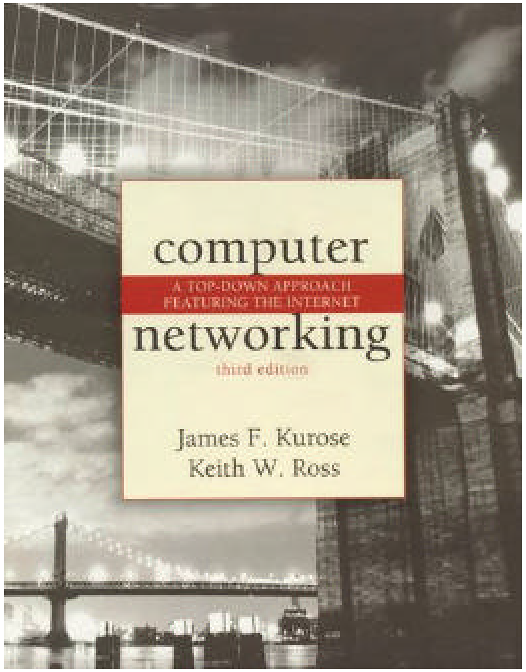
Chapter 3
Transport Layer



Computer Networking: A Top Down Approach Featuring the Internet, 3rd edition.
Jim Kurose, Keith Ross
Addison-Wesley, July 2004.

A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

❑ If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)

❑ If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Chapter 3: Transport Layer

Our goals:

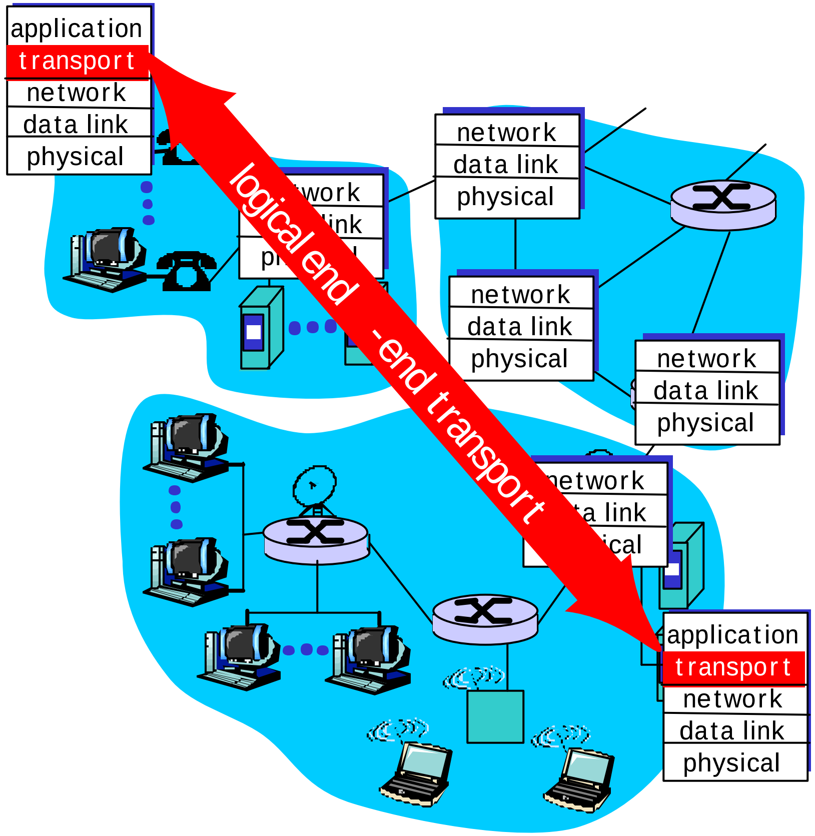
- understand principles behind transport layer services
- learn about transport layer protocols in the Internet

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
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps



Transport vs. network layer

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

Common Transport Layer Functions

- Demux to upper layer
  - Delivering data to correct application process
- Quality of service
  - Providing service guarantees in processing (buffers, process scheduling)
- Security
  - Authenticity, Privacy, Integrity for connection
- Connection setup
  - Providing a connection abstraction over a connectionless substrate
- Delivery semantics
  - Reliable or unreliable
  - Ordered or unordered
  - Unicast, multicast, anycast
- Flow control
  - Prevent overflow of receiver buffers
- Congestion control
  - Prevent overflow of network buffers
  - Avoid packet loss and packet delay

UDP and Transport Layer Functions

- Demux to upper layer
  - UDP port field
- Quality of service
  - none
- Security
  - None
- Connection setup
  - none
- Delivery semantics
  - Unordered, unicast or multicast
  - Unreliable, but data integrity provided by checksum
- Flow control
  - none
- Congestion control
  - none

TCP and Transport Layer Functions

- Demux to upper layer
  - TCP port field
- Quality of service
  - none
- Security
  - None, rely on TLS (SSL)
- Connection setup
  - 3-way handshake
- Delivery semantics
  - In-order, unicast
  - Data integrity provided via 32-bit checksum
- Flow control
  - Receiver advertised window
- Congestion control
  - Window-based

SCTP and Transport Layer Functions

- Demux to upper layer
  - SCTP port field
- Quality of service
  - none
- Security
  - Limited DoS protection via signed state cookie (SYN cookies)
  - Rely on TLS (SSL)
- Connection setup
  - 4-way handshake
- Delivery semantics
  - Optional ordering, unicast
  - Optional reliability, but data integrity provided via 32-bit CRC
- Flow control
  - Receiver advertised window
- Congestion control
  - Window-based

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

Multiplexing/demultiplexing

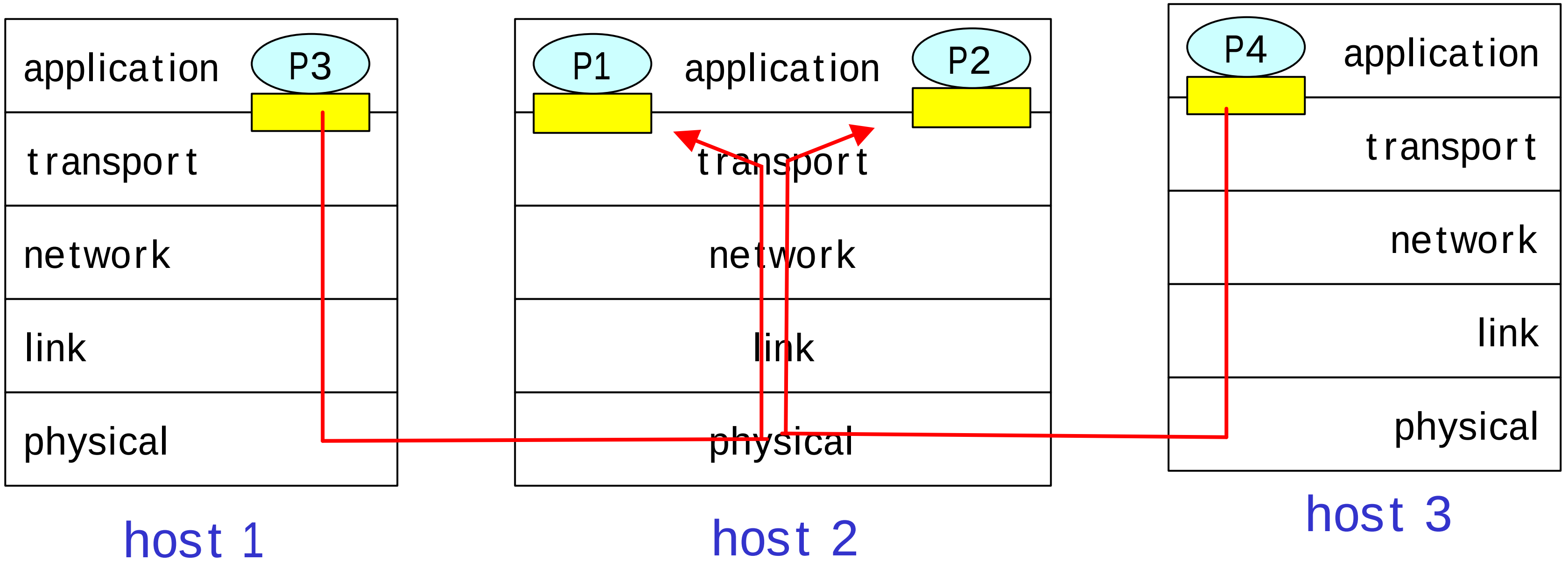
Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

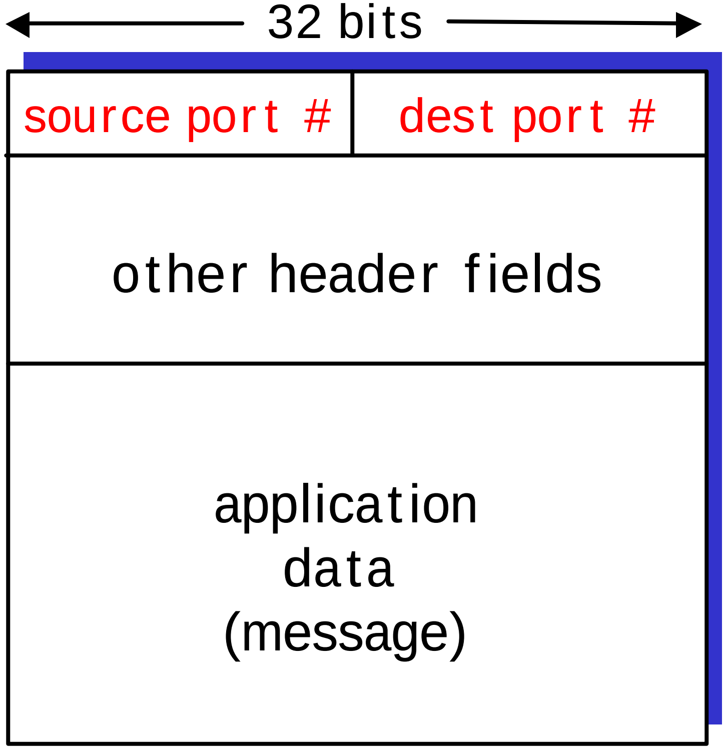
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

🟨 = socket ⬭ = process



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number
- host uses IP addresses & port numbers to direct segment to appropriate socket
  - source, dest port #s in each segment
  - recall: well-known port numbers for specific applications
  - Servers wait on well known ports (/etc/services)



TCP/UDP segment format

Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new
   DatagramSocket(99111);
DatagramSocket mySocket2 = new
   DatagramSocket(99222);
```

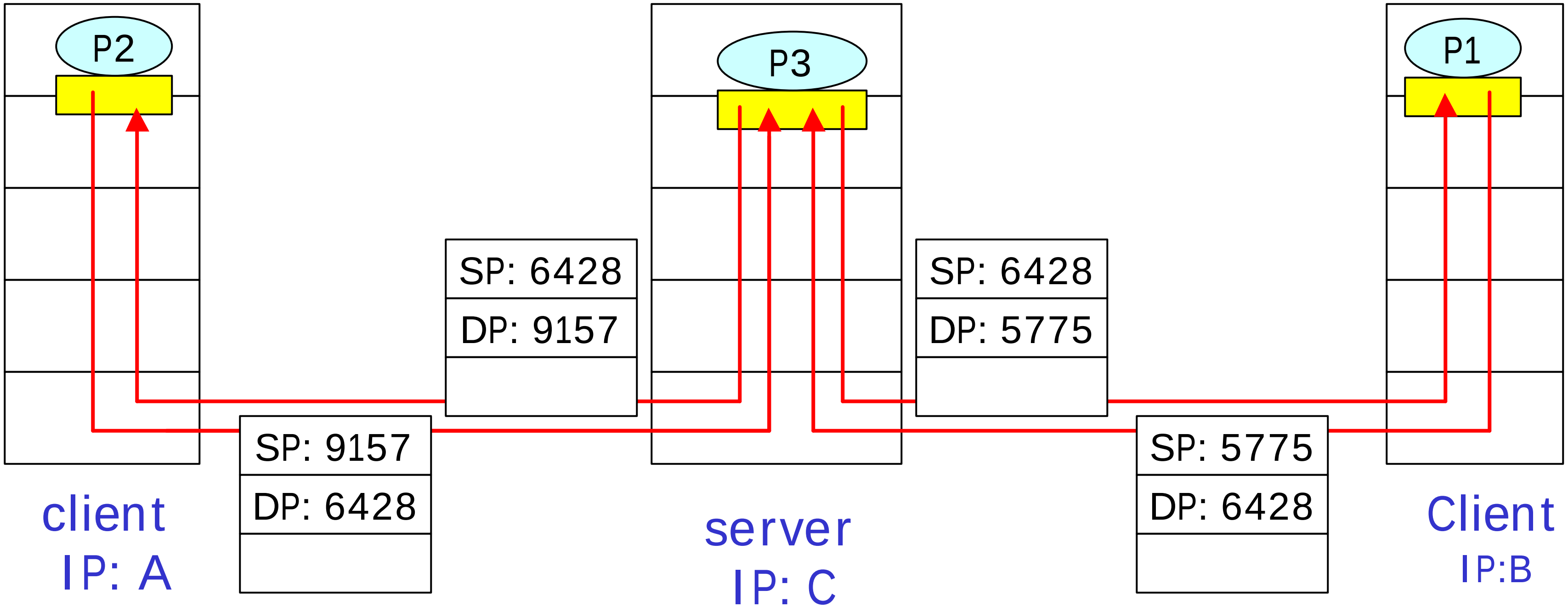
- UDP socket identified by two-tuple:

(dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



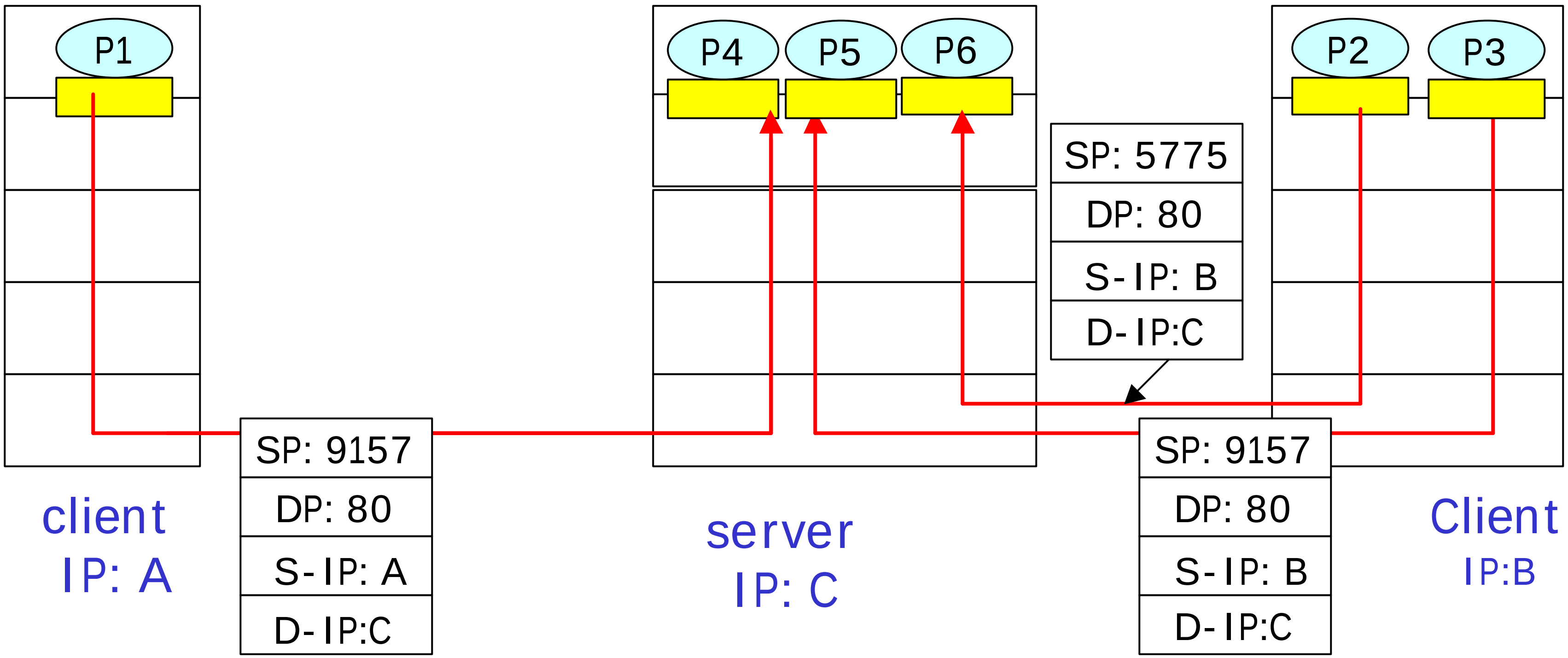
SP provides “return address”

Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host uses all four values to direct segment to appropriate socket

- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux (cont)



SP: 5775
DP: 80
S-IP: B
D-IP:C

client
IP: A

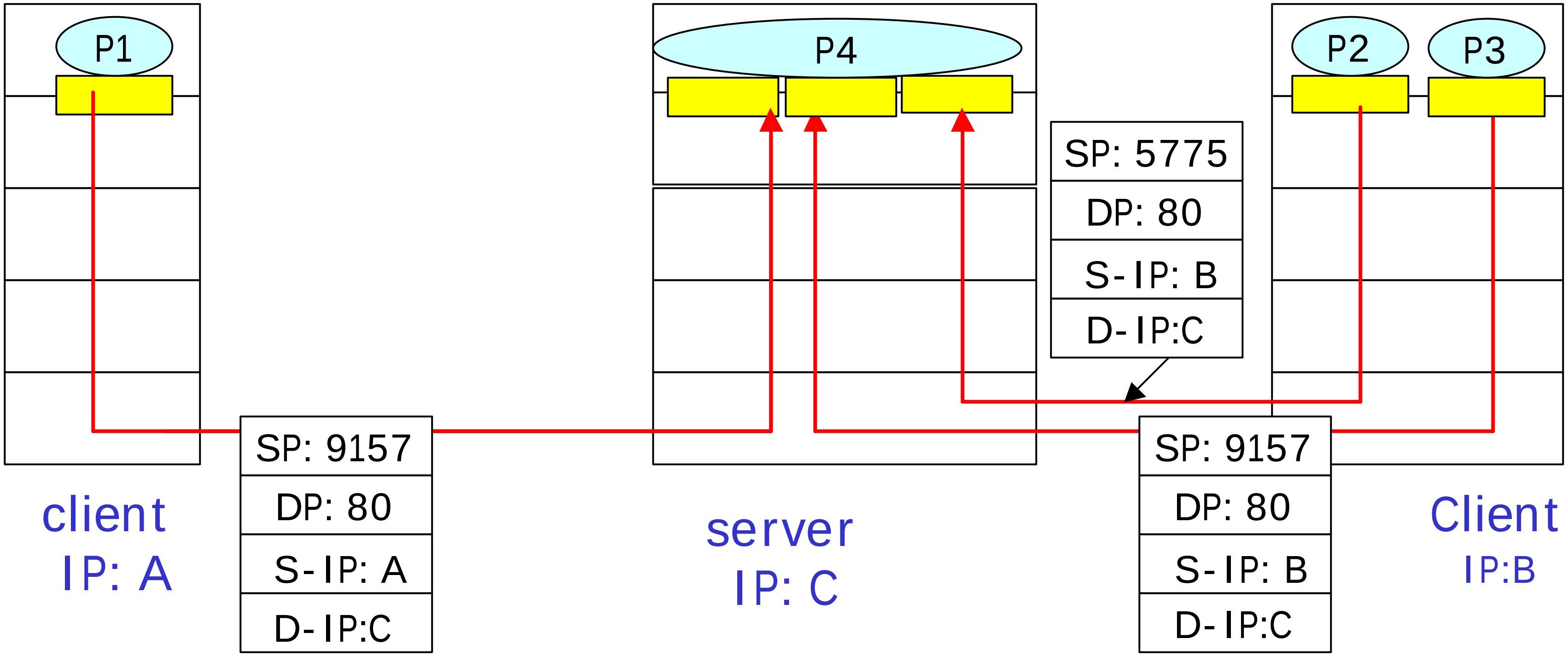
SP: 9157
DP: 80
S-IP: A
D-IP:C

server
IP: C

SP: 9157
DP: 80
S-IP: B
D-IP:C

Client
IP:B

Connection-oriented demux: Threaded Web Server



P1

P4

P2

P3

SP: 5775
DP: 80
S-IP: B
D-IP:C

SP: 9157
DP: 80
S-IP: A
D-IP:C

SP: 9157
DP: 80
S-IP: B
D-IP:C

client
IP: A

server
IP: C

Client
IP:B

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]

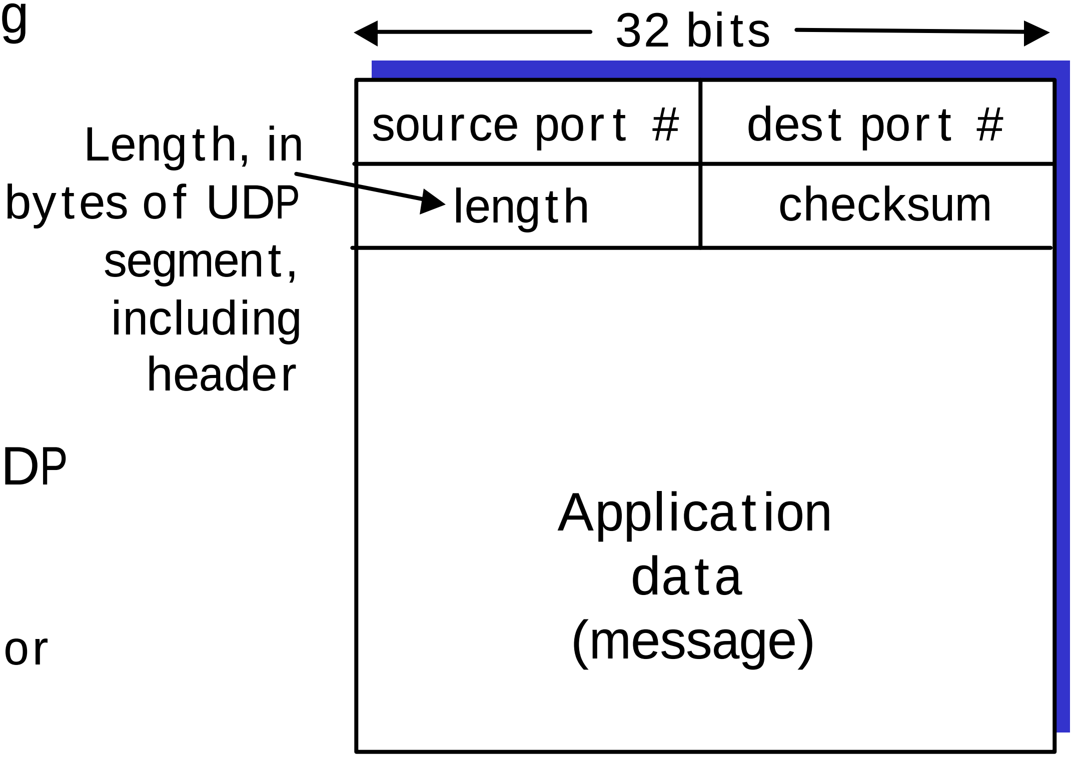
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

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 multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP
  - add reliability at application layer
  - application-specific error recovery!
  - Many applications re-implement reliability over UDP to bypass TCP
  - New transport protocols?



32 bits

source port #	dest port #
length	checksum

Application data (message)

Length, in bytes of UDP segment, including header

UDP segment format

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

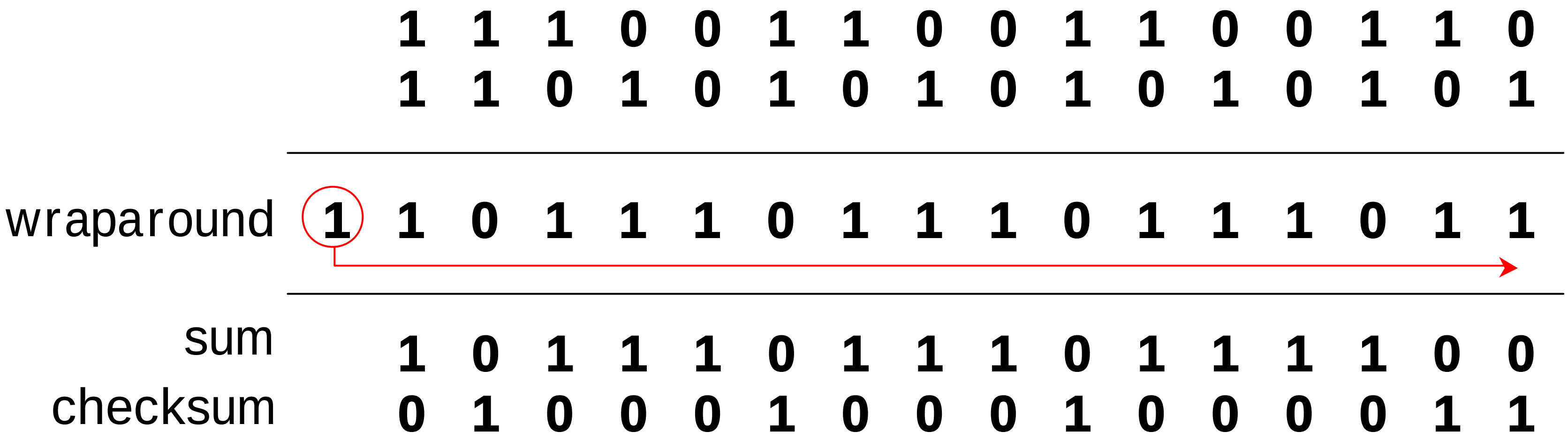
- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

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

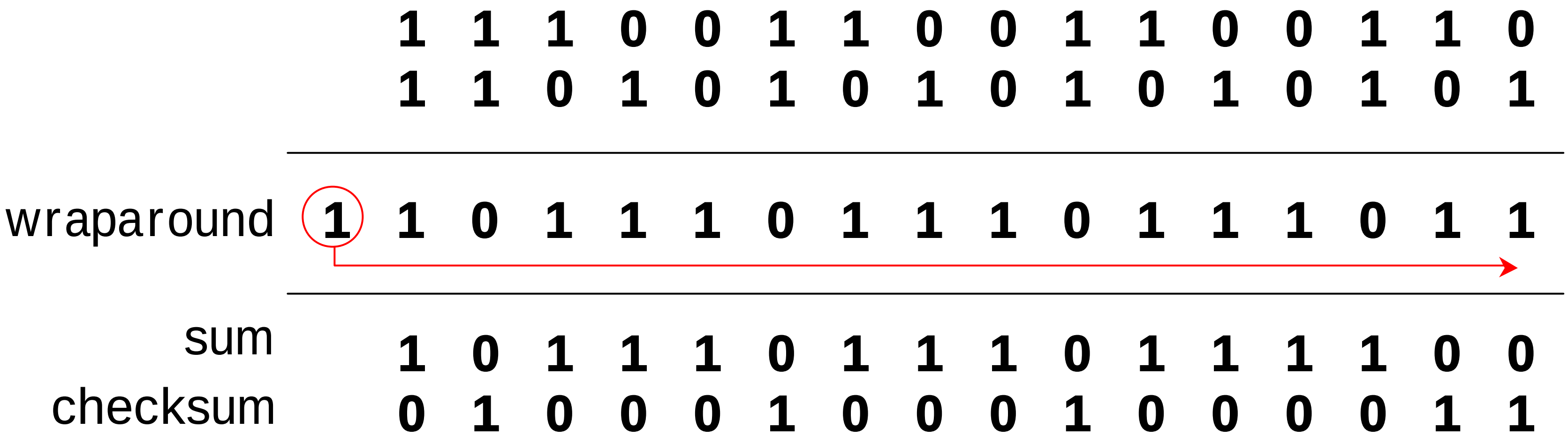
- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
  - 1s complement => convert 0 to 1 and 1 to 0
- Example: checksum for two 16-bit integers



		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

Internet Checksum Example

- Verification at receiver
  - Add all 16-bit words and checksum together
  - If no errors, sum will be all 1s



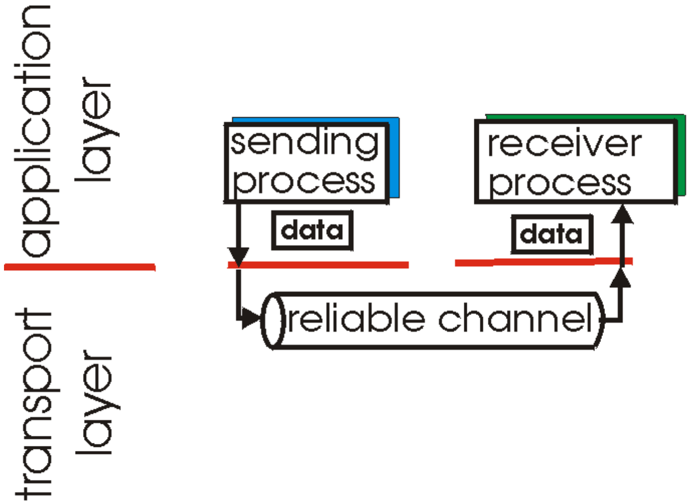
```
              1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
              1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
           -----------------------------------
wraparound  1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
           -----------------------------------
       sum    1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
  checksum    0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

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!

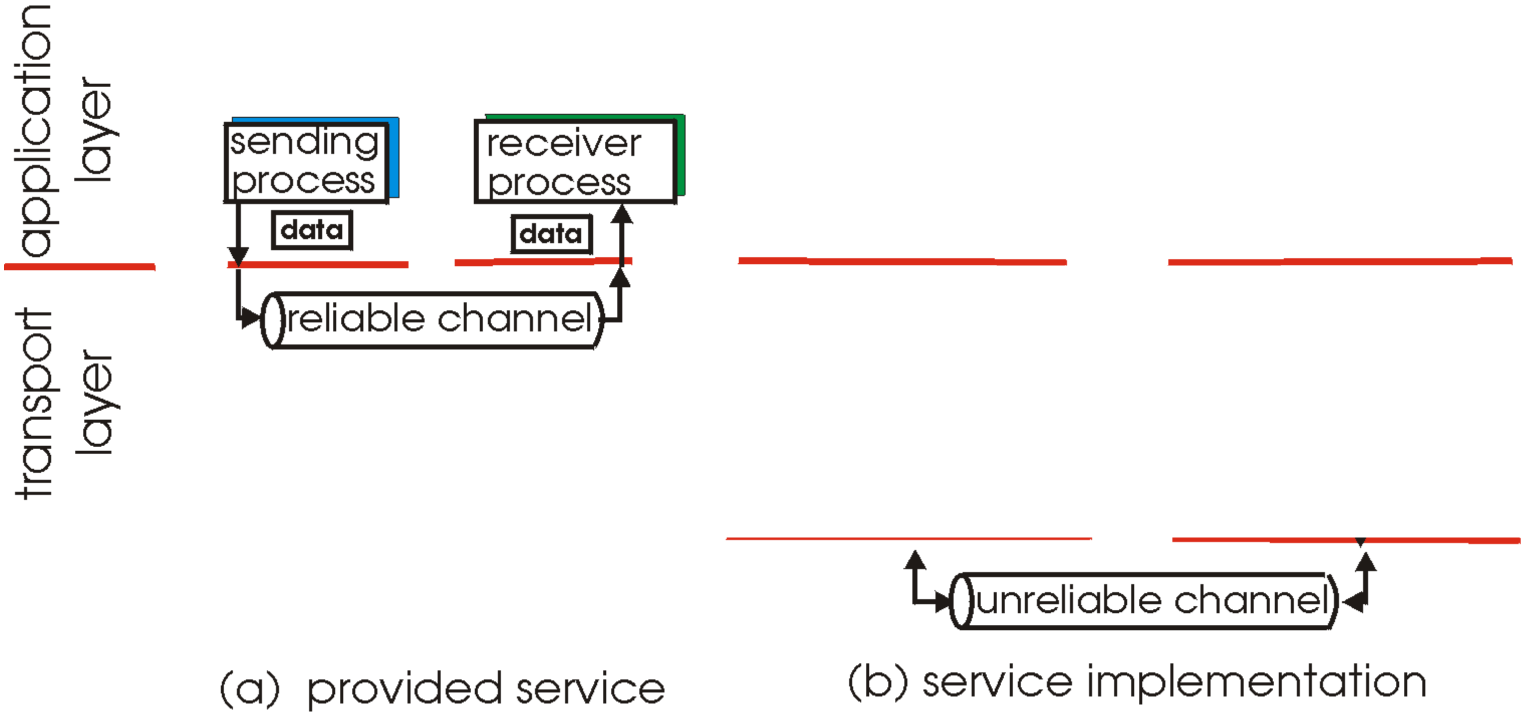


(a) provided service

- 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!



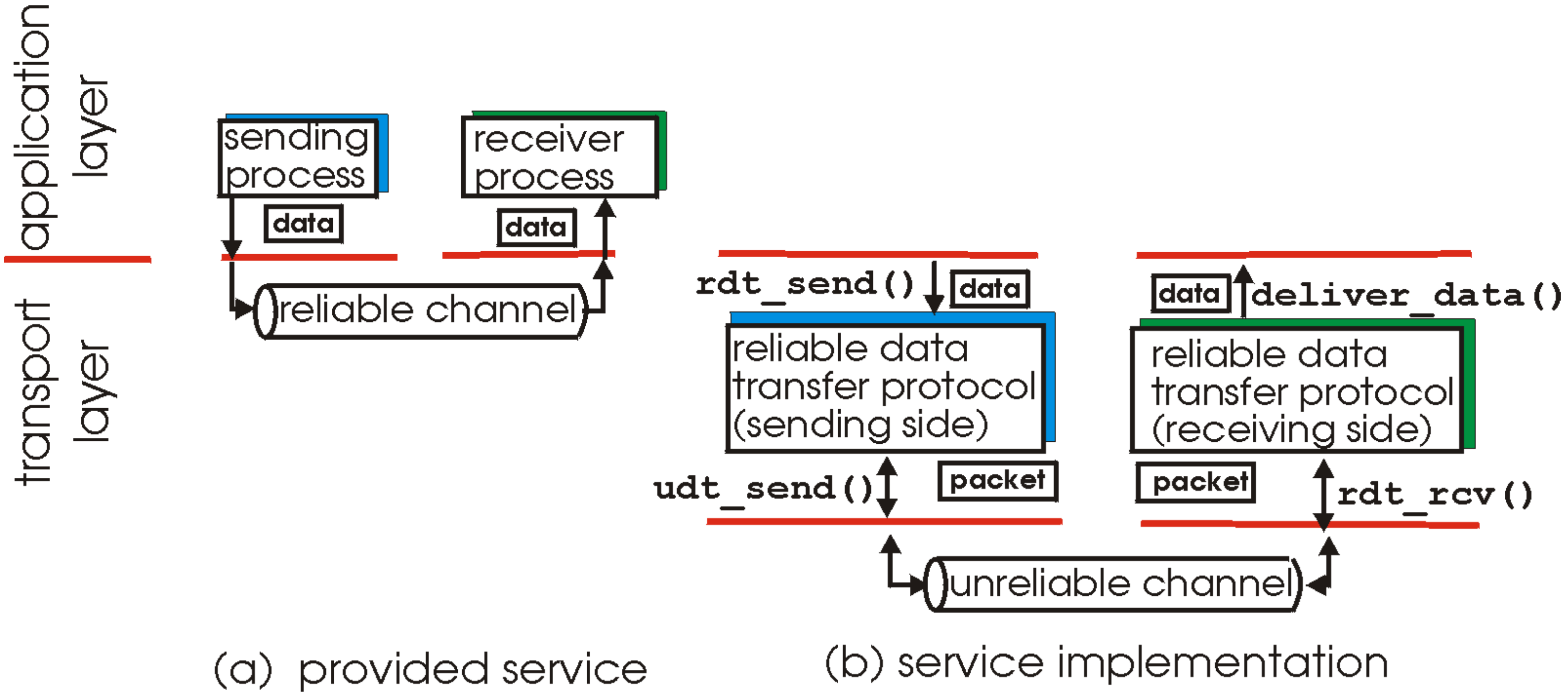
(a) provided service

(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!

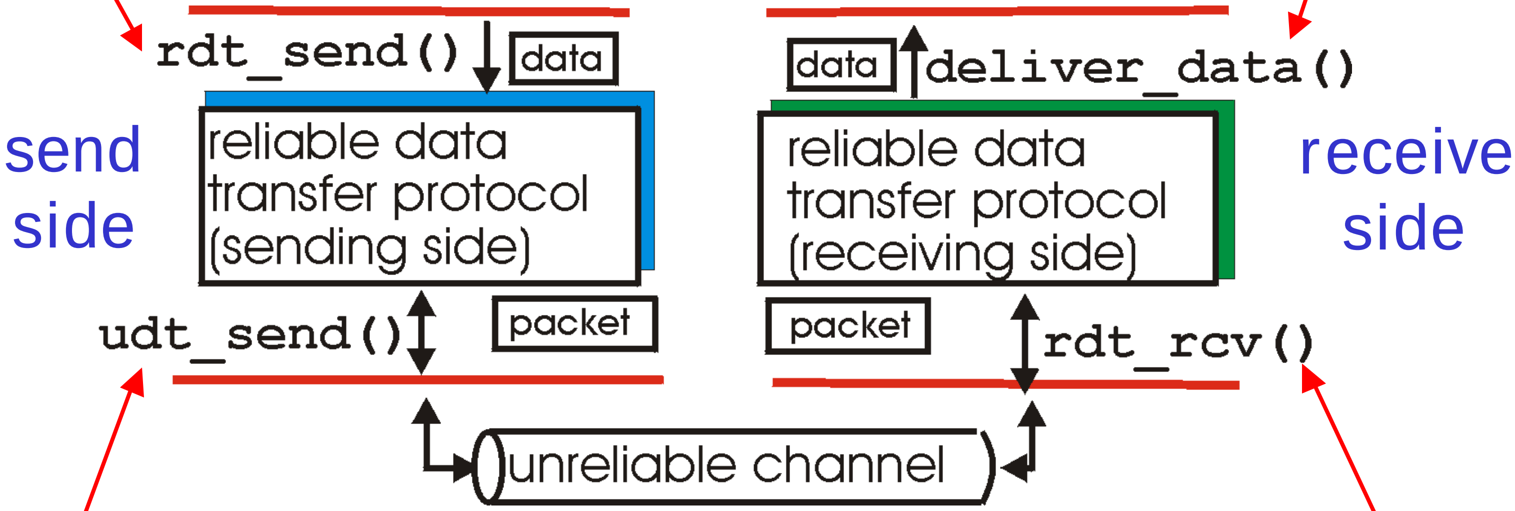


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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 basics

- Error detection, correction
- Retransmission
  - For lost or corrupted packets
- Duplicate detection
  - Spurious retransmissions identified
- Connection integrity
  - Bogus packets not included

rdt3.0 state machine

- See textbook and extra slides for issues in developing protocols and state machines for reliable data transfer
- Highlights
  - Sequence numbers (duplicate detection)
  - Acknowledgments (error and loss detection)
    - Positive or negative acks
    - Cumulative or selective acks
    - Rdt3.0: Cumulative positive acknowledgements
  - Checksum (error detection)
  - Retransmission via timer (loss recovery)
  - Problem: Stop-and-wait operation
    - Send one packet
    - Wait for ACK before sending next packet

Performance of Stop-and-Wait

- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:
- Assume no errors or loss

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

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

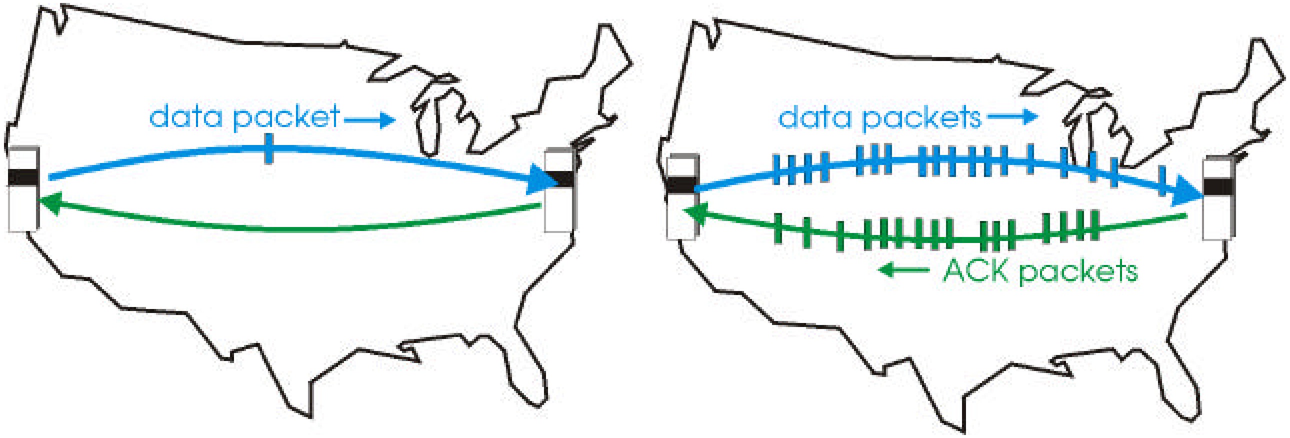
$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

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

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

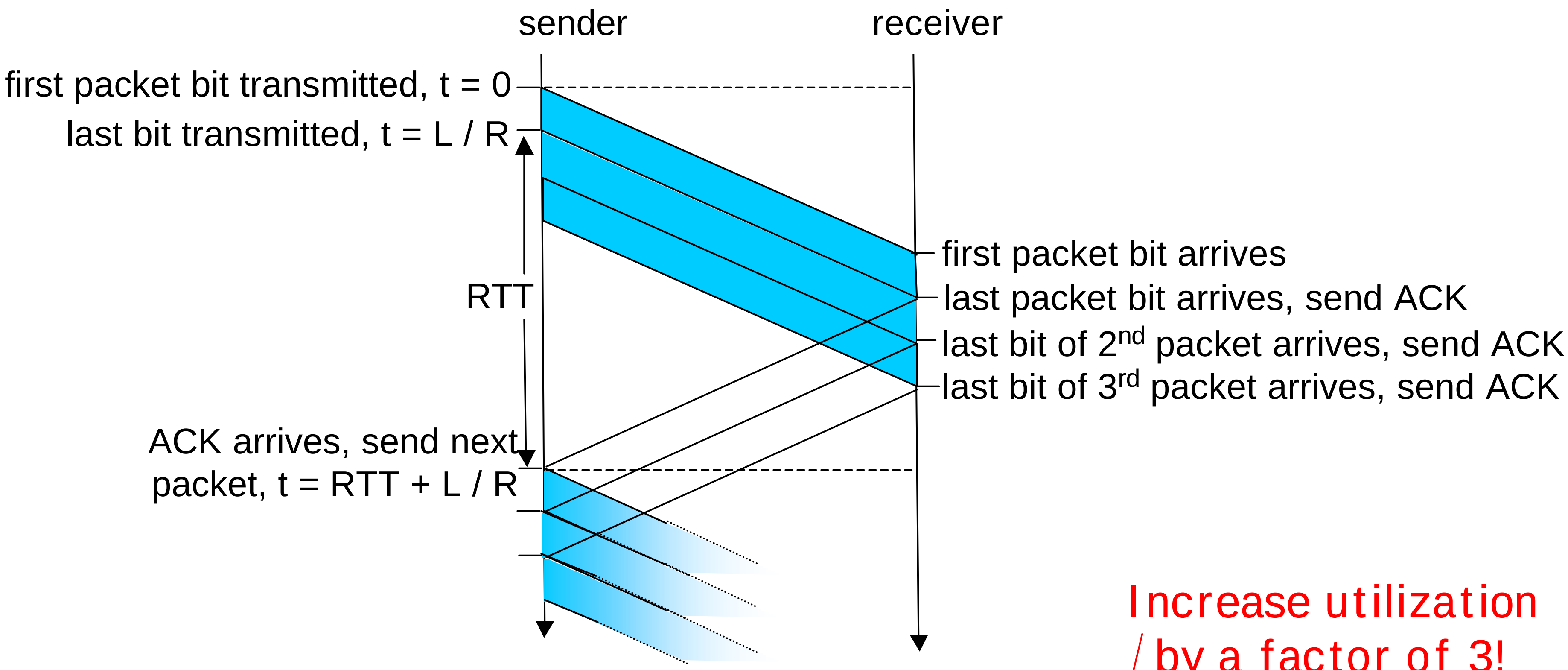


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

Pipelining: increased utilization


sender
receiver
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 2nd packet arrives, send ACK
last bit of 3rd packet arrives, send ACK
ACK arrives, send next packet, t = RTT + L / R

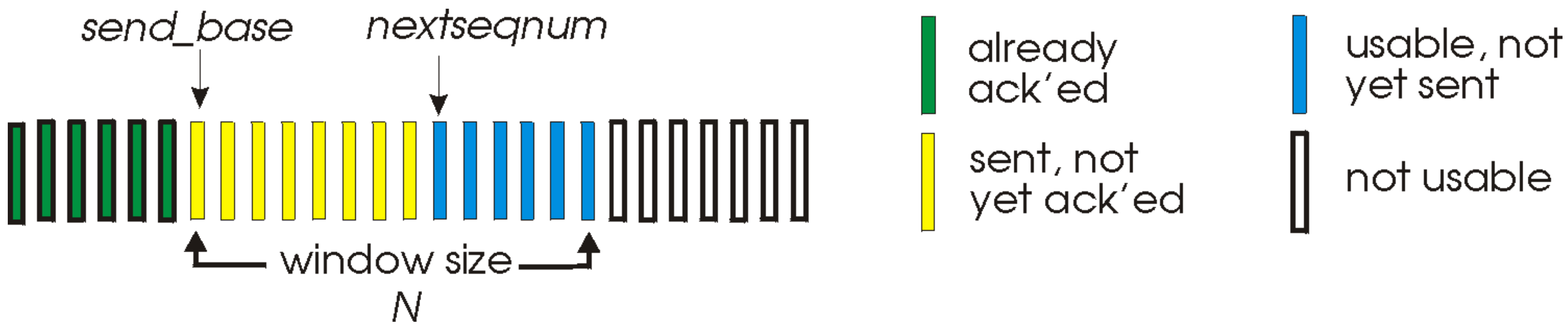
Increase utilization by a factor of 3!

