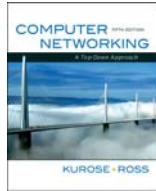


Chapter 3 Transport Layer



A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a lot of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form on a web site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! J.F.K./K.W.R.

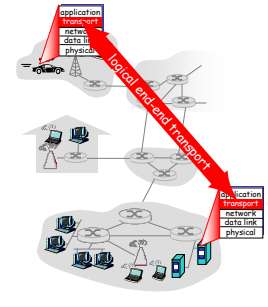
All material copyright 1996-2010
J.F. Kurose and K.W. Ross, All Rights Reserved

*Computer Networking:
A Top Down Approach
5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April
2009.*

Transport Layer 3-1

Transport services and protocols

- ❖ provide **logical communication** between app processes running on different hosts
- ❖ transport protocols run in end systems
 - send side: breaks app messages into **segments**, passes to network layer
 - rcv side: reassembles segments into messages, passes to app layer
- ❖ more than one transport protocol available to apps
 - Internet: TCP and UDP



Transport Layer 3-4

Chapter 3: Transport Layer

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

Transport Layer 3-2

Transport vs. network layer

- ❖ **network layer:** logical communication between hosts
- ❖ **transport layer:** logical communication between processes
 - relies on, enhances, network layer services

Household analogy:

- 12 kids sending letters to 12 kids
- ❖ processes = kids
 - ❖ app messages = letters in envelopes
 - ❖ hosts = houses
 - ❖ transport protocol = Ann and Bill who demux to in-house siblings
 - ❖ network-layer protocol = postal service

Transport Layer 3-5

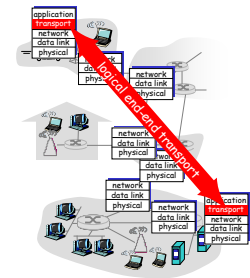
Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

Transport Layer 3-3

Internet transport-layer protocols

- ❖ reliable, in-order delivery (TCP)
 - congestion control
 - flow control
 - connection setup
- ❖ unreliable, unordered delivery: UDP
 - no-frills extension of "best-effort" IP
- ❖ services not available:
 - delay guarantees
 - bandwidth guarantees



Transport Layer 3-6

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

Transport Layer 3-7

Connectionless demultiplexing

- ❖ *recall*: create sockets with host-local port numbers:


```
DatagramSocket mySocket1 = new DatagramSocket(12534);
DatagramSocket mySocket2 = new DatagramSocket(12535);
```
- ❖ *recall*: when creating datagram to send into UDP socket, must specify (dest IP address, dest port number)
- ❖ when host receives UDP segment:
 - checks destination port number in segment
 - directs UDP segment to socket with that port number
- ❖ IP datagrams with different source IP addresses and/or source port numbers directed to same socket

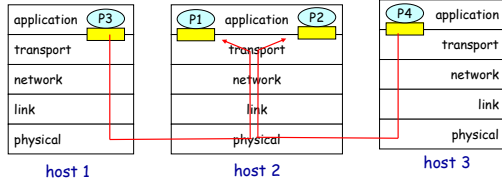
Transport Layer 3-10

Multiplexing/demultiplexing

Demultiplexing at rcv host:
delivering received segments to correct socket

Multiplexing at send host:
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

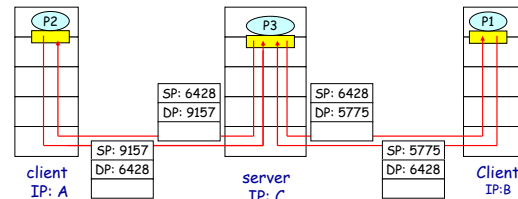
yellow rectangle = socket, blue oval = process



Transport Layer 3-8

Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

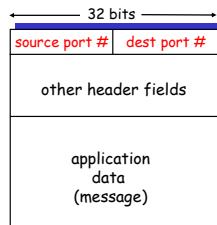


SP provides "return address"

Transport Layer 3-11

How demultiplexing works

- ❖ *host receives IP datagrams*
 - each datagram has source IP address, destination IP address
 - each datagram carries 1 transport-layer segment
 - each segment has source, destination port number
- ❖ *host uses IP addresses & port numbers to direct segment to appropriate socket*



TCP/UDP segment format

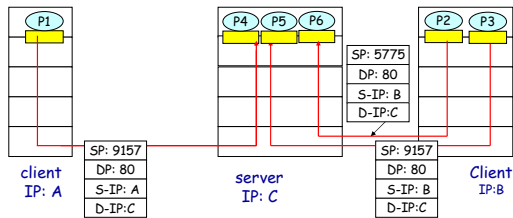
Transport Layer 3-9

Connection-oriented demux

- ❖ TCP socket identified by 4-tuple:
 - source IP address
 - source port number
 - dest IP address
 - dest port number
- ❖ *recv host uses all four values to direct segment to appropriate socket*
- ❖ server host may support many simultaneous TCP sockets:
 - each socket identified by its own 4-tuple
- ❖ web servers have different sockets for each connecting client
 - non-persistent HTTP will have different socket for each request

Transport Layer 3-12

Connection-oriented demux (cont)



Transport Layer 3-13

UDP: User Datagram Protocol [RFC 768]

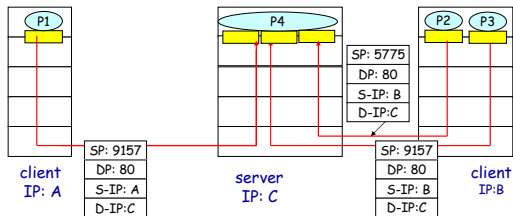
- ❖ "no frills," "bare bones" Internet transport protocol
- ❖ "best effort" service, UDP segments may be:
 - lost
 - delivered out of order to app
- ❖ **connectionless:**
 - no handshaking between UDP sender, receiver
 - each UDP segment handled independently of others

Why is there a UDP?

- ❖ no connection establishment (which can add delay)
- ❖ simple: no connection state at sender, receiver
- ❖ small segment header
- ❖ no congestion control: UDP can blast away as fast as desired
- ❖ Voice?

