

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb}/\text{pkt}}{10^{**9} \text{ b/sec}} = 8 \text{ microsec}$$

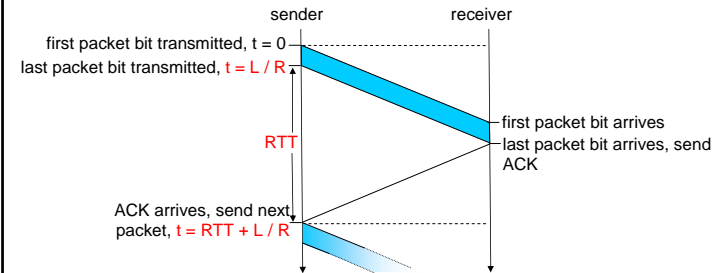
- U_{sender} : **utilization** - fraction of time sender busy sending

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

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

Transport Layer 3-1

rdt3.0: stop-and-wait operation



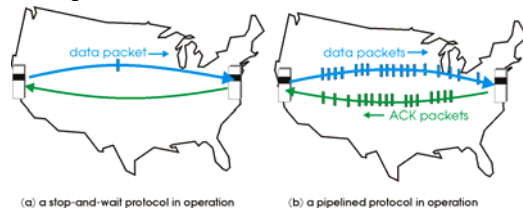
$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Transport Layer 3-2

Pipelined protocols

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

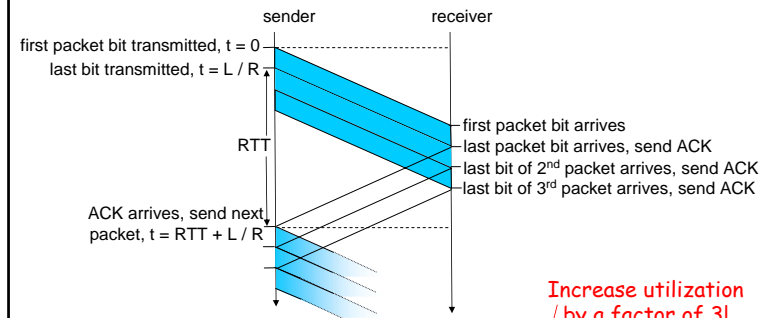
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

Transport Layer 3-3

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 3-4

Go-Back-N

Sender:

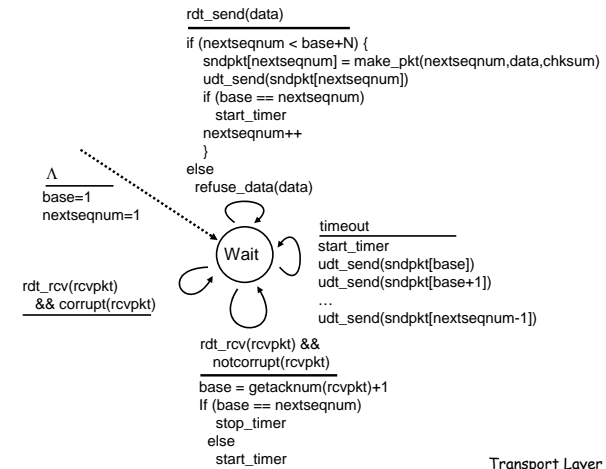
- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed



- 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

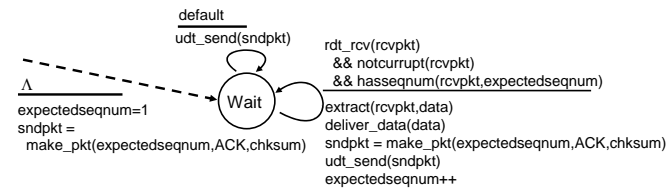
Transport Layer 3-5

GBN: sender extended FSM



Transport Layer 3-6

GBN: receiver extended FSM

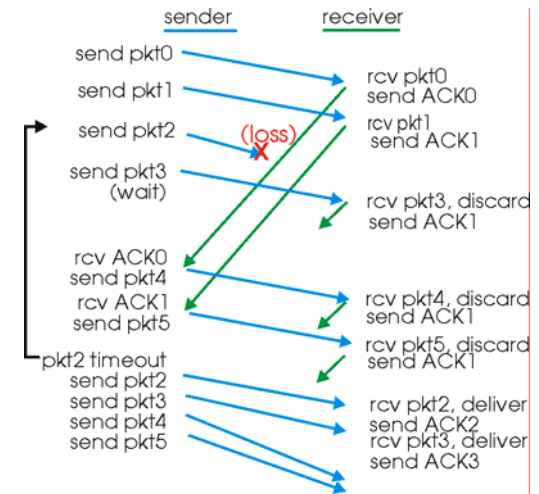


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- 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 3-7

