Computer Networking

Michaelmas/Lent Term
M/W/F 11:00-12:00
LT1 in Gates Building

Slide Set 2

Andrew W. Moore

andrew.moore@cl.cam.ac.uk

2014-2015

Topic 4: Network Layer

- understand principles behind network layer services:
  - network layer service models
  - forwarding versus routing (versus switching)
  - how a router works
  - routing (path selection)
  - IPv6
- For the most part, the Internet is our example – again.

2

Name: a *something*

Address: Where a *something* is

Routing: How do I get to the *something*

3

## Addressing (at a conceptual level)

- Assume all hosts have unique IDs

- No particular structure to those IDs

- Later in topic I will talk about real IP addressing

- Do I route on location or identifier?

- If a host moves, should its address change?
  - If not, how can you build scalable Internet?
  - If so, then what good is an address for identification?

4   4
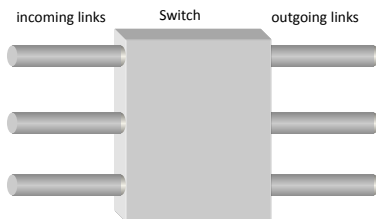

Packets (at a conceptual level)

- Assume packet headers contain:
  - Source ID, Destination ID, and perhaps other information

Destination Identifier
Source Identifier
Payload

Why include this?
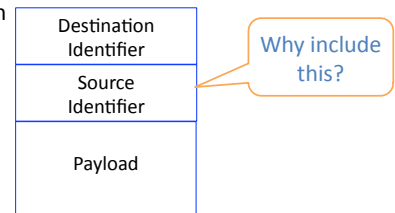
5

Switches/Routers

- Multiple ports (attached to other switches or hosts)

incoming links   Switch   outgoing links

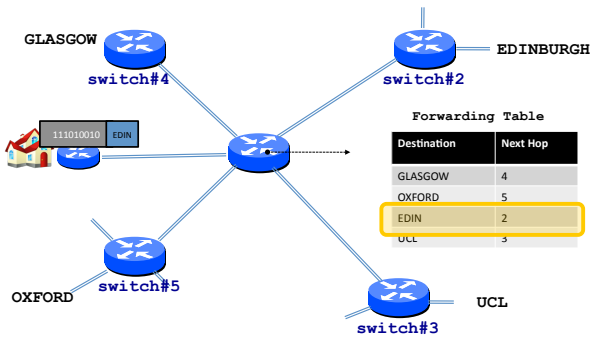- Ports are typically duplex (incoming and outgoing)

6

A Variety of Networks

- ISPs: carriers
  - Backbone
  - Edge
  - Border (to other ISPs)
- Enterprises: companies, universities
  - Core
  - Edge
  - Border (to outside)
- Datacenters: massive collections of machines
  - Top-of-Rack
  - Aggregation and Core
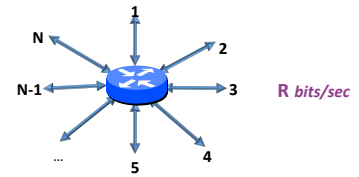 - Border (to outside)

7

Switches forward packets

GLASGOW switch#4

EDINBURGH switch#2

111010010 EDIN

Forwarding Table

Destination	Next Hop
GLASGOW	4
OXFORD	5
EDIN	2
UCL	3

OXFORD switch#5

UCL switch#3

8

Router definitions

N, N-1, 1, 2, 3, 4, 5, ...

R *bits/sec*

- N = number of external router "ports"
- R = speed ("line rate") of a port
- Router capacity = N x R

Networks and routers

JANET

home, small business

AT&T

core

edge (enterprise) INTEL

edge (ISP)

core

MIT
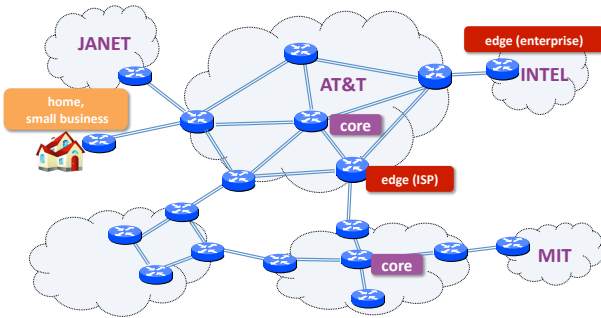
Examples of routers (core)

Cisco CRS
- R=10/40/100 Gbps
- NR = 922 Tbps
- Netflix: 0.7GB per hour (1.5Mb/s)
- ~600 million concurrent Netflix users

72 racks, >1MW

Examples of routers (edge)

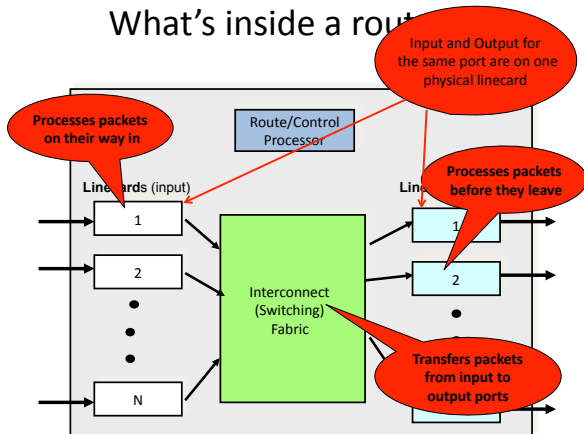**Cisco ASR**
- R=1/10/40 Gbps
- NR = 120 Gbps
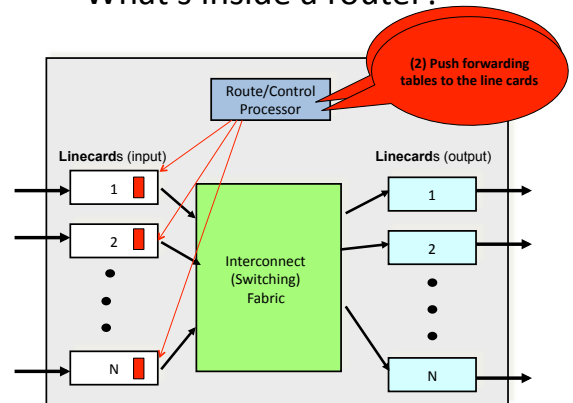
Examples of routers (small business)
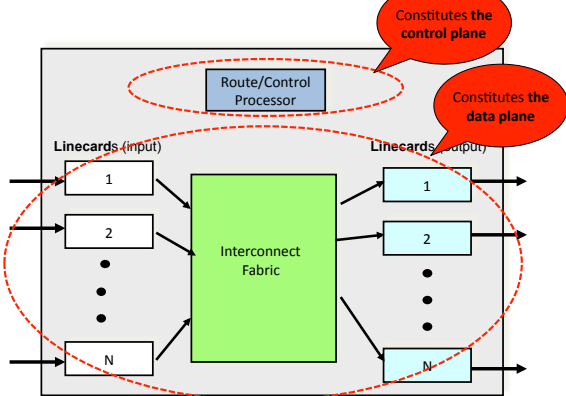
Cisco 3945E
- R = 10/100/1000 Mbps
- NR < 10 Gbps

What's inside a router?

Processes packets on their way in

Input and Output for the same port are on one physical linecard

Route/Control Processor

Processes packets before they leave

Linecards (input)

Interconnect (Switching) Fabric

Transfers packets from input to output ports

What's inside a router?

(2) Push forwarding tables to the line cards

Route/Control Processor

Linecards (input)

Interconnect (Switching) Fabric

Linecards (output)

What's inside a router?

Constitutes **the control plane**

Route/Control Processor

Constitutes **the data plane**

Linecards (input)

Interconnect Fabric
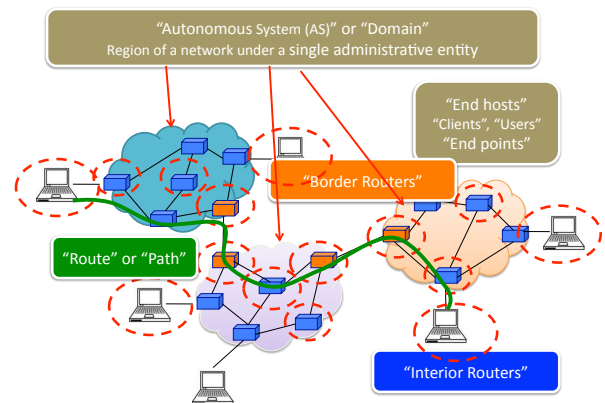
Linecards (output)

Forwarding Decisions

- When packet arrives..
  - Must decide which outgoing port to use
  - In single transmission time
  - Forwarding decisions must be *__simple__*

- Routing state dictates where to forward packets
  - Assume decisions are **deterministic**

- *Global routing state* means collection of routing state in each of the routers
  - Will focus on where this routing state comes from
  - But first, a few preliminaries….
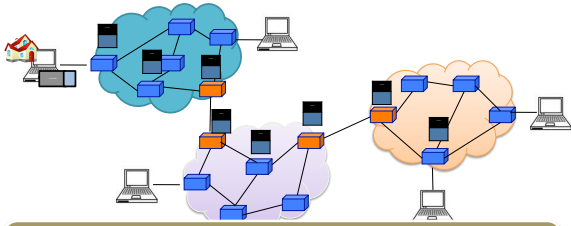
17

Forwarding vs Routing

- Forwarding: "data plane"
  - Directing a data packet to an outgoing link
  - Individual router using routing state
- Routing: "control plane"
  - Computing paths the packets will follow
  - Routers talking amongst themselves
  - Jointly creating the routing state
- Two very different timescales….

18

"Autonomous System (AS)" or "Domain"
Region of a network under a single administrative entity

"End hosts"
"Clients", "Users"
"End points"

"Border Routers"

"Route" or "Path"

"Interior Routers"

19

Context and Terminology

Internet routing protocols are responsible for constructing and updating the forwarding tables at routers

Routing Protocols

- Routing protocols implement the core function of a network
  - Establish paths between nodes
  - Part of the network's "control plane"

- Network modeled as a graph
  - Routers are graph vertices
  - Links are edges
  - Edges have an associated "cost"
    - e.g., distance, loss
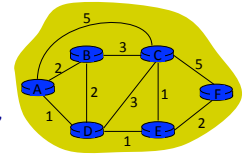
- Goal: compute a "good" path from source to destination
  - "good" usually means the shortest (least cost) path

Internet Routing

- Internet Routing works at two levels

- Each AS runs an intra-domain routing protocol that establishes routes within its domain
  - (AS -- region of network under a single administrative entity)
  - Link State, e.g., Open Shortest Path First (OSPF)
  - Distance Vector, e.g., Routing Information Protocol (RIP)

- ASes participate in an inter-domain routing protocol that establishes routes between domains
  - Path Vector, e.g., Border Gateway Protocol (BGP)

Addressing (for now)

- Assume each host has a unique ID (address)

- No particular structure to those IDs

- Later in course will talk about real IP addressing

Outline

- Link State
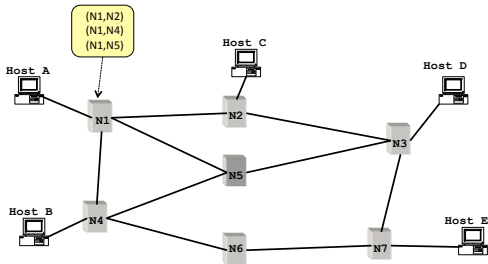- Distance Vector
- Routing: goals and metrics (if time)

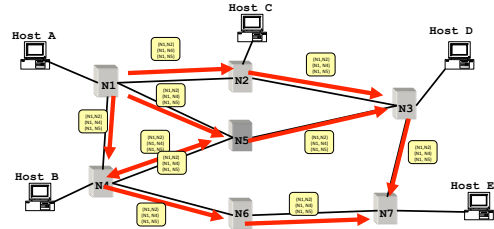## Link-State

Link State Routing

- Each node maintains its local "link state" (LS)
  - i.e., a list of its directly attached links and their costs


Link State Routing

-
- Each node floods its local link state
  - on receiving a new LS message, a router forwards the message to all its neighbors other than the one it received the message from

---

Link State Routing

- Each node maintains its local "link state" (LS)
- Each node floods its local link state
- Hence, each node learns the entire network topology
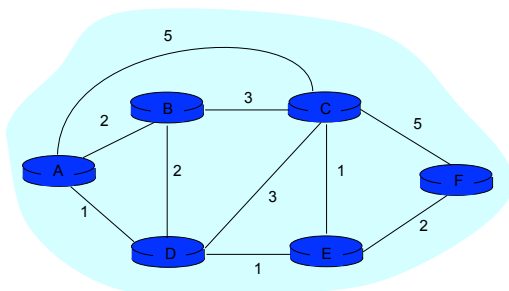 - Can use Dijkstra's to compute the shortest paths between nodes

---
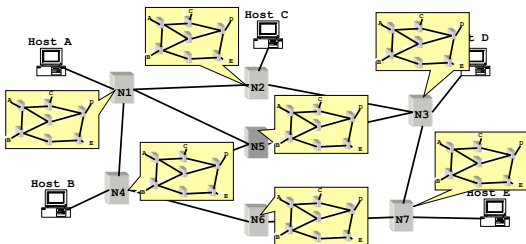
Dijkstra's Shortest Path Algorithm

- INPUT:
  - Network topology (graph), with link costs

- OUTPUT:
  - Least cost paths from one node to all other nodes

- Iterative: after *k* iterations, a node knows the least cost path to its *k* closest neighbors
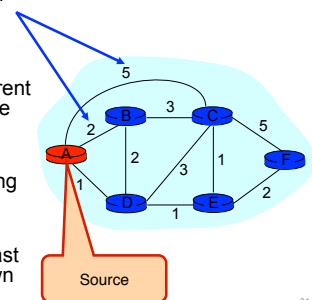

Example

Notation

- c(i,j): link cost from node *i* to *j*; cost is infinite if not direct neighbors; ≥ 0

- D(v): total cost of the current least cost path from source to destination *v*

- p(v): *v*'s predecessor along path from source to *v*

- S: set of nodes whose least cost path definitively known
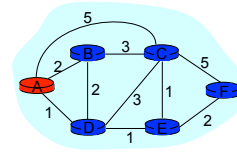

Source

Dijkstra's Algorithm

```
1  Initialization:
2    S = {A};
3    for all nodes v
4      if v adjacent to A
5        then D(v) = c(A,v);
6        else D(v) = ∞;
7
8  Loop
9    find w not in S such that D(w) is a minimum;
10   add w to S;
11   update D(v) for all v adjacent to w and not in S:
12     if  D(w) + c(w,v) < D(v) then
             // w gives us a shorter path to v than we've found so far
13       D(v) = D(w) + c(w,v); p(v) = w;
14 until all nodes in S;
```

- c(i,j): link cost from node *i* to *j*
- D(v): current cost source → v
- p(v): v's predecessor along path from source to v
- S: set of nodes whose least cost path definitively known

32

Example: Dijkstra's Algorithm

Step	set S	D(B),p(B)	D(C),p(C)	D(D),p(D)	D(E),p(E)	D(F),p(F)
0	A	2,A	5,A	1,A	∞	∞
1						
2						
3						
4						
5						

```
1  Initialization:
2    S = {A};
3    for all nodes v
4      if v adjacent to A
5        then D(v) = c(A,v);
6        else D(v) = ∞;
…
```

33

Example: Dijkstra's Algorithm
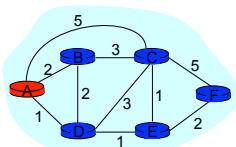
Step	set S	D(B),p(B)	D(C),p(C)	D(D),p(D)	D(E),p(E)	D(F),p(F)
0	A	2,A	5,A	1,A	∞	∞
1						
2						
3						
4						
5						

```
…
8  Loop
9    find w not in S s.t. D(w) is a minimum;
10   add w to S;
11   update D(v) for all v adjacent
        to w and not in S:
12   If D(w) + c(w,v) < D(v) then
13       D(v) = D(w) + c(w,v); p(v) = w;
14   until all nodes in S;
```

34

Example: Dijkstra's Algorithm

Step	set S	D(B),p(B)	D(C),p(C)	D(D),p(D)	D(E),p(E)	D(F),p(F)
0	A	2,A	5,A	1,A	∞	∞
1	AD					
2						
3						
4						
5						

```
…
8  Loop
9    find w not in S s.t. D(w) is a minimum;
10   add w to S;
11   update D(v) for all v adjacent
        to w and not in S:
12   If D(w) + c(w,v) < D(v) then
13       D(v) = D(w) + c(w,v); p(v) = w;
14   until all nodes in S;
```
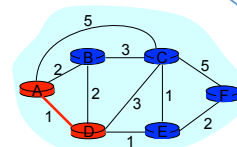
35

Example: Dijkstra's Algorithm

Step	set S	D(B),p(B)	D(C),p(C)	D(D),p(D)	D(E),p(E)	D(F),p(F)
0	A	2,A	5,A	1,A	∞	∞
1	AD		4,D		2,D	
2						
3						
4						
5						

```
…
8  Loop
9    find w not in S s.t. D(w) is a minimum;
10   add w to S;
11   update D(v) for all v adjacent
        to w and not in S:
12   If D(w) + c(w,v) < D(v) then
13       D(v) = D(w) + c(w,v); p(v) = w;
14   until all nodes in S;
```
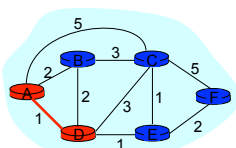
36

Example: Dijkstra's Algorithm

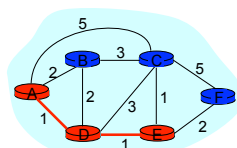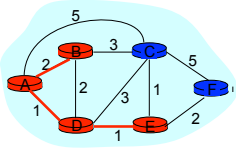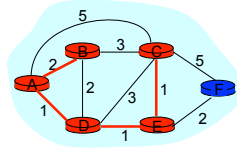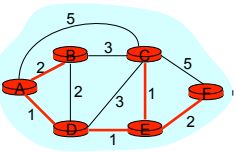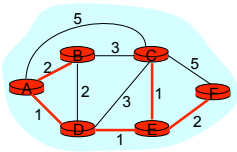| Step	set S	D(B),p(B)	D(C),p(C)	D(D),p(D)	D(E),p(E)	D(F),p(F)
0	A	2,A	5,A	1,A	∞	∞
1	AD		4,D		2,D	
2	ADE		3,E			4,E
3						
4						
5						

```
…
8  Loop
9    find w not in S s.t. D(w) is a minimum;
10   add w to S;
11   update D(v) for all v adjacent
        to w and not in S:
12   If D(w) + c(w,v) < D(v) then
13       D(v) = D(w) + c(w,v); p(v) = w;
14   until all nodes in S;
```

37

Example: Dijkstra's Algorithm

Step	set S	D(B),p(B)	D(C),p(C)	D(D),p(D)	D(E),p(E)	D(F),p(F)
0	A	2,A	5,A	1,A	∞	∞
1	AD		4,D		2,D	
2	ADE		3,E			4,E
3	ADEB					
4						
5						

...
8 **Loop**
9   find **w** not in **S** s.t. D(w) is a minimum;
10   add **w** to **S**;
11   update D(v) for all **v** adjacent
     to **w** and not in **S**:
12 If D(w) + c(w,v) < D(v) then
13   D(v) = D(w) + c(w,v); p(v) = w;
14   **until all nodes in S;**

Example: Dijkstra's Algorithm

Step	set S	D(B),p(B)	D(C),p(C)	D(D),p(D)	D(E),p(E)	D(F),p(F)
0	A	2,A	5,A	1,A	∞	∞
1	AD		4,D		2,D	
2	ADE		3,E			4,E
3	ADEB					
4	ADEBC					
5						

...
8 **Loop**
9   find **w** not in **S** s.t. D(w) is a minimum;
10   add **w** to **S**;
11   update D(v) for all **v** adjacent
     to **w** and not in **S**:
12 If D(w) + c(w,v) < D(v) then
13   D(v) = D(w) + c(w,v); p(v) = w;
14   **until all nodes in S;**

Example: Dijkstra's Algorithm

Step	set S	D(B),p(B)	D(C),p(C)	D(D),p(D)	D(E),p(E)	D(F),p(F)
0	A	2,A	5,A	1,A	∞	∞
1	AD		4,D		2,D	
2	ADE		3,E			4,E
3	ADEB					
4	ADEBC					
5	ADEBCF					

...
8 **Loop**
9   find **w** not in **S** s.t. D(w) is a minimum;
10   add **w** to **S**;
11   update D(v) for all **v** adjacent
     to **w** and not in **S**:
12 If D(w) + c(w,v) < D(v) then
13   D(v) = D(w) + c(w,v); p(v) = w;
14   **until all nodes in S;**

Example: Dijkstra's Algorithm

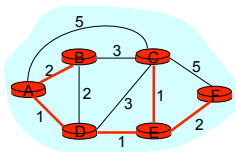| Step	set S	D(B),p(B)	D(C),p(C)	D(D),p(D)	D(E),p(E)	D(F),p(F)
0	A	2,A	5,A	1,A	∞	∞
1	AD		4,D		2,D	
2	ADE		3,E			4,E
3	ADEB					
4	ADEBC					
5	ADEBCF					

To determine path A → C (say),
work backward from C via p(v)

The Forwarding Table

- Running Dijkstra at node A gives the shortest path from A to all destinations
- We then construct the *forwarding table*

Destination	Link
B	(A,B)
C	(A,D)
D	(A,D)
E	(A,D)
F	(A,D)

Issue #1: Scalability

- How many messages needed to flood link state messages?
  - O(N x E), where N is #nodes; E is #edges in graph

- Processing complexity for Dijkstra's algorithm?
  - $O(N^2)$, because we check all nodes w not in S at each iteration and we have O(N) iterations
  - more efficient implementations: O(N log(N))

- How many entries in the LS topology database? O(E)

- How many entries in the forwarding table? O(N)

Issue#2: Transient Disruptions

- Inconsistent link-state database
  - Some routers know about failure before others
  - The shortest paths are no longer consistent
  - ~~sient~~ forwa~~

Loop!

