Electrical Engineering E6761
Computer Communication Networks
Lecture 4
Transport Layer Services:
TCP, Congestion Control

Professor Dan Rubenstein
Tues 4:10-6:40, Mudd 1127
Course URL:
http://www.cs.columbia.edu/~danr/EE6761

Today

- Project / PA#2
- Clarifications / Corrections from last lecture
- Transport Layer
  - Example protocol: TCP
    - connection setup / teardown
    - flow control
  - congestion control

Project

- The project assignment is not fixed.
  - Your group should come up with its own idea
  - If group can't decide, I can come up with some possible topics (in a few weeks)
  - Project style (programming, math analysis, etc.)
  - again, up to the group
  - could be 1 type or a mix (e.g., half programming, half analysis)
- Start thinking about forming groups

PA#2

- Much harder than PA#1
  - more coding
  - more creativity (decisions) you have to make
  - more complexity (maintaining window, timeouts, etc.)
- Recommendations:
  - Have the sender read in a file and send the file (or some other means of sending a variable-length msg)
  - You can assume your sender has an infinite buffer (but not the receiver)
- Extra-credit: checking for bit errors was not required. Include a checksum for extra credit

PA#2 cont'd

- useful function: gettimeofday()
  - gettimeofday(&t, NULL) stores # of clock ticks elapsed in t
  - struct timeval t {
    long tv_sec; /* elapsed seconds */
    long tv_usec; /* elapsed microseconds (0-999999) */
    }
  - useful for timing / timeouts (in conjunction w/ select)
- Q: how could your sender check for multiple timeouts, plus watch for incoming ACKs at the same time?

PA#2: use select()

e.g., selective-repeat

- maintain a window's worth of timeouts

```
struct TO_track {
  struct timeval TO_time;
  long int seqno;
}
struct TO_track TO[WINSIZE];
```

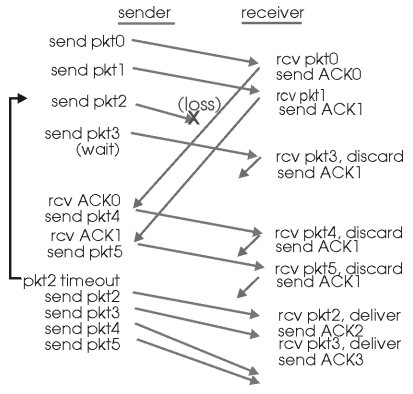
- Also, maintain
  - a timer for connection abort (struct timeval conn_abort)
  - a socket on which ACKs arrive (socket sock)

PA#2 cont'd: select()

```
struct timeval set next_TO, cur_time, select_wait_time;
fd_set readfds;

cur_time = gettimeofday(); /* current time */
/* you have to write min_time and DiffTime funcs */
next_TO = min_time(TO[i], conn_abort);
select_wait_time = TimeDiff(cur_time, next_TO);
FD_ZERO(&readfds); FD_SET(sock, &readfds);

status = select(sock+1, &readfds, NULL, NULL, &select_wait_time);

/* when select returns, either the earliest TO has expired or else sock
    has data to read */

if (FD_ISSET(sock, &readfds)){ /* can read from socket */ … }
else { /* Handle the appropriate TO */}
```


Review: GBN in action

Here, N=4

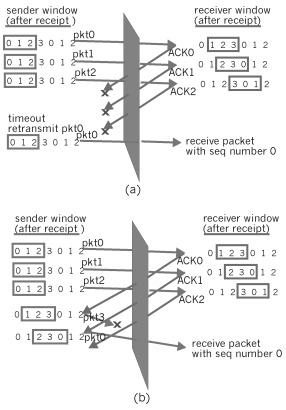


Review: Selective repeat: dilemma

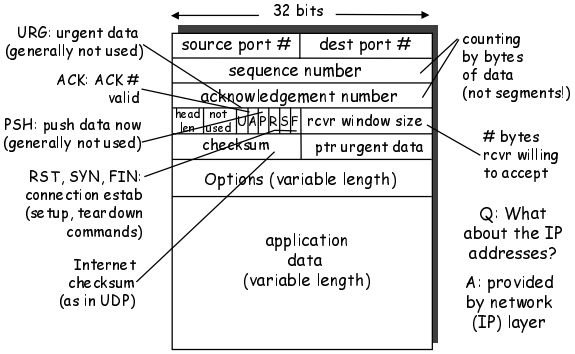
Example:

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

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



TCP segment structure



TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - 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's buffer
- congestion controlled:
  - sender will not overwhelm network resources

TCP seq. #'s and ACKs

Seq. #'s:

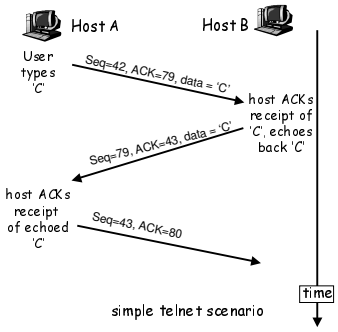
- byte stream "number" of first byte in segment's data

ACKs:

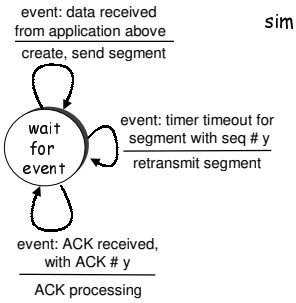
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments (i.e., drop v. buffer)

- A: TCP spec doesn't say, - up to implementor



TCP: reliable data transfer



simplified sender, assuming

- one way data transfer
- no flow, congestion control

TCP: reliable data transfer

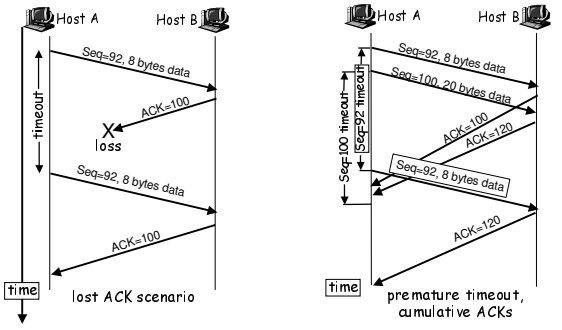
Simplified TCP sender

```
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



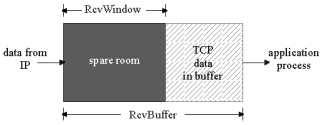
TCP Flow Control

flow control

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

RcvBuffer = size of TCP Receive Buffer

RcvWindow = amount of spare room in Buffer



receiver buffering

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

- **RcvWindow field** in TCP segment

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

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**

Exponentially Weighted Moving Average

Useful when average is time-varying

- Let A_t be the average computed for time t = 0,1,2,…
- Let S_t be the sample taken at time t
- Let x be the weight
- $A_0 = S_0$
- $A_t = (1-x)\, A_{t-1} + x\, S_t$ for $t > 0$

  $= (1-x)^t\, S_0 + x \sum_{i=1}^{t} (1-x)^{t-i}\, S_i$

A larger x means more emphasis on recent measurements, less on history

(e.g., x = 1 gives $A_t = S_t$)

- has "Desirable" average features:
  - If $S_i = C$ for all i, then $A_i = C$
  - if $\lim_{i\to\infty} S_i = C$, then $\lim_{i\to\infty} A_i = C$
  - if $C_1 \le S_i \le C_2$ for all i, then $C_1 \le A_i \le C_2$
- gives more "weight" to more recent samples

TCP Round Trip Time and Timeout

```
EstimatedRTT = (1-x)*EstimatedRTT + x*SampleRTT
```

- Exponential weighted moving average
- typical value of x: 0.1

Setting the timeout

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

