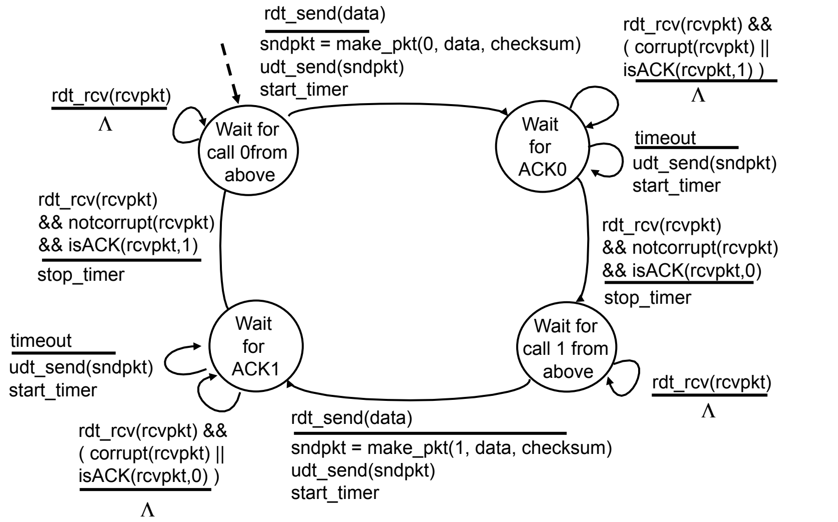
# rdt3.0: channels with errors *and* loss

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

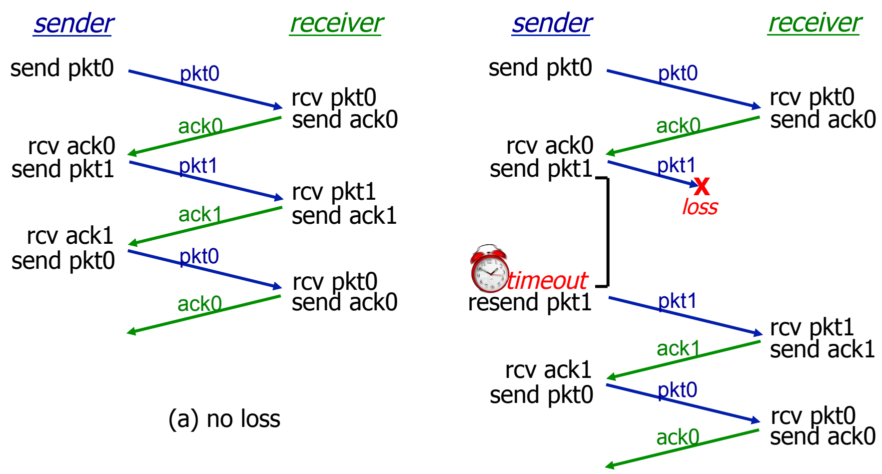
**approach:** sender waits "reasonable" amount of time for ACK
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer



# rdt3.0 sender



# rdt3.0 in action

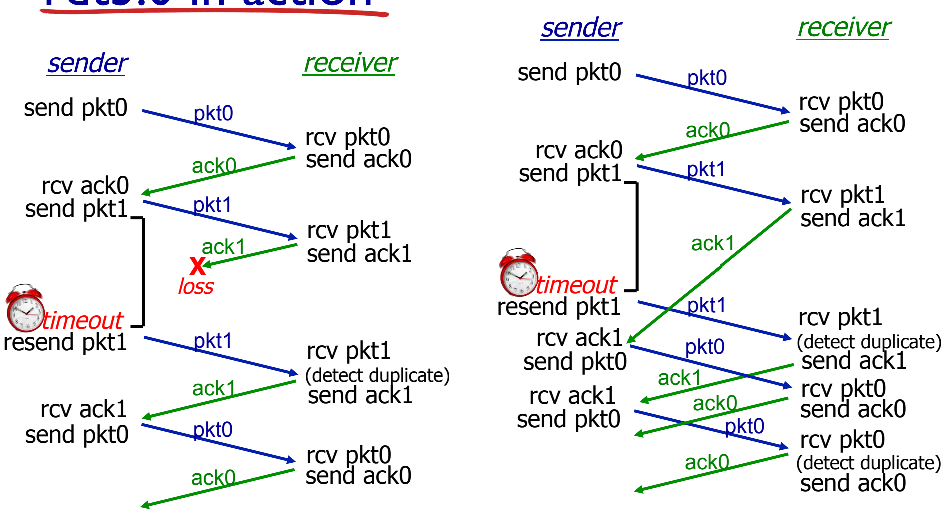



(a) no loss

(b) packet loss



# rdt3.0 in action




(c) ACK loss

(d) premature timeout/ delayed ACK

# Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

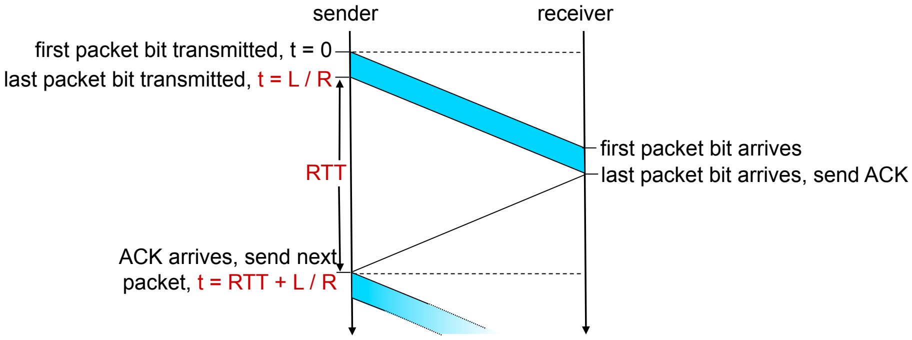
  - $U_{sender}$: *utilization* – fraction of time sender busy sending

