root_servers.pdf

- Queries for invalid TLD represent 22% of the total traffic at the roots
  - 20.6% during DITL 2007
- Top 10 invalid TLD represent 10.5% of the total traffic
- RFC 2606 reserves some TLD to avoid future conflicts
- We propose:
  - Include some of these TLD (local, lan, home, localdomain) to RFC 2606
  - Encourage cache implementations to answer queries for RFC 2606 TLDs locally (with data or error)

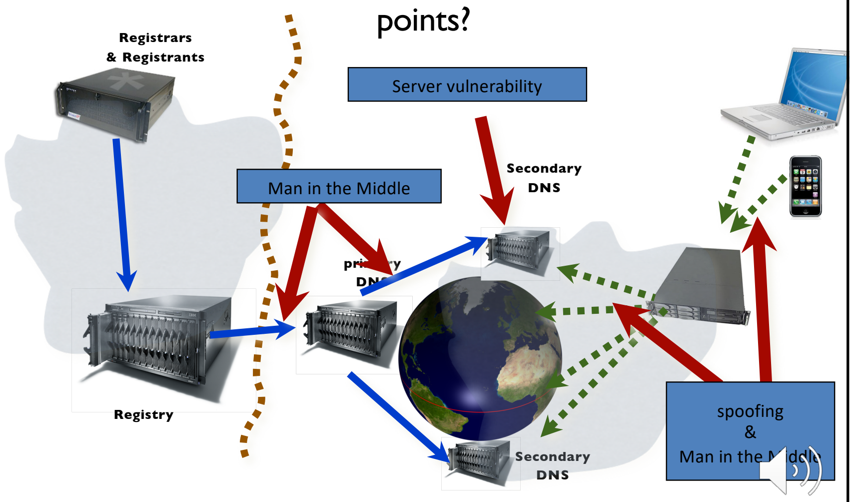
TLD	**Percentage of total queries**	
	2007	2008
local	5.018	5.098
belkin	0.436	0.781
localhost	2.205	0.710
lan	0.509	0.679
home	0.321	0.651
invalid	0.602	0.623
domain	0.778	0.550
localdomain	0.318	0.332
wpad	0.183	0.232
corp	0.150	0.231

awm22: at least WORKGROUP is no longer here!
It was the top in valid TLD for years…

9 18

18

Data flow through the DNS
Where are the vulnerable points?



19

DNS and Security

- No way to verify answers
  - Opens up DNS to many potential attacks
  - DNSSEC fixes this
- Most obvious vulnerability: recursive resolution
  - Using recursive resolution, host must trust DNS server
  - When at Starbucks, server is under their control
  - And can return whatever values it wants
- More subtle attack: Cache poisoning
  - Those "additional" records can be anything!

20

20

DNSSEC protects all these end-to-end

- provides message authentication and integrity verification through cryptographic signatures
  - You know who provided the signature
  - No modifications between signing and validation
- It does **not** provide authorization
- It does **not** provide confidentiality
- It does **not** provide protection against DDOS

21

DNSSEC in practice

- Scaling the key signing and key distribution

Solution: Using the DNS to Distribute Keys

- Distributing keys through DNS hierarchy:
  - Use one trusted key to establish authenticity of other keys
  - Building chains of trust from the root down
  - Parents need to sign the keys of their children
- Only the root key needed in ideal world
  - Parents always delegate security to child

22

22

Why is the web so successful?

- What do the web, youtube, facebook, twitter, instagram, ….. have in common?
  - The ability to self-publish
- Self-publishing that is easy, independent, *free*
- No interest in collaborative and idealistic endeavor
  - People aren't looking for Nirvana (or even Xanadu)
  - People also aren't looking for technical perfection
- Want to make their mark, and find something neat
  - Two sides of the same coin, creates synergy
  - "Performance" more important than dialogue….

23

23

Web Components

- Infrastructure:
  - Clients
  - Servers
  - Proxies
- Content:
  - Individual objects (files, etc.)
  - Web sites (coherent collection of objects)
- Implementation
  - HTML: formatting content
  - URL: naming content
  - HTTP: protocol for exchanging content

Any content not just HTML!

24

24

HTML: HyperText Markup Language

- A *Web page* has:
  - Base HTML file
  - Referenced objects (*e.g.*, images)
- HTML has several functions:
  - Format text
  - Reference images
  - Embed *hyperlinks* (HREF)

25

25

URL Syntax

protocol://hostname[:port]/directorypath/resource

protocol	http, ftp, https, smtp, rtsp, *etc.*
hostname	DNS name, IP address
port	Defaults to protocol's standard port *e.g.* http: 80 https: 443
directory path	Hierarchical, reflecting file system
resource	Identifies the desired resource Can also extend to program executions: `http://us.f413.mail.yahoo.com/ym/ShowLetter?box=%40B%40Bulk&MsgId=2604_1744106_29699_1123_1261_0_28917_3552_1289957100&Search=&Nhead=f&YY=31454&order=down&sort=date&pos=0&view=a&head=b`

26

26

HyperText Transfer Protocol (HTTP)

- Request-response protocol
- Reliance on a global namespace
- Resource *metadata*
- *Stateless*
- ASCII format (ok this changed….)

$ telnet www.cl.cam.ac.uk 80
GET /win HTTP/1.0
<blank line, i.e., CRLF>

27

27

Steps in HTTP Request

- HTTP Client initiates TCP connection to server
  - SYN
  - SYNACK
  - ACK
- Client sends HTTP request to server
  - Can be piggybacked on TCP's ACK
- HTTP Server responds to request
- Client receives the request, terminates connection
- TCP connection termination exchange

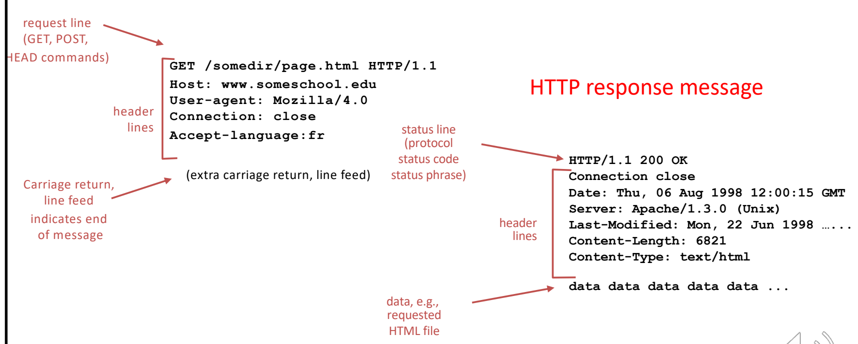
How many RTTs for a single request?