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

Go-Back-N

Sender:

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

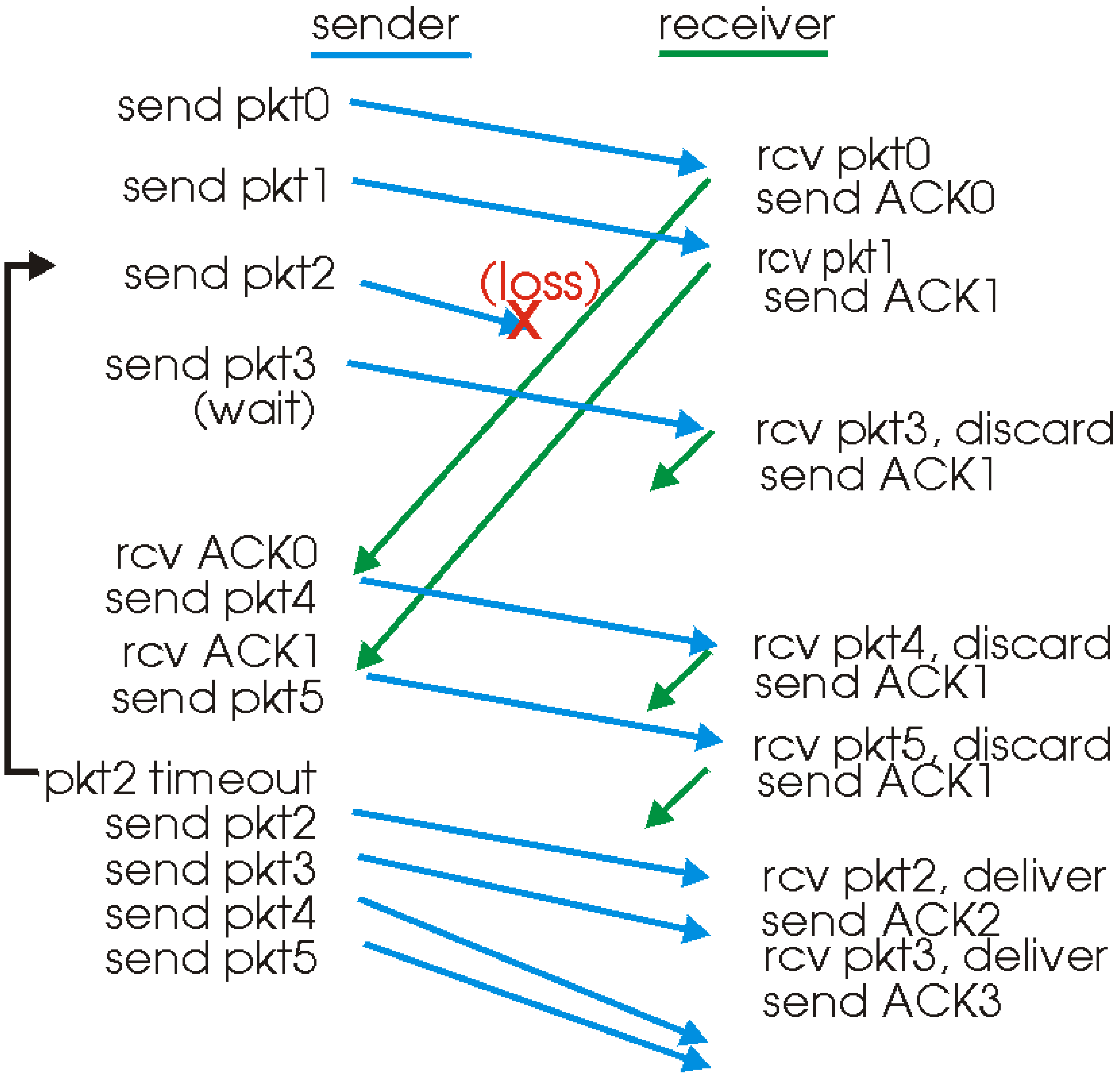


- Receiver sends cumulative ACK
  - i.e. Highest in-order sequence number received
  - may receive duplicate ACKs on loss or out-of-order delivery(see receiver)
- timer for each in-flight pkt
  - *timeout(n):* if no ACK received for n within timeout, retransmit pkt n and all higher seq # pkts in window

GBN: receiver

- Receiver simple
- 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 #

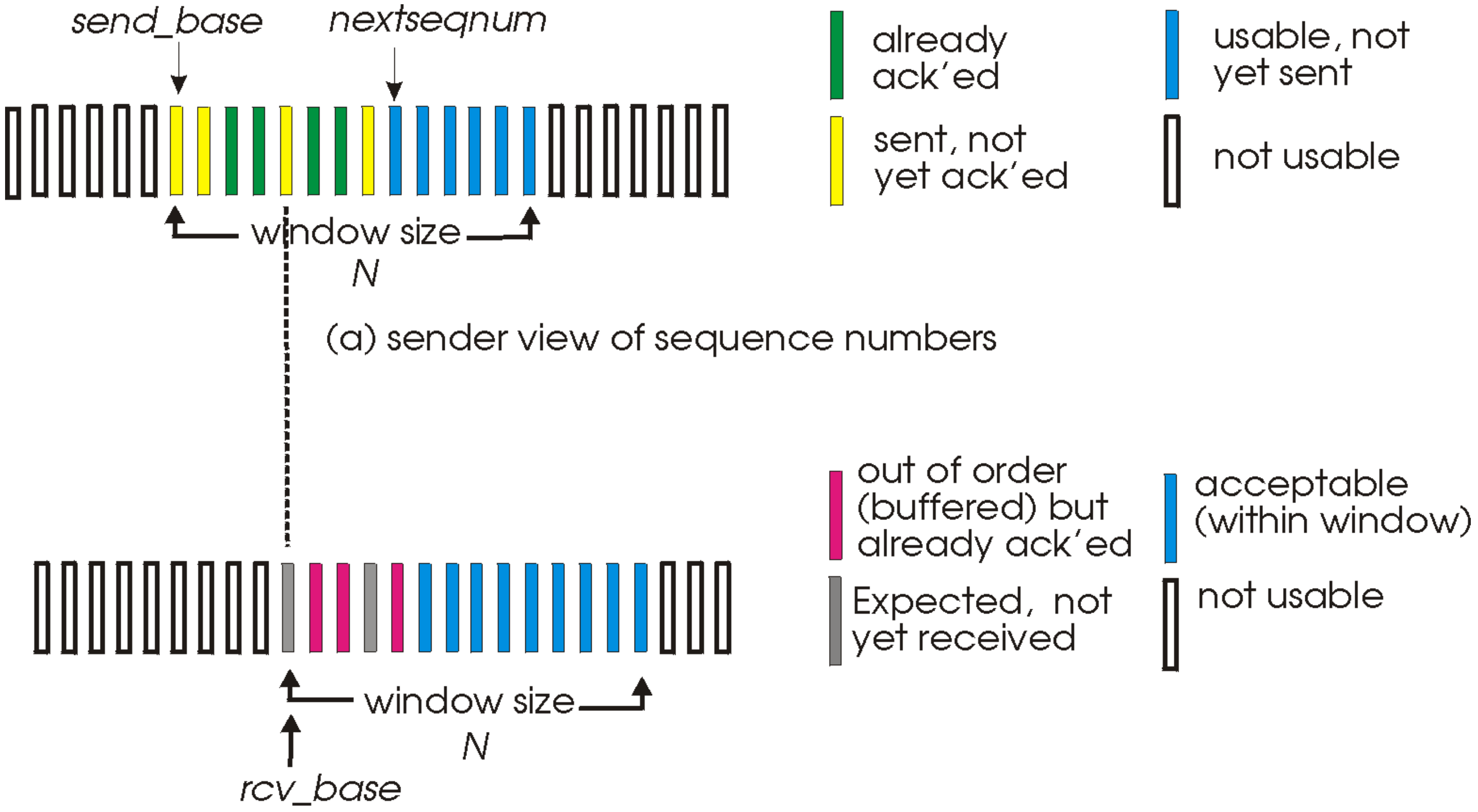
GBN in action



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



(a) sender view of sequence numbers

(b) receiver view of sequence numbers

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]

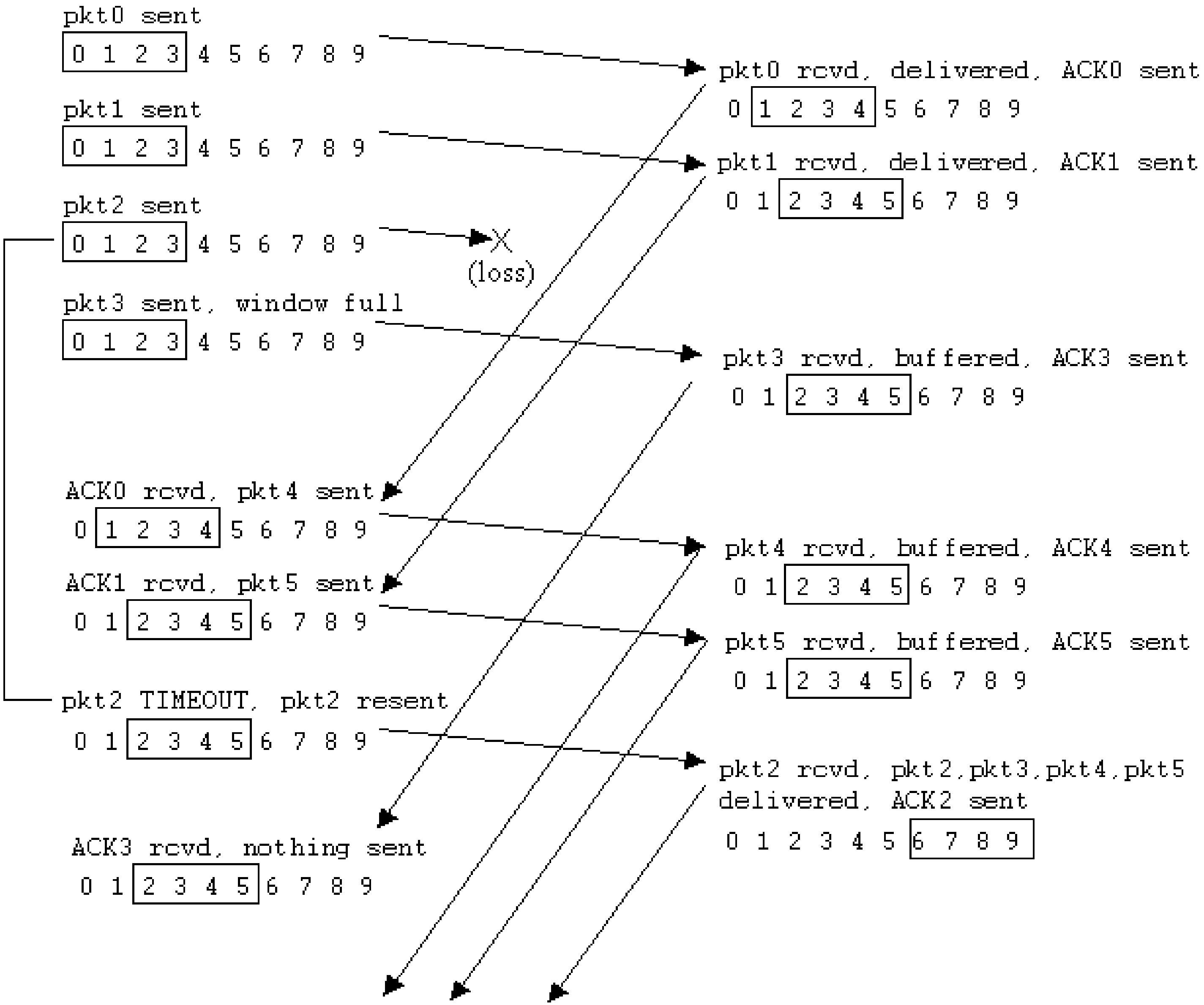
- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

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

- ACK(n)
  - ACK for pkt was lost, rexmit

otherwise: ignore

Selective repeat in action



pkt0 sent
0 1 2 3 4 5 6 7 8 9

pkt1 sent
0 1 2 3 4 5 6 7 8 9

pkt2 sent
0 1 2 3 4 5 6 7 8 9

(loss)

pkt3 sent, window full
0 1 2 3 4 5 6 7 8 9

ACK0 rcvd, pkt4 sent
0 1 2 3 4 5 6 7 8 9

ACK1 rcvd, pkt5 sent
0 1 2 3 4 5 6 7 8 9

pkt2 TIMEOUT, pkt2 resent
0 1 2 3 4 5 6 7 8 9

ACK3 rcvd, nothing sent
0 1 2 3 4 5 6 7 8 9

pkt0 rcvd, delivered, ACK0 sent
0 1 2 3 4 5 6 7 8 9

pkt1 rcvd, delivered, ACK1 sent
0 1 2 3 4 5 6 7 8 9

pkt3 rcvd, buffered, ACK3 sent
0 1 2 3 4 5 6 7 8 9

pkt4 rcvd, buffered, ACK4 sent
0 1 2 3 4 5 6 7 8 9

pkt5 rcvd, buffered, ACK5 sent
0 1 2 3 4 5 6 7 8 9

pkt2 rcvd, pkt2,pkt3,pkt4,pkt5
delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9

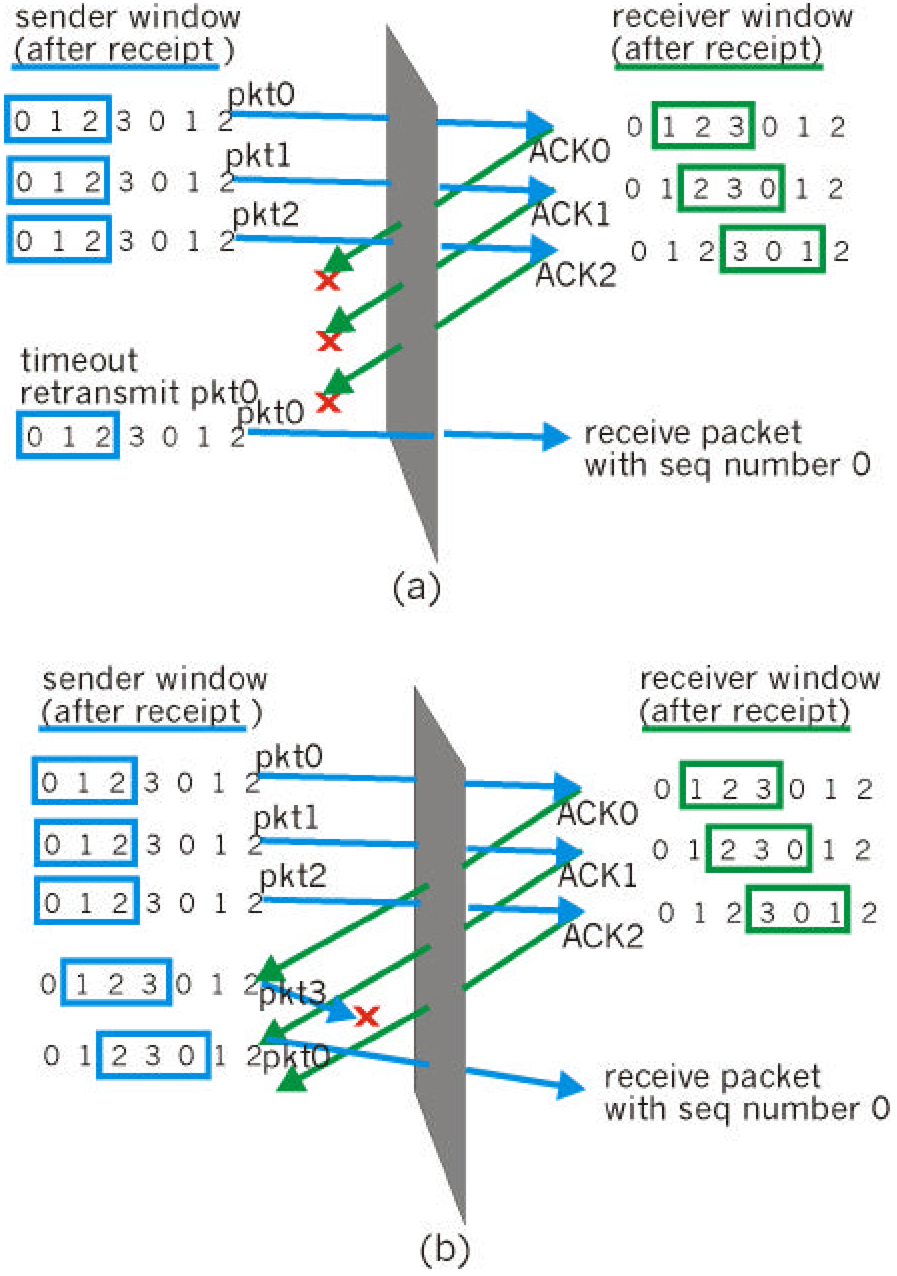
Selective repeat: dilemma

Example:

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

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

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



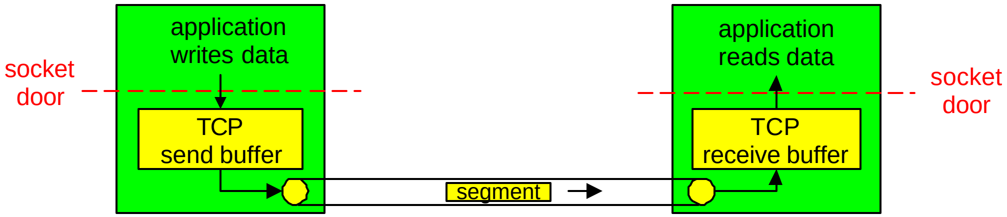
Chapter 3 outline

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

TCP: Overview

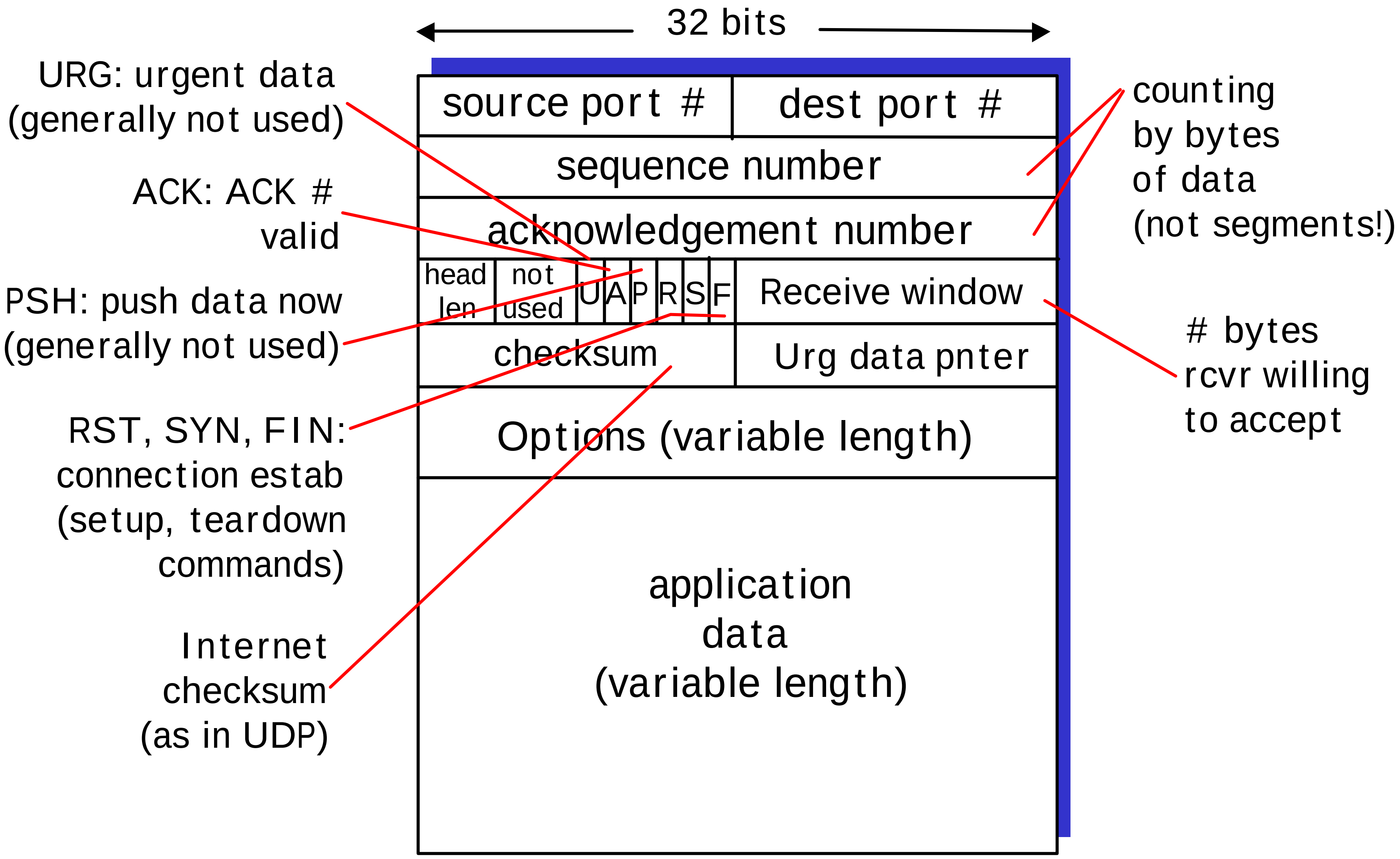
RFCs: 793, 1122, 1323, 2018, 2581

- 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

TCP segment structure



32 bits

source port #
dest port #
sequence number
acknowledgement number
head len
not used
U A P R S F
Receive window
checksum
Urg data pnter
Options (variable length)
application data (variable length)

URG: urgent data (generally not used)

ACK: ACK # valid

PSH: push data now (generally not used)

RST, SYN, FIN: connection estab (setup, teardown commands)

Internet checksum (as in UDP)

counting by bytes of data (not segments!)

# bytes rcvr willing to accept

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
- Segment integrity via checksum
- Cumulative acks
  - Receiver sends back the byte number it expects to receive next
  - Out of order packets generate duplicate acknowledgements
    - Receive 1, Ack 2
    - Receive 4, Ack 2
    - Receive 3, Ack 2
    - Receive 2, Ack 5
- Triggered retransmissions
  - Via timeout events
    - TCP uses single retransmission timer
    - Sender sends segment and sets a timer
    - Waits for an acknowledgement indicating segment was received
      - Send 1
      - Wait for Ack 2
      - No Ack 2 and timer expires
      - Send 1 again
  - Via duplicate acks
- Pipelined, congestion-controlled segments

TCP segment integrity

- Checksum included in header
- Is it sufficient to just checksum the packet contents?
- No, need to ensure correct source/destination
  - Pseudoheader – portion of IP hdr that are critical
  - Checksum covers Pseudoheader, transport hdr, and packet body
  - Layer violation, redundant with parts of IP checksum

TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
           smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
               start timer
          }

 }  /* end of loop forever */
```

Comment:
- SendBase-1: last cumulatively ack'ed byte

Example:
- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

TCP delayed acknowledgements

- Problem:
  - In request/response programs, you send separate ACK and Data packets for each transaction
    - Delay ACK in order to send ACK back along with data
- Solution:
  - Don’t ACK data immediately
    - Wait 200ms (must be less than 500ms – why?)
    - Must ACK every other packet
    - Must not delay duplicate ACKs
  - Without delayed ACK: 40 byte ack + data packet
  - With delayed ACK: data packet includes ACK
  - See web trace example
  - Extensions for asymmetric links
    - See later part of lecture

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 200ms 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	Immediately send ACK, provided that segment starts at lower end of gap

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?

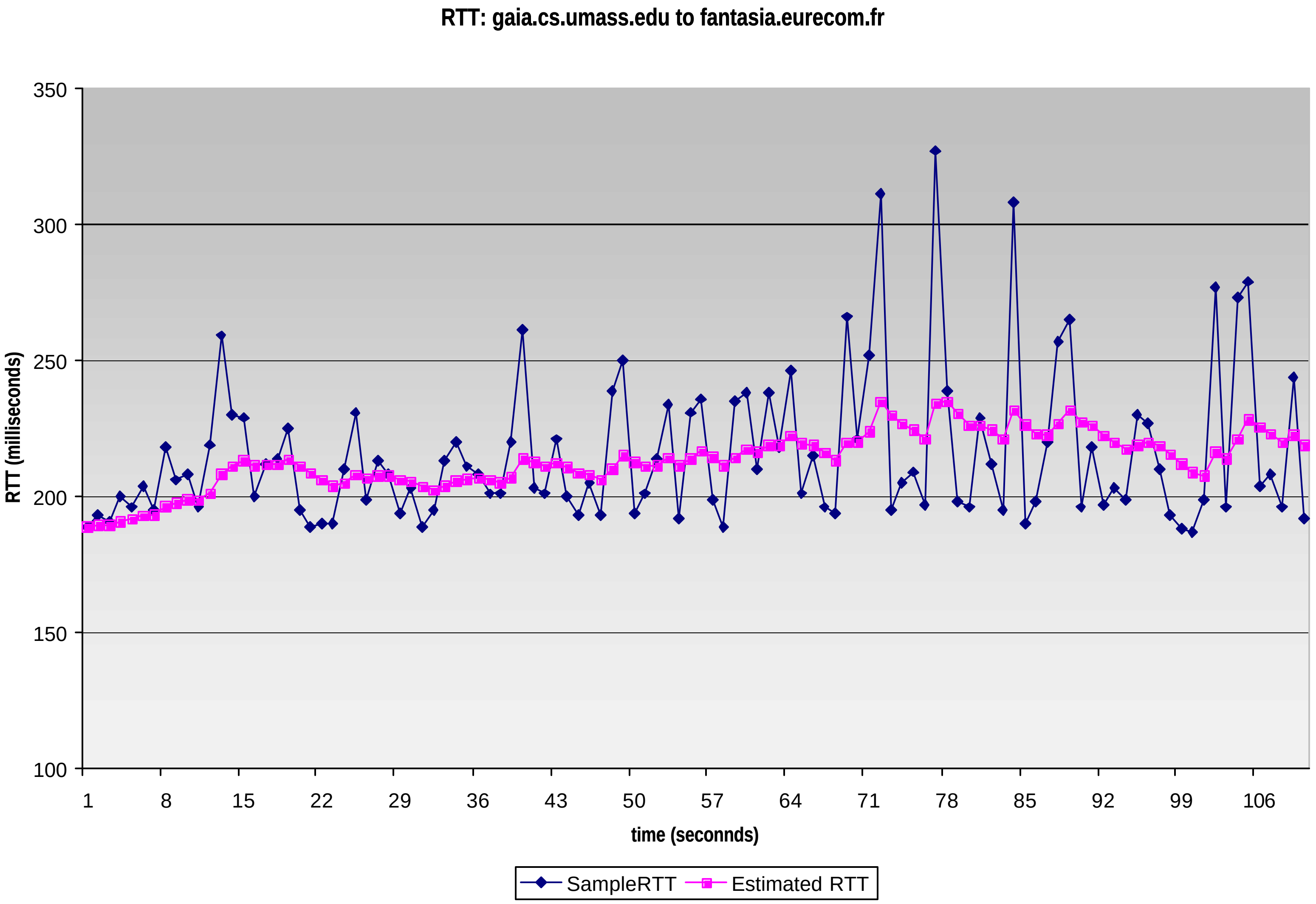
- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **`SampleRTT`**

TCP Round Trip Time Estimator and Timeout

EstimatedRTT = (1- a)*EstimatedRTT + a*SampleRTT

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: a = 0.125
- Initial retransmit timer set to β RTT, where β=2 currently
  - Not good at preventing spurious timeouts

Example RTT estimation:



TCP Round Trip Time and Timeout (Jacobson)

Setting the timeout

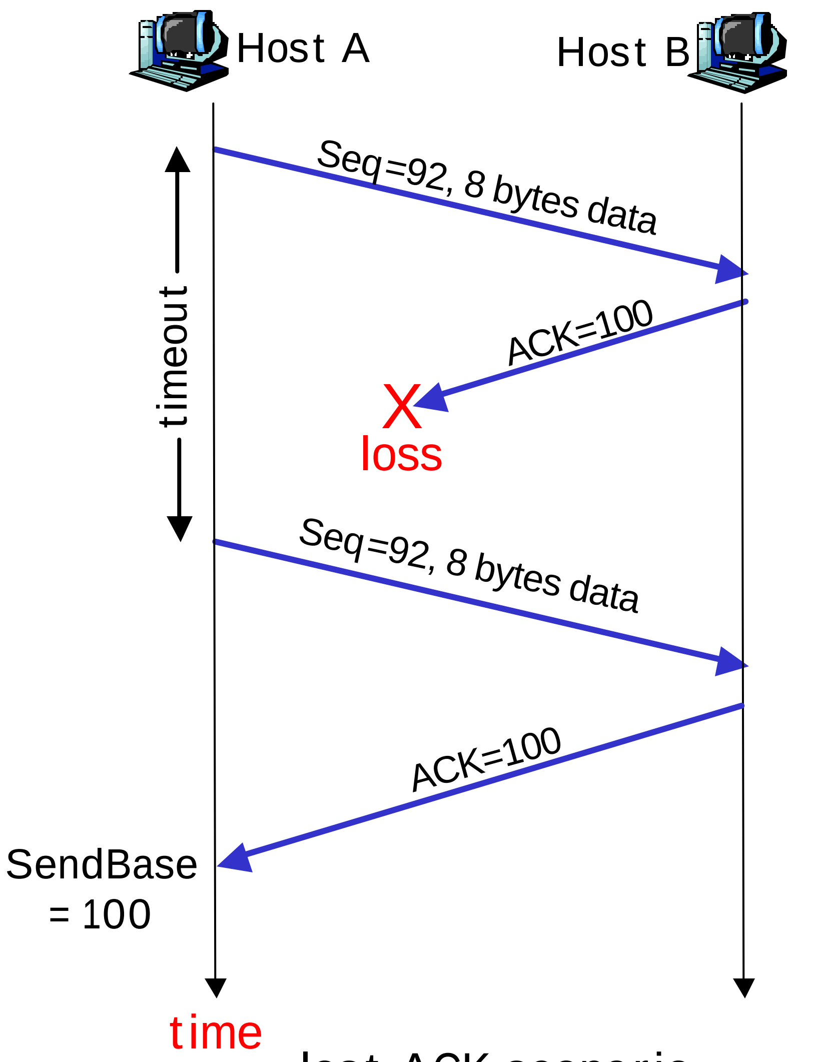
- first estimator produced spurious timeouts as RTT grew
- New estimator (Van Jacobson)
  - Observation: at high-loads RTT variance is high
  - Need larger safety margin with larger variations in RTT
    - **EstimtedRTT** plus "safety margin"
    - large variation in **EstimatedRTT ->** larger safety margin
  - first estimate of how much SampleRTT deviates from EstimatedRTT:

DevRTT = (1-b)*DevRTT + b*|SampleRTT-EstimatedRTT|
(typically, b = 0.25)

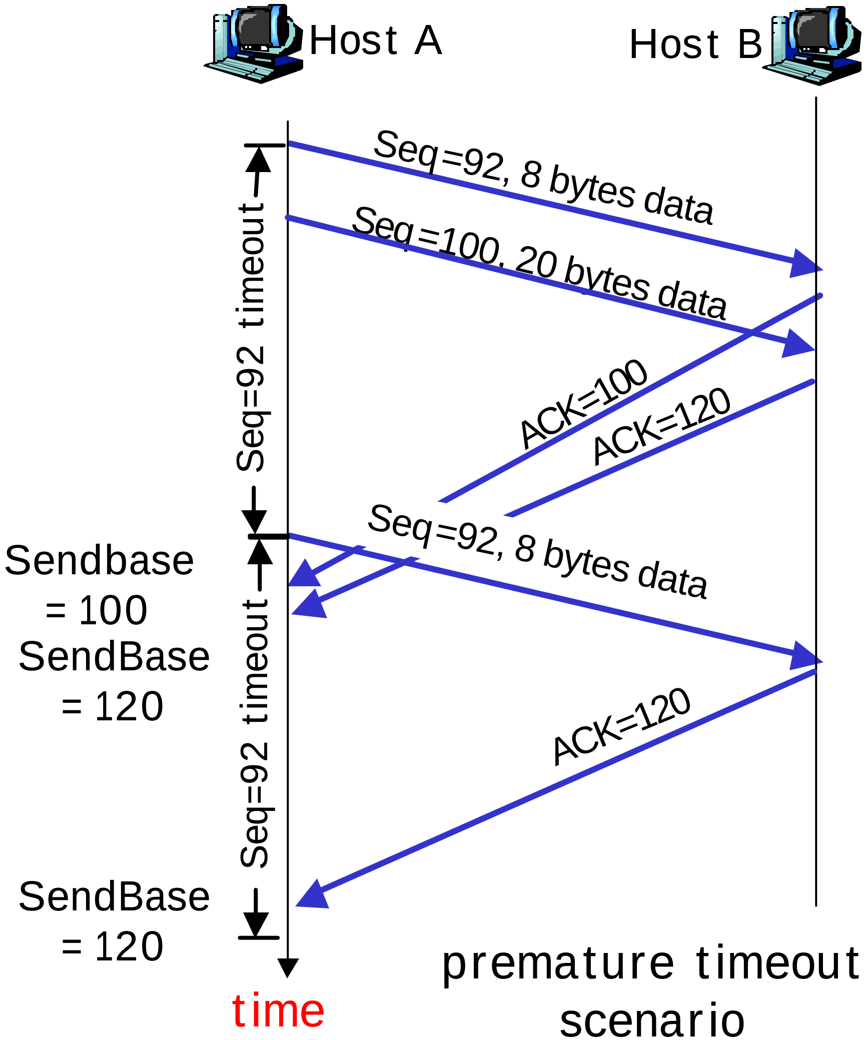
Then set timeout interval:

TimeoutInterval = EstimatedRTT + 4*DevRTT

TCP: retransmission scenarios

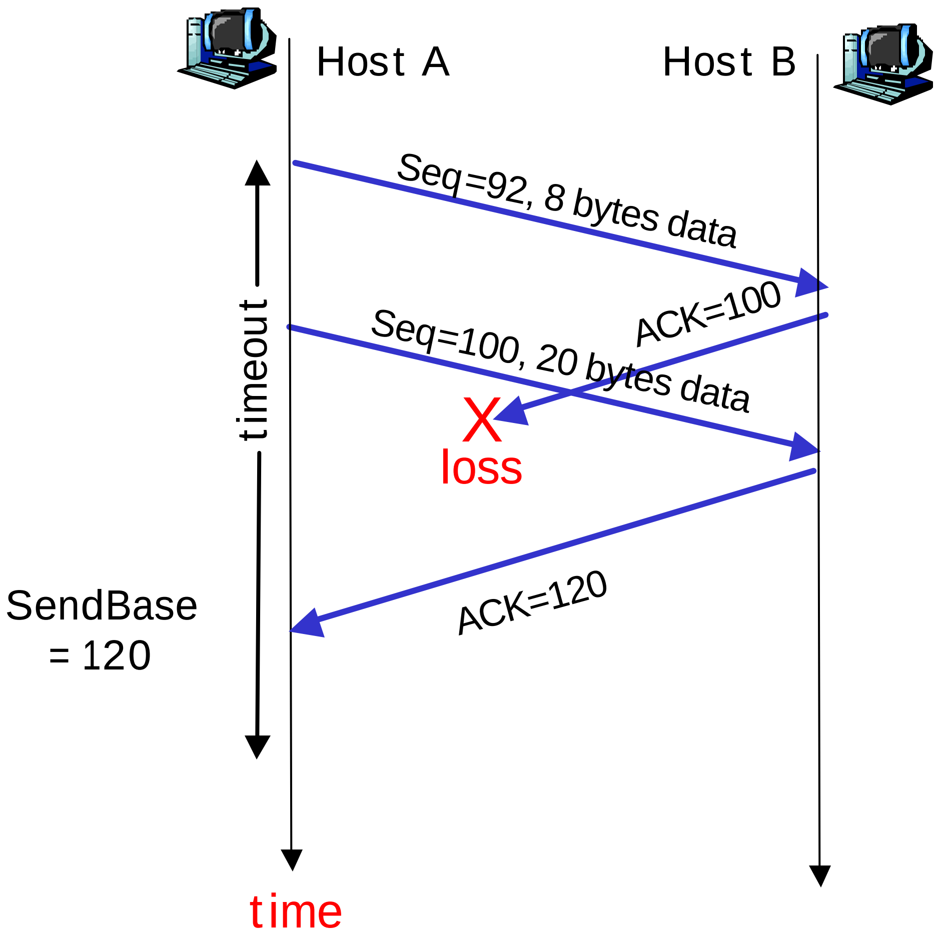


lost ACK scenario



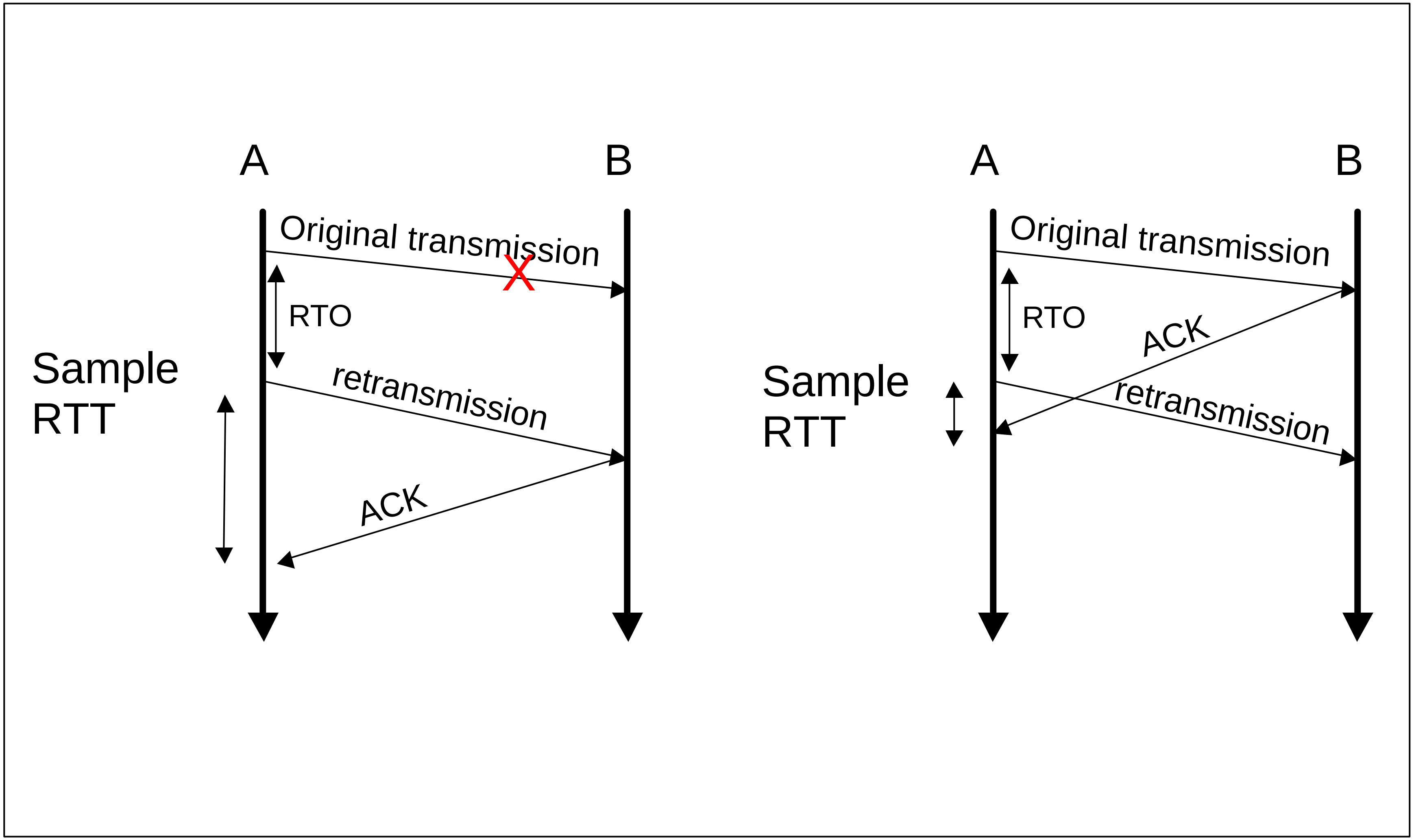
premature timeout scenario

TCP retransmission scenarios (more)



Cumulative ACK scenario

TCP retransmission ambiguity



Karn's algorithm

- Accounts for retransmission ambiguity
- If a segment has been retransmitted:
  - Don't count RTT sample on ACKs for this segment
  - Keep backed off time-out for next packet
  - Reuse RTT estimate only after one successful transmission

TCP retransmission miscelleny

- Backing off TCP's retransmission timeout
  - What if successive TCP retransmissions timeout?
    - Every time timer expires for same segment, RTO is doubled
    - Exponential back-off similar to Ethernet until successful retransmission

TCP retransmission miscellany

- TCP timer granularity
  - Many TCP implementations set RTO in multiples of 200,500,1000ms
  - Why?
    - Avoid spurious timeouts – RTTs can vary quickly due to cross traffic
    - Make timers interrupts efficient

Fast retrasmit
Recall TCP ACK generation....

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 startsat 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 is lost, there will likely be many duplicate ACKs.

- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
             }
          else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                }
```

a duplicate ACK for already ACKed segment

fast retransmit

Chapter 3 outline

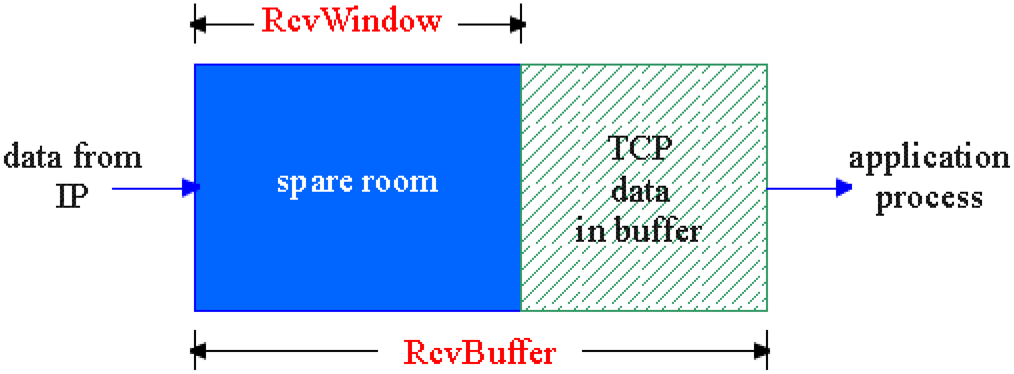
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of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

TCP Flow Control

- TCP is a sliding window protocol
  - For window size n, can send up to n bytes without receiving an acknowledgement
  - When the data is acknowledged then the window slides forward
- Each packet advertises a window size
  - Indicates number of bytes the receiver has space for
- Original TCP always sent entire window
  - Congestion control now limits this

TCP Flow Control

- receive side of TCP connection has a receive buffer:



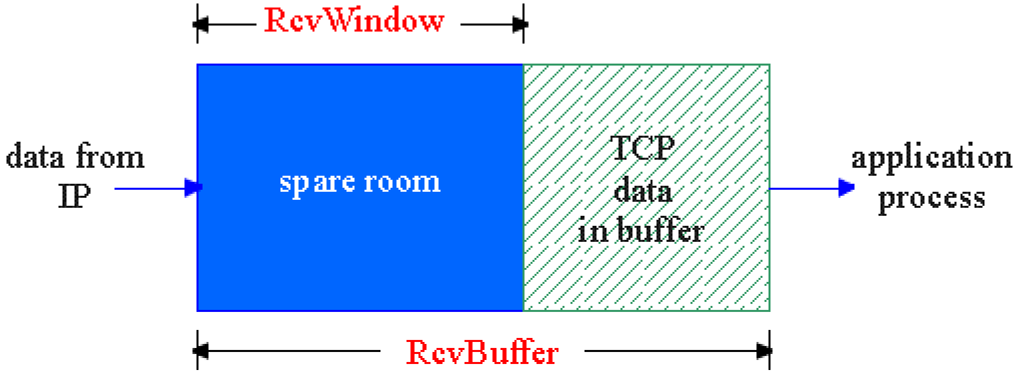
- app process may be slow at reading from buffer

flow control

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

- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works



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

- spare room in buffer

= RcvWindow

= RcvBuffer-[LastByteRcvd - LastByteRead]

- Rcvr advertises spare room by including value of **RcvWindow** in segments
- Sender limits unACKed data to **RcvWindow**
  - guarantees receive buffer doesn't overflow

TCP Flow control

- What happens if window is 0?
  - Receiver updates window when application reads data
  - What if this update is lost?
    - Deadlock
- TCP Persist timer
  - Sender periodically sends window probe packets
  - Receiver responds with ACK and up-to-date window advertisement

TCP flow control enhancements

- Problem: (Clark, 1982)
  - If receiver advertises small increases in the receive window then the sender may waste time sending lots of small packets
- What happens if window is small?
  - Small packet problem known as “Silly window syndrome”
    - Receiver advertises one byte window
    - Sender sends one byte packet (1 byte data, 40 byte header = 4000% overhead)

TCP flow control enhancements

- Solutions to silly window syndrome
  - Clark (1982)
    - receiver avoidance
    - prevent receiver from advertising small windows
    - increase advertised receiver window by min(MSS, RecvBuffer/2)

TCP flow control enhancements

- Solutions to silly window syndrome
  - Nagle's algorithm (1984)
    - sender avoidance
    - prevent sender from unnecessarily sending small packets
    - http://www.rfc-editor.org/rfc/rfc896.txt
      - Allow only one outstanding small (not full sized) segment that has not yet been acknowledged
      - Works for idle connections (no deadlock)
      - Works for telnet (send one-byte packets immediately)
      - Works for bulk data transfer (delay sending)

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`**)
  - Window scaling
- *client:* connection initiator

```
Socket clientSocket = new
 Socket("hostname","port

 number");
```

- *server:* contacted by client

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

Three way handshake:

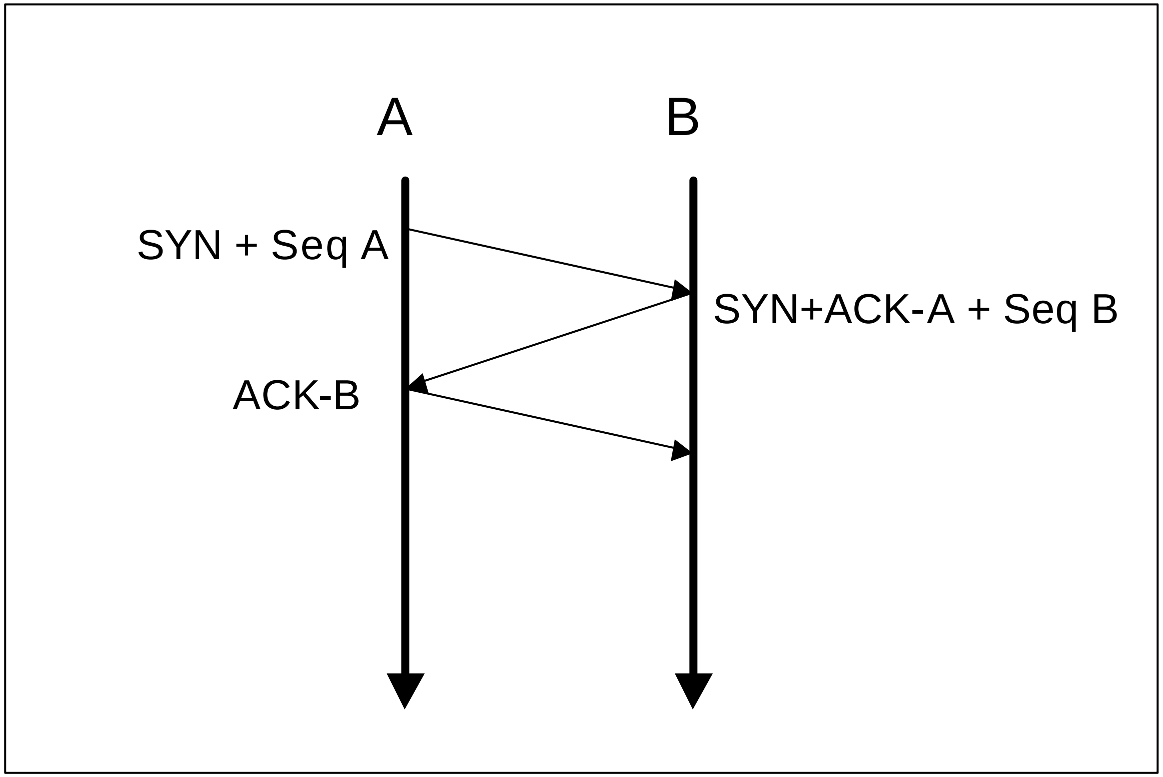
Step 1: client host sends TCP SYN segment to server
  - specifies initial seq #
  - no data, should be random

- Step 2: server host receives SYN, replies with SYNACK segment
  - server allocates buffers
  - specifies server initial seq. # and adv. window

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Establishment

- 3-way handshake with initial sequence number selection



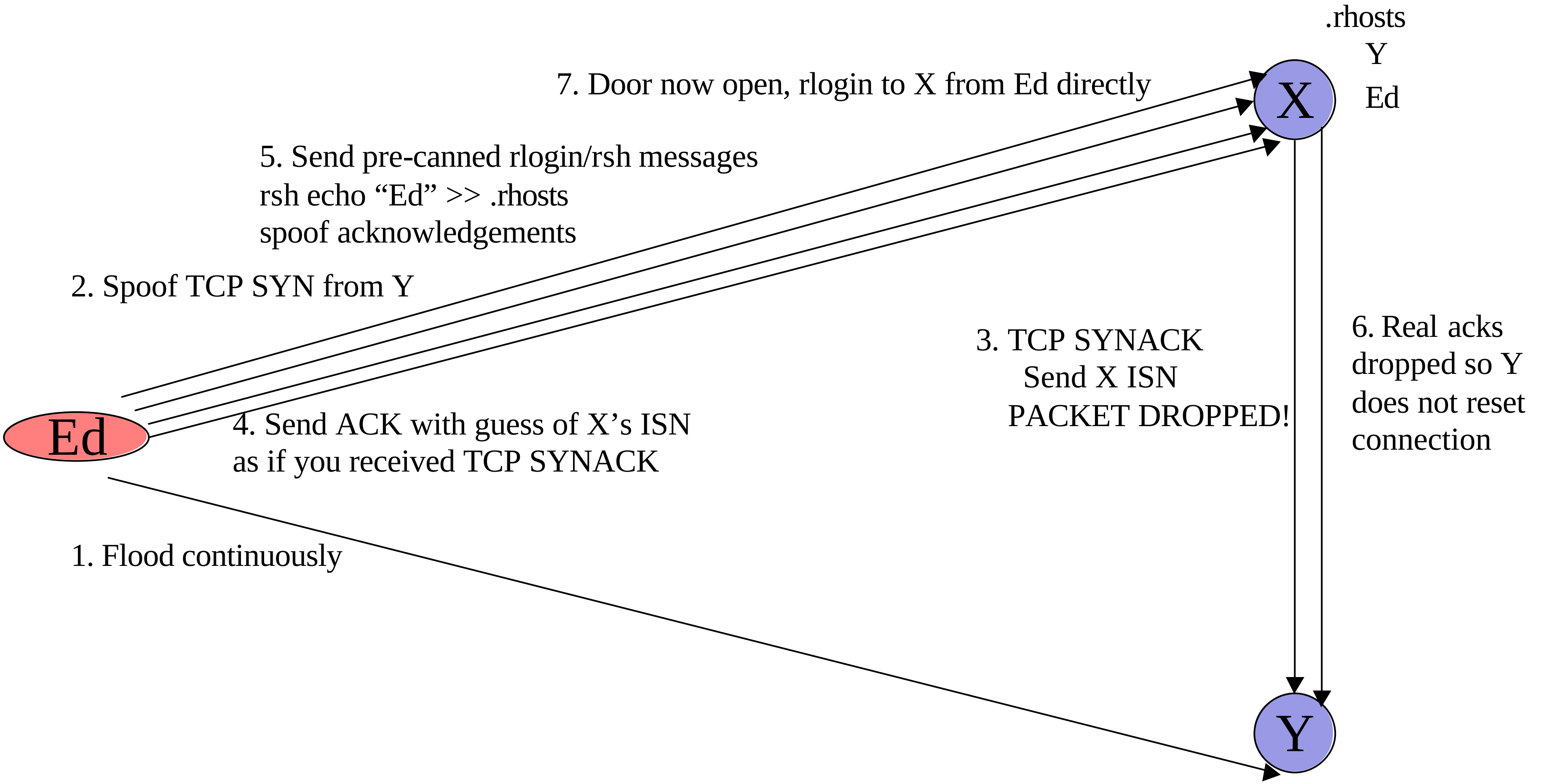
TCP Sequence Number Selection

- Why not simply chose 0?
- Must avoid overlap with earlier incarnation
- Client machine seq #0, initiates connection to server with seq #0.
  - Client sends one byte and machine crashes
  - Client reboots and initiates connection again
  - Server thinks new incarnation is the same as old connection