A and D think that this is the path to C

E thinks that this is the path to C
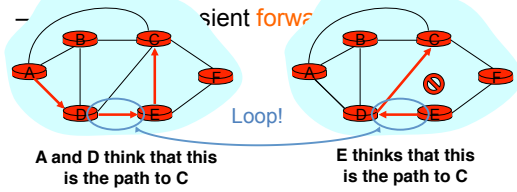
Distance Vector

Learn-By-Doing

Let's try to collectively develop
distance-vector routing from first principles

Experiment

- Your job: find the (route to) the youngest person in the room

- Ground Rules
  - **You may not** leave your seat, nor shout loudly across the class
  - **You may** talk with your immediate neighbors
    (N-S-E-W only)
   (hint: "exchange updates" with them)

- At the end of 5 minutes, I will pick a victim and ask:
  - who is the youngest person in the room? (date&name)
  - which one of your neighbors first told you this info.?

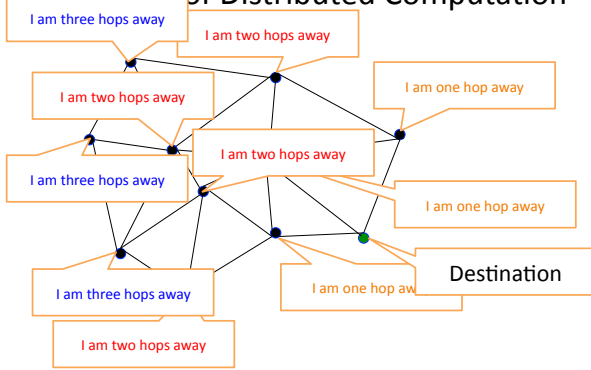---

Go!

Distance-Vector

Example of Distributed Computation

I am three hops away
I am two hops away
I am two hops away
I am one hop away
I am two hops away
I am three hops away
I am one hop away
I am three hops away
I am one hop away
Destination
I am two hops away

Distance Vector Routing

- Each router knows the links to its neighbors
  - Does *not* flood this information to the whole network
- Each router has provisional "shortest path" to every other router
  - E.g.: Router A: "I can get to router B with cost 11"
- Routers exchange this distance vector information with their neighboring routers
  - Vector because one entry per destination
- Routers look over the set of options offered by their neighbors and select the best one
- Iterative process converges to set of shortest paths

A few other inconvenient truths

- What if we use a non-additive metric?
  - E.g., maximal capacity

- What if routers don't use the same metric?
  - I want low delay, you want low loss rate?

- What happens if nodes lie?
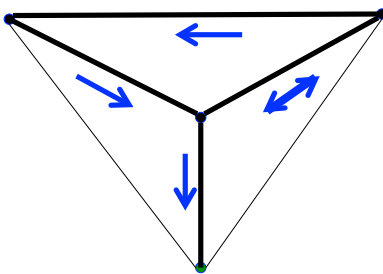

Can You Use Any Metric?

- I said that we can pick any metric. Really?
- What about maximizing capacity?

What Happens Here?

Problem:"cost" does not change around loop

Additive measures avoid this problem!

No agreement on metrics?
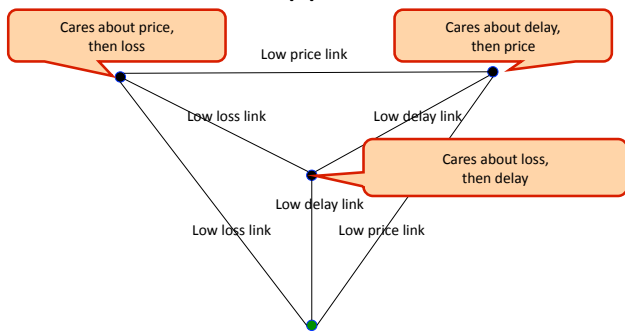
- If the nodes choose their paths according to different criteria, then bad things might happen
- Example
  - Node A is minimizing latency
  - Node B is minimizing loss rate
  - Node C is minimizing price
- Any of those goals are fine, if globally adopted
  - Only a problem when nodes use different criteria

- Consider a routing algorithm where paths are described by delay, cost, loss

What Happens Here?

Cares about price, then loss

Low price link

Cares about delay, then price

Low loss link

Low delay link

Cares about loss, then delay

Low delay link

Low loss link

Low price link

Must agree on loop-avoiding metric

- When all nodes minimize same metric

- And that metric increases around loops

- Then process is guaranteed to converge

What happens when routers lie?

- What if a router claims a 1-hop path to everywhere?

- All traffic from nearby routers gets sent there

- How can you tell if they are lying?

- Can this happen in real life?
  - It has, several times….

Link State vs. Distance Vector

- Core idea
  - LS: tell all nodes about your immediate neighbors
  - DV: tell your immediate neighbors about (your least cost distance to) all nodes

---

Link State vs. Distance Vector

- LS: each node learns the complete network map; each node computes shortest paths independently and in parallel

- DV: no node has the complete picture; nodes cooperate to compute shortest paths in a distributed manner

→LS has higher messaging overhead
→LS has higher processing complexity
→LS is less vulnerable to looping

Link State vs. Distance Vector

Message complexity
- LS: O(NxE) messages;
  - N is #nodes; E is #edges
- DV: O(#Iterations x E)
  - where #Iterations is ideally O(network diameter) but varies due to routing loops or the count-to-infinity problem

Processing complexity
- LS: O(N^2)
- DV: O(#Iterations x N)

Robustness: what happens if router malfunctions?
- LS:
  - node can advertise incorrect *link* cost
  - each node computes only its *own* table
- DV:
  - node can advertise incorrect *path* cost
  - each node's table used by others; error propagates through network

Routing: Just the Beginning

- Link state and distance-vector are the deployed routing paradigms for intra-domain routing

- Inter-domain routing (BGP)
  - more Part II (Principles of Communications)
  - A version of DV

62

What are desirable goals for a routing solution?

- "Good" paths (least cost)
- Fast convergence after change/failures
  - no/rare loops
- Scalable
  - #messages
  - table size
  - processing complexity
- Secure
- Policy
- Rich metrics (more later)

63

Delivery models

- What if a node wants to send to more than one destination?
  - broadcast: send to all
  - multicast: send to all members of a group
  - anycast: send to any member of a group

- What if a node wants to send along more than one path?

64

Metrics

- Propagation delay
- Congestion
- Load balance
- Bandwidth (available, capacity, maximal, bbw)
- Price
- Reliability
- Loss rate
- Combinations of the above

In practice, operators set abstract "weights" (much like our costs); how exactly is a bit of a black art
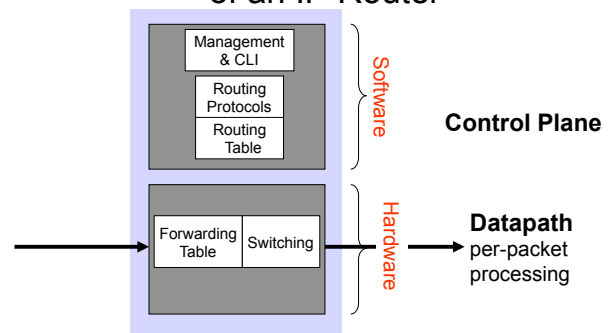
65

From Routing back to Forwarding

- Routing: "control plane"
  - Computing paths the packets will follow
  - Routers talking amongst themselves
  - Jointly creating the routing state
- Forwarding: "data plane"
  - Directing a data packet to an outgoing link
  - Individual router using routing state
- Two very different timescales….

66

Basic Architectural Components of an IP Router

67

Per-packet processing in an IP Router

1. Accept packet arriving on an incoming link.
2. Lookup packet destination address in the forwarding table, to identify outgoing port(s).
3. Manipulate packet header: e.g., decrement TTL, update header checksum.
4. Send packet to the outgoing port(s).
5. Buffer packet in the queue.
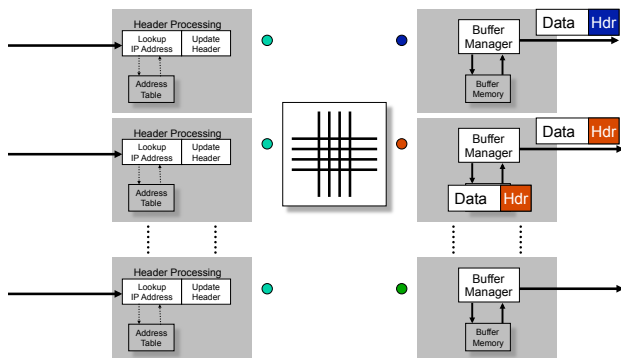6. Transmit packet onto outgoing link.

Generic Router Architecture
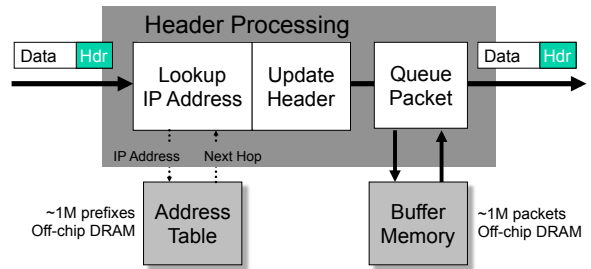
Generic Router Architecture

Forwarding tables

| IP address | � 32 bits wide → ~ 4 billion unique address |

Naïve approach:
One entry per address

Entry	Destination	Port
1	0.0.0.0	1
2	0.0.0.1	2
⋮	⋮	⋮
2^{32}	255.255.255.255	12

~ 4 billion entries

Improved approach:
Group entries to reduce table size

Entry	Destination	Port
1	0.0.0.0 – 127.255.255.255	1
2	128.0.0.1 – 128.255.255.255	2
⋮	⋮	⋮
50	248.0.0.0 – 255.255.255.255	12

IP addresses as a line

Entry	Destination	Port
1	Cambridge	1
2	Oxford	2
3	Europe	3
4	USA	4
5	Everywhere (default)	5

Longest Prefix Match (LPM)

Entry	Destination	Port
1	Cambridge	1
2	Oxford	2
3	Europe	3
4	USA	4
5	Everywhere (default)	5

Universities — entries 1, 2
Continents — entries 3, 4
Planet — entry 5

Matching entries:
- Cambridge   Most specific
- Europe
- Everywhere

To: Cambridge | Data

Longest Prefix Match (LPM)

Entry	Destination	Port
1	Cambridge	1
2	Oxford	2
3	Europe	3
4	USA	4
5	Everywhere (default)	5

} Universities
} Continents
Planet

Matching entries:
- Europe   Most specific
- Everywhere

To: France	Data

74

Implementing Longest Prefix Match

Entry	Destination	Port	
1	Cambridge	1	Searching
2	Oxford	2	
3	Europe	3	
4	USA	4	FOUND
5	Everywhere (default)	5	

Most specific
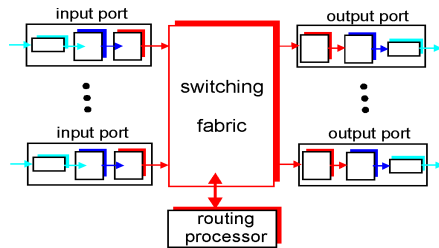↓
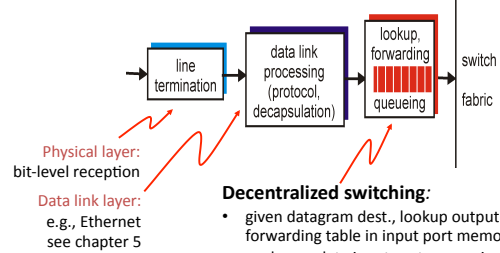Least specific
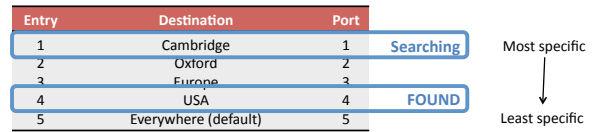
75

Router Architecture Overview

Two key router functions:
- run routing algorithms/protocol (RIP, OSPF, BGP)
- *forwarding* datagrams from incoming to outgoing link

input port
output port
switching fabric
routing processor

76

Input Port Functions

line termination

data link processing (protocol, decapsulation)

lookup, forwarding queueing

switch fabric

Physical layer: bit-level reception

Data link layer: e.g., Ethernet see chapter 5

Decentralized switching:
- given datagram dest., lookup output port using forwarding table in input port memory
- goal: complete input port processing at 'line speed'
- queuing: if datagrams arrive faster than forwarding rate into switch fabric

77

Three examples of switching fabrics
(comparison criteria: speed, contention, complexity)
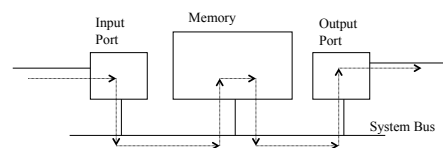

memory

bus

crossbar
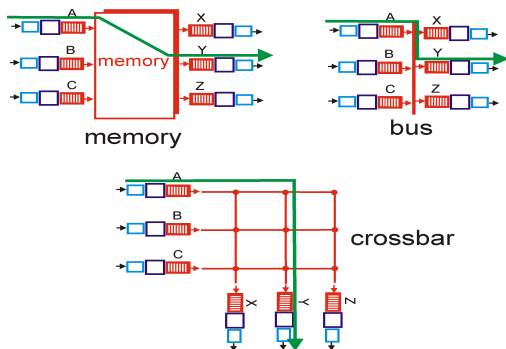
78
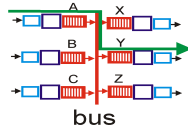

Switching Via Memory

First generation routers:
- traditional computers with switching under direct control of CPU
- packet copied to system's memory
- speed limited by memory bandwidth (2 bus crossings per datagram)

Input Port

Memory

Output Port

System Bus

79

Switching Via a Bus

- datagram from input port memory to output port memory via a shared bus
- bus contention: switching speed limited by bus bandwidth

- Lots of ports?? speed up the bus
  no contention bus speed =
      2 x port speed x port count

- 32 Gbps bus, Cisco 5600: sufficient speed for access routers

Switching Via An Interconnection Network

- overcome bus bandwidth limitations
- Banyan networks, other interconnection nets initially developed to connect processors in multiprocessor stages
- advanced design: fragmenting datagram into fixed length cells, switch cells through the fabric.
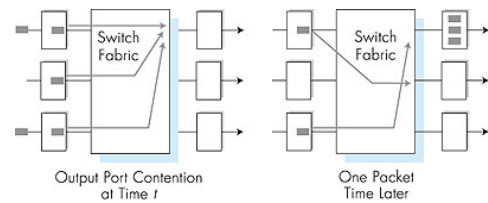- Cisco CRS-1: switches 1.2 Tbps through the interconnection network
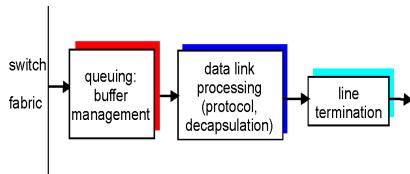
Output Ports

- *Buffering* required when datagrams arrive from fabric faster than the transmission rate

- *Scheduling discipline* chooses among queued datagrams for transmission
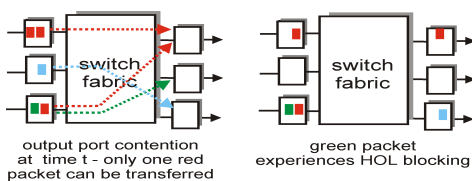   ➔ Who goes next?

Output port queueing

- buffering when arrival rate via switch exceeds output line speed
- *queueing (delay) and loss due to output port buffer overflow!*

Input Port Queuing

- Fabric slower than input ports combined -> queueing may occur at input queues
- Head-of-the-Line (HOL) blocking: queued datagram at front of queue prevents others in queue from moving forward
- *queueing delay and loss due to input buffer overflow!*

output port contention at time t - only one red packet can be transferred

green packet experiences HOL blocking

Buffers in Routers

- So how large should the buffers be?

Buffer size matters
  - End-to-end delay
    - Transmission, propagation, and queueing del
    - The only variable part is queueing delay
  - Router architecture
    - Board space, power consumption, and cos
    - On chip buffers: higher density, higher d
    - Optical buffers: all-optical routers

1.4m long spiral waveguide with input from HeNe laser
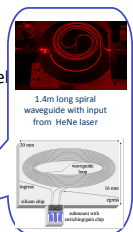
You are now touching the edge of the *research* zone......

Buffer Sizing Story

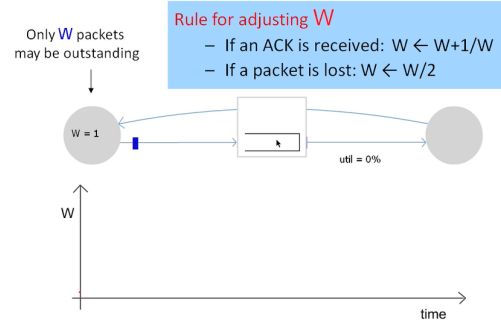|	Rule-of-thumb	$2T \times C$	Small Buffers	$\dfrac{2T \times C}{\sqrt{n}}$	Tiny Buffers	$O(\log W)$
# of packets		1,000,000		10,000		20 - 50
Intuition		TCP Sawtooth		Sawtooth Smoothing		Non-bursty Arrivals
Assume		Single TCP Flow, 100% Utilization		Many Flows, 100% Utilization		Paced TCP, 85-90% Utilization
Evidence		Simulation, Emulation		Simulations, Test-bed and Real Network Experiments		Simulations, Test-bed Experiments

---

Continuous ARQ (TCP) adapting to congestion

Only W packets may be outstanding

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

W = 1

util = 0%

W

time

Rule-of-thumb – Intuition

Only W packets may be outstanding

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

Source   Dest

Window size

W_{max}

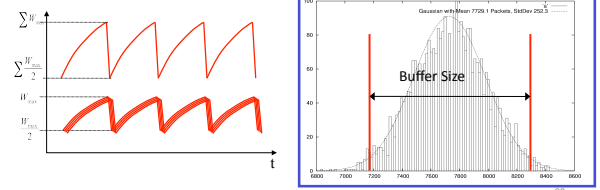$\dfrac{W_{max}}{2}$

$2T \times C$

$2T \times C$

t
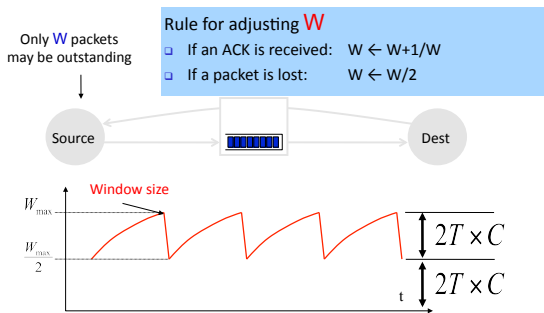
Small Buffers – Intuition

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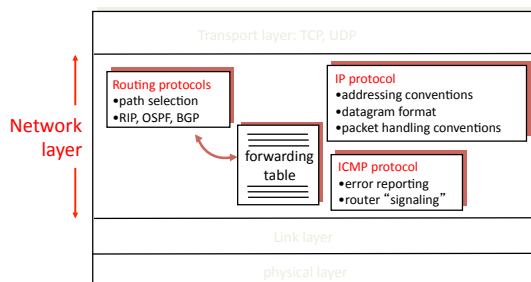- Rule-of-thumb still holds.