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

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



# rdt3.0: stop-and-wait operation

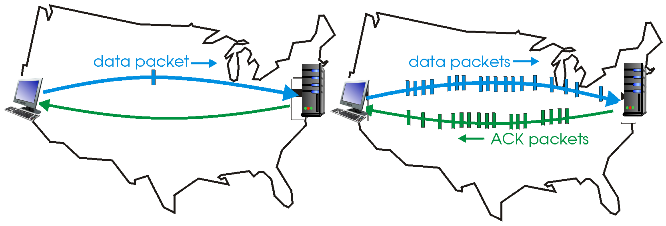



$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

# Pipelined protocols

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

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

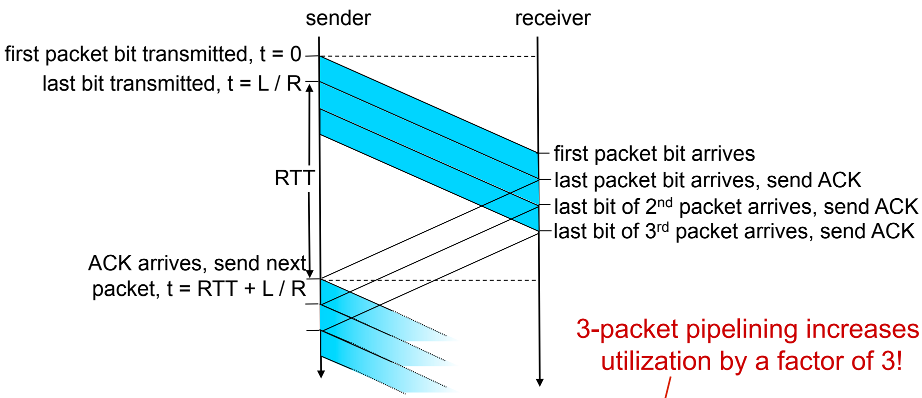



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

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



# Pipelining: increased utilization




3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

## 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



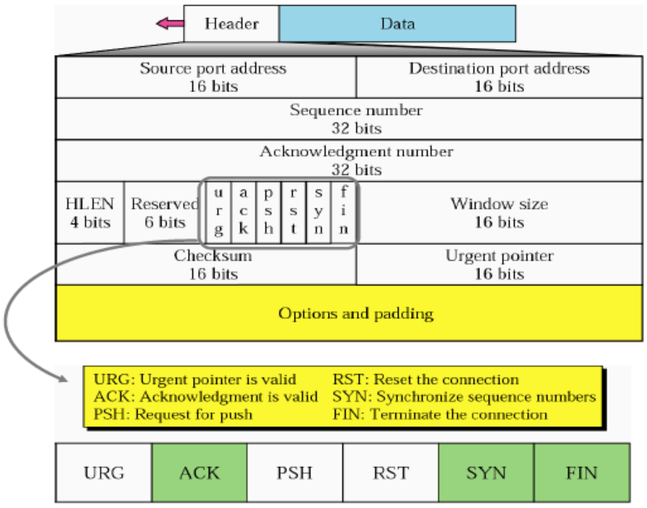
## TCP: Overview RFCs: 793,1122,1323, 2018, 2581

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

# TCP segment structure

**TCP Datagram** – **20- to 60- byte header + application data**

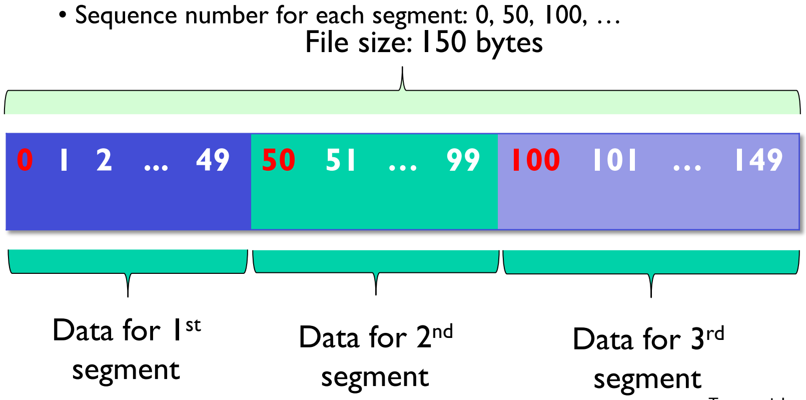
- **20-byte header if there are no “options”**






# TCP sequence number

- Sequence numbers:
  - 32-bit field
  - byte stream “number” of the first byte in the segment
- Example: file size=150 byte, max segment size=50 byte
  - Sequence number for each segment: 0, 50, 100, …



## TCP acknowledgment number

- **Acknowledgements:**
  - 32-bit field
  - Byte-steam number of next byte that host is expecting to receive from other side – cumulative ACKs
  - If the byte numbered "x" has been successfully received, "x+1" is the acknowledgment number
  - Pure acknowledgment = TCP segment without data – acknowledgment is said to be piggybacked
- Example of cumulative ACK

  Host A sent 1st segment containing 50 bytes to Host B

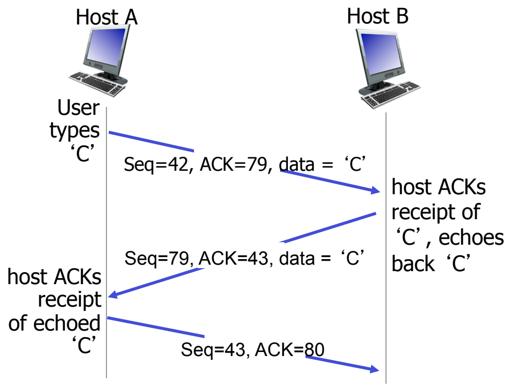
  Sequence number = 0 in Host A's segment to Host B

  If B receives the package correctly,

  Acknowledgment number = 50 in Host B's segment to Host A



## TCP seq. numbers, ACKs




simple telnet scenario

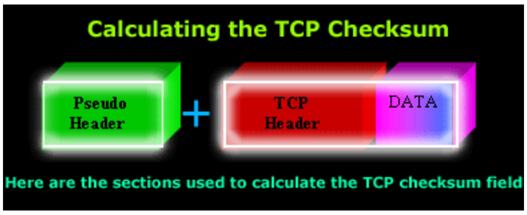
## TCP header length, reserved, window size

- **Header Length**
  - 4-bit field,
  - Represents the number of 4-byte words in the header
  - Header length 20-60 bytes → field value always 5-15
- **Reserved**
  - 6-bit field, reserved for future use
- **Window Size**
  - 16-bit field
  - Defines the number of bytes, beginning with sequence number indicated in the acknowledgment field that receiver is willing to accept
  - Used for flow control



