


# Transport Layer


## CS 3516 - Computer Networks



# Chapter 3: Transport Layer


**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



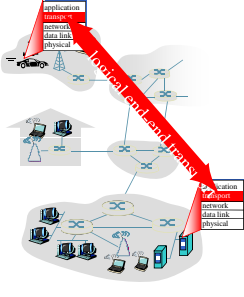

# Chapter 3 outline

- 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 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
  - receive side: reassembles segments into messages, passes to app layer
- More than one transport protocol available to apps
  - Internet: TCP and UDP





# Transport vs. Network layer

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

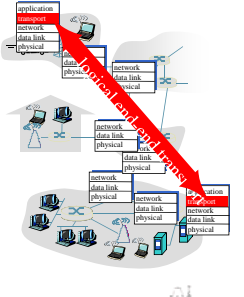

**Household analogy:**  
12 kids sending letters to 12 kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill (collect mail from siblings)
- network-layer protocol = postal service




# 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


## 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



## 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




## 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

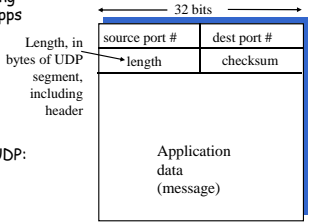
**Why is there a UDP?**

- 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




## UDP: more

- Often used for streaming (video/audio) or game apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!