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

$\sum W_i$

$\sum \dfrac{W_i}{2}$

W_{tot}

$\dfrac{W_{tot}}{2}$

t

Gaussian with Mean 7729.1 Packets, StdDev 252.3

Buffer Size

The Internet version of a Network layer

Host, router network layer functions:

Transport layer: TCP, UDP

Routing protocols
- path selection
- RIP, OSPF, BGP

IP protocol
- addressing conventions
- datagram format
- packet handling conventions

forwarding table

ICMP protocol
- error reporting
- router "signaling"

Network layer

Link layer

physical layer

IPv4 Packet Structure
20 Bytes of Standard Header, then Options

| 4-bit Version	4-bit Header Length	8-bit Type of Service (TOS)		16-bit Total Length (Bytes)	
16-bit Identification			3-bit Flags	13-bit Fragment Offset	
8-bit Time to Live (TTL)		8-bit Protocol		16-bit Header Checksum	
32-bit Source IP Address					
32-bit Destination IP Address					
Options (if any)					
Payload					

(Packet) Network Tasks One-by-One

- Read packet correctly
- Get packet to the destination
- Get responses to the packet back to source
- Carry data
- Tell host what to do with packet once arrived
- Specify any special network handling of the packet
- Deal with problems that arise along the path

Reading Packet Correctly
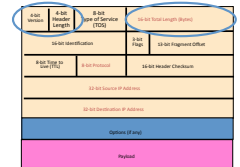
- Version number (4 bits)
  - Indicates the version of the IP protocol
  - Necessary to know what other fields to expect
  - Typically "4" (for IPv4), and sometimes "6" (for IPv6)
- Header length (4 bits)
  - Number of 32-bit words in the header
  - Typically "5" (for a 20-byte IPv4 header)
  - Can be more when IP options are used
- Total length (16 bits)
  - Number of bytes in the packet
  - Maximum size is 65,535 bytes (2^{16} -1)
  - … though underlying links may impose smaller limits

## Getting Packet to Destination and Back
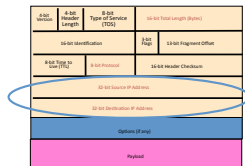
- Two IP addresses
  - Source IP address (32 bits)
  - Destination IP address (32 bits)
- Destination address
  - Unique identifier/locator for the receiving host
  - Allows each node to make forwarding decisions
- Source address
  - Unique identifier/locator for the sending host
  - Recipient can decide whether to accept packet
  - Enables recipient to send a reply back to source

Telling Host How to Handle Packet
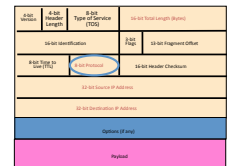
- Protocol (8 bits)
  - Identifies the higher-level protocol
  - Important for demultiplexing at receiving host
- Most common examples
  - E.g., "6" for the Transmission Control Protocol (TCP)
  - E.g., "17" for the User Datagram Protocol (UDP)

protocol=6	protocol=17
IP header	IP header
TCP header	UDP header
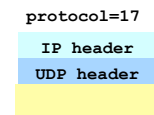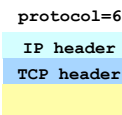
Special Handling
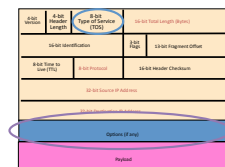
- Type-of-Service (8 bits)
  - Allow packets to be treated differently based on needs
  - E.g., low delay for audio, high bandwidth for bulk transfer
  - Has been redefined several times
- Options

Potential Problems

- Header Corrupted: **Checksum**

- Loop: **TTL**

- Packet too large: **Fragmentation**

Header Corruption

- Checksum (16 bits)
  - Particular form of checksum over packet header

- If not correct, router discards packets
  - So it doesn't act on bogus information

- Checksum recalculated at every router
  - Why?
  - Why include TTL?
  - Why only header?

98

Preventing Loops
(aka Internet Zombie plan)

- Forwarding loops cause packets to cycle forever
  - As these accumulate, eventually consume **all** capacity

- Time-to-Live (TTL) Field (8 bits)
  - Decremented at each hop, packet discarded if reaches 0
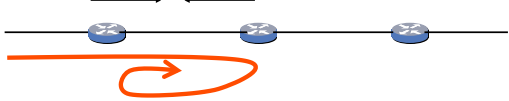 - …and "time exceeded" message is sent to the source
    - Using "ICMP" control message; basis for **traceroute**

99

Fragmentation
(some assembly required)

- Fragmentation: when forwarding a packet, an Internet router can split it into multiple pieces ("fragments") if too big for next hop link

- Must reassemble to recover original packet
  - Need fragmentation information (32 bits)
  - Packet identifier, flags, and fragment offset

100

IP Fragmentation & Reassembly

- network links have MTU (max.transfer size) - largest possible link-level frame.
  - different link types, different MTUs
- large IP datagram divided ("fragmented") within net
  - one datagram becomes several datagrams
  - "reassembled" only at final destination
  - IP header bits used to identify, order related fragments
- IPv6 does things differently…

fragmentation:
in: one large datagram
out: 3 smaller datagrams
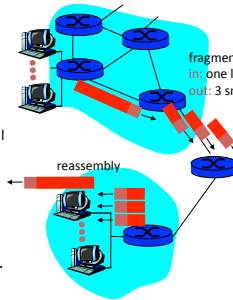
reassembly

101

IP Fragmentation and Reassembly

Example
- 4000 byte datagram
- MTU = 1500 bytes

| length =4000 | ID =x | fragflag =0 | offset =0 |

One large datagram becomes several smaller datagrams

1480 bytes in data field

offset = 1480/8

length =1500	ID =x	fragflag =1	offset =0
length =1500	ID =x	fragflag =1	offset =185
length =1040	ID =x	fragflag =0	offset =370

Pop quiz question: What happens when a fragment is lost?

102

Fragmentation Details

- Identifier (16 bits): used to tell which fragments belong together
- Flags (3 bits):
  - Reserved **(RF):** unused bit
  - Don't Fragment **(DF):** instruct routers to **not** fragment the packet even if it won't fit
    - Instead, they **drop** the packet and send back a "Too Large" ICMP control message
    - Forms the basis for "Path MTU Discovery"
  - More **(MF):** this fragment is not the last one
- Offset (13 bits): what part of datagram this fragment covers in 8-byte units

Pop quiz question: Why do frags use offset and not a frag number?

103

Options

- End of Options List
- No Operation (padding between options)
- Record Route
- Strict Source Route
- Loose Source Route
- Timestamp
- Traceroute
- Router Alert
- .....

104

IP Addressing: introduction

- IP address: 32-bit identifier for host, router *interface*
- *interface:* connection between host/router and physical link
  - router's typically have multiple interfaces
  - host typically has one interface
  - IP addresses associated with each interface

223.1.1.1 = 11011111 00000001 00000001 00000001

223   1   1   1

105
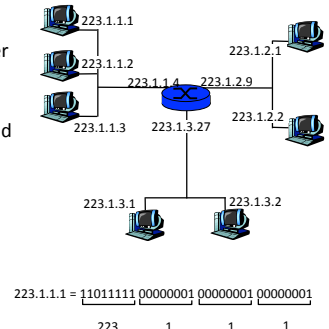
Subnets

- IP address:
  - subnet part (high order bits)
  - host part (low order bits)
- *What's a subnet ?*
  - device interfaces with same subnet part of IP address
  - can physically reach each other without intervening router

subnet part   host part
11011111 00000001 00000011 00000000

223.1.3.0/24
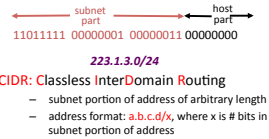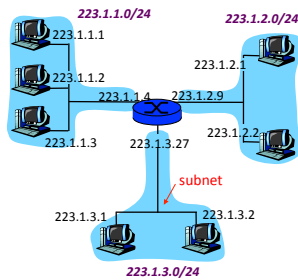
CIDR: Classless InterDomain Routing
- subnet portion of address of arbitrary length
- address format: a.b.c.d/x, where x is # bits in subnet portion of address

Subnet mask: /24

network consisting of 3 subnets

106

IP addresses: how to get one?

Q: How does a *host* get IP address?

- hard-coded by system admin in a file
  - Windows: control-panel->network->configuration->tcp/ip->properties
  - UNIX: /etc/rc.config (circa 1980's your mileage will vary)
- DHCP: Dynamic Host Configuration Protocol: dynamically get address from as server
  - "plug-and-play"

107

DHCP client-server scenario

Goal: allow host to *dynamically* obtain its IP address from network server when it joins network

Can renew its lease on address in use
Allows reuse of addresses (only hold address while connected an "on")
Support for mobile users who want to join network (more shortly)

108
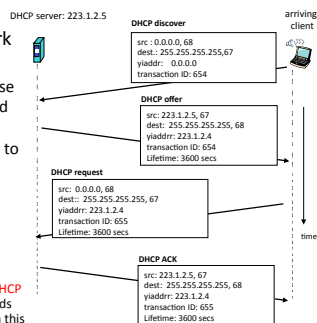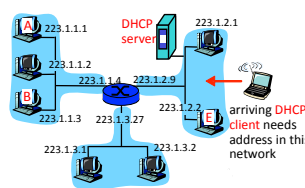
IP addresses: how to get one?

Q: How does *network* get subnet part of IP addr?
A: gets allocated portion of its provider ISP's address space

ISP's block   11001000 00010111 00010000 00000000   200.23.16.0/20

Organization 0   11001000 00010111 00010000 00000000   200.23.16.0/23
Organization 1   11001000 00010111 00010010 00000000   200.23.18.0/23
Organization 2   11001000 00010111 00010100 00000000   200.23.20.0/23
...   .
Organization 7   11001000 00010111 00011110 00000000   200.23.30.0/23

109

Hierarchical addressing: route aggregation

Hierarchical addressing allows efficient advertisement of routing information:

Organization 0
200.23.16.0/23

Organization 1
200.23.18.0/23

Organization 2
200.23.20.0/23

Organization 7
200.23.30.0/23

Fly-By-Night-ISP

"Send me anything with addresses beginning 200.23.16.0/20"

Internet

ISPs-R-Us

"Send me anything with addresses beginning 199.31.0.0/16"
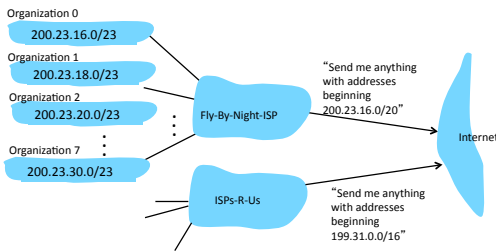
110

Hierarchical addressing: more specific routes

ISPs-R-Us has a more specific route to Organization 1

Organization 0
200.23.16.0/23

Organization 2
200.23.20.0/23

Organization 7
200.23.30.0/23

Fly-By-Night-ISP

"Send me anything with addresses beginning 200.23.16.0/20"

Internet

ISPs-R-Us

"Send me anything with addresses beginning 199.31.0.0/16 or 200.23.18.0/23"

Organization 1
200.23.18.0/23
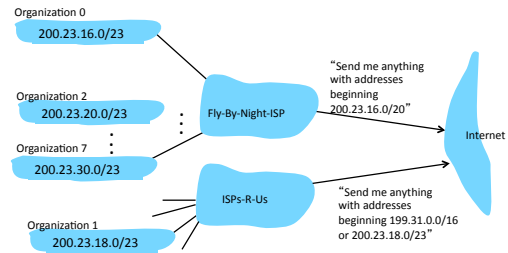
111

IP addressing: the last word...

Q: How does an ISP get a block of addresses?

A: ICANN: Internet Corporation for Assigned Names and Numbers

- allocates addresses
- manages DNS
- assigns domain names, resolves disputes

112

Cant get more IP addresses? well there is always.....

NAT: Network Address Translation

rest of Internet

local network (e.g., home network) 10.0.0/24

10.0.0.1

10.0.0.4

10.0.0.2

138.76.29.7

10.0.0.3

All datagrams *leaving* local network have same single source NAT IP address: 138.76.29.7, different source port numbers

Datagrams with source or destination in this network have 10.0.0/24 address for source, destination (as usual)
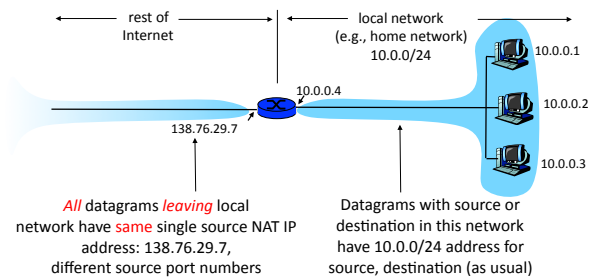
113

NAT: Network Address Translation

- Motivation: local network uses just one IP address as far as outside world is concerned:
  - range of addresses not needed from ISP: just one IP address for all devices
  - can change addresses of devices in local network without notifying outside world
  - can change ISP without changing addresses of devices in local network
  - devices inside local net not explicitly addressable, visible by outside world (a security plus).

114

NAT: Network Address Translation

Implementation: NAT router must:

- *outgoing datagrams: replace* (source IP address, port #) of every outgoing datagram to (NAT IP address, new port #)
  - . . . remote clients/servers will respond using (NAT IP address, new port #) as destination addr.
- *remember (in NAT translation table)* every (source IP address, port #) to (NAT IP address, new port #) translation pair
- *incoming datagrams: replace* (NAT IP address, new port #) in dest fields of every incoming datagram with corresponding (source IP address, port #) stored in NAT table

115

NAT: Network Address Translation

NAT translation table

WAN side addr	LAN side addr
138.76.29.7, 5001	10.0.0.1, 3345
......	......

2: NAT router changes datagram source addr from 10.0.0.1, 3345 to 138.76.29.7, 5001, updates table

1: host 10.0.0.1 sends datagram to 128.119.40.186, 80

S: 10.0.0.1, 3345
D: 128.119.40.186, 80

S: 138.76.29.7, 5001
D: 128.119.40.186, 80

10.0.0.1
10.0.0.2
10.0.0.3
10.0.0.4

138.76.29.7

S: 128.119.40.186, 80
D: 138.76.29.7, 5001

S: 128.119.40.186, 80
D: 10.0.0.1, 3345

3: Reply arrives dest. address: 138.76.29.7, 5001

4: NAT router changes datagram dest addr from 138.76.29.7, 5001 to 10.0.0.1, 3345

116

NAT: Network Address Translation

- 16-bit port-number field:
  - 60,000 simultaneous connections with a single LAN-side address!
- NAT is controversial:
  - routers should only process up to layer 3
  - violates end-to-end argument (?)
    - NAT possibility must be taken into account by app designers, eg, P2P applications
  - address shortage should instead be solved by IPv6

117

NAT traversal problem

- client wants to connect to server with address 10.0.0.1
  - server address 10.0.0.1 local to LAN (client can't use it as destination addr)
  - only one externally visible NATted address: 138.76.29.7
- solution 1: statically configure NAT to forward incoming connection requests at given port to server
  - e.g., (138.76.29.7, port 2500) always forwarded to 10.0.0.1 port 25000

Client
?
10.0.0.1
10.0.0.4
138.76.29.7
NAT router

118

NAT traversal problem

- solution 2: Universal Plug and Play (UPnP) Internet Gateway Device (IGD) Protocol. Allows NATted host to:
  - ❖ learn public IP address (138.76.29.7)
  - ❖ add/remove port mappings (with lease times)

i.e., automate static NAT port map configuration

10.0.0.1
IGD
10.0.0.4
138.76.29.7
NAT router

119

NAT traversal problem
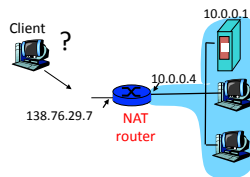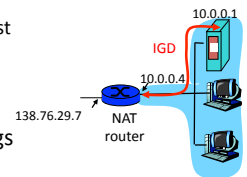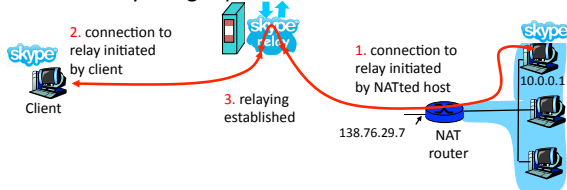
- solution 3: relaying (used in Skype)
  - NATed client establishes connection to relay
  - External client connects to relay
  - relay bridges packets between to connections

2. connection to relay initiated by client

Client

1. connection to relay initiated by NATted host

10.0.0.1

3. relaying established

138.76.29.7
NAT router

120

Remember this? Traceroute at work…

traceroute: rio.cl.cam.ac.uk to munnari.oz.au
(tracepath on pwf is similar)

Three delay measurements from rio.cl.cam.ac.uk to gatwick.net.cl.cam.ac.uk

trans-continent link

```
traceroute munnari.oz.au
traceroute to munnari.oz.au (202.29.151.3), 30 hops max, 60 byte packets
 1  gatwick.net.cl.cam.ac.uk (128.232.32.2)  0.416 ms  0.384 ms  0.427 ms
 2  cl-sby.route-nwest.net.cam.ac.uk (193.60.89.9)  0.393 ms  0.440 ms  0.494 ms
 3  route-nwest.route-mill.net.cam.ac.uk (192.84.5.137)  0.407 ms  0.448 ms  0.501 ms
 4  route-mill.route-enet.net.cam.ac.uk (192.84.5.94)  1.006 ms  1.091 ms  1.163 ms
 5  xe-11-3-0.camb-rbr1.eastern.ja.net (146.97.130.1)  0.300 ms  0.313 ms  0.350 ms
 6  ae24.lowdss-sbr1.ja.net (146.97.37.185)  2.679 ms  2.664 ms  2.712 ms
 7  ae28.londhx-sbr1.ja.net (146.97.33.17)  5.955 ms  5.953 ms  5.901 ms
 8  janet.mx1.lon.uk.geant.net (62.40.124.197)  6.059 ms  6.066 ms  6.052 ms
 9  ae0.mx1.par.fr.geant.net (62.40.98.77)  11.742 ms  11.779 ms  11.724 ms
10  ae1.mx1.mad.es.geant.net (62.40.98.64)  27.751 ms  27.734 ms  27.704 ms
11  mb-so-02-v4.bb.tein3.net (202.179.249.117)  138.296 ms  138.314 ms  138.282 ms
12  sg-so-04-v4.bb.tein3.net (202.179.249.53)  196.303 ms  196.293 ms  196.264 ms
13  th-pr-v4.bb.tein3.net (202.179.249.66)  225.153 ms  225.178 ms  225.196 ms
14  pyt-thairen-to-02-bdr-pyt.uni.net.th (202.29.12.10)  225.163 ms  223.343 ms  223.363 ms
15  202.28.227.126 (202.28.227.126)  241.038 ms  240.941 ms  240.834 ms
16  202.28.221.46 (202.28.221.46)  287.252 ms  287.306 ms  287.282 ms
17  * * *
18  * * *
19  * * *
20  coe-gw.psu.ac.th (202.29.149.70)  241.681 ms  241.715 ms  241.680 ms
21  munnari.OZ.AU (202.29.151.3)  241.610 ms  241.636 ms  241.537 ms
```

* means no response (probe lost, router not replying)

121

Traceroute and ICMP

- Source sends series of UDP segments to dest
  - First has TTL =1
  - Second has TTL=2, etc.
  - Unlikely port number
- When nth datagram arrives to nth router:
  - Router discards datagram
  - And sends to source an ICMP message (type 11, code 0)
  - Message includes name of router& IP address

- When ICMP message arrives, source calculates RTT
- Traceroute does this 3 times

Stopping criterion
- UDP segment eventually arrives at destination host
- Destination returns ICMP "host unreachable" packet (type 3, code 3)
- When source gets this ICMP, stops.