GBN in action



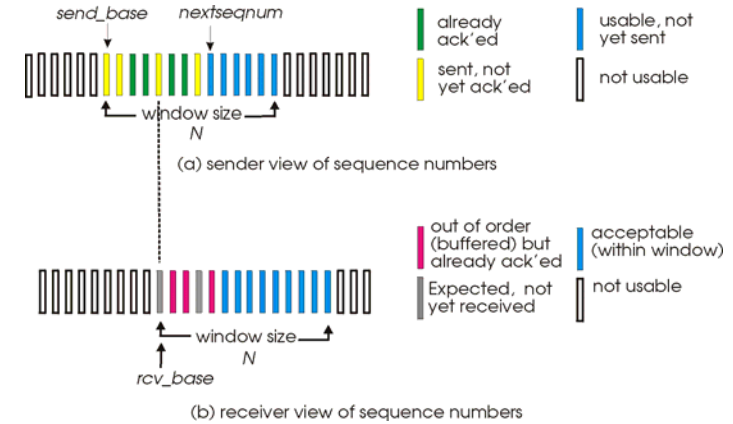
Transport Layer 3-8

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, unACKed pkts

Transport Layer 3-9

Selective repeat: sender, receiver windows



Transport Layer 3-10

Selective repeat

sender

data from above :

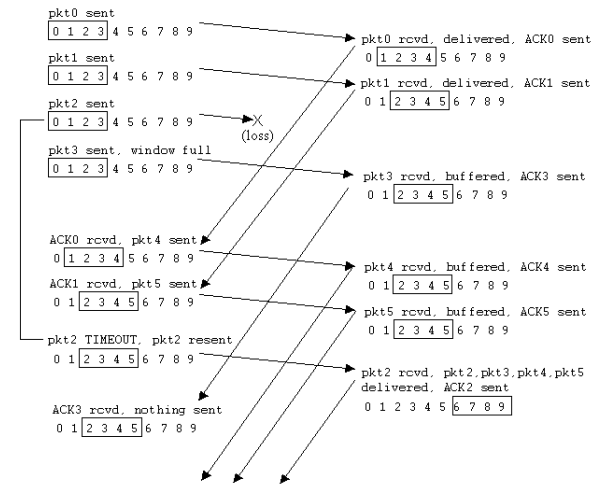
- if next available seq # in window, send pkt
- timeout(n):**
 - resend pkt n, restart timer
- ACK(n)** in [sendbase, sendbase+N]:
 - mark pkt n as received
 - if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

- pkt n** in [rcvbase, rcvbase+N-1]
 - send ACK(n)
 - out-of-order: buffer
 - in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
- pkt n** in [rcvbase-N, rcvbase-1]
 - ACK(n)
- otherwise:**
 - ignore

Transport Layer 3-11

Selective repeat in action



* Layer 3-12

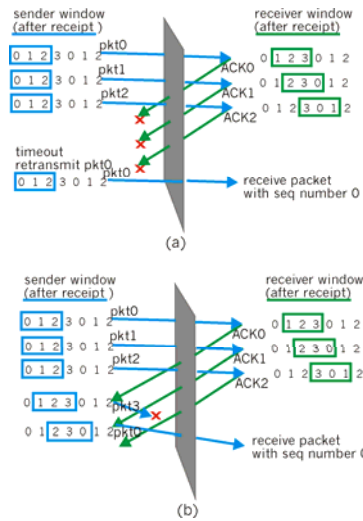
Selective repeat: dilemma

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?



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 3-14

TCP: Overview

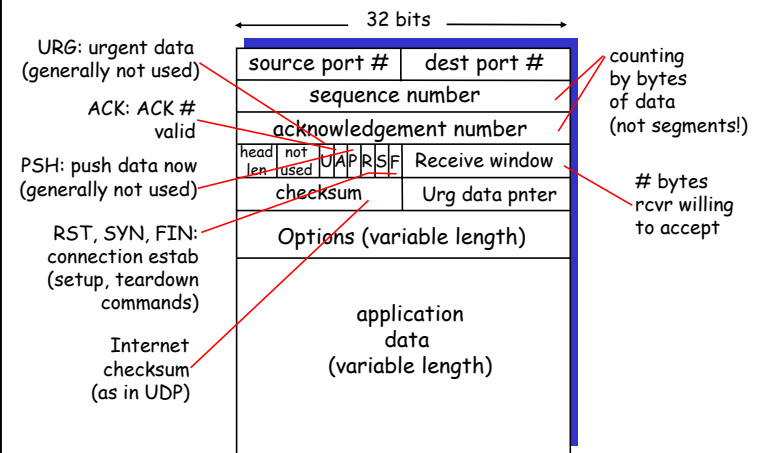
RFCs: 793, 1122, 1323, 2018, 2581

- 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) init's sender, receiver state before data exchange
- flow controlled:**
 - sender will not overwhelm receiver