28

28

Client-Server Communication

- two types of HTTP messages: *request, response*
- HTTP request message: (GET POST HEAD ….)



29

29

Different Forms of Server Response

- Return a file
  - URL matches a file (*e.g.,* `/www/index.html`)
  - Server returns file as the response
  - Server generates appropriate response header
- Generate response dynamically
  - URL triggers a program on the server
  - Server runs program and sends output to client
- Return meta-data with no body

30

30

HTTP Resource Meta-Data

- Meta-data
  - Info *about* a resource, stored as a separate entity
- Examples:
  - Size of resource, last modification time, type of content
- Usage example: Conditional GET Request
  - Client requests object "**`If-modified-since`**"
  - If unchanged, "**`HTTP/1.1 304 Not Modified`**"
  - No body in the server's response, only a header

31

31

HTTP is *Stateless*

- Each request-response treated independently
  - Servers *not* required to retain state
- **Good**: Improves scalability on the server-side
  - Failure handling is easier
  - Can handle higher rate of requests
  - Order of requests doesn't matter
- **Bad**: Some applications need persistent state
  - Need to uniquely identify user or store temporary info
  - *e.g.*, Shopping cart, user profiles, usage tracking, …

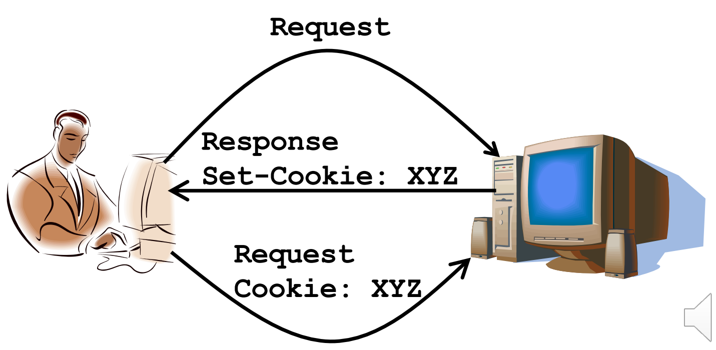
32

32

State in a Stateless Protocol:

Cookies

- *Client-side* state maintenance
  - Client stores small(?) state on behalf of server
  - Client sends state in future requests to the server
- Can provide authentication



33

33

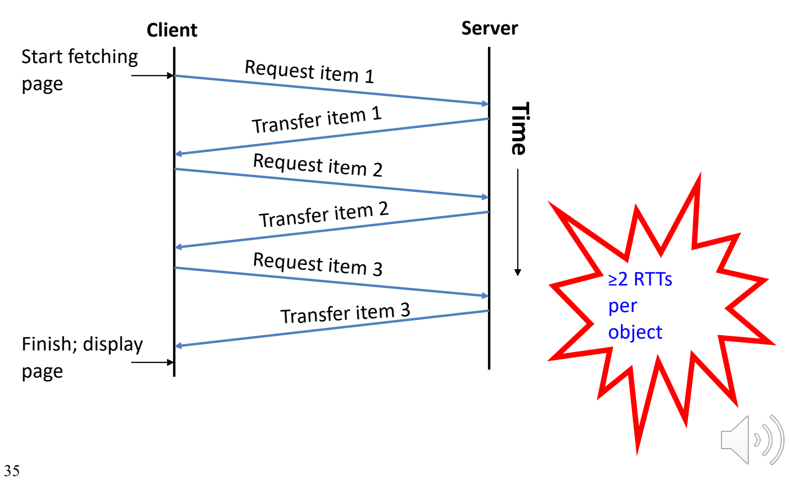
HTTP Performance

- Most Web pages have multiple objects
  - *e.g.,* HTML file and a bunch of embedded images
- How do you retrieve those objects (naively)?
  - *One item at a time*
- Put stuff in the optimal place?
  - *Where is that precisely?*
    - ***Enter the Web cache and the CDN***

34

34

Fetch HTTP Items: Stop & Wait



35

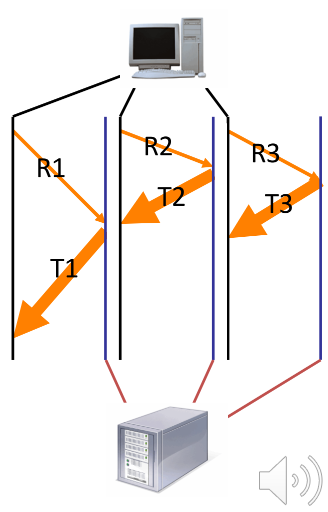
35

Improving HTTP Performance:

Concurrent Requests & Responses

- Use multiple connections *in parallel*
- Does not necessarily maintain order of responses

- Client = ☺
- Server = ☺
- Network = ☹ Why?