ICMP: Internet Control Message Protocol

- used by hosts & routers to communicate network-level information
  - error reporting: unreachable host, network, port, protocol
  - echo request/reply (used by ping)
- network-layer "above" IP:
  - ICMP msgs carried in IP datagrams
- ICMP message: type, code plus first 8 bytes of IP datagram causing error

Type	Code	description
0	0	echo reply (ping)
3	0	dest. network unreachable
3	1	dest host unreachable
3	2	dest protocol unreachable
3	3	dest port unreachable
3	6	dest network unknown
3	7	dest host unknown
4	0	source quench (congestion control - not used)
8	0	echo request (ping)
9	0	route advertisement
10	0	router discovery
11	0	TTL expired
12	0	bad IP header

IPv6

- Motivated (prematurely) by address exhaustion
  - Address field *four* times as long

- Steve Deering focused on simplifying IP
  - Got rid of all fields that were not absolutely necessary
  - "Spring Cleaning" for IP

- Result is an elegant, if unambitious, protocol

IPv4 and IPv6 Header Comparison

IPv4

Version	IHL	Type of Service	Total Length	
Identification			Flags	Fragment Offset
Time to Live		Protocol	Header Checksum	
Source Address				
Destination Address				
Options				Padding

IPv6

Version	Traffic Class	Flow Label	
Payload Length		Next Header	Hop Limit
Source Address			
Destination Address			

- Field name kept from IPv4 to IPv6
- Fields not kept in IPv6
- Name & position changed in IPv6
- New field in IPv6
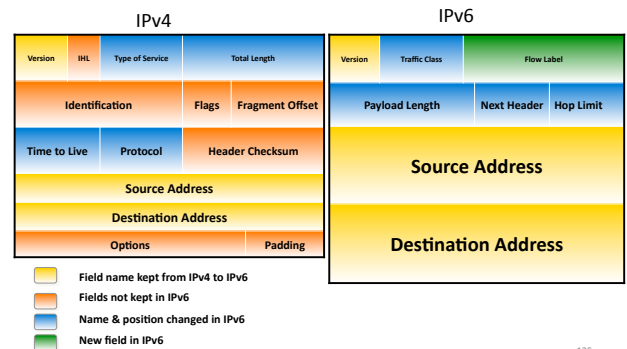
Summary of Changes

- Eliminated fragmentation *(why?)*
- Eliminated header length *(why?)*
- Eliminated checksum *(why?)*
- New options mechanism (next header) *(why?)*
- Expanded addresses *(why?)*
- Added Flow Label *(why?)*

IPv4 and IPv6 Header Comparison

IPv4

Version	IHL	Type of Service	Total Length	
Identification			Flags	Fragment Offset
Time to Live		Protocol	Header Checksum	
Source Address				
Destination Address				
Options				Padding

IPv6

Version	Traffic Class	Flow Label	
Payload Length		Next Header	Hop Limit
Source Address			
Destination Address			

- Field name kept from IPv4 to IPv6
- Fields not kept in IPv6
- Name & position changed in IPv6
- New field in IPv6
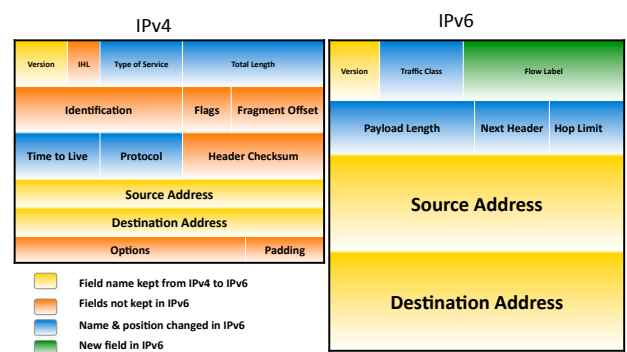
Philosophy of Changes

- Don't deal with problems: leave to ends
  - Eliminated fragmentation
  - Eliminated checksum
  - *Why retain TTL?*
- Simplify handling:
  - New options mechanism (uses next header approach)
  - Eliminated header length
    - *Why couldn't IPv4 do this?*
- Provide general flow label for packet
  - Not tied to semantics
  - Provides great flexibility

128

Comparison of Design Philosophy

IPv4

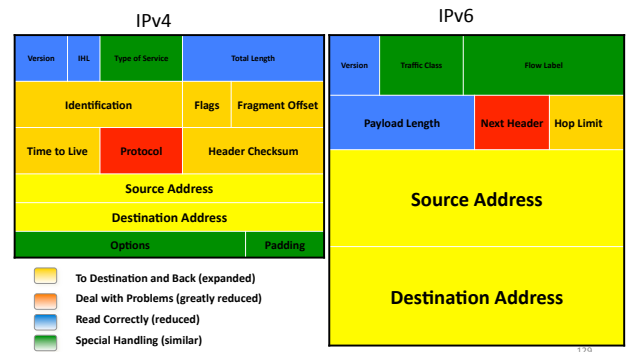| Version | IHL | Type of Service | Total Length |
Identification		Flags	Fragment Offset
Time to Live		Protocol	Header Checksum
Source Address			
Destination Address			
Options			Padding

IPv6

Version	Traffic Class	Flow Label
Payload Length		Next Header	Hop Limit
Source Address		
Destination Address		

- To Destination and Back (expanded)
- Deal with Problems (greatly reduced)
- Read Correctly (reduced)
- Special Handling (similar)

129

Transition From IPv4 To IPv6

- Not all routers can be upgraded simultaneous
  - no "flag days"
  - How will the network operate with mixed IPv4 and IPv6 routers?
- *Tunneling:* IPv6 carried as payload in IPv4 datagram among IPv4 routers

130

Tunneling
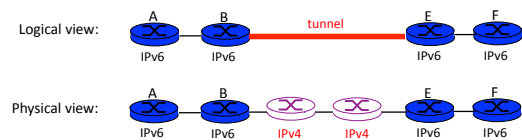
131

Tunneling
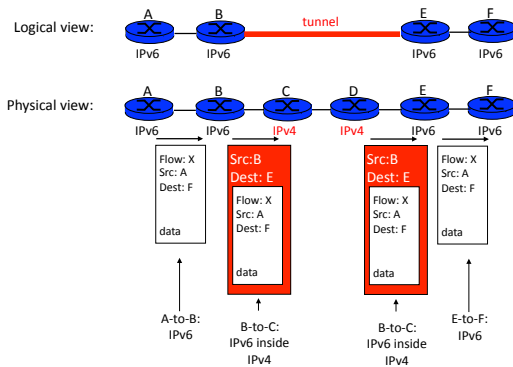
132

Improving on IPv4 and IPv6?

- Why include unverifiable source address?
  - Would like accountability **and** anonymity (now neither)
  - Return address can be communicated at higher layer
- Why packet header used at edge same as core?
  - Edge: host tells network what service it wants
  - Core: packet tells switch how to handle it
    - One is local to host, one is global to network
- Some kind of payment/responsibility field?
  - Who is responsible for paying for packet delivery?
  - Source, destination, other?
- Other ideas?

133

Gluing it together:
How does my Network (address) interact
with my Data-Link (address) ?

Switches vs. Routers Summary

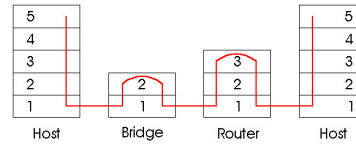- both store-and-forward devices
  - routers: network layer devices (examine network layer headers)
  - switches are link layer devices
- routers maintain routing tables, implement routing algorithms
- switches maintain switch tables, implement filtering, learning algorithms

MAC Addresses (and IPv4 ARP)
or How do I glue my network to my data-link?

- 32-bit IP address:
  - *network-layer* address
  - used to get datagram to destination IP subnet
- MAC (or LAN or physical or Ethernet) address:
  - function: *get frame from one interface to another physically-connected interface (same network)*
  - 48 bit MAC address (for most LANs)
    - burned in NIC ROM, also (commonly) software settable

LAN Addresses and ARP

Each adapter on LAN has unique LAN address

Ethernet
Broadcast address =
FF-FF-FF-FF-FF-FF

= adapter
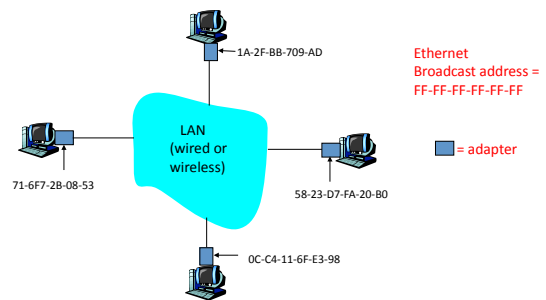
Address Resolution Protocol

- Every node maintains an ARP table
  - <IP address, MAC address> pair

- Consult the table when sending a packet
  - Map destination IP address to destination MAC address
  - Encapsulate and transmit the data packet

- But: what if IP address not in the table?
  - Sender broadcasts: "**Who has IP address 1.2.3.156**?"
  - Receiver responds: "**MAC address 58-23-D7-FA-20-B0**"
  - Sender caches result in its ARP table

Example: A Sending a Packet to B
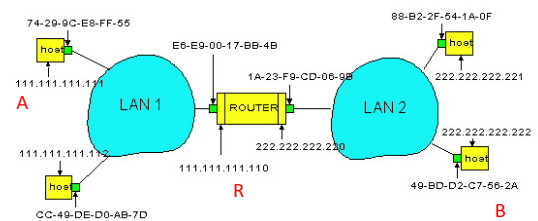
How does host A send an IP packet to host B?

Example: A Sending a Packet to B

How does host A send an IP packet to host B?

74-29-9C-E8-FF-55
E6-E9-00-17-BB-4B
88-B2-2F-54-1A-0F
111.111.111.111
1A-23-F9-CD-06-9B
222.222.222.221
A
LAN 1
ROUTER
LAN 2
111.111.111.112
222.222.222.222
222.222.222.220
B
111.111.111.110
49-BD-D2-C7-56-2A
CC-49-DE-D0-AB-7D
R

1. A sends packet to R.
2. R sends packet to B.

Host A Decides to Send Through R

- Host A constructs an IP packet to send to B
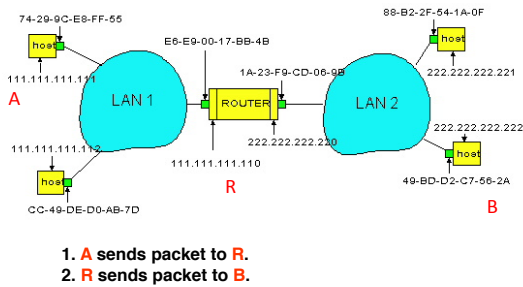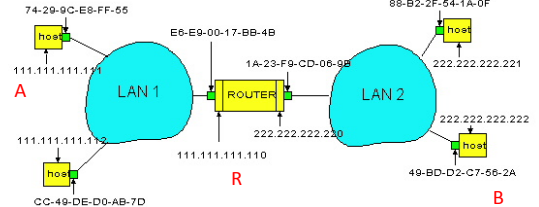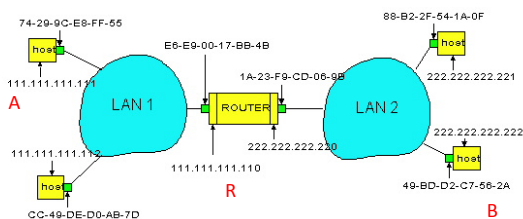 - Source 111.111.111.111, destination 222.222.222.222
- Host A has a gateway router R
  - Used to reach destinations outside of 111.111.111.0/24
  - Address 111.111.111.110 for R learned via DHCP/config

74-29-9C-E8-FF-55
E6-E9-00-17-BB-4B
88-B2-2F-54-1A-0F
111.111.111.111
1A-23-F9-CD-06-9B
222.222.222.221
A
LAN 1
ROUTER
LAN 2
111.111.111.112
222.222.222.222
222.222.222.220
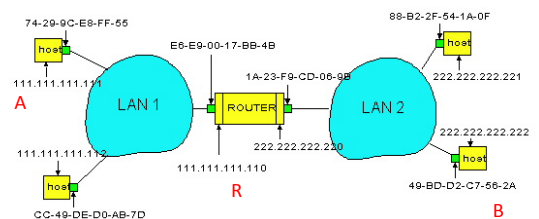B
111.111.111.110
49-BD-D2-C7-56-2A
CC-49-DE-D0-AB-7D
R

Host A Sends Packet Through R

- Host A learns the MAC address of R's interface
  - ARP request: broadcast request for 111.111.111.110
  - ARP response: R responds with E6-E9-00-17-BB-4B
- Host A encapsulates the packet and sends to R

74-29-9C-E8-FF-55
E6-E9-00-17-BB-4B
88-B2-2F-54-1A-0F
111.111.111.111
1A-23-F9-CD-06-9B
222.222.222.221
A
LAN 1
ROUTER
LAN 2
111.111.111.112
222.222.222.222
222.222.222.220
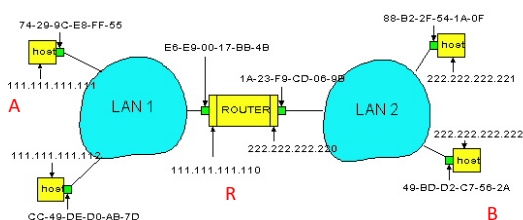B
111.111.111.110
49-BD-D2-C7-56-2A
CC-49-DE-D0-AB-7D
R

R Decides how to Forward Packet

- Router R's adaptor receives the packet
  - R extracts the IP packet from the Ethernet frame
  - R sees the IP packet is destined to 222.222.222.222
- Router R consults its forwarding table
  - Packet matches 222.222.222.0/24 via other adaptor

74-29-9C-E8-FF-55
E6-E9-00-17-BB-4B
88-B2-2F-54-1A-0F
111.111.111.111
1A-23-F9-CD-06-9B
222.222.222.221
A
LAN 1
ROUTER
LAN 2
111.111.111.112
222.222.222.222
222.222.222.220
B
111.111.111.110
49-BD-D2-C7-56-2A
CC-49-DE-D0-AB-7D
R

R Sends Packet to B

- Router R's learns the MAC address of host B
  - ARP request: broadcast request for 222.222.222.222
  - ARP response: B responds with 49-BD-D2-C7-52A
- Router R encapsulates the packet and sends to B

74-29-9C-E8-FF-55
E6-E9-00-17-BB-4B
88-B2-2F-54-1A-0F
111.111.111.111
1A-23-F9-CD-06-9B
222.222.222.221
A
LAN 1
ROUTER
LAN 2
111.111.111.112
222.222.222.222
222.222.222.220
B
111.111.111.110
49-BD-D2-C7-56-2A
CC-49-DE-D0-AB-7D
R

Security Analysis of ARP

- Impersonation
  - Any node that hears request can answer …
  - … and can say whatever they want

- Actual legit receiver never sees a problem
  - Because even though later packets carry its IP address, its NIC doesn't capture them since not its MAC address

Key Ideas in Both ARP and DHCP

- **Broadcasting**: Can use broadcast to make contact
  - Scalable because of limited size

- **Caching**: remember the past for a while
  - Store the information you learn to reduce overhead
  - Remember your own address & other host's addresses

- **Soft state**: eventually forget the past
  - Associate a time-to-live field with the information
  - … and either refresh or discard the information
  - Key for robustness in the face of unpredictable change

146

Why Not Use DNS-Like Tables?

- When host arrives:
  - Assign it an IP address that will last as long it is present
  - Add an entry into a table in DNS-server that maps MAC to IP addresses

- Answer:
  - Names: explicit creation, and are plentiful
  - Hosts: come and go without informing network
    - Must do mapping on demand
  - Addresses: not plentiful, need to reuse and remap
    - Soft-state enables dynamic reuse

147

Summary Network Layer

- understand principles behind network layer services:
  - network layer service models
  - forwarding versus routing (versus switching)
  - how a router works
  - routing (path selection)
  - IPv6
- Algorthims
  - Two routing approaches (LS vs DV)
  - One of these in detail (LS)
  - ARP

148

Topic 5 – Transport

Our goals:
- understand principles behind transport layer services:
  - multiplexing/ demultiplexing
  - reliable data transfer
  - flow control
  - congestion control

- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

2

Transport Layer

- Commonly a layer at end-hosts, between the application and network layer
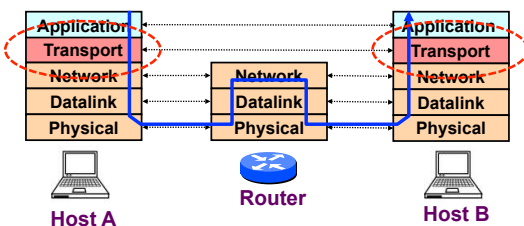
3

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 (*more multiplexing*)
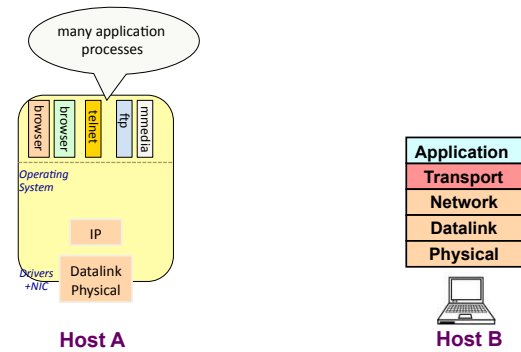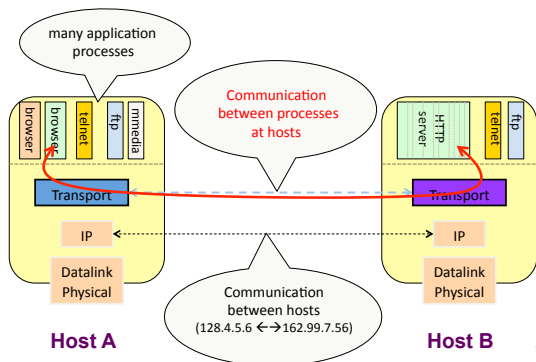
4

Why a transport layer?

