

Transport Layer

Michalis Faloutsos
Many slides from Kurose-Ross

Transport Layer Functionality

- ✴ Hide network from application layer
- ✴ Transport layer resides at end points
- ✴ Sees the network as a black box



Transport Layers of the Internet

- ✴ **TCP: reliable protocol**
 - Guarantees end-to-end delivery
 - Self-controls rate: congestion and flow control
 - Connection oriented: handshake, state
 - Ordered delivery of packets to application
- ✴ **UDP: unreliable protocol**
 - Non-regulated sending rate
 - Multiplexing-demultiplexing

Excerpts From Quiz

- ✴ TCP drops packets when there is congestion
- ✴ TCP provides QoS
- ✴ **UDP is better for video streaming, because even if packets are lost, it is still ok.**

TCP overview

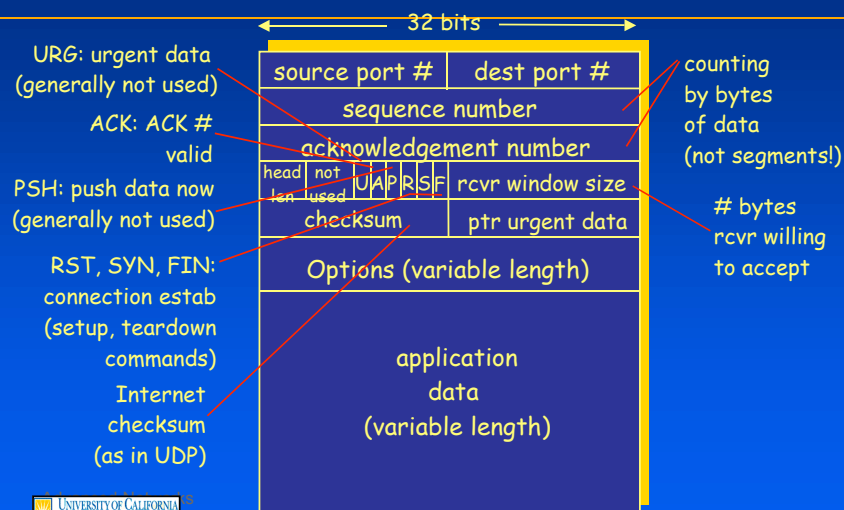
TCP: What and How

For more: 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



TCP segment structure



TCP seq. #'s and ACKs

Seq. #'s:

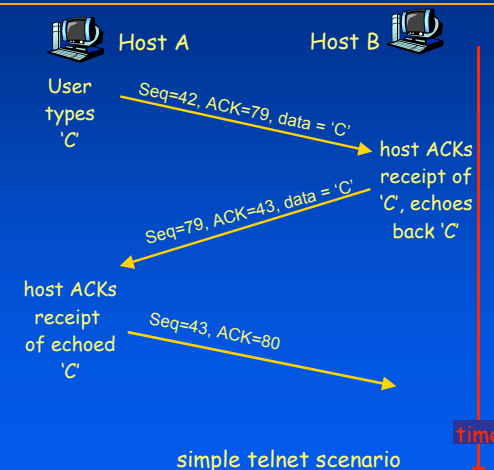
- byte stream "number" of first byte in segment's data

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



simple telnet scenario

TCP in a nutshell

☀ Slow start (actually this is fast increase)

- Increase by one 1 max size segment
- Do this up to a threshold: ssthresh

☀ Congestion control

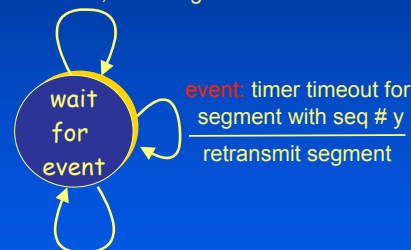
- Increase by 1 max size segment every RTT
- Drop window in half, if there is congestion
 - Packet loss : duplicate ACKs
 - Time expiration