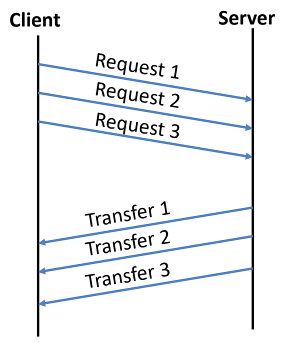
36

36

Improving HTTP Performance:

Pipelined Requests & Responses

- *Batch* requests and responses
  - Reduce connection overhead
  - Multiple requests sent in a single batch
  - Maintains order of responses
  - Item 1 always arrives before item 2
- How is this different from concurrent requests/responses?
  - Single TCP connection



37

37

Improving HTTP Performance:

Persistent Connections

- Enables multiple transfers per connection
  - Maintain TCP connection across multiple requests
  - Including transfers subsequent to current page
  - Client or server can tear down connection
- Performance advantages:
  - Avoid overhead of connection set-up and tear-down
  - Allow TCP to learn more accurate RTT estimate
  - Allow TCP congestion window to increase
  - i.e., leverage previously discovered bandwidth
- Default in HTTP/1.1

38

38

HTTP *evolution*

- 1.0 – one object per TCP: simple but slow
- Parallel connections - multiple TCP, one object each: wastes b/w, may be svr limited, out of order
- 1.1 pipelining – aggregate retrieval time: ordered, multiple objects sharing single TCP
- 1.1 persistent – aggregate TCP overhead: lower overhead in time, increase overhead at ends (e.g., when should/do you close the connection?)

39

39

Scorecard: Getting n Small Objects

Time dominated by latency

- One-at-a-time: ~2n RTT
- Persistent: ~ (n+1)RTT
- M concurrent: ~2[n/m] RTT
- Pipelined: ~2 RTT
- Pipelined/Persistent: ~2 RTT first time, RTT later

40

40

Scorecard: Getting n Large Objects

Time dominated by bandwidth

- One-at-a-time: ~ nF/B
- M concurrent: ~ [n/m] F/B
  - assuming shared with large population of users
- Pipelined and/or persistent: ~ nF/B
  - The only thing that helps is getting more bandwidth..

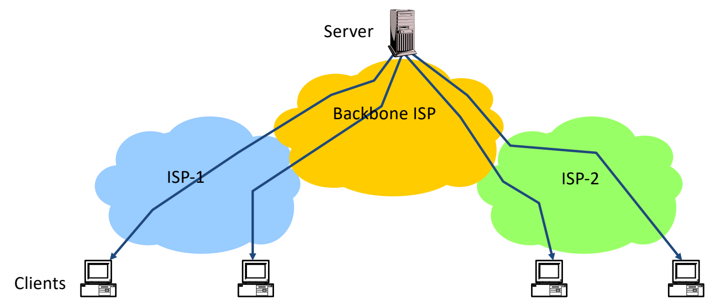
41

41

Improving HTTP Performance:

Caching

- Many clients transfer the same information
  - Generates redundant server and network load
  - Clients experience unnecessary latency



42

42

Improving HTTP Performance:

Caching: How

- Modifier to GET requests:
  - `If-modified-since` – returns "not modified" if resource not modified since specified time
- Response header:
  - `Expires` – how long it's safe to cache the resource
  - `No-cache` – ignore all caches; always get resource directly from server

43

43

Improving HTTP Performance:

Caching: Why

- Motive for placing content closer to client:
  - User gets better response time
  - Content providers get happier users
    - Time is money, really!
  - Network gets reduced load
- Why does caching work?
  - Exploits *locality of reference*
- How well does caching work?
  - Very well, up to a limit
  - Large overlap in content
  - But many unique requests

44

44

Improving HTTP Performance:

Caching on the Client

Example: Conditional GET Request

- Return resource only if it has changed at the server
  - Save server resources!

Request from client to server:

```
GET /~awm22/win HTTP/1.1
Host: www.cl.cam.ac.uk
User-Agent: Mozilla/4.03
If-Modified-Since: Sun, 27 Aug 2006 22:25:50 GMT
<CR><LF>
```

- How?
  - Client specifies "if-modified-since" time in request
  - Server compares this against "last modified" time of desired resource
  - Server returns "304 Not Modified" if resource has not changed
  - …. or a "200 OK" with the latest version otherwise

45

45

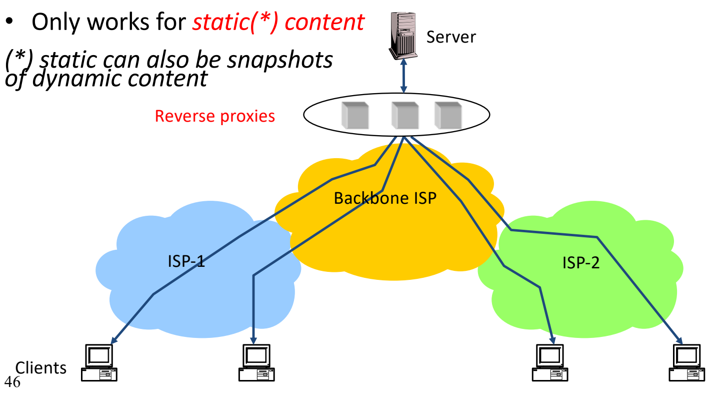
Improving HTTP Performance:

Caching with Reverse Proxies

Cache documents close to **server**
→ decrease server load

- Typically done by content providers
- Only works for *static(*) content*

(*) static can also be snapshots of dynamic content



46

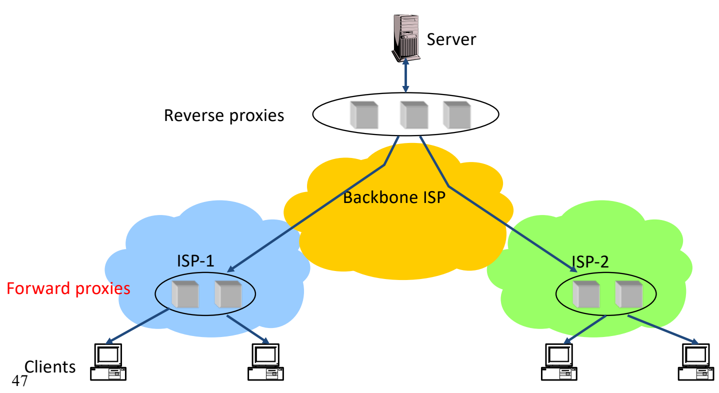
46

Improving HTTP Performance:

Caching with Forward Proxies

Cache documents close to **clients**
→ reduce network traffic and decrease latency

- Typically done by ISPs or corporate LANs



47

47

Improving HTTP Performance:

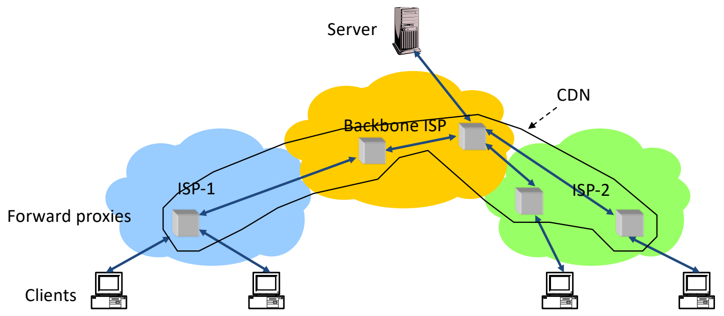
Caching w/ Content Distribution Networks

- Integrate forward and reverse caching functionality
  - One overlay network (usually) administered by one entity
  - *e.g.,* Akamai
- Provide document caching
  - **Pull:** Direct result of clients' requests
  - **Push:** Expectation of high access rate
- Also do some processing
  - Handle *dynamic* web pages
  - *Transcoding*
  - *Maybe do some security function – watermark IP*

48

48

Improving HTTP Performance:
Caching with CDNs (cont.)



49

49

Improving HTTP Performance:
CDN Example – Akamai

- Akamai creates new domain names for each client content provider.
  - e.g., *a128.g.akamai.net*
- The CDN's DNS servers are authoritative for the new domains
- The client content provider modifies its content so that embedded URLs reference the new domains.
  - "Akamaize" content
  - e.g.: *http://www.bbc.co.uk/popular-image.jpg* becomes *http://a128.g.akamai.net/popular-image.jpg*
- *Requests now sent to CDN's infrastructure…*

50

50

Hosting: Multiple Sites Per Machine

- Multiple Web sites on a single machine
  - Hosting company runs the Web server on behalf of multiple sites (*e.g.*, www.foo.com and www.bar.com)
- Problem: `GET /index.html`
  - `www.foo.com/index.html` or `www.bar.com/index.html`?
- Solutions:
  - Multiple server processes on the same machine
    - Have a separate IP address (or port) for each server
  - Include site name in HTTP request
    - Single Web server process with a single IP address
    - Client includes "Host" header (*e.g.*, `Host: www.foo.com`)
    - *Required header* with HTTP/1.1

51

51

Hosting: Multiple Machines Per Site

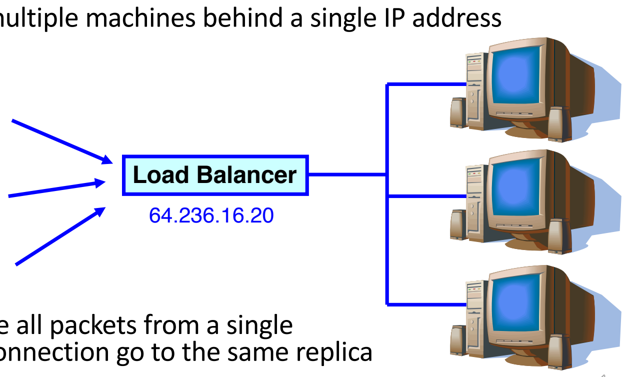
- Replicate popular Web site across many machines
  - Helps to handle the load
  - Places content closer to clients
- Helps when content isn't cacheable
- Problem: Want to direct client to particular replica
  - Balance load across server replicas
  - Pair clients with nearby servers

52

52

Multi-Hosting at Single Location

- Single IP address, multiple machines
  - Run multiple machines behind a single IP address



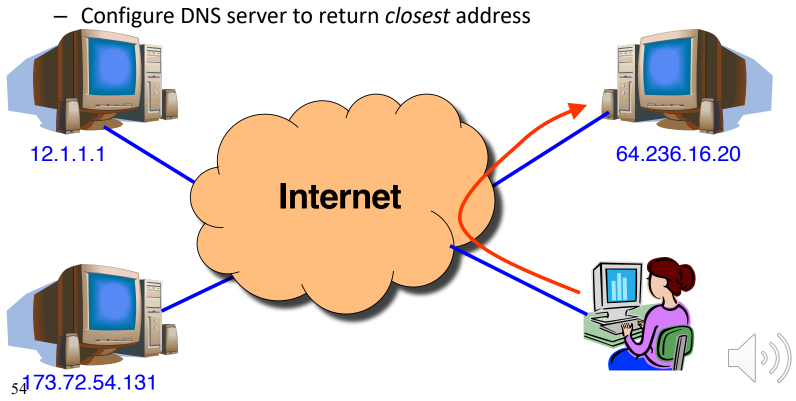
  - Ensure all packets from a single TCP connection go to the same replica

53

53

Multi-Hosting at Several Locations

- Multiple addresses, multiple machines
  - Same name but different addresses for all of the replicas
  - Configure DNS server to return *closest* address



54

54

CDN examples round-up

- CDN using DNS
  DNS has information on loading/distribution/location
- CDN using anycast
  same address from DNS name but local routes
- CDN based on rewriting HTML URLs
  (akami example just covered – akami uses DNS too)

55

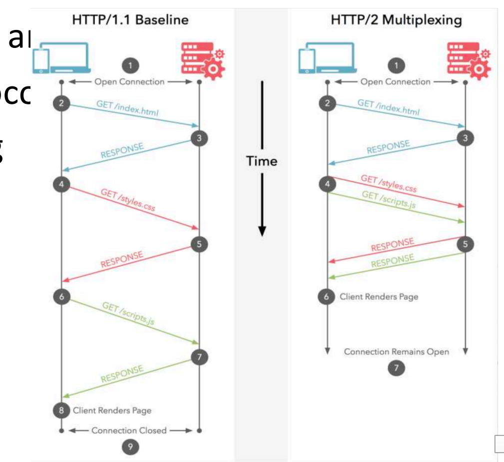
After HTTP/1.1

SPDY (speedy) and its moral successor HTTP/2
- Binary protocol
  – More efficient to parse
  – More compact on the wire
  – Much less error prone as compared
  – to textual protocols

