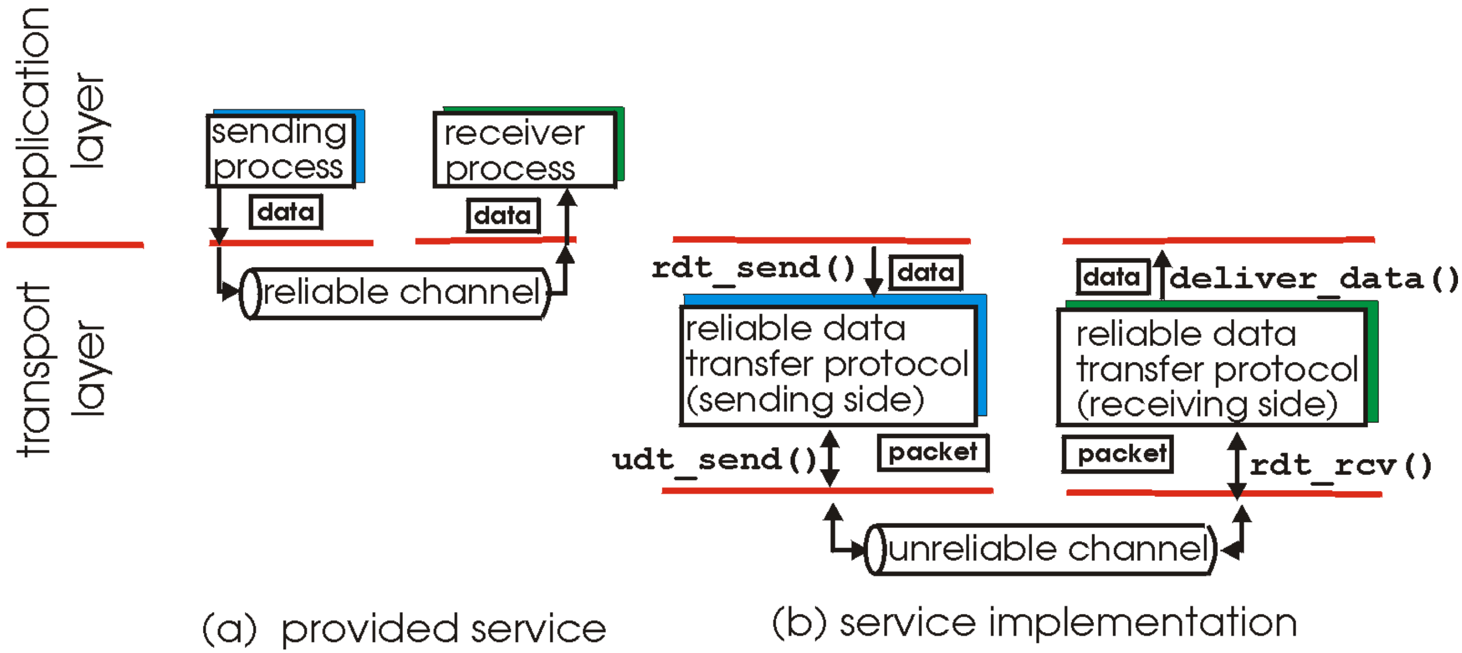
Sliding Window Protocol and TCP Congestion Control