Transport Layer 3-16

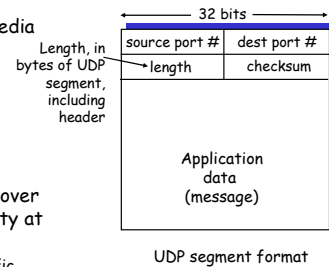
Connection-oriented demux: Threaded Web Server



Transport Layer 3-14

UDP: more

- ❖ often used for streaming multimedia apps
 - loss tolerant
 - rate sensitive
- ❖ other UDP uses
 - DNS
 - SNMP
- ❖ reliable transfer over UDP: add reliability at application layer
 - application-specific error recovery!



Transport Layer 3-17

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 **Connectionless transport: UDP**
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

Transport Layer 3-15

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

- ❖ treat segment contents as sequence of 16-bit integers
- ❖ checksum: addition (1's complement sum) of segment contents
- ❖ sender puts checksum value into UDP checksum field

Receiver:

- ❖ compute checksum of received segment
- ❖ check if computed checksum equals checksum field value:
 - NO - error detected
 - YES - no error detected. *But maybe errors nonetheless? More later*

Transport Layer 3-18

Internet Checksum Example

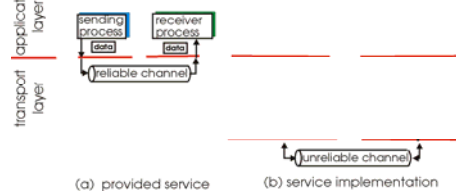
- ❖ Note: when adding numbers, a carryout from the most significant bit needs to be added to the result
- ❖ Example: add two 16-bit integers

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	0	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Transport Layer 3-19

Principles of Reliable data transfer

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



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

Transport Layer 3-22

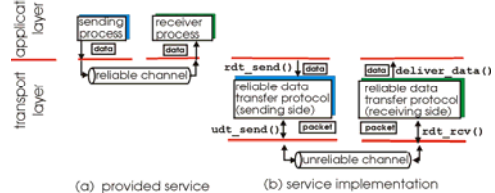
Chapter 3 outline

- | | |
|--|--|
| 3.1 Transport-layer services | 3.5 Connection-oriented transport: TCP |
| 3.2 Multiplexing and demultiplexing | ▪ segment structure |
| 3.3 Connectionless transport: UDP | ▪ reliable data transfer |
| 3.4 Principles of reliable data transfer | ▪ flow control |
| | ▪ connection management |
| | 3.6 Principles of congestion control |
| | 3.7 TCP congestion control |

Transport Layer 3-20

Principles of Reliable data transfer

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

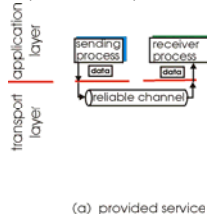


- ❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Transport Layer 3-23

Principles of Reliable data transfer

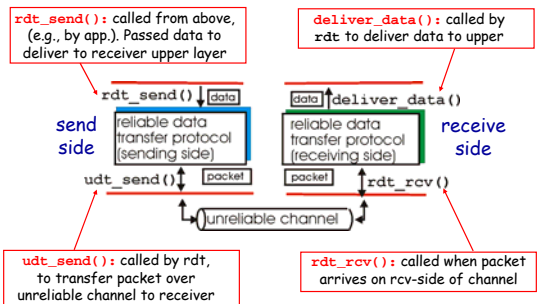
- ❖ important in app., transport, link layers
- ❖ top-10 list of important networking topics!



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

Transport Layer 3-21

Reliable data transfer: getting started



Transport Layer 3-24

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



Transport Layer 3-25

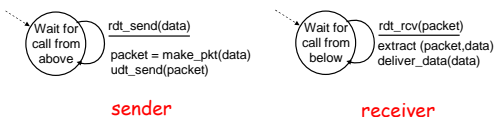
Rdt2.0: channel with bit errors

- 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 pkt received OK
 - negative acknowledgements (NAKs): receiver explicitly tells sender that pkt had errors
 - sender retransmits pkt on receipt of NAK
- new mechanisms in rdt2.0 (beyond rdt1.0):
 - error detection
 - receiver feedback: control msgs (ACK,NAK) rcvr→sender

Transport Layer 3-28

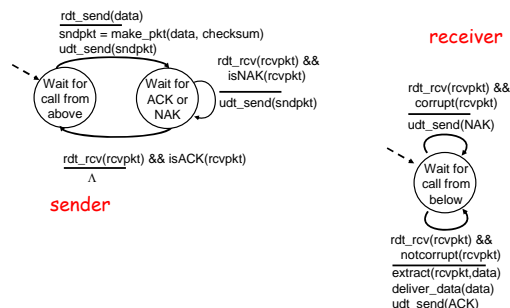
Rdt1.0: reliable transfer over a reliable channel

- 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



Transport Layer 3-26

rdt2.0: FSM specification



Transport Layer 3-29

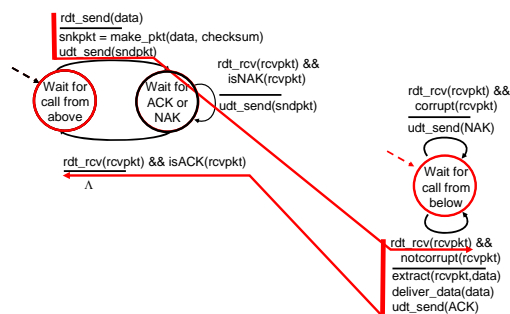
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
 - checksum to detect bit errors
- the question: how to recover from errors:

How do humans recover from "errors" during conversation?