TCP Sequence Number Selection

- Why is selecting a random ISN Important?
- Suppose machine X selects ISN based on predictable sequence
- Fred has .rhosts to allow login to X from Y
- Evil Ed attacks
  - Disables host Y – denial of service attack
  - Determines ISN pattern at X
    - Make a bunch of connections to host X
    - Determine ISN pattern a guess next ISN
  - Blindly masquerade as Y using guessed ISN of X
    - Ed never sees real ISN of X since it is sent to Y
  - Attack popularized by K. Mitnick

TCP ISN selection and spoofing attacks



1. Flood continuously

2. Spoof TCP SYN from Y

3. TCP SYNACK
Send X ISN
PACKET DROPPED!

4. Send ACK with guess of X’s ISN
as if you received TCP SYNACK

5. Send pre-canned rlogin/rsh messages
rsh echo “Ed” >> .rhosts
spoof acknowledgements

6. Real acks
dropped so Y
does not reset
connection

7. Door now open, rlogin to X from Ed directly

TCP connections

Data transfer for established connections using sequence numbers and sliding windows with cumulative ACKs

Seq. #'s:

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

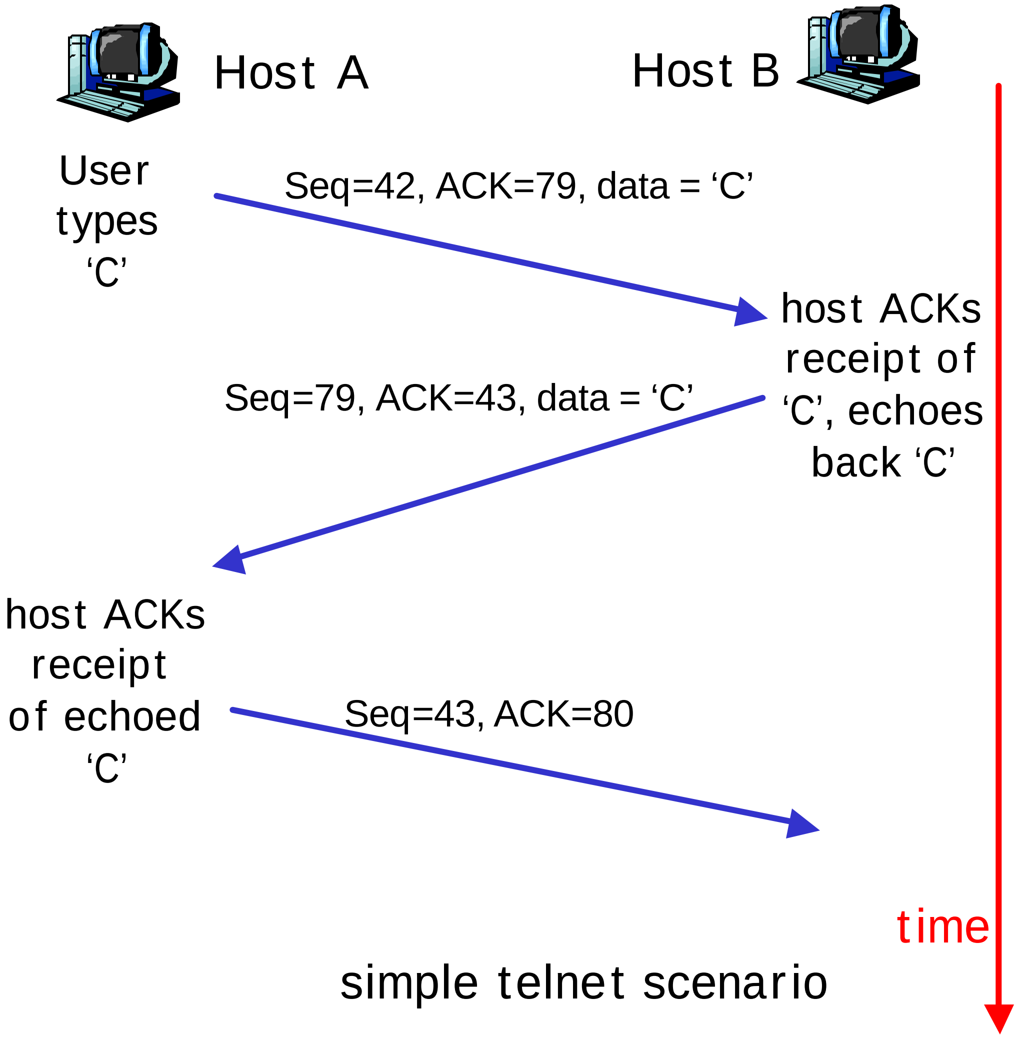
ACKs:

- seq # of next byte expected from other side
- cumulative ACK
- duplicate acks sent when out-of-order packet received

See web trace

Java API

```
connectionSocket.receive();
clientSocket.send();
```


Host A
Host B
User types 'C'
Seq=42, ACK=79, data = 'C'
host ACKs receipt of 'C', echoes back 'C'
Seq=79, ACK=43, data = 'C'
host ACKs receipt of echoed 'C'
Seq=43, ACK=80
time

simple telnet scenario

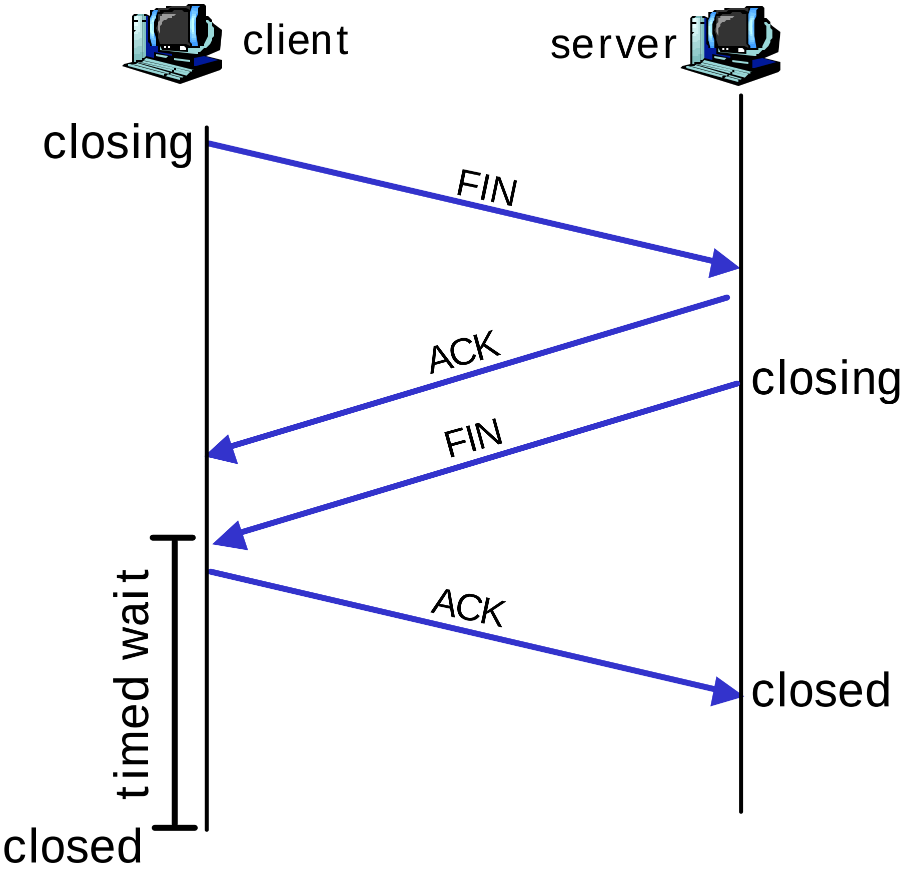
TCP Connection Management (cont.)

Closing a connection:

Client-initiated close (reverse for server-initiated close):
`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.



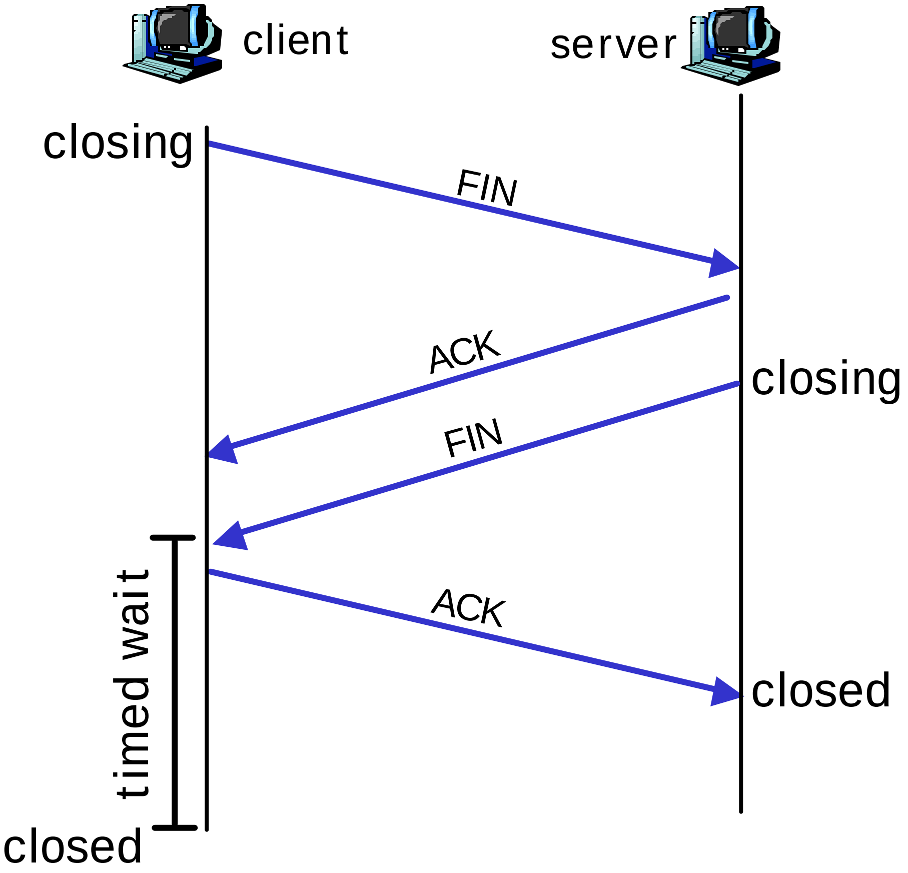
TCP Connection Management (cont.)

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.

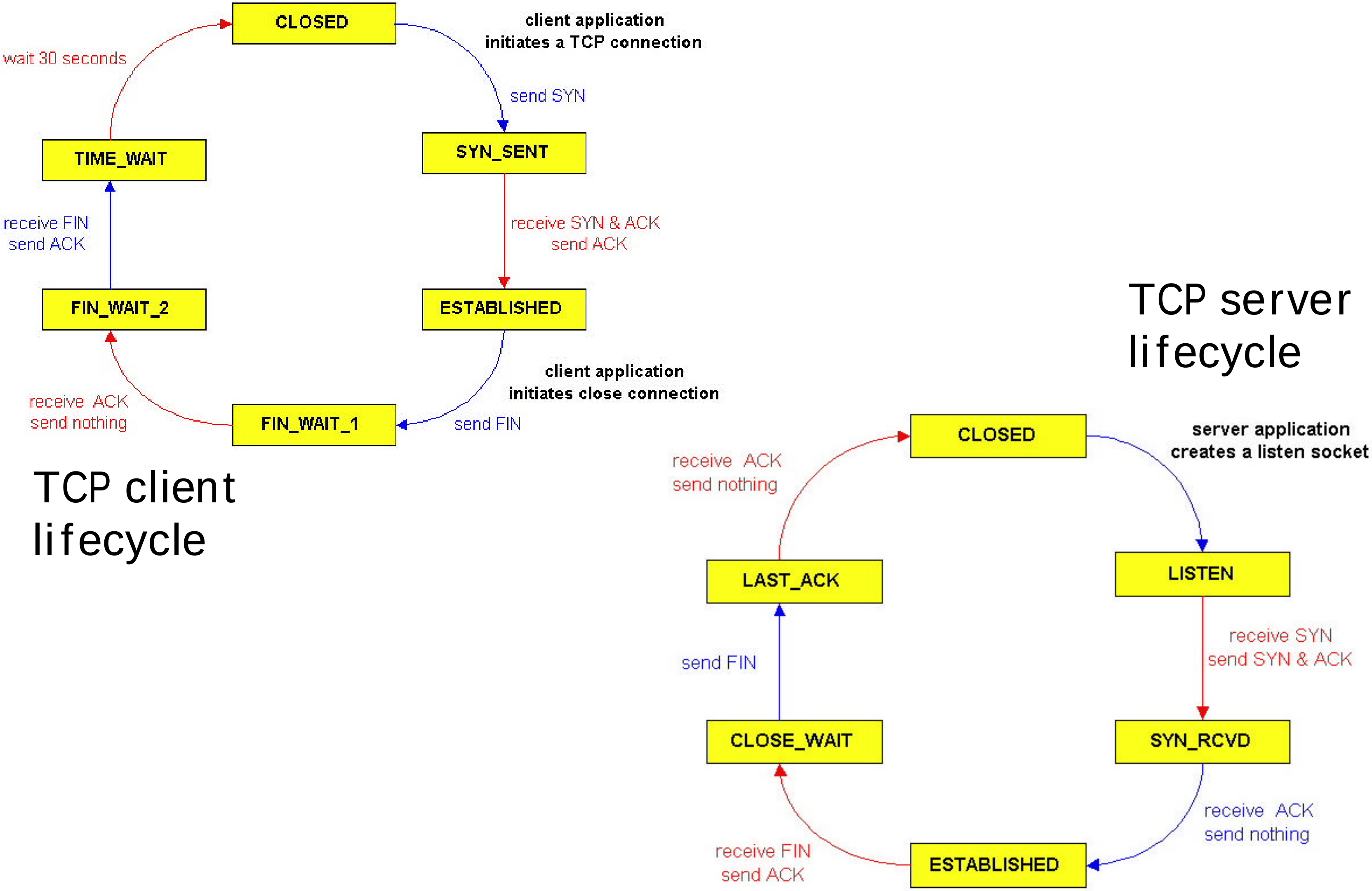
Note: with small modification, can handle simultaneous FINs.



Time Wait Issues

- Cannot close connection immediately after receiving FIN
  - What if a new connection restarts and uses same sequence number?
- Web servers not clients close connection first
  - Established -> Fin-Wait -> Time-Wait -> Closed
  - Why would this be a problem?
- Time-Wait state lasts for 2 * MSL
  - MSL is should be 120 seconds (is often 60s)
  - Servers often have order of magnitude more connections in Time-Wait

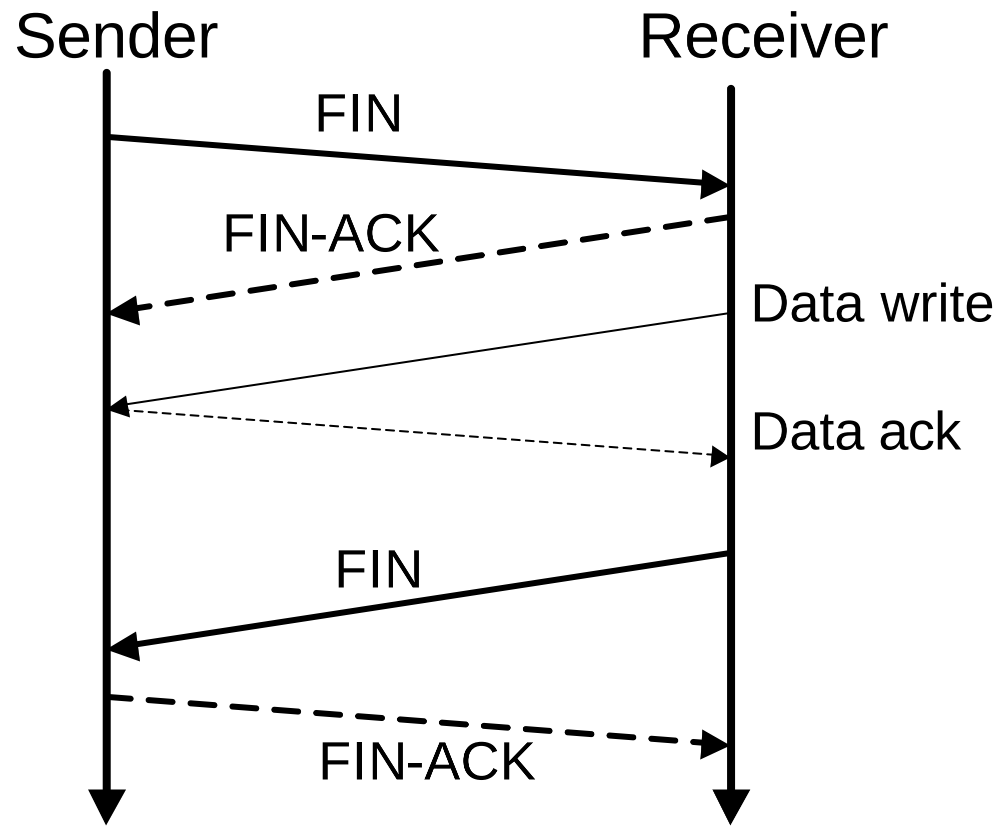
TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle

TCP Half-Close



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

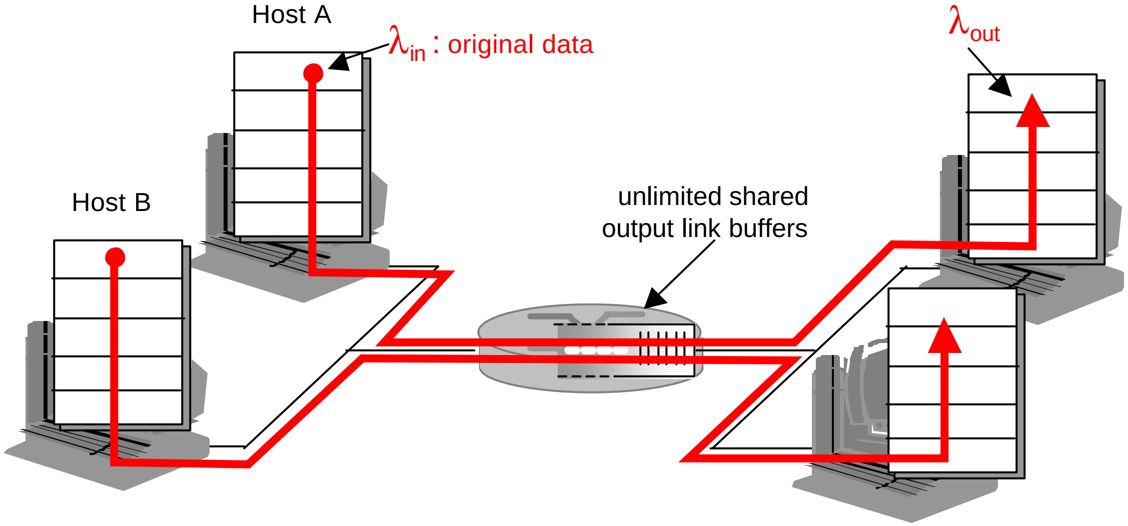
Principles of Congestion Control

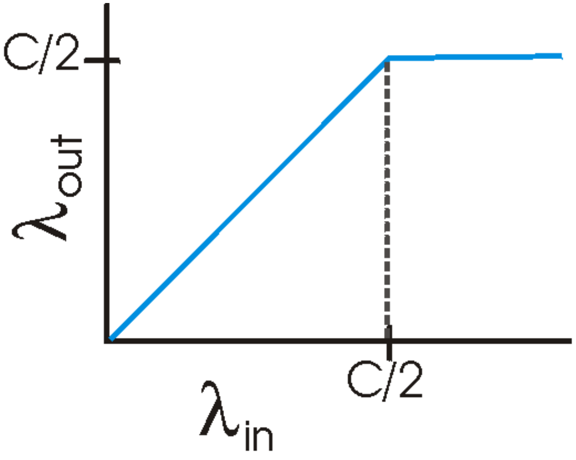
Congestion:

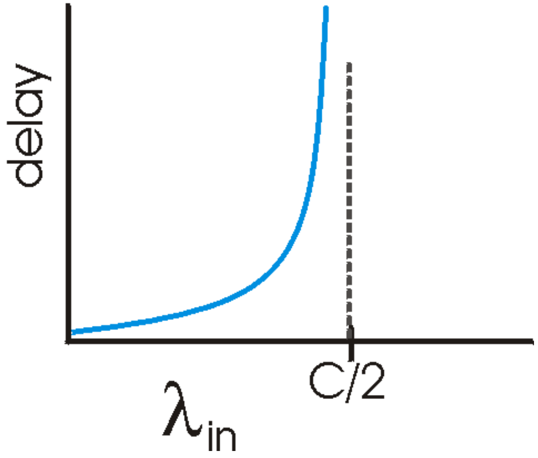
- informally: “too many sources sending too much data too fast for *network* to handle”
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

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


Host A
λ_{in} : original data
Host B
unlimited shared output link buffers
λ_{out}

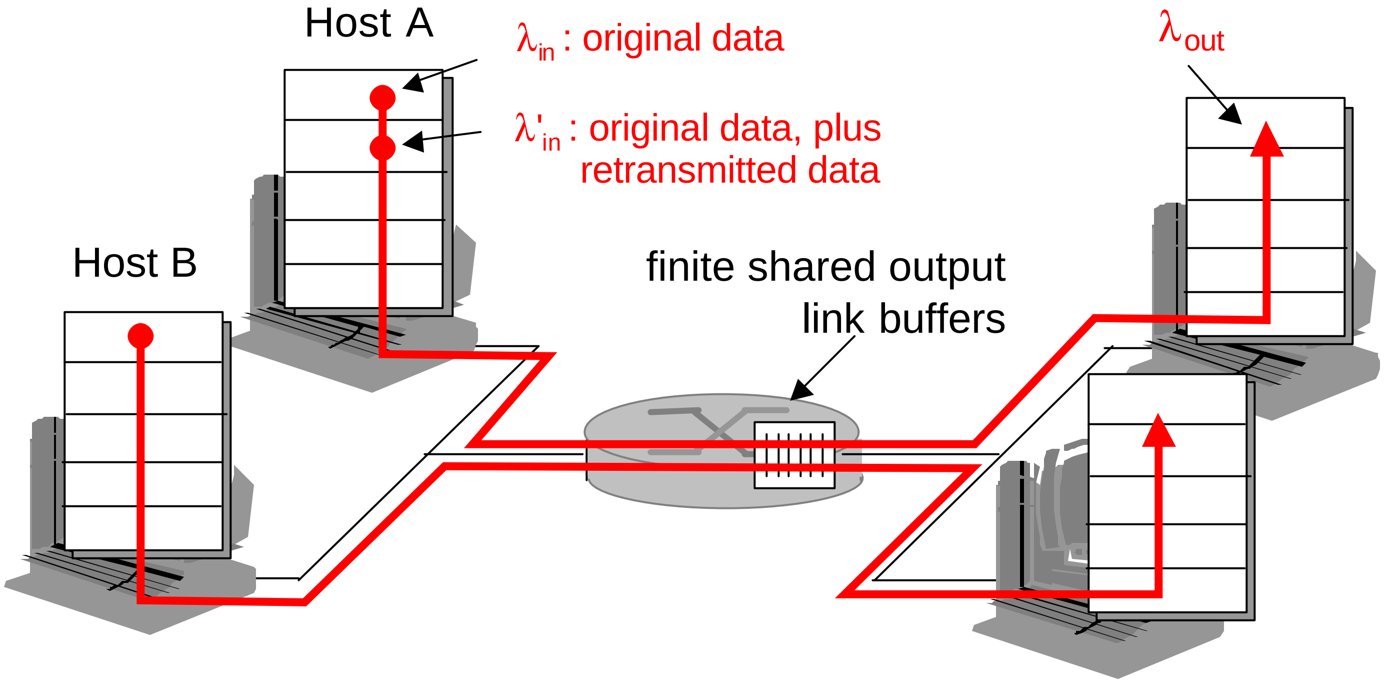




- large delays when congested
- maximum achievable throughput

Causes/costs of congestion: scenario 2

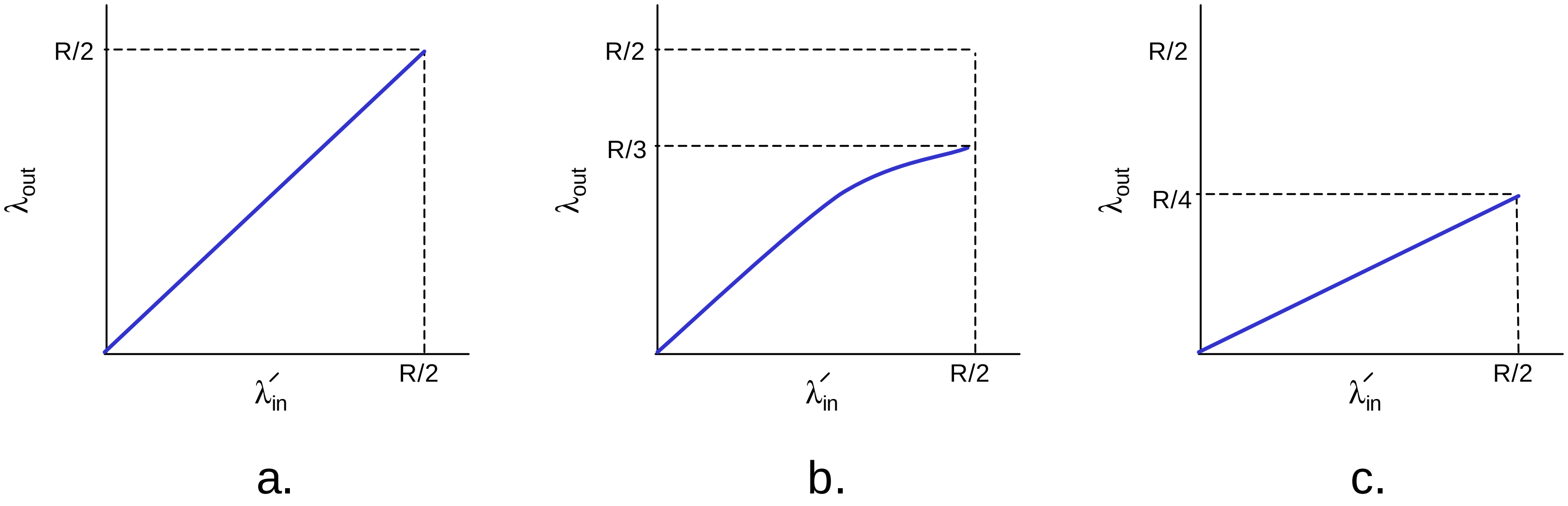
- one router, *finite* buffers
- sender retransmission of lost packet



Host A
λ_{in} : original data
λ'_{in} : original data, plus retransmitted data
λ_{out}
Host B
finite shared output link buffers

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}



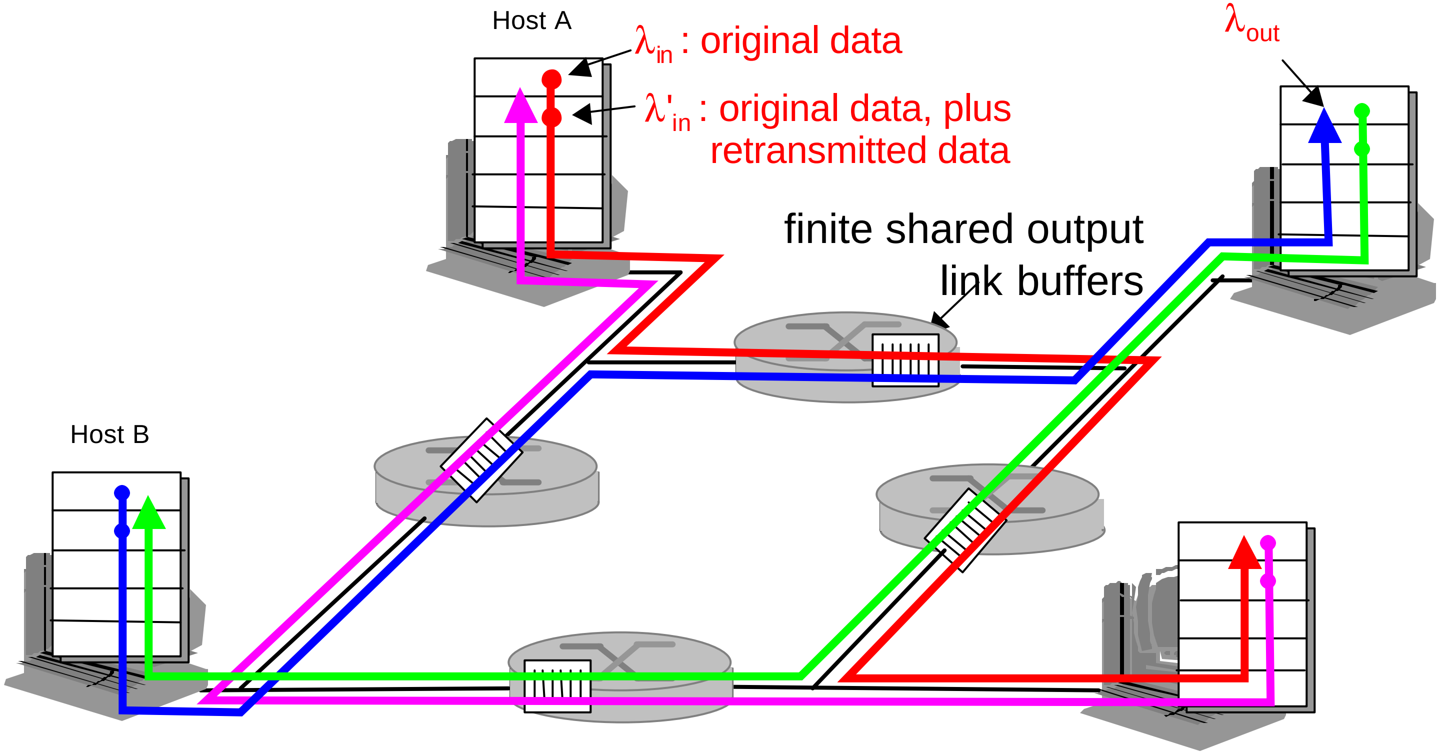
"costs" of congestion:

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

Causes/costs of congestion: scenario 3

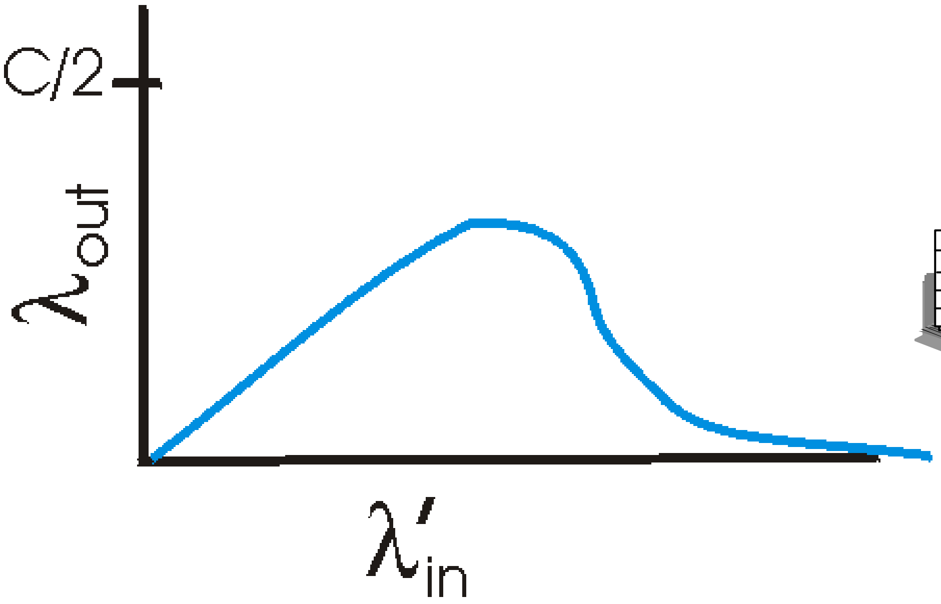
- four senders
- multihop paths
- timeout/retransmit

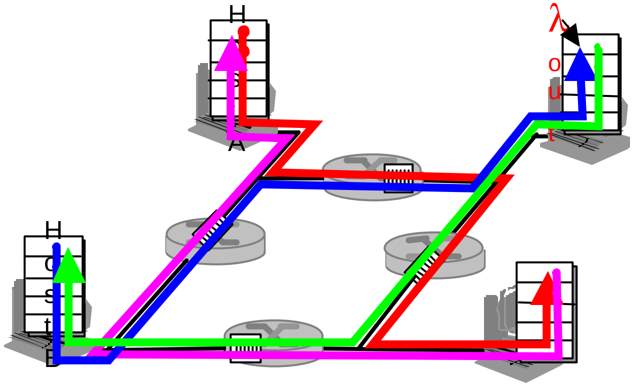
Q: what happens as λ_{in} and λ'_{in} increase ?



Host A
λ_{in} : original data
λ'_{in} : original data, plus retransmitted data
λ_{out}
finite shared output link buffers
Host B

Causes/costs of congestion: scenario 3





Another "cost" of congestion:

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

Congestion Collapse

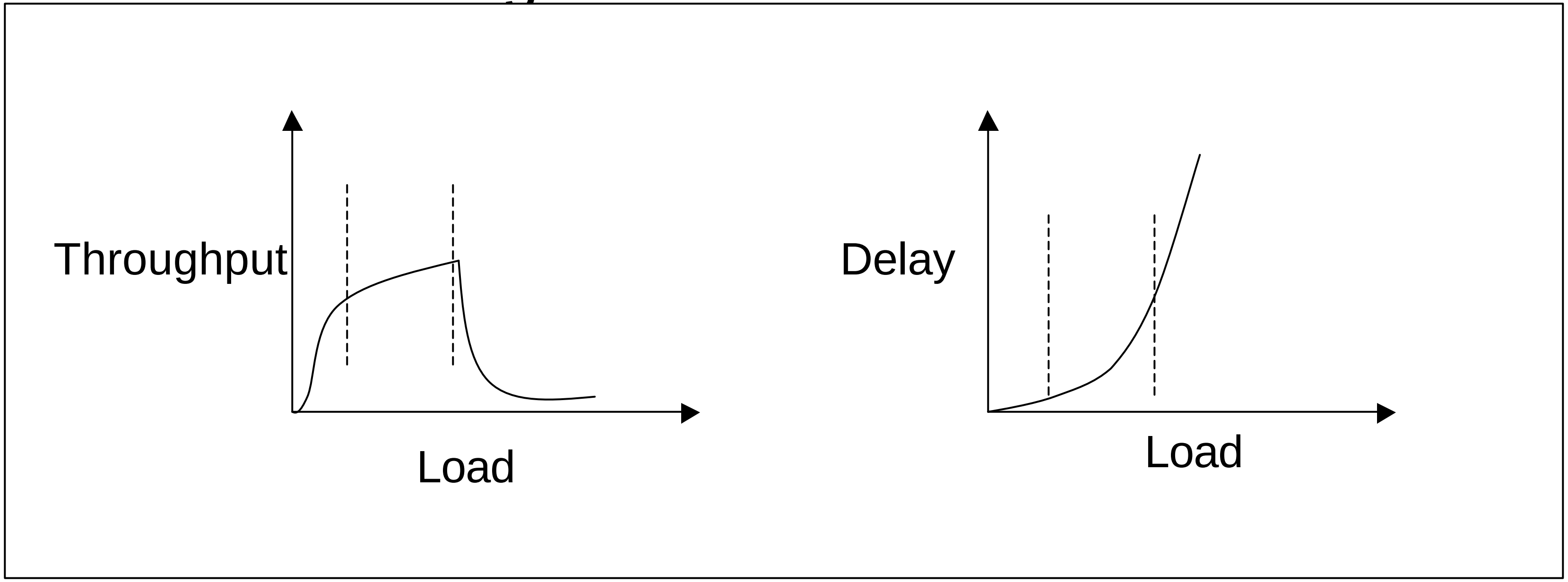
- Increase in network load results in decrease of useful work done
  - Spurious retransmissions of packets still in flight
    - Classical congestion collapse
    - Solution: better timers and congestion control
  - Undelivered packets
    - Packets consume resources and are dropped elsewhere in network
    - Solution: congestion control for ALL traffic
  - Fragments
    - Mismatch of transmission and retransmission units
    - Solutions:
      - Make network drop all fragments of a packet (early packet discard in ATM)
      - Do path MTU discovery
  - Control traffic
    - Large percentage of traffic is for control
    - Headers, routing messages, DNS, etc.
  - Stale or unwanted packets
    - Packets that are delayed on long queues
    - Solution: better congestion control and active queue management

Goals for congestion control

- Use network resources efficiently
  - 100% link utilization, 0% packet loss, Low delay
  - Maximize network power: (throughput$^{\alpha}$/delay)
  - Efficiency/goodput: $X_{knee} = \Sigma x_i(t)$
- Preserve fair network resource allocation
  - Fairness: $(\Sigma x_i)2/n(\Sigma x_i 2)$
  - Max-min fair sharing
    - Small flows get all of the bandwidth they require
    - Large flows evenly share leftover
  - Example: 100Mbs link
    - S1 and S2 are 1Mbs streams, S3 and S4 are greedy streams
    - S1 and S2 each get 1Mbs, S3 and S4 each get 49Mbs
- Convergence and stability
- Distributed operation
- Simple router and end-host behavior

Congestion Control vs. Avoidance

- Avoidance keeps the system performing at the knee/cliff
- Control kicks in once the system has reached a congested state



Congestion control approaches

- End-host vs. network controlled
  - Trust hosts to do the right thing
    - Hosts adjust rate based on detected congestion (TCP)
  - Don’t trust hosts and enforce within network
    - Network adjusts rates at congestion points
      - Scheduling
      - Queue management
    - Hard to prevent global collapse conditions locally
- Implicit vs. explicit network feedback
  - Implicit: infer congestion from packet loss or delay
    - Increase rate in absence of loss, decrease on loss (TCP Tahoe/Reno)
    - Increase rate based on RTT behavior (TCP Vegas, Packet pair)
  - Explicit: signalled from network
    - Congestion notification (IBM SNA, DECbit, ECN)
    - Rate signaling (ATM ABR)

Case study: ATM ABR congestion control

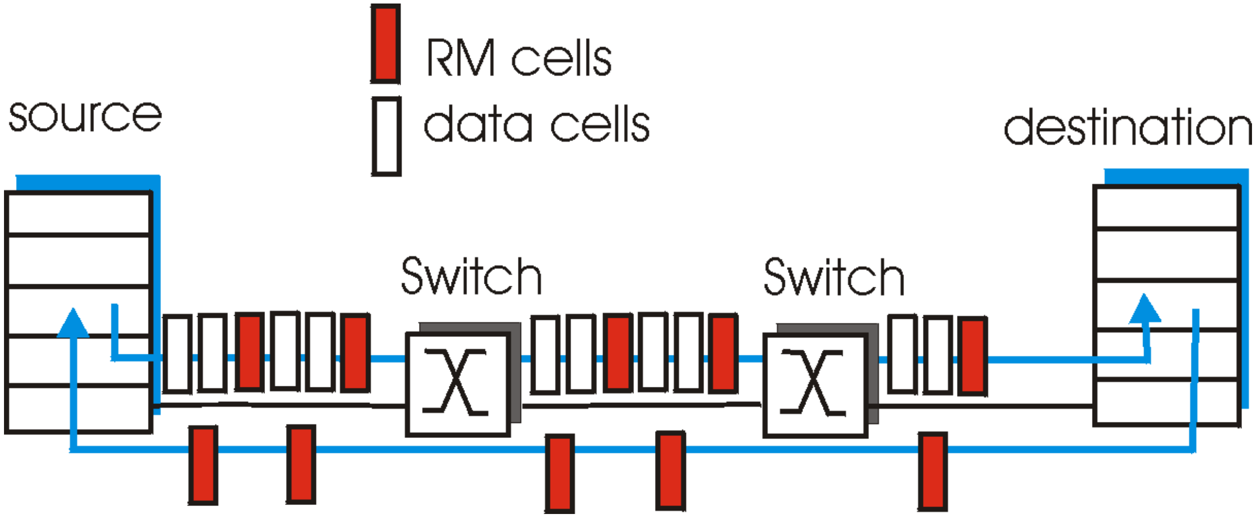
ABR: available bit rate:

- "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, sender 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

- Motivated by ARPANET congestion collapse
  - Flow control, but no congestion control
  - Sender sends as much as the receiver resources allows
  - Go-back-N on loss, burst out advertised window
- Congestion control
  - Extending control to network resources
  - Underlying design principle: packet conservation
    - At equilibrium, inject packet into network only when one is removed
    - Basis for stability of physical systems (fluid model)
- Why was this not working before?
  - No equilibrium
    - Solved by self-clocking
  - Spurious retransmissions
    - Solved by accurate RTO estimation (see earlier discussion)
  - Network resource limitations not considered
    - Solved by congestion window and congestion avoidance algorithms

TCP Congestion Control

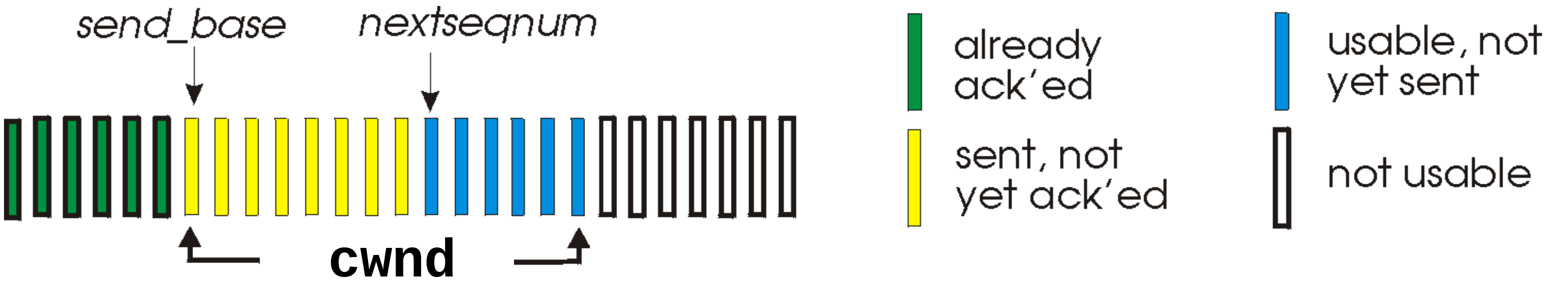
- Of all ways to do congestion, the Internet (TCP) chooses….
  - Mainly end-host, window-based congestion control
    - Only place to really prevent collapse is at end-host
    - Reduce sender window when congestion is perceived
    - Increase sender window otherwise (probe for bandwidth)
  - Congestion signaling and detection
    - Mark/drop packets when queues fill, overflow
    - Will cover this separately in later lecture

TCP congestion control basics

- Keep a congestion window, (`snd_cwnd`)
  - Book calls this "**`Congwin`**", also called just "**`cwnd`**"
  - Denotes how much network is able to absorb
- Receiver's advertised window (`rcv_wnd`)
  - Sent back in TCP header
- Sender's maximum window:
  - min (`rcv_wnd`, `snd_cwnd`)
- In operation, sender's actual window:
  - min(`rcv_wnd`, `snd_cwnd`) - unacknowledged segments

TCP Congestion Control

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



- For fixed window of w segments of MSS bytes length

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

TCP Congestion Control: details

- sender limits transmission:

  LastByteSent-LastByteAcked

  £ **CongWin**

- Roughly,

$$\boxed{\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}}$$

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

How does sender perceive congestion?