UDP segment format



## 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*



## Internet Checksum Example


- 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 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 1 0 0
checksum     0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1


```

- At receiver, add 2 integers and checksum ... should be all 1's. If not, bit error (correction? → next)



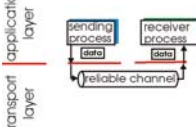
## Chapter 3 outline

- 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




## Principles of Reliable data transfer

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



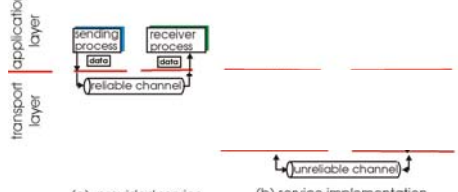
(a) provided service

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




## Principles of Reliable data transfer

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



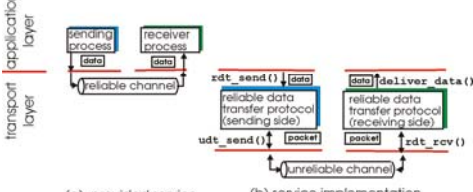
(a) provided service      (b) service implementation

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



## Principles of Reliable Data Transfer


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



(a) provided service      (b) service implementation

(Zoom next slide)

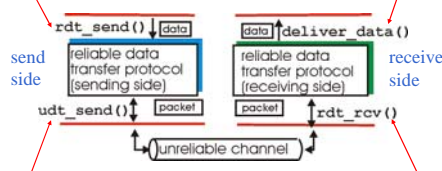
- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)



## Reliable Data Transfer: Getting Started


**rdt\_send()**: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

**deliver\_data()**: called by rdt to deliver data to upper



**udt\_send()**: called by rdt, to transfer packet over unreliable channel to receiver

**rdt\_rcv()**: called when packet arrives on rcv-side of channel

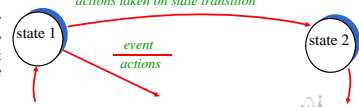


## Reliable Data Transfer: Getting Started


**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

*event causing state transition*  
*actions taken on state transition*