56

After HTTP/1.1

SPDY (speedy) a
- Binary protoco
- Multiplexing
  – Interleaved



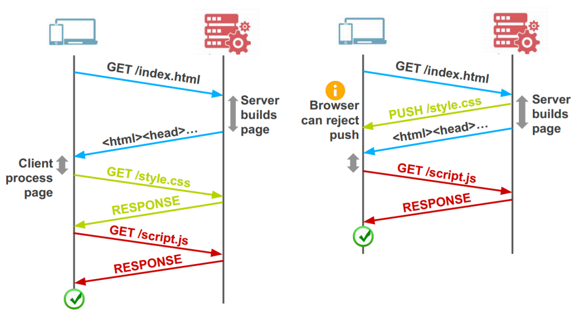
57

After HTTP/1.1

SPDY (speedy) and its moral successor HTTP/2
- Binary protocol
- Multiplexing
- Priority control over Frames
- Header Compression
- Server Push
  – Proactively push stuff to client that it will need

58

After HTTP/1.1



- Server Push
  – Proactively push stuff to client that it will need

59

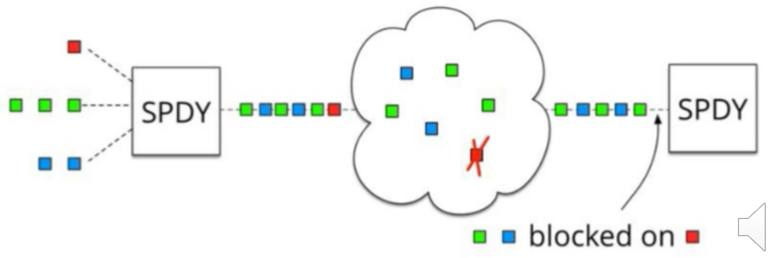
After HTTP/1.1

SPDY (speedy) and its moral successor HTTP/2
- Binary protocol
- Multiplexing
- Priority control over Frames
- Header Compression
- Server Push

60

SPDY

- SPDY + HTTP/2: One single TCP connection instead of multiple
- Downside: Head of line blocking
- In TCP, packets need to be processed in



61

Add QUIC and stir…
Quick UDP Internet Connections

Objective: Combine speed of UDP protocol with TCP's reliability

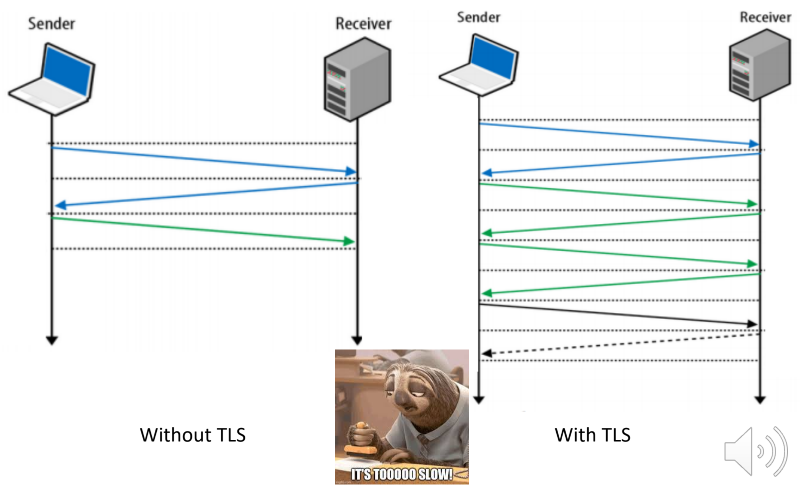
- Very hard to make changes to TCP
- *Faster to implement new protocol on top of UDP*
- Roll out features in TCP if they prove theory

QUIC:

- Reliable transport over UDP (seriously)
- Uses FEC
- Default crypto
- Restartable connections

62

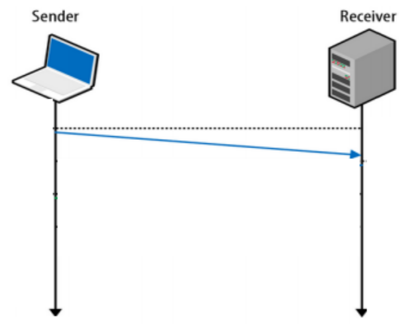
3-Way Handshake



63

UDP

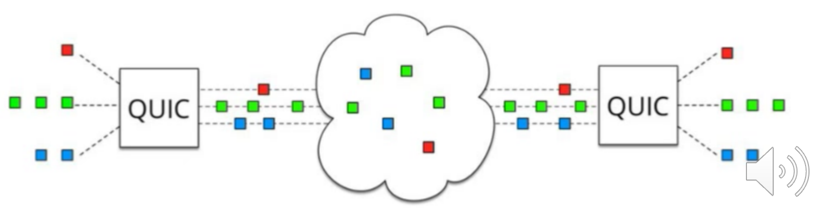
- Fire and forget
  - Less time spent to validate packets
  - Downside - no reliability, this has to be built on top of UDP



64

QUIC

- UDP does NOT depend on order of arriving packets
- Lost packets will only impact an individual resource, e.g., CSS or JS file.
- QUIC is combining best parts of HTTP/2 over UDP:
  - Multiplexing on top of non-blocking transport protocol



65

QUIC – more than just UDP

- QUIC outshines TCP under poor network conditions, shaving a full second off the Google Search page load time for the slowest 1% of connections.
- These benefits are even more apparent for video services like YouTube. Users report 30% fewer rebuffers when watching videos over QUIC.