- loss event = timeout *or* 3 duplicate acks
- TCP sender reduces rate (**CongWin**) after loss event

three mechanisms:

- AIMD
- slow start
- Exponential backoff on RTO

TCP congestion control

- "probing" for usable bandwidth:
  - *ideally:* transmit as fast as possible (**cwnd** as large as possible) without loss
  - *increase* **cwnd** until loss (congestion)
  - loss: *decrease* **cwnd**, then begin probing (increasing) again

- two "phases" (TCP Tahoe)
  - slow start
  - congestion avoidance
- important variables:
  - **cwnd**
  - **ssthresh:** defines threshold between two slow start phase, congestion avoidance phase (Book calls this **threshold**)
- useful reference
  - http://www.aciri.org/floyd/papers/sacks.ps.Z

TCP Slow Start

- When connection begins, **`CongWin`** = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate

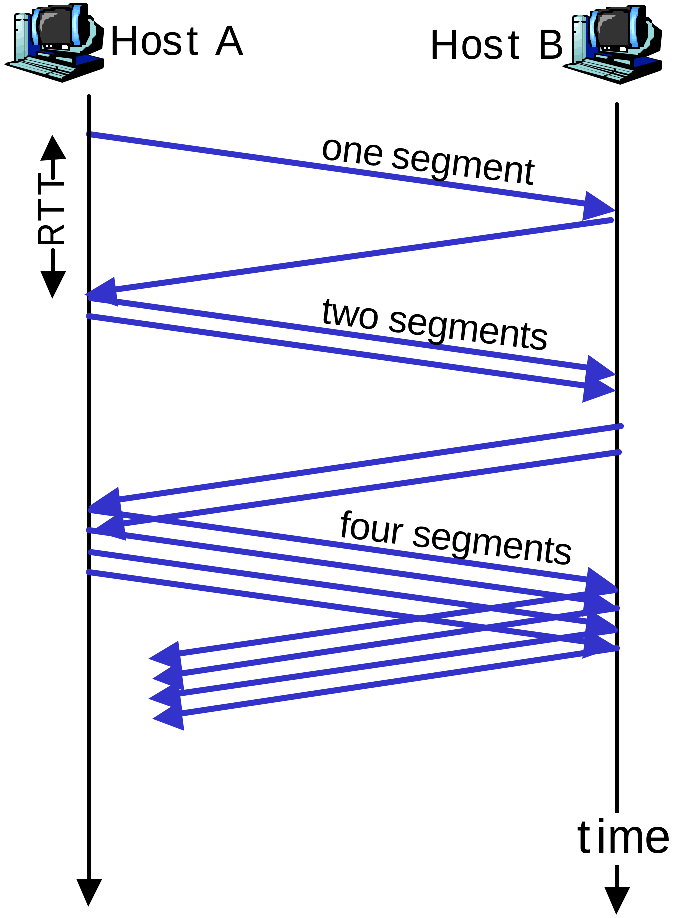
- When connection begins, increase rate exponentially fast until first loss event

TCP slow start

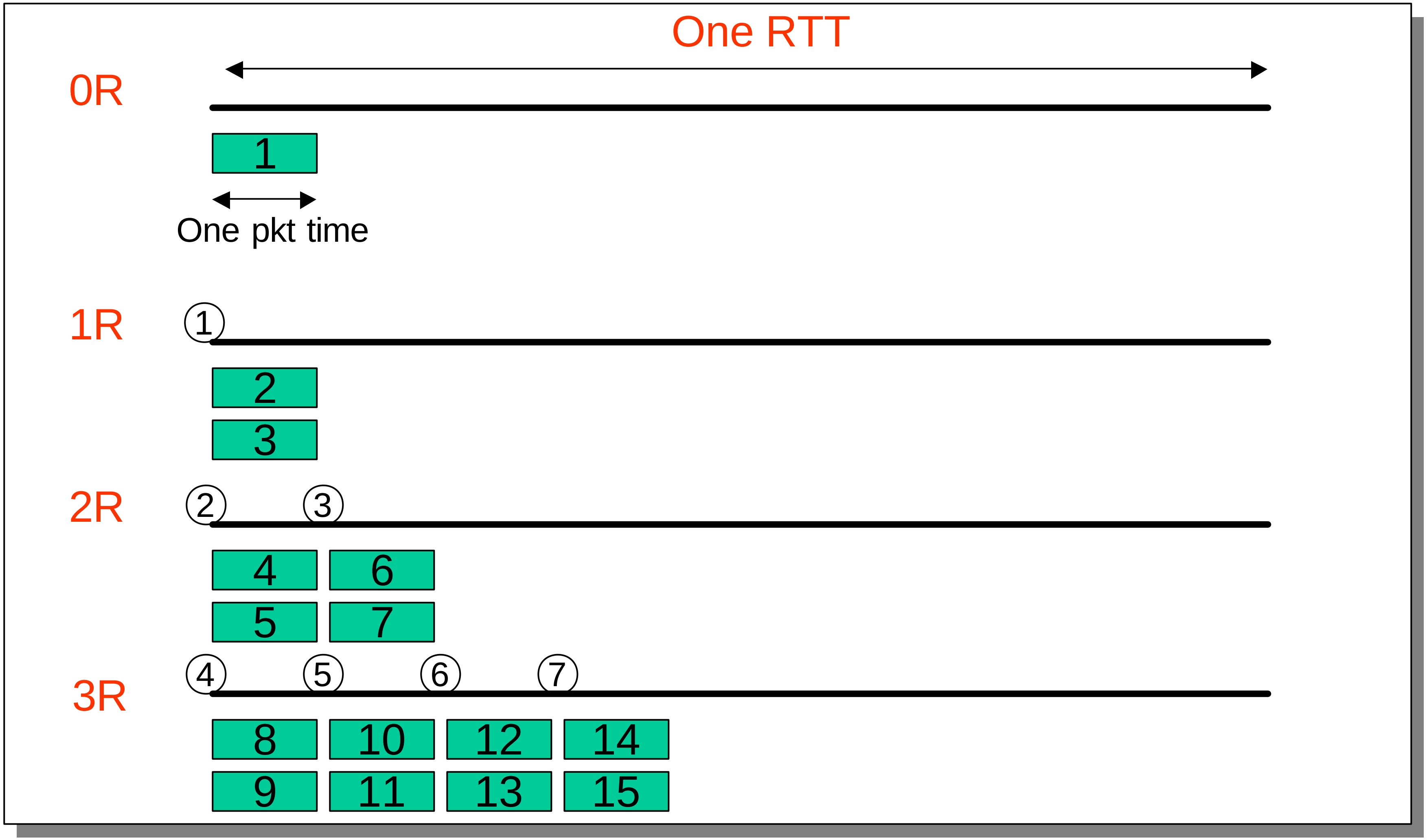
Slowstart algorithm

```
initialize: cwnd = 1
for (each segment ACKed)
      cwnd++
until (loss event OR
        cwnd > ssthresh)
```

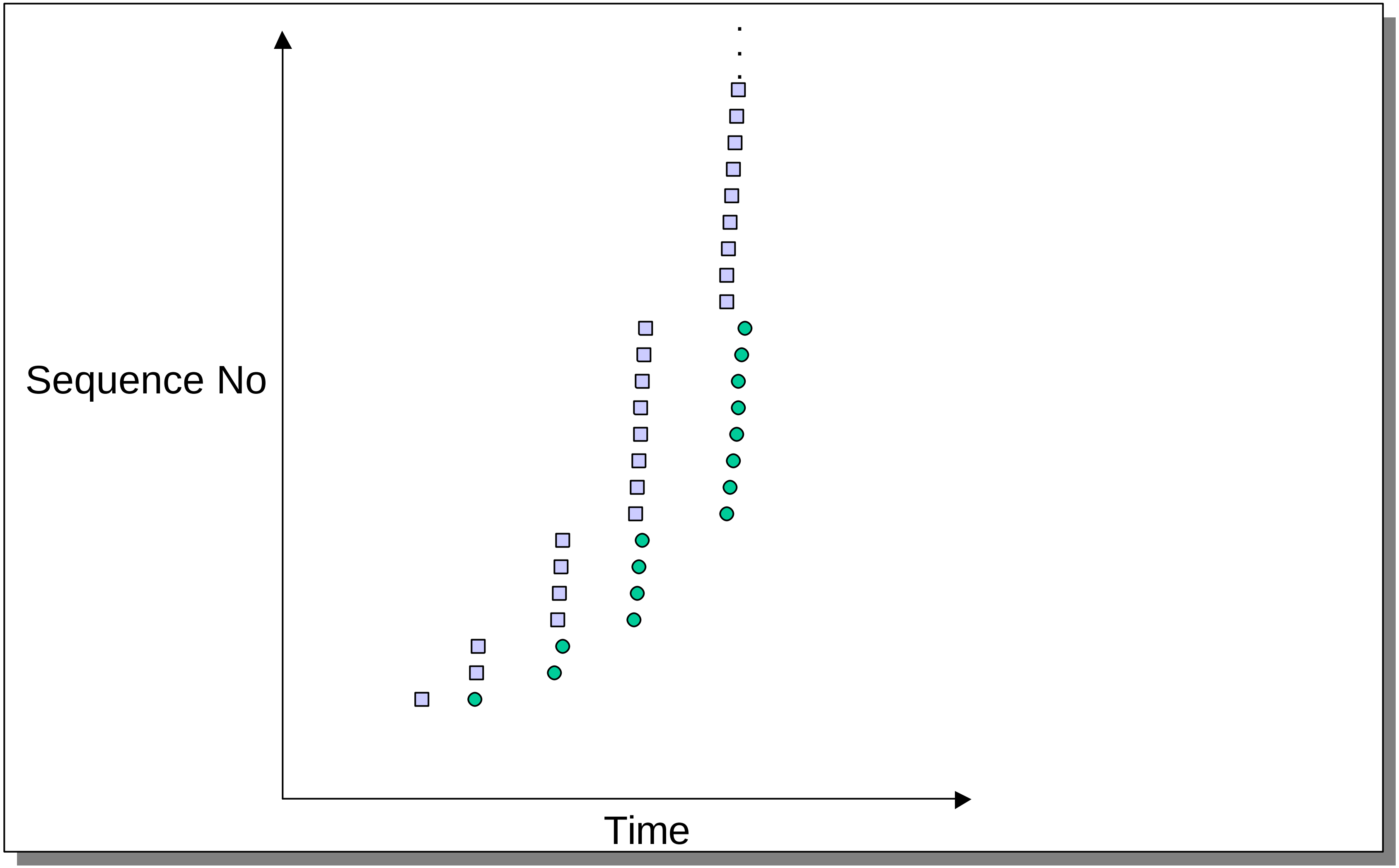
- exponential increase (per RTT) in window size
  - Start with cwnd=1, increase cwnd by 1 with every ACK
  - Window doubled every RTT
  - Increases to W in RTT * $\log_2(W)$
  - Can overshoot window and cause packet loss



TCP slow start example



TCP slow start sequence plot



Refinement (TCP congestion avoidance)

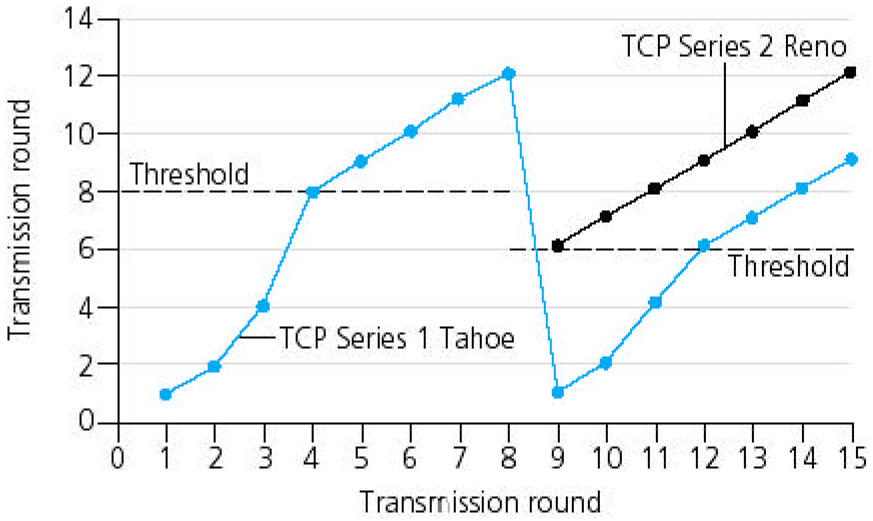
Q: When should the exponential increase switch to linear?

A: When **CongWin** gets to 1/2 of its value before timeout

Keep ssthresh and set to ½ CongWin at loss event

Congestion avoidance

```
/* slowstart is over       */
/* cwnd > ssthresh */
Until (loss event) {
  every w segments ACKed:
      cwnd++
  }
ssthresh = cwnd/2
If (Tahoe) cwnd=1;
If (Reno)  cwnd=ssthresh;
```

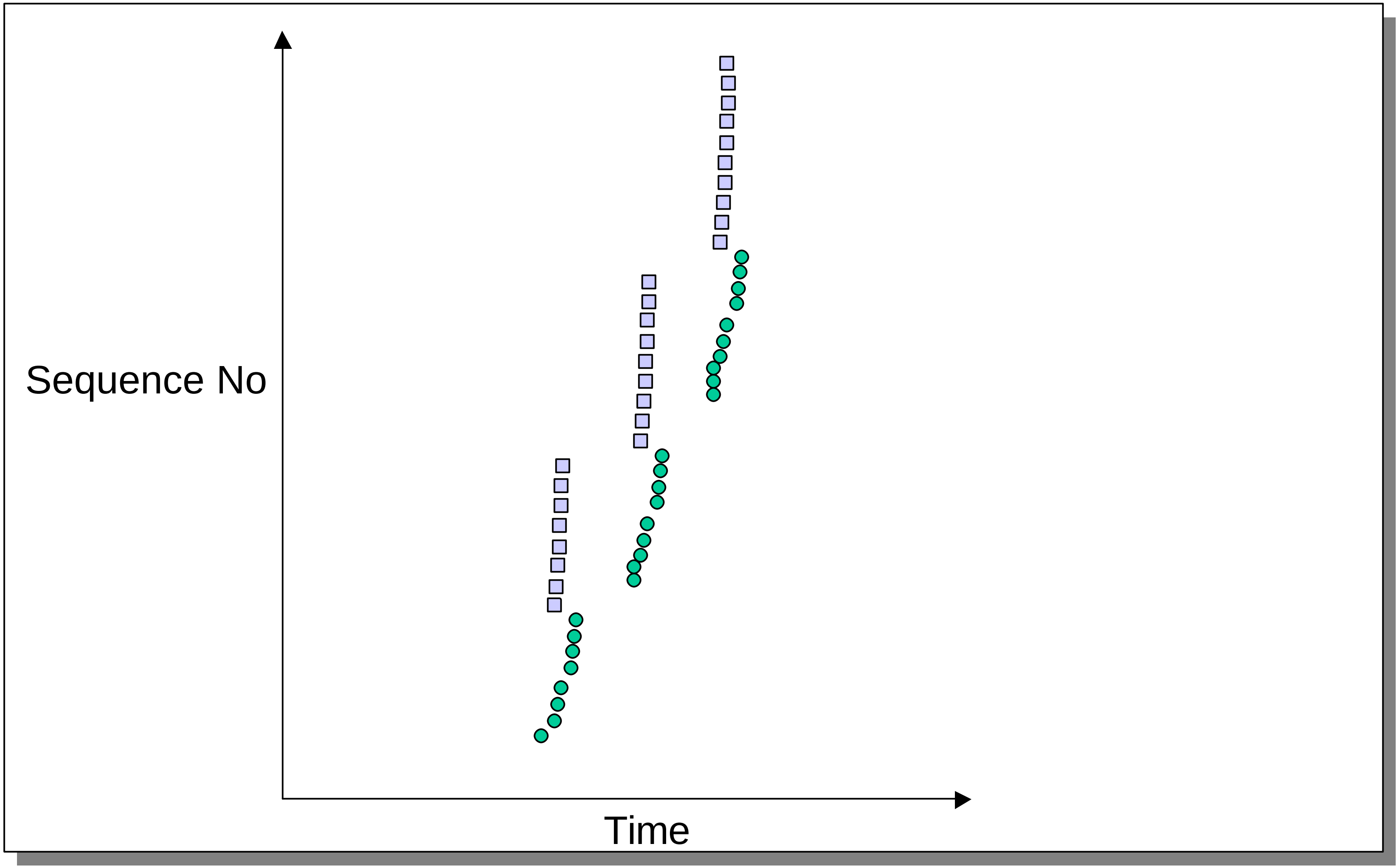


TCP Reno halves cwnd and skips slowstart after three duplicate ACKs
"Fast Recovery" mechanism => more later

TCP congestion avoidance

- Loss implies congestion – why?
  - Not necessarily true on all link types
- If loss occurs when cwnd = W
  - Network can handle 0.5W ~ W segments
  - Set ssthresh to 0.5W and slow-start from cwnd=1
- Upon receiving ACK with cwnd > ssthresh
  - Increase cwnd by 1/cwnd
  - Results in additive increase

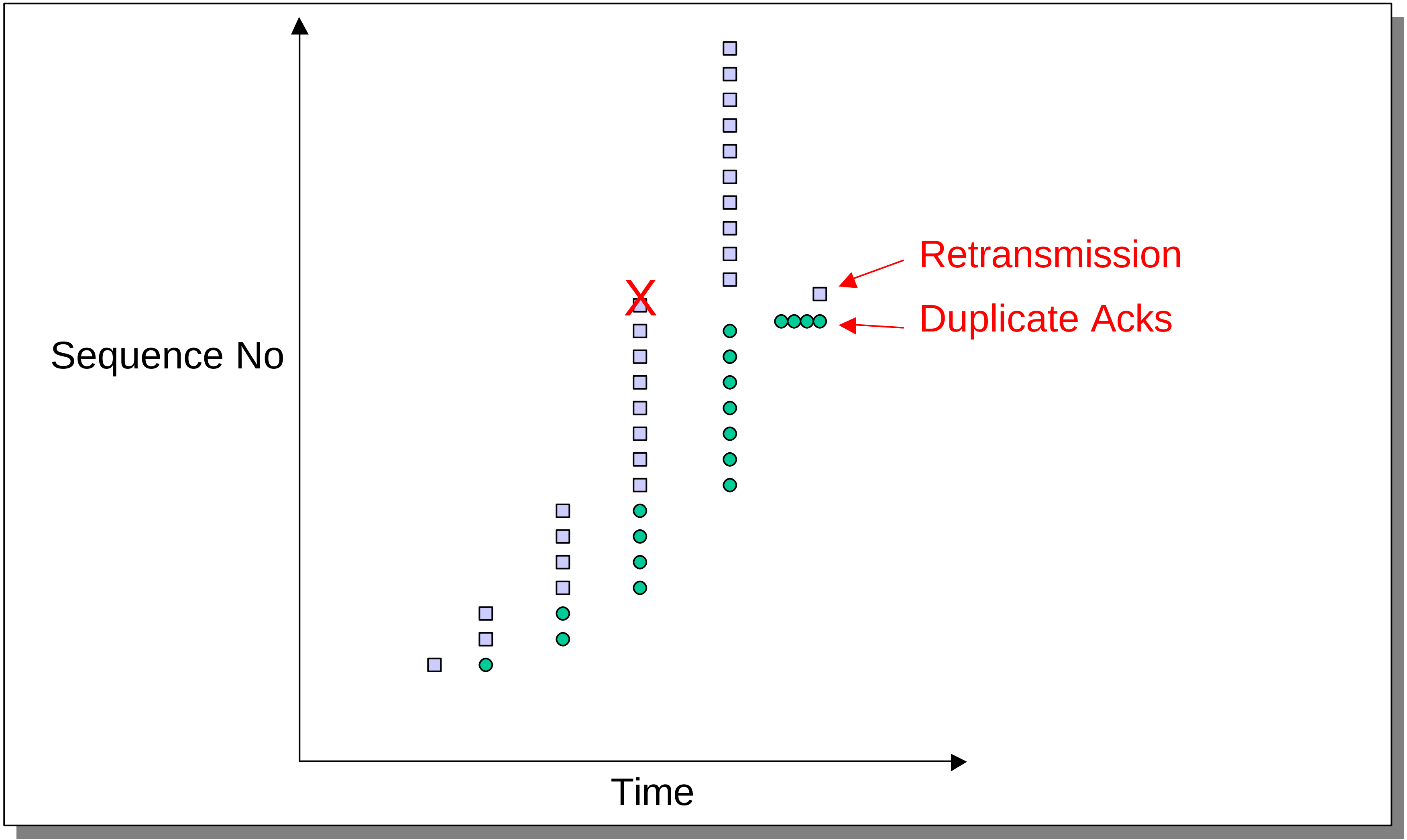
TCP congestion avoidance plot



TCP fast retransmit

- Timeouts (see previous)
- Duplicate acknowledgements (dupacks)
  - Repeated acks for the same sequence number
  - When can duplicate acks occur?
    - Loss
    - Packet re-ordering
    - Window update – advertisement of new flow control window
- Fast retransmit
  - Assume re-ordering is infrequent and not of large magnitude
  - Use receipt of 3 or more duplicate acks as indication of loss
  - Don’t wait for timeout to retransmit packet

TCP fast retransmit



TCP fast recovery

- Skip slow start
- After 3 dup ACKs:
  - **`CongWin`** is cut in half
  - window then grows linearly
- But after timeout event:
  - **`CongWin`** instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

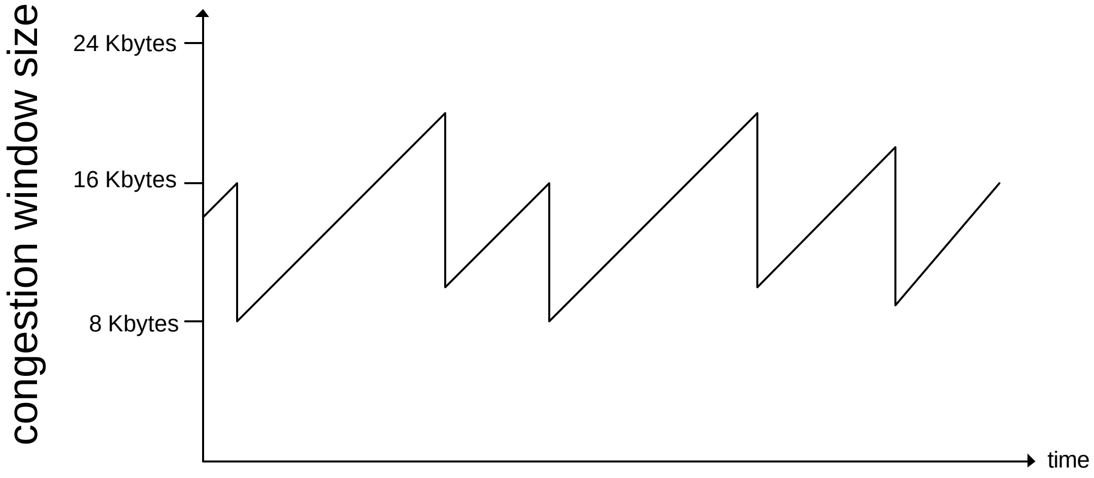
Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a "more alarming" congestion scenario

TCP fast retransmit & recovery (Reno)

- Combining congestion avoidance, fast retrasmit, and fast recovery gives....
  - *additive increase:* increase **CongWin** by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut **CongWin** in half after loss

Saw tooth behavior: probing for bandwidth



Interaction of flow and congestion control

- Sender's max window
  - min (advertised window, congestion window)
  - Question:
    - Can flow control mechanisms interact poorly with congestion control mechanisms?
  - Answer:
    - Yes.....Delayed acknowledgements and congestion windows
- Delayed Acknowledgements
  - TCP congestion control triggered by acks
    - If receive half as many acks -> window grows half as fast
  - Slow start with window = 1
    - Will trigger delayed ack timer
    - First exchange will take at least 200ms
    - Start with > 1 initial window
      - Bug in BSD, now a "feature"/standard

Summary: TCP Congestion Control

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

TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to “Congestion Avoidance”	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT

SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2,
		CongWin = Threshold,
		Set state to “Congestion Avoidance”
			Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.

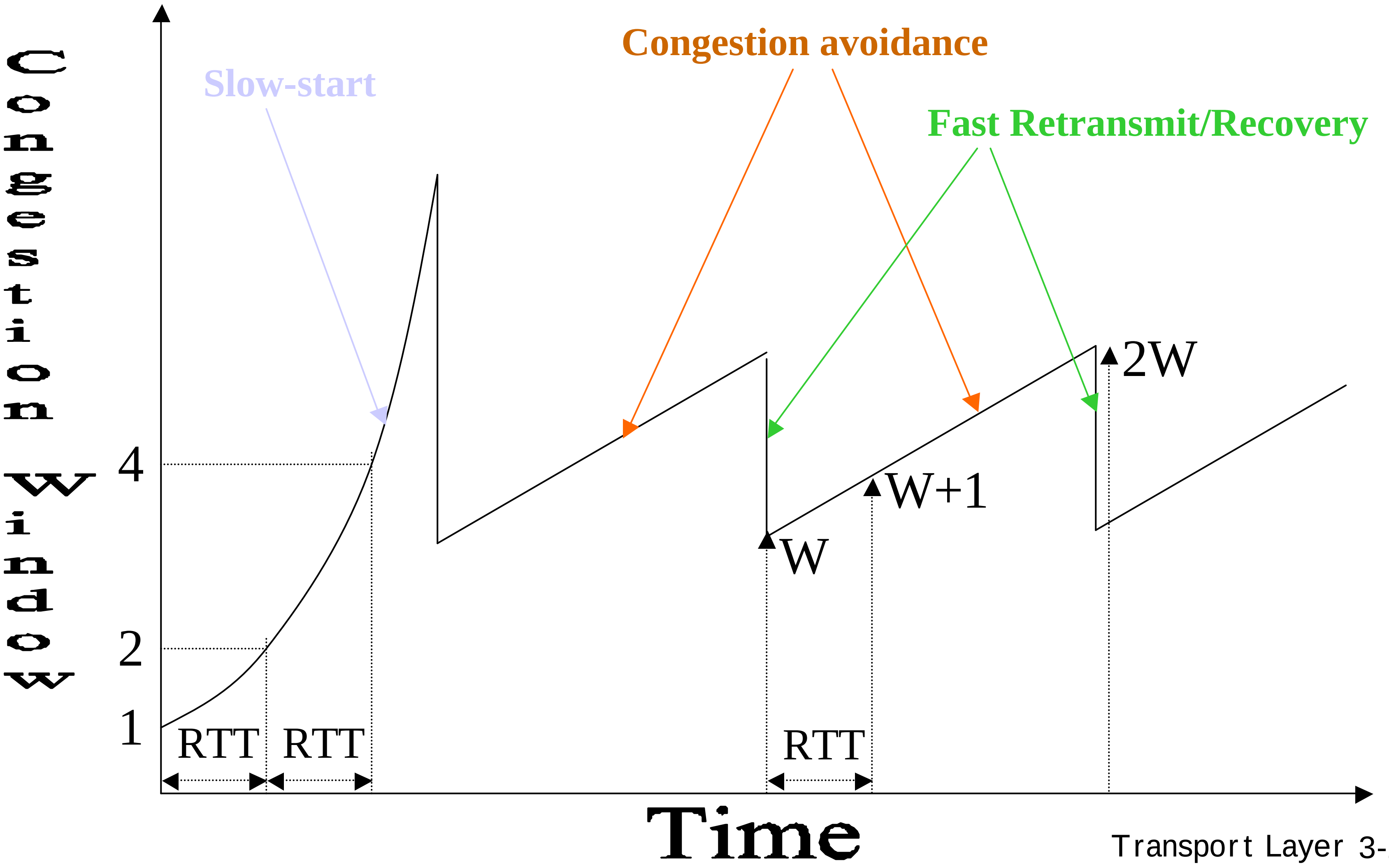
SS or CA	Timeout	Threshold = CongWin/2,
		CongWin = 1 MSS,
		Set state to “Slow Start”
			Enter slow start

SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP throughput

- What's the average throughout of TCP as a function of window size and RTT?
  - Ignore slow start
- Let 2W be the window size when loss occurs.
- When window is 2W, throughput is 2W/RTT
- Just after loss, window drops to W, throughput to W/RTT.
- Average throughout: 1.5W/RTT

TCP throughput



TCP Futures

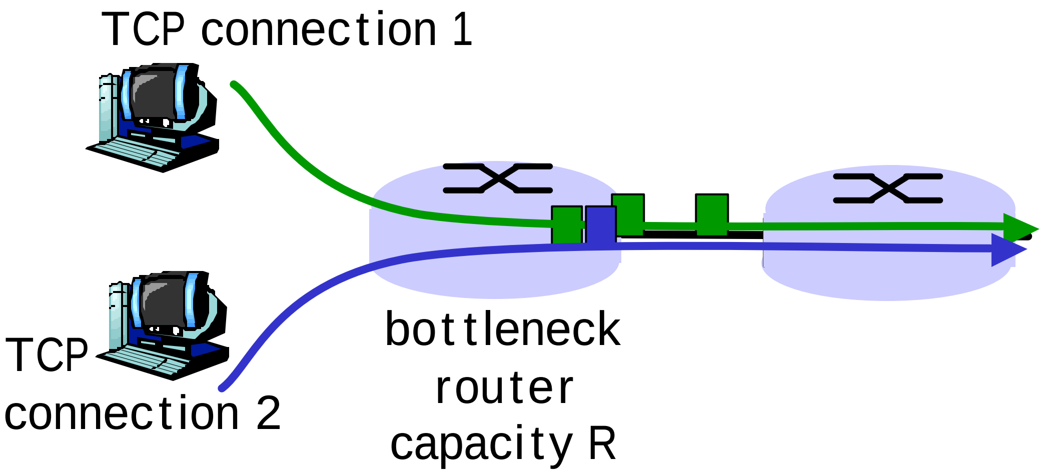
- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
  - BW*Delay = 10Gbs * 0.1s = 1Gbit
    - In bytes, 1Gbit/8 = 125MB
    - In packets 1Gbit/(8*1500) = 83,333 segments
      - W = 83,333 in-flight segments
  - Advertised window => 16 bits given in bytes!
    - Maximum of 64KB !!

TCP Futures

- Throughput
  - Sawtooth length = W*RTT
  - Packets xferred in sawtooth
    - W + (W+1) + (W+2) .... + 2W = (3W/2) * (W+1) = 1.5W(W+1)
    - For W=83,333
      - Packets xferred in sawtooth between losses = 10.4 billion
- Loss rate
  - 1 packet loss per sawtooth
    - ? L = 10^{-10} *Wow*
- New versions of TCP for high-speed needed!

TCP Fairness

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



Basic Control Model

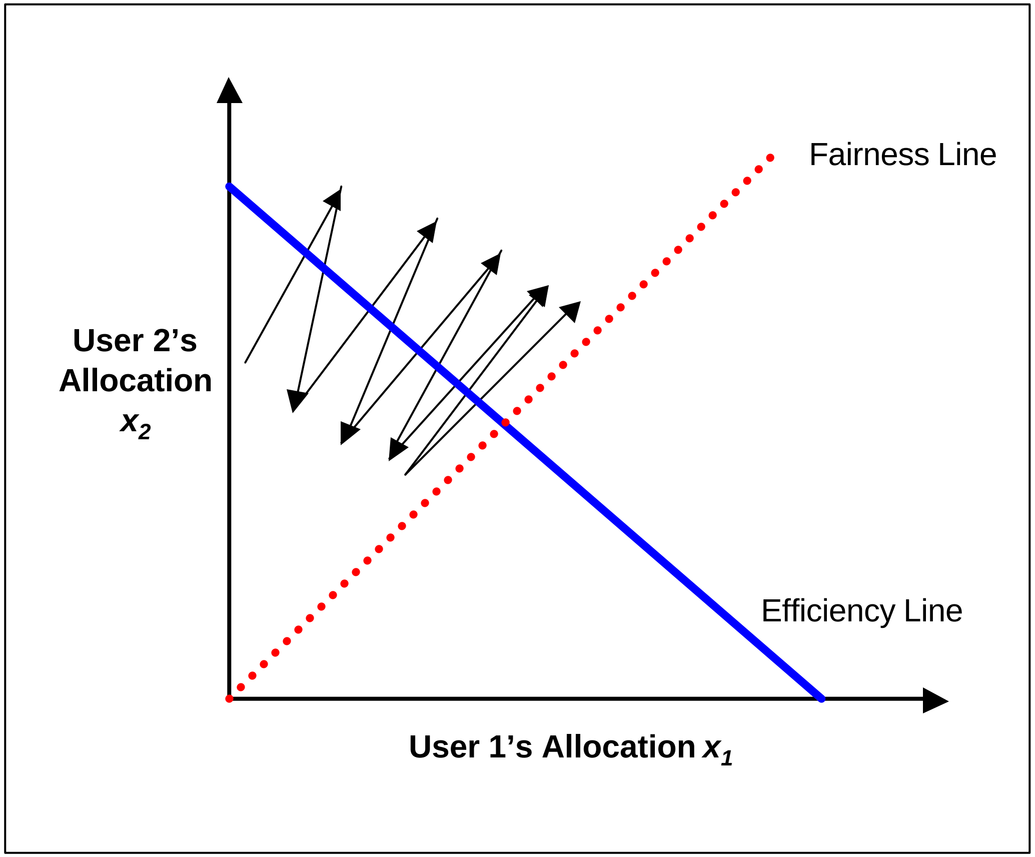
- Does TCP's congestion control algorithm promote fairness between flows?

Linear Control

- Many different possibilities for reaction to congestion and probing
  - Examine simple linear controls
  - Window(t + 1) = a + b Window(t)
  - Different a_i/b_i for increase and a_d/b_d for decrease
- Supports various reaction to signals
  - Increase/decrease additively
  - Increase/decrease multiplicatively
  - Which of the four combinations is optimal?

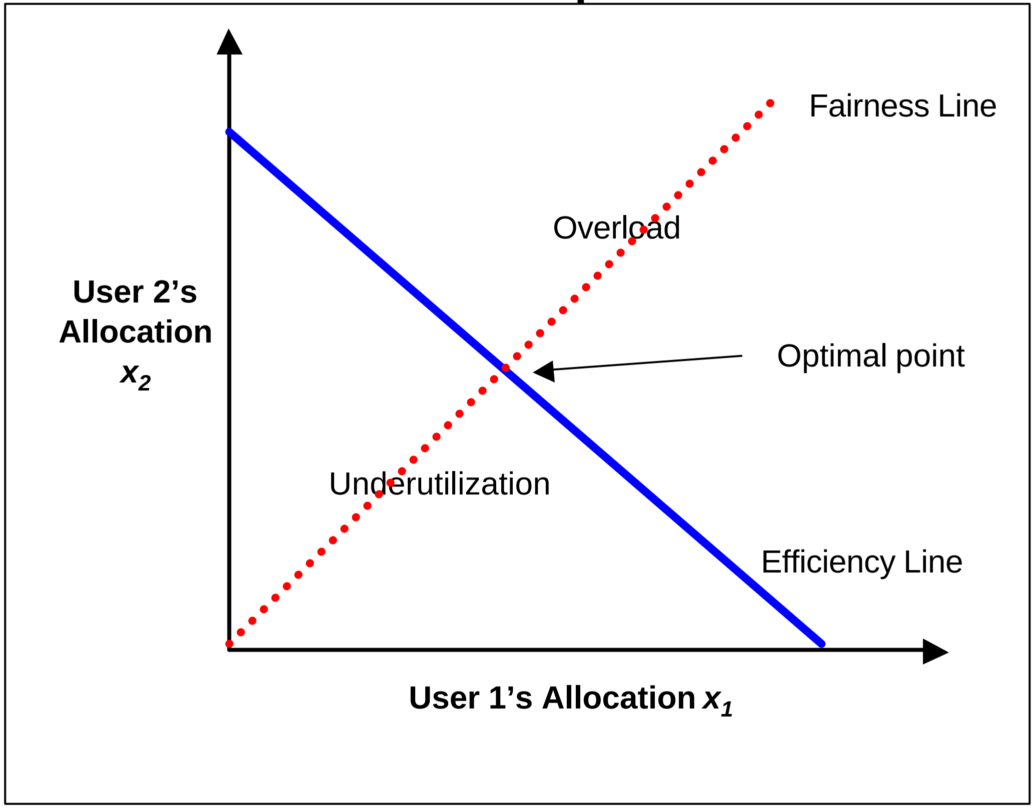
Phase plots

- Simple way to visualize behavior of competing connections over time



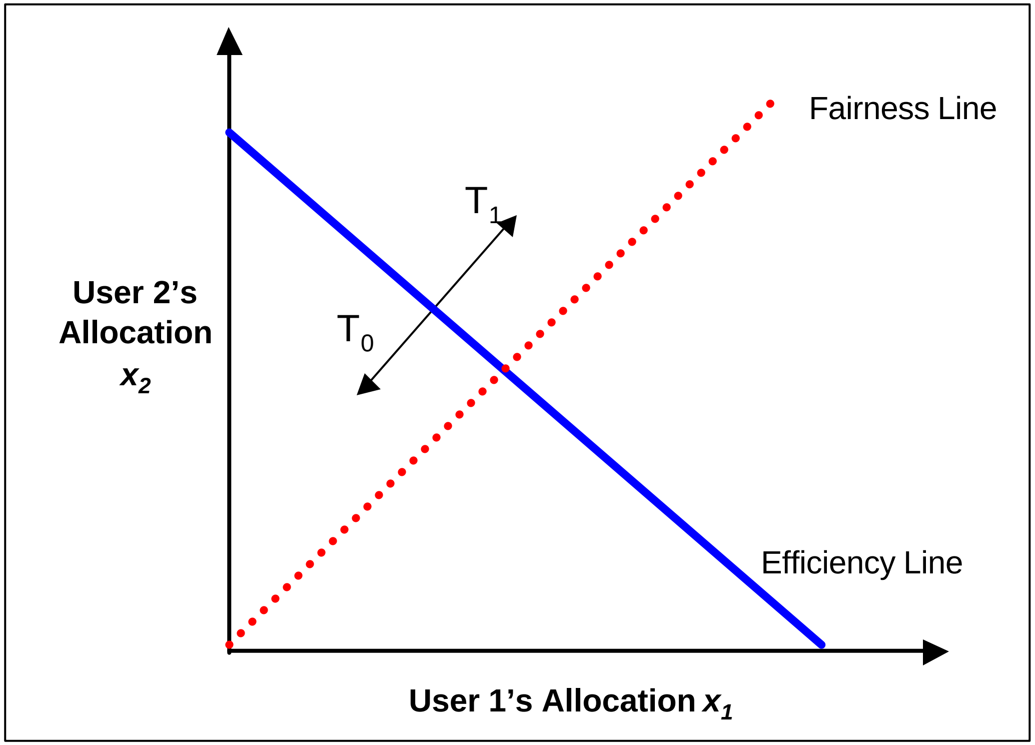
Phase plots

- What are desirable properties?
- What if flows are not equal?



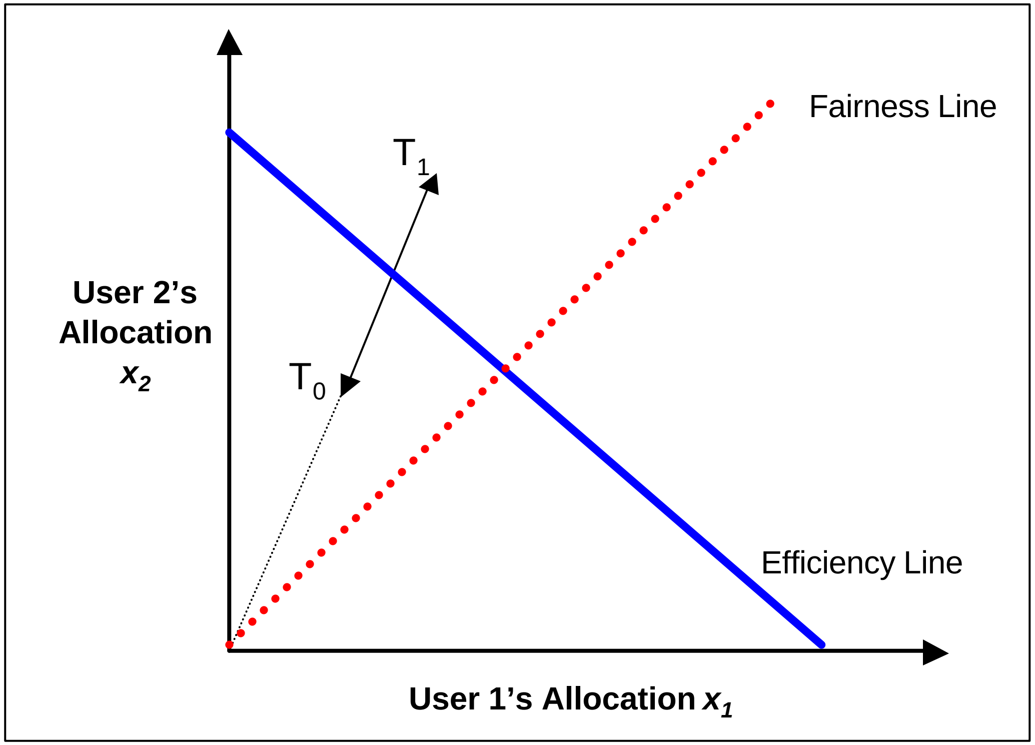
Additive Increase/Decrease

- Both X_1 and X_2 increase/decrease by the same amount over time
  - Additive increase improves fairness and additive decrease reduces fairness



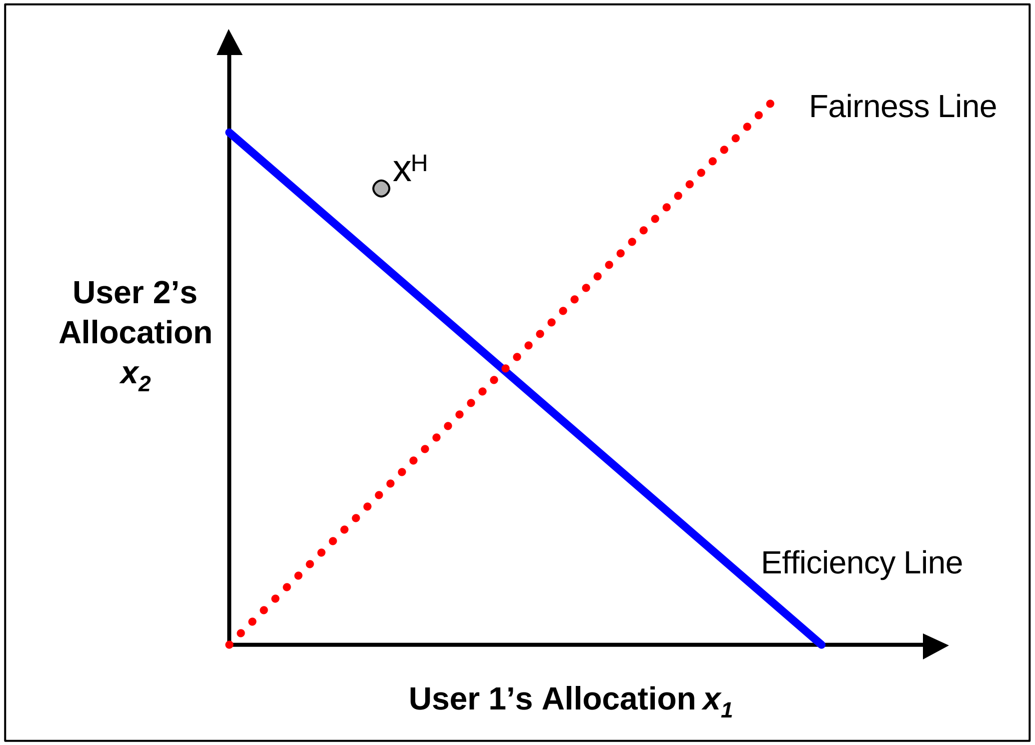
Multiplicative Increase/Decrease

- Both X_1 and X_2 increase by the same factor over time
  - Extension from origin – constant fairness



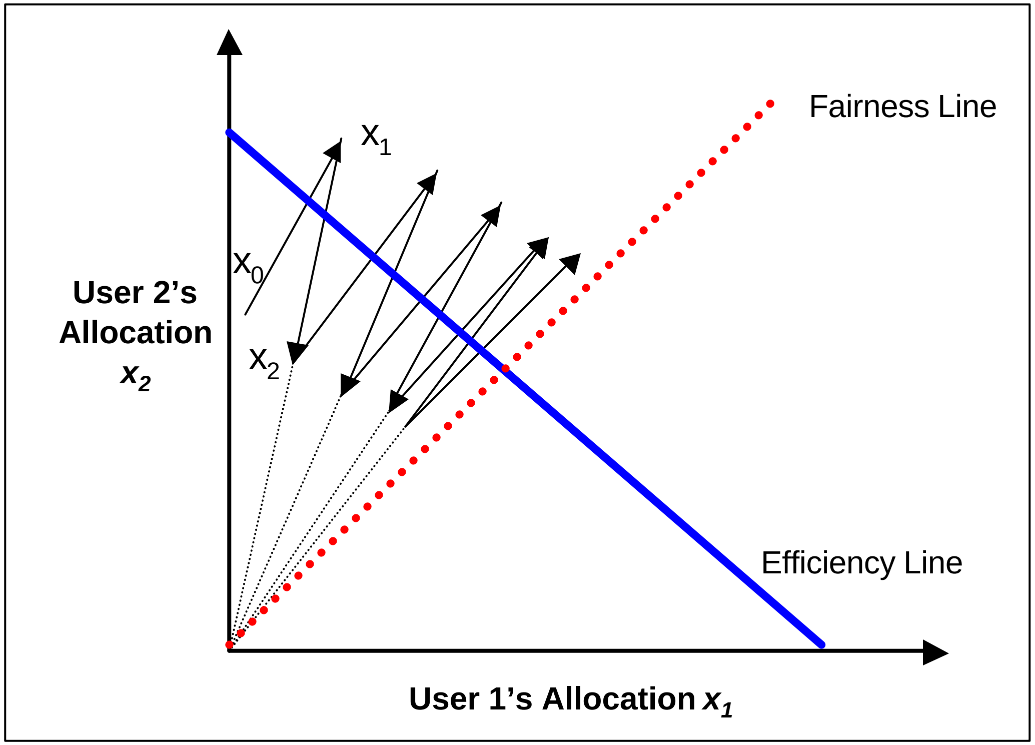
Convergence to Efficiency & Fairness

- From any point, want to converge quickly to intersection of fairness and efficiency lines



What is the Right Choice?

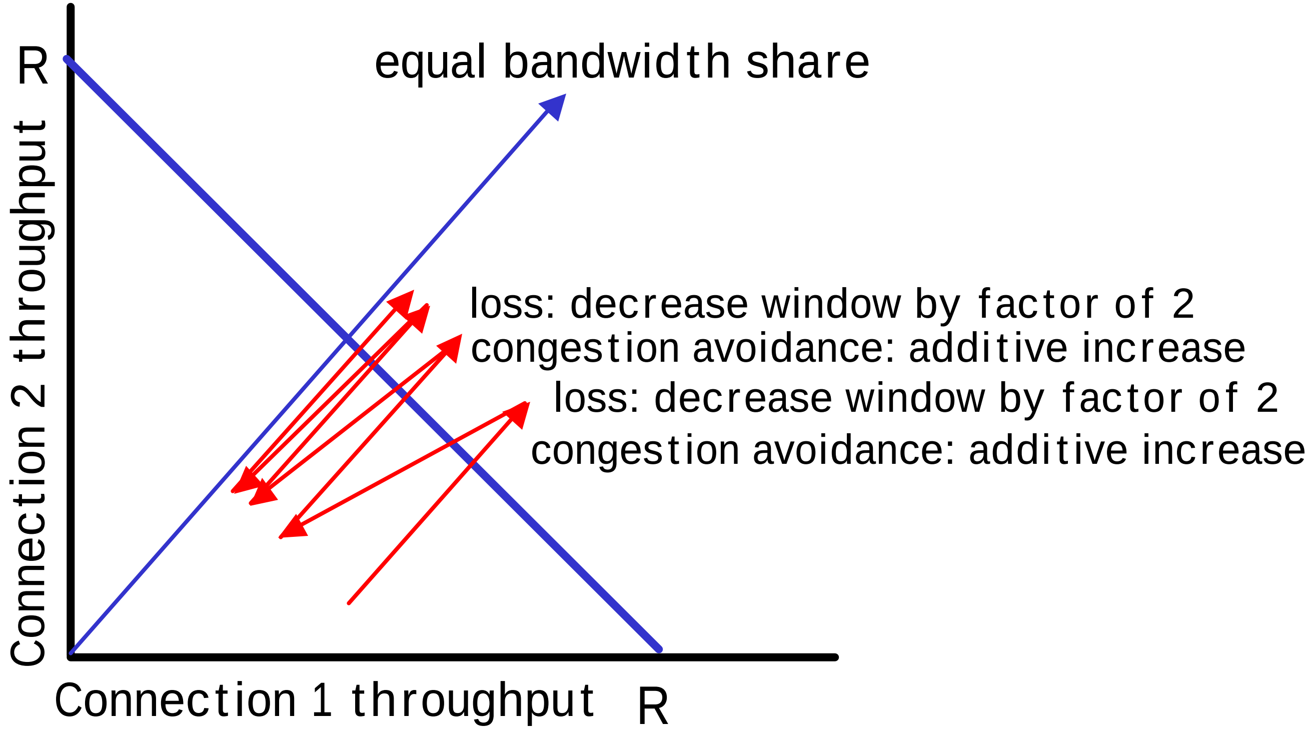
- Constraints limit us to AIMD
  - AIMD moves towards optimal point



Why is TCP fair?

Two competing sessions:

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


R
equal bandwidth share
loss: decrease window by factor of 2
congestion avoidance: additive increase
loss: decrease window by factor of 2
congestion avoidance: additive increase
Connection 2 throughput
Connection 1 throughput
R

Fairness (more)

Fairness and UDP

- Multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

Fairness and parallel TCP connections

- nothing prevents app from opening parallel connections between 2 hosts.
- Web browsers do this
- Example: link of rate R supporting 9 cnctions;
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2 !