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



### 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

sender      Easy!      receiver

### What if Taking a Message over Phone?

- Message is clear?
- Message is garbled?

### What if Taking a Message over Phone?

- Message is clear?
  - Ok
- Message is garbled?
  - Ask to repeat
  - May not need whole message
- In networks, called Automatic Repeat reQuest (ARQ)
  - Need error detection
  - Receiver feedback
  - Retransmission

### Rdt2.0: Channel with Bit Errors (no Loss)

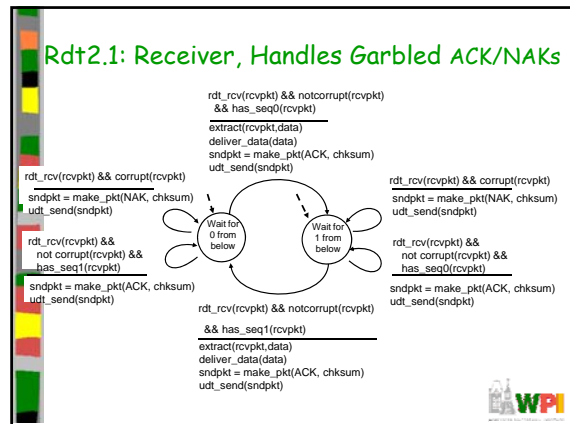
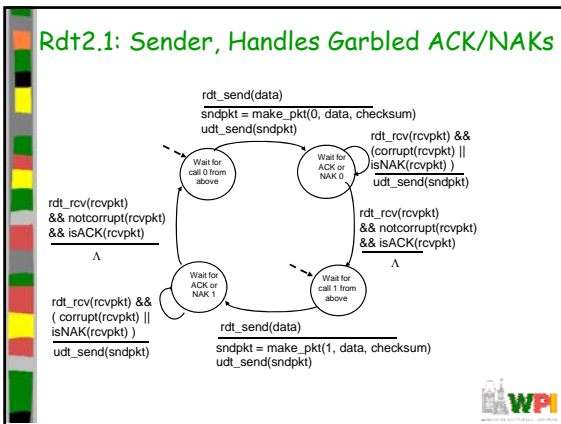
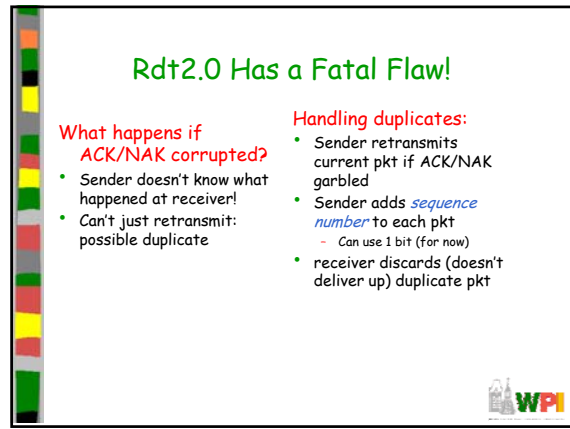
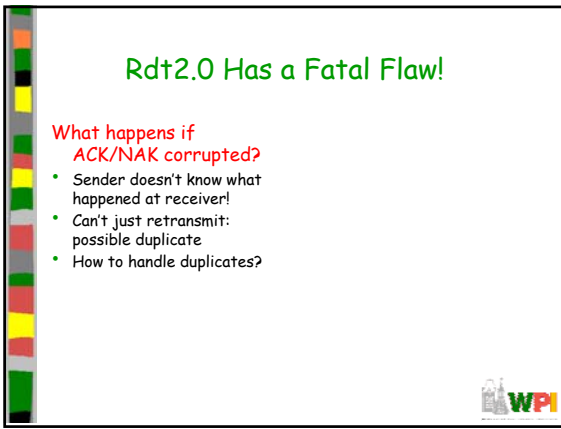
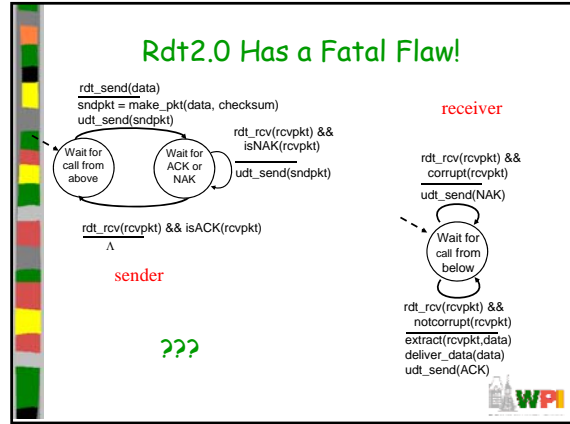
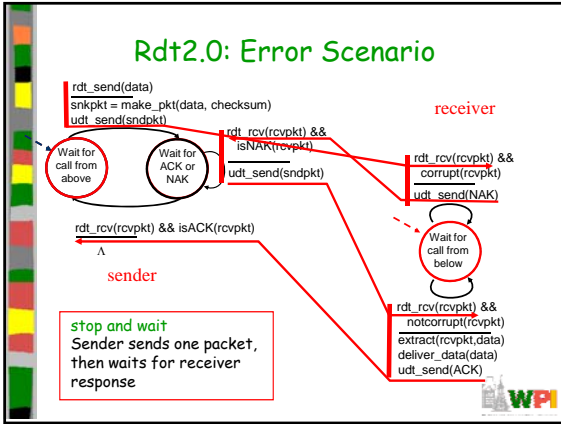
- Underlying channel may flip bits in packet
  - Checksum to detect bit errors
- The question: how to recover from errors:
  - acknowledgements (ACKs): receiver explicitly tells sender that pkt received OK
  - negative acknowledgements (NAKs): receiver explicitly tells sender that pkt had errors
  - 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

### Rdt2.0: FSM Specification

sender      receiver

### Rdt2.0: Operation with No Errors

sender      receiver



## Rdt2.1: Discussion

### Sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice
- 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

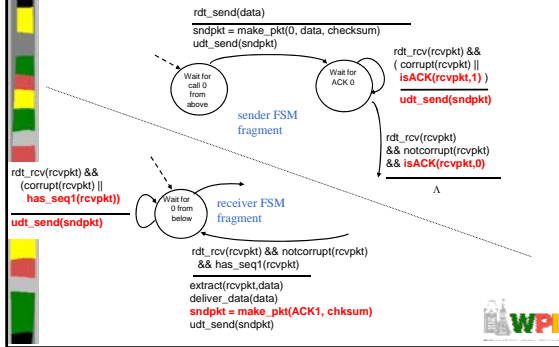


## Rdt2.2: a NAK-free Protocol

- Reduce type of response → ACK only
- 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*



## Rdt2.2: Sender & Receiver Fragments



## 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

How to determine if a packet is lost?



## Rdt3.0: Channels with Errors and Loss

### New assumption:

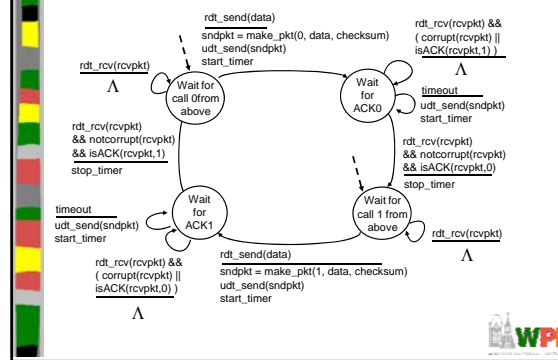