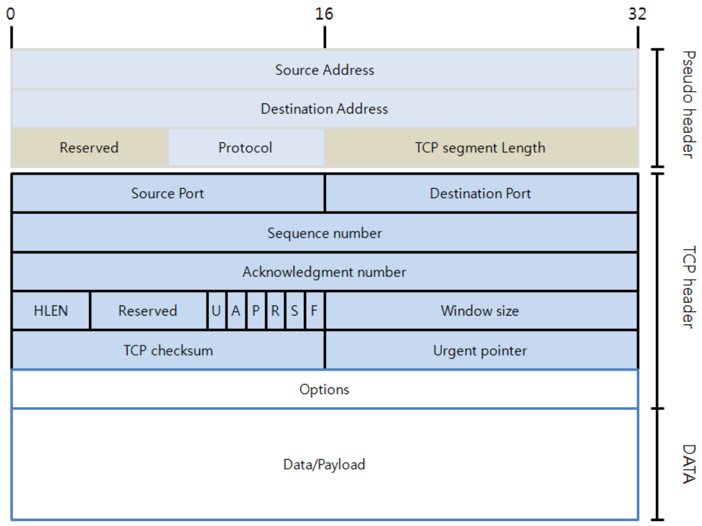
## TCP checksum

- **Checksum**
  - 16-bit field,
  - Used to detect errors over entire TCP datagram (header +data) + 96-bit pseudoheader conceptually prefixed to header at the time of calculation
    - Pseudoheader contains several field from IP header: source and destination IP addresses, protocol and segment length filed



# TCP segment example

- Pseudoheader added to the TCP datagram








# TCP pointer, options, padding

- Urgent pointer
  - 16-bit field,
  - Valid only if the urgent flag is set
  - Contains the sequence number of the last byte in a sequence of urgent data
- Options
  - There can be up to 40 bytes of optional information in the TPC header mostly related to flow/congestion control
- Padding
  - Ensures that TCP header ends and data begins on 32-bit boundary
  - Padding is composed of 0-s

## TCP control flags

| Flag | Description |
|---|---|
| URG | If this bit field is set, the receiving TCP should interpret the urgent pointer field. Used when a section of data should be read out by the receiving application quickly. The rest of the segment is processed normally. |
| ACK | If this bit field is set, the acknowledgement field is valid. |
| PSH | If this bit field is set, the receiver should deliver this segment to the receiving application as soon as possible, without waiting for *receive window* to get filled. |
| RST | If this bit is present, it signals the receiver that the sender is aborting the connection and all queued data and allocated buffers for the connection can be freely relinquished. |
| SYN | When present, this bit field signifies that sender is attempting to "synchronize" sequence numbers. This bit is used during the initial stages of connection establishment between a sender and receiver. |
| FIN | If set, this bit field tells the receiver that the sender has reached the end of its byte stream for the current TCP connection. |

## TCP round trip time, 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

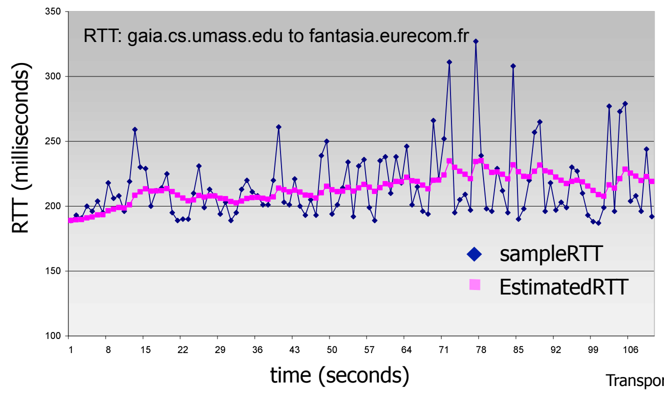
Q: how to estimate RTT?