Simon S. Lam
Department of Computer Science
The University of Texas at Austin

Reliable data transfer

- important in app., transport, link layers



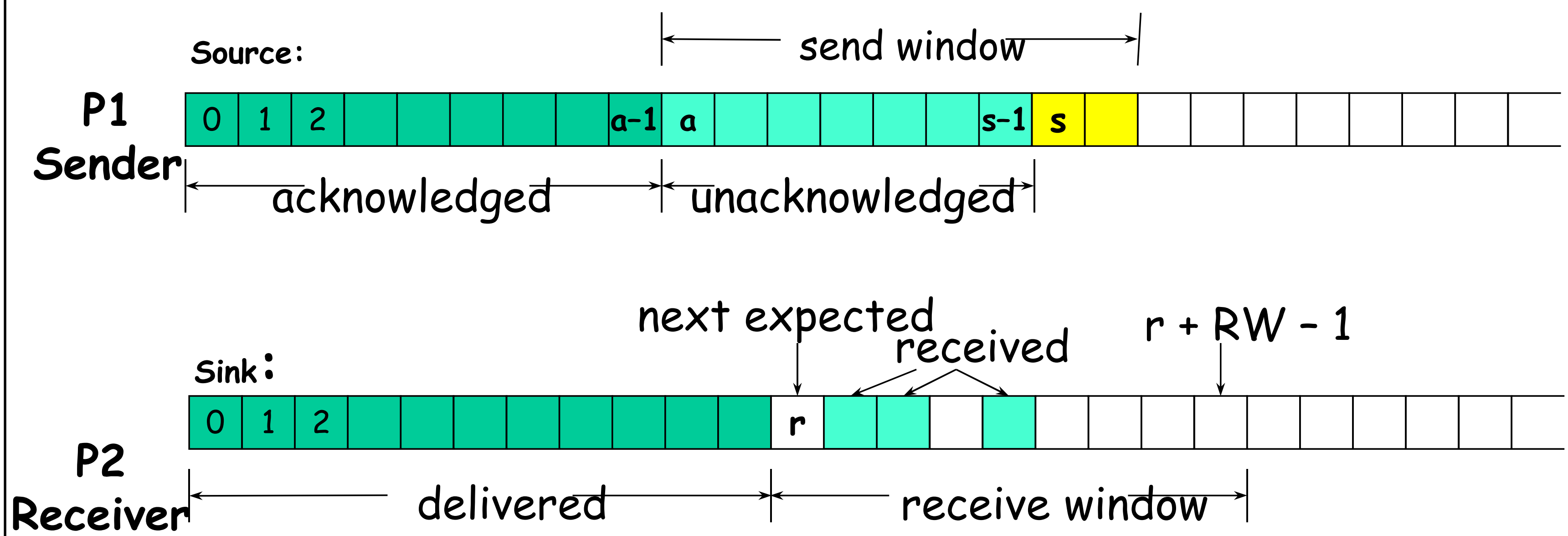
- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Channel Abstractions

- Lossy FIFO channel
  - delivers a subsequence in FIFO order
  - example: delivery service provided by a physical link
- Lossy, reordering, duplicative (LRD) channel
  - example: delivery service provided by IP or by UDP protocol

Sliding Window Protocol

- Consider an infinite array, Source, at the sender, and an infinite array, Sink, at the receiver.