TCP: reliable data transfer

event: data received
from application above
create, send segment

simplified sender, assuming

- one way data transfer
- no flow, congestion control



event: ACK received,
with ACK # y
ACK processing

TCP: reliable data transfer

Simplified
TCP
sender

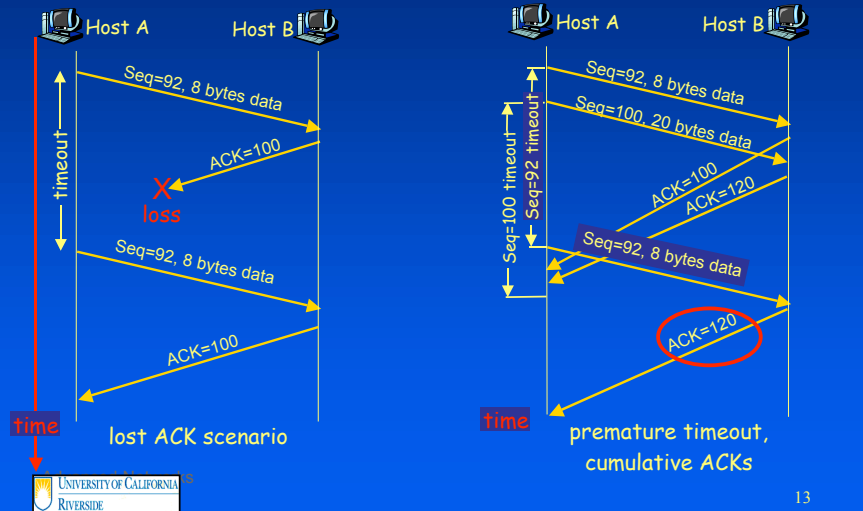
```

00 sendbase = initial_sequence number
01 nextseqnum = initial_sequence number
02
03 loop (forever) {
04     switch(event)
05     event: data received from application above
06         create TCP segment with sequence number nextseqnum
07         start timer for segment nextseqnum
08         pass segment to IP
09         nextseqnum = nextseqnum + length(data)
10     event: timer timeout for segment with sequence number y
11         retransmit segment with sequence number y
12         compute new timeout interval for segment y
13         restart timer for sequence number y
14     event: ACK received, with ACK field value of y
15         if (y > sendbase) { /* cumulative ACK of all data up to y */
16             cancel all timers for segments with sequence numbers < y
17             sendbase = y
18         }
19         else { /* a duplicate ACK for already ACKed segment */
20             increment number of duplicate ACKs received for y
21             if (number of duplicate ACKs received for y == 3) {
22                 /* TCP fast retransmit */
23                 resend segment with sequence number y
24                 restart timer for segment y
25             }
26     } /* end of loop forever */
    
```

TCP ACK generation [RFC 1122, RFC 2581]

Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expect seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

TCP: retransmission scenarios



13

TCP Flow Control

flow control

sender won't overrun receiver's buffers by transmitting too much, too fast

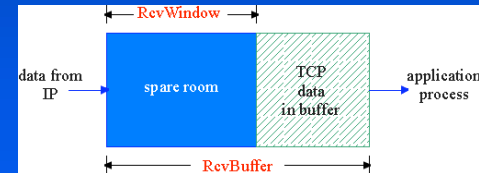
receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

- **RecvWindow** field in TCP segment

sender: keeps the amount of transmitted, unACKed data less than most recently received RecvWindow

RecvBuffer = size of TCP Receive Buffer

RecvWindow = amount of spare room in Buffer



receiver buffering

14

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
 - note: RTT will vary
- 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, cumulatively ACKed segments
- SampleRTT will vary, want estimated RTT "smoother"
 - use several recent measurements, not just current SampleRTT

15

TCP Round Trip Time and Timeout

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

Exponential weighted moving average
influence of given sample decreases exponentially fast
typical value of x: 0.1

Setting the timeout

- EstimatedRTT plus "safety margin"
- large variation in EstimatedRTT -> larger safety margin

$$\text{Timeout} = \text{EstimatedRTT} + 4 * \text{Deviation}$$

$$\text{Deviation} = (1-x) * \text{Deviation} + x * |\text{SampleRTT} - \text{EstimatedRTT}|$$

16

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 end system sends TCP SYN control segment to server

- specifies initial seq #

Step 2: server end system receives SYN, replies with SYNACK control segment

- ACKs received SYN
- allocates buffers
- specifies server-> receiver initial seq. #

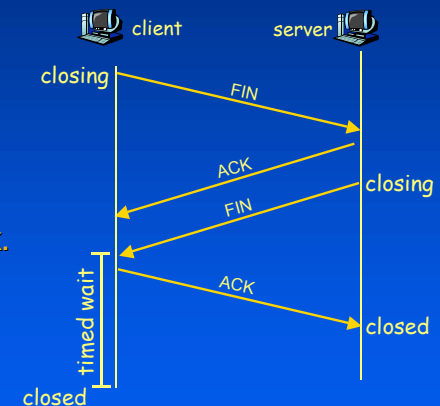
Step 3: Client replies with an ACK (using servers seq number)

TCP Connection Management (cont.)