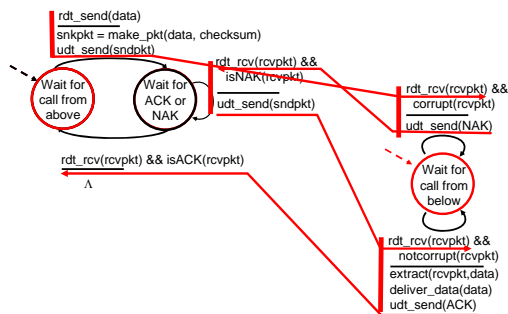
Transport Layer 3-27

rdt2.0: operation with no errors



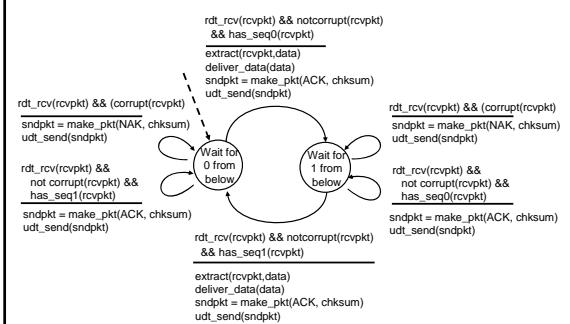
Transport Layer 3-30

rdt2.0: error scenario



Transport Layer 3-31

rdt2.1: receiver, handles garbled ACK/NAKs



Transport Layer 3-34

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 pkt if ACK/NAK garbled
- ❖ sender adds *sequence number* to each pkt
- ❖ receiver discards (doesn't deliver up) duplicate pkt

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

Transport Layer 3-32

rdt2.1: discussion

Sender:

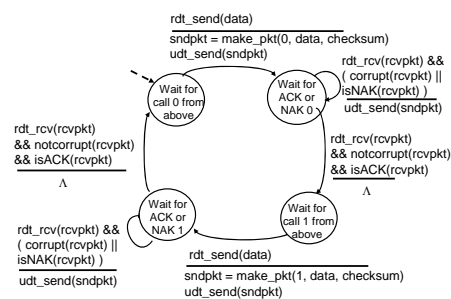
- ❖ seq # added to pkt
- ❖ two seq. #'s (0,1) will suffice. Why?
- ❖ must check if received ACK/NAK corrupted
- ❖ twice as many states
 - state must "remember" whether "current" pkt has 0 or 1 seq. #

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

Transport Layer 3-35

rdt2.1: sender, handles garbled ACK/NAKs



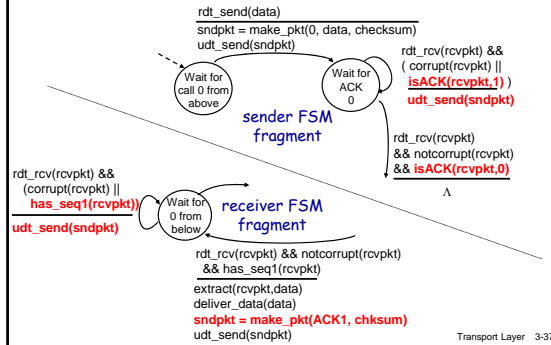
Transport Layer 3-33

rdt2.2: a NAK-free protocol

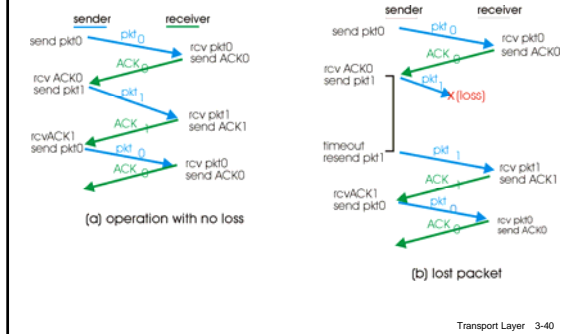
- ❖ same functionality as rdt2.1, using ACKs only
- ❖ instead of NAK, receiver sends ACK for last pkt received OK
 - receiver must *explicitly* include seq # of pkt being ACKed
- ❖ duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

Transport Layer 3-36

rdt2.2: sender, receiver fragments



rdt3.0 in action



rdt3.0: channels with errors and loss

New assumption:

underlying channel can also lose packets (data or ACKs)

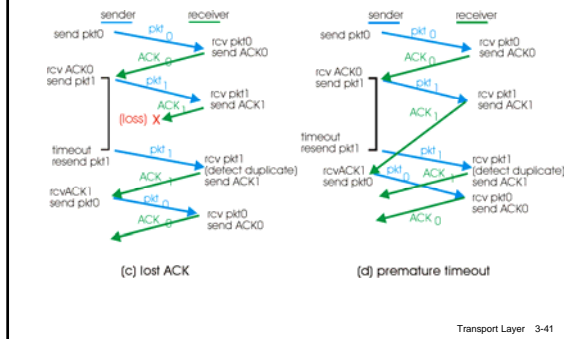
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Approach: sender waits "reasonable" amount of time for ACK

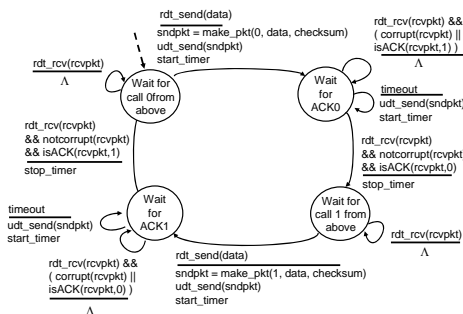
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
 - retransmission will be duplicate, but use of seq. #'s already handles this
 - receiver must specify seq. # of pkt being ACKed
- requires countdown timer

Transport Layer 3-38

rdt3.0 in action



rdt3.0 sender



Performance of rdt3.0

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

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

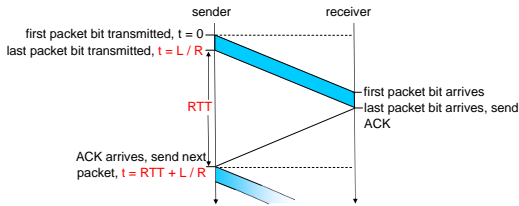
- U_{sender} : utilization - fraction of time sender busy sending

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

- if RTT=30 msec, 1KB pkt every 30 msec → 33KB/sec thrupt over 1 Gbps link
- network protocol limits use of physical resources!