```
Timeout = EstimatedRTT + 4*Deviation

Deviation = (1-x)*Deviation +
            x*|SampleRTT-EstimatedRTT|
```

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
  `connect()`
- *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. #

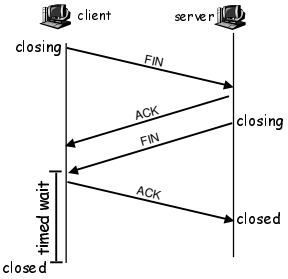
TCP Connection Management (cont.)

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.

Q: why use a timed wait at end instead of another ACK?



TCP Connection Management (cont.)

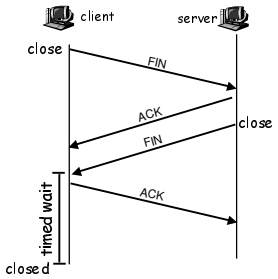
Closing a connection:

here (in example), client closes socket:
`clientSocket.close();`

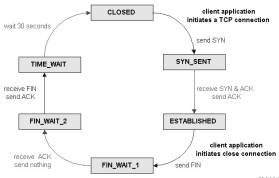
In practice, either side can close (NOTE: closes communication in **both** directions)

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

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

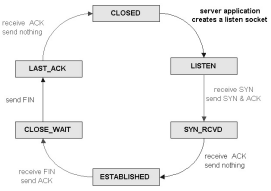


TCP Connection Management (cont)



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)

25

Some Defintions for Congestion Control

- Throughput: rate at which bits are pumped into a network or link or router (incl. retransmitted bits)
- Goodput: rate at which new data (bits) exits the network or link or router
- Efficiency: = Goodput / Throughput

26

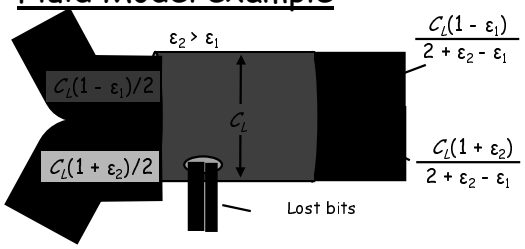
CC Network model #1

Fluid model

- Each link, L, is a pipe with some capacity C_L
- Each session, S, is a fluid pumped in at a rate R_S
- Link drop rate, D_L:
  - assume N fluids enter L at rates $e_1, e_2, \ldots, e_N$
  - Let $E_L = e_1+e_2+\ldots+e_N$
  - $$D_L = \begin{cases} 1 - C_L / E_L & E_L > C_L \\ 0 & \text{otherwise} \end{cases}$$
  - Each flow loses a fraction, D_L, of bits through L
  - fluids exit L at rate $e_1(1 - D_L), e_2(1 - D_L), \ldots, e_N(1 - D_L)$

27

Fluid Model example

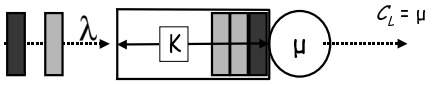


- Red flow: transmission rate a bit less than $.5 C_L$
- Green flow: transmission ratebit more than $.5\ C_L$
- Red+Green: together transmit a bit more than C_L

28

CC Network Model #2

Queuing model (each router or link rep'd by a queue)

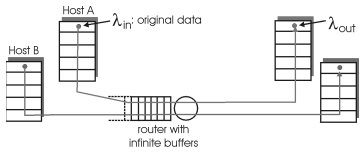


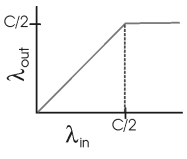
- Buffer of size K
- Packets arrive at rate λ
- Packets are processed at rate μ
  (hence, link speed out equals μ)
- Rates and distributions affect "levels" of congestion

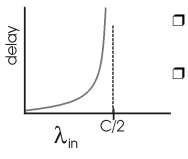
Queuing Models will reappear later in course

Causes/costs of congestion: scenario 1

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



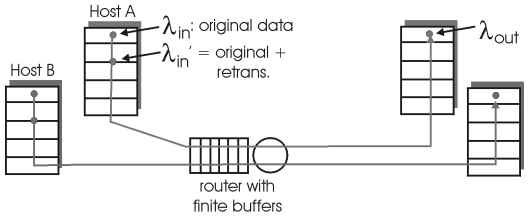




- large delays when congested
- maximum achievable throughput

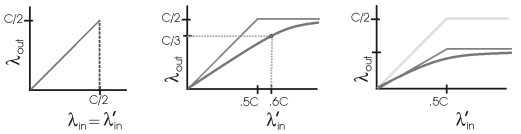
Causes/costs of congestion: scenario 2

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



Causes/costs of congestion: scenario 2