Application
Transport
Network
Datalink
Physical

Host A

Application
Transport
Network
Datalink
Physical

Host B 5

Why a transport layer?

many application processes

browser
browser
telnet
ftp
mmedia

Operating System

IP

Drivers +NIC

Datalink
Physical

Host A

Application
Transport
Network
Datalink
Physical

Host B 6

Why a transport layer?

many application processes

browser
browser
telnet
ftp
mmedia

Communication between processes at hosts

HTTP server
telnet
ftp

Transport

IP

Datalink
Physical

Host A

Transport

IP

Datalink
Physical

Communication between hosts
(128.4.5.6 ←→162.99.7.56)

Host B 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

Role of the Transport Layer

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

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

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

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

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

Role of the Transport Layer

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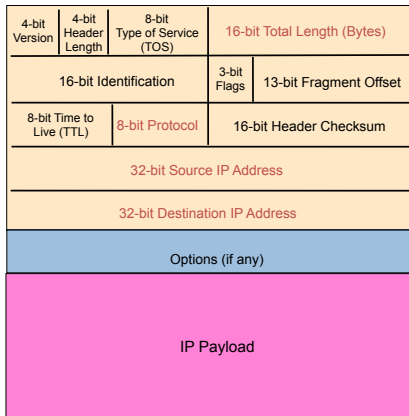 - implemented using ports

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
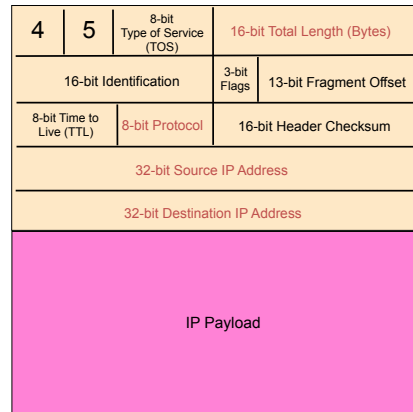 - TCP socket: TYPE is SOCK_STREAM

15

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

Slide 17

4-bit Version	4-bit Header Length	8-bit Type of Service (TOS)	16-bit Total Length (Bytes)	
16-bit Identification			3-bit Flags	13-bit Fragment Offset
8-bit Time to Live (TTL)		8-bit Protocol	16-bit Header Checksum	
32-bit Source IP Address				
32-bit Destination IP Address				
Options (if any)				
IP Payload				

17

Slide 18

4	5	8-bit Type of Service (TOS)	16-bit Total Length (Bytes)	
16-bit Identification			3-bit Flags	13-bit Fragment Offset
8-bit Time to Live (TTL)		8-bit Protocol	16-bit Header Checksum	
32-bit Source IP Address				
32-bit Destination IP Address				
IP Payload				

18

Slide 19

4	5	8-bit Type of Service (TOS)	16-bit Total Length (Bytes)	
16-bit Identification			3-bit Flags	13-bit Fragment Offset
8-bit Time to Live (TTL)		6 = TCP 17 = UDP	16-bit Header Checksum	
32-bit Source IP Address				
32-bit Destination IP Address				
TCP or UDP header and Payload				

19

Slide 20

4	5	8-bit Type of Service (TOS)	16-bit Total Length (Bytes)	
16-bit Identification			3-bit Flags	13-bit Fragment Offset
8-bit Time to Live (TTL)		6 = TCP 17 = UDP	16-bit Header Checksum	
32-bit Source IP Address				
32-bit Destination IP Address				
16-bit Source Port			16-bit Destination Port	
More transport header fields ….				
TCP or UDP header and Payload				

20

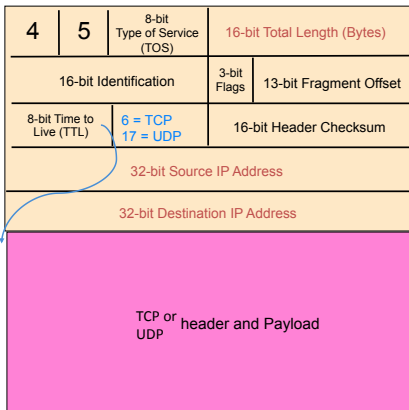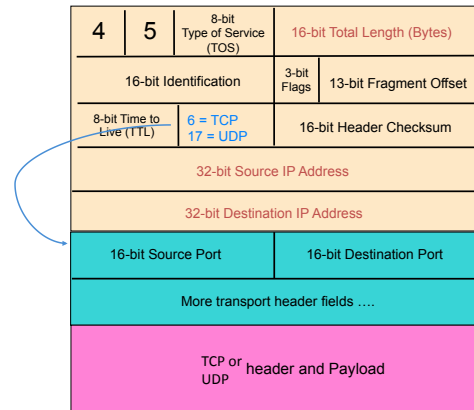## Recap: Multiplexing and Demultiplexing

- 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

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
  - helps client know server's port

- Ephemeral ports (most 1024-65535): dynamically selected: as the source port for a client process

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")

SRC port	DST port
checksum	length
DATA	
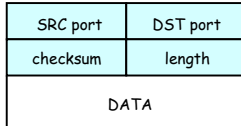
23

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

Principles of Reliable data transfer

- important in app., transport, link layers
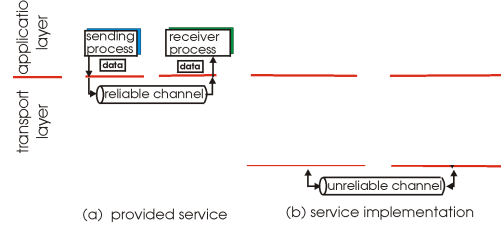- top-10 list of important networking topics!
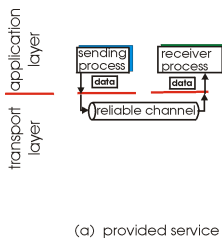
- In a perfect world, reliable transport is easy

  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?*)

(a) provided service

25

Principles of Reliable data transfer

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

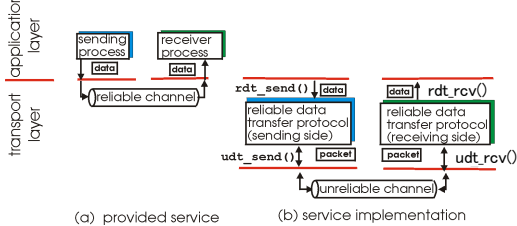(a) provided service   (b) service implementation

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

26

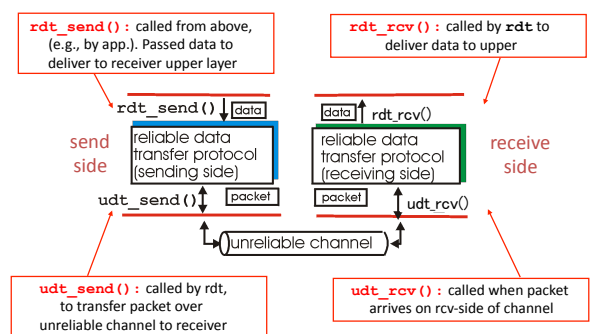---

Principles of Reliable data transfer

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

(a) provided service   (b) service implementation

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

27

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

rdt_rcv(): called by **rdt** to deliver data to upper

send side

receive side

udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

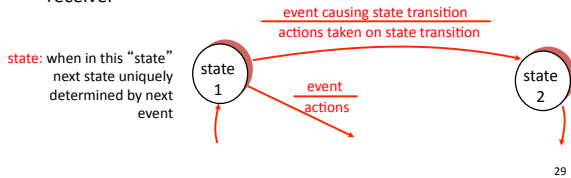**udt_rcv():** called when packet arrives on rcv-side of channel

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

state: when in this "state" next state uniquely determined by next event

event causing state transition
actions taken on state transition

state 1

event
actions

state 2

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.

state: when in this "state" next state uniquely determined by next event

Relevant event causing state transition
Relevant action taken on state transition

State name

event
actions

State name

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
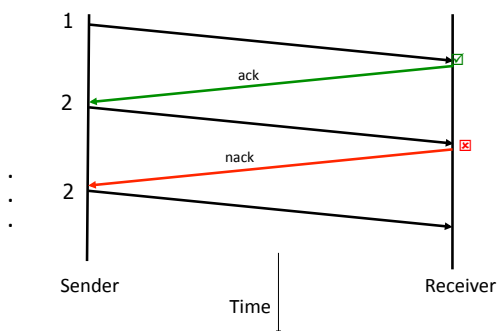 - receiver read data from underlying channel

Event

IDLE

rdt_send(data)
udt_send(packet)

IDLE

udt_rcv(packet)
rdt_rcv(data)

Action

sender   **receiver**

31
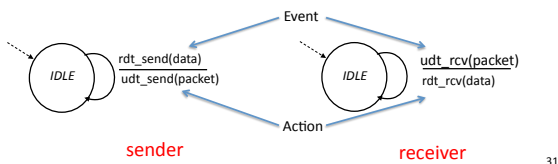
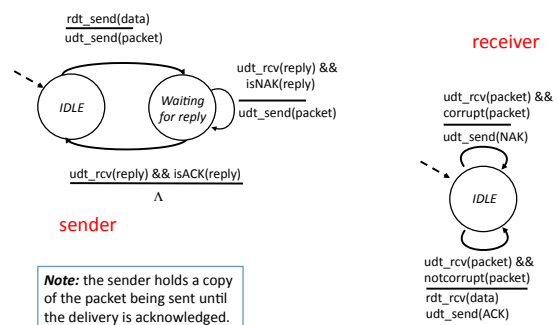## Rdt2.0: channel with bit errors

- 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
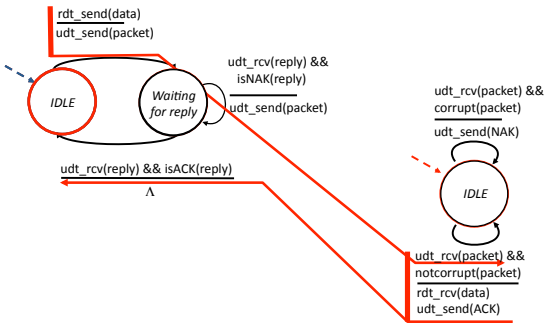 - receiver feedback: control msgs (ACK,NAK) receiver->sender
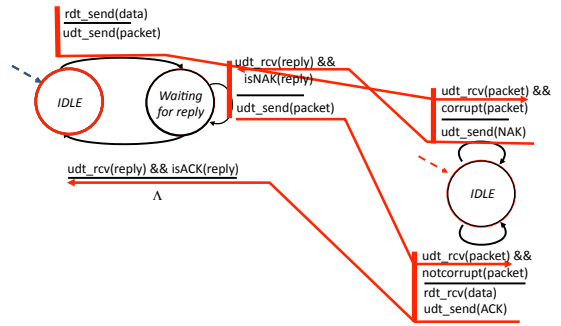
32

Dealing with Packet Corruption

1

ack

2

nack

2

Sender   Receiver

Time

33

rdt2.0: FSM specification

rdt_send(data)
udt_send(packet)

IDLE   Waiting for reply

udt_rcv(reply) && isNAK(reply)
udt_send(packet)

udt_rcv(reply) && isACK(reply)
Λ

receiver

udt_rcv(packet) && corrupt(packet)
udt_send(NAK)

IDLE

udt_rcv(packet) && notcorrupt(packet)
rdt_rcv(data)
udt_send(ACK)

sender

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

34

rdt2.0: operation with no errors

rdt_send(data)
udt_send(packet)

udt_rcv(reply) &&
isNAK(reply)
udt_send(packet)

IDLE

Waiting for reply

udt_rcv(reply) && isACK(reply)
Λ

udt_rcv(packet) &&
corrupt(packet)
udt_send(NAK)

IDLE

udt_rcv(packet) &&
notcorrupt(packet)
rdt_rcv(data)
udt_send(ACK)

---

rdt2.0: error scenario

rdt_send(data)
udt_send(packet)

udt_rcv(reply) &&
isNAK(reply)
udt_send(packet)

IDLE

Waiting for reply

udt_rcv(reply) && isACK(reply)
Λ

udt_rcv(packet) &&
corrupt(packet)
udt_send(NAK)

IDLE

udt_rcv(packet) &&
notcorrupt(packet)
rdt_rcv(data)
udt_send(ACK)

---

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

Dealing with Packet Corruption

1   P(1)

ack(1)

1   P(1)

ack(1)

2   P(2)

Packet #1 or #2?

time

Data and ACK packets carry _sequence numbers_

---
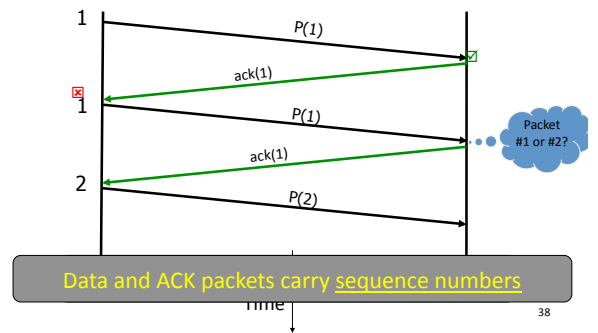
rdt2.1: sender, handles garbled ACK/NAKs

rdt_send(data)
sequence=0
udt_send(packet)

IDLE

Waiting For reply

udt_rcv(reply) &&
(corrupt(reply) ||
isNAK(reply))
udt_send(packet)

udt_rcv(reply)
&& notcorrupt(reply)
&& isACK(reply)
Λ

udt_rcv(reply)
&& notcorrupt(reply)
&& isACK(reply)
Λ

Waiting for reply

IDLE

udt_rcv(reply) &&
(corrupt(reply) ||
isNAK(reply))
udt_send(packet)

rdt_send(data)
sequence=1
udt_send(packet)

---
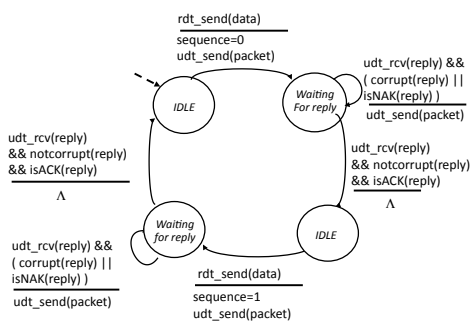
rdt2.1: receiver, handles garbled ACK/NAKs

udt_rcv(packet) && not corrupt(packet)
&& has_seq0(packet)
udt_send(ACK)
rdt_rcv(data)

receive(packet) && corrupt(packet)
udt_send(NAK)

udt_rcv(packet) && corrupt(packet)
udt_send(NAK)

receive(packet) &&
not corrupt(packet) &&
has_seq1(packet)
udt_send(ACK)

Wait for 0 from below

Wait for 1 from below

receive(packet) &&
not corrupt(packet) &&
has_seq0(packet)
udt_send(ACK)

udt_rcv(packet) && not corrupt(packet)
&& has_seq1(packet)
udt_send(ACK)
rdt_rcv(data)

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

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

rdt2.2: sender, receiver fragments
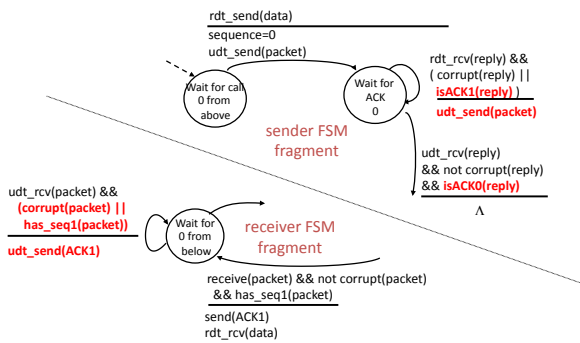
43

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

rdt3.0 sender
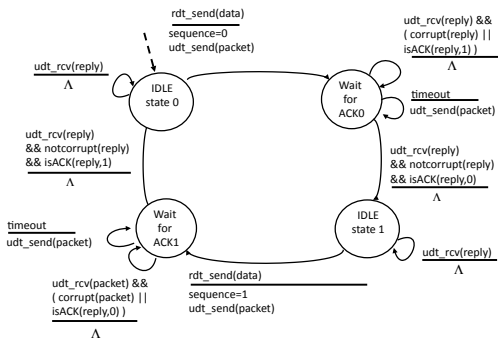
45

Dealing with Packet Loss

Timer-driven loss detection
Set timer when packet is sent; retransmit on timeout

Dealing with Packet Loss

Timeout

Sender — Time — Receiver

P(1)
P(1)
ack(1)
P(2)

duplicate!

47

Dealing with Packet Loss

Timeout

P(1)
P(1)
ack(1)
ack(1)
P(2)

duplicate!
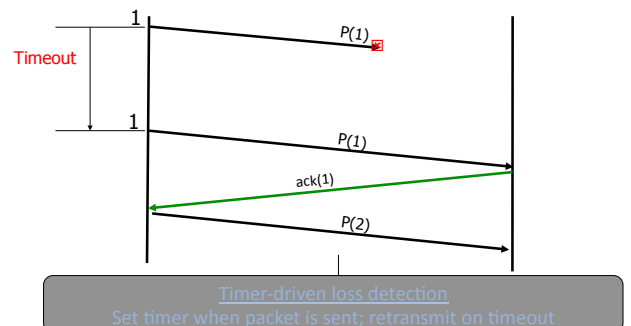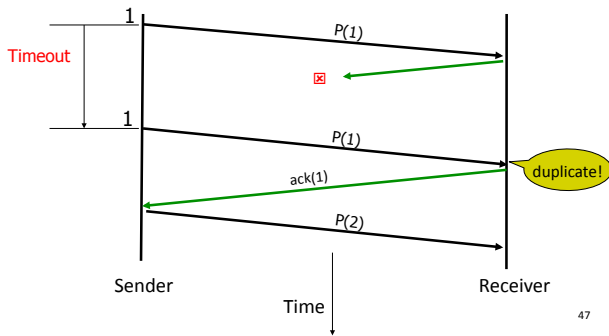
Timer-driven retx. can lead to duplicates
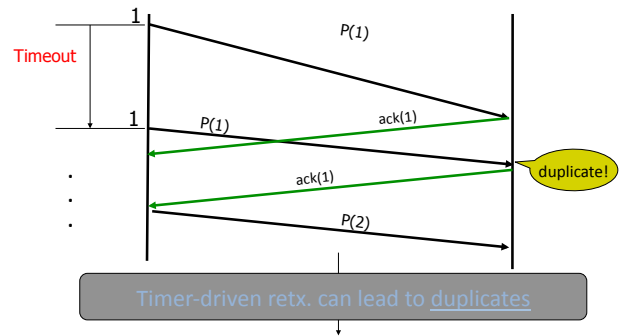
Performance of rdt3.0

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

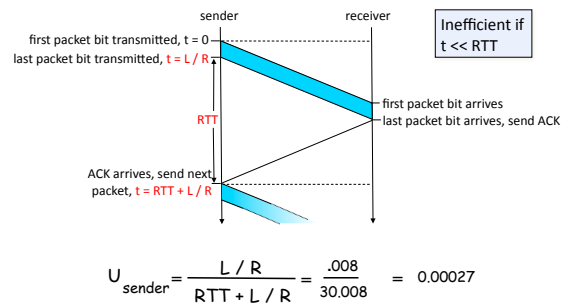$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\text{microseconds}$$

❍ 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

rdt3.0: stop-and-wait operation

sender   receiver

first packet bit transmitted, t = 0
last packet bit transmitted, t = L / R

RTT

first packet bit arrives
last packet bit arrives, send ACK

ACK arrives, send next
packet, t = RTT + L / R

Inefficient if
t << RTT

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

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

data packet
data packets
ACK packets

(a) a stop-and-wait protocol in operation   (b) a pipelined protocol in operation
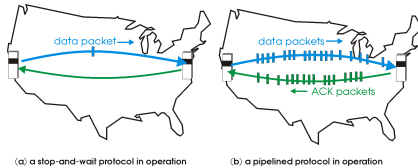
51

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

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}
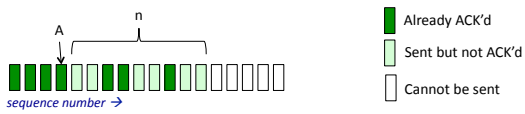
- Already ACK'd
- Sent but not ACK'd
- Cannot be sent

sequence number →

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

- Received and ACK'd
- Acceptable but not yet received
- Cannot be received

53

Acknowledgements w/ Sliding Window

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

54

Cumulative Acknowledgements (1)

- At receiver

- Received and ACK'd
- Acceptable but not yet received
- Cannot be received
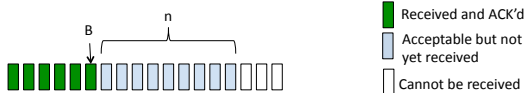
- After receiving B+1, B+2

$B_{new} = B+2$
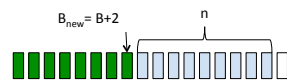
- Receiver sends ACK(B_{new}+1)

55

Cumulative Acknowledgements (2)

- At receiver

- Received and ACK'd
- Acceptable but not yet received
- Cannot be received
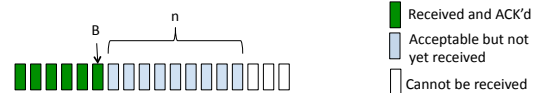
- After receiving B+4, B+5
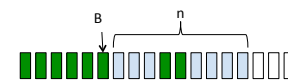
How do we recover?