- `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, timeout

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

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






# TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
                β*|SampleRTT-EstimatedRTT|

                (typically, β = 0.25)
```

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

"safety margin"

## 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
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

let's 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:*

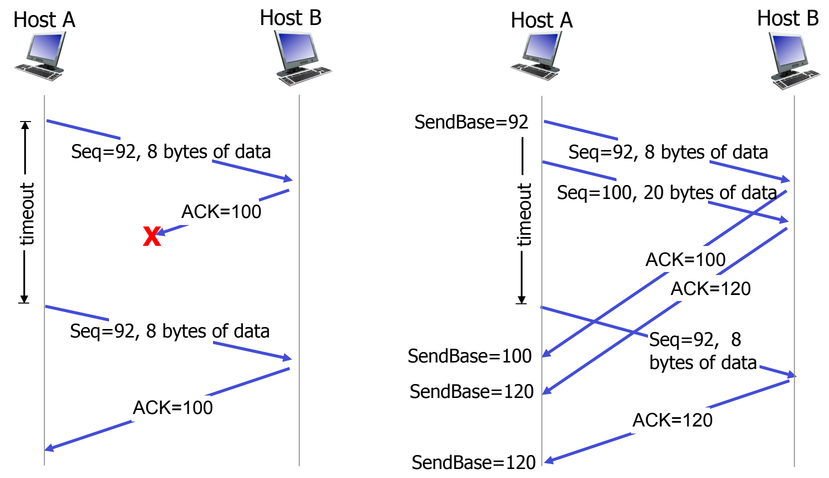
- retransmit segment that caused timeout
- restart timer

*ack rcvd:*

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments



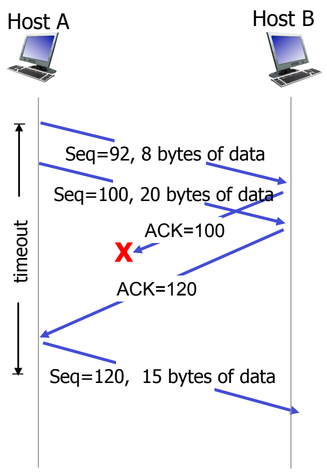
# TCP: retransmission scenarios




lost ACK scenario

premature timeout

# TCP: retransmission scenarios




cumulative ACK



# 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-expect seq. # . Gap detected | immediately send *duplicate ACK,* indicating seq. # of next expected byte |
| arrival of segment that partially or completely fills gap | immediate send ACK, provided that segment starts at lower end of gap |

# TCP fast retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

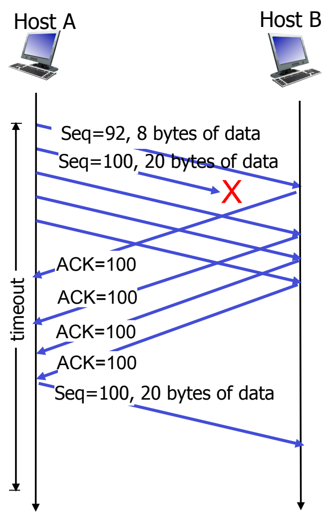
*TCP fast retransmit*

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout



# TCP fast retransmit




fast retransmit after sender receipt of triple duplicate ACK

# 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

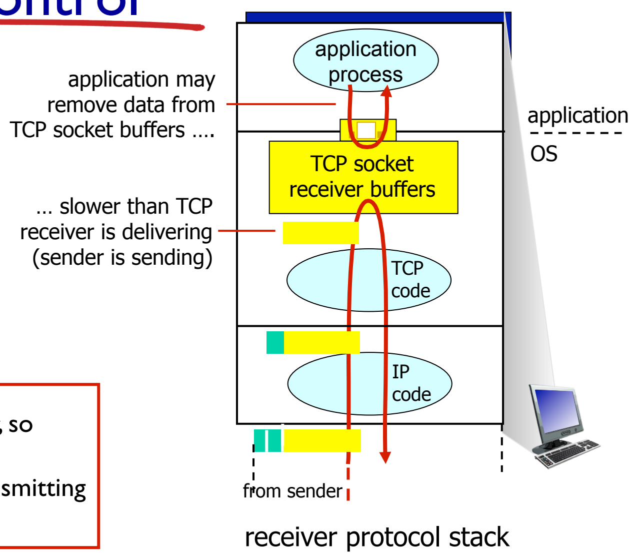
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control



# TCP flow control

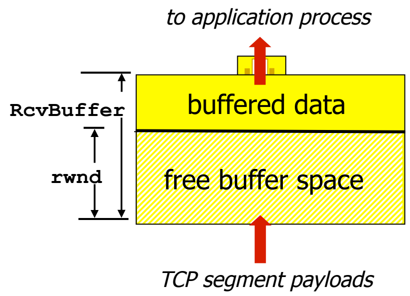



*flow control*

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

# TCP flow control

- receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- guarantees receive buffer will not overflow




*receiver-side buffering*



# 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

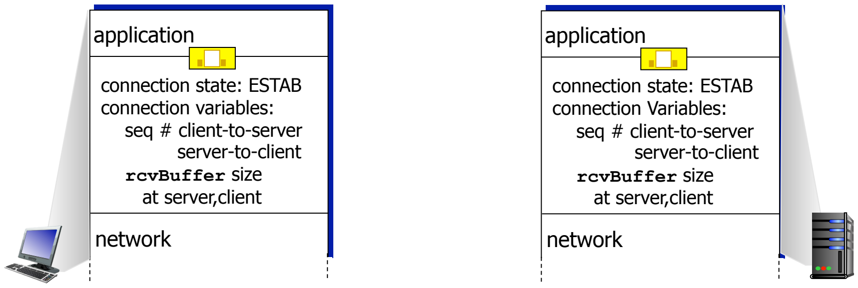
3.6 principles of congestion control

3.7 TCP congestion control

# Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters



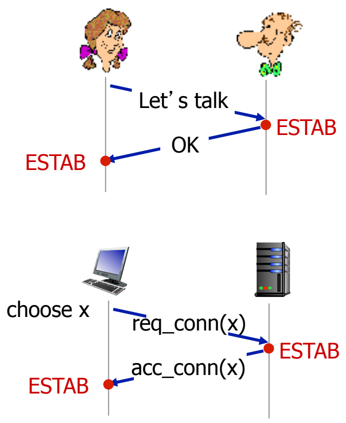