Advanced transport topics

- Ambiguous acknowledgements
  - TCP SACK (Selective acknowledgements)
- Redundant header fields
  - Many header fields fixed or change slightly
    - TCP header compression
    - Compress header to save bandwidth
- RTT ambiguity for retransmitted packets
  - TCP timestamp option
  - Sender puts timestamp in packet that receiver echoes
- Sequence number wraparound
  - 32-bit sequence/ack # wraps around
  - 10Mbs: 57 min., 100Mbs: 6 min., 622Mbs: 55 sec. < MSL!
  - Use timestamp option to disambiguate
  - TCP sequence number wraparound (TCP PAWS)

Advanced transport topics

- Long, fat pipes
  - 16-bit advertised window can't support large bandwidth*delay networks
  - For 100ms network, need 122KB for 10Mbs (16-bit window = 64KB)
  - 1.2MB for 100Mbs, 7.4MB for 622Mbs
  - TCP window scaling option
    - Scaling factor on advertised window specifies # of bits to shift to the left
    - Scaling factor exchanged during connection setup
- Non-responsive, aggressive applications
  - Applications written to take advantage of network resources (multiple TCP connections)
  - Network-level enforcement, end-host enforcement of fairness

Advanced transport topics

- Asymmetric pipes
  - TCP over highly asymmetric links is limited by ACK throughput (40 byte ack for every MTU-sized segment)
  - Coalesce multiple acknowledgements into single one
- Wireless networks
  - TCP infers loss on wireless links as congestion and backs off
  - Add link-layer retransmission and explicit loss notification (to squelch RTO)
- Short transfers slow
  - Flows timeout on loss if cwnd < 3
    - Change dupack threshold for small cwnd
  - 3-4 packet flows (most HTTP transfers) need 2-3 round-trips to complete
    - Use larger initial cwnd (IETF approved initial cwnd = 3 or 4)

Advanced transport topics

- Congestion information sharing
  - Individual connections each probe for bandwidth (to set ssthresh)
  - Share information between connections on same machine or nearby machines (SPAND, Congestion Manager)
- Non-TCP traffic
  - Multimedia applications do not work well over TCP's sawtooth
  - TCP-friendly rate control
  - Derive smooth, stable equilibrium rate via equations based on loss rate
- Better congestion control algorithms
  - TCP Vegas
    - TCP increases rate until loss
    - Avoid losses by backing off sending rate when delays increase

Advanced transport topics

- ATM
  - TCP uses implicit information to fix sender's rate
  - Explicitly signal rate from network elements
- ECN
  - TCP uses packet loss as means for congestion control
  - Add bit in IP header to signal congestion (hybrid between TCP approach and ATM approach)
- Active queue management
  - Congestion signal the result of congestion not a signal of imminent congestion
  - Actively detect and signal congestion beforehand

Advanced transport topics

- Security
  - Layer underneath application layer and above transport layer (See Chapter 8)
  - SSL, TLS
  - Provides TCP/IP connection the following....
    - Data encryption
    - Server authentication
    - Message integrity
    - Optional client authentication
  - Original implementation: Secure Sockets Layer (SSL)
    - Netscape (circa 1994)
    - http://www.openssl.org/ for more information
    - Submitted to W3 and IETF
  - New version: Transport Layer Security (TLS)
    - http://www.ietf.org/html.charters/tls-charter.html

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

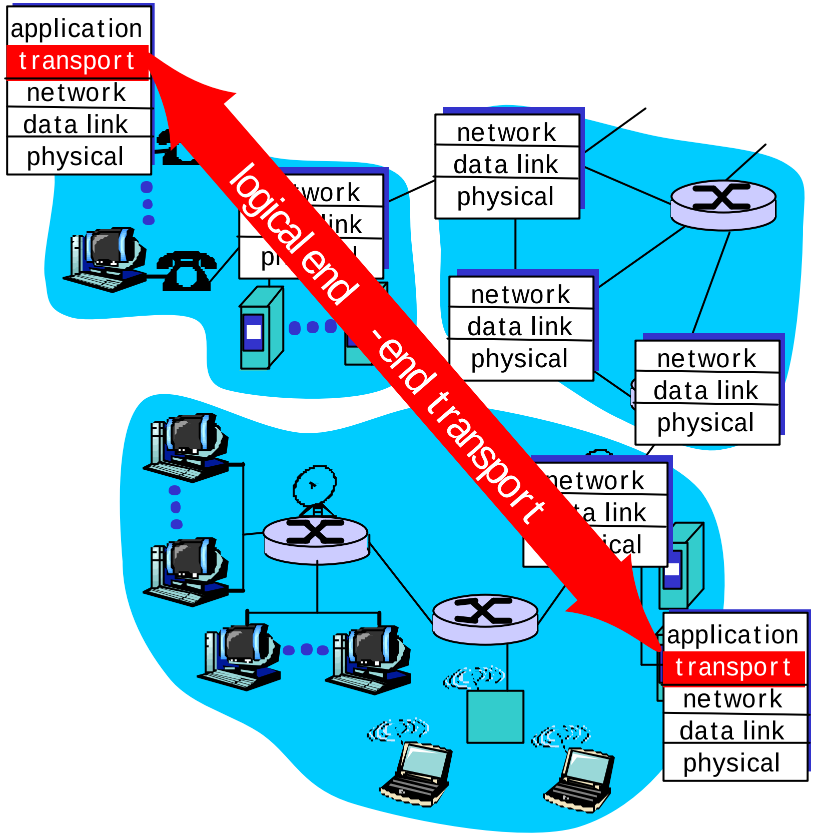
Next:

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

Extra slides

Internet transport-layer protocols

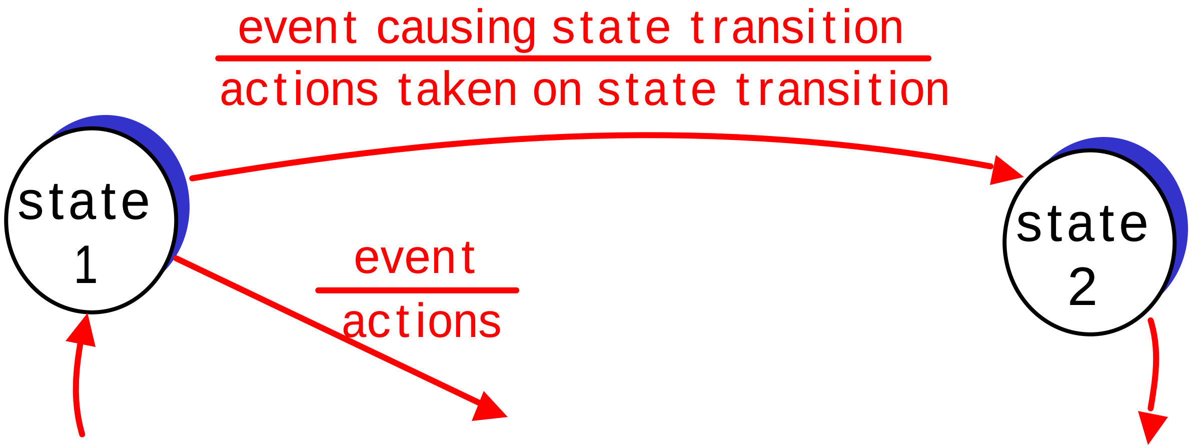
- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP



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

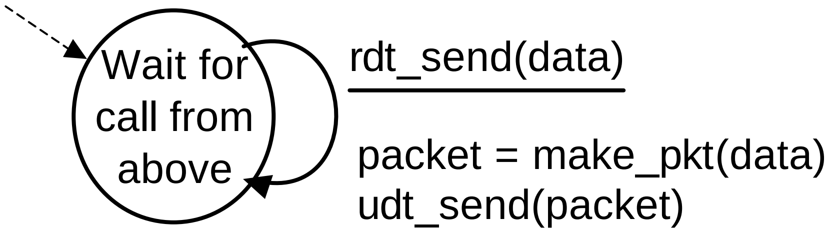


event causing state transition
actions taken on state transition
state 1
state 2
event
actions

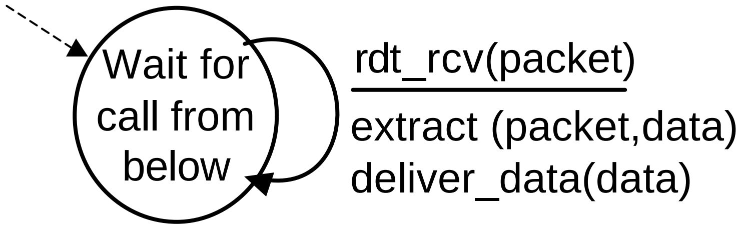
state: when in this "state" next state uniquely determined by next event

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

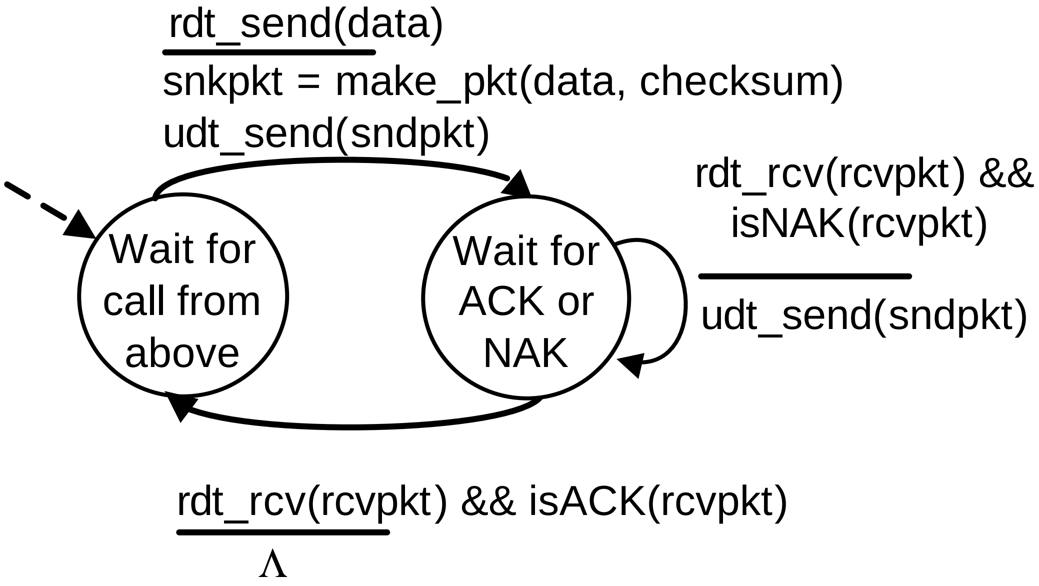


receiver

Rdt2.0: channel with bit errors

- 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



rdt_send(data)

snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)

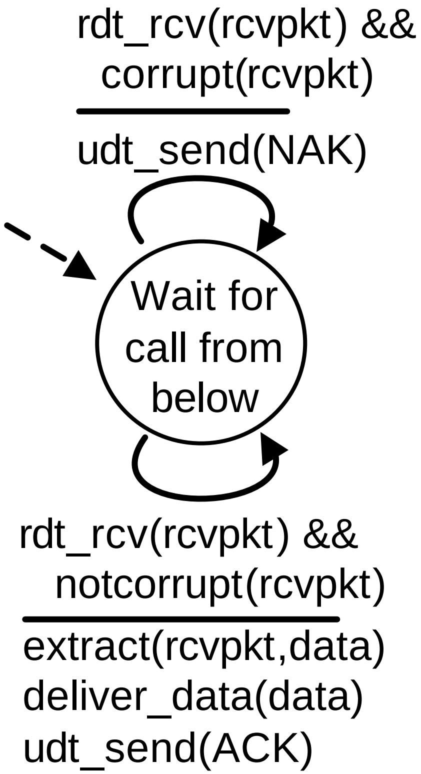
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

sender

receiver



rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)

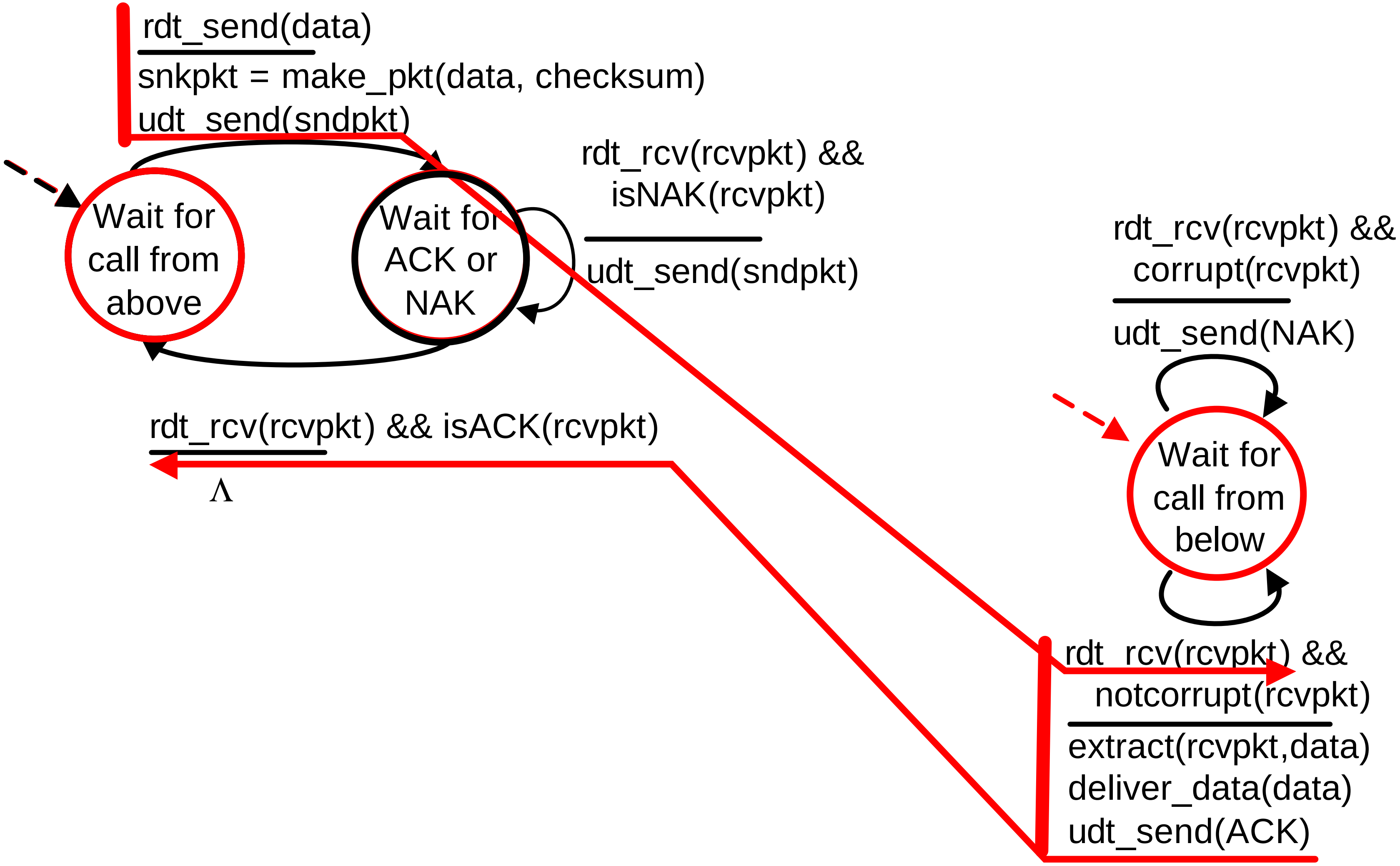
udt_send(NAK)

Wait for call from below

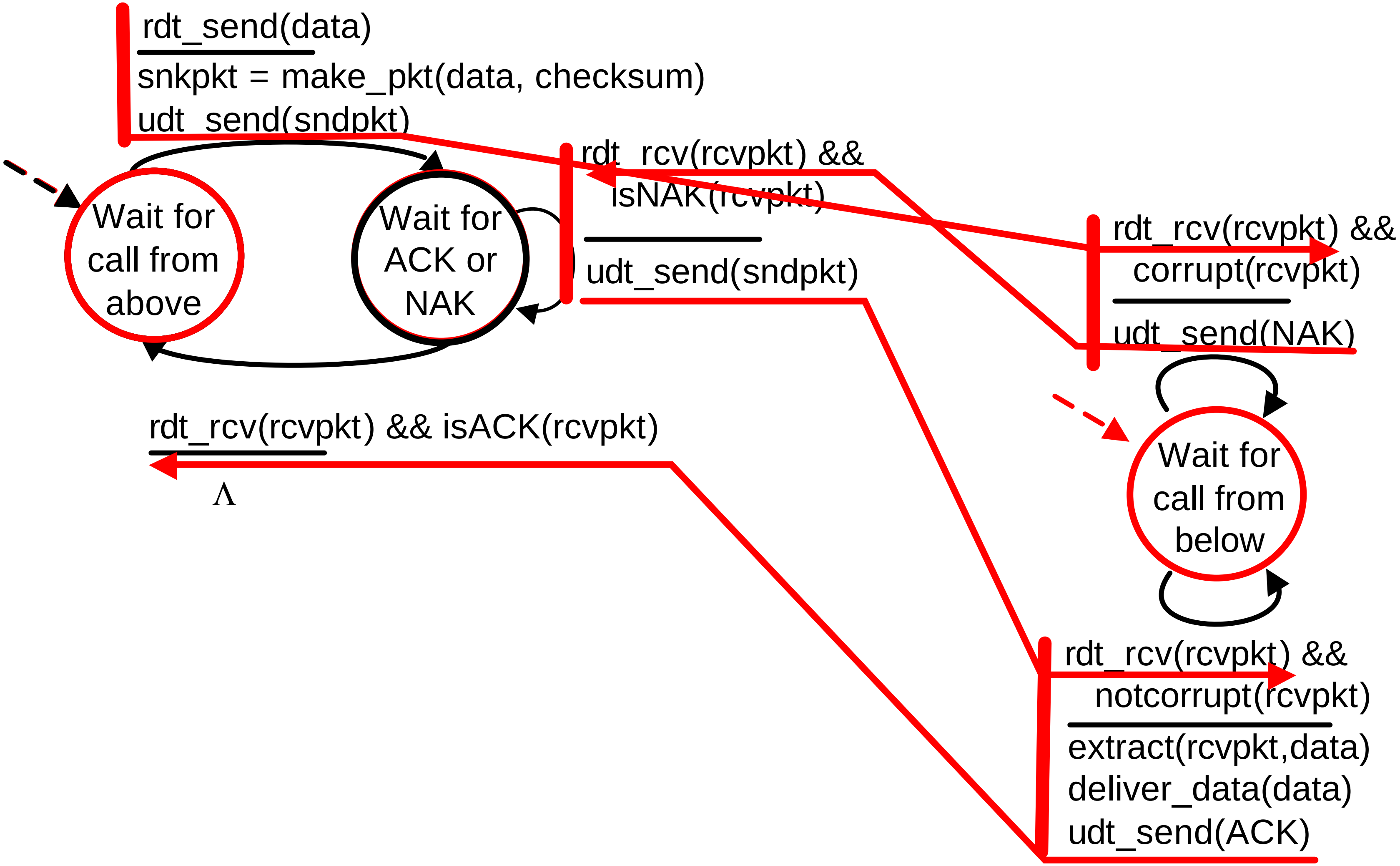
rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

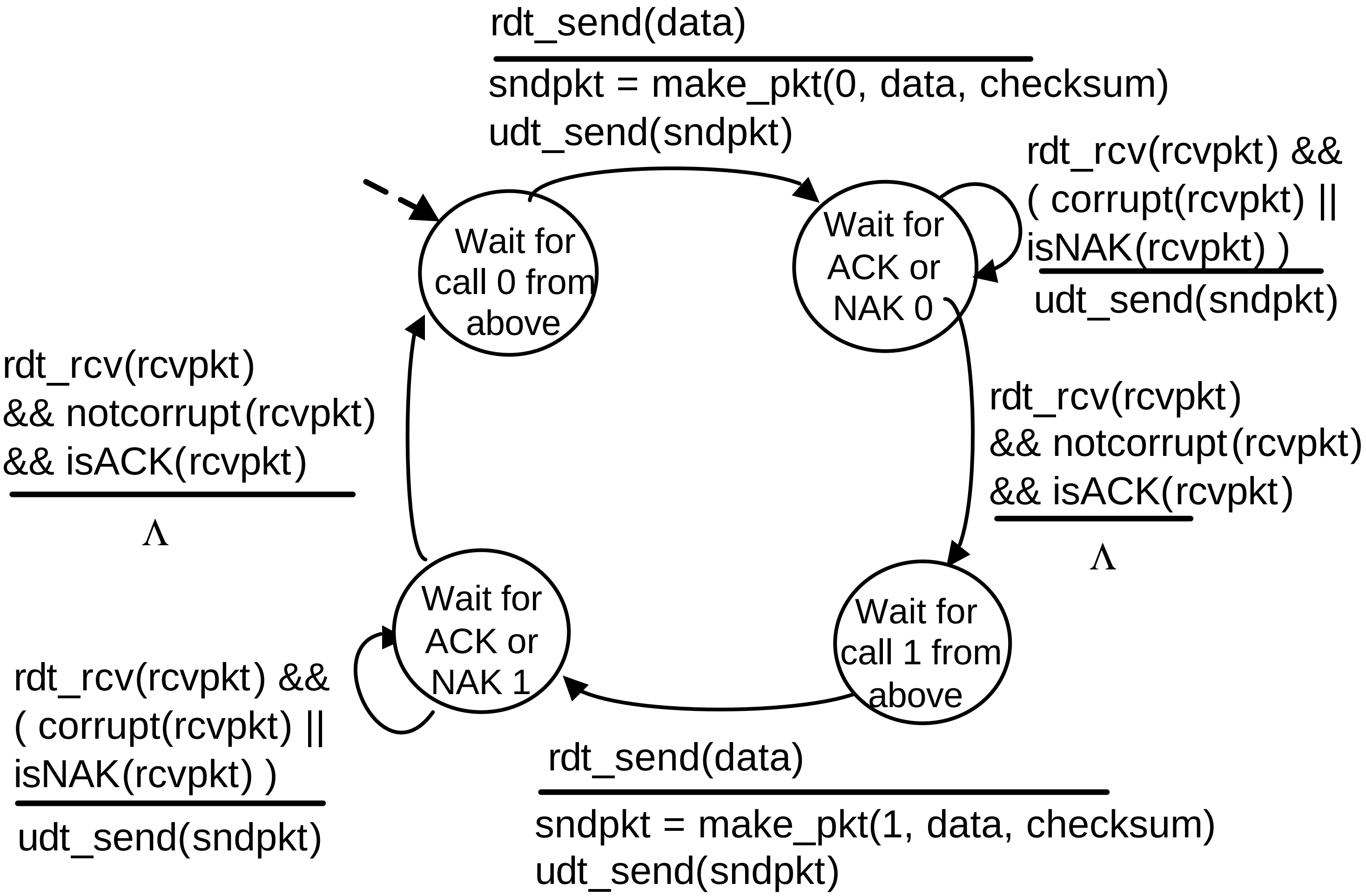
Handling duplicates:

- sender retransmits current pkt if ACK/NAK garbled
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

Sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs



Wait for call 0 from above

rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

Wait for ACK or NAK 0

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))

udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)

Λ

Wait for call 1 from above

rdt_send(data)

sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

Wait for ACK or NAK 1

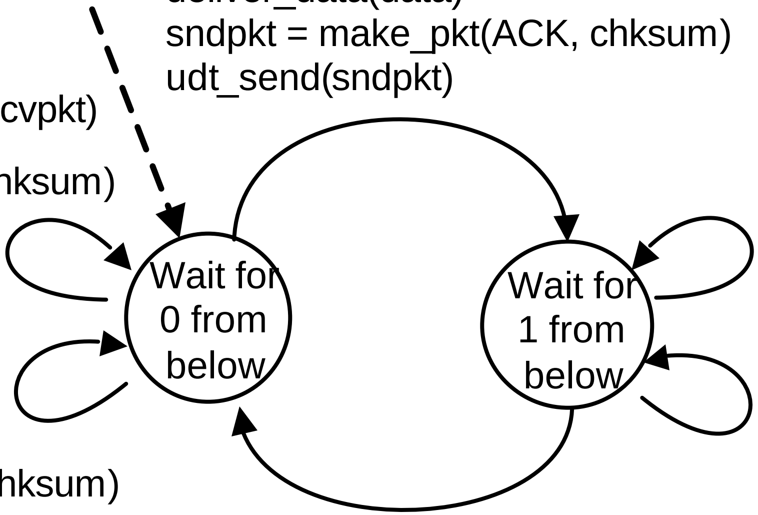
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))

udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)

Λ

rdt2.1: receiver, handles garbled ACK/NAKs



rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
 && has_seq0(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
 not corrupt(rcvpkt) &&
 has_seq1(rcvpkt)

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
 not corrupt(rcvpkt) &&
 has_seq0(rcvpkt)

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
 && has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt2.1: discussion

Sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "current" pkt has 0 or 1 seq. #

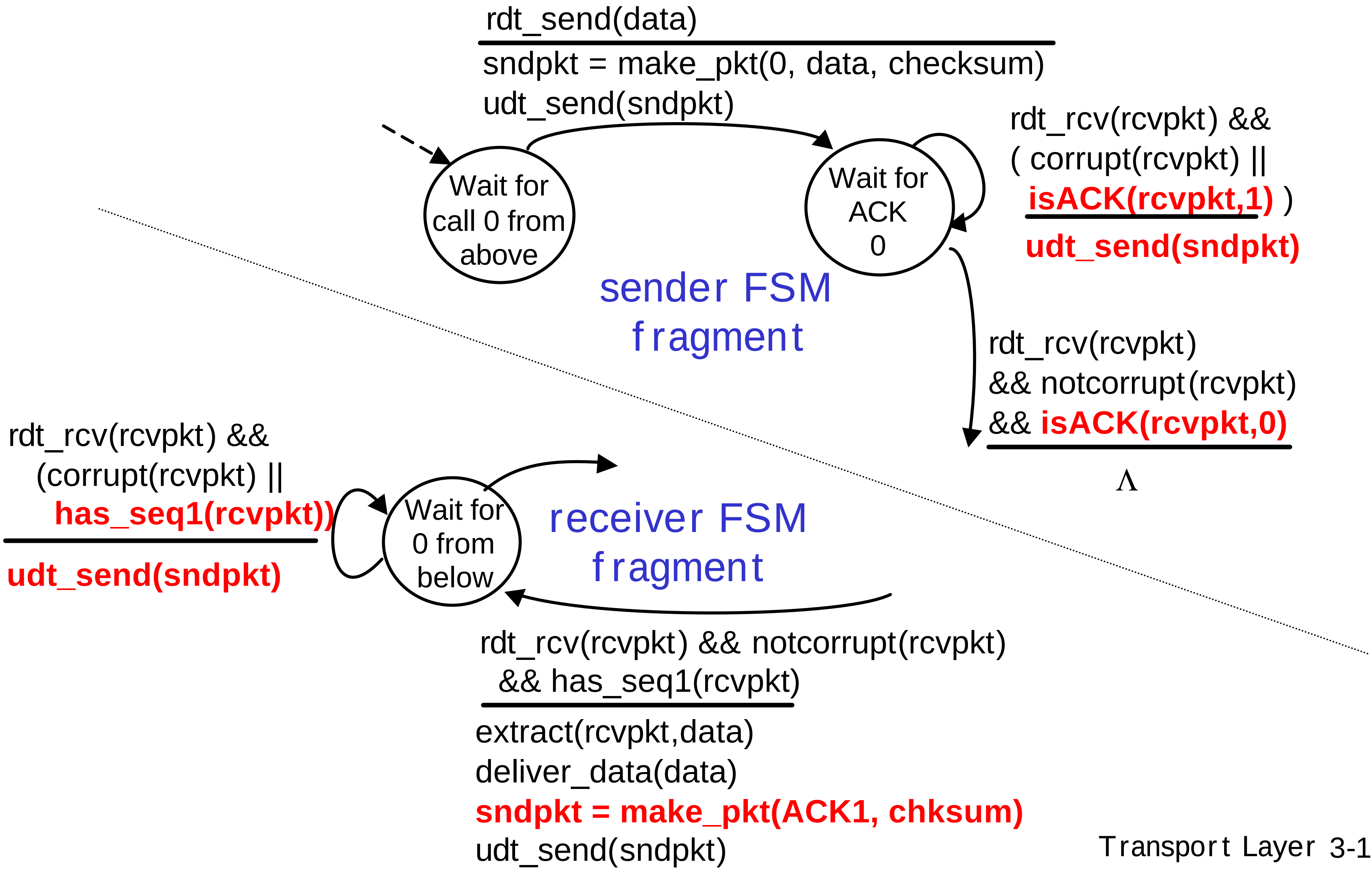
Receiver:

- 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

- 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



rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

Wait for call 0 from above

Wait for ACK 0

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))

udt_send(sndpkt)

sender FSM fragment

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& **isACK(rcvpkt,0)**

Λ

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
has_seq1(rcvpkt))

udt_send(sndpkt)

Wait for 0 from below

receiver FSM fragment

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)

rdt3.0: channels with errors *and* loss

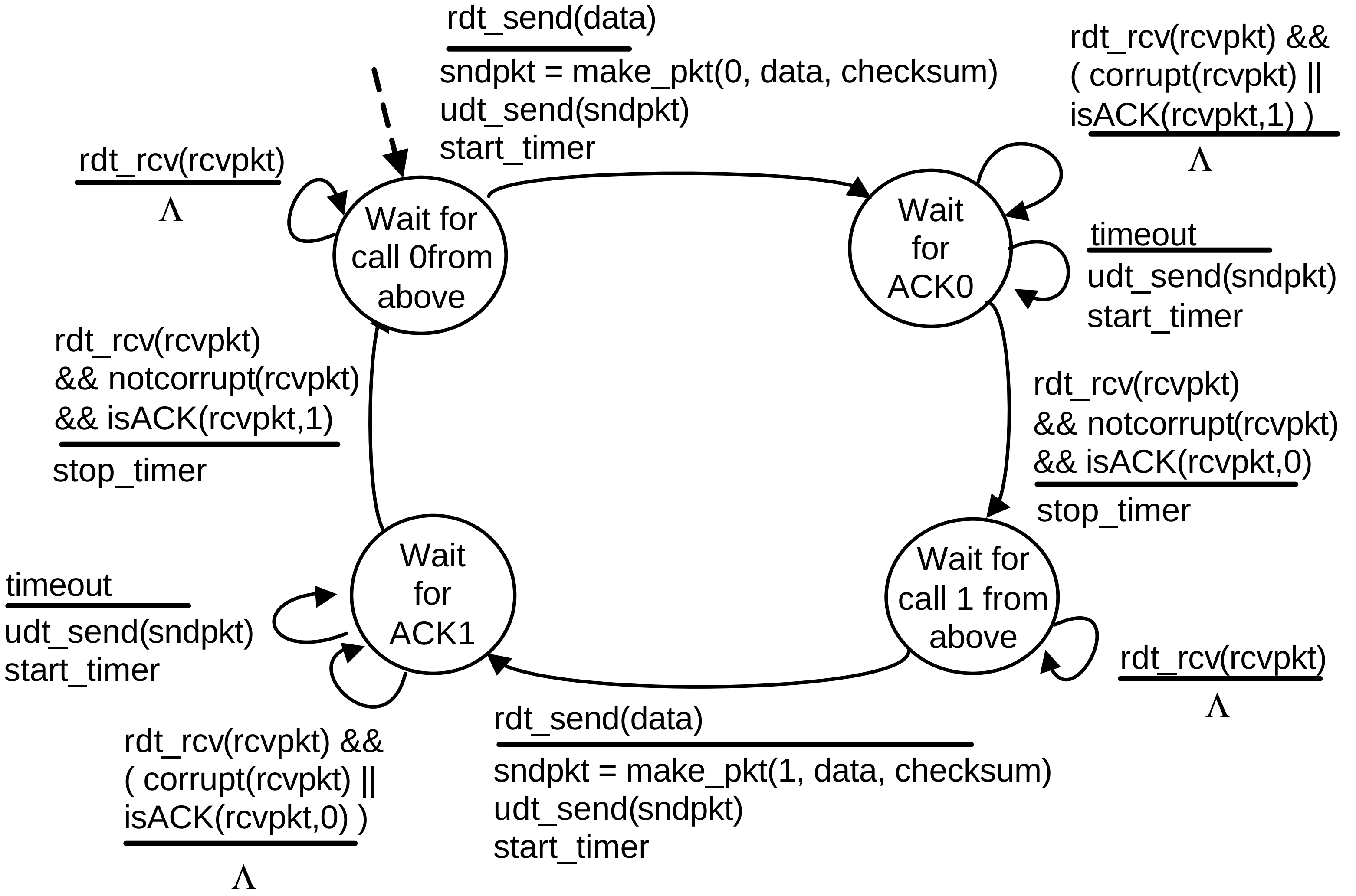
New assumption: underlying channel can also lose packets (data or ACKs)

- 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



rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)

Λ

Wait for call 0from above

Wait for ACK0

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))

Λ

timeout

udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)

stop_timer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)

stop_timer

Wait for ACK1

Wait for call 1 from above

timeout

udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)

Λ

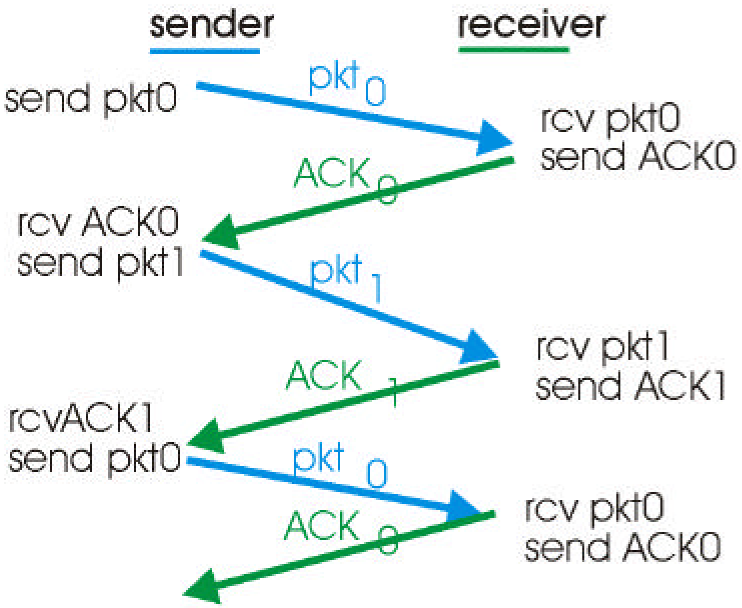
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))

Λ

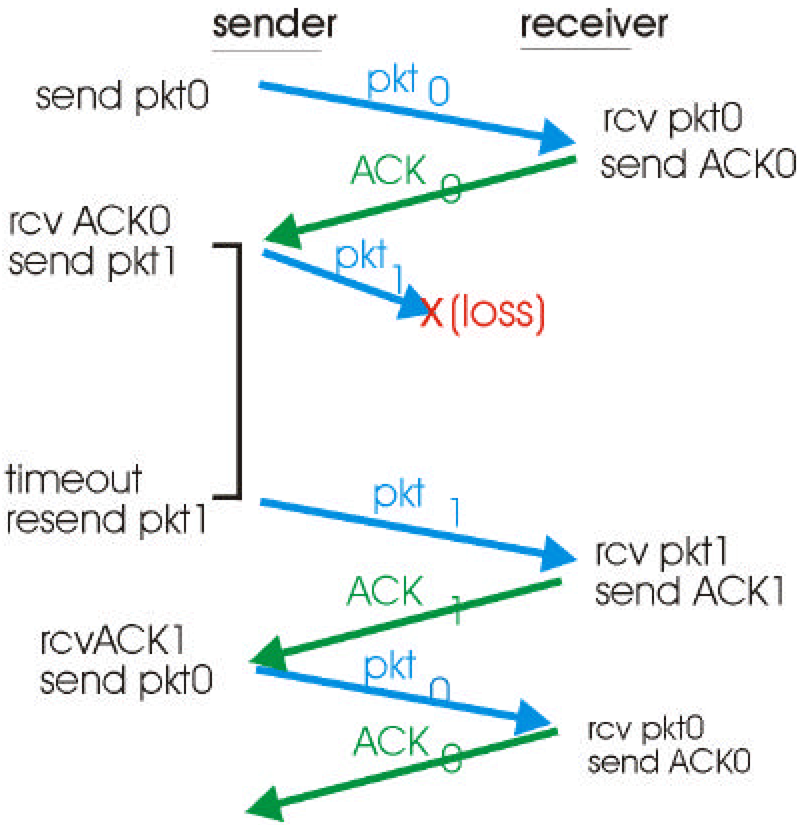
rdt_send(data)

sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

rdt3.0 in action

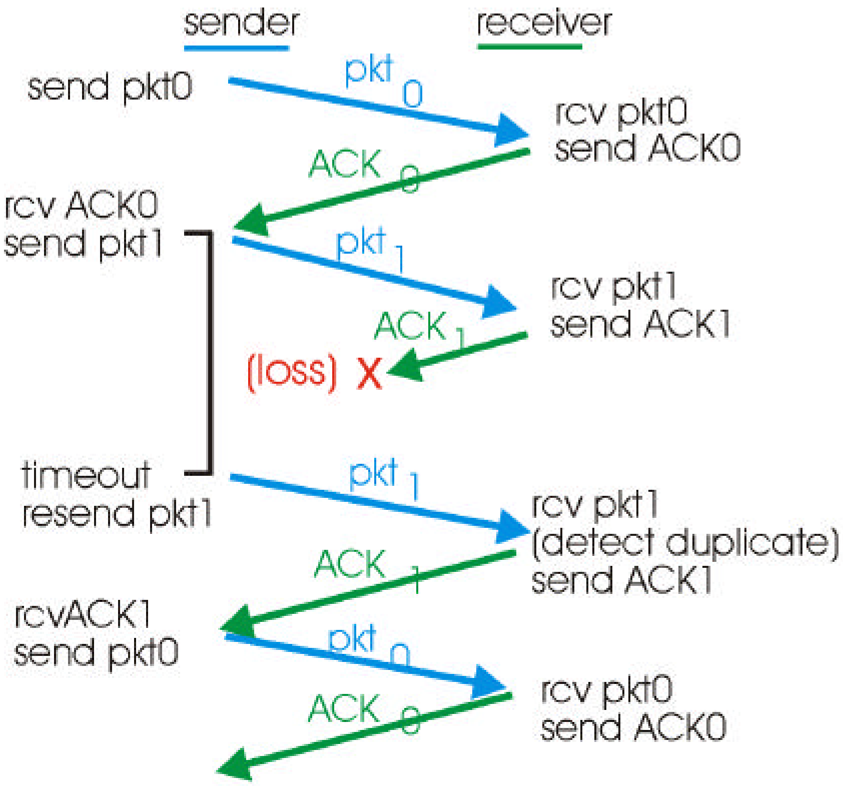


(a) operation with no loss

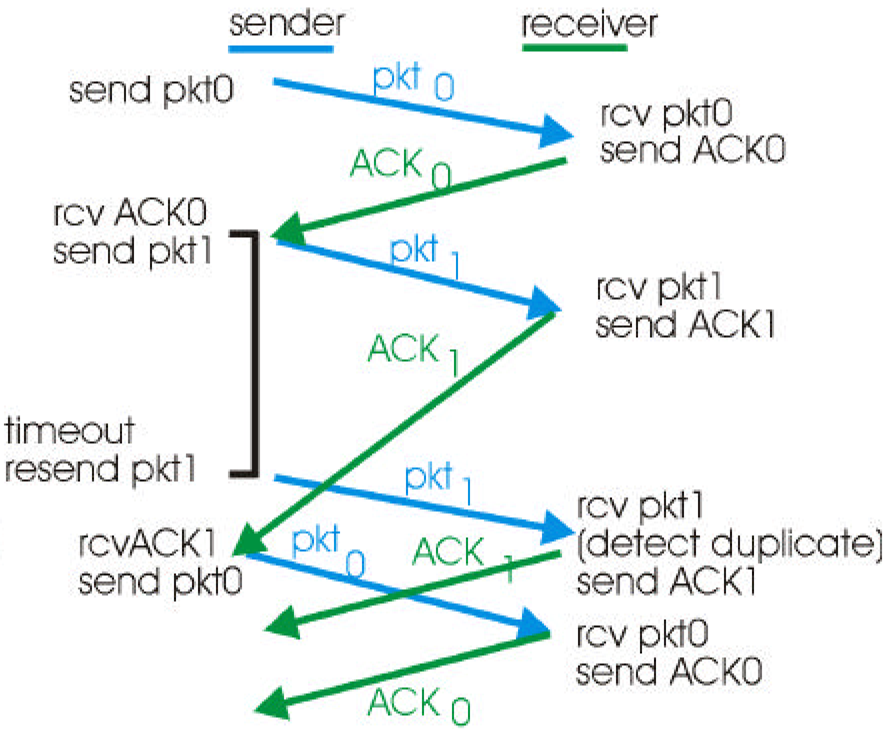


(b) lost packet

rdt3.0 in action



(c) lost ACK



(d) premature timeout

Performance of rdt3.0

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

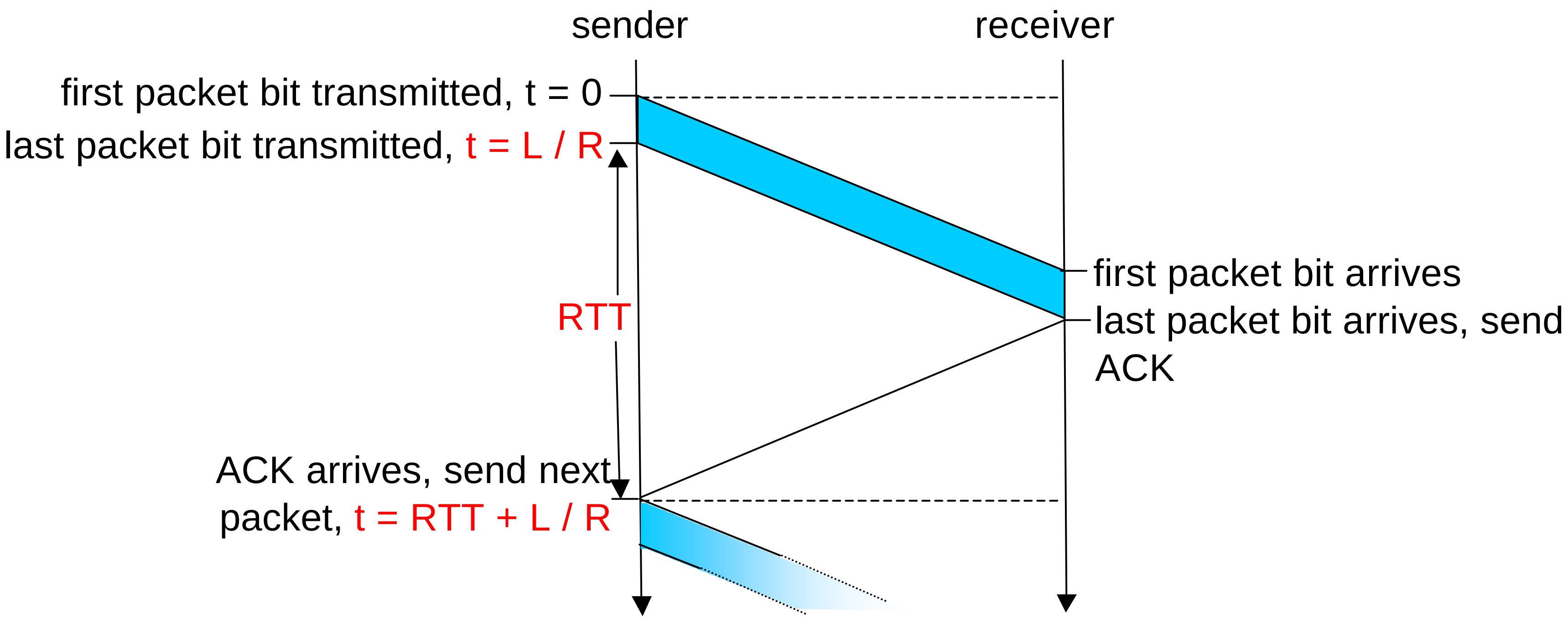
$$T_{\text{transmit}} = \frac{\text{L (packet length in bits)}}{\text{R (transmission rate, bps)}} = \frac{\text{8kb/pkt}}{\text{10**9 b/sec}} = \text{8 microsec}$$

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

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

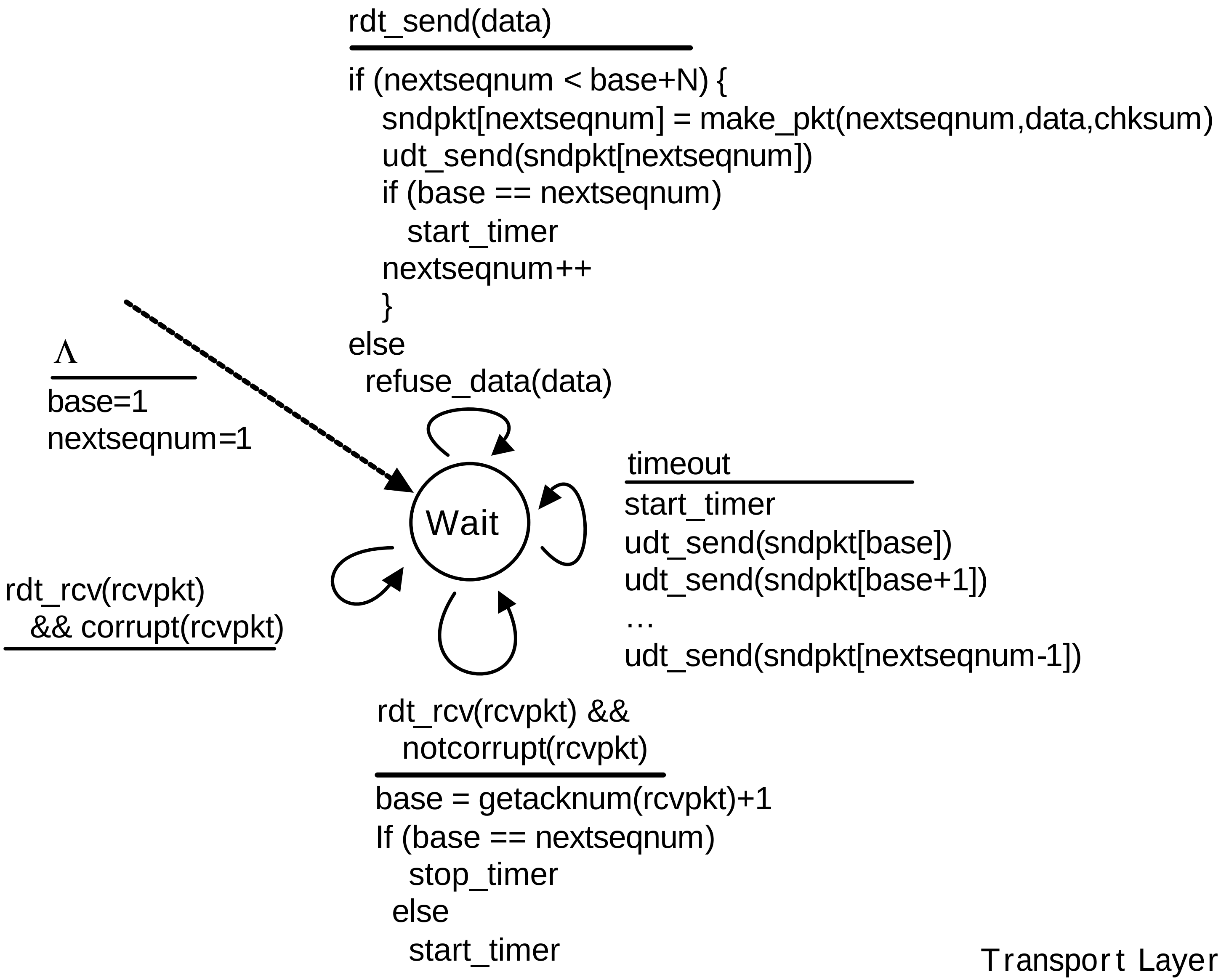
- 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation


sender
receiver
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

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

GBN: sender extended FSM



rdt_send(data)

```
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
       start_timer
    nextseqnum++
    }
else
  refuse_data(data)
```

Λ

```
base=1
nextseqnum=1
```

timeout

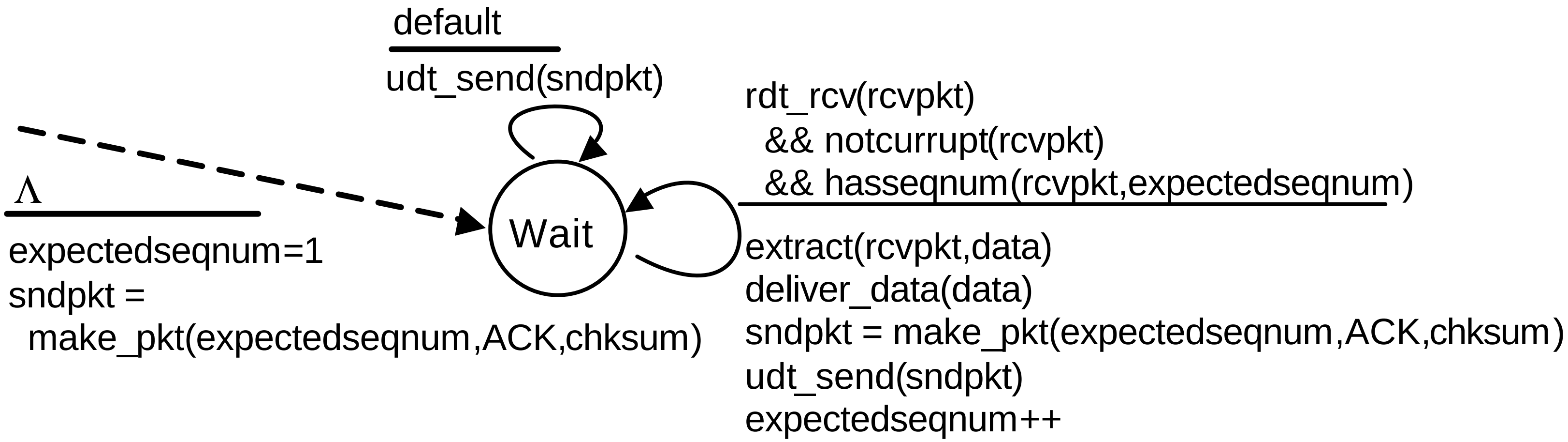
```
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)