- Receiver sends ACK(B+1)

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

Sliding Window with GBN

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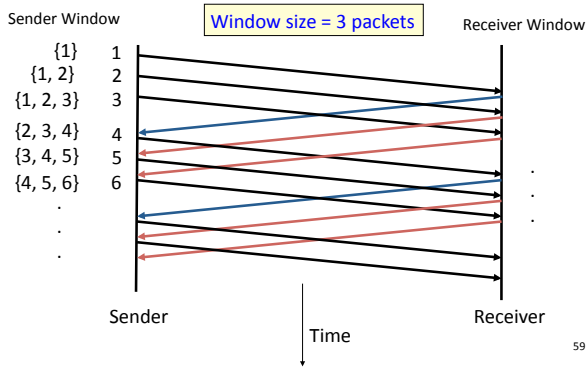 then window of sender = {A+1, A+2, …, A+n}

- Already ACK'd
- Sent but not ACK'd
- Cannot be sent

sequence number →

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

- Received and ACK'd
- Acceptable but not yet received
- Cannot be received

58

GBN Example w/o Errors

Window size = 3 packets

Sender Window

{1}	1
{1, 2}	2
{1, 2, 3}	3
{2, 3, 4}	4
{3, 4, 5}	5
{4, 5, 6}	6

Receiver Window

Sender

Receiver

Time

59

GBN Example with Errors

Window size = 3 packets

1
2
3
4
5
6

Timeout Packet 4

4
5
6

Sender

Receiver

60

GBN: sender extended FSM

rdt_send(data)
if (nextseqnum < base+N) {
    udt_send(packet[nextseqnum])
    nextseqnum++
}
else
    refuse_data(data)   **Block?**

Λ
base=1
nextseqnum=1

Wait

udt_rcv(reply)
&& corrupt(reply)
Λ

timeout
udt_send(packet[base])
udt_send(packet[base+1])
…
udt_send(packet[nextseqnum-1])

udt_rcv(reply) &&
notcorrupt(reply)
base = getacknum(reply)+1

61

GBN: receiver extended FSM

Λ
udt_send(reply)

udt_rcv(packet)
&& notcurrupt(packet)
&& hasseqnum(rcvpkt,expectedseqnum)

Λ
expectedseqnum=1

Wait

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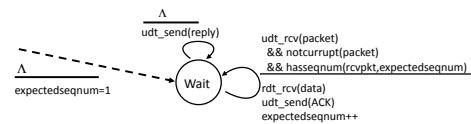
ACK-only: always send an ACK for correctly-received packet with the highest *in-order* seq #
- may generate duplicate ACKs
- need only remember **expectedseqnum**

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

62

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

63

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

64

SR Example with Errors

| Window size = 3 packets |

```
{1}        1
{1, 2}     2
{1, 2, 3}  3
{2, 3, 4}  4
           5
           6
{4,5,6}    4
{4,5,6}
                          Time
{7, 8, 9}  7
```

Timeout Packet 4

ACK=5
ACK=6
ACK=4

Sender   Receiver   65
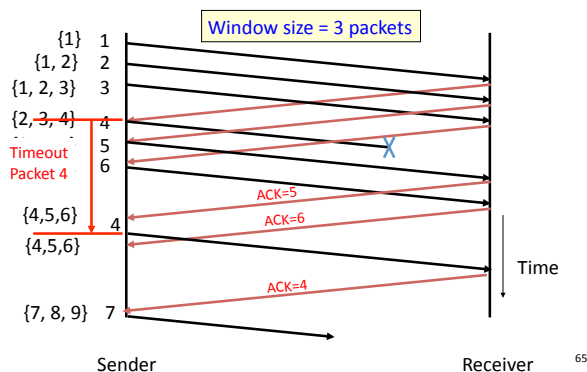
Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size 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)

66

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

67

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 (next time)
- Introduces timeout estimation algorithms (next time)

More in Topic 5b

Automatic Repeat Request (ARQ)

+ Self-clocking (Automatic)   Next lets move from
                                     the generic to the
+ Adaptive   specific….

+ Flexible   TCP arguably the most
                                     successful protocol in the
- Slow to start / adapt   Internet…..
consider high Bandwidth/Delay product

                                     its an ARQ protocol

69

TCP Header

Used to mux and demux

Source port	Destination port
Sequence number	
Acknowledgment	
HdrLen	0	Flags	Advertised window
Checksum	Urgent pointer
Options (variable)	
Data	
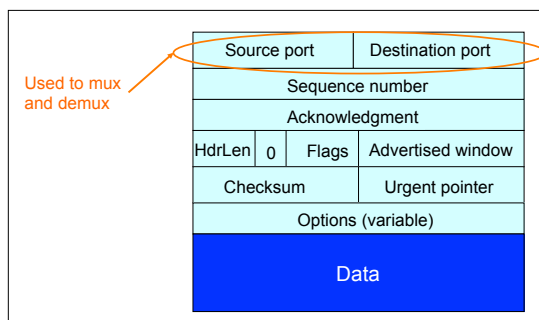
70

Last time: Components of a solution for reliable transport

- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - cumulative
  - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)
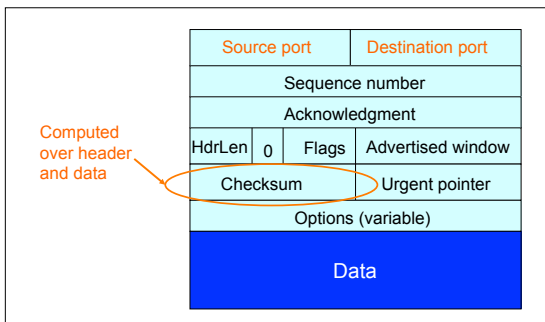 - Go-Back-N (GBN)
  - Selective Replay (SR)

What does TCP do?

Many of our previous ideas, but some key differences
- Checksum

TCP Header

Source port	Destination port
Sequence number	
Acknowledgment	

Computed over header and data

HdrLen	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			
Data			

What does TCP do?

Many of our previous ideas, but some key differences
- Checksum
- **Sequence numbers are byte offsets**

TCP: Segments and Sequence Numbers

TCP "Stream of Bytes" Service…

Application @ Host A
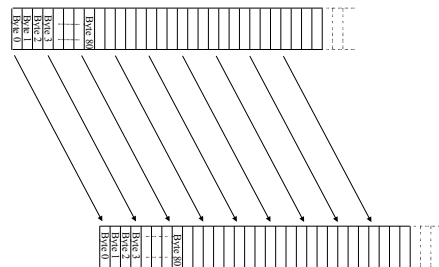
Application @ Host B

… Provided Using TCP "Segments"

Host A

TCP Data

Segment sent when:
1. Segment full (Max Segment Size),
2. Not full, but times out

TCP Data

Host B

TCP Segment

IP Data

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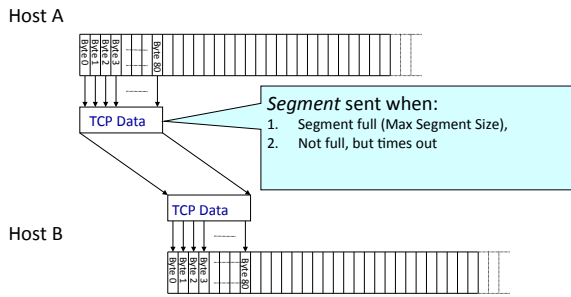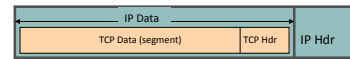
- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes with Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header ≥ 20 bytes long
- TCP **segment**
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes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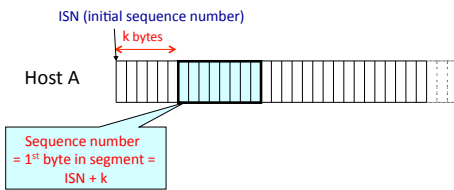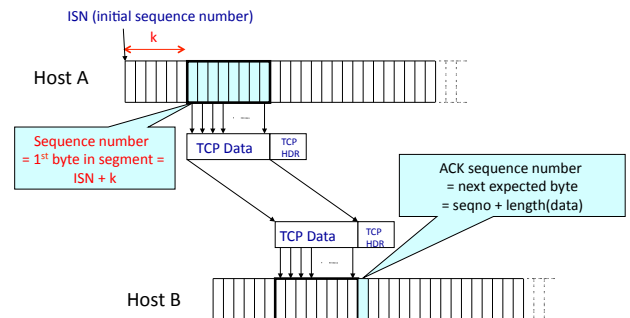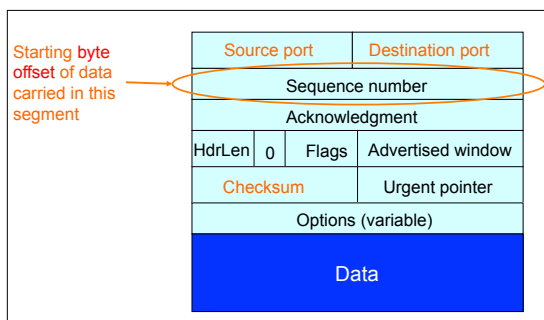
Sequence Numbers

ISN (initial sequence number)

k bytes

Host A

Sequence number
= 1st byte in segment =
ISN + k

Sequence Numbers

ISN (initial sequence number)

k

Host A

Sequence number
= 1st byte in segment =
ISN + k

TCP Data | TCP HDR

ACK sequence number
= next expected byte
= seqno + length(data)

TCP Data | TCP HDR

Host B

TCP Header

Starting byte offset of data carried in this segment

Source port	Destination port
Sequence number	
Acknowledgment	
HdrLen	0	Flags	Advertised window
Checksum	Urgent pointer
Options (variable)	
Data	

- What does TCP do?

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)

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

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
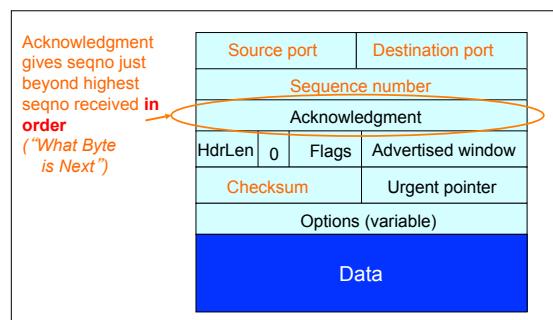
TCP Header

Acknowledgment gives seqno just beyond highest seqno received **in order**
(*"What Byte is Next"*)

Source port	Destination port
Sequence number	
Acknowledgment	

HdrLen	0	Flags	Advertised window

Checksum	Urgent pointer
Options (variable)	
Data	

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)

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,…

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

89

Loss with cumulative ACKs

- "Duplicate ACKs" are a sign of an <u>isolated</u> 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....

90

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?

91

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

92

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?

93

Timing Illustration

Timeout too long → inefficient

Timeout too short → duplicate packets
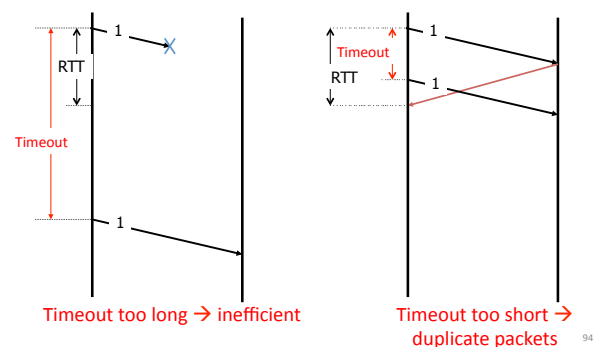
94

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?

RTT Estimation

- Use exponential averaging of RTT samples

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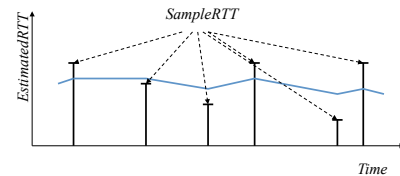
Exponential Averaging Example

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

Assume RTT is constant → $SampleRTT$ = RTT

$EstimatedRTT$ ($\alpha = 0.5$)

$EstimatedRTT$ ($\alpha = 0.8$)
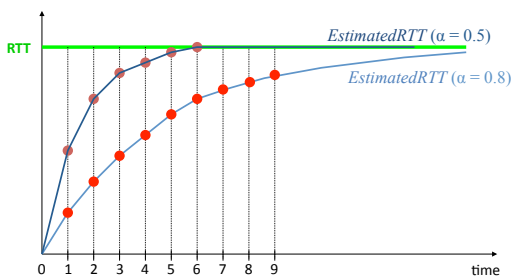
Problem: Ambiguous Measurements

- How do we differentiate between the real ACK, and ACK of the retransmitted packet?
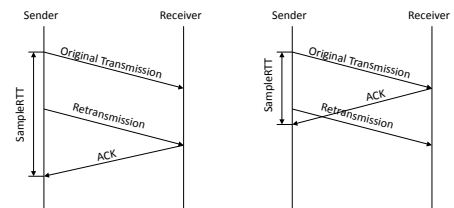
Karn/Partridge Algorithm

- 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 ← 2·RTO
  - (Up to maximum ≥ 60 sec)
  - Every time new measurement comes in (= successful original transmission), collapse RTO back to 2 × EstimatedRTT

Karn/Partridge in action

Figure 5: Performance of an RFC793 retransmit timer
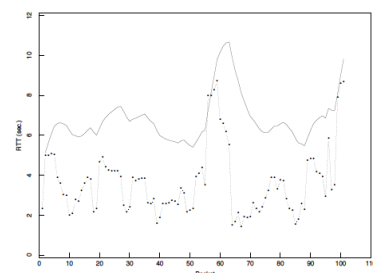
from Jacobson and Karels, SIGCOMM 1988

Jacobson/Karels Algorithm

- Problem: need to better capture variability in RTT
  - Directly measure deviation

- Deviation = | SampleRTT – EstimatedRTT |
- EstimatedDeviation: exponential average of Deviation

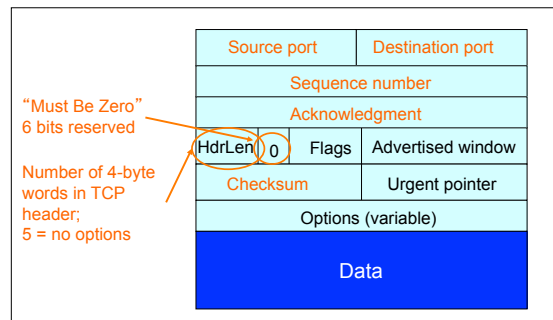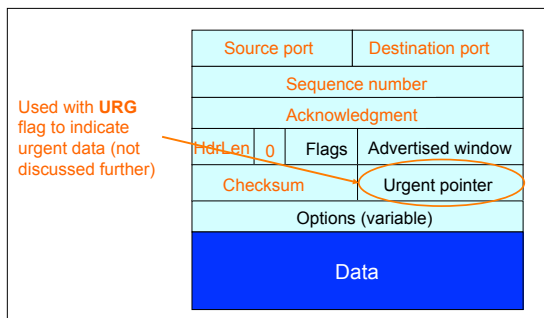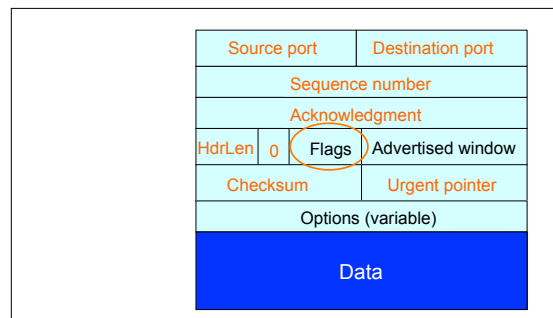- RTO = EstimatedRTT + 4 x EstimatedDeviation

With Jacobson/Karels
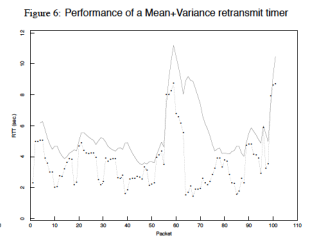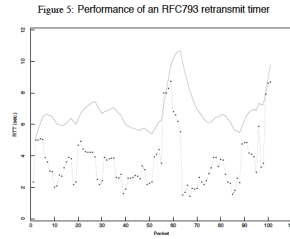
Figure 5: Performance of an RFC793 retransmit timer

Figure 6: Performance of a Mean+Variance retransmit timer

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

TCP Header: What's left?

"Must Be Zero"
6 bits reserved

Number of 4-byte words in TCP header;
5 = no options

Source port	Destination port
Sequence number	
Acknowledgment	
HdrLen 0 Flags	Advertised window
Checksum	Urgent pointer
Options (variable)	
Data	

TCP Header: What's left?

Used with **URG** flag to indicate urgent data (not discussed further)

Source port	Destination port
Sequence number	
Acknowledgment	
HdrLen 0 Flags	Advertised window
Checksum	Urgent pointer
Options (variable)	
Data	

TCP Header: What's left?

Source port	Destination port
Sequence number	
Acknowledgment	
HdrLen 0 Flags	Advertised window
Checksum	Urgent pointer
Options (variable)	
Data	

TCP Connection Establishment and Initial Sequence Numbers

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

Establishing a TCP Connection
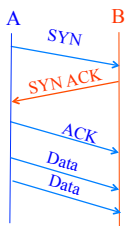
Each host tells its ISN to the other host.

- 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

TCP Header

Flags: **SYN**
ACK
FIN
RST
PSH
URG
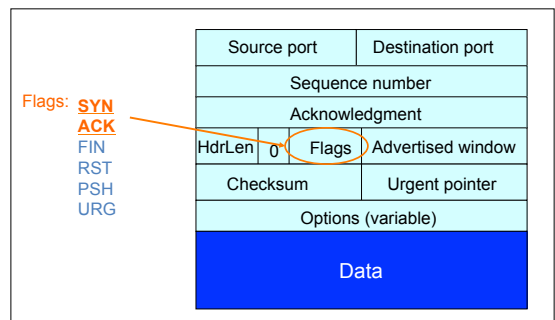
Step 1: A's Initial SYN Packet

Flags: SYN
ACK
FIN
RST
PSH
URG

A tells B it wants to open a connection…

Step 2: B's SYN-ACK Packet

Flags: SYN
ACK
FIN
RST
PSH
URG

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

… upon receiving this packet, A can start sending data
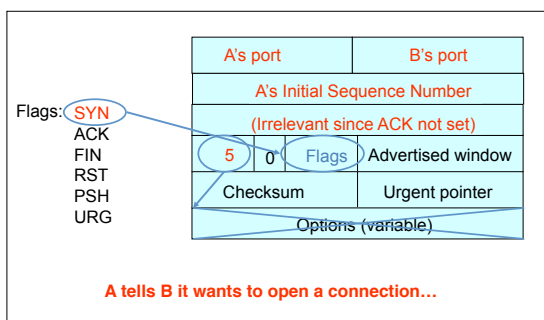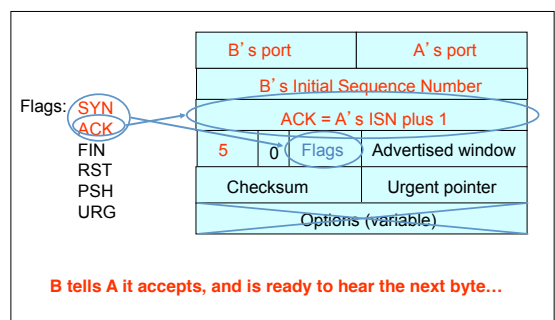
Step 3: A's ACK of the SYN-ACK

A's port	B's port
A's Initial Sequence Number	
B's ISN plus 1	

Flags: SYN
ACK
FIN
RST
PSH
URG

20B	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			

A tells B it's likewise okay to start sending
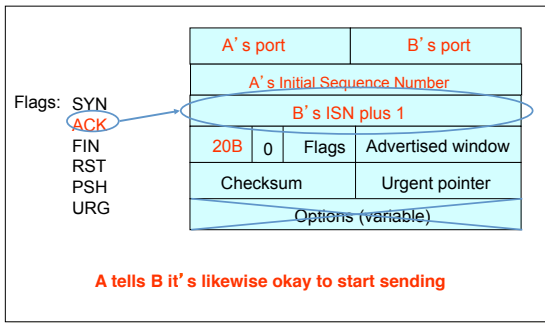
… upon receiving this packet, B can start sending data   113

Timing Diagram: 3-Way Handshaking

Passive Open

Active Open

Server

Client (initiator)

`connect()`

`listen()`

SYN, SeqNum = x

SYN + ACK, SeqNum = y, Ack = x + 1

ACK, Ack = y + 1

114
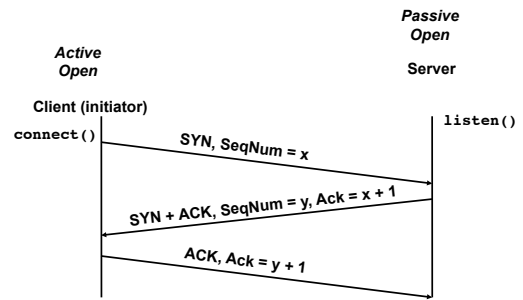
What if the SYN Packet Gets Lost?