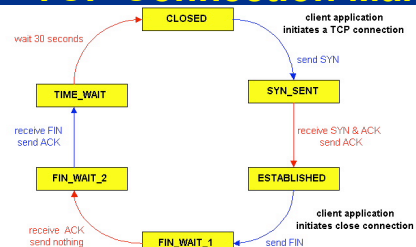
Step 3: client receives FIN, replies with ACK.

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

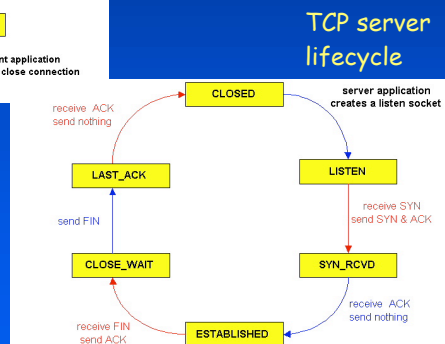
Step 4: server, receives ACK. Connection closed.



TCP Connection Management (cont)



TCP client lifecycle



TCP server lifecycle

Principles of Congestion Control

Congestion:

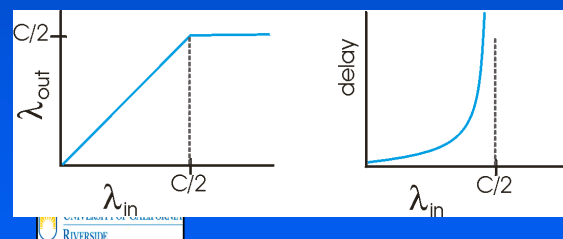
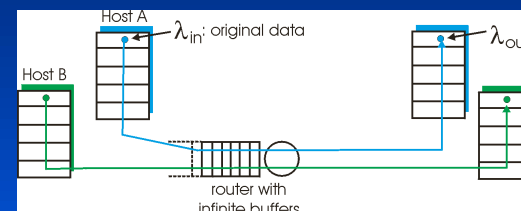
- ☀ informally: "too many sources sending too much data too fast for network to handle"
- ☀ different from flow control!
- ☀ manifestations:
 - lost packets (buffer overflow at routers)
 - long delays (queueing in router buffers)
- ☀ Major research issue

Consequences of Congestion

- ☀ **Large delays: throughput vs delay trade-off**
 - We don't want to operate near capacity
- ☀ **Finite buffers: lost packets**
- ☀ **Resending of packets causes**
 - More packets for the same goodput
 - Wasted bandwidth of the packet that gets dropped

Causes/costs of congestion: scenario 1

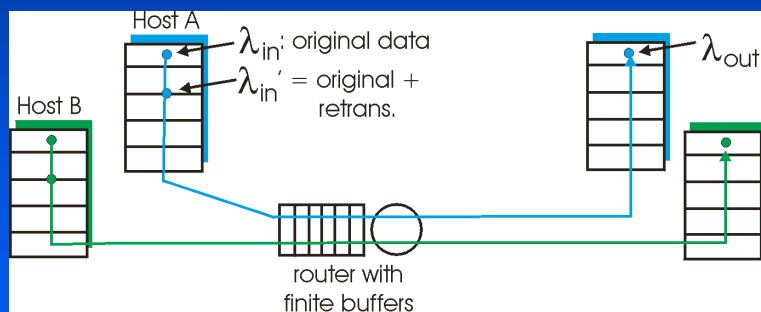
- ☀ **two senders, two receivers**
- ☀ **one router, infinite buffers**
- ☀ **no retransmission**