```
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer
```

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 #

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

Approaches towards congestion control

Two broad approaches towards congestion control:

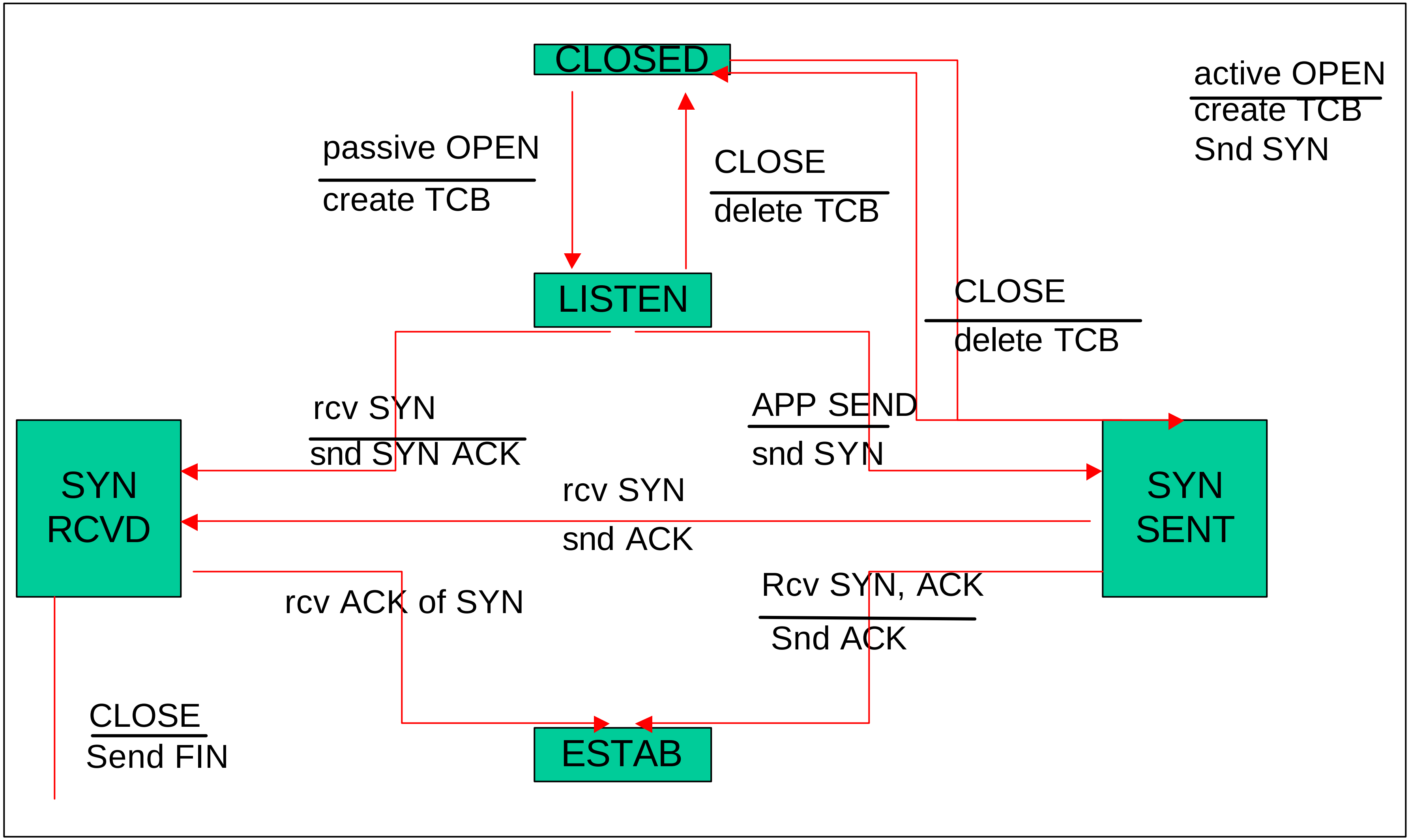
End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

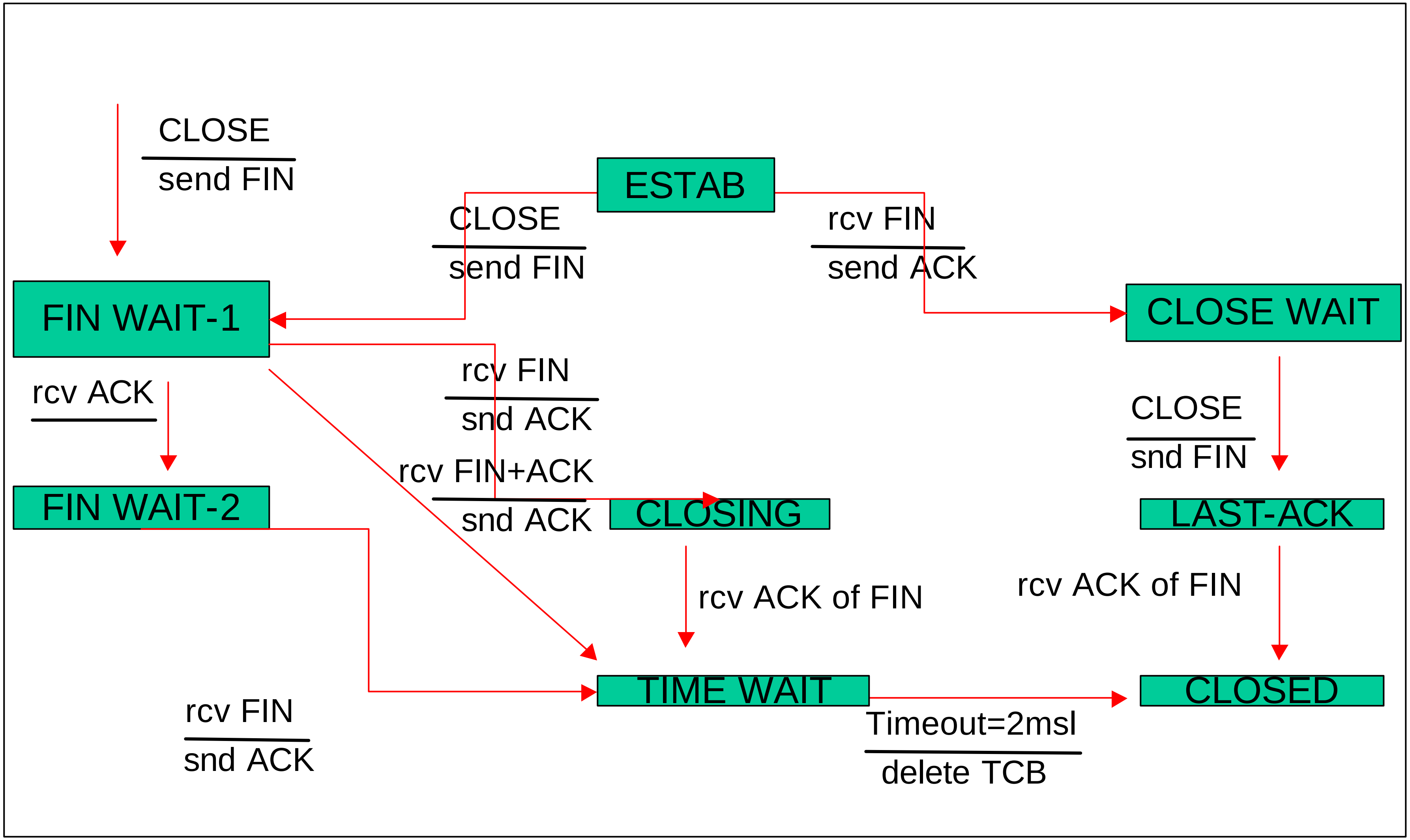
Network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at

TCP connection setup

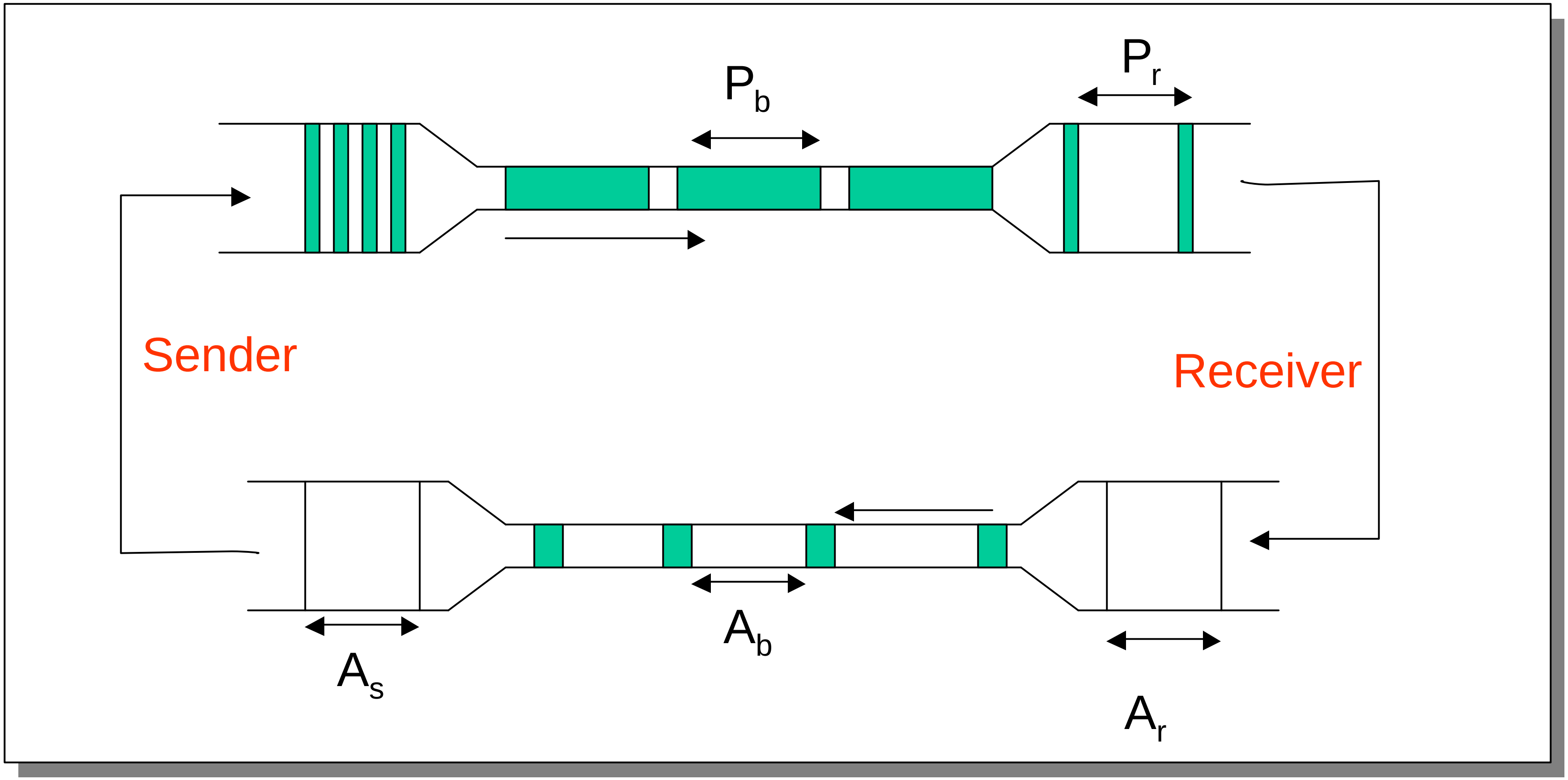


TCP Connection Tear-down



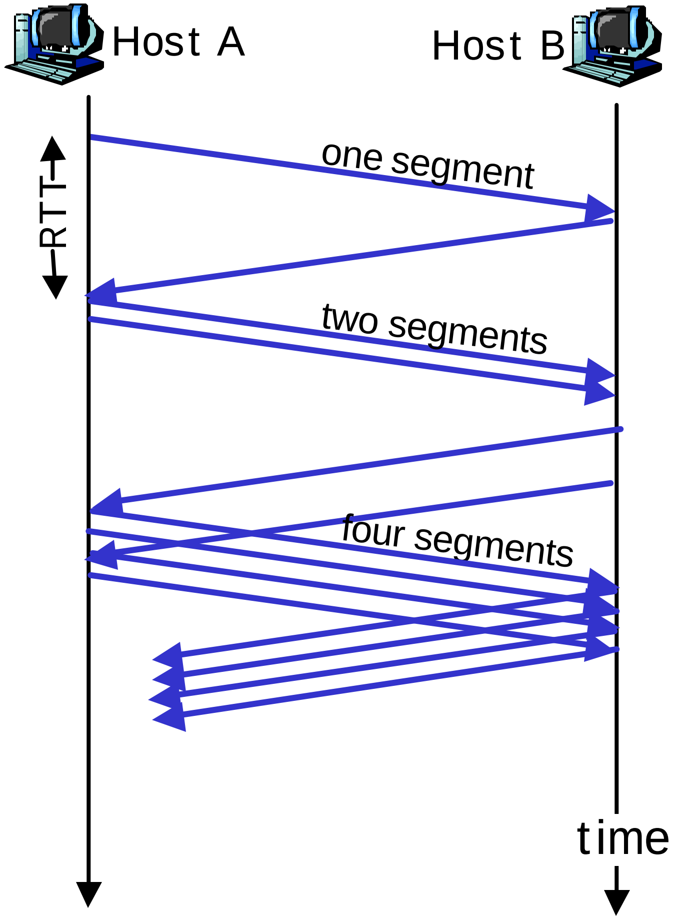
TL: TCP slow start (Tahoe)

- Start the self-clocking behavior of TCP
  - Use acks to clock sending new data
  - Do not send entire advertised window in one shot



TCP Slow Start (more)

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



TL: TCP Reno

- All mechanisms in Tahoe
- Add delayed acks (see flow control section)
- Header prediction
  - Implementation designed to improve performance
  - Has common case code inlined
- Add "fast recovery" to Tahoe's fast retransmit
  - Do not revert to slow-start on fast retransmit
  - Upon detection of 3 duplicate acknowledgments
    - Trigger retransmission (fast retransmission)
    - Set cwnd to 0.5W (multiplicative decrease) and set threshold to 0.5W (skip slow-start)
    - Go directly into congestion avoidance
  - If loss causes timeout (i.e. self-clocking lost), revert to TCP Tahoe

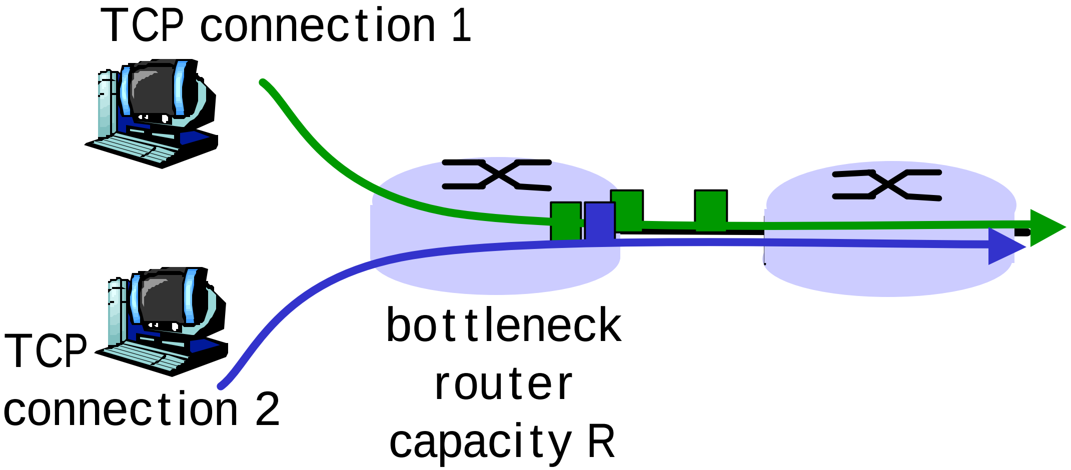
TL: TCP Reno congestion avoidance

Congestion avoidance