- 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

115

SYN Loss and Web Downloads

- User clicks on a hypertext link
  – Browser creates a socket and does a "connect"
  – The "connect" triggers the OS to transmit a SYN

- If the SYN is lost…
  – 3-6 seconds of delay: can be very long
  – User may become impatient
  – … and click the hyperlink again, or click "reload"

- User triggers an "abort" of the "connect"
  – Browser creates a new socket and another "connect"
  – Essentially, forces a faster send of a new SYN packet!
 – Sometimes very effective, and the page comes quickly

116

Tearing Down the Connection

117

Normal Termination, One Side At A Time

B

SYN   SYN ACK   ACK   Data   ACK   •••   FIN   ACK   FIN   ACK

A
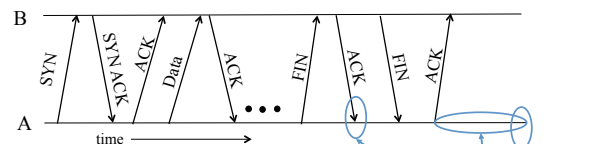
time

- 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   118

Normal Termination, Both Together

TIME_WAIT:
Avoid reincarnation
Can retransmit
FIN ACK if ACK lost

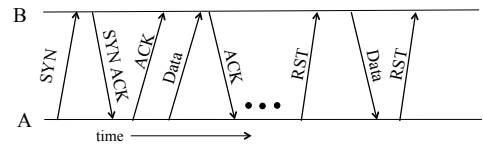Connection now closed
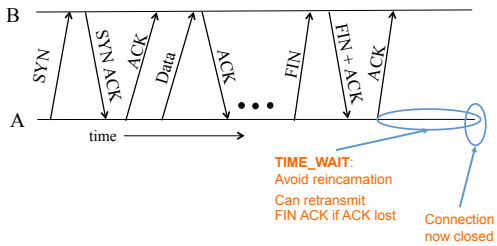
- Same as before, but B sets **FIN** with their ack of A's **FIN**

119

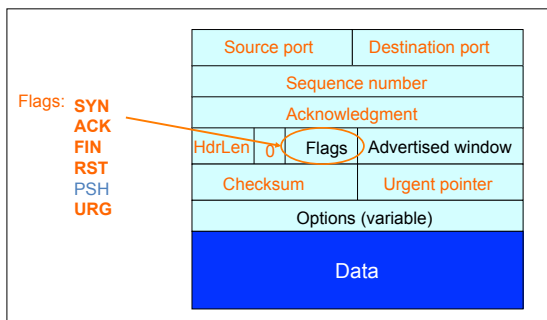## Abrupt Termination

- 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
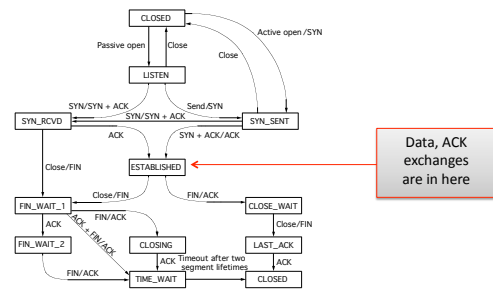 - And: any data in flight is lost
  - But: if B sends anything more, will elicit another **RST**

120

TCP Header

Flags: **SYN ACK FIN RST** PSH **URG**

Source port	Destination port
Sequence number	
Acknowledgment	
HdrLen 0 Flags	Advertised window
Checksum	Urgent pointer
Options (variable)	
Data	

121

TCP State Transitions

Data, ACK exchanges are in here

122

An Simpler View of the Client Side

SYN (Send)

CLOSED

TIME_WAIT

Rcv. FIN, Send ACK

FIN_WAIT2

Rcv. ACK, Send Nothing

FIN_WAIT1

Send FIN

ESTABLISHED

SYN_SENT

Rcv. SYN+ACK, Send ACK

123

TCP Header

Used to negotiate use of additional features
(details in section)

Source port	Destination port
Sequence number	
Acknowledgment	
HdrLen 0 Flags	Advertised window
Checksum	Urgent pointer
Options (variable)	
Data	

124

TCP Header

Source port	Destination port
Sequence number	
Acknowledgment	
HdrLen	0	Flags	Advertised window
Checksum	Urgent pointer
Options (variable)	
Data	

125

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

126

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

127

Sliding Window at Sender (so far)

Sending process

TCP

Buffer size (B)

Previously ACKed bytes

Last byte written

First unACKed byte

Last byte can send
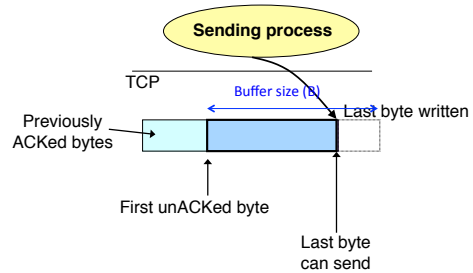
128

Sliding Window at Receiver (so far)

Receiving process

Last byte read

Buffer size (B)

Received and ACKed

Next byte needed (1st byte not received)

Last byte received

Sender might overrun the receiver's buffer
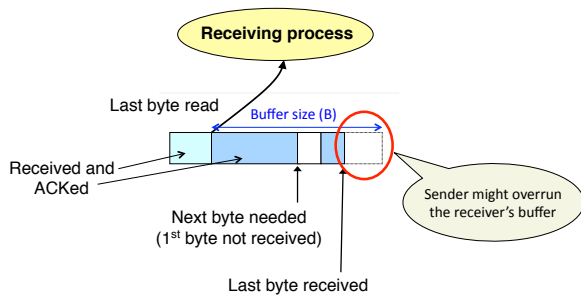
129

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

130

Sliding Window at Receiver

$W = B - (LastByteReceived - LastByteRead)$

Last byte read

Buffer size (B)

Next byte needed
(1st byte not received)

Last byte received

131

Sliding Window at Sender (so far)

Sending process

TCP

W

Last byte written

First unACKed byte

Last byte
can send
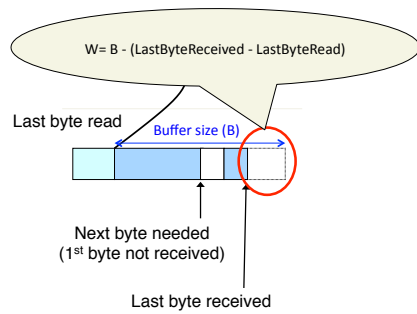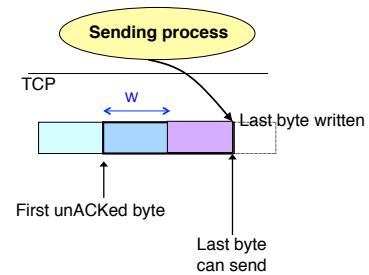
132

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

133

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?

134

TCP

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

135

TCP

- The concepts underlying TCP are simple
- But tricky in the details
  – How do we set timers?
  – What is the seqno for an ACK-only packet?
  – What happens if advertised window = 0?
  – What if the advertised window is ½ an MSS?
  – Should receiver acknowledge packets right away?
  – What if the application generates data in units of 0.1 MSS?
  – What happens if I get a duplicate SYN? Or a RST while I'm in FIN_WAIT, *etc., etc., etc.*

136

TCP

- The concepts underlying TCP are simple
- But tricky in the details
- Do the details matter?

Sizing Windows for Congestion Control

- What are the problems?
- How might we address them?

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

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*

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
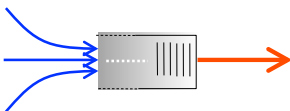
Congestion is undesirable

Typical queuing system with bursty arrivals

Average Packet delay

Load

Average Packet loss

Load
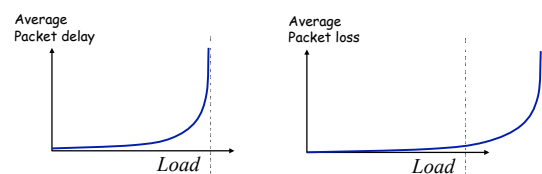
Must balance utilization versus delay and loss

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

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

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

Three Issues to Consider

- Discovering the available (bottleneck) bandwidth

- Adjusting to variations in bandwidth

- Sharing bandwidth between flows

Abstract View

Sending Host   Buffer in Router   Receiving Host
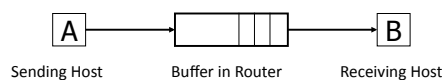
- Ignore internal structure of router and model it as having a single queue for a particular input-output pair

Discovering available bandwidth

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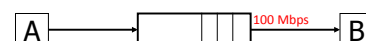
- Pick sending rate to match bottleneck bandwidth
  - Without any *a priori* knowledge
  - Could be gigabit link, could be a modem
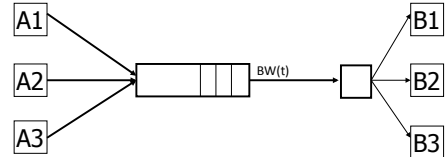
Adjusting to variations in bandwidth

A → [ ][ ][ ] BW(t) → B
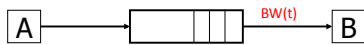
- Adjust rate to match **instantaneous** bandwidth
  - Assuming you have rough idea of bandwidth
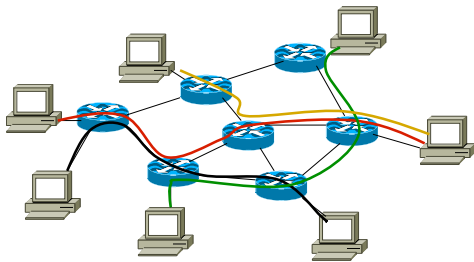
149

Multiple flows and sharing bandwidth

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

A1
A2   [ ][ ][ ] BW(t) [ ]
A3

B1
B2
B3

150

Reality

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

151

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

knee   cliff   packet loss

Throughput

congestion collapse

Load

Delay

Load
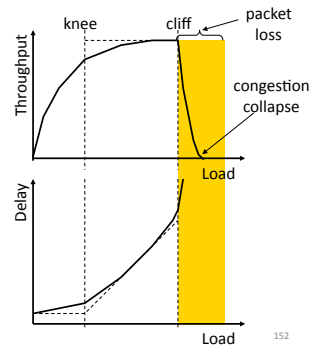
152

General Approaches

(0) Send without care
  - Many packet drops

153

General Approaches

(0) Send without care
(1) Reservations
  - Pre-arrange bandwidth allocations
  - Requires negotiation before sending packets
  - Low utilization

154

General Approaches

(0) Send without care
(1) Reservations
(2) Pricing
  – Don't drop packets for the high-bidders
  – Requires payment model

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

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

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

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 lecture that RWND >> CWND

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

• Keep in mind that real implementations maintain CWND in bytes

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

Detecting Congestion

- Packet delays
  - Tricky: noisy signal (delay often varies considerably)

- Router tell endhosts 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

Not All Losses the Same

- Duplicate ACKs: isolated loss
  - Still getting ACKs

- Timeout: much more serious
  - Not enough dupacks
  - Must have suffered several losses

- Will adjust rate differently for each case

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

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!

"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

Slow Start in Action

- For each RTT: double CWND

- Simpler implementation: for each ACK, CWND += 1
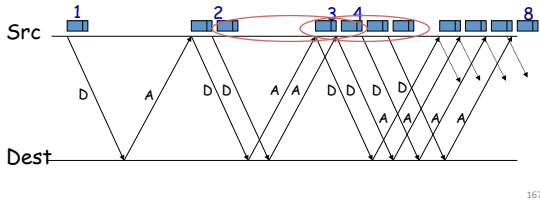
Adjusting to Varying Bandwidth

- 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)

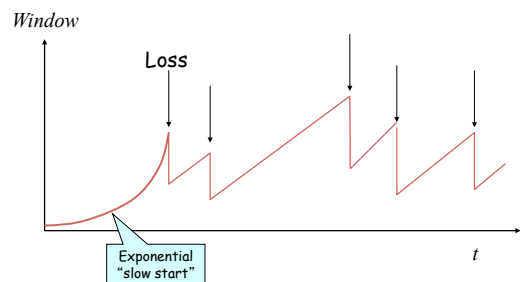- TCP uses: "Additive Increase Multiplicative Decrease" (AIMD)
  - We'll see why shortly…

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

Leads to the TCP "Sawtooth"

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

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

Why AIMD?

Recall: Three Issues

- Discovering the available (bottleneck) bandwidth
  - Slow Start

- Adjusting to variations in bandwidth
  - AIMD

- Sharing bandwidth between flows

Goals for bandwidth sharing

- Efficiency: High utilization of link bandwidth
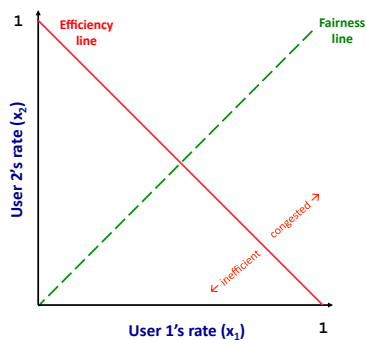- Fairness: Each flow gets equal share

---

Why AIMD?

- Some rate adjustment options: Every RTT, we can
  - Multiplicative increase or decrease: CWND→ a*CWND
  - Additive increase or decrease: CWND→ CWND + b

- Four alternatives:
  - AIAD: gentle increase, gentle decrease
  - AIMD: gentle increase, drastic decrease
  - MIAD: drastic increase, gentle decrease
  - MIMD: drastic increase and decrease

Simple Model of Congestion Control

- Two users
  - rates x_1 and x_2

- Congestion when $x_1 + x_2 > 1$
- Unused capacity when $x_1 + x_2 < 1$
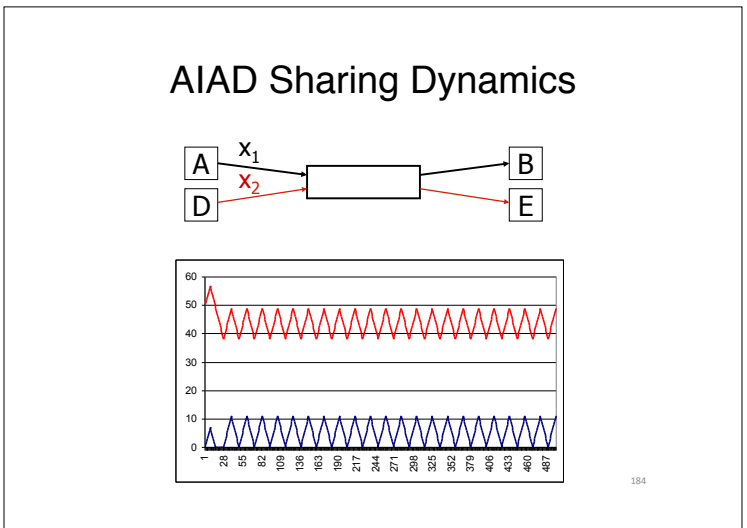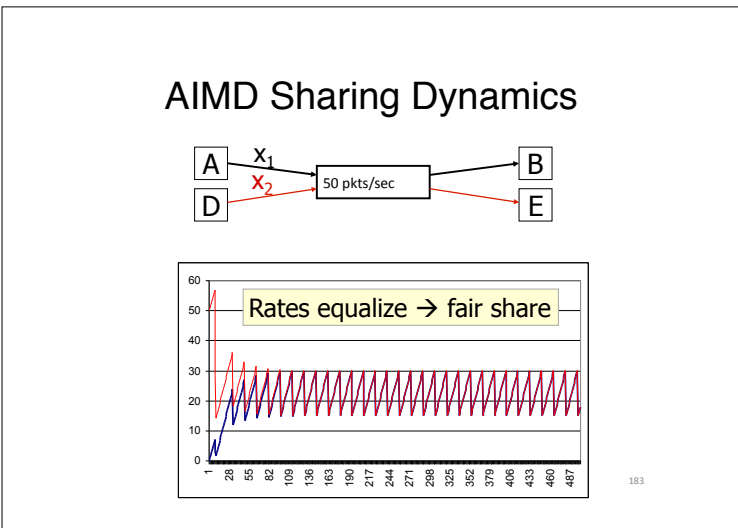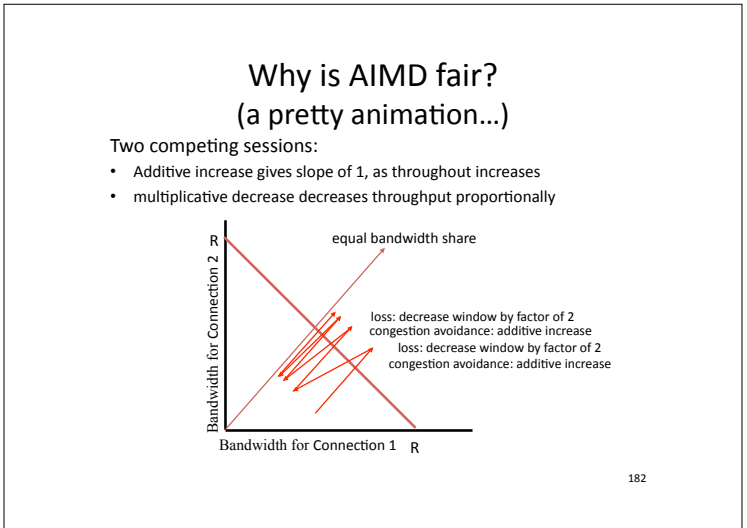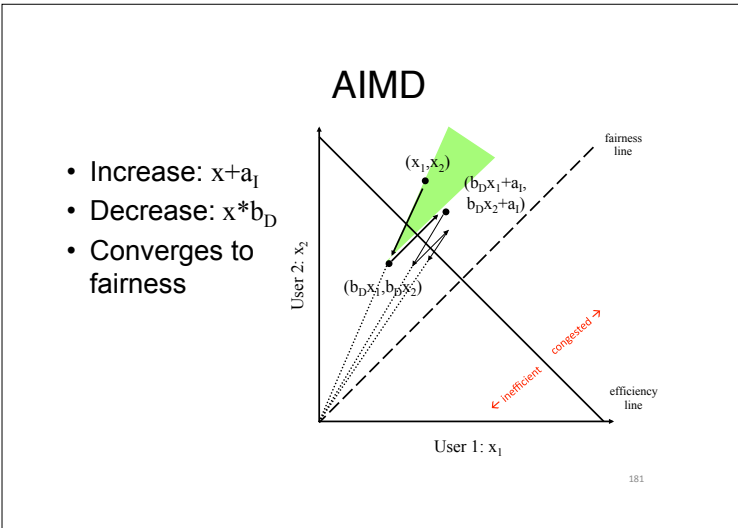
- Fair when $x_1 = x_2$

Efficiency line
Fairness line
User 2's rate (x_2)
congested
inefficient
User 1's rate (x_1)

Example

fairness line
Efficient: $x_1 + x_2 = 1$ Fair
Congested: $x_1 + x_2 = 1.2$
User 2: x_2
(0.2, 0.5)
(0.5, 0.5)
(0.7, 0.5)
Inefficient: $x_1 + x_2 = 0.7$
(0.7, 0.3)
Efficient: $x_1 + x_2 = 1$ Not fair
efficiency line
User 1: x_1

AIAD

- Increase: $x + a_I$
- Decrease: $x - a_D$
- Does not converge to fairness

$(x_1-a_D+a_I, x_2-a_D+a_I))$
(x_1,x_2)
(x_1-a_D, x_2-a_D)
fairness line
inefficient
congested
efficiency line
User 2: x_2
User 1: x_1

MIMD

- Increase: $x*b_I$
- Decrease: $x*b_D$
- Does not converge to fairness

(x_1,x_2)
$(b_I b_D x_1, b_I b_D x_2)$
$(b_d x_1, b_d x_2)$
fairness line
inefficient
congested
efficiency line
User 2: x_2
User 1: x_1

AIMD

- Increase: $x+a_I$
- Decrease: $x*b_D$
- Converges to fairness

(x_1,x_2)
$(b_D x_1+a_I, b_D x_2+a_I)$
$(b_D x_1, b_D x_2)$
fairness line
inefficient
congested
efficiency line
User 2: x_2
User 1: x_1

Why is AIMD fair?
(a pretty animation...)