- ☀ **large delays when congested**
- ☀ **maximum achievable throughput**

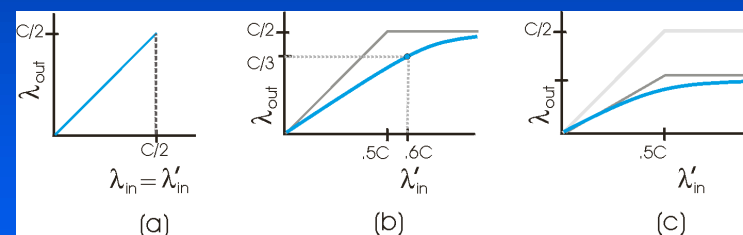
Causes/costs of congestion: scenario 2

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



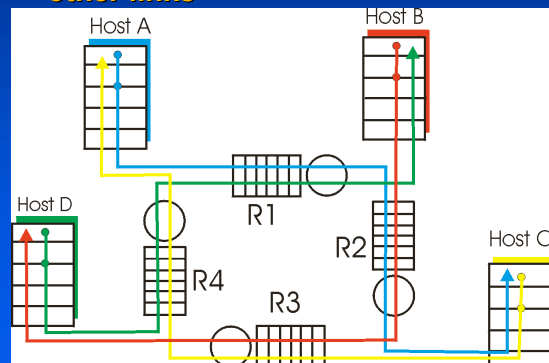
Causes/costs of congestion: scenario 2

- ☀ **Always: $\lambda_{in} = \lambda_{out}$ (goodput)**
- ☀ **If packets are dropped: $\lambda'_{in} > \lambda_{out}$**

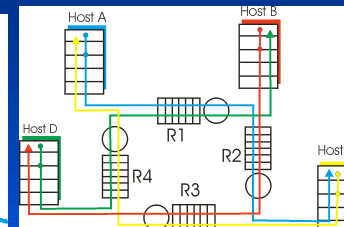
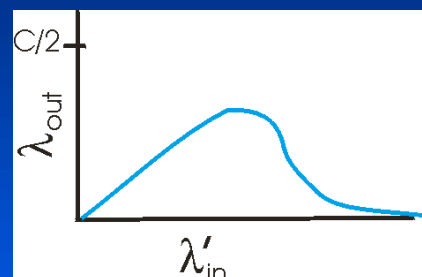


Causes/costs of congestion: scenario 3

- Four senders, multihop paths, timeout/retransmit
- Congestion in one link \rightarrow retransmits \rightarrow congestion in other links



Causes/costs of congestion: scenario 3



Another “cost” of congestion:
when packet dropped, any “upstream transmission capacity
used for that packet was wasted!

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

Case study: ATM ABR congestion control

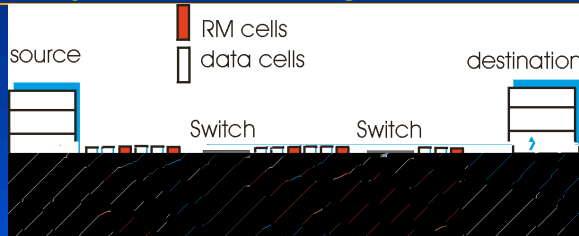
ABR: available bit rate:

- “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

Case study: ATM ABR congestion control



two-byte ER (explicit rate) field in RM cell

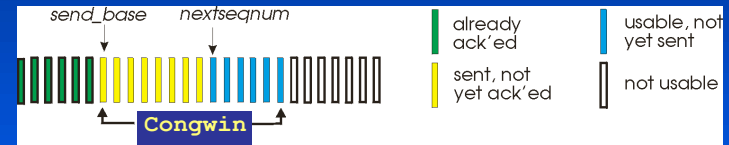
- congested switch may lower ER value in cell
- sender's send rate thus minimum supportable rate on path

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

TCP Congestion Control

- end-end control (no network assistance)
- transmission rate limited by congestion window size, Congwin, over segments:



w segments, each with MSS bytes sent in one RTT:

$$\text{throughput} = \frac{w * \text{MSS}}{\text{RTT}} \text{ Bytes/sec}$$

TCP congestion control:

"probing" for usable bandwidth:

- ideally: transmit as fast as possible (Congwin as large as possible) without loss
- increase Congwin until loss (congestion)
- loss: decrease Congwin, then begin probing (increasing) again

two "phases"

- slow start
- congestion avoidance

important variables:

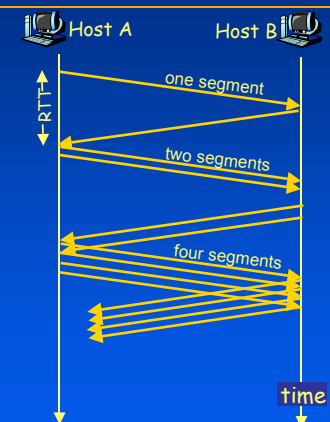
- Congwin
- threshold: defines threshold between two slow start phase, congestion control phase

TCP Slowstart

Slowstart algorithm

initialize: Congwin = 1
for (each segment ACKed)
Congwin++
until (loss event OR
CongWin > threshold)

- exponential increase (per RTT) in window size
- loss event: timeout (Tahoe TCP) and/or or three duplicate ACKs (Reno TCP)

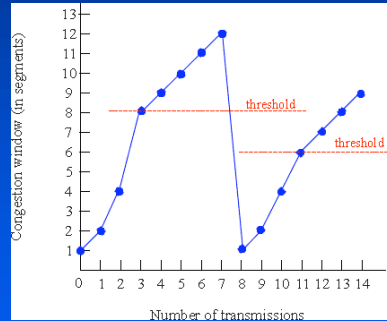


TCP Congestion Avoidance

Congestion avoidance

```

/* slowstart is over */
/* Congwin > threshold */
Until (loss event) {
  every w segments ACKed:
    Congwin++
}
threshold = Congwin/2
Congwin = 1
perform slowstart1
    
```



1: TCP Reno skips slowstart (fast recovery) after three duplicate ACKs

TCP Congestion: Real Life is Hairy!

Congestion avoidance

```

/* slowstart is over */
/* Congwin > threshold */
Until (loss event) {
  every w segments ACKed:
    Congwin++
}
threshold = Congwin/2
Congwin = 1
perform slowstart1
    
```

Remember: bytes vs packets!

$CW += MSS * MSS/CW$

$Thres = \text{Max}(2 * MSS, \text{InFlightData}/2)$

MSS: max segment size

InFlightData: un-ACK-ed data

RFC 2581: TCP Congestion Control

AIMD

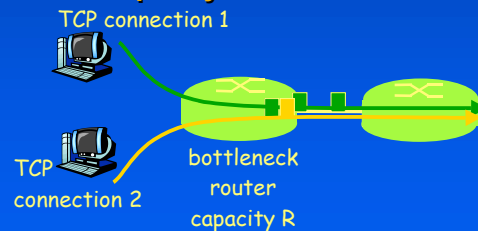
TCP congestion avoidance:

AIMD: additive increase, multiplicative decrease

- increase window by 1 per RTT
- decrease window by factor of 2 on loss event

TCP Fairness

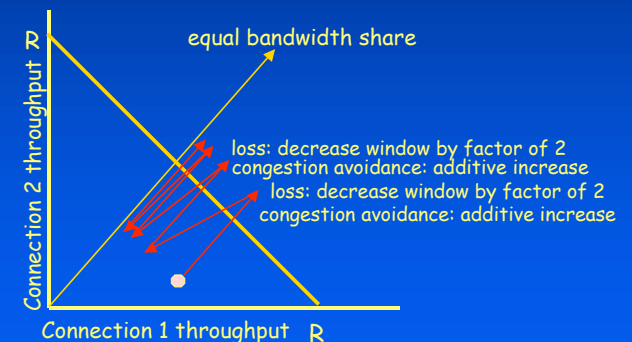
Fairness goal: if N TCP sessions share same bottleneck link, each should get 1/N of link capacity



Why is TCP fair?

Two competing sessions:

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



Macroscopic Description of Throughput

- Assume window toggling: $W/2$ to W
- High rate: $W * MSS / RTT$
- Low rate: $W * MSS / 2 RTT$
- Rate increase is linearly between two extremes
- Average throughput:
 - $0.75 * W * MSS / RTT$

TCP latency modeling

Q: How long does it take to receive an object from a Web server after sending a request?