- underlying channel can also lose packets (data or ACKs)
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

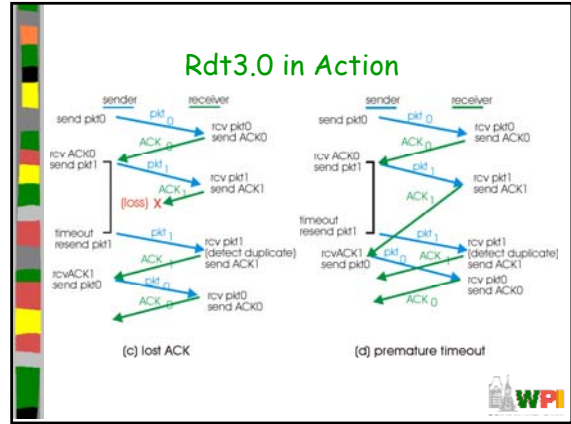
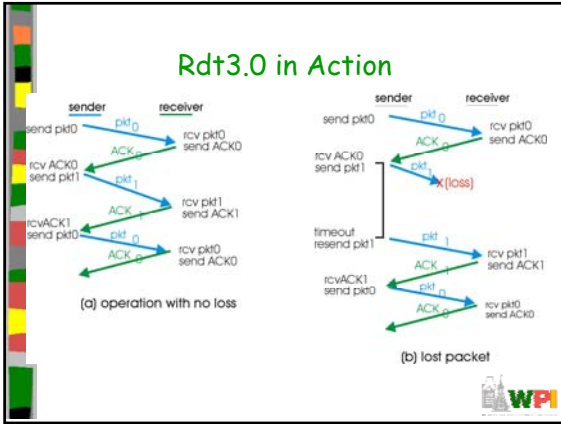
**Approach:** sender waits "reasonable" amount of time for ACK

- 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



## 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

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

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

(Picture next slide)

WPI

### Rdt3.0: Stop-and-Wait Operation

first packet bit transmitted,  $t = 0$

last packet bit transmitted,  $t = L/R$

RTT

first packet bit arrives

last packet bit arrives, send ACK

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

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

WPI

### Pipelined Protocols

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

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

(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

WPI

### Pipelining: Increased Utilization

first packet bit transmitted,  $t = 0$

last bit transmitted,  $t = L/R$

RTT

first packet bit arrives

last packet bit arrives, send ACK

last bit of 2<sup>nd</sup> packet arrives, send ACK

last bit of 3<sup>rd</sup> packet arrives, send ACK

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

Increase utilization by a factor of 3!

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

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

WPI

## Pipelining Protocols

### Go-back-N: overview

- **sender:** up to N unACKed pkts in pipeline
- **receiver:** only sends cumulative ACKs
  - doesn't ACK pkt if there's a gap
- **sender:** has timer for oldest unACKed pkt
  - if timer expires: retransmit all unACKed packets

### Selective Repeat: overview

- **sender:** up to N unACKed packets in pipeline