Transport Layer 3-42

rdt3.0: stop-and-wait operation



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

Transport Layer 3-43

Pipelined Protocols

Go-back-N: big picture:

- ❖ sender can have up to N unacked packets in pipeline
- ❖ rcvr only sends *cumulative* acks
 - doesn't ack packet if there's a gap
- ❖ sender has timer for oldest unacked packet
 - if timer expires, retransmit all unack'd packets

Selective Repeat: big pic

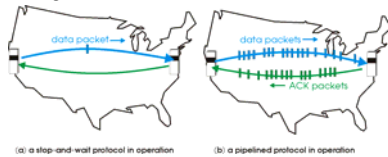
- ❖ sender can have up to N unacked packets in pipeline
- ❖ rcvr sends *individual ack* for each packet
- ❖ sender maintains timer for each unacked packet
 - when timer expires, retransmit only unack'd packet

Transport Layer 3-46

Pipelined 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



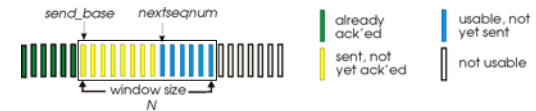
- ❖ two generic forms of pipelined protocols: *go-Back-N*, *selective repeat*

Transport Layer 3-44

Go-Back-N

Sender:

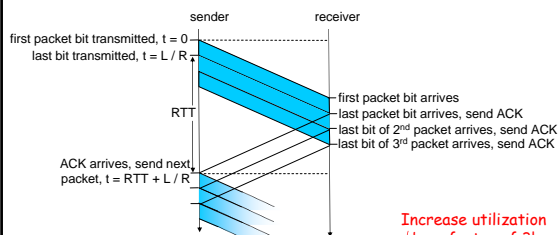
- ❖ k-bit seq # in pkt header
- ❖ "window" of up to N, consecutive unack'd pkts allowed



- ❖ ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
 - may receive duplicate ACKs (see receiver)
- ❖ timer for each in-flight pkt
- ❖ *timeout(n)*: retransmit pkt n and all higher seq # pkts in window

Transport Layer 3-47

Pipelining: increased utilization

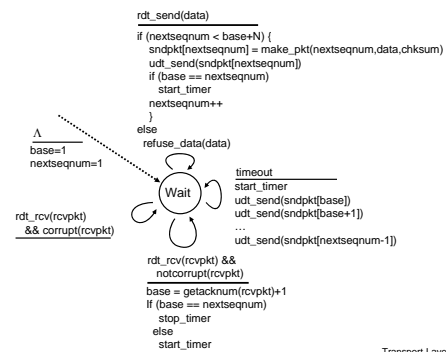


$$U_{\text{sender}} = \frac{3 * L/R}{RTT + L/R} = \frac{.024}{30.008} = 0.0008$$

Increase utilization by a factor of 3!

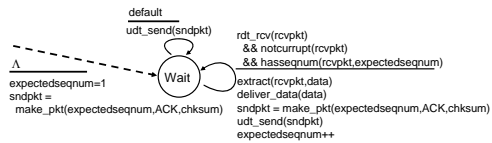
Transport Layer 3-45

GBN: sender extended FSM



Transport Layer 3-48

GBN: receiver extended FSM

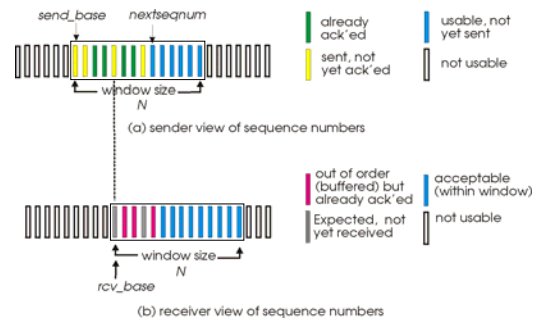


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **expectedseqnum**
- ❖ out-of-order pkt:
 - discard (don't buffer) → **no receiver buffering!**
 - Re-ACK pkt with highest in-order seq #

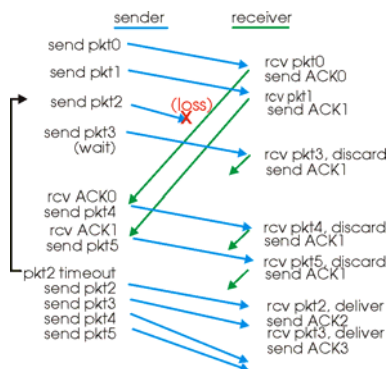
Transport Layer 3-49

Selective repeat: sender, receiver windows



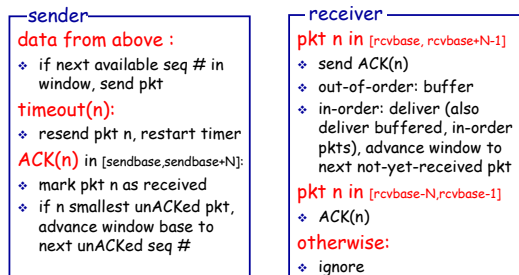
Transport Layer 3-52

GBN in action



Transport Layer 3-50

Selective repeat



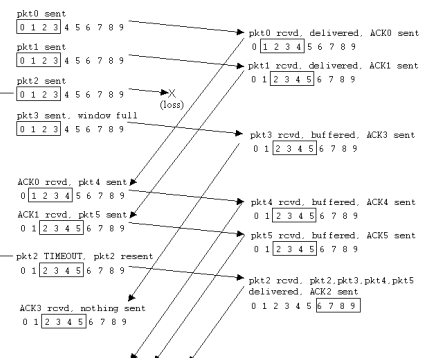
Transport Layer 3-53

Selective Repeat

- ❖ receiver *individually* acknowledges all correctly received pkts
 - buffers pkts, as needed, for eventual in-order delivery to upper layer
- ❖ sender only resends pkts for which ACK not received
 - sender timer for each unACKed pkt
- ❖ sender window
 - N consecutive seq #'s
 - again limits seq #'s of sent, unACK'ed pkts