```
/* slowstart is over       */
/* cwnd > ssthresh */
Until (loss detected) {
  every w segments ACKed:
      cwnd++
  }
/* fast retrasmit */
if (3 duplicate ACKs) {
  ssthresh = cwnd/2
  cwnd = cwnd/2
  skip slow start
  go to fast recovery
}
```

TL: Is TCP Reno fair?

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



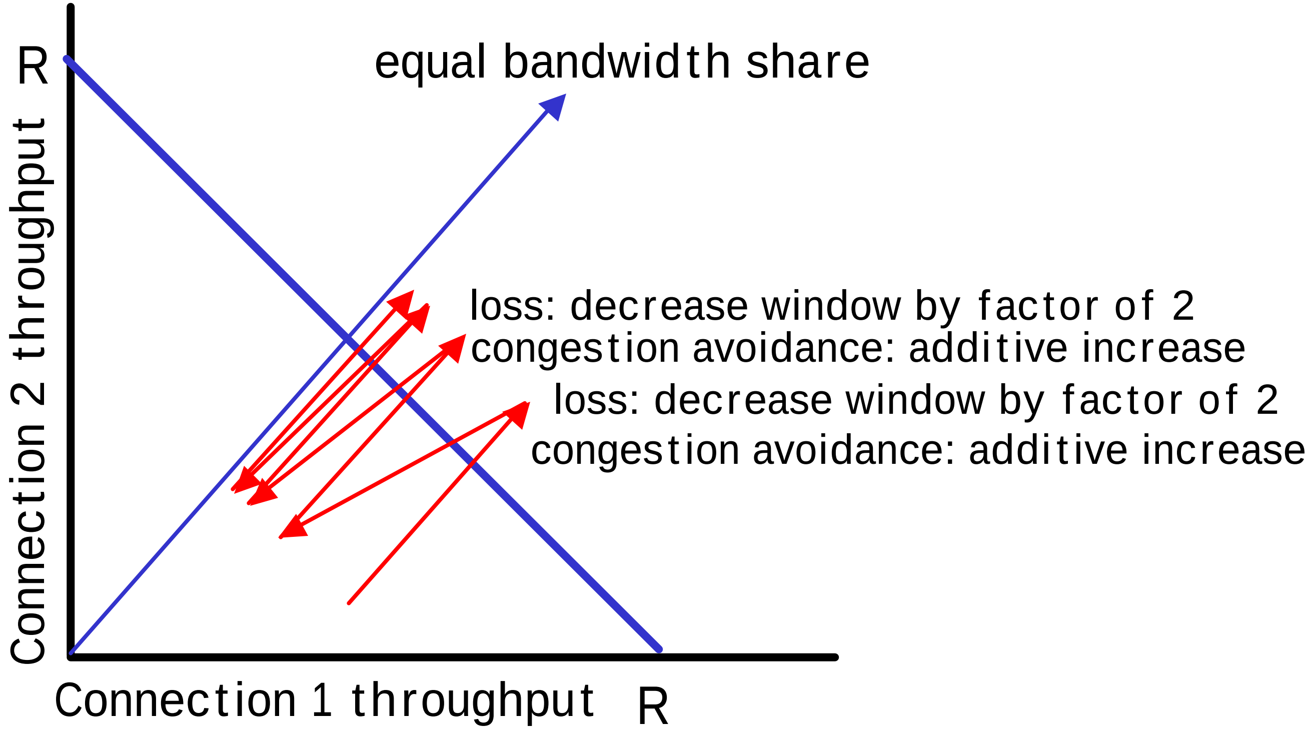
TCP congestion avoidance:

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

TL: Why is TCP Reno fair?

Recall phase plot discussion with two competing sessions:

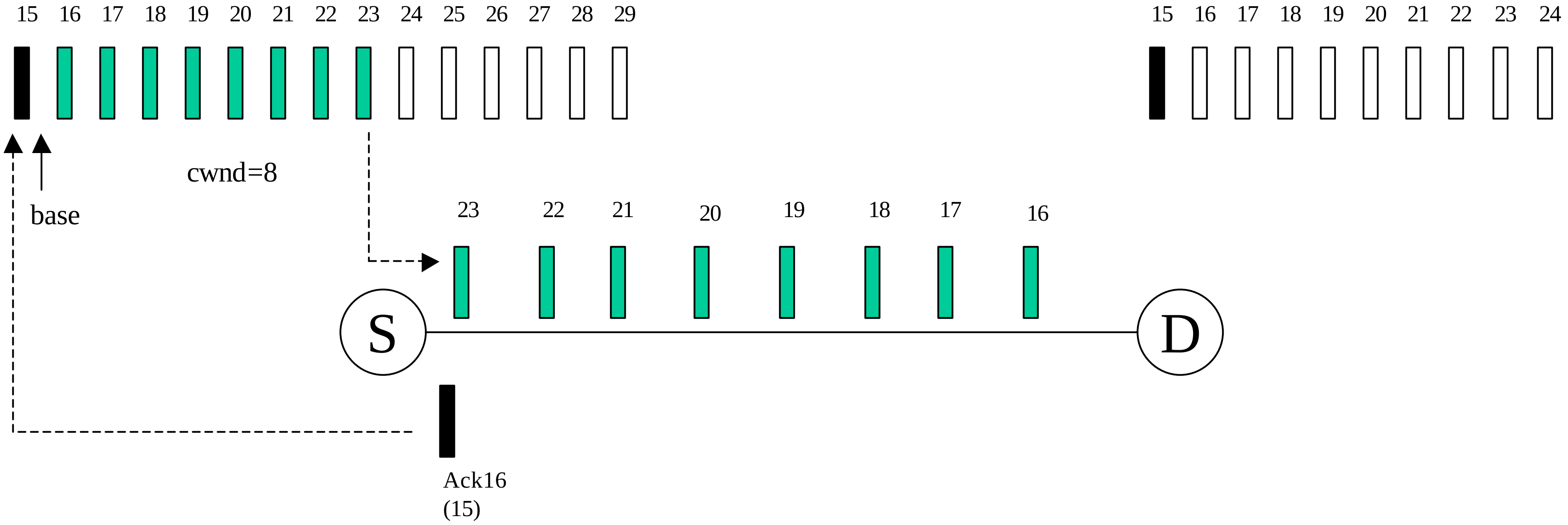
- ❒ Additive increase gives slope of 1, as throughout increases
- ❒ multiplicative decrease decreases throughput proportionally


R
equal bandwidth share
loss: decrease window by factor of 2
congestion avoidance: additive increase
loss: decrease window by factor of 2
congestion avoidance: additive increase
Connection 2 throughput
Connection 1 throughput
R

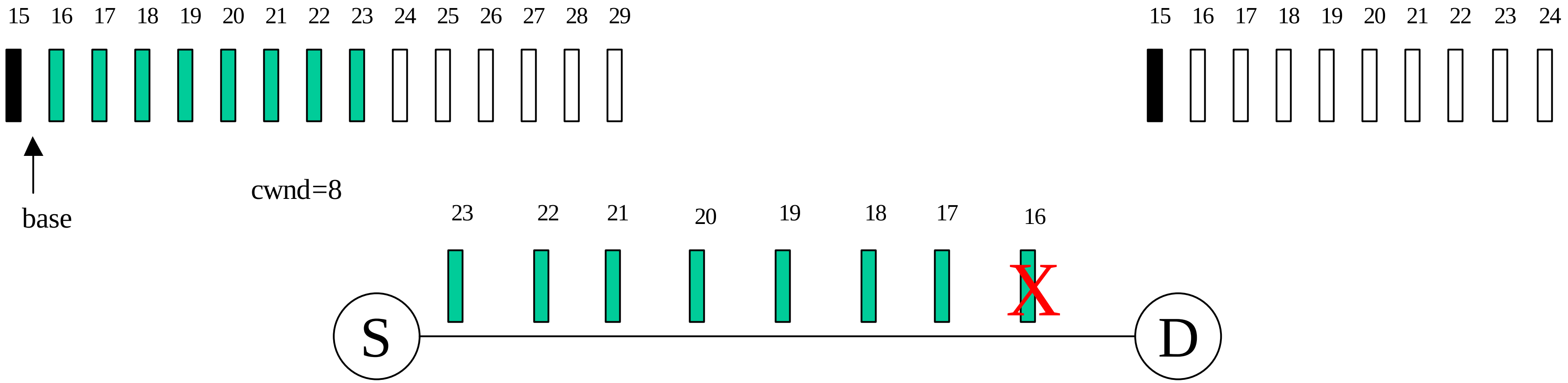
TL: TCP Reno fast recovery mechanism

- Tahoe
  - Loses self-clocking
- Issues in recovering from loss
  - Cumulative acknowledgments freeze window after fast retransmit
    - On a single loss, get almost a window's worth of duplicate acknowledgements
  - Dividing cwnd abruptly in half further reduces sender's ability to transmit
- Reno
  - Use fast recovery to transition smoothly into congestion avoidance
  - Each duplicate ack notifies sender that single packet has cleared network
  - Inflate window temporarily while recovering lost segment
  - Allow new packets out with each subsequent duplicate acknowledgement to maintain self-clocking
  - Deflate window to cwnd/2 after lost packet is recovered

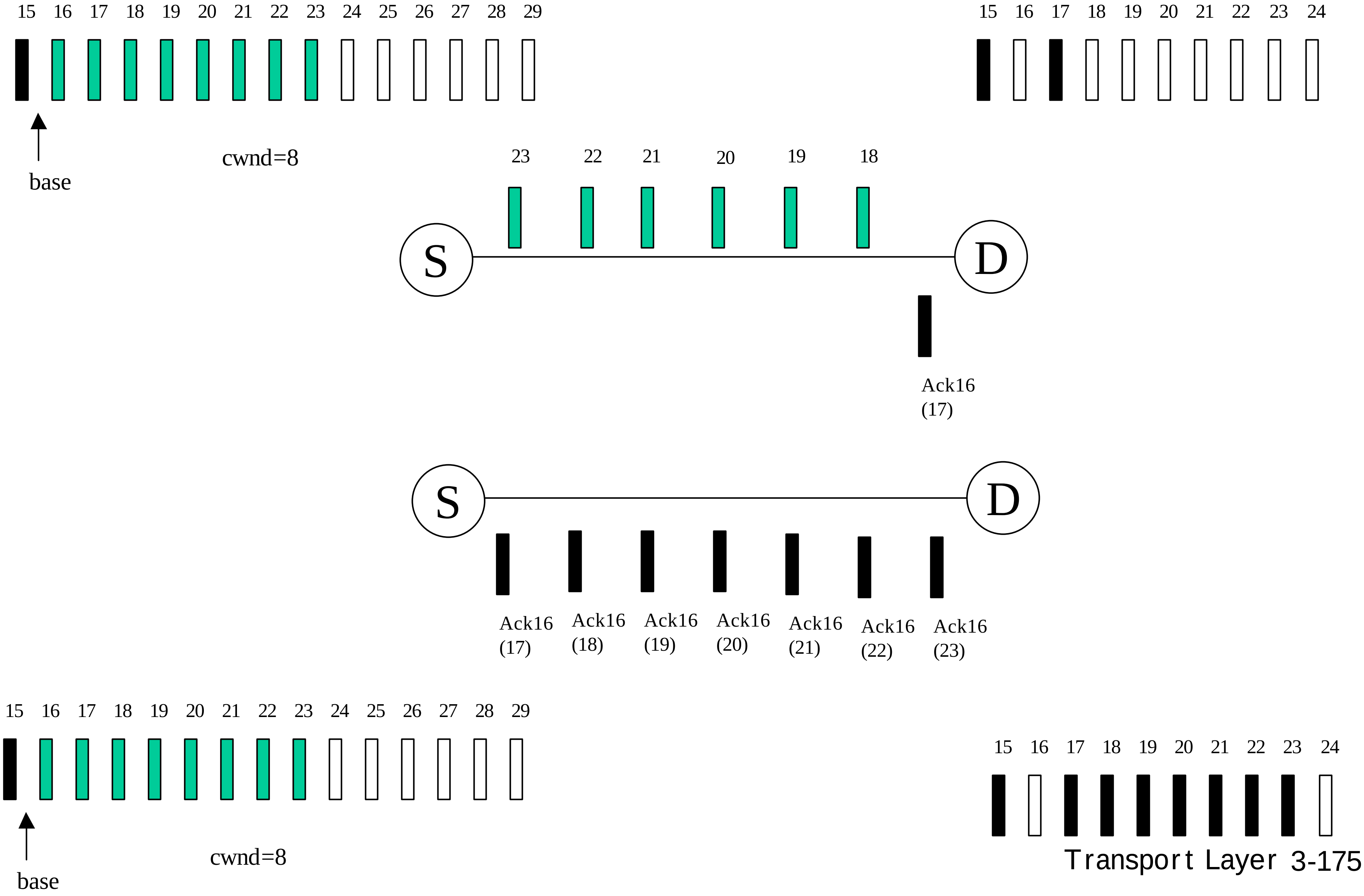
TL: Reno fast recovery example



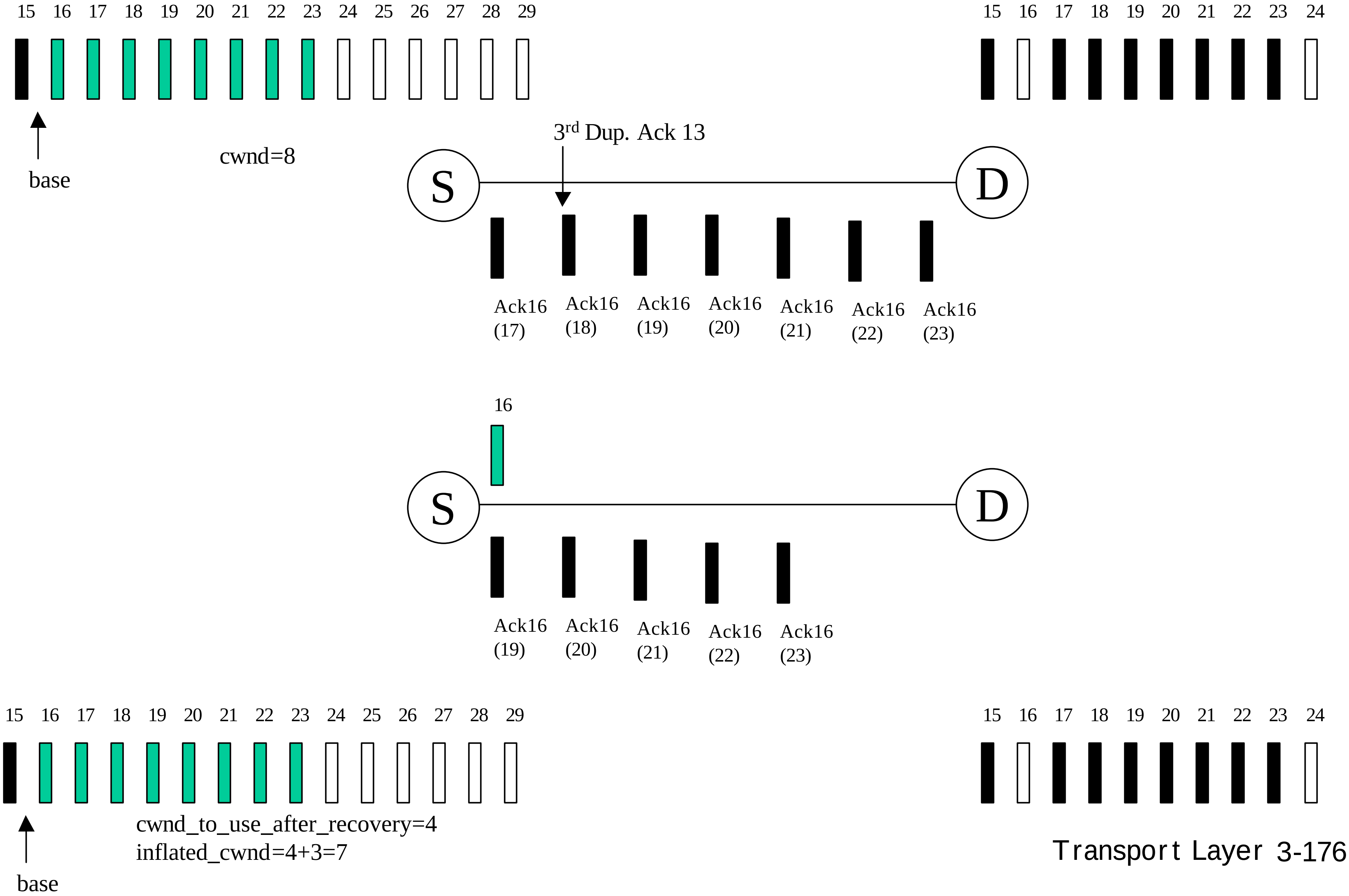
TL: Reno fast recovery example



TL: Reno fast recovery example



TL: Reno fast recovery example



15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

base

cwnd=8

15 16 17 18 19 20 21 22 23 24

3rd Dup. Ack 13

S

D

Ack16 (17)

Ack16 (18)

Ack16 (19)

Ack16 (20)

Ack16 (21)

Ack16 (22)

Ack16 (23)

16

S

D

Ack16 (19)

Ack16 (20)

Ack16 (21)

Ack16 (22)

Ack16 (23)

15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

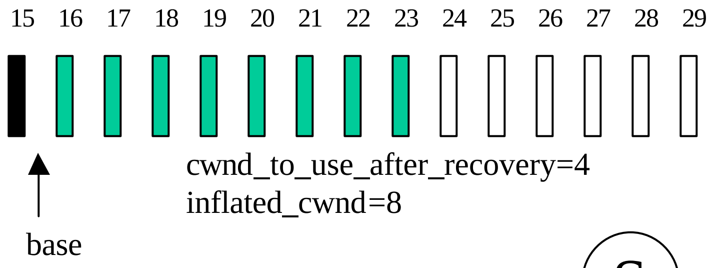
15 16 17 18 19 20 21 22 23 24

cwnd_to_use_after_recovery=4

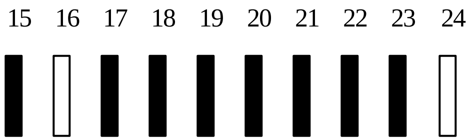
inflated_cwnd=4+3=7

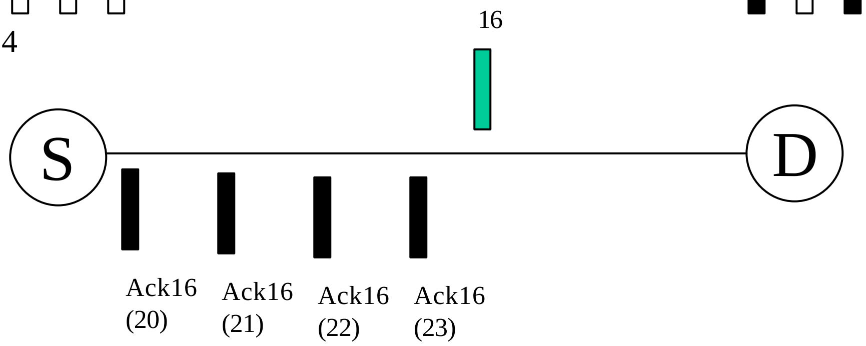
base

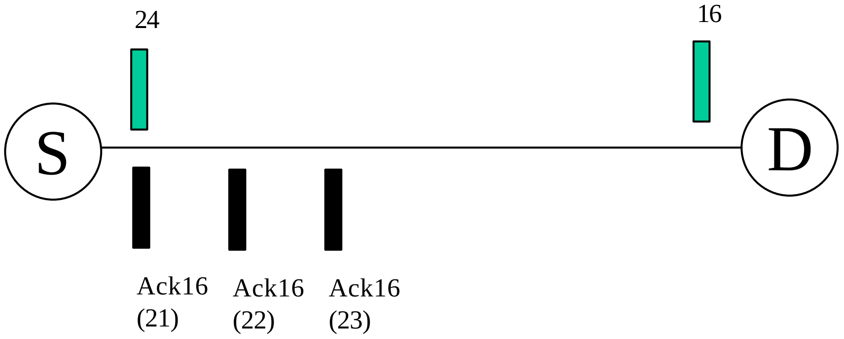
TL: Reno fast recovery example

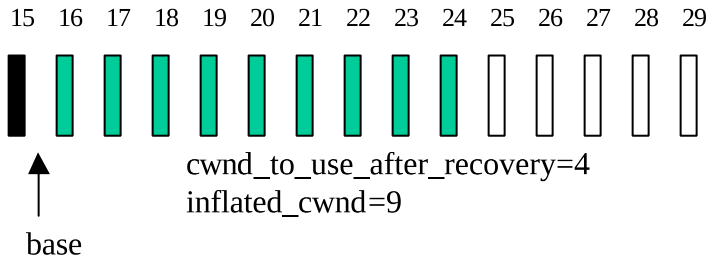


cwnd_to_use_after_recovery=4
inflated_cwnd=8



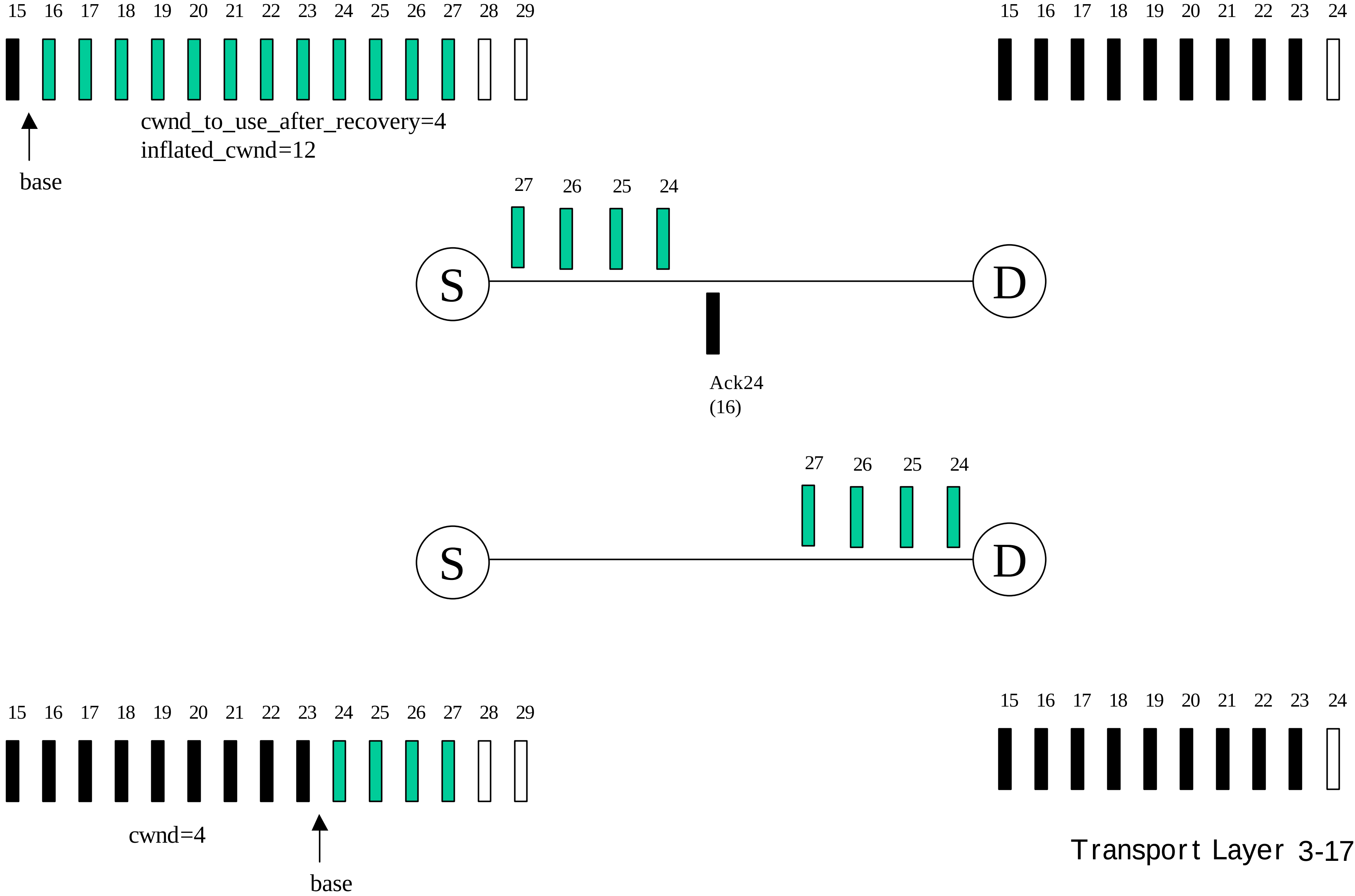






cwnd_to_use_after_recovery=4
inflated_cwnd=9

TL: Reno fast recovery example



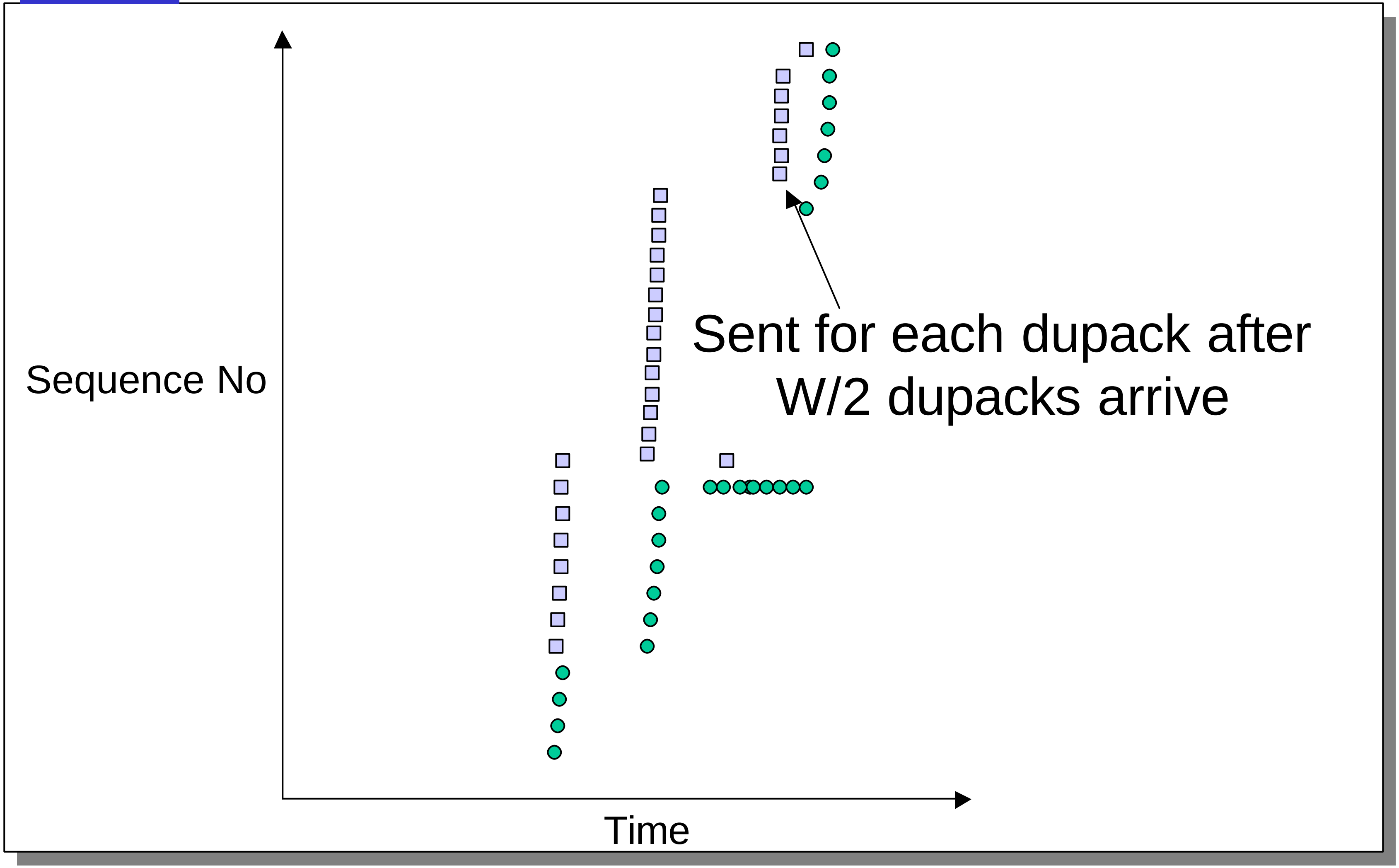
TL: TCP Reno fast recovery behavior

- Behavior
  - Sender idle after halving window
  - Sender continues to get dupacks
    - Waiting for ½ cwnd worth of dupacks
    - Window inflation puts “inflated cwnd” at original cwnd after ½ cwnd worth of dupacks
    - Additional dupacks push “inflated cwnd” beyond original cwnd allowing for additional data to be pushed out during recovery
  - After pausing for ½ cwnd worth of dupacks
    - Transmits at original rate after wait
    - Ack clocking rate is same as before loss
  - Results in ½ RTT time idle, ½ RTT time at old rate
  - Upon recovery of lost segment, cwnd deflated to cwnd/2

TL: Reno fast recovery example

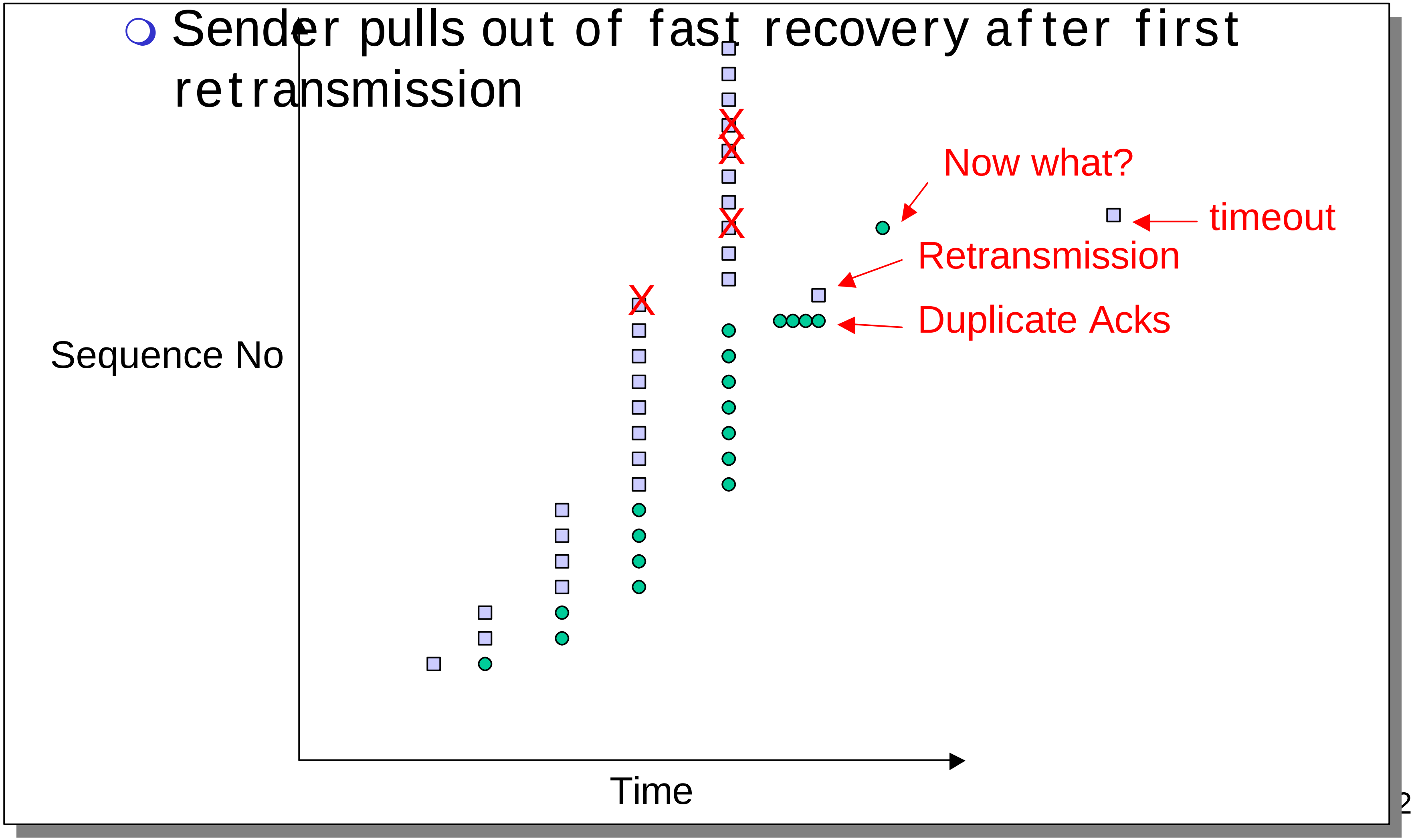
- What if the retransmission is lost?
  - Window inflation to support sending at halved rate until eventual RTO
- Reference
  - http://www.rfc-editor.org/rfc/rfc2001

TL: TCP Reno fast recovery plot



TCP Reno and multiple losses

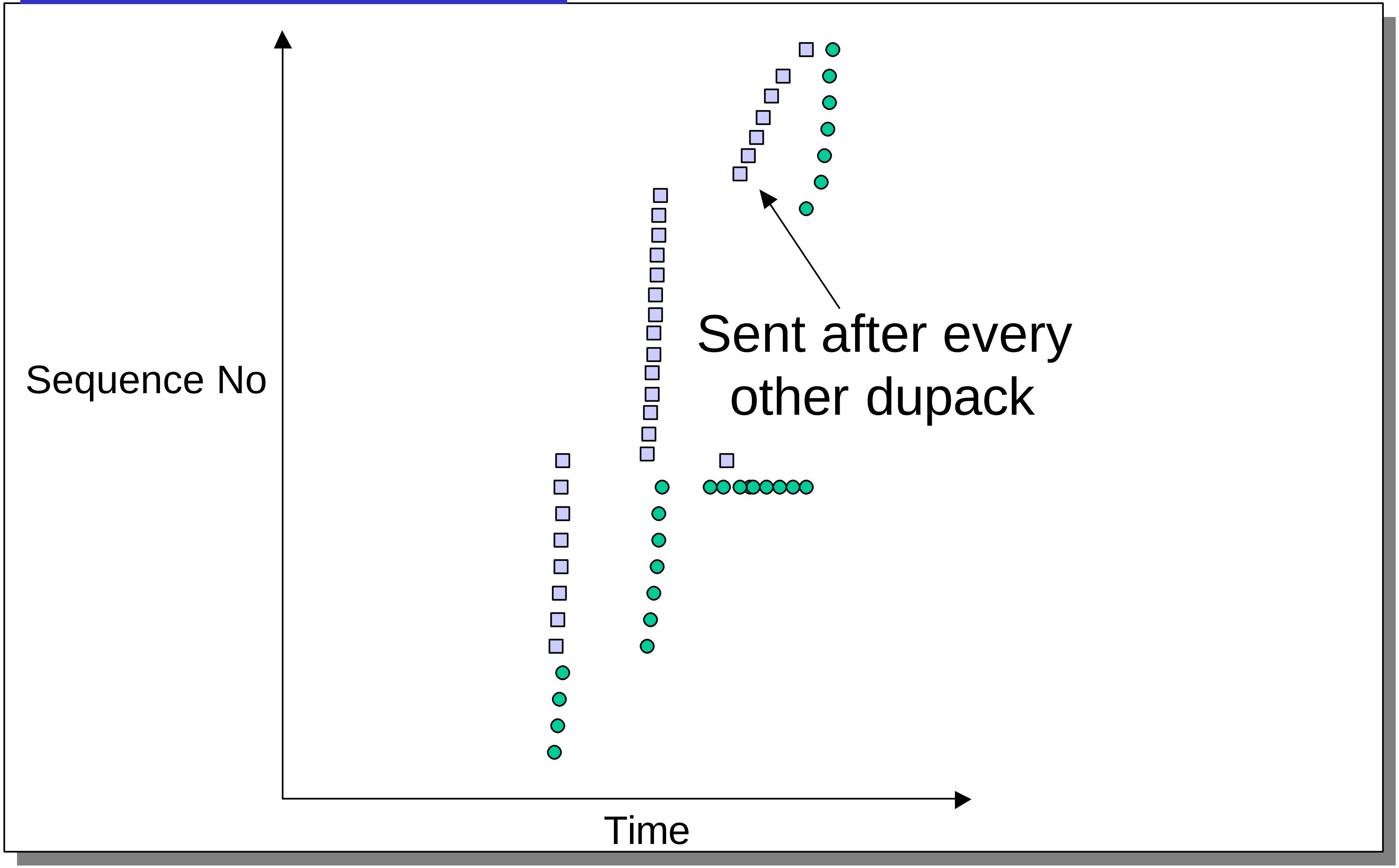
- Multiple losses cause timeout in TCP Reno
  - Sender pulls out of fast recovery after first retransmission



TL: TCP NewReno changes

- More intelligent slow-start
  - Estimate ssthresh based while in slow-start
- Gradual adaptation to new window
  - Send a new packet out for each pair of dupacks
  - Do not wait for ½ cwnd worth of duplicate acks to clear
- Address multiple losses in window

TL: TCP NewReno gradual fast recovery plot

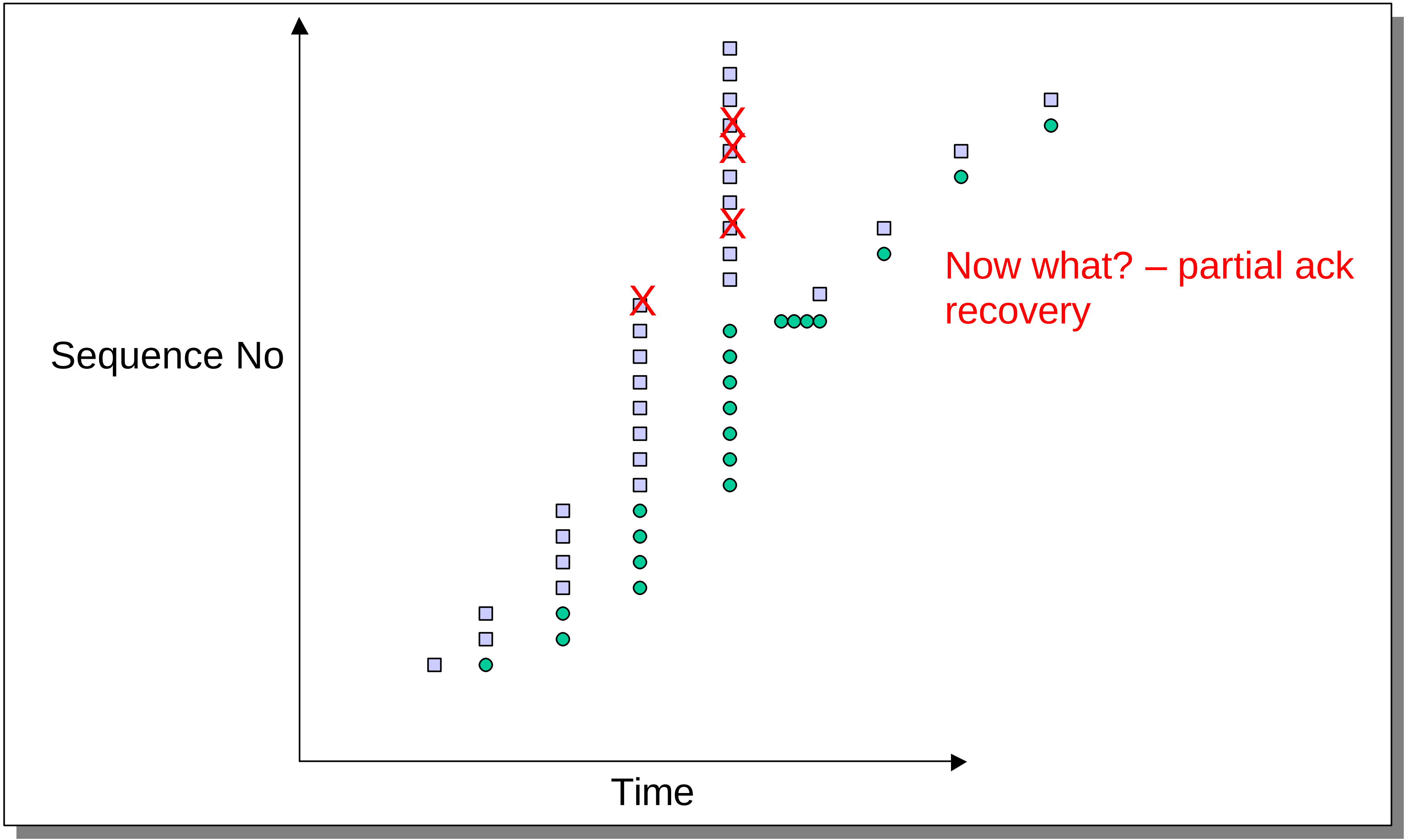


Sent after every other dupack

TL: TCP NewReno and multiple losses

- Partial acknowledgements
  - Window is advanced, but only to the next lost segment
  - Stay in fast recovery for this case, keep inflating window on subsequent duplicate acknowledgements
  - Remain in fast recovery until all segments in window at the time loss occurred have been acknowledged
  - Do not halve congestion window again until recovery is completed
- When does NewReno timeout?
  - When there are fewer than three dupacks for first loss
  - When partial ack is lost
- How quickly does NewReno recover multiple losses?
  - At a rate of one loss per RTT

TL: TCP NewReno multiple loss plot

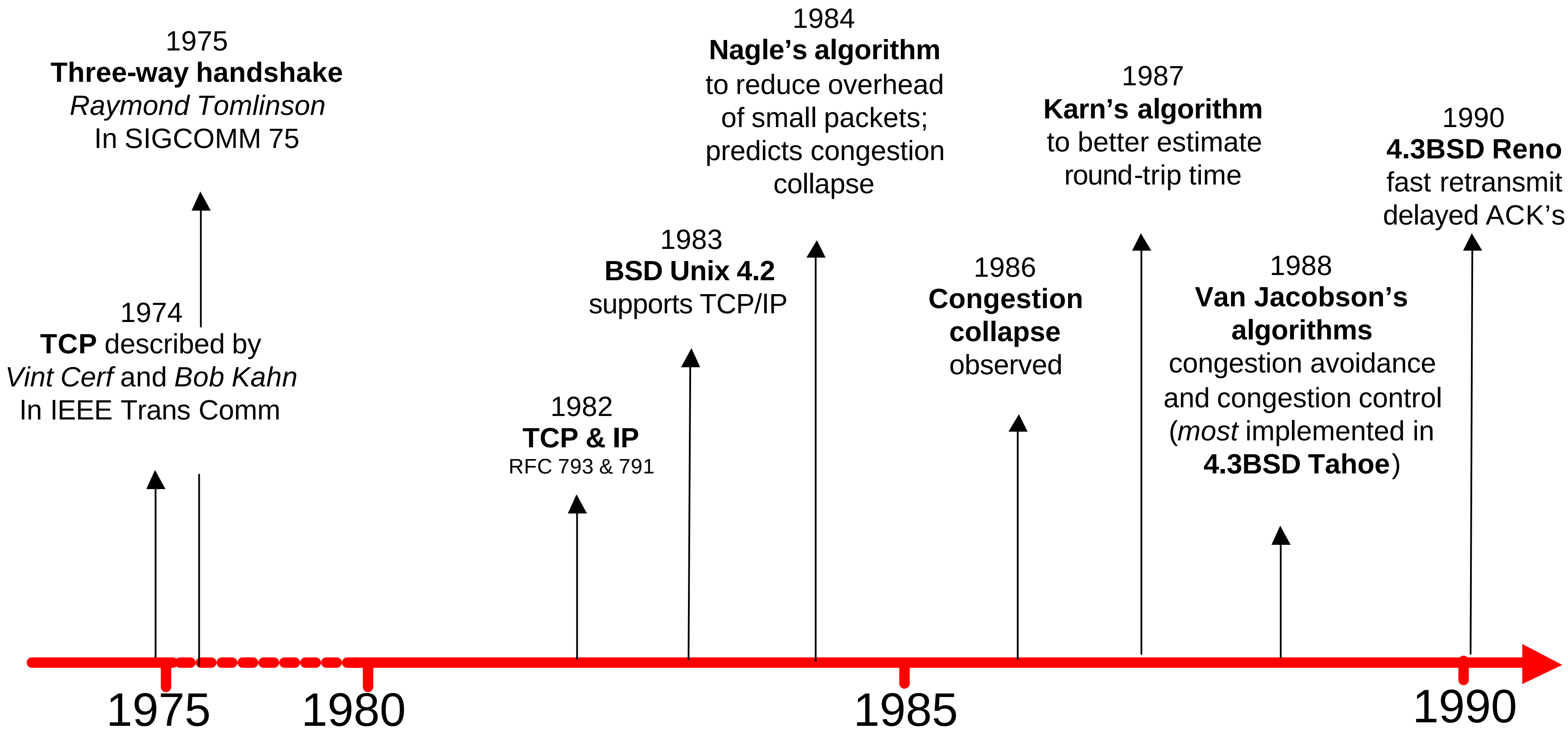


Now what? – partial ack recovery

TL: TCP Flavors

- Tahoe, Reno, NewReno Vegas
- TCP Tahoe (distributed with 4.3BSD Unix)
  - Original implementation of Van Jacobson’s mechanisms
  - Includes slow start, congestion avoidance, fast retransmit
- TCP Reno
  - Fast recovery
- TCP NewReno, SACK, FACK
  - Improved slow start, fast retransmit, and fast recovery

TL: Evolution of TCP



1974
TCP described by
Vint Cerf and *Bob Kahn*
In IEEE Trans Comm

1975
Three-way handshake
Raymond Tomlinson
In SIGCOMM 75

1982
TCP & IP
RFC 793 & 791

1983
BSD Unix 4.2
supports TCP/IP

1984
Nagle's algorithm
to reduce overhead
of small packets;
predicts congestion
collapse

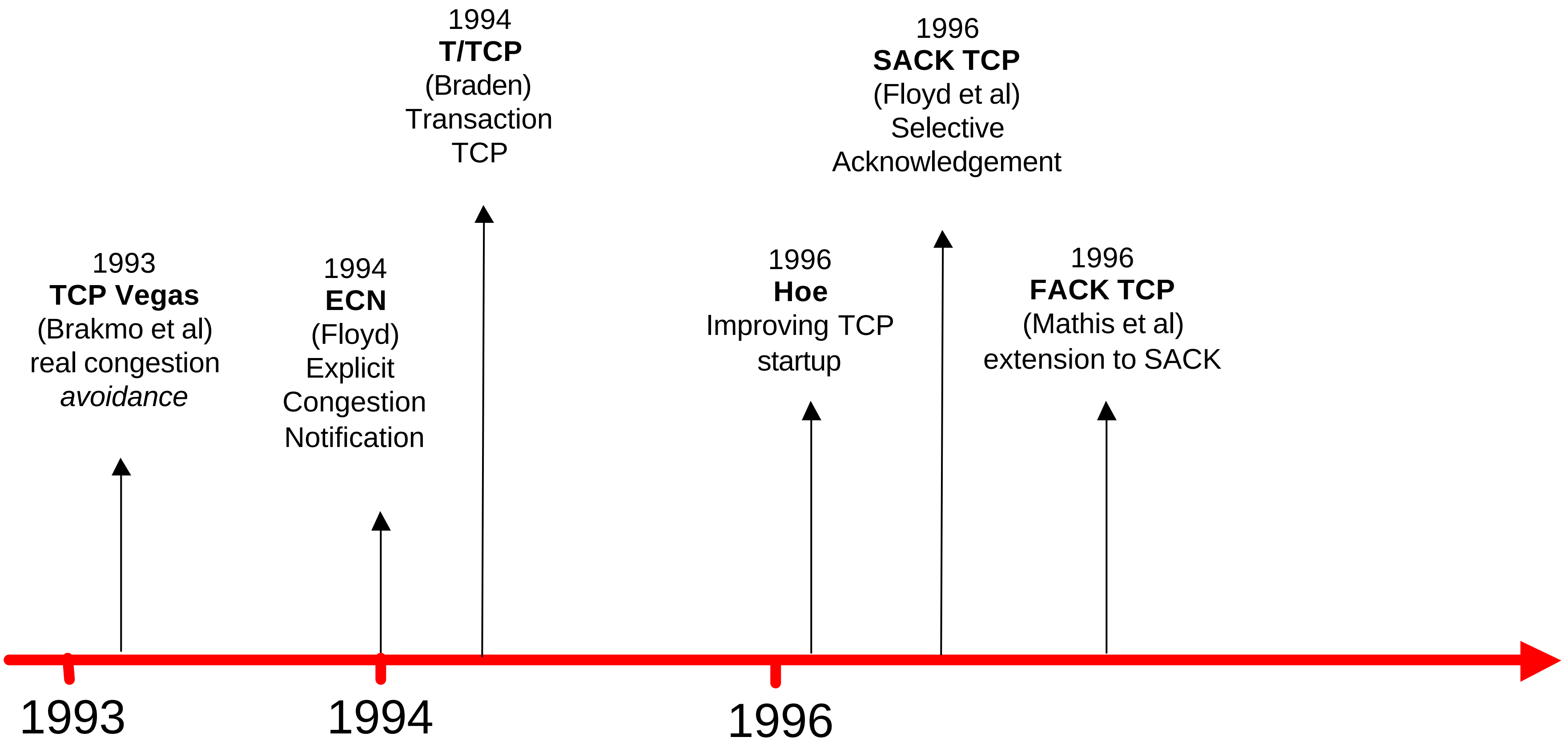
1986
Congestion
collapse
observed

1987
Karn's algorithm
to better estimate
round-trip time

1988
Van Jacobson's
algorithms
congestion avoidance
and congestion control
(*most* implemented in
4.3BSD Tahoe)

1990
4.3BSD Reno
fast retransmit
delayed ACK's

TL: TCP Through the 1990s



1993
TCP Vegas
(Brakmo et al)
real congestion
avoidance

1994
ECN
(Floyd)
Explicit
Congestion
Notification

1994
T/TCP
(Braden)
Transaction
TCP

1996
Hoe
Improving TCP
startup

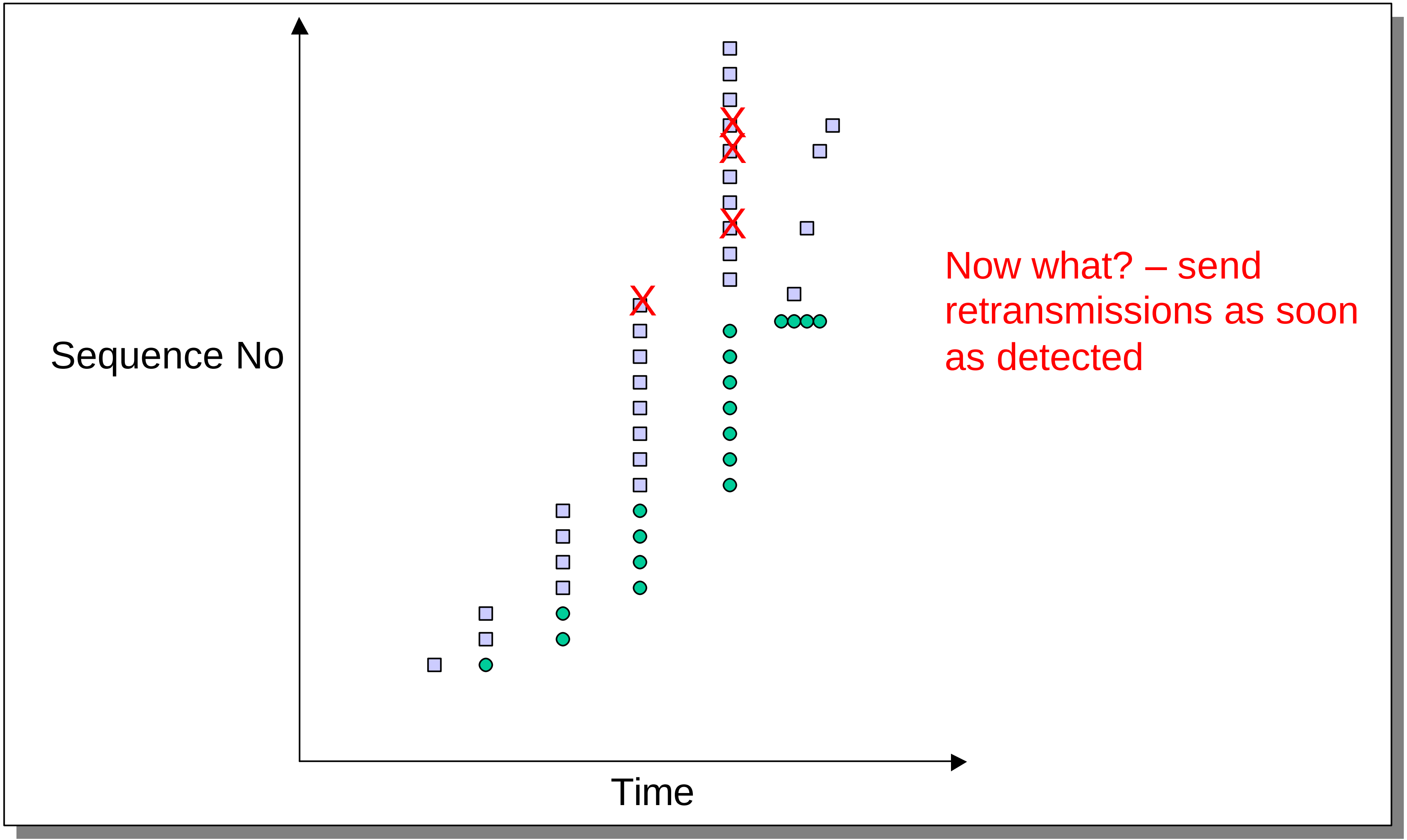
1996
SACK TCP
(Floyd et al)
Selective
Acknowledgement

1996
FACK TCP
(Mathis et al)
extension to SACK

TCP with SACK

- Basic problem is that cumulative acks only provide little information
  - Add selective acknowledgements
    - ACK for exact packets received
    - Not used extensively (yet)
    - Carry information as bitmask of packets received
  - Allows multiple loss recovery per RTT via bitmask
- How to deal with reordering?

TCP with SACK plot



Now what? – send retransmissions as soon as detected

Overview

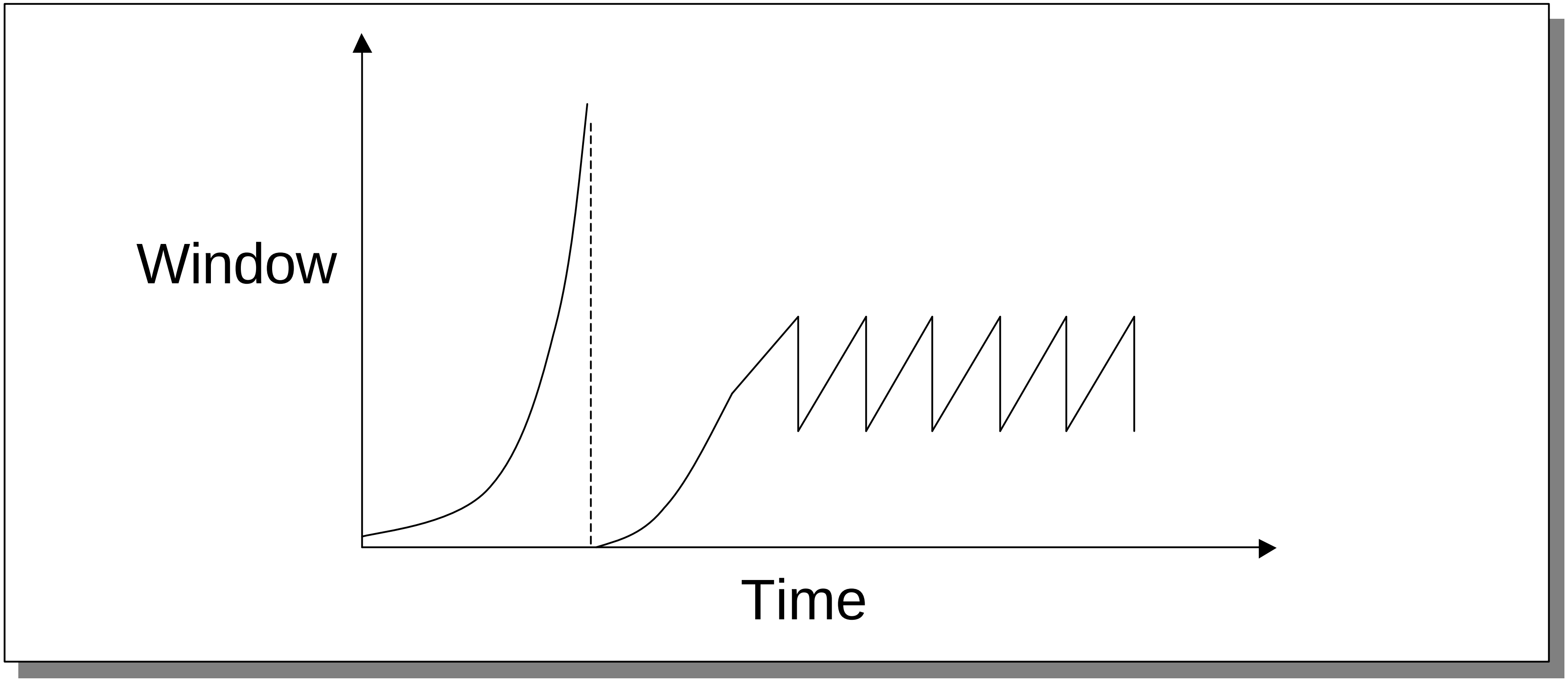
- TCP Vegas
- TCP Modeling
- TFRC and Other Congestion Control
- Changing Workloads
- Header Compression

TCP Modeling

- Given the congestion behavior of TCP can we predict what type of performance we should get?
- What are the important factors
  - Loss rate
    - Affects how often window is reduced
  - RTT
    - Affects increase rate and relates BW to window
  - RTO
    - Affects performance during loss recovery
  - MSS
    - Affects increase rate

Overall TCP Behavior

- Let's concentrate on steady state behavior with no timeouts and perfect loss recovery



Simple TCP Model

- Some additional assumptions
  - Fixed RTT
  - No delayed ACKs
- In steady state, TCP losses packet each time window reaches W packets
  - Window drops to W/2 packets
  - Each RTT window increases by 1 packet→W/2 * RTT before next loss
  - BW = MSS * avg window/RTT = MSS * (W + W/2)/(2 * RTT) = .75 * MSS * W / RTT

Simple Loss Model

- What was the loss rate?
  - Packets transferred = (.75 W/RTT) * (W/2 * RTT) = $3W^2/8$
  - 1 packet lost → loss rate = p = $8/3W^2$
  - W = sqrt(8 / (3 * loss rate))
- BW = .75 * MSS * W / RTT
  - BW = MSS / (RTT * sqrt (2/3p))

TCP Friendliness

- What does it mean to be TCP friendly?
  - TCP is not going away
  - Any new congestion control must compete with TCP flows
    - Should not clobber TCP flows and grab bulk of link
    - Should also be able to hold its own, i.e. grab its fair share, or it will never become popular
- How is this quantified/shown?
  - Has evolved into evaluating loss/throughput behavior
  - If it shows 1/sqrt(p) behavior it is ok
  - But is this really true?

Overview

- TCP Vegas
- TCP Modeling
- TFRC and Other Congestion Control
- Changing Workloads
- Header Compression

TCP Friendly Rate Control (TFRC)

- Equation 1 – real TCP response
  - 1st term corresponds to simple derivation
  - 2nd term corresponds to more complicated timeout behavior
    - Is critical in situations with > 5% loss rates → where timeouts occur frequently
- Key parameters
  - RTO
  - RTT
  - Loss rate

RTO Estimation

- Not used to actually determine retransmissions
  - Used to model TCP's extremely slow transmission rate in this mode
  - Only important when loss rate is high
  - Accuracy is not as critical
- Different TCP's have different RTO calculation
  - Clock granularity critical →500ms typical, 100ms, 200ms, 1s also common
  - RTO = 4 * RTT is close enough for reasonable operation

RTT Estimation

- EWMA ($RTT_{n+1} = (1-\alpha)RTT_n + \alpha RTTSAMP$)
- α = ?
  - Small (.1) → long oscillations due to overshooting link rate
  - Large (.5) → short oscillations due to delay in feedback (1 RTT) and strong dependence on RTT
  - Solution: use large α in T rate calculation but use ratio of $RTTSAMP^{.5}/RTT^{.5}$ for inter-packet spacing

Loss Estimation

- Loss event rate vs. loss rate
- Characteristics
  - Should work well in steady loss rate
  - Should weight recent samples more
  - Should increase only with a new loss
  - Should decrease only with long period without loss
- Possible choices
  - Dynamic window – loss rate over last X packets
  - EWMA of interval between losses
  - Weighted average of last n intervals
    - Last n/2 have equal weight

Loss Estimation

- Dynamic windows has many flaws
- Difficult to chose weight for EWMA
- Solution WMA
  - Choose simple linear decrease in weight for last n/2 samples in weighted average
  - What about the last interval?
  - Include it when it actually increases WMA value
  - What if there is a long period of no losses?
  - Special case (history discounting) when current interval > 2 * avg

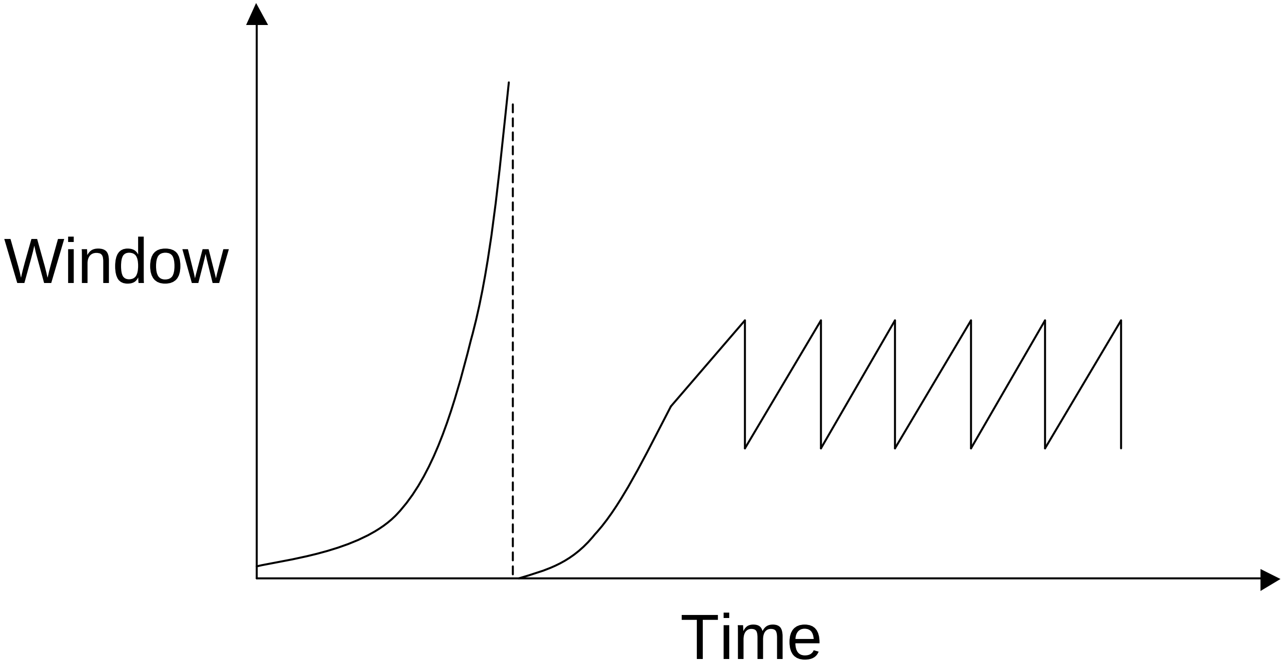
Slow Start

- Used in TCP to get rough estimate of network and establish ack clock
  - Don’t need it for ack clock
  - TCP ensures that overshoot is not > 2x
  - Rate based protocols have no such limitation – why?
- TFRC slow start
  - New rate set to min(2 * sent, 2 * recvd)
  - Ends with first loss report → rate set to ½ current rate

Congestion Avoidance

- Loss interval increases in order to increase rate
  - Primarily due to the transmission of new packets in current interval
  - History discounting increases interval by removing old intervals
  - .14 packets per RTT without history discounting
  - .22 packets per RTT with discounting
- Much slower increase than TCP
- Decrease is also slower
  - 4 – 8 RTTs to halve speed

Overall TCP Behavior



Delay modeling

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

Ignoring congestion, delay is influenced by:

- TCP connection establishment
- data transmission delay
- slow start

Notation, assumptions:

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

Window size:

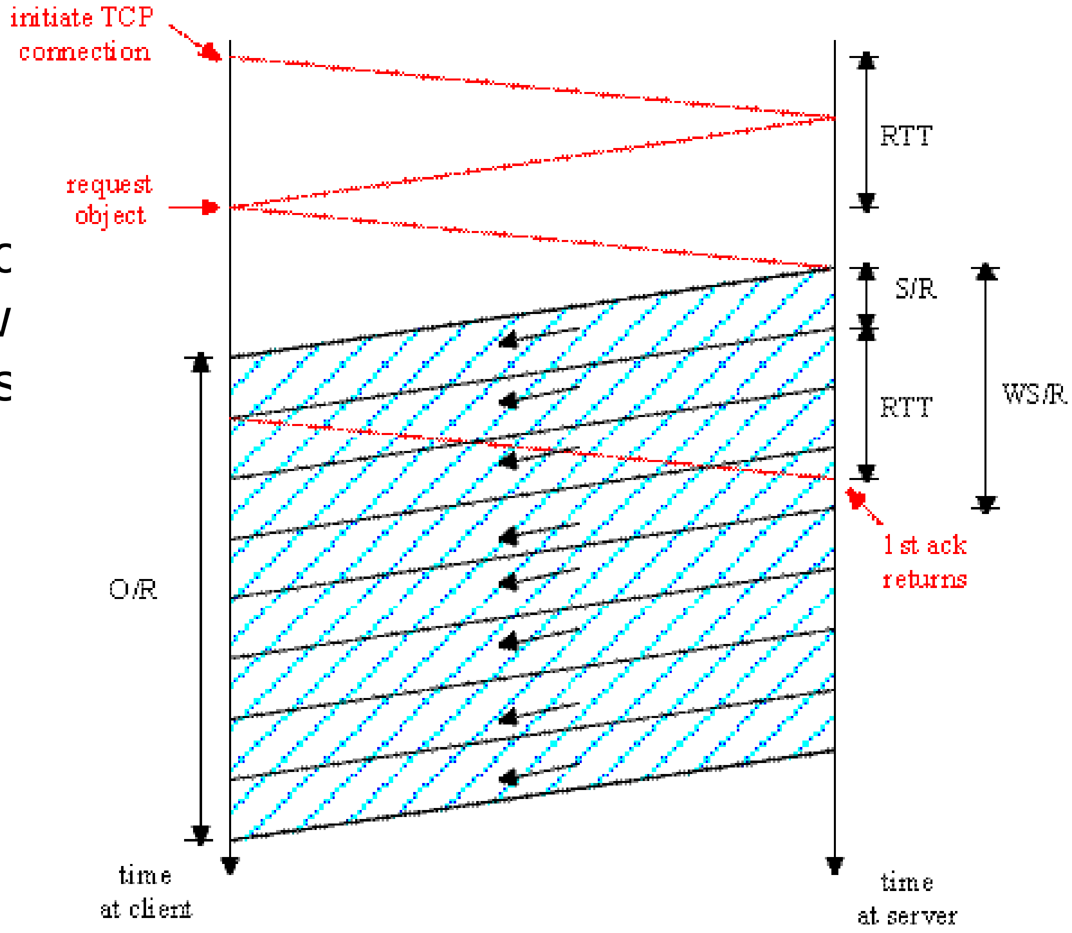
- First assume: fixed congestion window, W segments
- Then dynamic window, modeling slow start

Fixed congestion window (1)

First case:

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

delay = 2RTT + O/R

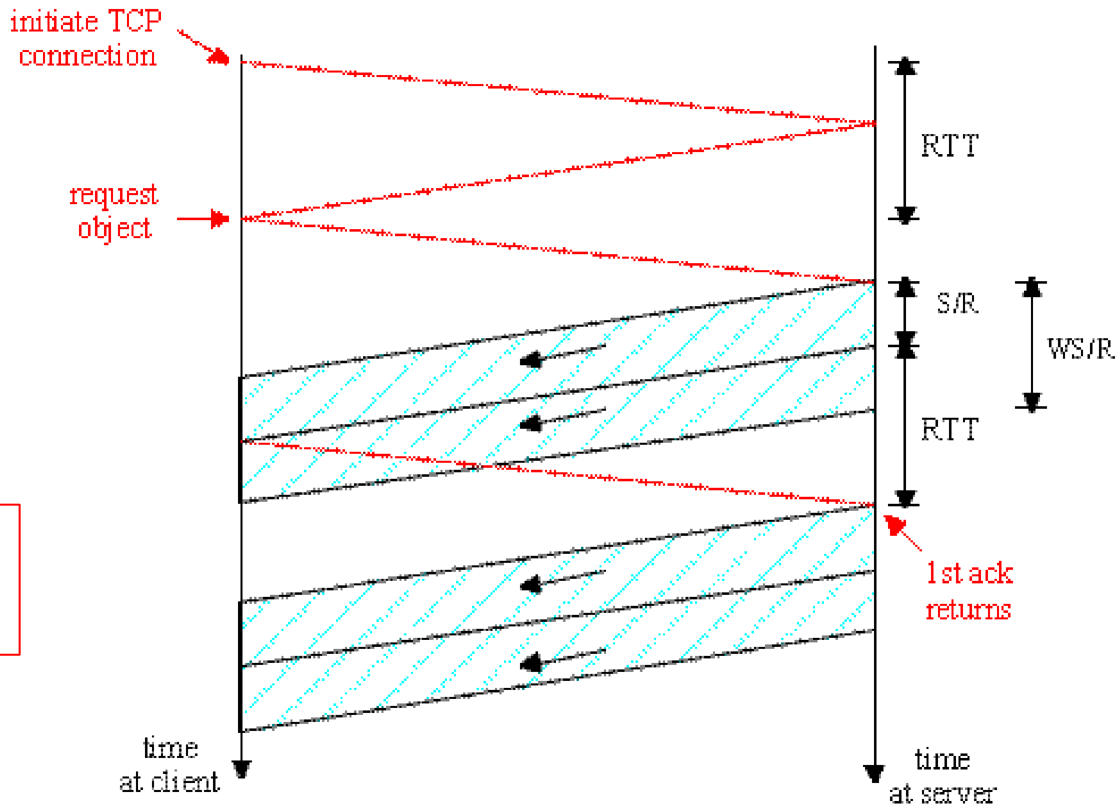


Fixed congestion window (2)

Second case:

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

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



TCP Delay Modeling: Slow Start (1)

Now suppose window grows according to slow start

Will show that the delay for one object is:

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

where *P* is the number of times TCP idles at server:

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

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

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

TCP Delay Modeling: Slow Start (2)

Delay components:

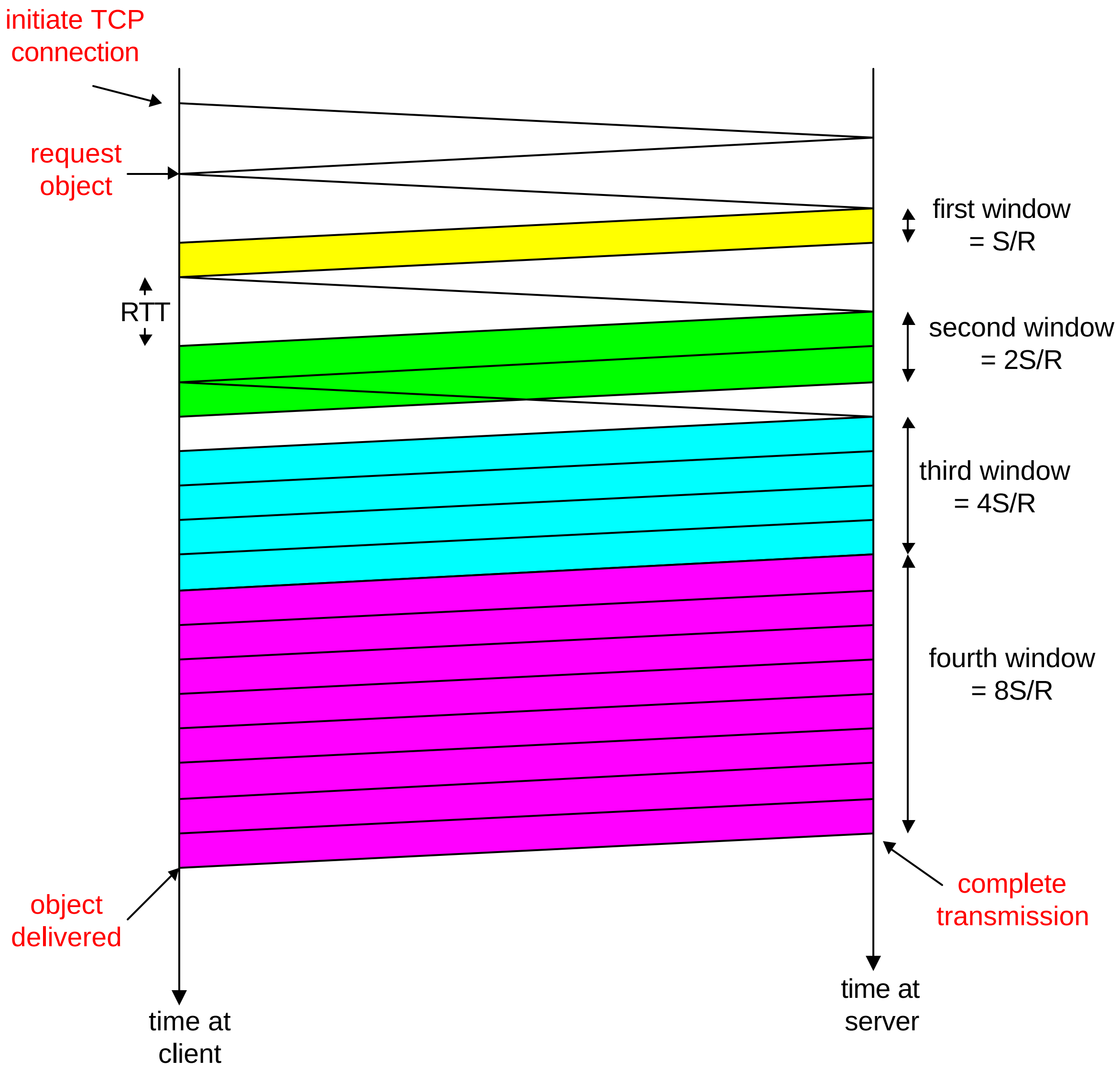
- 2 RTT for connection estab and request
- O/R to transmit object
- time server idles due to slow start

Server idles:
P = min{K-1,Q} times

Example:

- O/S = 15 segments
- K = 4 windows
- Q = 2
- P = min{K-1,Q} = 2

Server idles P=2 times



TCP Delay Modeling (3)

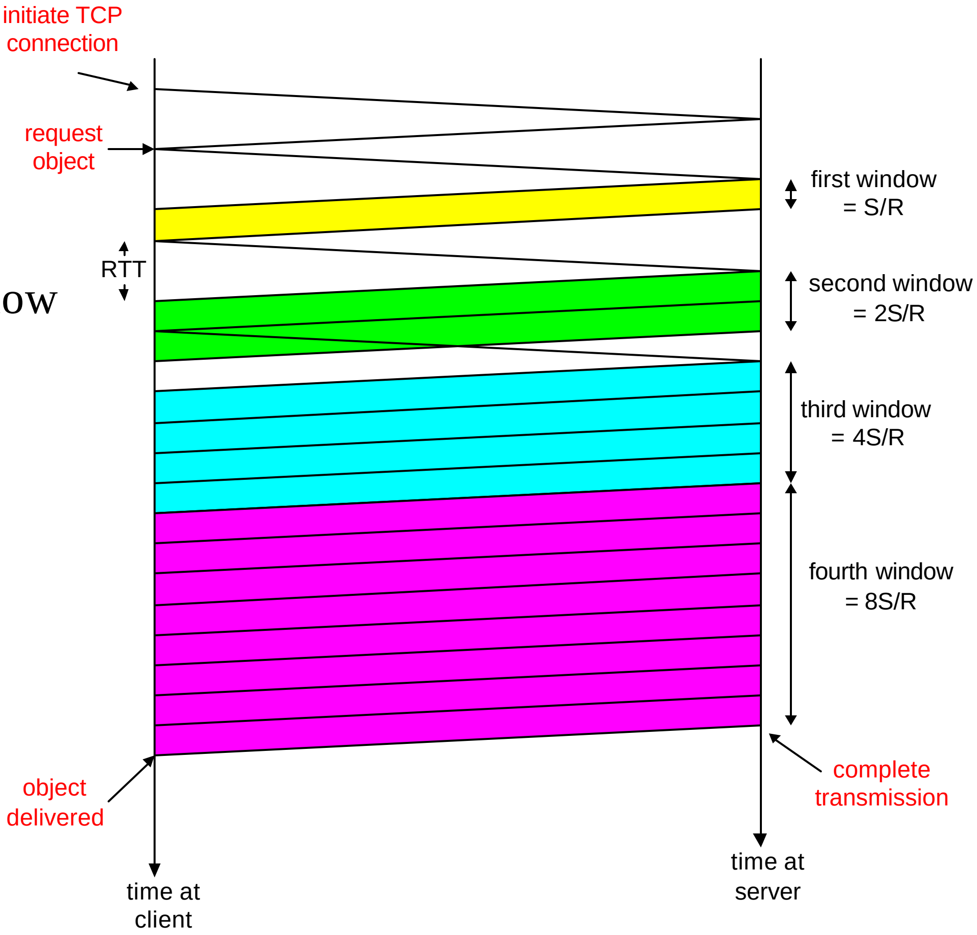
$$\frac{S}{R} + RTT = \text{time from when server starts to send segment}$$

$$\text{until server receives acknowledgement}$$

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

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

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



TCP Delay Modeling (4)

Recall K = number of windows that cover object

How do we calculate K ?

$$
\begin{aligned}
K &= \min\{k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O\} \\
&= \min\{k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq O / S\} \\
&= \min\{k : 2^k - 1 \geq \frac{O}{S}\} \\
&= \min\{k : k \geq \log_2 (\frac{O}{S} + 1)\} \\
&= \left\lceil \log_2 (\frac{O}{S} + 1) \right\rceil
\end{aligned}
$$

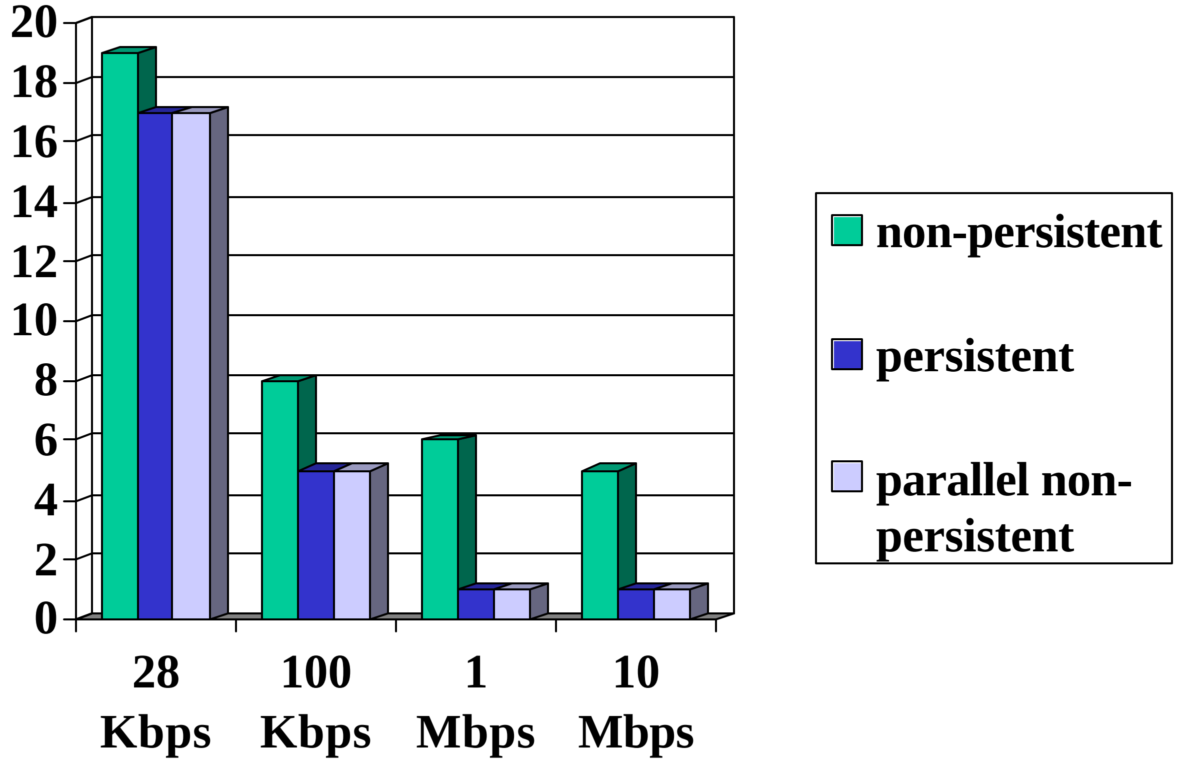
Calculation of Q, number of idles for infinite-size object, is similar (see HW).

HTTP Modeling

- Assume Web page consists of:
  - *1* base HTML page (of size *O* bits)
  - *M* images (each of size *O* bits)
- Non-persistent HTTP:
  - *M+1* TCP connections in series
  - *Response time = (M+1)O/R + (M+1)2RTT + sum of idle times*
- Persistent HTTP:
  - *2 RTT* to request and receive base HTML file
  - *1 RTT* to request and receive M images
  - *Response time = (M+1)O/R + 3RTT + sum of idle times*
- Non-persistent HTTP with X parallel connections
  - Suppose M/X integer.
  - 1 TCP connection for base file
  - M/X sets of parallel connections for images.
  - *Response time = (M+1)O/R + (M/X + 1)2RTT + sum of idle times*

HTTP Response time (in seconds)

RTT = 100 msec, O = 5 Kbytes, M=10 and X=5

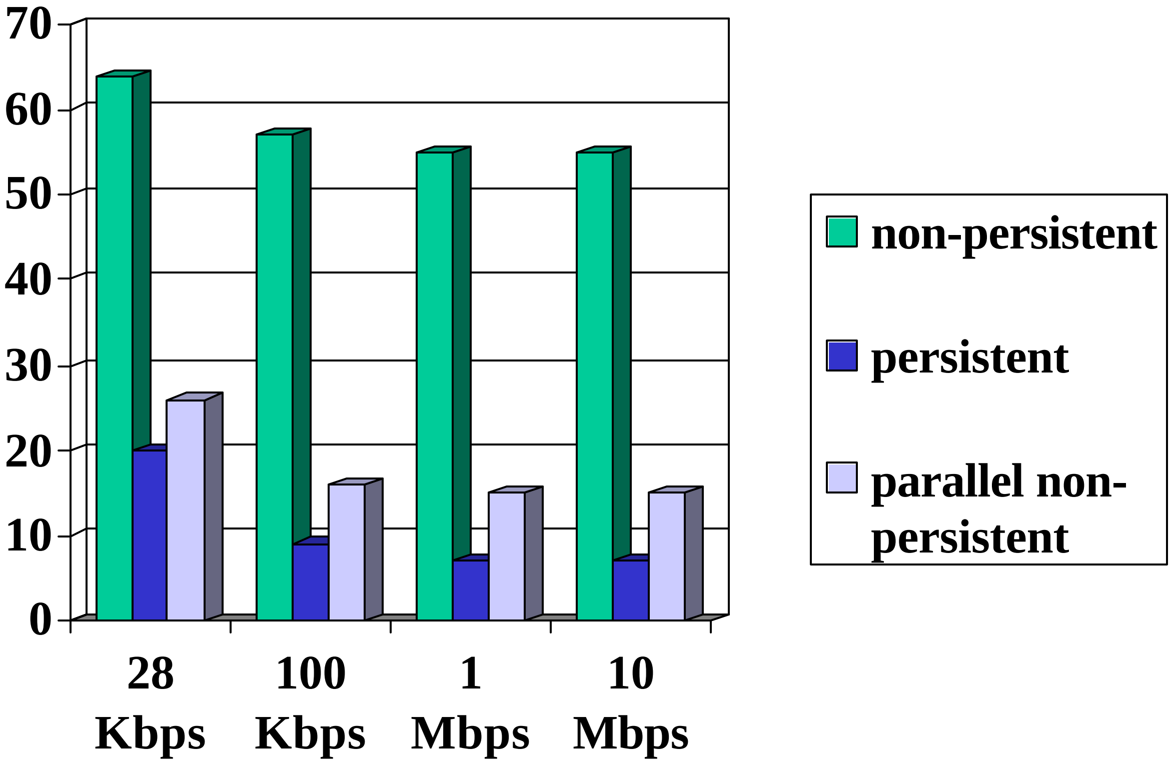


For low bandwidth, connection & response time dominated by transmission time.

Persistent connections only give minor improvement over parallel connections.

HTTP Response time (in seconds)

RTT =1 sec, O = 5 Kbytes, M=10 and X=5



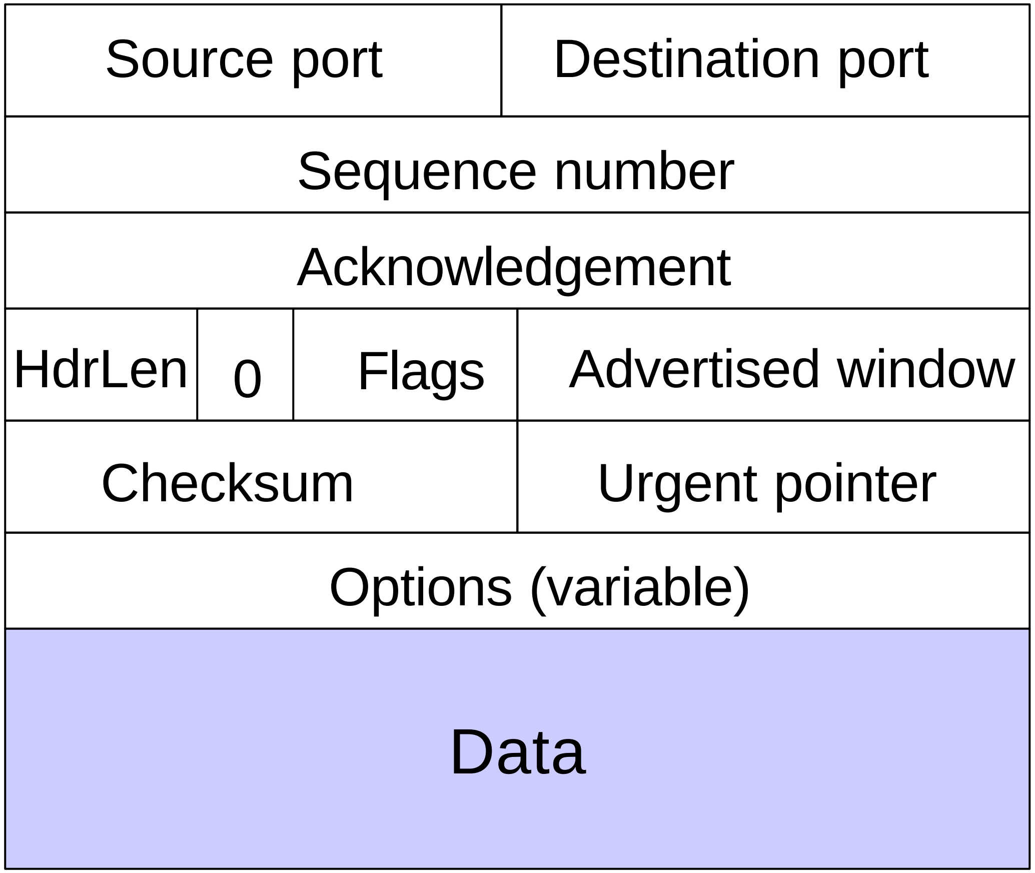
For larger RTT, response time dominated by TCP establishment & slow start delays. Persistent connections now give important improvement: particularly in high delay•bandwidth networks.

TL: TCP header compression

- Why?
  - Low Bandwidth Links
  - Efficiency for interactive
    - 40byte headers vs payload size – 1 byte payload for telnet
- Header compression
  - What fields change between packets?
  - 3 types – fixed, random, differential
  - Mostly applied to TCP, but generic to ALL protocol headers
  - Retransmit all packets uncompressed when compression state is lost

TL: TCP Header

Flags: SYN
FIN
RESET
PUSH
URG
ACK



Source port
Destination port
Sequence number
Acknowledgement
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

TL: TCP Header Compression

- What happens if packets are lost or corrupted?
  - Packets created with incorrect fields
  - Checksum makes it possible to identify
  - How is this state recovered from?
- TCP retransmissions are sent with complete headers
  - Large performance penalty – must take a timeout, no data-driven loss recovery
  - How do you handle other protocols?

TL: Non-reliable Protocols

- IPv6 and other protocols are adding large headers
  - However, these protocols don't have loss recovery
  - How to recover compression state
- Decaying refresh of compression state
  - Suppose compression state is installed by packet X
  - Send full state with X+2, X+4, X+8 until next state
  - Prevents large number of packets being corrupted
- Heuristics to correct packet
  - Apply differencing fields multiple times
- Do we need to define new formats for each protocol?
  - Not really – can define packet description language [mobicom99]

TL: TCP Extensions

- Implemented using TCP options
  - Timestamp
  - Protection from sequence number wraparound
  - Large windows

TL: TCP Timestamp Extension

- Used to improve timeout mechanism by more accurate measurement of RTT
- When sending a packet, insert current timestamp into option
  - 4 bytes for seconds, 4 bytes for microseconds
- Receiver echoes timestamp in ACK
  - Actually will echo whatever is in timestamp
- Removes retransmission ambiguity
  - Can get RTT sample on any packet

TL: TCP and Sequence Number Wraparound

- TCP PAWS
  - Protection Against Wrapped Sequence Numbers
- Wraparound time vs. Link speed
  - 1.5Mbps: 6.4 hours
  - 10Mbps: 57 minutes
  - 45Mbps: 13 minutes
  - 100Mbps: 6 minutes
  - 622Mbps: 55 seconds → < MSL!
  - 1.2Gbps: 28 seconds
- Use timestamp to distinguish sequence number wraparound

TL: TCP and Large Windows

- Delay-bandwidth product for 100ms delay
    - 1.5Mbps: 18KB
    - 10Mbps: 122KB > max 16bit window
    - 45Mbps: 549KB
    - 100Mbps: 1.2MB
    - 622Mbps: 7.4MB
    - 1.2Gbps: 14.8MB
- Scaling factor on advertised window
    - Specifies how many bits window must be shifted to the left
    - Scaling factor exchanged during connection setup

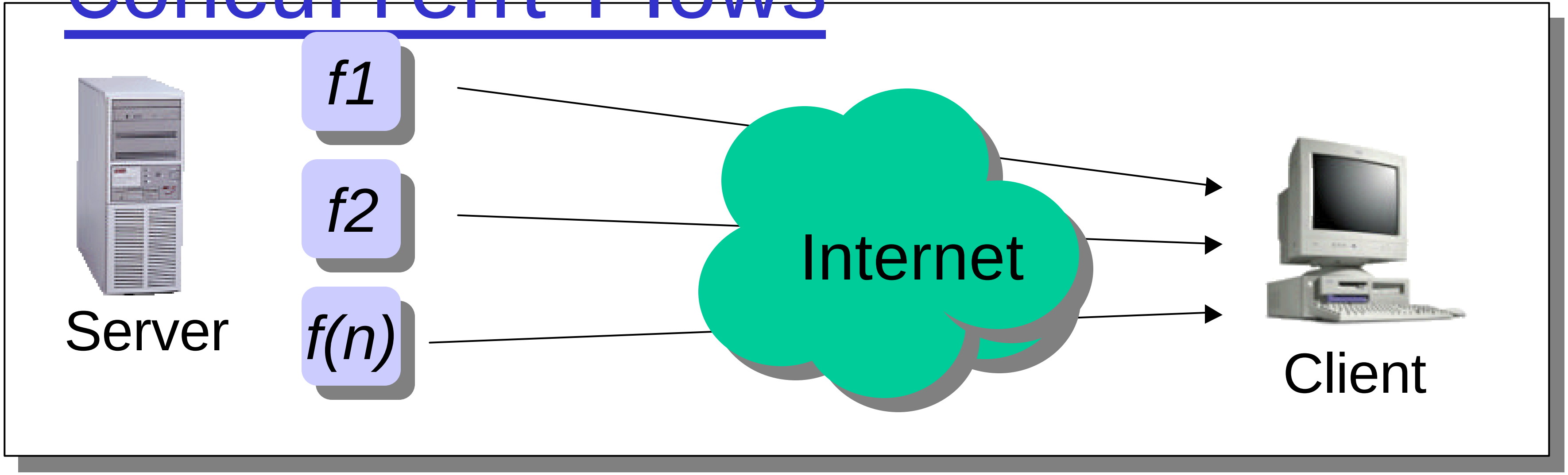
TL: Maximum Segment Size (MSS)

- Exchanged at connection setup
  - Typically pick MTU of local link
- What all does this effect?
  - Efficiency
  - Congestion control
  - Retransmission
- Path MTU discovery
  - Why should MTU match MSS?

TL: Changing Workloads (Aggressive Applications)

- New applications are changing the way TCP is used
- 1980's Internet
  - Telnet & FTP → long lived flows
  - Well behaved end hosts
  - Homogenous end host capabilities
  - Simple symmetric routing
- 2000's Internet
  - Web & more Web → large number of short xfers
  - Wild west – everyone is playing games to get bandwidth
  - Cell phones and toasters on the Internet
  - Policy routing

TL: Problems with Short Concurrent Flows



- Compete for resources
  - N “slow starts” = aggressive
  - No shared learning = inefficient
- Entire life is in slow start
- Fast retransmission is rare

TL: Well Behaved vs. Wild West

- How to ensure hosts/applications do proper congestion control?
- Who can we trust?
  - Only routers that we control
  - Can we ask routers to keep track of each flow
    - No, we must avoid introducing per flow state into routers
  - Active router mechanisms for control in next lecture

TL: Congestion information sharing

- Congestion control
  - Share a single congestion window across all connections to a destination
- Advantages
  - Applications can’t defeat congestion control by opening multiple connections simultaneously
  - Overall loss rate of the network drops
  - Possibly better performance for applications like Web
- Disadvantages?
  - What if you’re the only one doing this? → you get lousy throughput
  - What about hosts like proxies?

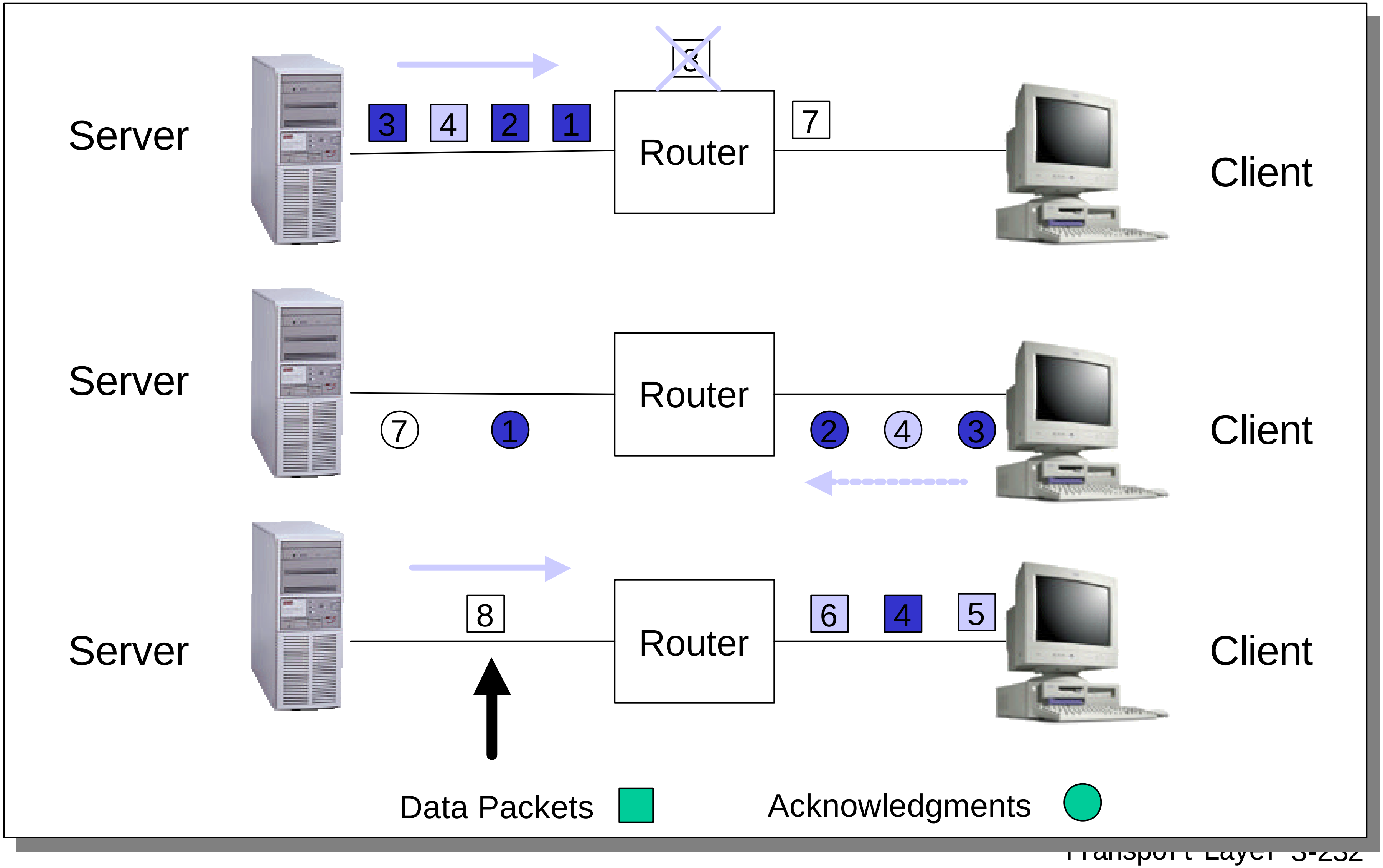
TL: Sharing Congestion Information

- Intra-host sharing
  - Multiple web connections from a host