- **receiver:** ACKs individual pkts
- **sender:** maintains timer for each unACKed pkt
  - if timer expires: retransmit only unACKed packet



## Go-Back-N

### Sender:

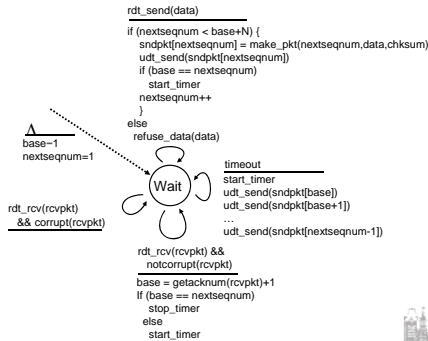
- k-bit seq # in pkt header
- "window" of up to N, consecutive unACKed pkts allowed



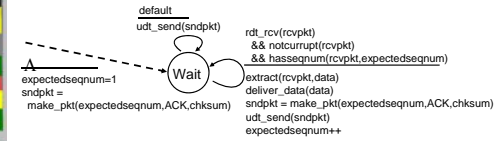
- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- Timer for each in-flight pkt
- Timeout(n): retransmit pkt n and all higher seq # pkts in window



## GBN: Sender Extended FSM



## GBN: Receiver Extended FSM



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

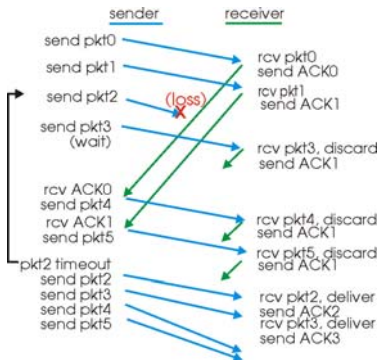
- may generate duplicate ACKs
- need only remember `expectedseqnum`

### out-of-order pkt:

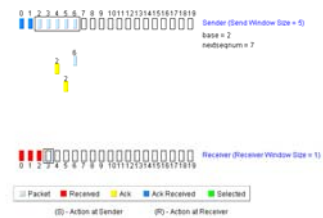
- discard (don't buffer) -> **no receiver buffering!**
- Re-ACK pkt with highest in-order seq #



## GBN in action



## GBN Applet!




[http://media.pearsoncmg.com/api/v1/kuross-network\\_4/applets-go-back-n/index.html](http://media.pearsoncmg.com/api/v1/kuross-network_4/applets-go-back-n/index.html)



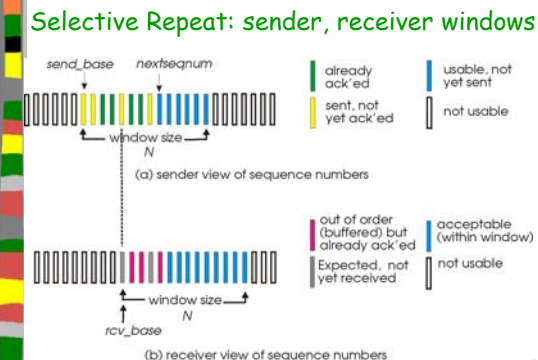



## Selective Repeat

- 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



## Selective Repeat: sender, receiver windows

## 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]**

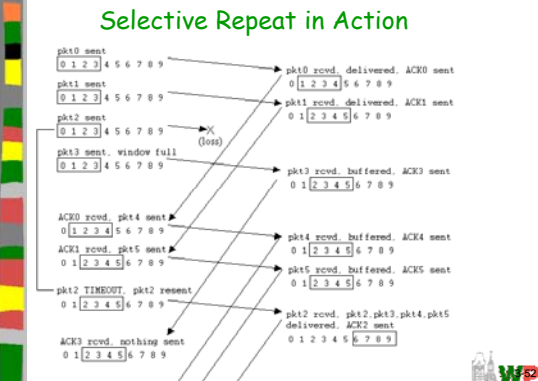

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

**pkt n in [rcvbase-N, rcvbase-1]**

- ACK(n)
- ignore



## Selective Repeat in Action

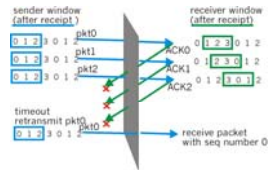



## Selective Repeat: Dilemma

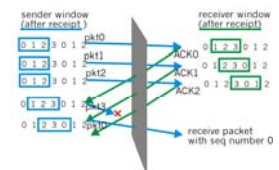
**Example:**

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


(a)




(b)




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