- TCP connection establishment
- data transfer delay

Notation, assumptions:

- Assume one link between client and server of rate R
- Assume: fixed congestion window, W segments
- S : MSS (bits)
- O : object size (bits)
- no retransmissions (no loss, no corruption)

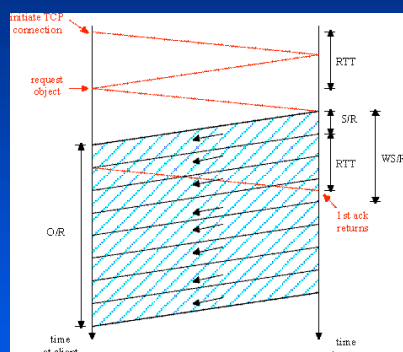
Two cases to consider:

$WS/R > RTT + S/R$: ACK for first segment in window returns before window's worth of data sent

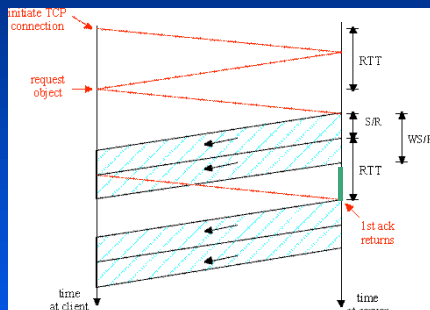
$WS/R < RTT + S/R$: wait for ACK after sending window's worth of data sent

TCP latency Modeling

$$K := O/WS$$



Case 1: latency = $2RTT + O/R$



Case 2: latency = $2RTT + O/R + (K-1)[S/R + RTT - WS/R]$

Green lag

TCP Latency Modeling: Slow Start

- Now suppose window grows according to slow start.
- Will show that the latency of one object of size O is:

$$Latency = 2RTT + \frac{O}{R} + P \left[RTT + \frac{S}{R} \right] - (2^P - 1) \frac{S}{R}$$

where P is the number of times TCP stalls at server:

$$P = \min\{Q, K - 1\}$$

- where Q is the number of times the server would stall if the object were of infinite size.

- and K is the number of windows that cover the object.

TCP Latency Modeling: Slow Start (cont.)

Example:

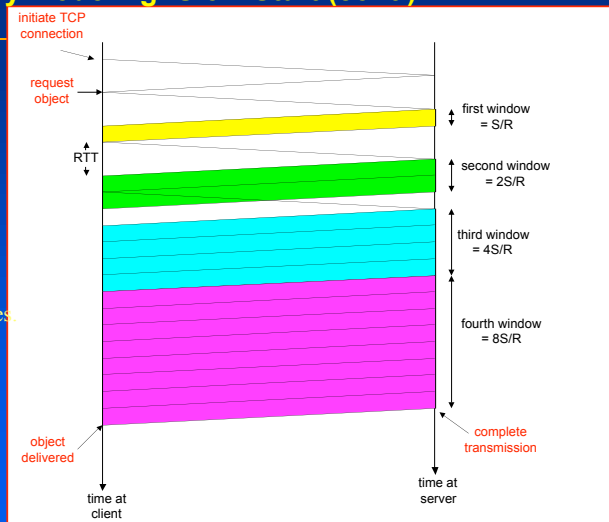
$O/S = 15$ segments

$K = 4$ windows

$Q = 2$

$P = \min\{K-1, Q\} = 2$

Server stalls $P=2$ times



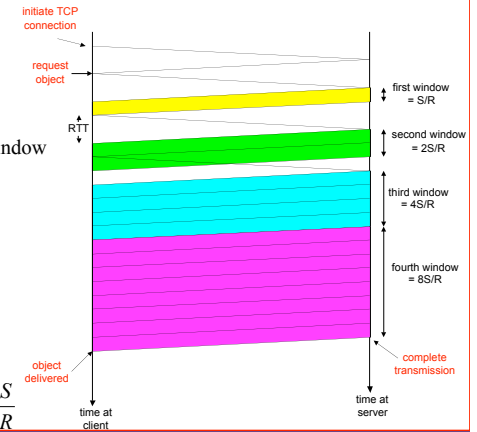
TCP Latency Modeling: Slow Start (cont.)