Transport Layer 3-51

Selective repeat in action



port Layer 3-54

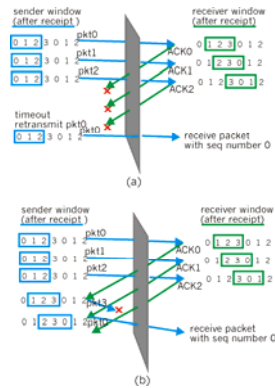
Selective repeat: dilemma

Example:

- seq #'s: 0, 1, 2, 3
- window size=3

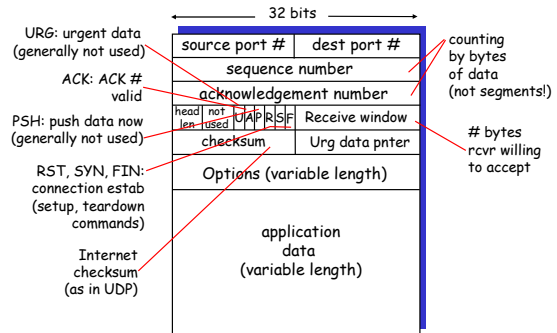
- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?



Transport Layer 3-55

TCP segment structure



Transport Layer 3-58

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

Transport Layer 3-56

TCP seq. #'s and ACKs

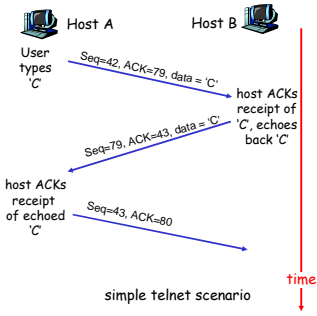
Seq. #'s:

- byte stream
- "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

- Q: how receiver handles out-of-order segments
- A: TCP spec doesn't say, - up to implementor



Transport Layer 3-59

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
 - one sender, one receiver
- reliable, in-order byte stream:
 - no "message boundaries"
- pipelined:
 - TCP congestion and flow control set window size
- send & receive buffers

- full duplex data:
 - bi-directional data flow in same connection
 - MSS: maximum segment size
- connection-oriented:
 - handshaking (exchange of control msgs) initiates sender, receiver state before data exchange
- flow controlled:
 - sender will not overwhelm receiver



Transport Layer 3-57

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
 - but RTT varies
- too short: premature timeout
 - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

- sampleRTT: measured time from segment transmission until ACK receipt
 - ignore retransmissions
- sampleRTT will vary, want estimated RTT "smoother"
 - average several recent measurements, not just current sampleRTT

Transport Layer 3-60

TCP Round Trip Time and Timeout

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

- ❖ Exponential weighted moving average
- ❖ influence of past sample decreases exponentially fast
- ❖ typical value: $\alpha = 0.125$

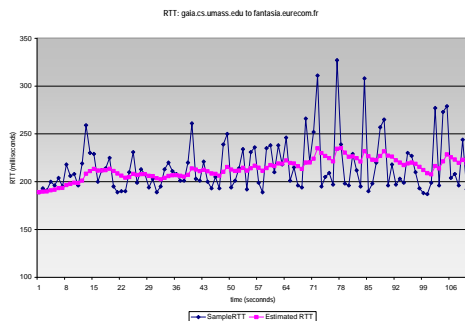
Transport Layer 3-61

Chapter 3 outline

- | | |
|--|--|
| 3.1 Transport-layer services | 3.5 Connection-oriented transport: TCP |
| 3.2 Multiplexing and demultiplexing | ▪ segment structure |
| 3.3 Connectionless transport: UDP | ▪ reliable data transfer |
| 3.4 Principles of reliable data transfer | ▪ flow control |
| | ▪ connection management |
| | 3.6 Principles of congestion control |
| | 3.7 TCP congestion control |

Transport Layer 3-64

Example RTT estimation:



Transport Layer 3-62

TCP reliable data transfer

- | | |
|---|---|
| ❖ TCP creates rdt service on top of IP's unreliable service | ❖ retransmissions are triggered by: |
| ❖ pipelined segments | ▪ timeout events |
| ❖ cumulative acks | ▪ duplicate acks |
| ❖ TCP uses single retransmission timer | ❖ initially consider simplified TCP sender: |
| | ▪ ignore duplicate acks |
| | ▪ ignore flow control, congestion control |

Transport Layer 3-65

TCP Round Trip Time and Timeout

Setting the timeout

- ❖ EstimatedRTT plus "safety margin"
 - large variation in EstimatedRTT → larger safety margin
- ❖ first estimate of how much SampleRTT deviates from EstimatedRTT:

$$\text{DevRTT} = (1 - \beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

Then set timeout interval:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$

Transport Layer 3-63

TCP sender events:

data rcvd from app:

- ❖ Create segment with seq #
- ❖ seq # is byte-stream number of first data byte in segment
- ❖ start timer if not already running (think of timer as for oldest unacked segment)
- ❖ expiration interval: TimeoutInterval

timeout:

- ❖ retransmit segment that caused timeout
- ❖ restart timer

Ack rcvd:

- ❖ If acknowledges previously unacked segments
 - update what is known to be acked
 - start timer if there are outstanding segments

Transport Layer 3-66

```

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

```

```

loop (forever) {
  switch(event)

```

```

  event: data received from application above
  create TCP segment with sequence number NextSeqNum
  if (timer currently not running)
    start timer
  pass segment to IP
  NextSeqNum = NextSeqNum + length(data)

```

```

  event: timer timeout
  retransmit not-yet-acknowledged segment with
  smallest sequence number
  start timer

```

```

  event: ACK received, with ACK field value of y
  if (y > SendBase) {
    SendBase = y
    if (there are currently not-yet-acknowledged segments)
      start timer
  }

```

```

} /* end of loop forever */

```

TCP sender (simplified)

Comment:

- SendBase-1: last cumulatively acked byte

Example:

- SendBase-1 = 71; y = 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

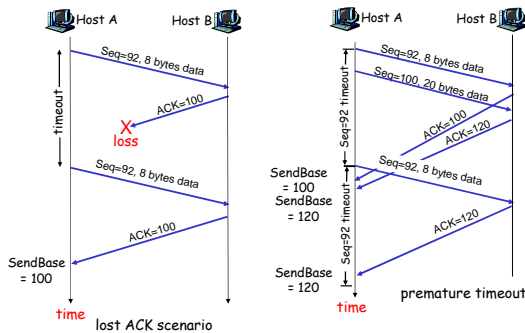
Transport Layer 3-67

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expected seq. #. Gap detected	Immediately send duplicate ACK , indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

Transport Layer 3-70

TCP: retransmission scenarios



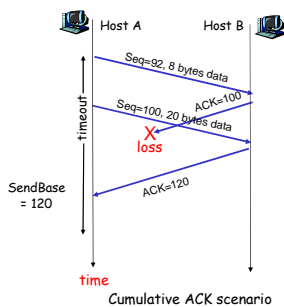
Transport Layer 3-68

Fast Retransmit

- ❖ time-out period often relatively long:
 - long delay before resending lost packet
- ❖ detect lost segments via duplicate ACKs.
 - sender often sends many segments back-to-back
 - if segment is lost, there will likely be many duplicate ACKs.
- ❖ if sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
 - **fast retransmit:** resend segment before timer expires

Transport Layer 3-71

TCP retransmission scenarios (more)



Transport Layer 3-69

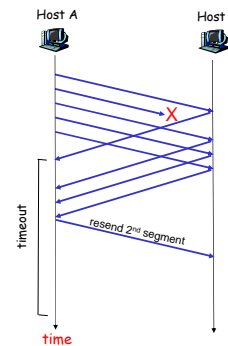


Figure 3.37 Resending a segment after triple duplicate ACK

Transport Layer 3-72

Fast retransmit algorithm:

```

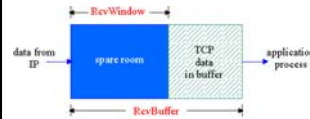
event: ACK received, with ACK field value of y
if (y > SendBase) {
    SendBase = y
    if (there are currently not-yet-acknowledged segments)
        start timer
}
else {
    increment count of dup ACKs received for y
    if (count of dup ACKs received for y = 3) {
        resend segment with sequence number y
    }
}
    
```

a duplicate ACK for already ACKed segment

fast retransmit

Transport Layer 3-73

TCP Flow control: how it works



(suppose TCP receiver discards out-of-order segments)

❖ spare room in buffer

= RcvWindow

= RcvBuffer - [LastByteRcvd - LastByteRead]

❖ rcvr advertises spare room by including value of RcvWindow in segments

❖ sender limits unACKed data to RcvWindow

- guarantees receive buffer doesn't overflow

Transport Layer 3-76

Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

❖ 3.5 Connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

Transport Layer 3-74

Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

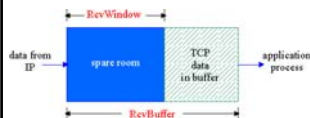
3.6 Principles of congestion control

3.7 TCP congestion control

Transport Layer 3-77

TCP Flow Control

❖ receive side of TCP connection has a receive buffer:



❖ app process may be slow at reading from buffer

flow control
sender won't overflow receiver's buffer by transmitting too much, too fast

❖ speed-matching service: matching the send rate to the receiving app's drain rate

Transport Layer 3-75

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

❖ initialize TCP variables:

- seq. #s
- buffers, flow control info (e.g. RcvWindow)

❖ **client:** connection initiator

```
Socket clientSocket = new Socket("hostname", "port number");
```

❖ **server:** contacted by client

```
Socket connectionSocket = welcomeSocket.accept();
```

Three way handshake:

Step 1: client host sends TCP SYN segment to server

- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment

- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

Transport Layer 3-78

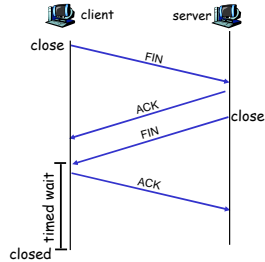
TCP Connection Management (cont.)

Closing a connection:

client closes socket:
`clientSocket.close();`

Step 1: client end system
sends TCP FIN control
segment to server

Step 2: server receives
FIN, replies with ACK.
Closes connection, sends
FIN.



Transport Layer 3-79

Chapter 3 outline

3.1 Transport-layer
services

3.2 Multiplexing and
demultiplexing

3.3 Connectionless
transport: UDP

3.4 Principles of reliable
data transfer

3.5 Connection-oriented
transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of
congestion control

3.7 TCP congestion control

Transport Layer 3-82

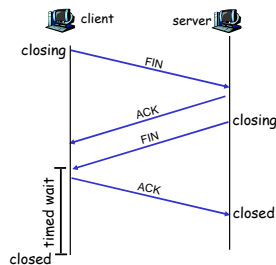
TCP Connection Management (cont.)

Step 3: client receives FIN,
replies with ACK.

- Enters "timed wait" -
will respond with ACK
to received FINs

Step 4: server, receives
ACK. Connection closed.

Note: with small
modification, can handle
simultaneous FINs.



Transport Layer 3-80

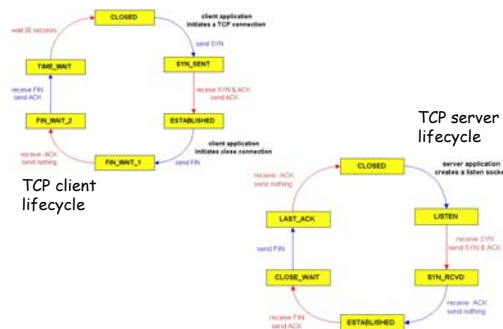
Principles of Congestion Control

Congestion:

- informally: "too many sources sending too much
data too fast for *network* to handle"
- different from flow control!
- manifestations:
 - lost packets (buffer overflow at routers)
 - long delays (queueing in router buffers)
- a top-10 problem!

Transport Layer 3-83

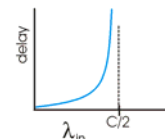
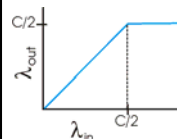
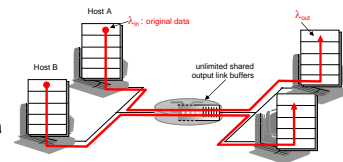
TCP Connection Management (cont.)