```
Socket clientSocket =
  newSocket("hostname","port
  number");
```

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



# Agreeing to establish a connection

2-way handshake:

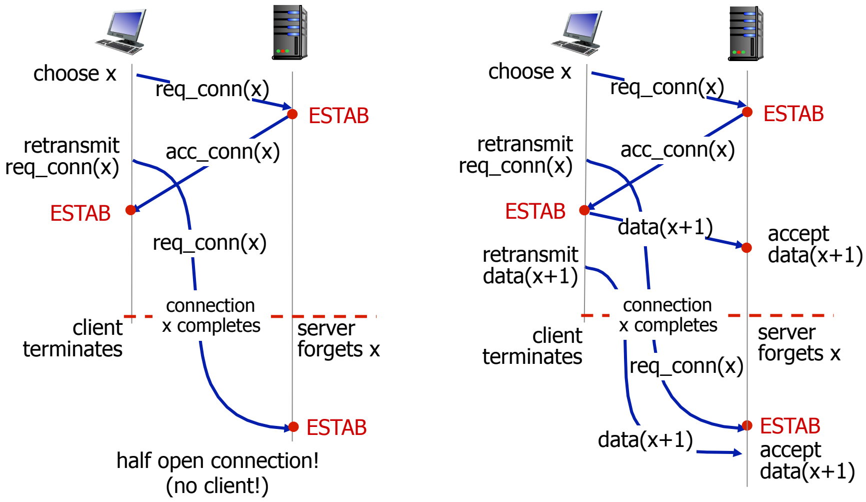



Q: will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

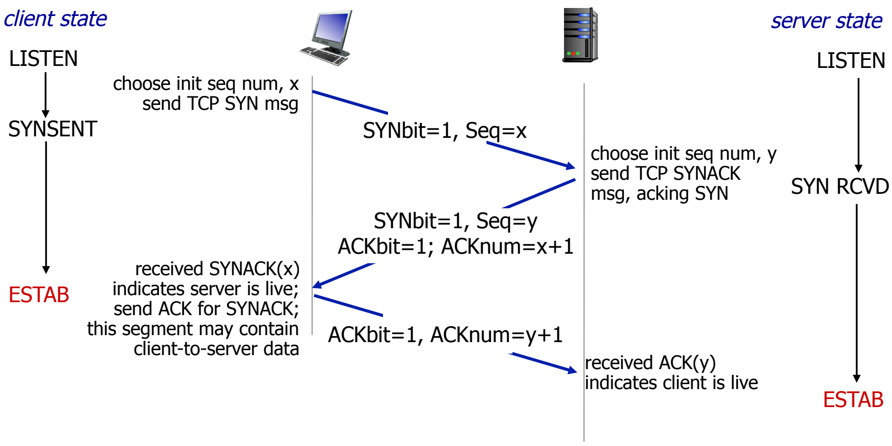
# Agreeing to establish a connection

2-way handshake failure scenarios:




Transport Layer 3-76

# TCP 3-way handshake



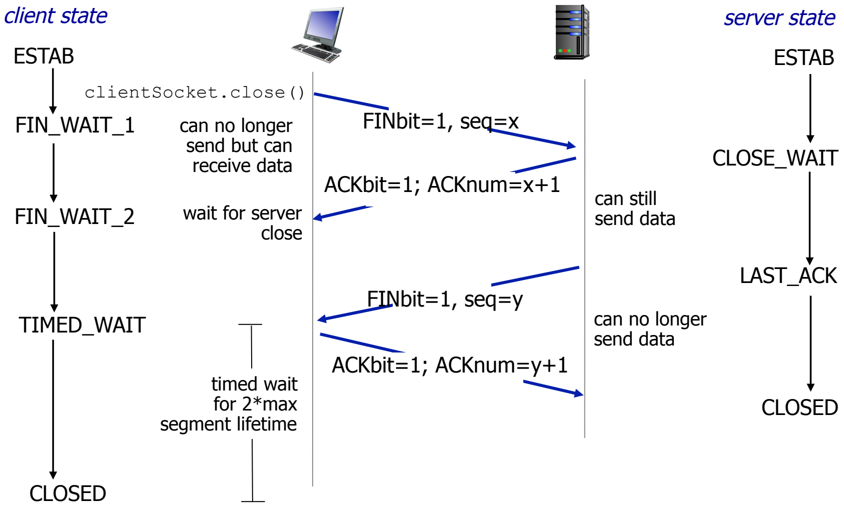

Transport Layer 3-77

# TCP: closing a connection

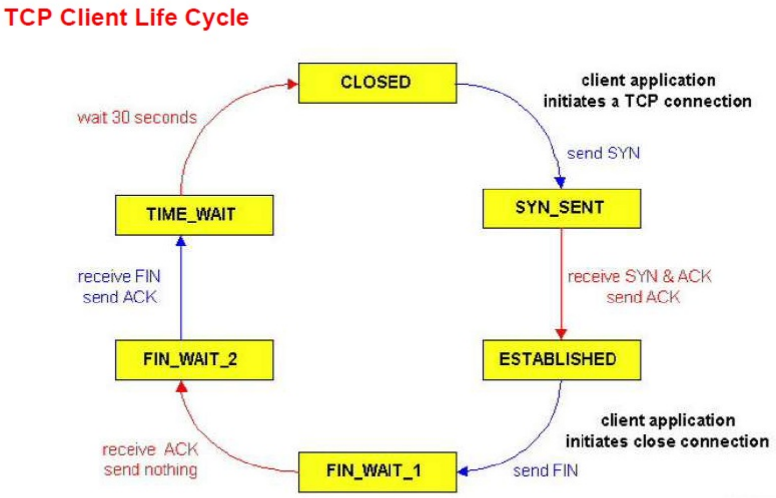
- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled



# TCP: closing a connection



# TCP client lifecycle






# TCP client lifecycle(1)

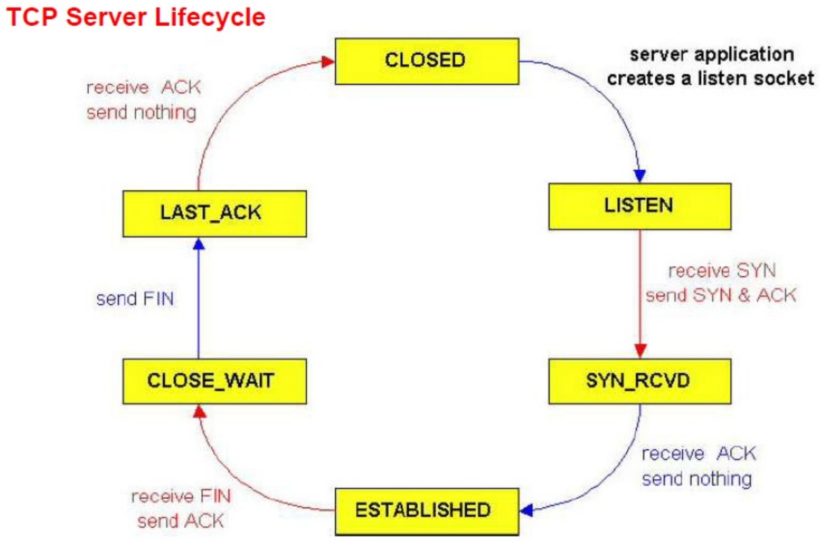
1. TCP client starts in CLOSED state
2. While in this state, TCP client can receive an *active* open request from client application program. It, then, sends a SYN segment to TCP server and goes to the SYN-SENT state
3. While SYN-SENT state, TCP client can receive a SYN +ACK segment from TCP server. It, then, sends an ACT to TCP server and goes to ESTABLISHED (date transfer) state. TPC client remains in this state as long as it sends and receives data.
4. While in ESTABLISHED state, TCP client can receive a close request from the client application program. It sends a FIN segment to TCP server and goes to FIN-WAIN-1 state

# TCP client lifecycle (2)

5. While in FIN-WAIT-1 state, TCP client waits to receive an ACK from TCP server. When the ACK is received, TCP client goes to FIN-WAIT-2 state. It does not send anything. Now the connection is closed in one direction.
6. TCP client remains in FIN-WAIT-2 state, waiting for TCP server to close the connection from its end. Once TCP client receivers a FIN segment from TCP server, it sends an ACK segment and goes to the TIME-WAIT state.
7. When in TIME-WAIT state, TCP client starts a timer and waits until the timer goes off. The TIME-WAIT timer is set twice the maximum segment lifetime(2MSL). The client remains in this state before totally closing to ensure that ACK segment it sent was received (if another FIN arrives from TCP server, ACK segment is retransmitted and the TIME-WAIT timer is restarted at 2MSL).