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

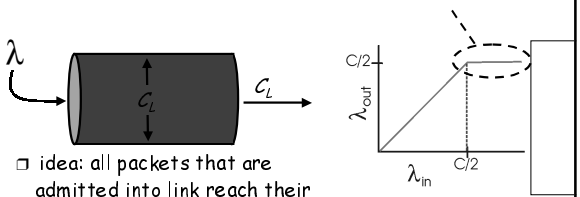


"costs" of congestion:

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

Full network utilization?

- Idea: make buffers small
  - little delay (i.e. reduces duplicates problem)
  - packet lost at entry to link, simply retransmit
  - i.e., throughput in @ $\lambda > C_L$, goodput out at C_L

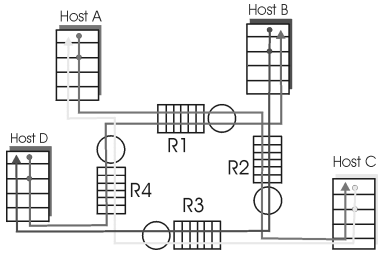


- idea: all packets that are admitted into link reach their destination. Any problems?

Multiple Hops: scenario 3

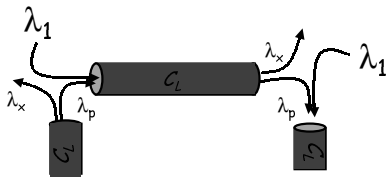
- four senders
- multihop paths
- timeout/retransmit

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



Fluid model of 2-hop system

- Assume symmetry at each link:
  - link has capacity C_L
  - is 1st hop for one flow (into link @ rate λ_1)
  - is 2nd hop for other (into link @ rate λ_p)
  - is last hop for other (out of prev. rate λ_x)



Fluid model, 2 hop (cont'd)

$\lambda_1 > C_L / 2$

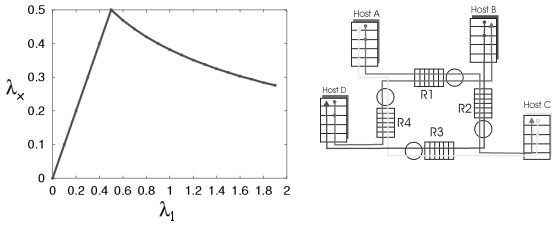
- $\lambda_x + \lambda_p = C_L$
- $D_L = 1 - C_L / (\lambda_1 + \lambda_p)$
- $\lambda_x = \lambda_p (1 - D_L)$
- $\lambda_p = \lambda_1 (1 - D_L)$
- Sol'n:

$\lambda_x = C_L + (\lambda_1 - \sqrt{\lambda_1^2 + 4C_L\lambda_1})/2$

$\lambda_1 \leq C_L / 2$ (link under-utilized)

- $\lambda_x = \lambda_p = \lambda_1$
- $D_L = 0$

Causes/costs of congestion: scenario 3



results from 2-hop fluid model

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

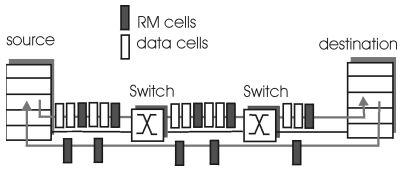
ABR: available bit rate:

- "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' 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, dest. 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:

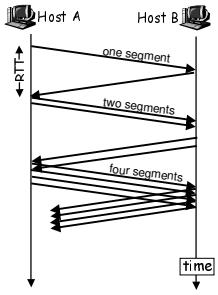
$$\text{throughput} = \frac{w * MSS}{RTT} \text{ Bytes/sec}$$

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 (not so slow!)
- loss event: timeout (Tahoe TCP) and/or or three duplicate ACKs (Reno TCP)



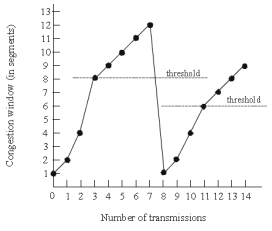
TCP congestion control:

- "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 Congestion Avoidance

Congestion avoidance

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



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

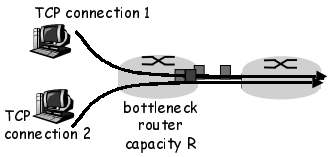
AIMD

TCP congestion avoidance:

- AIMD: *additive increase, multiplicative decrease*
  - increase window by 1 per RTT
  - decrease window by factor of 2 on loss event

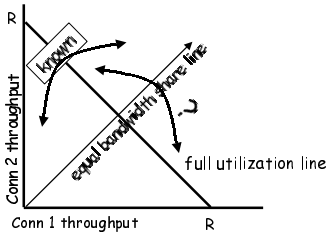
TCP Fairness

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



Chiu/Jain model assumptions

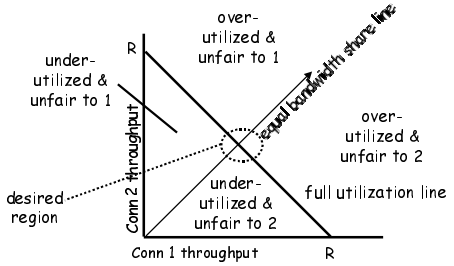
- Sessions can sense whether link is overused or underused (e.g., via lost pkts)