Transport Layer 3-81

Causes/costs of congestion: scenario 1

- two senders, two
receivers
- one router,
infinite buffers
- no retransmission

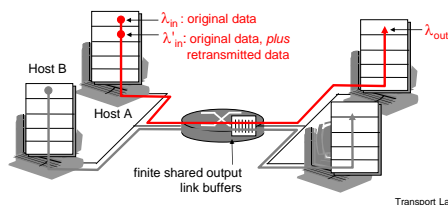


- large delays
when congested
- maximum
achievable
throughput

Transport Layer 3-84

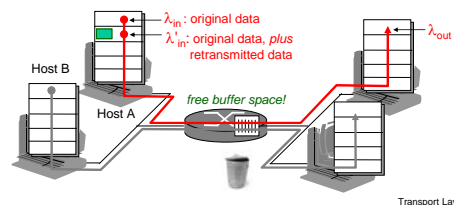
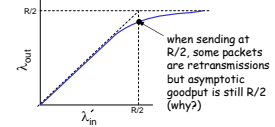
Causes/costs of congestion: scenario 2

- ❖ one router, *finite* buffers
- ❖ sender retransmission of timed-out packet
 - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
 - transport-layer input includes *retransmissions*: $\lambda'_{in} \geq \lambda_{in}$



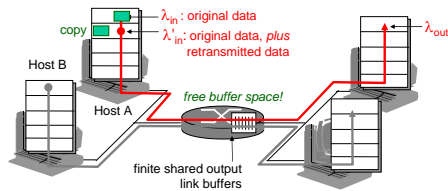
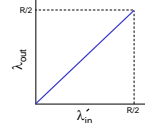
Congestion scenario 2b: *known loss*

- ❖ packets may get dropped at router due to full buffers
 - sometimes not lost
- ❖ sender only resends if packet *known* to be lost (admittedly idealized)



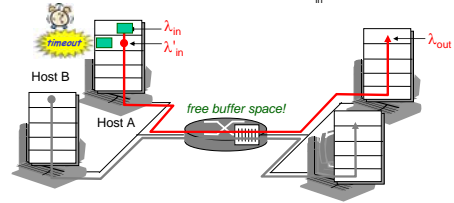
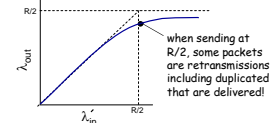
Congestion scenario 2a: *ideal case*

- ❖ sender sends only when router buffers available



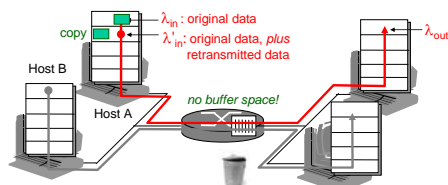
Congestion scenario 2c: *duplicates*

- ❖ packets may get dropped at router due to full buffers
- ❖ sender times out prematurely, sending *two* copies, both of which are delivered



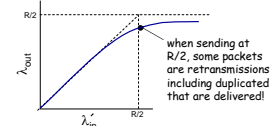
Congestion scenario 2b: *known loss*

- ❖ packets may get dropped at router due to full buffers
 - sometimes lost
- ❖ sender only resends if packet *known* to be lost (admittedly idealized)



Congestion scenario 2c: *duplicates*

- ❖ packets may get dropped at router due to full buffers
- ❖ sender times out prematurely, sending *two* copies, both of which are delivered



"costs" of congestion:

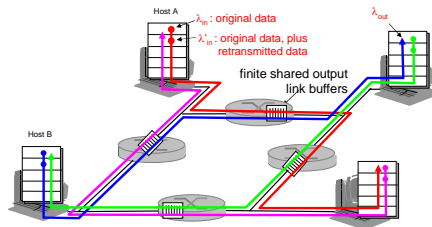
- ❖ more work (retrans) for given "goodput"
- ❖ *unnecessary* retransmissions: link carries multiple copies of pkt
 - decreasing goodput

Transport Layer 3-90

Causes/costs of congestion: scenario 3

- ❖ four senders
- ❖ multihop paths
- ❖ timeout/retransmit

Q: what happens as λ_{in} and λ'_{in} increase?



Transport Layer 3-91

Case study: ATM ABR congestion control

ABR: available bit rate:

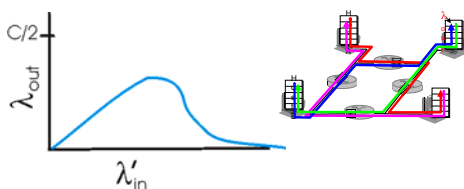
- ❖ "elastic service"
- ❖ if sender's path "underloaded":
 - sender should use available bandwidth
- ❖ if sender's path congested:
 - sender throttled to minimum guaranteed rate

RM (resource management) cells:

- ❖ sent by sender, interspersed with data cells
- ❖ bits in RM cell set by switches ("network-assisted")
 - NI bit: no increase in rate (mild congestion)
 - CI bit: congestion indication
- ❖ RM cells returned to sender by receiver, with bits intact

Transport Layer 3-94

Causes/costs of congestion: scenario 3

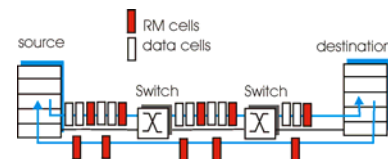


another "cost" of congestion:

- ❖ when packet dropped, any "upstream transmission capacity used for that packet was wasted!"