# TCP server lifecycle



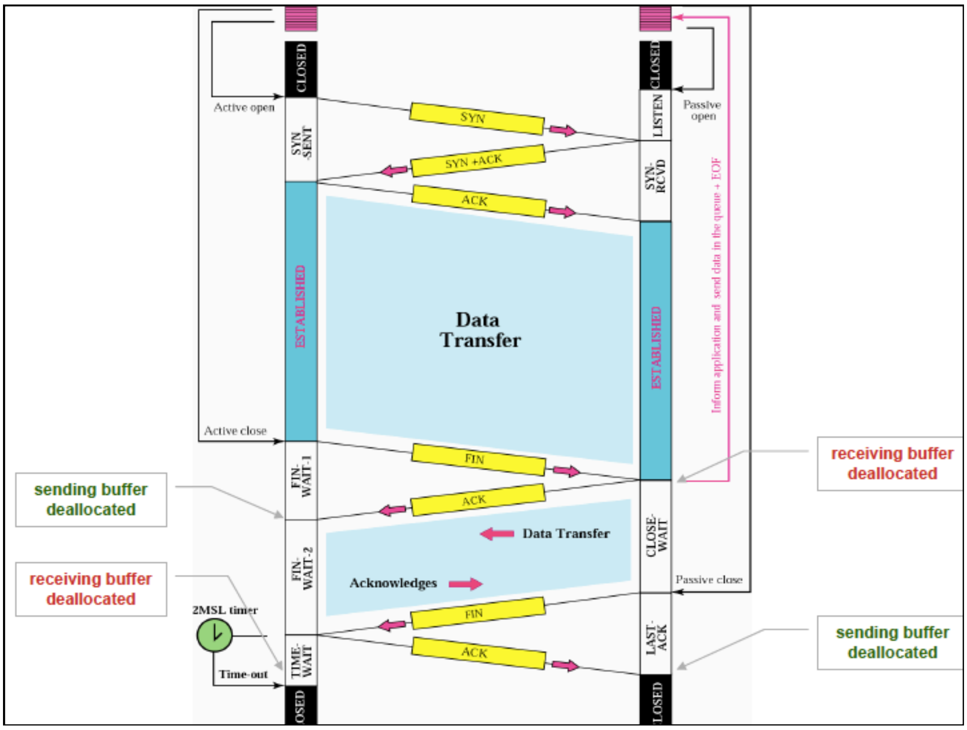
## TCP server lifecycle(1)

1. TCP server starts in CLOSED state
2. While in this state, TCP server can receive an *passive* open request from server application program. It, then, goes to the LISTEN state
3. While LISTEN state, TCP server can receive a SYN segment from TCP client. It sends a SYN+ACT segment to TCP client and then goes to SYN-RCVD state.
4. While in SYN-RCVD state, TCP server can receive an ACK segment from client TCP. It, then, goes to ESTABLISHED (data transfer) state. TCP client remains in this state as long as it sends and receives data.



## TCP server lifecycle(2)

5. While in ESTABLISTED state, TCP server can receive a FIN segment from TCP client, which means that client wants to close the connection. TCP server then sends an ACK segment to TCP client and goes to CLOSE-WAIT state.
6. While in CLOSE-WAIT state, TCP server waits until it receives a close request from its own server program/ applications. It then sends a FIN segment from TCP client and goes to LAST-ACK state.
7. When in LAST-ACK state, TCP server waits for the last ACK segment. It then goes to CLOSED state.




# 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

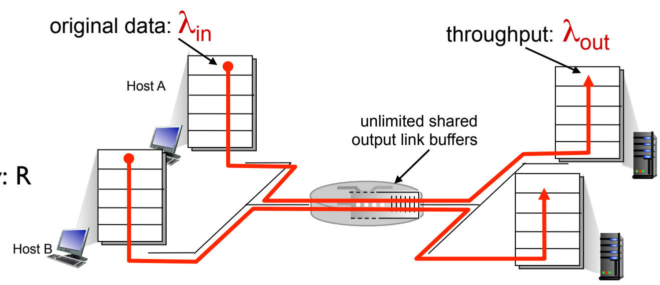
*congestion:*

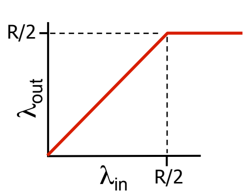
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

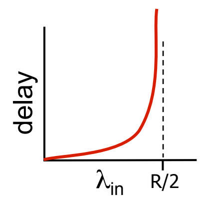


# Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission



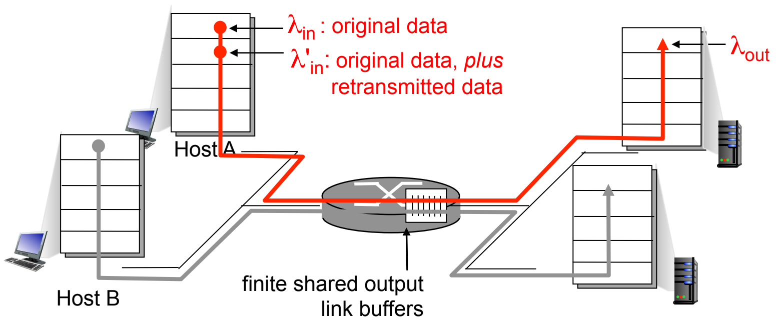




- maximum per-connection throughput: R/2
- large delays as arrival rate, $\lambda_{in}$, approaches capacity

## Causes/costs of congestion: scenario 2

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

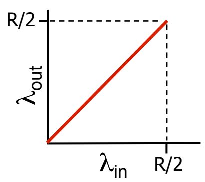


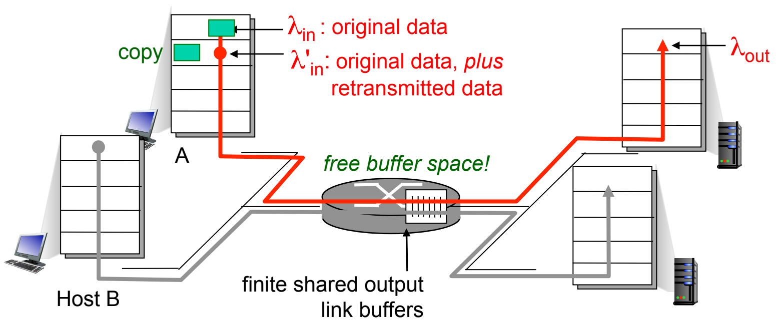



## Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- sender sends only when router buffers available