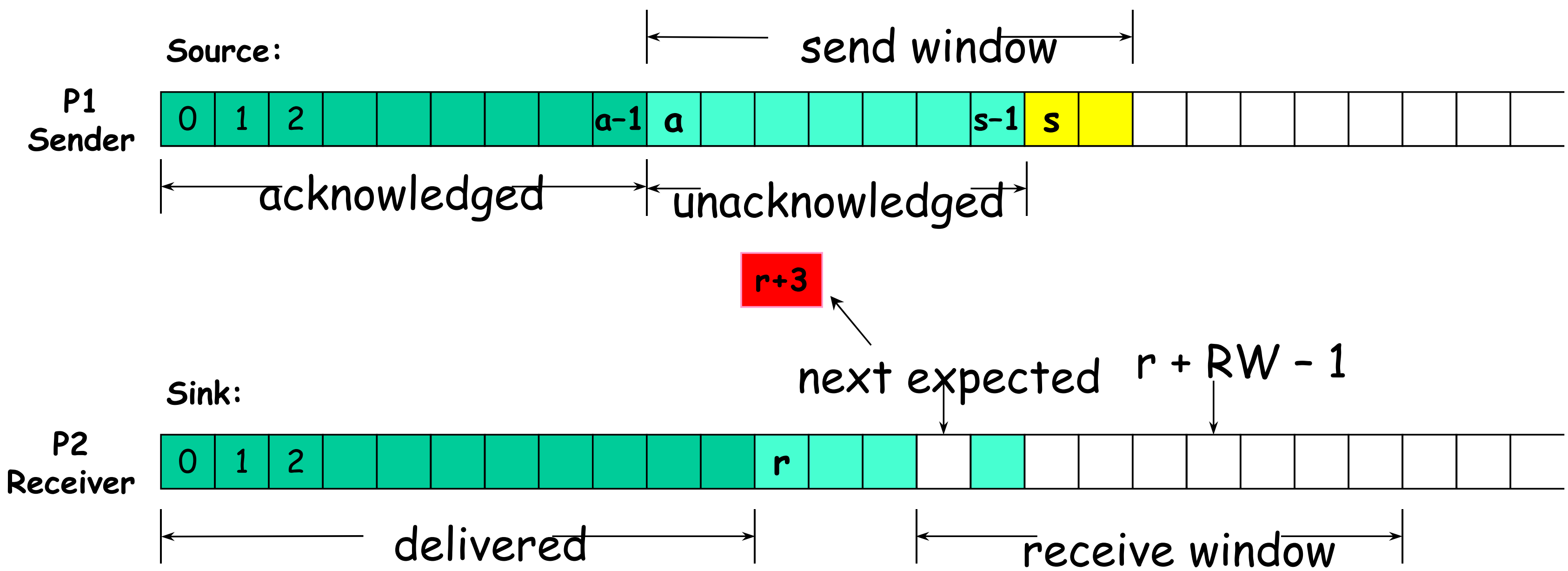


SW send window size $(s - a \leq SW)$

RW receive window size

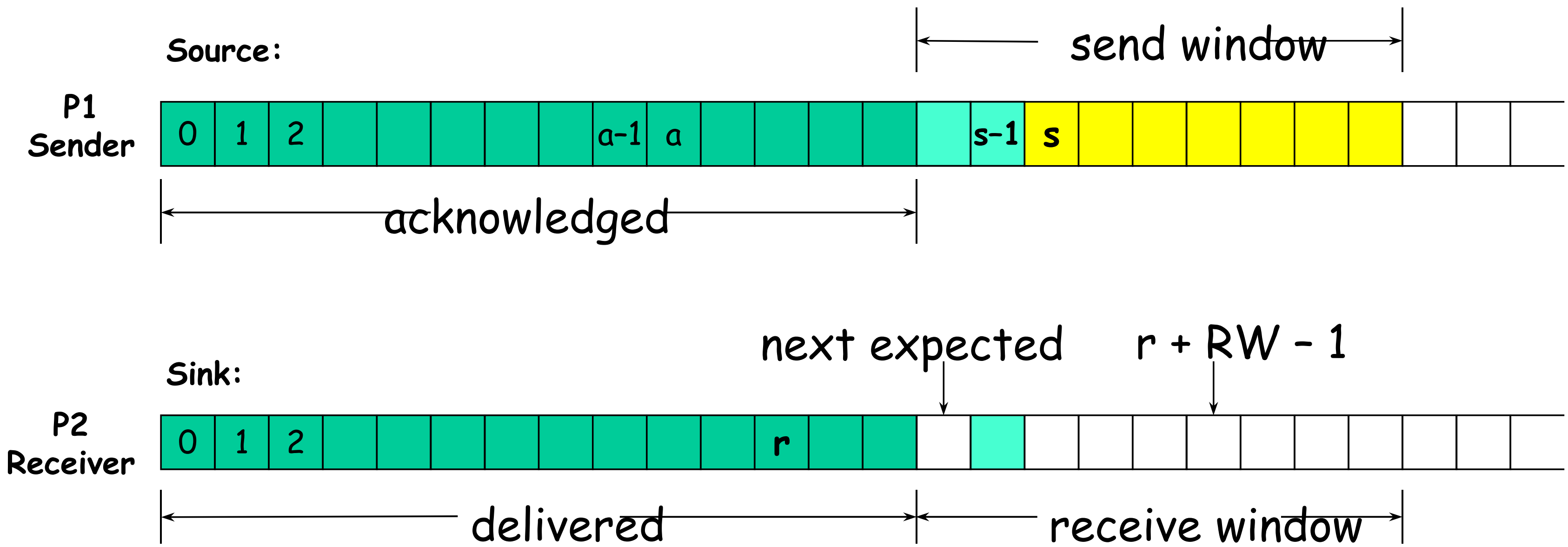
Sliding Windows in Action

- Data unit r has just been received by P2
  - Receive window slides forward
- P2 sends cumulative ack with sequence number it expects to receive next (r+3)