Transport Layer 3-92

Case study: ATM ABR congestion control



- ❖ two-byte ER (explicit rate) field in RM cell
 - congested switch may lower ER value in cell
 - sender's send rate thus maximum supportable rate on path
- ❖ EFCI bit in data cells: set to 1 in congested switch
 - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

Transport Layer 3-95

Approaches towards congestion control

Two broad approaches towards congestion control:

end-end congestion control:

- ❖ no explicit feedback from network
- ❖ congestion inferred from end-system observed loss, delay
- ❖ approach taken by TCP

network-assisted congestion control:

- ❖ routers provide feedback to end systems
 - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
 - explicit rate sender should send at

Transport Layer 3-93

Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

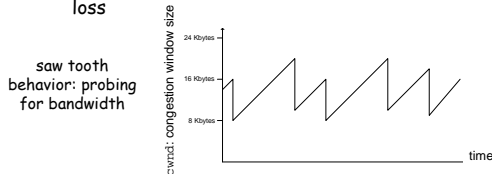
3.6 Principles of congestion control

3.7 TCP congestion control

Transport Layer 3-96

TCP congestion control: additive increase, multiplicative decrease

- ❖ **approach**: increase transmission rate (window size), probing for usable bandwidth, until loss occurs
 - **additive increase**: increase `cwnd` by 1 MSS every RTT until loss detected
 - **multiplicative decrease**: cut `cwnd` in half after loss



Transport Layer 3-97

Refinement: inferring loss

- ❖ after 3 dup ACKs:
 - `cwnd` is cut in half
 - window then grows linearly
- ❖ **but** after timeout event:
 - `cwnd` instead set to 1 MSS;
 - window then grows exponentially
 - to a threshold, then grows linearly

Philosophy:

- ❖ 3 dup ACKs indicates network capable of delivering some segments
- ❖ timeout indicates a "more alarming" congestion scenario

Transport Layer 3-100

TCP Congestion Control: details

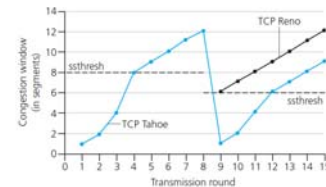
- ❖ sender limits transmission: $\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$
 - ❖ roughly,

$$\text{rate} = \frac{\text{cwnd}}{\text{RTT}} \text{ Bytes/sec}$$
 - ❖ `cwnd` is dynamic, function of perceived network congestion
- How does sender perceive congestion?**
- ❖ loss event = timeout or 3 duplicate acks
 - ❖ TCP sender reduces rate (`cwnd`) after loss event
- three mechanisms:**
- AIMD
 - slow start
 - conservative after timeout events

Transport Layer 3-98

Refinement

- Q:** when should the exponential increase switch to linear?
- A:** when `cwnd` gets to 1/2 of its value before timeout.



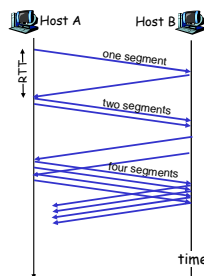
Implementation:

- ❖ variable `ssthresh`
- ❖ on loss event, `ssthresh` is set to 1/2 of `cwnd` just before loss event

Transport Layer 3-101

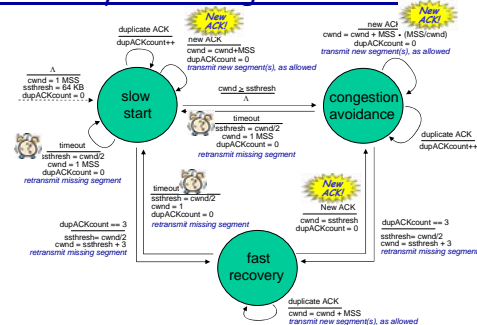
TCP Slow Start

- ❖ when connection begins, increase rate exponentially until first loss event:
 - initially `cwnd` = 1 MSS
 - double `cwnd` every RTT
 - done by incrementing `cwnd` for every ACK received
- ❖ **summary**: initial rate is slow but ramps up exponentially fast



Transport Layer 3-99

Summary: TCP Congestion Control



Transport Layer 3-102

TCP throughput

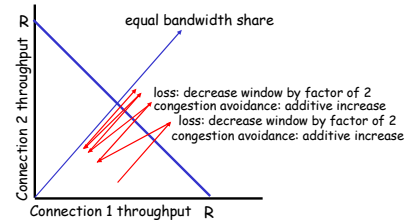
- ❖ what's the average throughput of TCP as a function of window size and RTT?
 - ignore slow start
- ❖ let W be the window size when loss occurs.
 - when window is W , throughput is W/RTT
 - just after loss, window drops to $W/2$, throughput to $W/2RTT$.
 - average throughput: $.75 W/RTT$

Transport Layer 3-103

Why is TCP fair?

two competing sessions:

- ❖ additive increase gives slope of 1, as throughput increases
- ❖ multiplicative decrease decreases throughput proportionally



Transport Layer 3-106

TCP Futures: TCP over "long, fat pipes"

- ❖ example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- ❖ requires window size $W = 83,333$ in-flight segments
- ❖ throughput in terms of loss rate:

$$\frac{1.22 \cdot MSS}{RTT \sqrt{L}}$$
- ❖ $L = 2 \cdot 10^{-10}$ *Wow - a very small loss rate!*
- ❖ new versions of TCP for high-speed

Transport Layer 3-104

Fairness (more)

Fairness and UDP

- ❖ multimedia apps often do not use TCP
 - do not want rate throttled by congestion control
- ❖ instead use UDP:
 - pump audio/video at constant rate, tolerate packet loss

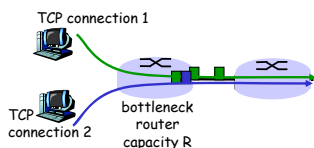
Fairness and parallel TCP connections

- ❖ nothing prevents app from opening parallel connections between 2 hosts.
- ❖ web browsers do this
 - example: link of rate R supporting 9 connections;
 - new app asks for 1 TCP, gets rate $R/10$
 - new app asks for 11 TCPs, gets $R/2$!

Transport Layer 3-107

TCP Fairness

fairness goal: if K TCP sessions share same bottleneck link of bandwidth R , each should have average rate of R/K



Transport Layer 3-105

Chapter 3: Summary

- ❖ principles behind transport layer services:
 - multiplexing, demultiplexing
 - reliable data transfer
 - flow control
 - congestion control
- ❖ instantiation and implementation in the Internet
 - UDP
 - TCP

Next:

- ❖ leaving the network "edge" (application, transport layers)
- ❖ into the network "core"

Transport Layer 3-108