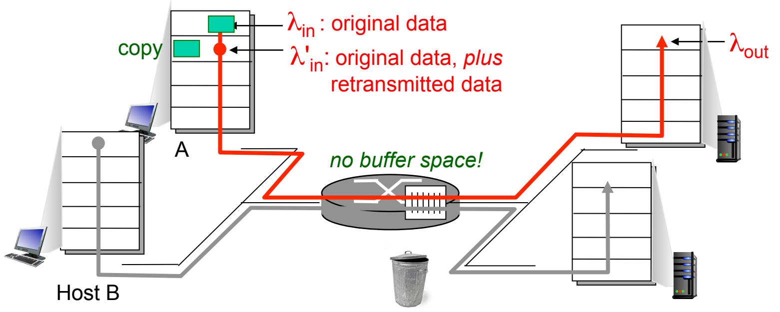


# Causes/costs of congestion: scenario 2

*Idealization: known loss*
packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost

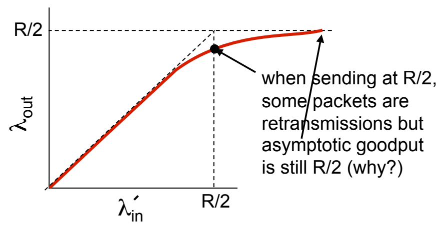


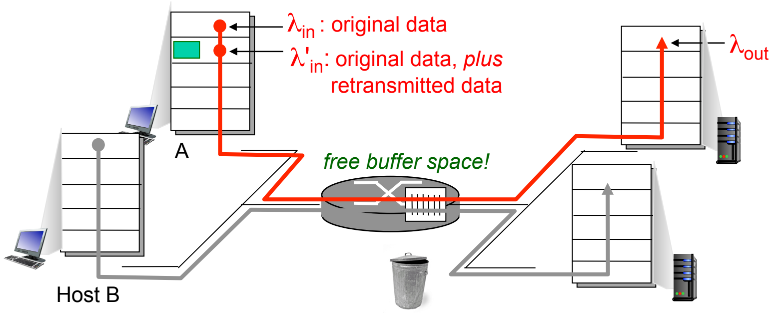



# Causes/costs of congestion: scenario 2

*Idealization: known loss*
packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost

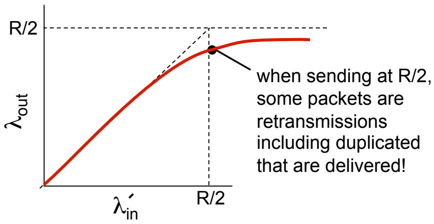


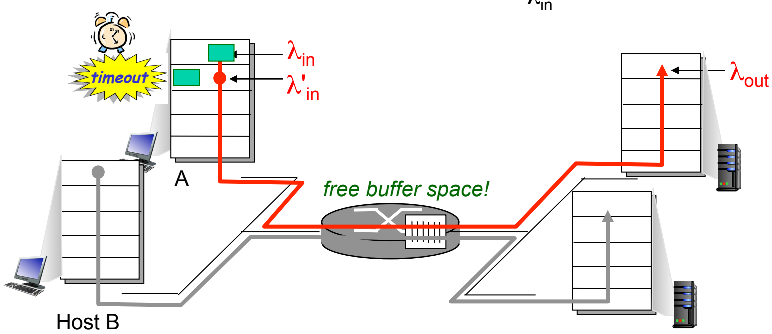



## Causes/costs of congestion: scenario 2

*Realistic: duplicates*

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered



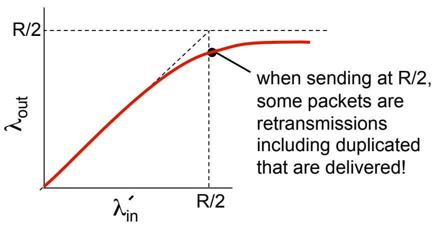






## Causes/costs of congestion: scenario 2

*Realistic: duplicates*

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered



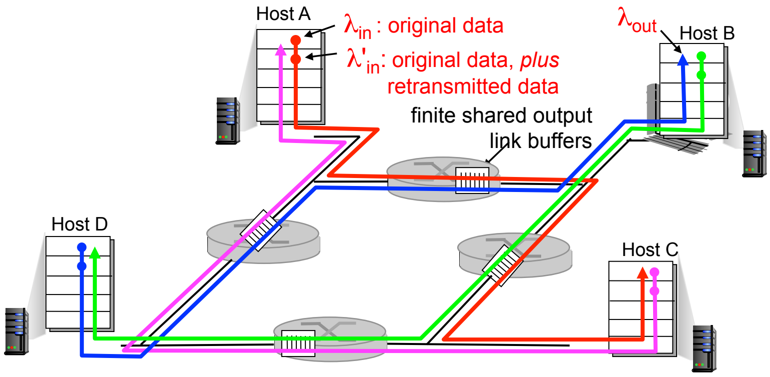

"costs" of congestion:
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

## Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

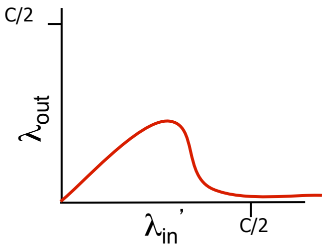
Q: what happens as $\lambda_{in}$ and $\lambda_{in}'$ increase ?

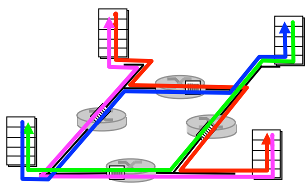
A: as red $\lambda_{in}'$ increases, all arriving blue pkts at upper queue are dropped, blue throughput $\rightarrow$ 0






## 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 for sender to send at



## Case study: ATM ABR congestion control

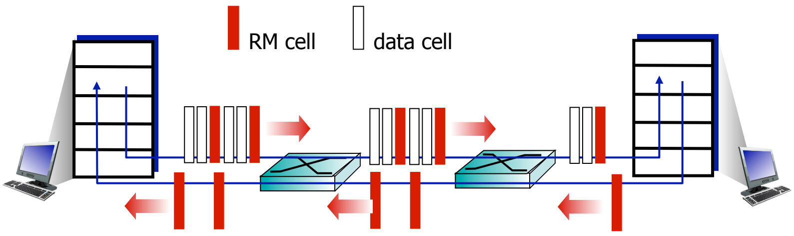
**ABR: available bit rate:**

- "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
  - senders' send rate thus max supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell



## 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

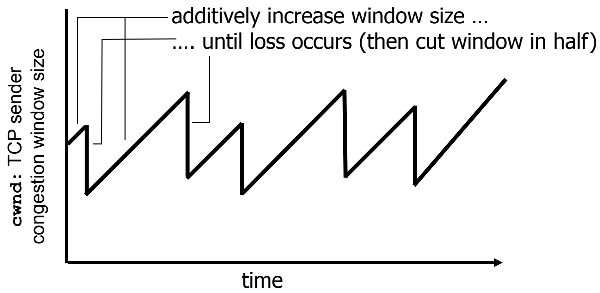
3.6 principles of congestion control

3.7 TCP congestion control

## TCP congestion control: additive increase multiplicative decrease

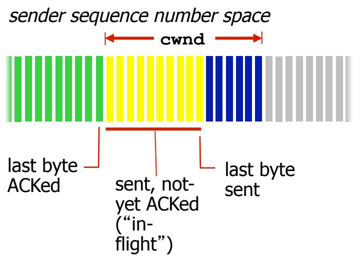
- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase congestion window (**cwnd**) by 1 MSS every RTT until loss detected
  - *multiplicative decrease:* cut **cwnd** in half after loss

AIMD saw tooth behavior: probing for bandwidth