Transport Layer 3-15

TCP segment structure



Transport Layer 3-16

TCP seq. #'s and ACKs

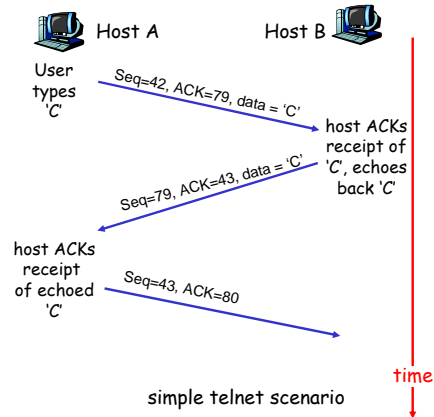
Seq. #'s:

- 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 3-17

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 3-18

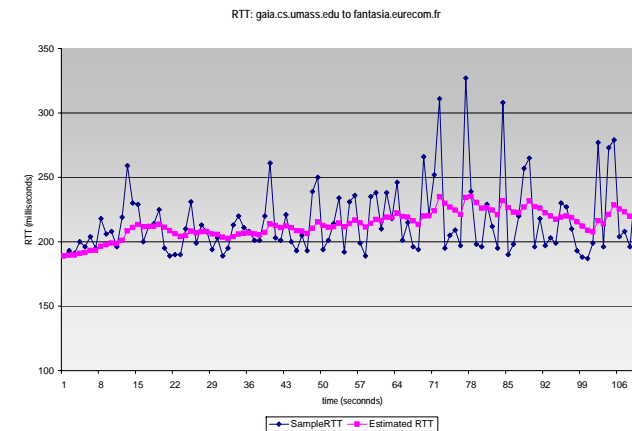
TCP Round Trip Time and Timeout

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

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

Transport Layer 3-19

Example RTT estimation:



Transport Layer 3-20

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 3-21

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 3-22

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
 - timeout events
 - duplicate acks
- Initially consider simplified TCP sender:
 - ignore duplicate acks
 - ignore flow control, congestion control

Transport Layer 3-23

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 3-24

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 3-29

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 3-30

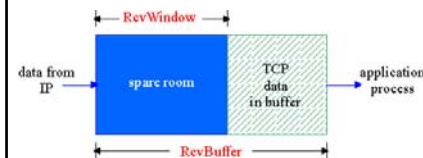
Chapter 3 outline

- 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 3-31

TCP Flow Control

- receive side of TCP connection has a receive buffer:



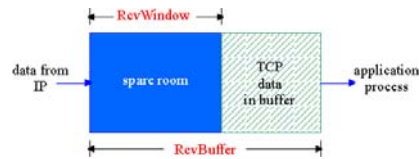
- app process may be slow at reading from buffer

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 3-32

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 3-33

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 3-34

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

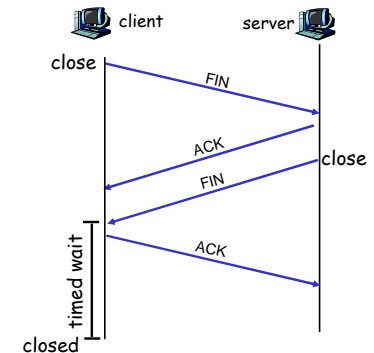
Transport Layer 3-35

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 3-36

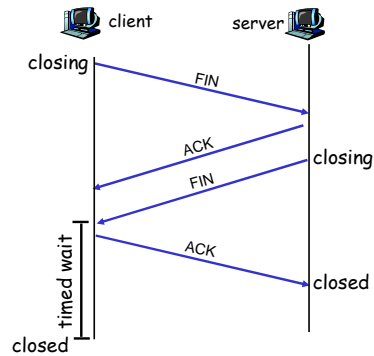
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