## SR Applet!



[http://media.pearsoncmg.com/aw/aw\\_kurose\\_network\\_4/applets/SR/index.html](http://media.pearsoncmg.com/aw/aw_kurose_network_4/applets/SR/index.html)



## 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

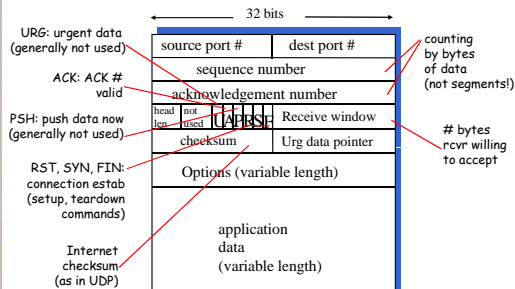


## TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

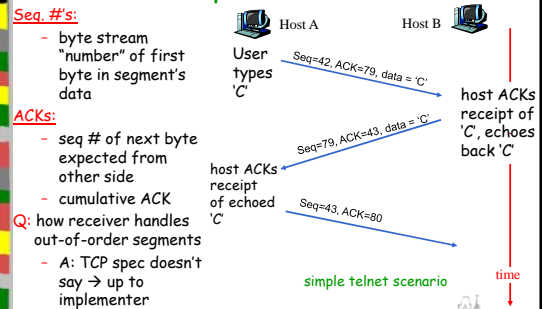
- point-to-point:**
  - one sender, one receiver
- reliable, in-order byte stream:**
  - no "message boundaries"
- pipelined:**
  - TCP congestion and flow control set window size
- send & receive buffers**
- full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:**
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:**
  - sender will not overwhelm receiver



## TCP Segment Structure



## TCP Seq. #'s and ACKs



## TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?



## TCP Round Trip Time and Timeout

- Q: how to set TCP timeout value?      Q: how to estimate RTT?
- Longer than RTT
    - but RTT varies
  - Too short? premature timeout
    - unnecessary retransmissions
  - Too long? slow reaction to segment loss



## 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**



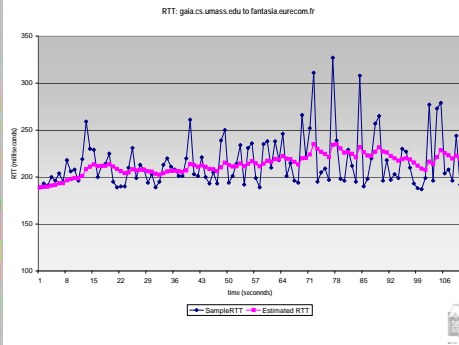
## TCP Round Trip Time and Timeout

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

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value:  $\alpha = 1/8^{\text{th}}$  (or 0.125)



## Example Round Trip Time Estimation



## 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:

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



## 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



## 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



## 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

## TCP Sender (simplified)

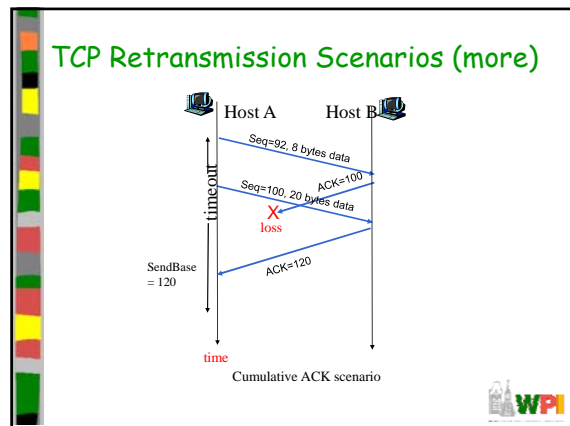
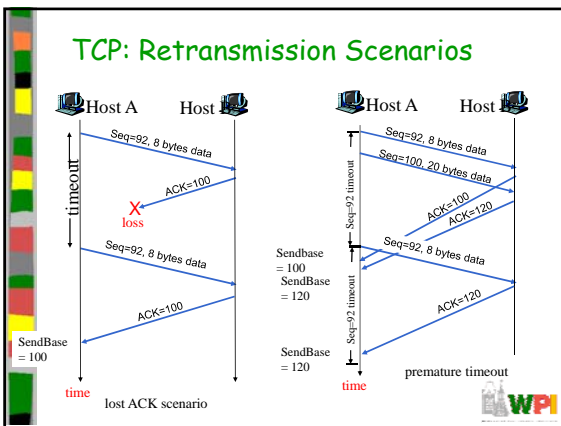
```