## TCP Congestion Control: details




- sender limits transmission:

$$\texttt{LastByteSent-LastByteAcked} \leq \texttt{cwnd}$$

- **cwnd** is dynamic, function of perceived network congestion

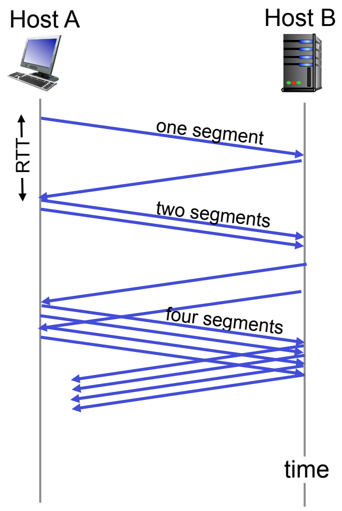
*TCP sending rate:*

- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

## TCP Slow Start

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





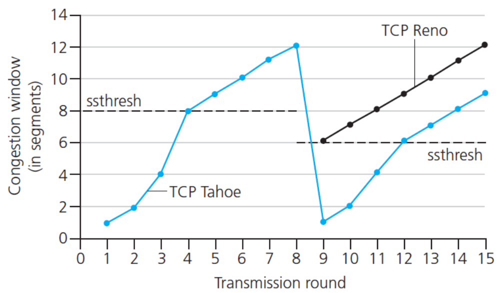

## TCP: detecting, reacting to loss

- loss indicated by timeout:
  - **cwnd** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - **cwnd** is cut in half window then grows linearly
- TCP Tahoe always sets **cwnd** to 1 (timeout or 3 duplicate acks)

# TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

A: when **cwnd** gets to 1/2 of its value before timeout.

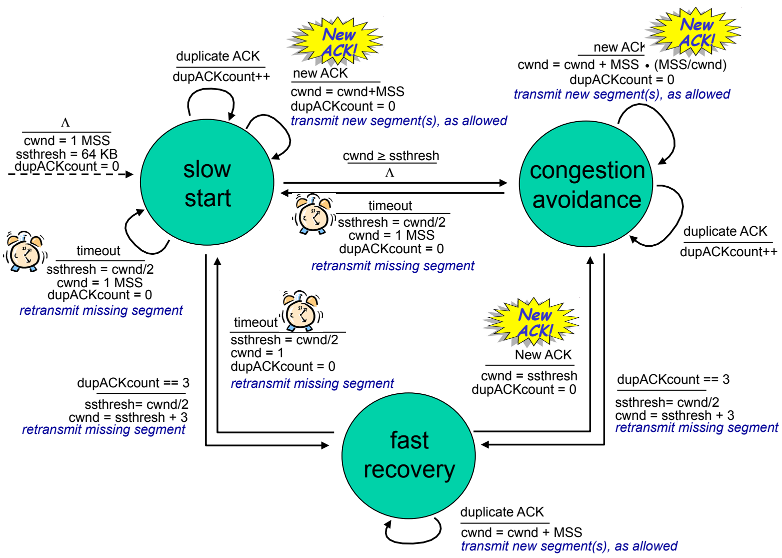



## Implementation:

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



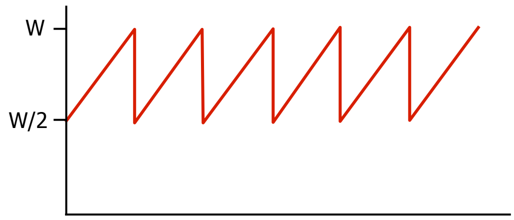
# Summary: TCP Congestion Control



# TCP throughput

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

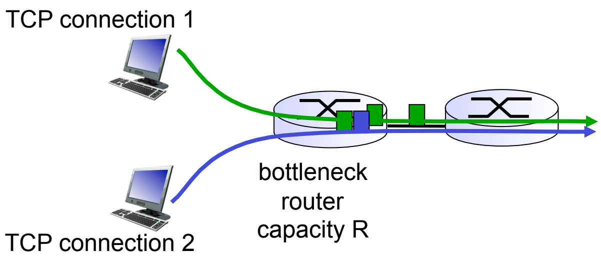
$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$






# TCP Fairness

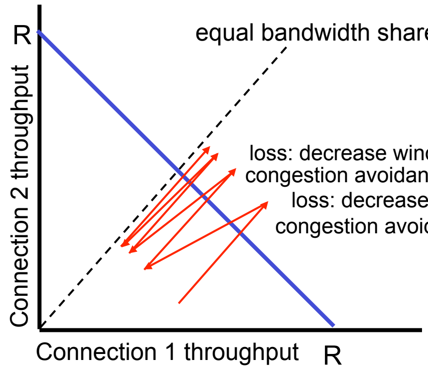
*fairness goal:* if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



# Why is TCP fair?

two competing sessions:

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






# Fairness (more)

*Fairness and UDP*

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

*Fairness, parallel TCP connections*

- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

# Chapter 3: summary

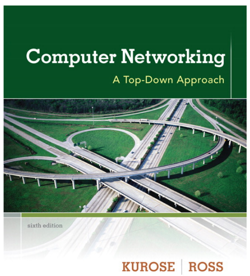
- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

next:

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



# A note on these slides




Part of PPT slides were adopted from Prof. Natalija Vlajic' early CSE3214 course and the rest were adopted from the book "Computer Networking: A Top Down Approach" 6th Edition by Jim Kurose and Keith Ross

*Computer Networking: A Top Down Approach*
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012