Sliding Windows in Action

- P1 has just received cumulative ack with r+3 as next expected sequence number
  - Send window slides forward



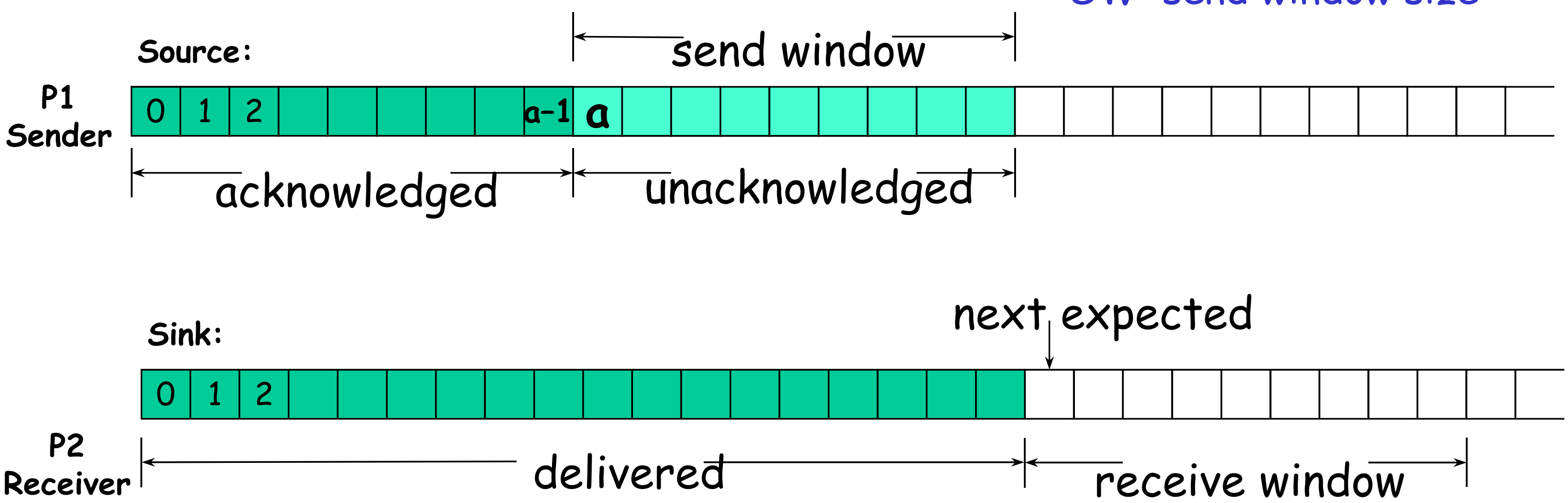
Sliding Window protocol

- Functions provided
  - error control (reliable delivery)
  - in-order delivery
  - flow and congestion control (by varying send window size)
- TCP uses cumulative acks (needed for correctness)
- Other kinds of acks (to improve performance)
  - selective nack
  - selective ack (TCP SACK)
  - bit-vector representing entire state of receive window (in addition to first sequence number of window)

Sliding Windows for Lossy FIFO Channels

- A small number of bits in packet header for sequence number
- Necessary and sufficient condition for correct operation: SW + RW ≤ MaxSeqNum
- *Necessity*:

RW receive window size
SW send window size



Sliding Windows for Lossy FIFO Channels

- Sufficiency can only be demonstrated by using a formal method to prove that the protocol provides reliable in-order delivery. See Shankar and Lam, *ACM TOPLAS*, Vol. 14, No. 3, July 1992.
- Interesting special cases
  - SW = RW = 1

    alternating-bit protocol
  - SW = 7, RW = 1

    out-of-order arrivals not accepted, e.g., HDLC
  - SW = RW

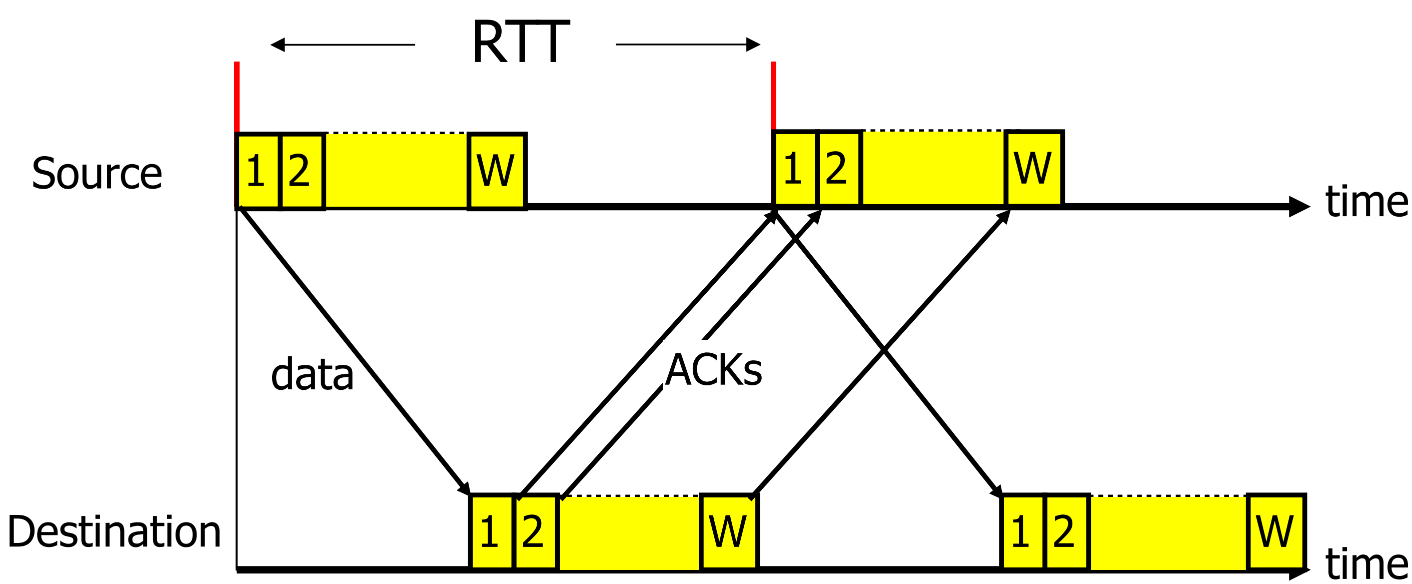
Sliding Windows for LRD Channels

- Assumption: Packets have bounded lifetime L
- Be careful how fast sequence numbers are consumed (i.e., by arrival of data to be sent into network)

  $$(\text{send rate}) \times L < \text{MaxSeqNum}$$

- TCP
  - 32-bit sequence numbers
  - counts bytes
  - assumes that datagrams will be discarded by IP if too old

Window Size Controls Sending Rate



- ~ W packets per RTT when no loss

Throughput

- Limit the number of unacked transmitted packets in the network to window size W

- Max. throughput $\simeq \dfrac{W}{RTT}$ packets/sec

$$= \frac{W \times MSS}{RTT} \text{ bytes/sec}$$

(assuming no loss, *MSS* denotes maximum segment size)

- Where did we apply Little's Law?

  Answer : Consider the TCP send buffer

Throughput or send rate?