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

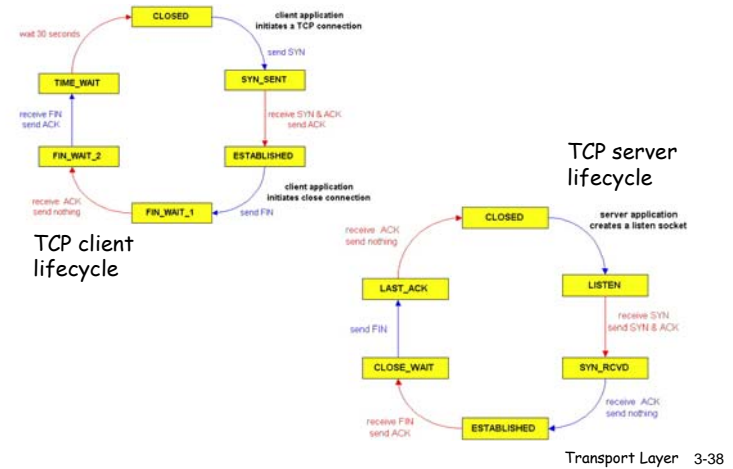
Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



Transport Layer 3-37

TCP Connection Management (cont)



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 3-39

Principles of Congestion Control

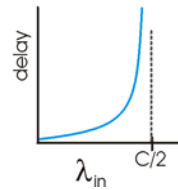
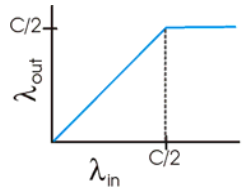
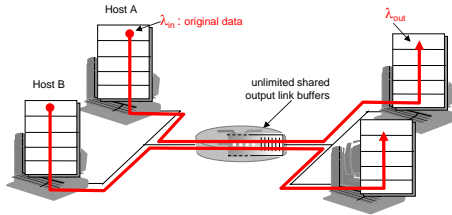
Congestion:

- 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 3-40

Causes/costs of congestion: scenario 1

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

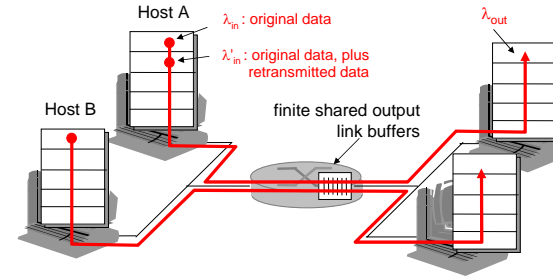


- large delays when congested
- maximum achievable throughput

Transport Layer 3-41

Causes/costs of congestion: scenario 2

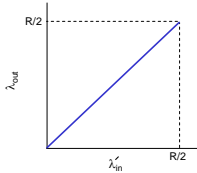
- one router, *finite* buffers
- sender retransmission of lost packet



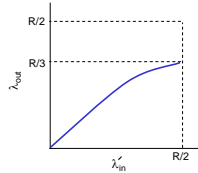
Transport Layer 3-42

Causes/costs of congestion: scenario 2

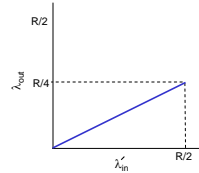
- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$, λ'_{in} larger
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



a.



b.



c.

"costs" of congestion:

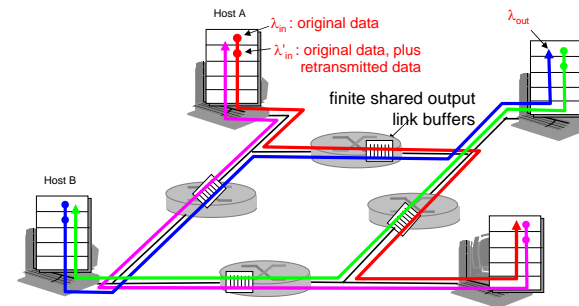
- more work (retrans) for given "goodput"
- unnecessary retransmissions: link carries multiple copies of pkt

Transport Layer 3-43

Causes/costs of congestion: scenario 3

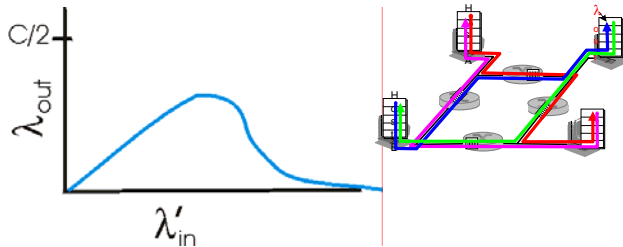
- four senders
- multihop paths
- timeout/retransmit

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



Transport Layer 3-44

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 3-45

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 3-46

Case study: ATM ABR congestion control

ABR: available bit rate:

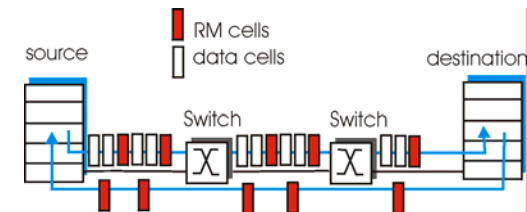
- "elastic service"
- if sender's path "underloaded":
 - sender should use available bandwidth
- if sender's path congested:
 - sender throttled to minimum guaranteed rate

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 3-47

Case study: ATM ABR congestion control



- two-byte ER (explicit rate) field in RM cell
 - congested switch may lower ER value in cell
 - sender' 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 3-48

Chapter 3 outline

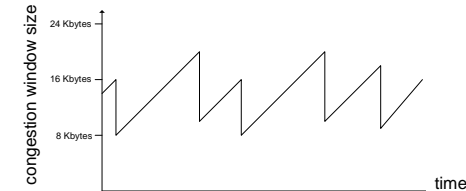
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 Principles of congestion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

Transport Layer 3-49

TCP congestion control: additive increase, multiplicative decrease

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

Saw tooth behavior: probing for bandwidth



Transport Layer 3-50

TCP Congestion Control: details

- sender limits transmission: $\text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin}$
 - Roughly,

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

Transport Layer 3-51

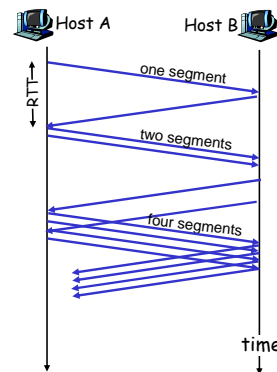
TCP Slow Start

- When connection begins, CongWin = 1 MSS
 - Example: MSS = 500 bytes & RTT = 200 msec
 - initial rate = 20 kbps
- available bandwidth may be $\gg \text{MSS}/\text{RTT}$
 - desirable to quickly ramp up to respectable rate
- When connection begins, increase rate exponentially fast until first loss event

Transport Layer 3-52

TCP Slow Start (more)

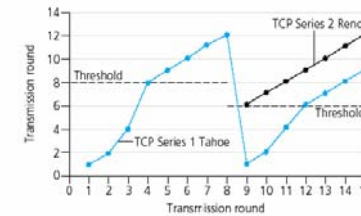
- When connection begins, increase rate exponentially until first loss event:
 - double CongWin every RTT
 - done by incrementing CongWin for every ACK received
- **Summary:** initial rate is slow but ramps up exponentially fast



Transport Layer 3-53

Refinement

- Q: When should the exponential increase switch to linear?
- A: When CongWin gets to 1/2 of its value before timeout.



Implementation:

- Variable Threshold
- At loss event, Threshold is set to 1/2 of CongWin just before loss event

Transport Layer 3-54

Refinement: inferring loss

- After 3 dup ACKs:
 - CongWin is cut in half
 - window then grows linearly
- **But** after timeout event:
 - CongWin 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 3-55

Summary: TCP Congestion Control

- When CongWin is below Threshold, sender in **slow-start** phase, window grows exponentially.
- When CongWin is above Threshold, sender is in **congestion-avoidance** phase, window grows linearly.
- When a **triple duplicate ACK** occurs, Threshold set to CongWin/2 and CongWin set to Threshold.
- When **timeout** occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS.

Transport Layer 3-56

TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin + MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

Transport Layer 3-57

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 3-58

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:

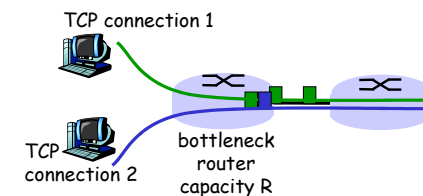
$$\frac{1.22 \cdot MSS}{RTT \sqrt{L}}$$

- $\rightarrow L = 2 \cdot 10^{10}$ *Wow*
- New versions of TCP for high-speed

Transport Layer 3-59

TCP Fairness

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

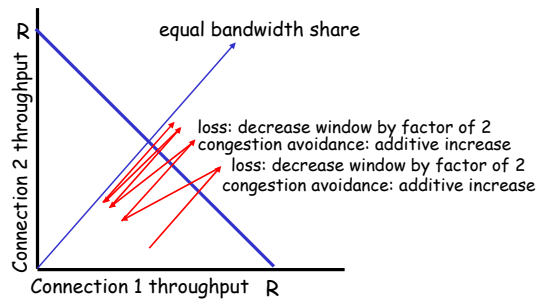


Transport Layer 3-60

Why is TCP fair?

Two competing sessions:

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



Transport Layer 3-61

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
- Research area: TCP friendly

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 3-62

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 3-63