66

Why QUIC over UDP and not a new proto

- IP proto value for new transport layer
- Change the protocol – risk the wraith of
  - Legacy code
  - Firewalls
  - Load-balancer
  - NATs (the high-priest of middlebox)

- Same problem faces any significant TCP change

Honda M. et al. “**Is it still possible to extend TCP?”,** IMC’11
https://dl.acm.org/doi/abs/10.1145/2068816.2068834

67

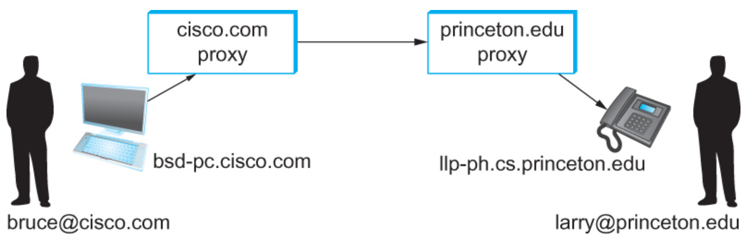
SIP – Session Initiation Protocol

Session?

Anyone smell an OSI / ISO standards document burning?

68

SIP - VoIP



Establishing communication through SIP proxies.

69

SIP?

- SIP – bringing the fun/complexity of telephony to the Internet
  - User location
  - User availability
  - User capabilities
  - Session setup
  - Session management
    - (e.g. “call forwarding”)

70

H.323 – ITU

- Why have one standard when there are at least two….

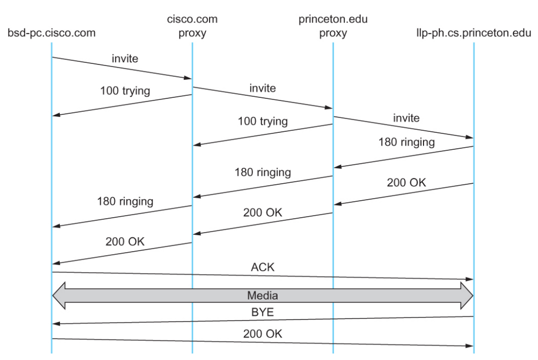
- The full H.323 is hundreds of pages
  - The protocol is known for its complexity – an ITU hallmark

- SIP is not much better

  - IETF grew up and became the ITU….

71

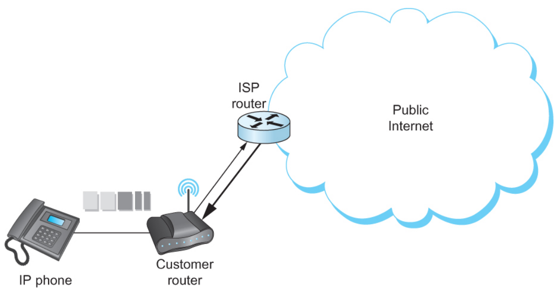
Multimedia Applications



Message flow for a basic SIP session

72

The (still?) missing piece:
Resource Allocation for Multimedia Applications



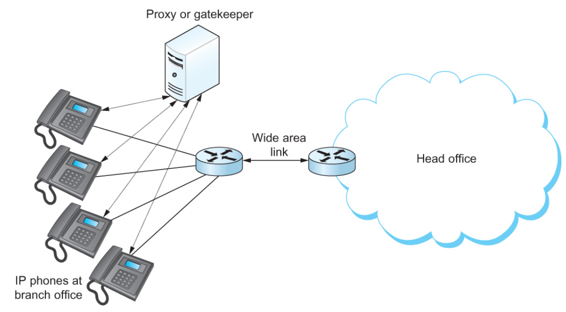
I can 'differentiate' VoIP from data but…
I can only control data going into the Internet

73

73

Multimedia Applications

- Resource Allocation for Multimedia Applications



Admission control using session control protocol.

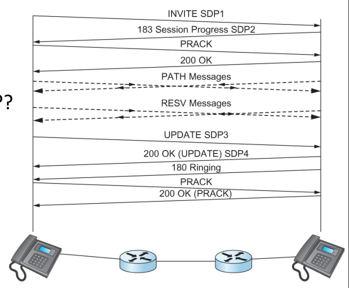
74

74

Resource Allocation for Multimedia Applications

Coming soon… ~~1995~~
~~2000~~
~~2010~~
~~2020~~
who are we kidding??

Co-ordination of SIP signaling and resource reservation.



So where does it happen?
Inside single institutions or *domains of control…..*
(Universities, Hospitals, big corp…)

What about my aDSL/CABLE/etc it combines voice and data?
Phone company **controls** the multiplexing on the line
and throughout their own network too…… everywhere else is *best effort*

75

75

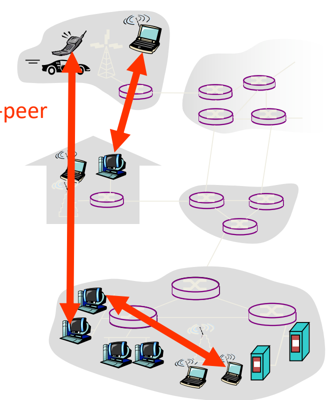
Every host is a server:
Peer-2-Peer

76

76

Pure P2P architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses



- Three topics:
  - File distribution
  - Searching for information
  - Case Study: Skype

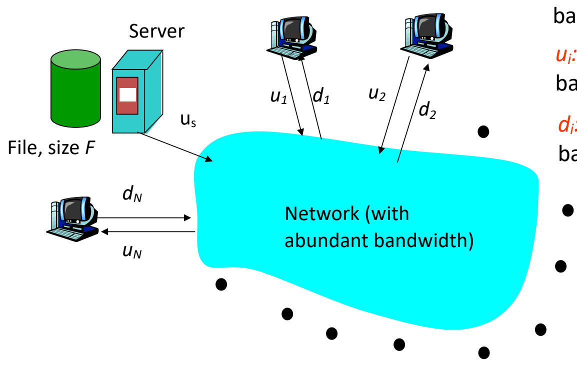
77

77

File Distribution: Server-Client vs P2P

Question : How much time to distribute file from one server to *N peers*?