$\frac{S}{R} + RTT$ = time from when server starts to send segment
until server receives acknowledgement

$2^{k-1} \frac{S}{R}$ = time to transmit the k th window

$\left[\frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right]^+$ = stall time after the k th window

$$\begin{aligned} \text{latency} &= \frac{O}{R} + 2RTT + \sum_{p=1}^P \text{stallTime}_p \\ &= \frac{O}{R} + 2RTT + \sum_{k=1}^P \left[\frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right] \\ &= \frac{O}{R} + 2RTT + P \left[RTT + \frac{S}{R} \right] - (2^P - 1) \frac{S}{R} \end{aligned}$$



Current TCP Versions

✳ TCP specs can be implemented in different ways

✳ TCP versions:

- Tahoe
- Reno
- Las Vegas

TCP Reno

✳ Most popular TCP implementation

✳ Fast retransmit on 3 duplicate ACKs

✳ Fast recovery: cancel slow start after fast retransmission

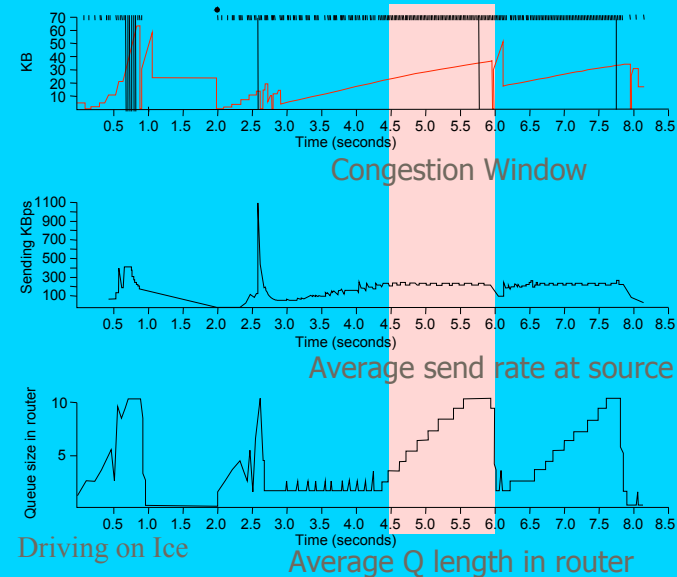
• Optimistic Rationale:

- I hope there was only one packet lost
- Since I sent it, I hope it arrives this time

TCP Vegas

- ☀ **Idea: infer problems from RTT delay**
 - Reduce rate before you have loss
- ☀ **What is a “sign” of congestion:**
 - When RTT increases above a threshold
 - Sending rate flattens
- ☀ **Decrease sending rate linearly**
- ☀ **Issues:**
 - Estimate RTT
 - Set appropriate threshold

Intuition



TCP Vegas Details

- ☀ Value of throughput with no congestion is compared to current throughput
- ☀ If current difference is small, increase window size linearly
- ☀ If current difference is large, decrease window size linearly
- ☀ The change in the Slow Start Mechanism consists of doubling the window every other RTT, rather than every RTT and of using a boundary in the difference between throughputs to exit the Slow Start phase, rather than a window size value.

The TCP Vegas: Algorithm

- ☀ Let BaseRTT be the minimum of all measured RTTs (commonly the RTT of the first packet)
- ☀ If not overflowing the connection, then
 - $\text{ExpectedRate} = \text{CongestionWindow} / \text{BaseRTT}$
- ☀ Source calculates current sending rate (ActualRate) once per RTT
- ☀ Source compares ActualRate with ExpectedRate
 - $\text{Diff} = \text{ExpectedRate} - \text{ActualRate}$
 - if $\text{Diff} < \alpha$
 - \rightarrow increase CongestionWindow linearly
 - else if $\text{Diff} > \beta$
 - \rightarrow decrease CongestionWindow linearly
 - else
 - \rightarrow leave CongestionWindow unchanged