Two competing sessions:
- Additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally

R
equal bandwidth share
Bandwidth for Connection 2
loss: decrease window by factor of 2
congestion avoidance: additive increase
loss: decrease window by factor of 2
congestion avoidance: additive increase
Bandwidth for Connection 1   R

AIMD Sharing Dynamics

A — x_1
D — x_2
50 pkts/sec
B
E

Rates equalize → fair share

AIAD Sharing Dynamics

A — x_1
D — x_2
B
E

TCP Congestion Control Details

Implementation

- **State at sender**
  - CWND (initialized to a small constant)
  - ssthresh (initialized to a large constant)
  - [Also dupACKcount and timer, as before]

- **Events**
  - ACK (new data)
  - dupACK (duplicate ACK for old data)
  - Timeout

Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

  > - CWND packets per RTT
  > - Hence after one RTT with no drops:
  >   CWND = 2xCWND

Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

  Slow start phase

- Else
  - CWND = CWND + 1/CWND

  "Congestion Avoidance" phase (additive increase)

  > - CWND packets per RTT
  > - Hence after one RTT with no drops:
  >   CWND = CWND + 1

Event: TimeOut

- On Timeout
  - ssthresh ← CWND/2
  - CWND ← 1

Event: dupACK

- dupACKcount ++

- If dupACKcount = 3 /* fast retransmit */
  - ssthresh = CWND/2
  - CWND = CWND/2

Example
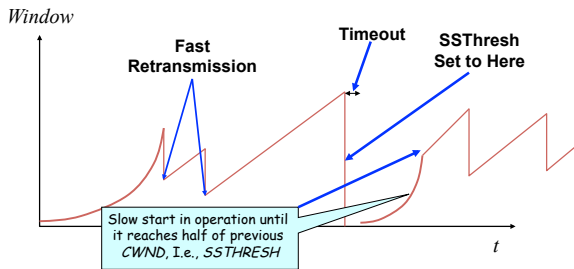
Slow-start restart: Go back to CWND = 1 MSS, but take advantage of knowing the previous value of CWND

191

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

One Final Phase: Fast Recovery

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

193

Example (in units of MSS, not bytes)
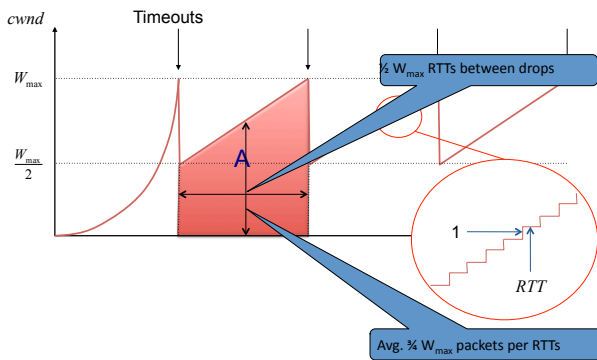
- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101

- 10 packets [101, 102, 103,…, 110] are in flight
  - Packet 101 is dropped
  - What ACKs do they generate?
  - And how does the sender respond?

194

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

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

196

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

197

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

Putting it all together:
The TCP State Machine (partial)
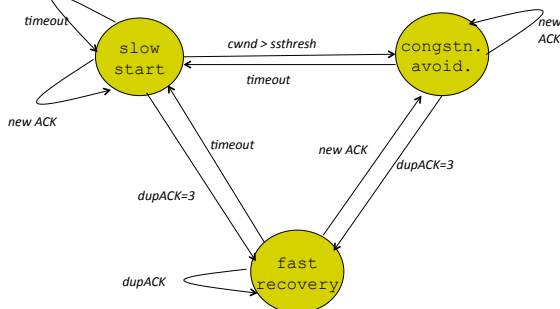
- How are ssthresh, CWND and dupACKcount updated for each event that causes a state transition?

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

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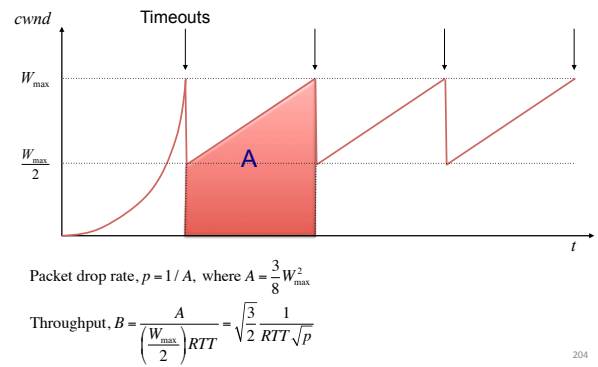 - AIMD, Fast-Recovery, Throughput

201

TCP Throughput Equation

202

A Simple Model for TCP Throughput

$cwnd$

Timeouts

W_{max}

$\frac{W_{max}}{2}$

A

½ W_{max} RTTs between drops

1 $\rightarrow$ RTT

Avg. ¾ W_{max} packets per RTTs

A Simple Model for TCP Throughput

$cwnd$

Timeouts

W_{max}

$\frac{W_{max}}{2}$

A

t

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

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

204

Some implications: (1) Fairness

Throughput, $B = \sqrt{\dfrac{3}{2}}\dfrac{1}{RTT\sqrt{p}}$

- Flows get throughput inversely proportional to RTT
  - Is this fair?

Some Implications: (2) How does this look at high speed?

- Assume that RTT = 100ms, MSS=1500bytes

- What value of p is required to go 100Gbps?
  - Roughly 2×10^{-12}
- How long between drops?
  - Roughly 16.6 hours
- How much data has been sent in this time?
  - Roughly 6 petabits

- These are not practical numbers!

206

Some implications: (3) Rate-based Congestion Control

Throughput, $B = \sqrt{\dfrac{3}{2}}\dfrac{1}{RTT\sqrt{p}}$

- One can dispense with TCP and just match eqtn:
  - Equation-based congestion control
  - Measure drop percentage p, and set rate accordingly
  - Useful for streaming applications

Some Implications: (4) Lossy Links

- TCP assumes all losses are due to congestion

- What happens when the link is lossy?

- Throughput ~ 1/sqrt(p) where p is loss prob.

- This applies even for non-congestion losses!

208

Other Issues: Cheating

- Cheating pays off

- Some favorite approaches to cheating:
  - Increasing CWND faster than 1 per RTT
  - Using large initial CWND
  - Opening many connections
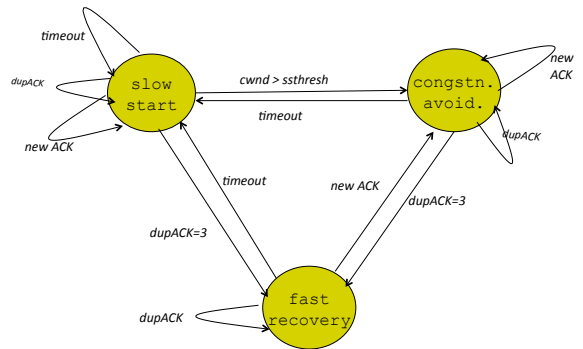
Increasing CWND Faster

x increases by 2 per RTT
y increases by 1 per RTT

Limit rates:
x = 2y
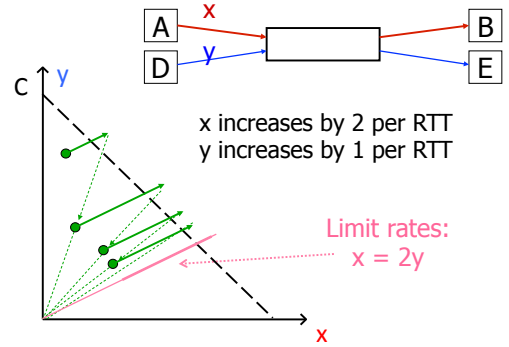
210

A Closer look at problems with TCP Congestion Control

211

TCP State Machine

212

TCP State Machine
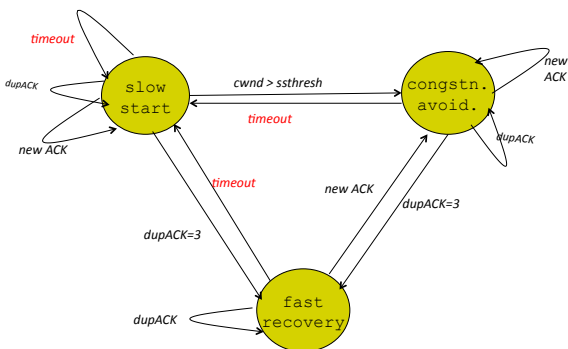
TCP State Machine
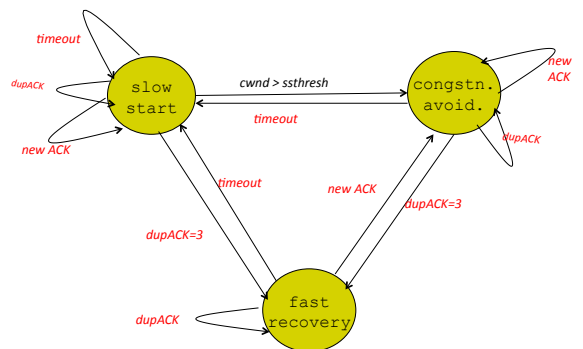
214

TCP State Machine
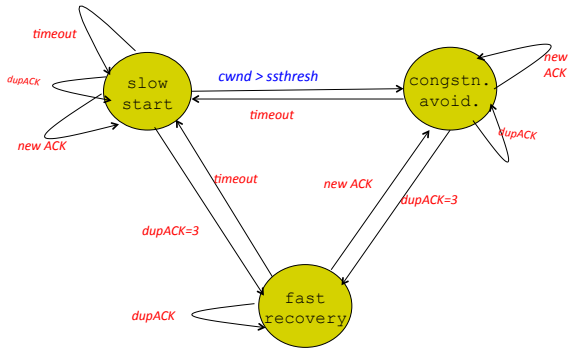
TCP Flavors

- 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
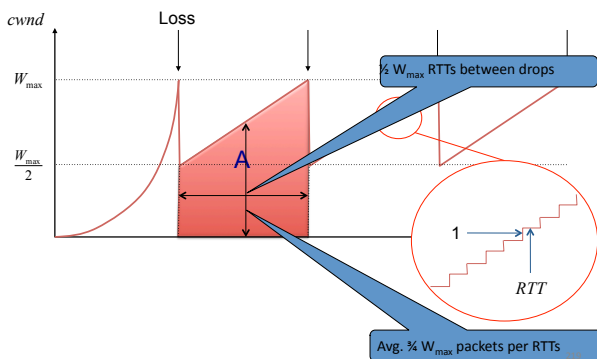
Our default assumption

216

Interoperability

- How can all these algorithms coexist? Don't we need a single, uniform standard?

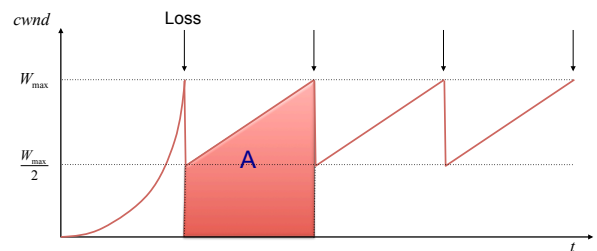- What happens if I'm using Reno and you are using Tahoe, and we try to communicate?

217

TCP Throughput Equation

218

A Simple Model for TCP Throughput

Loss

W_{max}

$\frac{W_{max}}{2}$

A

½ W_{max} RTTs between drops

1

RTT

Avg. ¾ W_{max} packets per RTTs

A Simple Model for TCP Throughput

Loss

W_{max}

$\frac{W_{max}}{2}$

A

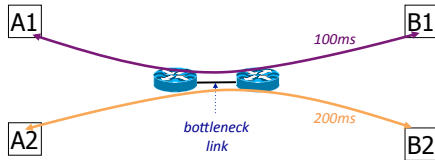Packet drop rate, $p = 1/A$, where $A = \frac{3}{8}W_{max}^2$

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

220

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!

A1 — B1 *100ms*

A2 — B2 *200ms*

bottleneck link

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
  - $\sim 2 \times 10^{-12}$
- How long between drops?
  - $\sim$ 16.6 hours
- How much data has been sent in this time?
  - $\sim$ 6 petabits
- These are not practical numbers!

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 (hack but works today)
  - Router-assisted approaches (will see shortly)

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

- 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

Other Limitations of TCP Congestion Control

(4) Loss not due to congestion?

- TCP will confuse **any loss event** with congestion

- Flow will cut its rate
  - Throughput ~ 1/sqrt(p) where p is loss prob.
  - Applies even for non-congestion losses!

- We'll look at proposed solutions shortly…

(5) How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB

- Implication (1): short flows never leave slow start!
  - short flows never attain their fair share

- Implication (2): too few packets to trigger dupACKs
  - Isolated loss may lead to timeouts
  - At typical timeout values of ~500ms, might severely impact flow completion time

(6) TCP fills up queues → long delays

- A flow deliberately overshoots capacity, until it experiences a drop

- Means that delays are large for *everyone*
  - Consider a flow transferring a 10GB file sharing a bottleneck link with 10 flows transferring 100B
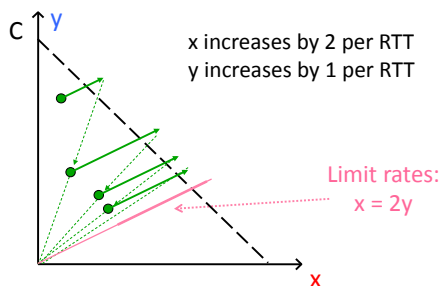
(7) Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT

Increasing CWND Faster

x increases by 2 per RTT
y increases by 1 per RTT

Limit rates:
x = 2y

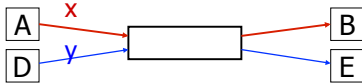## (7) Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT
  - Opening many connections

Open Many Connections

Assume
• A starts 10 connections to B
• D starts 1 connection to E
• Each connection gets about the same throughput

Then A gets 10 times more throughput than D

(7) Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT
  - Opening many connections
  - Using large initial CWND

- Why hasn't the Internet suffered a congestion collapse yet?

(8) CC intertwined with reliability

- Mechanisms for CC and reliability are tightly coupled
  - CWND adjusted based on ACKs and timeouts
  - Cumulative ACKs and fast retransmit/recovery rules

- Complicates evolution
  - Consider changing from cumulative to selective ACKs
  - A failure of modularity, not layering

- Sometimes we want CC but not reliability
  - e.g., real-time applications
- Sometimes we want reliability but not CC (?)

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!

- 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

Router-Assisted Congestion Control

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

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

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
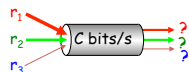
- What does "fair" mean exactly?

Max-Min Fairness

- Given set of bandwidth demands r_i and total bandwidth C, max-min bandwidth allocations are:
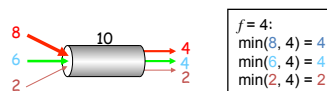$$a_i = \min(f, r_i)$$
where f is the unique value such that Sum(a_i) = C

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: $f = 4$

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

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 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

---

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

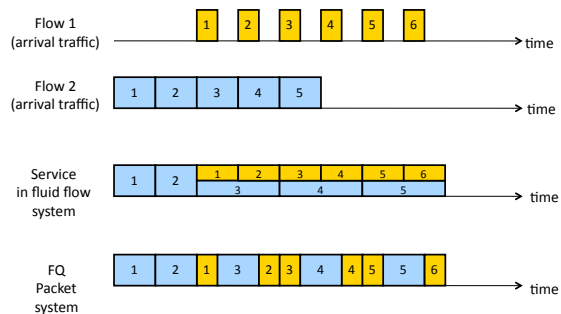- But we can approximate it
  - This is what "fair queuing" routers do

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

Example

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
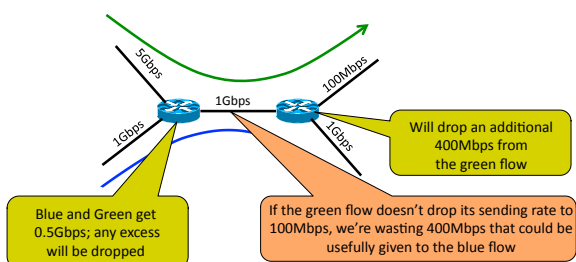 - Mostly used to isolate traffic at larger granularities (e.g., per-prefix)

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

FQ in the big picture

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

Blue and Green get 0.5Gbps; any excess will be dropped

If the green flow doesn't drop its sending rate to 100Mbps, we're wasting 400Mbps that could be usefully given to the blue flow

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?

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?
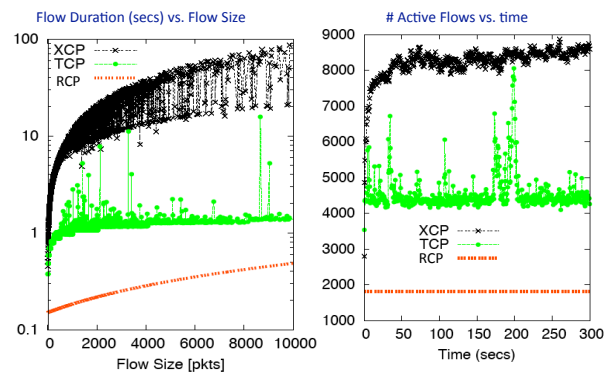 - TCP connection
  - Source-Destination pair?
  - Source?

Router-Assisted Congestion Control

- CC has three different tasks:
  - Isolation/fairness
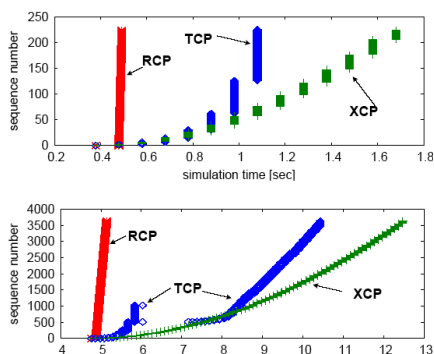 - Rate adjustment
  - Detecting congestion

Why not just let routers tell endhosts what rate they should use?

- Packets carry "rate field"

- Routers insert "fair share" *f* in packet header
  - Calculated as with FQ

- End-hosts set sending rate (or window size) to *f*
  - hopefully (still need some policing of endhosts!)

- This is the basic idea behind the "Rate Control Protocol" (RCP) from Dukkipati *et al.* '07

Flow Completion Time: TCP vs. RCP (Ignore XCP)

Why the improvement?

Router-Assisted Congestion Control

- CC has three different tasks:
  - Isolation/fairness
  - Rate adjustment
  - Detecting congestion

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)

257

One final proposal: Charge people for congestion!

- Use ECN as congestion markers

- Whenever I get an ECN bit set, I have to pay $$

- Now, there's no debate over what a flow is, or what fair is…

- Idea started by Frank Kelly here in Cambridge
  - "optimal" solution, backed by much math
  - Great idea: simple, elegant, effective
  - Unclear that it will impact practice – although London congestion works
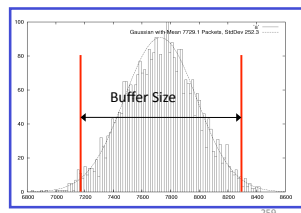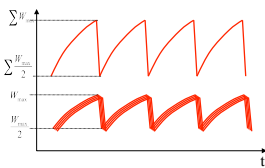
258

Some TCP issues outstanding…

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

259

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

260

Recap

- 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

261