u_s: server upload bandwidth
u_i: peer i upload bandwidth
d_i: peer i download bandwidth

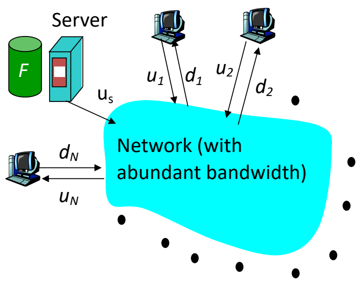


78

78

File distribution time: server-client

- server sequentially sends N copies:
  – NF/u_s time
- client i takes F/d_i time to download



Time to distribute F to N clients using client/server approach $= d_{cs} = \max \{ NF/u_s, F/\min_i(d_i) \}$

increases linearly in N (for large N)

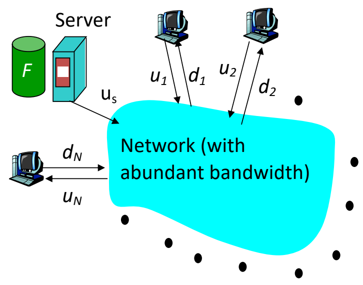
79

79

File distribution time: P2P

- server must send one copy: F/u_s time
- client i takes F/d_i time to download
- NF bits must be downloaded (aggregate)

r fastest possible upload rate: $u_s + \sum u_i$



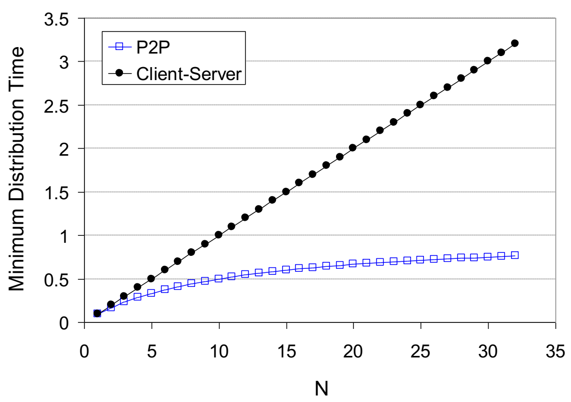
$d_{P2P} = \max \{ F/u_s, F/\min_i(d_i), NF/(u_s + \sum u_i) \}$

80

80

Server-client vs. P2P: example

Client upload rate = u, F/u = 1 hour, $u_s = 10u$, $d_{min} \geq u_s$



81

81

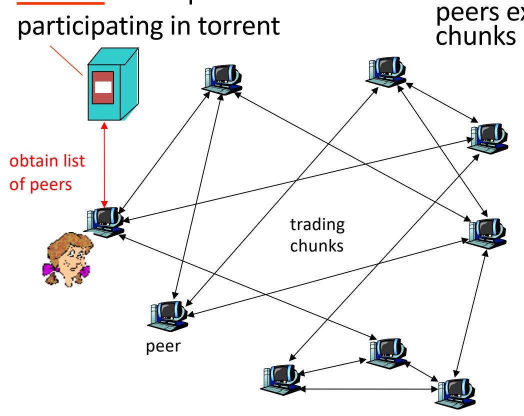
File distribution: BitTorrent*

*rather old BitTorrent

r P2P file distribution

tracker: tracks peers participating in torrent

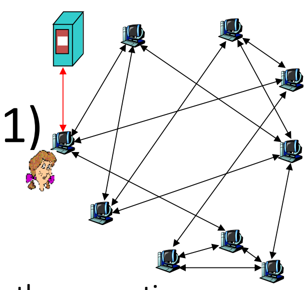
torrent: group of peers exchanging chunks of a file



82

82

BitTorrent (1)



- file divided into 256KB *chunks*.
- peer joining torrent:
  – has no chunks, but will accumulate them over time
  – registers with tracker to get list of peers, connects to subset of peers ("neighbors")
- while downloading, peer uploads chunks to other peers.
- peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain

83

83

BitTorrent (2)

Pulling Chunks

- at any given time, different peers have different subsets of file chunks
- periodically, a peer (Alice) asks each neighbor for list of chunks that they have.
- Alice sends requests for her missing chunks
  – rarest first

Sending Chunks: tit-for-tat

r Alice sends chunks to four neighbors currently sending her chunks *at the highest rate*
  ❖ re-evaluate top 4 every 10 secs

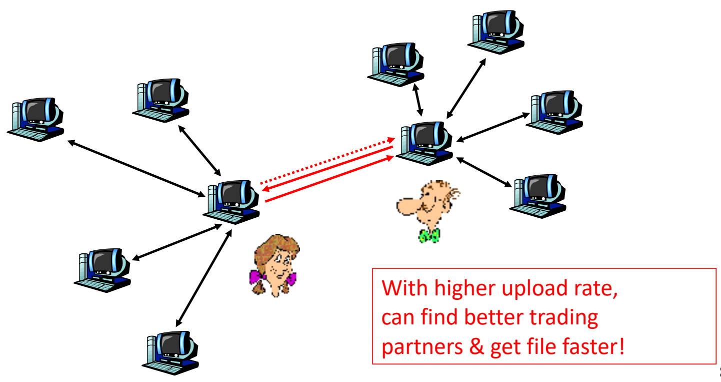
r every 30 secs: randomly select another peer, starts sending chunks
  ❖ newly chosen peer may join top 4
  ❖ "optimistically unchoke"

84

84

BitTorrent: Tit-for-tat

(1) Alice "optimistically unchokes" Bob
(2) Alice becomes one of Bob's top-four providers; Bob reciprocates
(3) Bob becomes one of Alice's top-four providers



85

85

Distributed Hash Table (DHT)

- DHT = distributed P2P database
- Database has (key, value) pairs;
  - key: ss number; value: human name
  - key: content type; value: IP address
- Peers query DB with key
  - DB returns values that match the key
- Peers can also insert (key, value) peers