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)
  event: data received from application above
    create TCP segment w/seq # NextSeqNum
    if (timer currently not running)
      start timer
    pass segment to IP
    NextSeqNum = NextSeqNum + length(data)
  event: timer timeout
    retransmit not-yet-acked segment with
    smallest sequence number
    start timer
  event: ACK received, with ACK field value of y
    if (y > SendBase) {
      SendBase = y
      if (there are not-yet-acked segments)
        start timer
    }
} /* end of loop forever */
  
```

**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

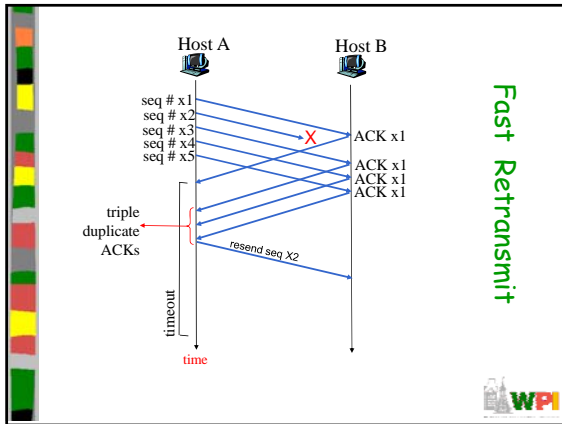


## TCP ACK generation [RFC 1122, RFC 2581]

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 <b>duplicate ACK</b> , 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

## 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 lost, there will likely be many duplicate ACKs for that segment
- If sender receives 3 ACKs for same data, it assumes that segment after ACKed data was lost:
  - **fast retransmit:** resend segment before timer expires



### 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

### TCP Flow Control

- Receive side of TCP connection has a receive buffer:

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

- **speed-matching service**: matching send rate to receiving application's drain rate

- App process may be slow at reading from buffer

### TCP Flow Control: How it Works

- Receiver: advertises unused buffer space by including  $rwnd$  value in segment header
- sender: limits # of unACKed bytes to  $rwnd$ 
  - guarantees receiver's buffer doesn't overflow

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

- unused buffer space:  
 $= rwnd$   
 $= RcvBuffer - [LastByteRcvd - LastByteRead]$

### 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

### 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#);`
- **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

### TCP Connection Management (cont.)

**Closing a connection:**

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

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

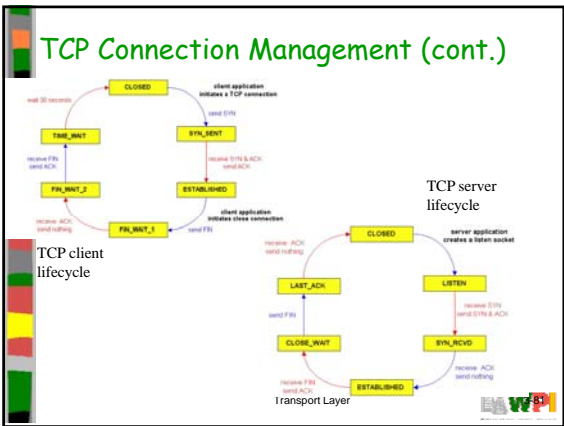
**Step 2:** server receives FIN, replies with ACK. Closes connection, sends FIN.

### 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.



### 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

### 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!

### 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

Host A:  $\lambda_{in}$ : original data  
 $\lambda'_{in}$ : original data, plus retransmitted data

Router: finite shared output link buffers

Host B:  $\lambda_{in}$

Router:  $\lambda_{out}$

WPI

### Causes/costs of congestion: Scenario 2

- Always:  $\lambda_{in} = \lambda_{out}$  (goodput)
- "Perfect" retransmission only when loss:  $\lambda'_{in} > \lambda_{out}$ ,  $\lambda'_{in}$  larger (than perfect case) for same  $\lambda_{out}$

a.  $\lambda_{out} = \lambda_{in}$  for  $\lambda_{in} \leq R/2$

b.  $\lambda_{out} = \lambda_{in}$  for  $\lambda_{in} \leq R/2$

c.  $\lambda_{out} = \lambda_{in}$  for  $\lambda_{in} \leq R/4$

"Costs" of congestion:

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

WPI

### Causes/costs of Congestion: Scenario 3

- Four senders
- Multihop paths
- Timeout/retransmit

Q: what happens as  $\lambda_{in}$  increase?

Host A:  $\lambda_{in}$ : original data  
 $\lambda'_{in}$ : original data, plus retransmitted data

Router: finite shared output link buffers

Host B:  $\lambda_{in}$

Router:  $\lambda_{out}$

WPI

### Causes/costs of Congestion: Scenario 3

Another "cost" of congestion:

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

WPI

### Approaches towards congestion control

Broadly:

<p><b>End-end congestion control:</b></p> <ul style="list-style-type: none"> <li>No explicit feedback from network</li> <li>Congestion inferred from end-system observed loss, delay</li> <li>Approach taken by TCP</li> </ul>	<p><b>Network-assisted congestion control:</b></p> <ul style="list-style-type: none"> <li>Routers provide feedback to end systems <ul style="list-style-type: none"> <li>Single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)</li> <li>Explicit rate sender should send at</li> </ul> </li> </ul>