- Sessions cannot compare relative rates (i.e., don't know of each other's existence)



- Sessions adapt rates round-by-round
  - adapt simultaneously
  - in same direction (both increase or both decrease)

Why is AIMD fair and congestion- avoiding?

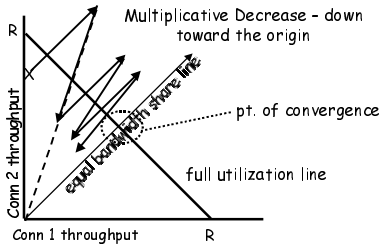
Pictorial View: Two sessions compete for a link's bandwidth, R (see Chiu/Jain paper)



A good CC protocol will always converge toward the desired region

AIMD Convergence (Chiu/Jain)

Additive Increase - up at 45° angle



C/J also show other combos (e.g., AIAD) don't converge!

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)

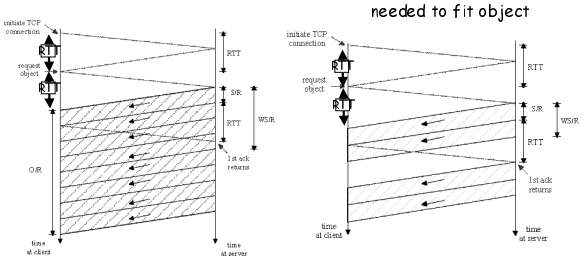
Two cases to consider:

- 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

S/R = time to a packet's bits into the link

TCP latency Modeling

K:=⌈O/WS⌉= # of windows needed to fit object



Case 1: latency = 2RTT + O/R

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

idle time bet. window transmissions

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:

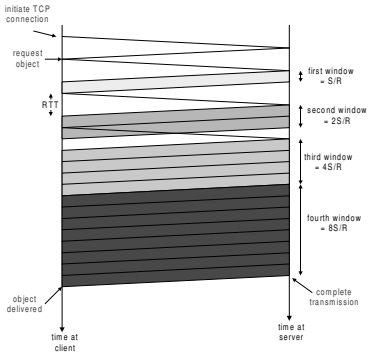
O/S = 15 segments

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

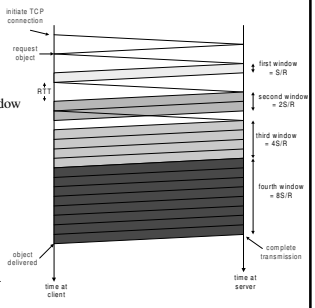
$2^{k-1}\frac{S}{R}$ = time to transmit the kth window

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

$$\text{latency} = \frac{O}{R} + 2RTT + \sum_{p=1}^{P} stallTime_p$$

$$= \frac{O}{R} + 2RTT + \sum_{k=1}^{P}[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}]$$

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



Non-unicast modes of communication

- So far, we have only looked at unicast (one host to one host) communication
- Other forms of communication
  - broadcast
  - multicast
  - anycast

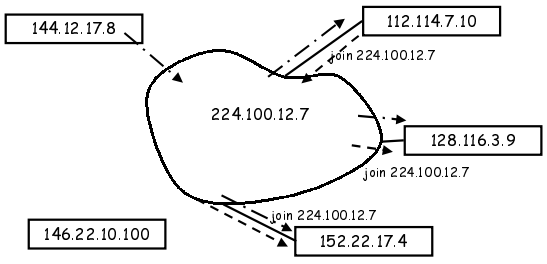
Transport Layer Multicast

Requires Multicast IP addressing

- class D addresses (224.0.0.0 - 239.255.255.255) reserved for multicast
- each address identifies a **multicast group**
  - address not explicitly associated with any host
  - hosts must join to the group to receive data sent to the group
- Any sender that sends to the multicast group will have its transmission delivered to all receivers joined to the multicast group

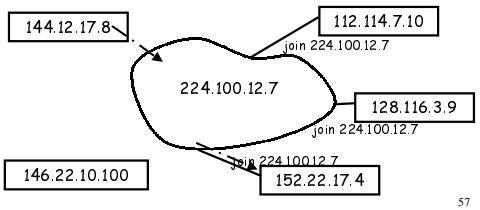
  (Note: delivery is UDP-like: unreliable, no order guarantees, etc.)
- joins accomplished through a socket interface

Multicast Example



Transport Layer Anycast

- Multicast: packet delivered to all group members
- Anycast: packet delivered to just one (any) member (still under development in Internet)
- Useful for locating some (replicated) host
- Possible mode of operation:



57

Transport Layer: Summary

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

Next time:

- leaving the network "edge" (application transport layer)
- into the network "core"

58