Vegas Parameters

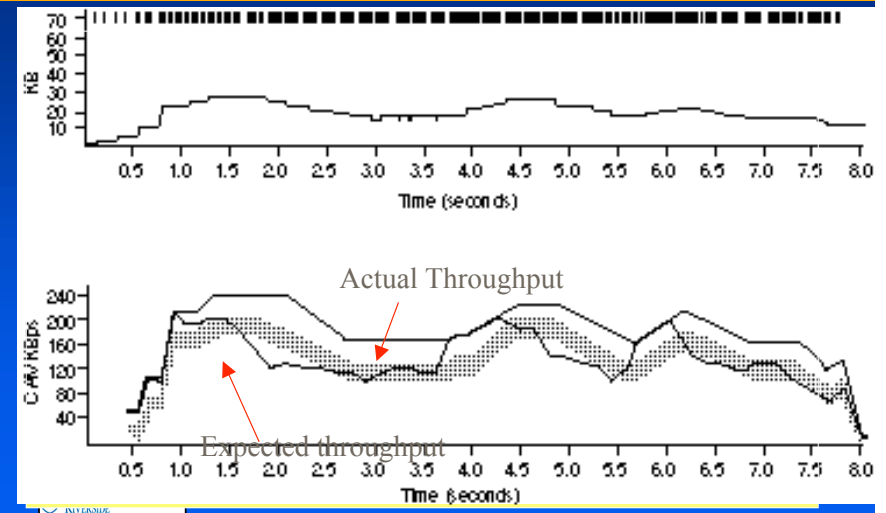
Parameters

- α : 1 packet
- β : 3 packets

Even faster retransmit

- keep fine-grained timestamps for each packet
- check for timeout on first duplicate ACK

Example TCP Vegas



Router Assisted Congestion Control

- Random Early Detection
- Explicit Congestion Notification

Note: often this is referred to as Active Networking: ie routers are involved in performance.

Active Nets is a much more general idea

RED: Random Early Detection

- Idea: routers start dropping packets before they are congested
- Benefits: make behavior smoother
- How:
 - When queue is above a thres-1: drop packets with probability p
- Issues:
 - setting the parameters
 - Estimating the queue size

Thresholds

- two queue length thresholds
 - if $AvgLen \leq MinThreshold$ then
 - enqueue the packet
 - if $MinThreshold < AvgLen < MaxThreshold$
 - calculate probability P
 - drop arriving packet with probability P
 - if $MaxThreshold \leq AvgLen$
 - drop arriving packet

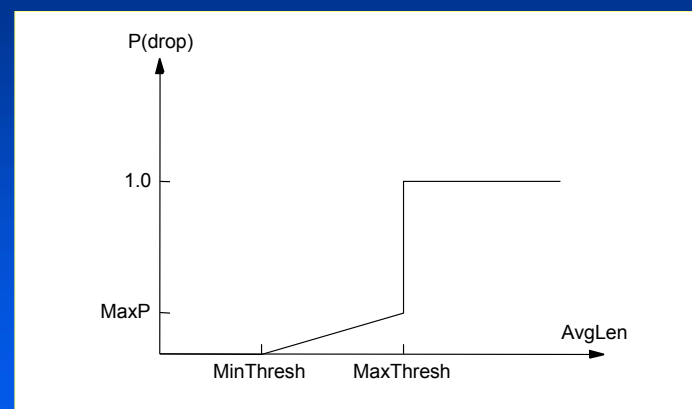
RED: probability P

- **Not fixed**
- **Function of $AvgLen$ and how long since last drop (count) keeps track of new packets that have been queued while $AvgLen$ has been between the two thresholds**
 - $TempP = MaxP * (AvgLen - MinThreshold) / (MaxThreshold - MinThreshold)$
 - $P = TempP / (1 - count * TempP)$
- **$MaxP$ is often set to 0.02, meaning that the gateway drops 1 out of 50 packets when queue size is halfway between $MinThreshold$ and $MaxThreshold$**

Comments on RED

- **Probability of dropping a particular flow's packet(s) is roughly proportional to the share of the bandwidth that flow is currently getting**
- **$MaxP$ is typically set to 0.02, meaning that when the average queue size is halfway between the two thresholds, the gateway drops roughly one out of 50 packets.**

RED: Dropping probability



Selecting Parameters

- ✱ if traffic is bursty, then MinThreshold should be sufficiently large to allow link utilization to be maintained at an acceptably high level
- ✱ difference between two thresholds should be larger than the typical increase in the calculated average queue length in one RTT; setting MaxThreshold to twice MinThreshold is reasonable for traffic on today's Internet

Explicit Congestion Notification

- ✱ Dropping packets = Warn of congestion
- ✱ Idea: mark packets to notify congestion
- ✱ How:
 - Congested router marks packet (sets a bit)
 - Receiver "copies" bit in the ACK
 - Sender reduces its window
- ✱ Benefit: proactive without losing packets
- ✱ Problem: sender can ignore it

Current Beliefs

- ✱ RED + ECN are considered to be good
- ✱ RED alone has problems

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 layer)
 - ✱ into the network "core"

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.

Last ACK is never ACK-ed!!