  - [Padmanabhan98, Touch97]
- Inter-host sharing
  - For a large server farm or a large client population
  - How much potential is there?

TL: Sharing Information

- Loss recovery
  - How is loss detected?
    - By the arrival of later packets from source
    - Why does this have to be later packets on the same connection?
  - Sender keeps order of packets transmitted across all connections
  - When packet is not acked but later packets on other connections are acked, retransmit packet
    - Can we just follow standard 3 packet reordering rule?
    - No, delayed acknowledgments make the conditions more complicated

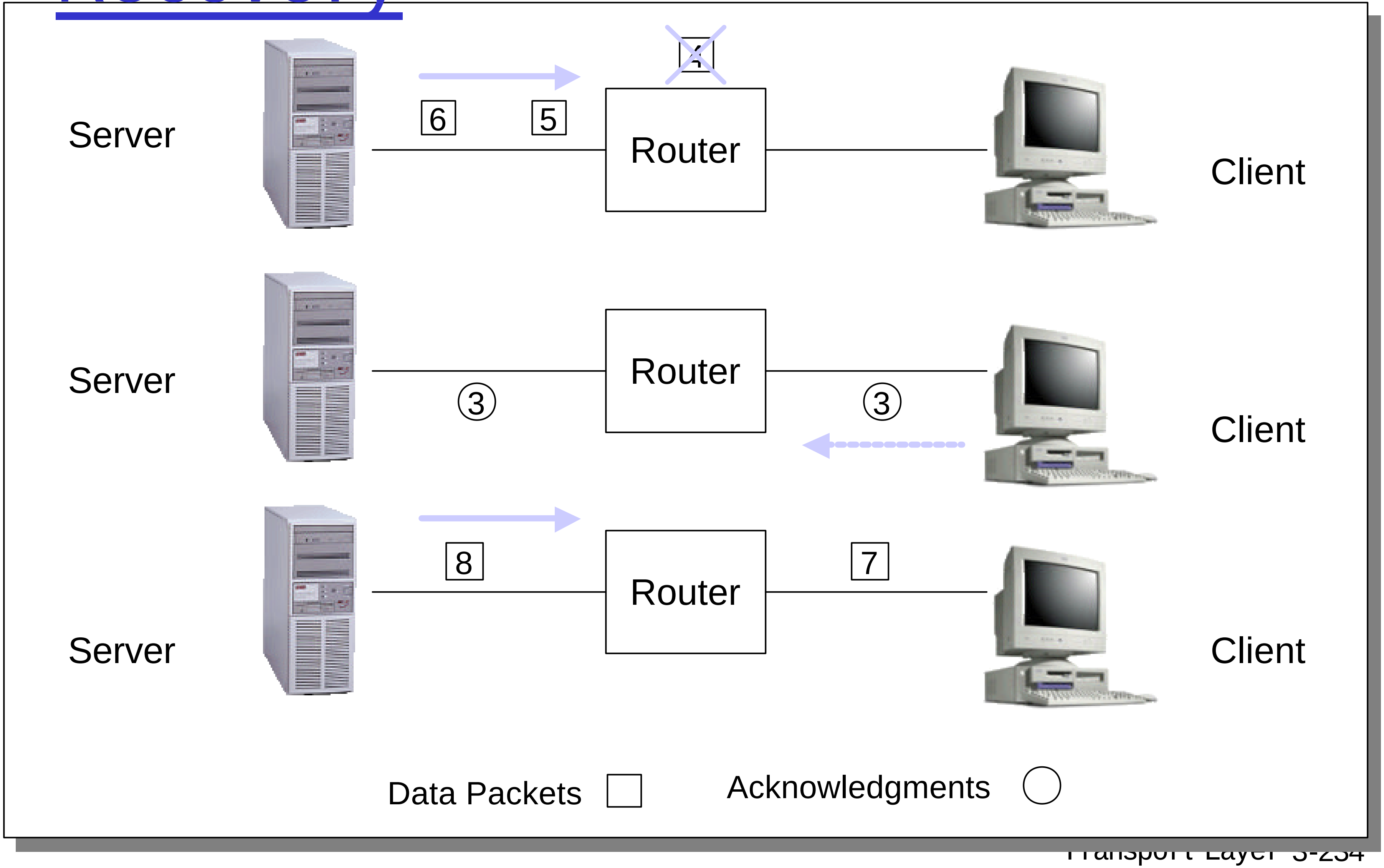
TL: Integrated Loss Recovery



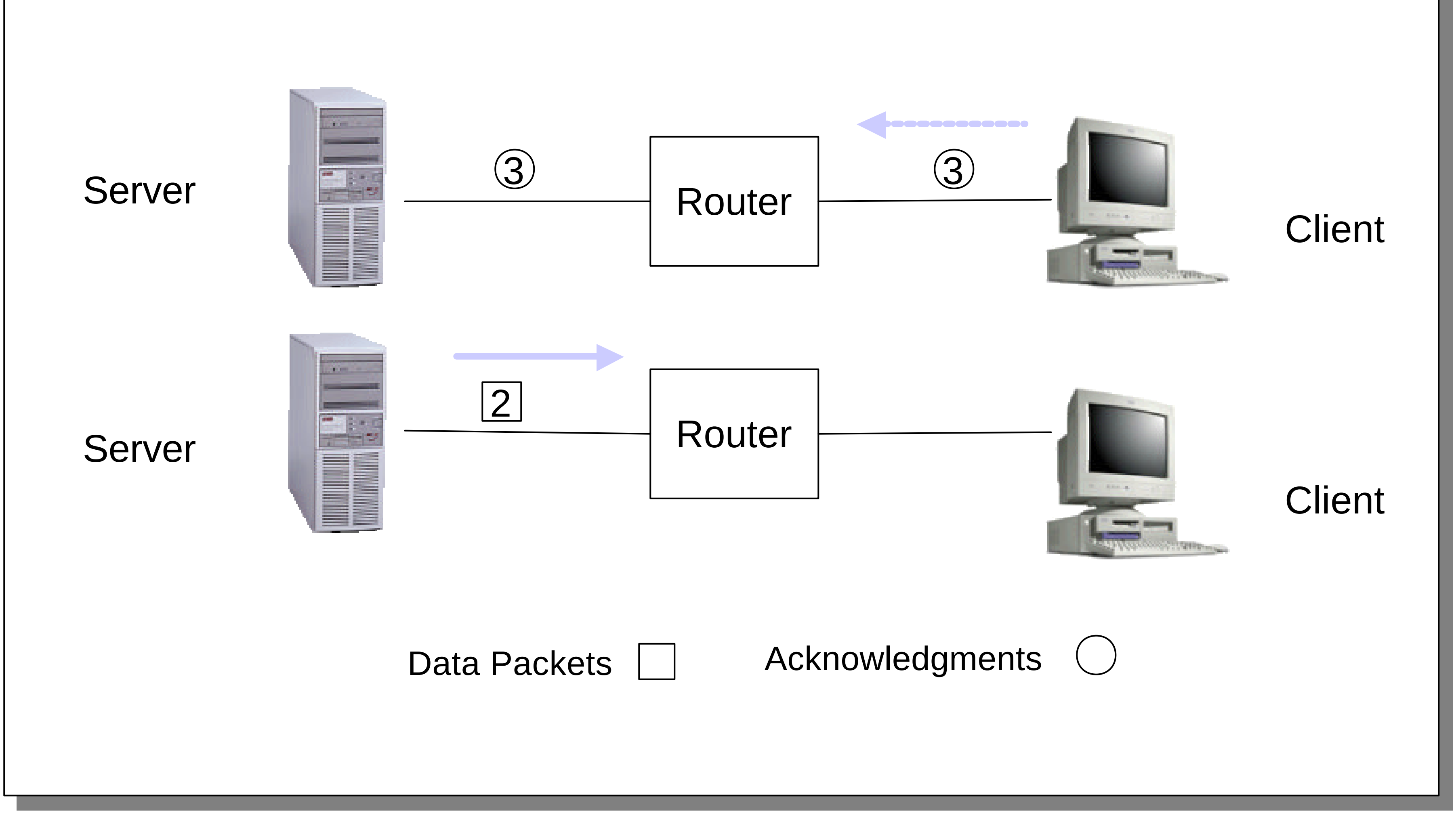
TL: Short Transfers

- Fast retransmission needs at least a window of 4 packets
  - To detect reordering
- Should not be necessary if small outstanding number of packets
  - Adjust threshold to min(3, cwnd/outstanding)
- Some paths have much more reordering than others
  - Adapt threshold to past reordering
- Allow new packets to be transmitted for first few dupacks
  - Will create new dupacks and force retransmission
  - Will not reduce goodput in situations of reordering
  - Follows packet conservation

TL: Enhanced TCP Loss Recovery



TL: Enhanced TCP Loss Recovery



TL: Short Transfers

- Short transfer performance is limited by slow start → RTT
  - Start with a larger initial window
  - What is a safe value?
    - TCP already burst 3 packets into network during slow start
    - Large initial window = min (4*MSS, max (2*MSS, 4380 bytes)) [rfc2414]
    - Enables fast retransmission
    - Only used in initial slow start not in any subsequent slow start

TL: Asymmetric Behavior

- Three important characteristics of a path
  - Loss
  - Delay
  - Bandwidth
- Forward and reverse paths are often independent even when they traverse the same set of routers
  - Many link types are unidirectional and are used in pairs to create bi-directional link

TL: Asymetric Loss

- Loss
  - Information in acks is very redundant
  - Low levels of ack loss will not create problems
  - TCP relies on ack clocking – will burst out packets when cumulative ack covers large amount of data
    - Burst will in turn cause queue overflow/loss
  - Max burst size for TCP and/or simple rate pacing
    - Critical also during restart after idle

TL: Ack Compression

- What if acks encounter queuing delay?
  - Ack clocking is destroyed
    - Basic assumption that acks are spaced due to packets traversing forward bottleneck is violated
  - Sender receives a burst of acks at the same time and sends out corresponding burst of data
  - Has been observed and does lead to slightly higher loss rate in subsequent window

TL: Bandwidth Asymmetry

- Could congestion on the reverse path ever limit the throughput on the forward link?
- Let's assume MSS = 1500bytes and delayed acks
  - For every 3000 bytes of data need 40 bytes of acks
  - 75:1 ratio of bandwidth can be supported
  - Modem uplink (28.8Kbps) can support 2Mbps downlink
  - Many cable and satellite links are worse than this
  - Header compression solves this
    - A bi-directional transfer makes this much worse and more clever techniques are needed

TL: ATM 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

TL: Case study: ATM ABR congestion control

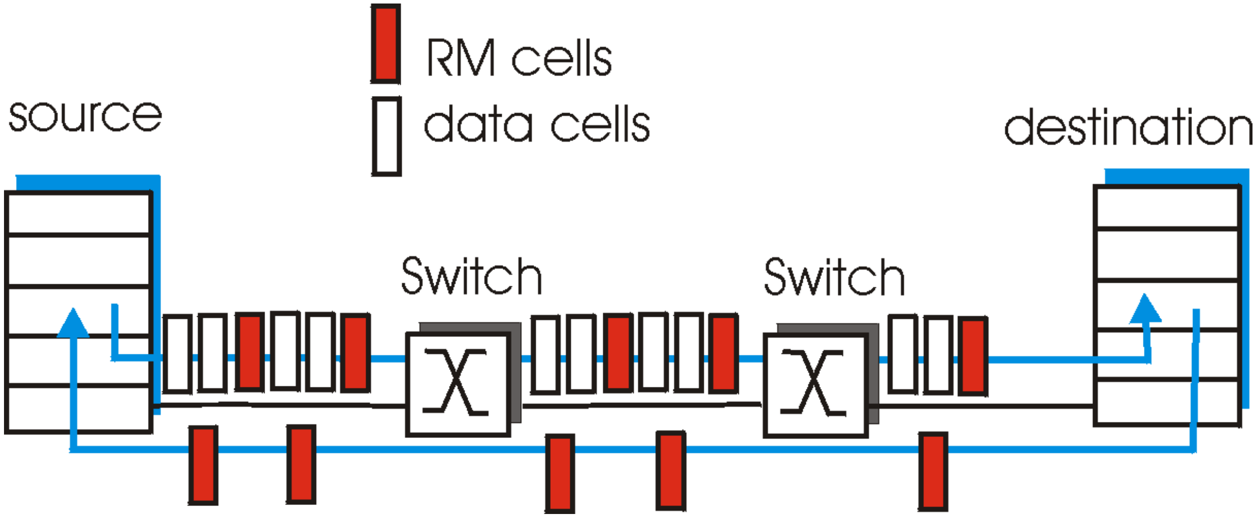
ABR: available bit rate:

- "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

TL: 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, sender sets CI bit in returned RM cell

Chapter 3: Summary

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

Next:

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

TL: TCP Connection Integrity

	TCP A				TCP B
1.	(CRASH)				(send 300, receive 100)
2.	CLOSED				ESTABLISHED
3.	SYN-SENT	→	<SEQ=400><CTL=SYN>	→	(??)
4.	(!!)	←	<SEQ=300><ACK=100><CTL=ACK>	←	ESTABLISHED
5.	SYN-SENT	→	<SEQ=100><CTL=RST>	→	(Abort!!)
6.	SYN-SENT				CLOSED
7.	SYN-SENT	→	<SEQ=400><CTL=SYN>	→	

15-744: Computer Networking

L-10 Alternatives

Transport Alternatives

- TCP Vegas
- Alternative Congestion Control
- Header Compression
- Assigned reading
  - [BP95] TCP Vegas: End to End Congestion Avoidance on a Global Internet
  - [FHPW00] Equation-Based Congestion Control for Unicast Applications

Overview

- TCP Vegas
- TCP Modeling
- TFRC and Other Congestion Control
- Changing Workloads
- Header Compression

TCP Vegas Slow Start

- ssthresh estimation via packet pair
- Only increase every other RTT
  - Tests new window size before increasing

Packet Pair

- What would happen if a source transmitted a pair of packets back-to-back?
- Spacing of these packets would be determined by bottleneck link
  - Basis for ack clocking in TCP
- What type of bottleneck router behavior would affect this spacing
  - Queuing scheduling

Packet Pair

- FIFO scheduling
  - Unlikely that another flows packet will get inserted in-between
  - Packets sent back-to-back are likely to be queued/forwarded back-to-back
  - Spacing will reflect link bandwidth
- Fair queuing
  - Router alternates between different flows
  - Bottleneck router will separate packet pair at exactly fair share rate

Packet Pair in Practice

- Most Internet routers are FIFO/Drop-Tail
- Easy to measure link bandwidths
  - Bprobe, pathchar, pchar, nettimer, etc.
- How can this be used?
  - NewReno and Vegas use it to initialize ssthresh
  - Prevents large overshoot of available bandwidth
  - Want a high estimate – otherwise will take a long time in linear growth to reach desired bandwidth

TCP Vegas Congestion Avoidance

- Only reduce cwnd if packet sent after last such action
  - Reaction per congestion episode not per loss
- Congestion avoidance vs. control
- Use change in observed end-to-end delay to detect onset of congestion
  - Compare expected to actual throughput
  - Expected = window size / round trip time
  - Actual = acks / round trip time

TCP Vegas

- If actual < expected < actual + α
  - Queues decreasing → increase rate
- If actual + α < expected < actual + β
  - Don't do anything
- If expected > actual + β
  - Queues increasing → decrease rate before packet drop
- Thresholds of α and β correspond to how many packets Vegas is willing to have in queues

TCP Vegas

- Fine grain timers
  - Check RTO every time a dupack is received or for "partial ack"
  - If RTO expired, then re-xmit packet
  - Standard Reno only checks at 500ms
- Allows packets to be retransmitted earlier
  - Not the real source of performance gain
- Allows retransmission of packet that would have timed-out
  - Small windows/loss of most of window
  - Real source of performance gain
  - Shouldn't comparison be against NewReno/SACK

TCP Vegas

- Flaws
  - Sensitivity to delay variation
  - Paper did not do great job of explaining where performance gains came from
- Some ideas have been incorporated into more recent implementations
- Overall
  - Some very intriguing ideas
  - Controversies killed it

Overview

- TCP Vegas
- TCP Modeling
- Other Congestion Control
- Changing Workloads
- Header Compression

Binomial Congestion Control

- In AIMD
  - Increase: $W_{n+1} = W_n + \alpha$
  - Decrease: $W_{n+1} = (1- \beta) W_n$
- In Binomial
  - Increase: $W_{n+1} = W_n + \alpha / W_n^k$
  - Decrease: $W_{n+1} = W_n - \beta W_n^l$
  - $k=0$ & $l=1$ → AIMD
  - $l < 1$ results in less than multiplicative decrease
    - Good for multimedia applications

Binomial Congestion Control

- Rate ~ 1/ (loss rate)$^{1/(k+l+1)}$
- If k+l=1 → rate ~ $1/p^{0.5}$
  - TCP friendly if l ☯ 1
- AIMD (k=0, l=1) is the most aggressive of this class
  - Good for applications that want to probe quickly and can use any available bandwidth

Next Lecture: Queue Management

- RED
- Blue
- Assigned reading
  - [FJ93] Random Early Detection Gateways for Congestion Avoidance
  - [Fen99] Blue: A New Class of Active Queue Management Algorithms

15-744: Computer Networking

L-11 Queue Management

Queue Management

- RED
- Blue
- Assigned reading
  - [FJ93] Random Early Detection Gateways for Congestion Avoidance
  - [Fen99] Blue: A New Class of Active Queue Management Algorithms

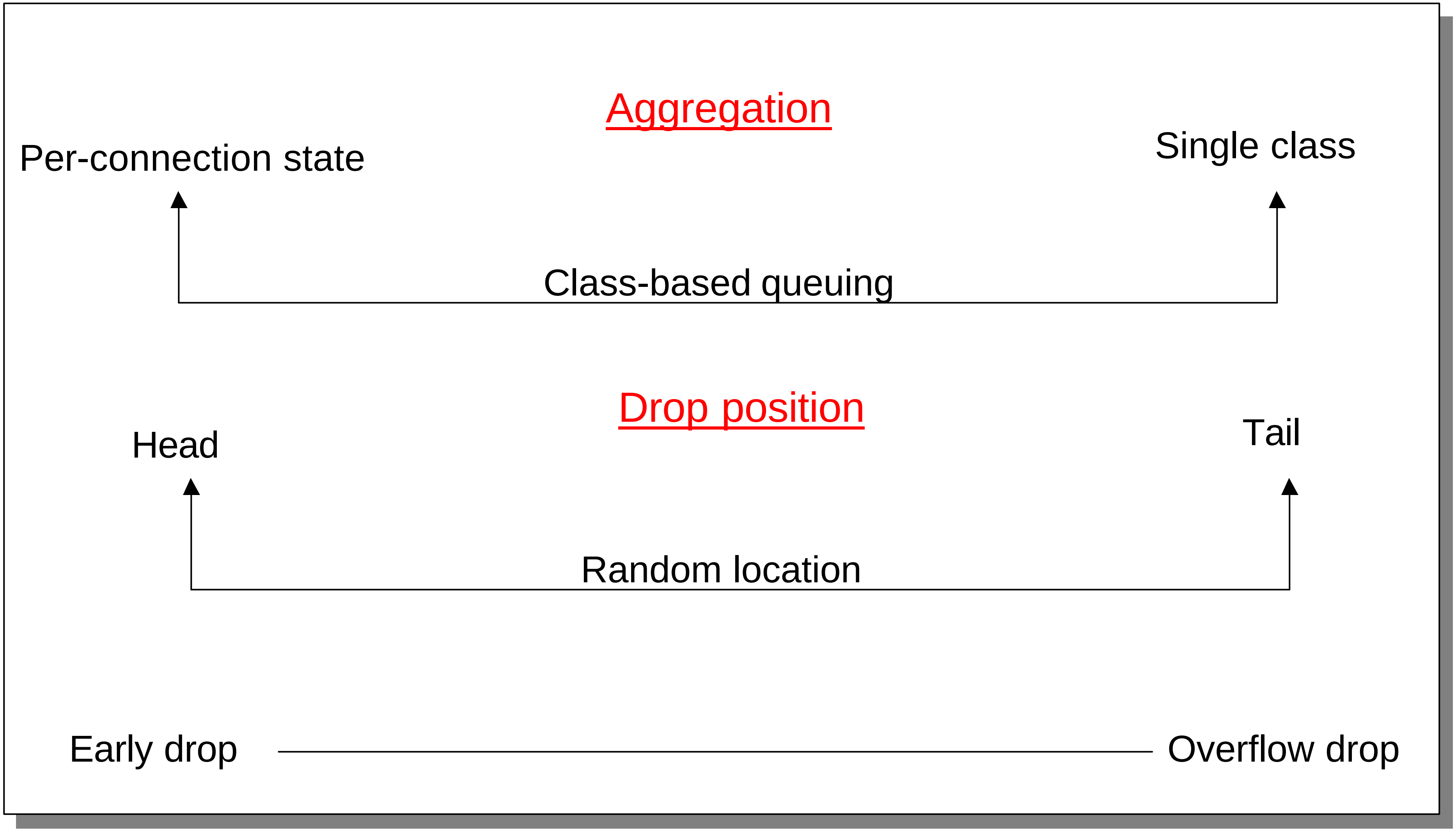
Overview

- Queuing Disciplines
- DECbit
- RED
- RED Alternatives
- BLUE

Queuing Disciplines

- Each router must implement some queuing discipline
- Queuing allocates both bandwidth and buffer space:
  - Bandwidth: which packet to serve (transmit) next
  - Buffer space: which packet to drop next (when required)
- Queuing also affects latency

Packet Drop Dimensions



Aggregation

Per-connection state

Single class

Class-based queuing

Drop position

Head

Tail

Random location

Early drop

Overflow drop

Typical Internet Queuing

- FIFO + drop-tail
  - Simplest choice
  - Used widely in the Internet
- FIFO (first-in-first-out)
  - Implies single class of traffic
- Drop-tail
  - Arriving packets get dropped when queue is full regardless of flow or importance
- Important distinction:
  - FIFO: scheduling discipline
  - Drop-tail: drop policy

FIFO + Drop-tail Problems

- Leaves responsibility of congestion control to edges (e.g., TCP)
- Does not separate between different flows
- No policing: send more packets → get more service
- Synchronization: end hosts react to same events

Active Queue Management

- Design active router queue management to aid congestion control
- Why?
  - Router has unified view of queuing behavior
  - Routers can distinguish between propagation and persistent queuing delays
  - Routers can decide on transient congestion, based on workload

Active Queue Designs

- Modify both router and hosts
  - DECbit -- congestion bit in packet header
- Modify router, hosts use TCP
  - Fair queuing
    - Per-connection buffer allocation
  - RED (Random Early Detection)
    - Drop packet or set bit in packet header as soon as congestion is starting

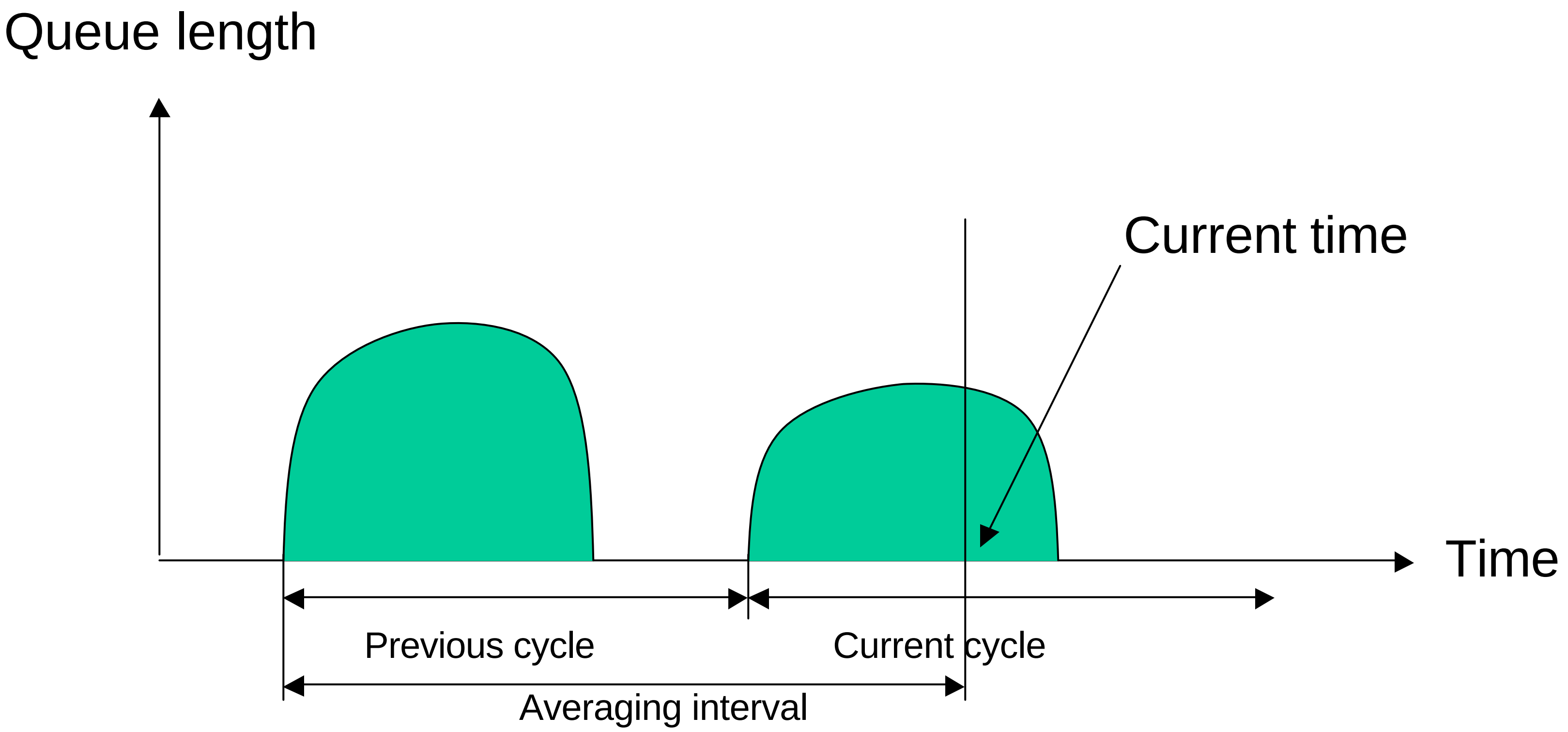
Overview

- Queuing Disciplines
- DECbit
- RED
- RED Alternatives
- BLUE

The DECbit Scheme

- Basic ideas:
  - On congestion, router sets congestion indication (CI) bit on packet
  - Receiver relays bit to sender
  - Sender adjusts sending rate
- Key design questions:
  - When to set CI bit?
  - How does sender respond to CI?

Setting CI Bit



AVG queue length = (previous busy+idle + current interval)/(averaging interval)

DECbit Routers

- Router tracks average queue length
  - Regeneration cycle: queue goes from empty to non-empty to empty
  - Average from start of previous cycle
  - If average > 1 → router sets bit for flows sending more than their share
  - If average > 2 → router sets bit in every packet
  - Threshold is a trade-off between queuing and delay
  - Optimizes power = (throughput / delay)
  - Compromise between sensitivity and stability
- Acks carry bit back to source

DECbit Source

- Source averages across acks in window
  - Congestion if > 50% of bits set
  - Will detect congestion earlier than TCP
- Additive increase, multiplicative decrease
  - Decrease factor = 0.875
    - Lower than TCP (1/2) – why?
  - Increase factor = 1 packet
  - After change, ignore DECbit for packets in flight (vs. TCP ignore other drops in window)
- No slow start

DECbit Evaluation

- Relatively easy to implement
- No per-connection state
- Stable
- Assumes cooperative sources
- Conservative window increase policy

Overview

- Queuing Disciplines
- DECbit
- RED
- RED Alternatives
- BLUE

Internet Problems

- Full queues
  - Routers are forced to have have large queues to maintain high utilizations
  - TCP detects congestion from loss
    - Forces network to have long standing queues in steady-state
- Lock-out problem
  - Drop-tail routers treat bursty traffic poorly
  - Traffic gets synchronized easily → allows a few flows to monopolize the queue space

Design Objectives

- Keep throughput high and delay low
- Accommodate bursts
- Queue size should reflect ability to accept bursts rather than steady-state queuing
- Improve TCP performance with minimal hardware changes

Lock-out Problem

- Random drop
  - Packet arriving when queue is full causes some random packet to be dropped
- Drop front
  - On full queue, drop packet at head of queue
- Random drop and drop front solve the lock-out problem but not the full-queues problem

Full Queues Problem

- Drop packets before queue becomes full (early drop)
- Intuition: notify senders of incipient congestion
  - Example: early random drop (ERD):
    - If qlen > drop level, drop each new packet with fixed probability *p*
    - Does not control misbehaving users

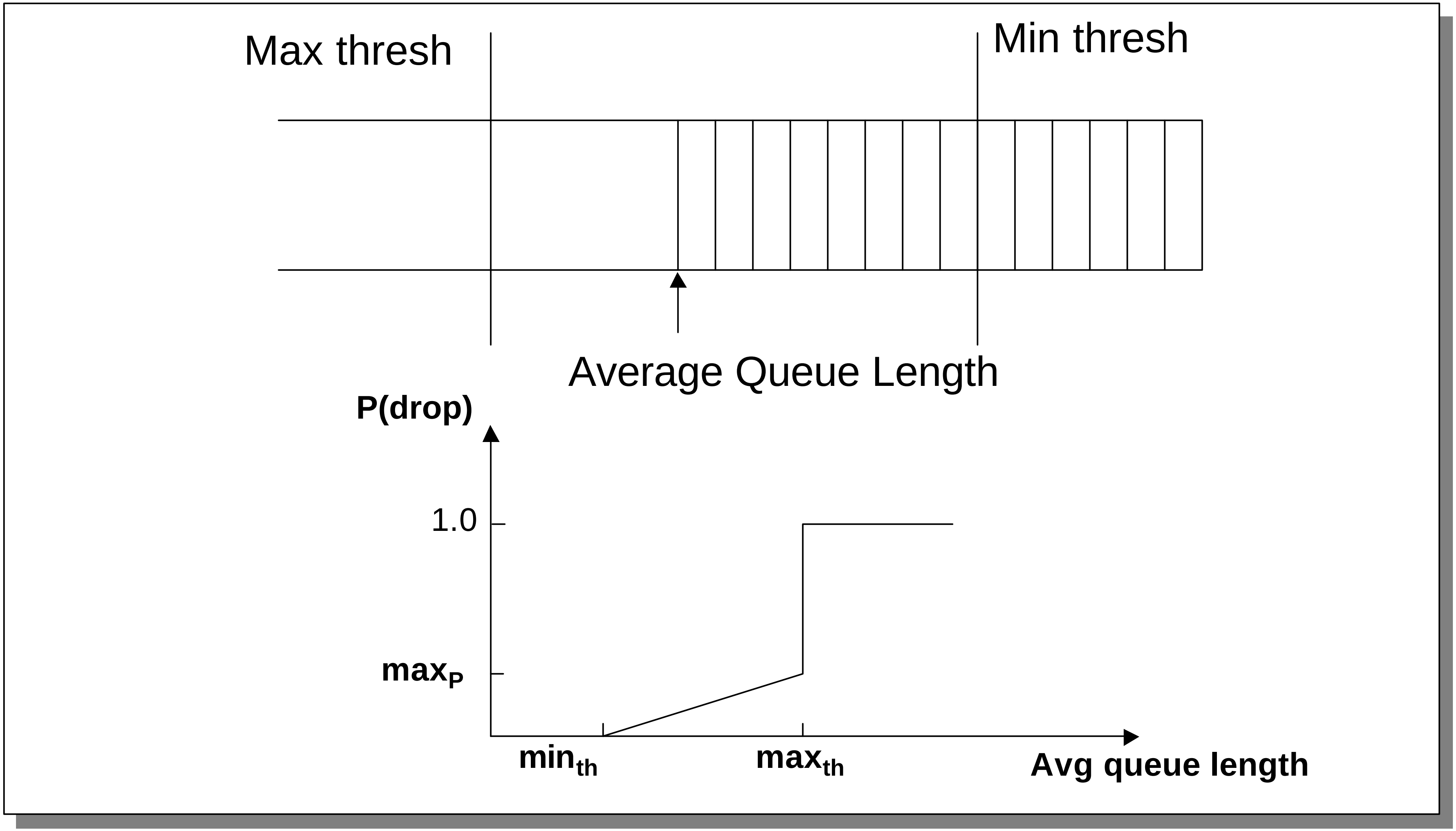
Random Early Detection (RED)

- Detect incipient congestion, allow bursts
- Keep power (throughput/delay) high
  - Keep average queue size low
  - Assume hosts respond to lost packets
- Avoid window synchronization
  - Randomly mark packets
- Avoid bias against bursty traffic
- Some protection against ill-behaved users

RED Algorithm

- Maintain running average of queue length
- If avg < $\min_{th}$ do nothing
  - Low queuing, send packets through
- If avg > $\max_{th}$, drop packet
  - Protection from misbehaving sources
- Else mark packet in a manner proportional to queue length
  - Notify sources of incipient congestion

RED Operation



RED Algorithm

- Maintain running average of queue length
  - Byte mode vs. packet mode – why?
- For each packet arrival
  - Calculate average queue size (avg)
  - If min_{th} ☯ avg < max_{th}
    - Calculate probability P_a
    - With probability P_a
      - Mark the arriving packet
    - Else if max_{th} ☯ avg
      - Mark the arriving packet

Queue Estimation

- Standard EWMA: avg - (1-w_q) avg + $w_q$qlen
  - Special fix for idle periods – why?
- Upper bound on w_q depends on min_{th}
  - Want to ignore transient congestion
  - Can calculate the queue average if a burst arrives
    - Set w_q such that certain burst size does not exceed min_{th}
- Lower bound on w_q to detect congestion relatively quickly
- Typical w_q = 0.002

Thresholds

- $\min_{th}$ determined by the utilization requirement
  - Tradeoff between queuing delay and utilization
- Relationship between $\max_{th}$ and $\min_{th}$
  - Want to ensure that feedback has enough time to make difference in load
  - Depends on average queue increase in one RTT
  - Paper suggest ratio of 2
    - Current rule of thumb is factor of 3

Packet Marking

- Marking probability based on queue length
  - $P_b = max_p(avg - min_{th}) / (max_{th} - min_{th})$
- Just marking based on P_b can lead to clustered marking
  - Could result in synchronization
  - Better to bias P_b by history of unmarked packets
  - $P_a = P_b/(1 - count*P_b)$

Packet Marking

- max_p is reflective of typical loss rates
- Paper uses 0.02
    - 0.1 is more realistic value
- If network needs marking of 20-30% then need to buy a better link!

Extending RED for Flow Isolation

- Problem: what to do with non-cooperative flows?
- Fair queuing achieves isolation using per-flow state – expensive at backbone routers
  - How can we isolate unresponsive flows without per-flow state?
- RED penalty box
  - Monitor history for packet drops, identify flows that use disproportionate bandwidth
  - Isolate and punish those flows

Overview

- Queuing Disciplines
- DEC-bit
- RED
- RED Alternatives
- BLUE

FRED

- Fair Random Early Drop (Sigcomm, 1997)
- Maintain per flow state only for active flows (ones having packets in the buffer)
- min_q and max_q → min and max number of buffers a flow is allowed occupy
- avgcq = average buffers per flow
- Strike count of number of times flow has exceeded max_q

FRED – Fragile Flows

- Flows that send little data and want to avoid loss
- min_q is meant to protect these
- What should min_q be?
    - When large number of flows → 2-4 packets
        - Needed for TCP behavior
    - When small number of flows → increase to avgcq

FRED

- Non-adaptive flows
  - Flows with high strike count are not allowed more than avgcq buffers
  - Allows adaptive flows to occasionally burst to $\max_q$ but repeated attempts incur penalty
- Fixes to queue averaging
  - RED only modifies average on packet arrival
  - What if queue is 500 and slowly empties out?
    - Add averaging on exit as well

CHOKe

- CHOse and Keep/Kill (Infocom 2000)
  - Existing schemes to penalize unresponsive flows (FRED/penalty box) introduce additional complexity
  - Simple, stateless scheme
- During congested periods
  - Compare new packet with random pkt in queue
  - If from same flow, drop both
  - If not, use RED to decide fate of new packet

CHOKe

- Can improve behavior by selecting more than one comparison packet
  - Needed when more than one misbehaving flow
- Does not completely solve problem
  - Aggressive flows are punished but not limited to fair share

Overview

- Queuing Disciplines
- DEC-bit
- RED
- RED Alternatives
- BLUE

Blue

- Uses packet loss and link idle events instead of average queue length – why?
  - Hard to decide what is transient and what is severe with queue length
  - Based on observation that RED is often forced into drop-tail mode
  - Adapt to how bursty and persistent congestion is by looking at loss/idle events

Blue

- Basic algorithm
  - Upon packet loss, if no update in freeze_time then increase p_m by d1
  - Upon link idle, if no update in freeze_time then decrease p_m by d2
  - d1 >> d2 → why ?
    - More critical to react quickly to increase in load

Comparison: Blue vs. RED

- max_p set to 1
  - Normally only 0.1
  - Based on type of tests & measurement objectives
    - Want to avoid loss → marking is not penalized
    - Enough connections to ensure utilization is good
    - Is this realistic though?
- Blue advantages
  - More stable marking rate & queue length
  - Avoids dropping packets
  - Much better behavior with small buffers

Stochastic Fair Blue

- Same objective as RED Penalty Box
  - Identify and penalize misbehaving flows
- Create L hashes with N bins each
  - Each bin keeps track of separate marking rate (p_m)
  - Rate is updated using standard technique and a bin size
  - Flow uses minimum p_m of all L bins it belongs to
  - Non-misbehaving flows hopefully belong to at least one bin without a bad flow
    - Large numbers of bad flows may cause false positives

Stochastic Fair Blue

- Is able to differentiate between approx. N^L flows
- Bins do not actually map to buffers
  - Each bin only keeps drop rate
  - Can statistically multiplex buffers to bins
  - Works well since Blue handles small queues
  - Has difficulties when large number of misbehaving flows

Stochastic Fair Blue

- False positives can continuously penalize same flow
- Solution: moving hash function over time
  - Bad flow no longer shares bin with same flows
  - Is history reset →does bad flow get to make trouble until detected again?
    - No, can perform hash warmup in background

Next Lecture: Fair Queuing

- Fair Queuing
- Core-stateless Fair queuing
- Assigned reading
  - [DKS90] Analysis and Simulation of a Fair Queueing Algorithm, Internetworking: Research and Experience
  - [SSZ98] Core-Stateless Fair Queueing: Achieving Approximately Fair Allocations in High Speed Networks

TCP Futures

- Throughput in terms of loss rate

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

- ? L = 2?10^{-10} *Wow*
- New versions of TCP for high-speed needed!