--	--

WPI


### 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

WPI

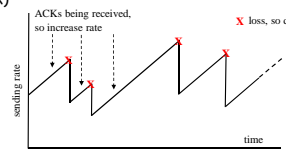
### TCP Congestion Control:

- Goal:** TCP sender should transmit as fast as possible, but without congesting network
  - Q:** how to find rate, just below congestion level?
- Decentralized: each TCP sender sets its own rate, based on **implicit** feedback:
  - ACK:** segment received (a good thing!), network not congested, so increase sending rate
  - lost segment:** assume loss due to congested network, so decrease sending rate




### TCP Congestion Control: Bandwidth Probing

- "Probing for bandwidth":** increase transmission rate on receipt of ACK, until eventually loss occurs, then decrease transmission rate
  - continue to increase on ACK, decrease on loss (since available bandwidth is changing, depending on other connections in network)



TCP's "sawtooth" behavior

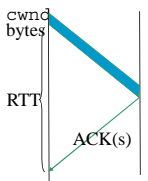

- Q:** how fast to increase/decrease?
  - details to follow



### TCP Congestion Control: details


- sender limits rate by limiting number of unACKed bytes "in pipeline":
  - $LastByteSent - LastByteAcked \leq cwnd$
  - cwnd:** differs from **rwnd** (how, why?)
  - sender limited by  $\min(cwnd, rwnd)$
- roughly,
 

$$rate = \frac{cwnd}{RTT} \text{ bytes/sec}$$
- cwnd** is dynamic, function of perceived network congestion

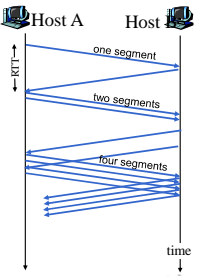

### TCP Congestion Control: more details

- segment loss event: reducing cwnd**
  - timeout: no response from receiver
    - cut cwnd to 1
  - 3 duplicate ACKs: at least some segments getting through (recall fast retransmit)
    - cut cwnd in half, less aggressively than on timeout
- ACK received: increase cwnd**
  - slowstart phase:
    - increase exponentially fast (despite name) at connection start, or following timeout
  - congestion avoidance:
    - increase linearly



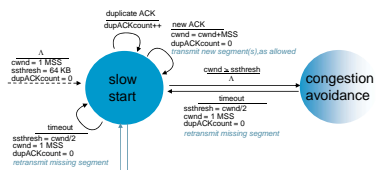

### TCP Slow Start

- when connection begins, **cwnd** = 1 MSS
  - example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be  $\gg$  MSS/RTT
  - desirable to quickly ramp up to respectable rate
- increase rate exponentially until first loss event or when threshold reached
  - double cwnd every RTT
  - done by incrementing cwnd by 1 for every ACK received

### Transitioning into/out of slowstart

- ssthresh:** cwnd threshold maintained by TCP
- on loss event: set **ssthresh** to **cwnd/2**
  - remember (half of) TCP rate when congestion last occurred
- when **cwnd**  $\geq$  **ssthresh**: transition from slowstart to congestion avoidance phase




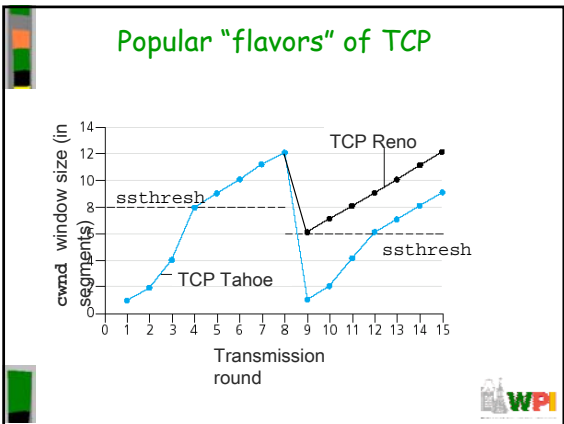
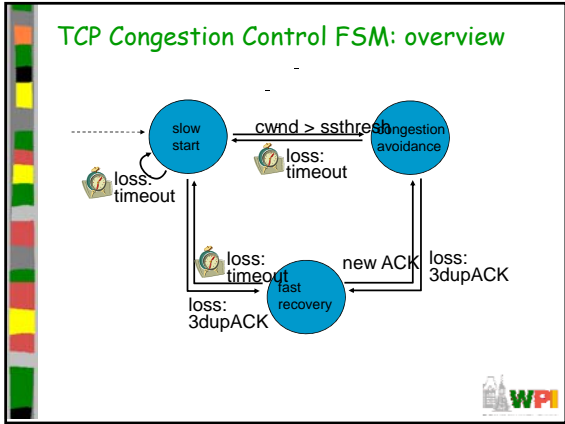
### TCP: Congestion Avoidance


- When  $cwnd > ssthresh$  grow  $cwnd$  linearly
  - increase  $cwnd$  by 1 MSS per RTT
  - approach possible congestion slower than in slowstart
  - implementation:  $cwnd = cwnd + MSS/cwnd$  for each ACK received


**AIMD**


- ACKs:** increase  $cwnd$  by 1 MSS per RTT: additive increase
- loss:** cut  $cwnd$  in half (non-timeout-detected loss): multiplicative decrease

AIMD: Additive Increase  
Multiplicative Decrease

- ### Summary: TCP Congestion Control
- when  $cwnd < ssthresh$ , sender in **slow-start** phase, window grows exponentially.
  - when  $cwnd \geq ssthresh$ , sender is in **congestion-avoidance** phase, window grows linearly.
  - when **triple duplicate ACK** occurs,  $ssthresh$  set to  $cwnd/2$ ,  $cwnd$  set to  $\sim ssthresh$
  - when **timeout** occurs,  $ssthresh$  set to  $cwnd/2$ ,  $cwnd$  set to 1 MSS.
- 

- ### TCP throughput
- Q:** what's average throughput of TCP as function of window size, RTT?
    - ignoring slow start
  - Let  $W$  be 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:  $.75 W/RTT$
- 

- ### 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
- 

## TCP Fairness

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

WPI

## Why is TCP fair?

Two competing sessions:

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

WPI

## 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  $R/2$  !

WPI

## 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"

WPI