86

86

Distributed Hash Table (DHT)

- DHT = distributed P2P database
- Database has (key, value) pairs;
  - key: ss number; value: human name
  - key: content type; value: IP address
- Peers query DB with key
  - DB returns values that match the key
- Peers can also insert (key, value) peers

87

87

DHT Identifiers

- Assign integer identifier to each peer in range $[0,2^n-1]$.
  - Each identifier can be represented by n bits.
- Require each key to be an integer in same range.
- To get integer keys, hash original key.
  - eg, key = h("Game of Thrones season 29")
  - This is why they call it a distributed "hash" table

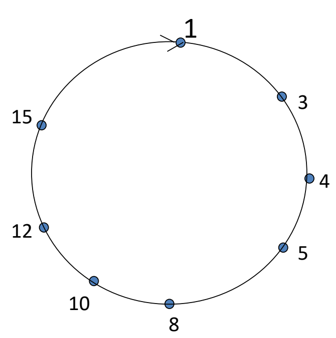
88

How to assign keys to peers?

- Central issue:
  - Assigning (key, value) pairs to peers.
- Rule: assign key to the peer that has the closest ID.
- Convention in lecture: closest is the immediate successor of the key.
- Ex: n=4; peers: 1,3,4,5,8,10,12,14;
  - key = 13, then successor peer = 14
  - key = 15, then successor peer = 1

89

Circular DHT (1)

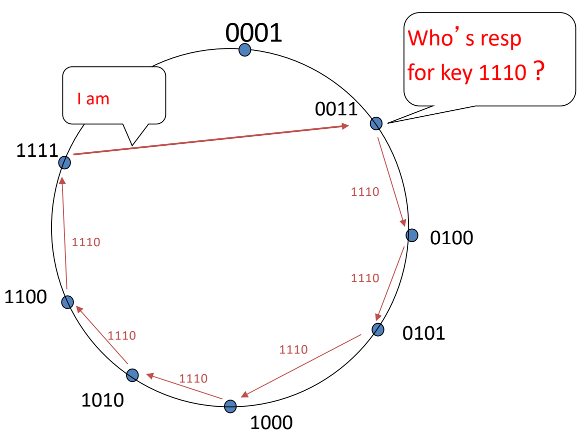


- Each peer *only* aware of immediate successor and predecessor.
- "Overlay network" – logical structure

90

Circle DHT (2)

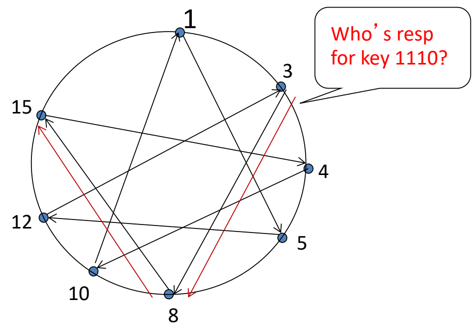
O(N) messages on avg to resolve query, when there are N peers



Define closest as closest successor

91

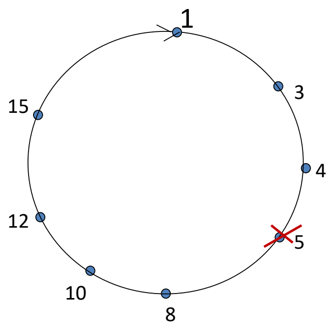
Circular DHT with Shortcuts



- Each peer keeps track of IP addresses of predecessor, successor, short cuts.
- Reduced from 6 to 2 messages.
- Possible to design shortcuts so O(log N) neighbors, O(log N) messages in query

92

Peer Churn



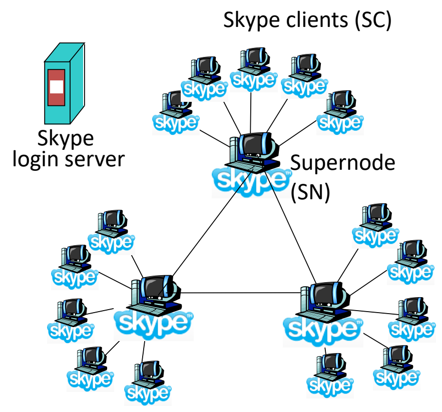
•To handle peer churn, require each peer to know the IP address of its two successors.
• Each peer periodically pings its two successors to see if they are still alive.

- Peer 5 abruptly leaves
- Peer 4 detects; makes 8 its immediate successor; asks 8 who its immediate successor is; makes 8's immediate successor its second successor.
- What if peer 13 wants to join?

93

P2P Case study: Skype (pre-Microsoft)

- inherently P2P: pairs of users communicate.
- proprietary application-layer protocol (inferred via reverse engineering)
- hierarchical overlay with SNs
- Index maps usernames to IP addresses; distributed over SNs

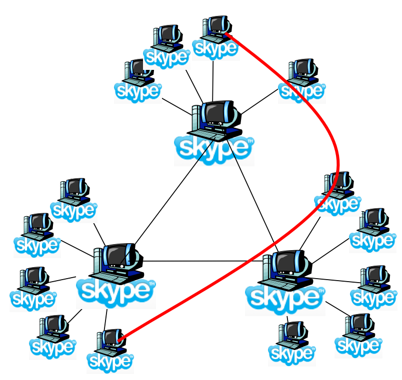


94

94

Peers as relays

- Problem when both Alice and Bob are behind "NATs".
  - NAT prevents an outside peer from initiating a call to insider peer
- Solution:
  - Using Alice's and Bob's SNs, Relay is chosen
  - Each peer initiates session with relay.
  - Peers can now communicate through NATs via relay



95

95

Summary.

- Applications have protocols too
- We covered examples from
  - Traditional Applications (web)
  - Scaling and Speeding the web (CDN/Cache tricks)
- Infrastructure Services (DNS)
  - Cache and Hierarchy
- Multimedia Applications (SIP)
  - Extremely hard to do better than worst-effort
- P2P Network examples

96

96