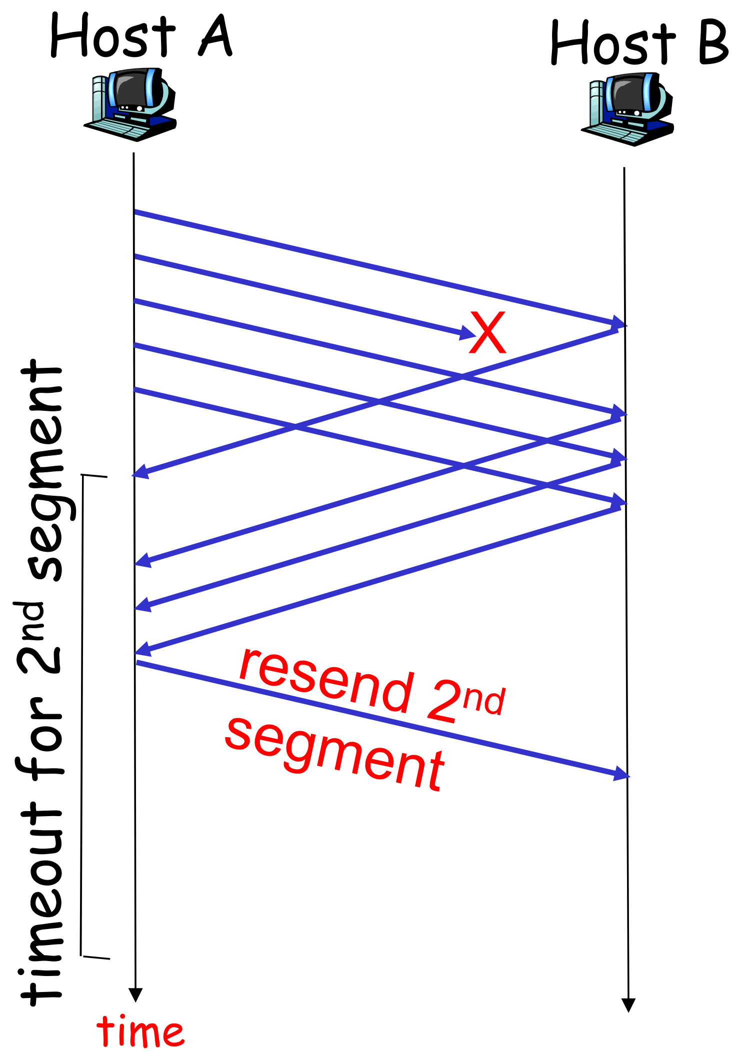
- Previous formula actually provides an upper bound
  - Average number in the send buffer is less than W unless packet arrival rate to send buffer is infinite
  - If a packet is lost in the network with probability p, then the average time in send buffer is $(1-p)\times RTT + p\times T_O$
    Since T_O > RTT, actual throughput is smaller.

- The throughput of a host's TCP send buffer is the host's send rate into the network (including original transmissions and retransmissions)

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 *duplicate ACKs*** for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment *before timer expires*

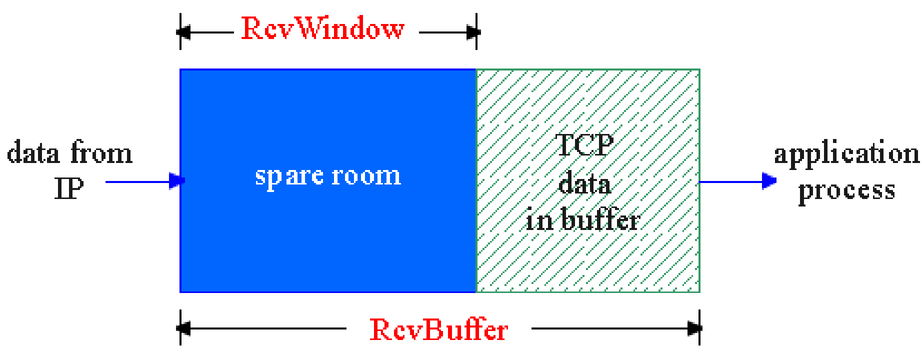


Resending a segment after triple duplicate ACK without waiting for timeout

TCP Flow Control

flow control

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



buffer at receive side of a TCP connection

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

- `RcvWindow` **field** in TCP segment header

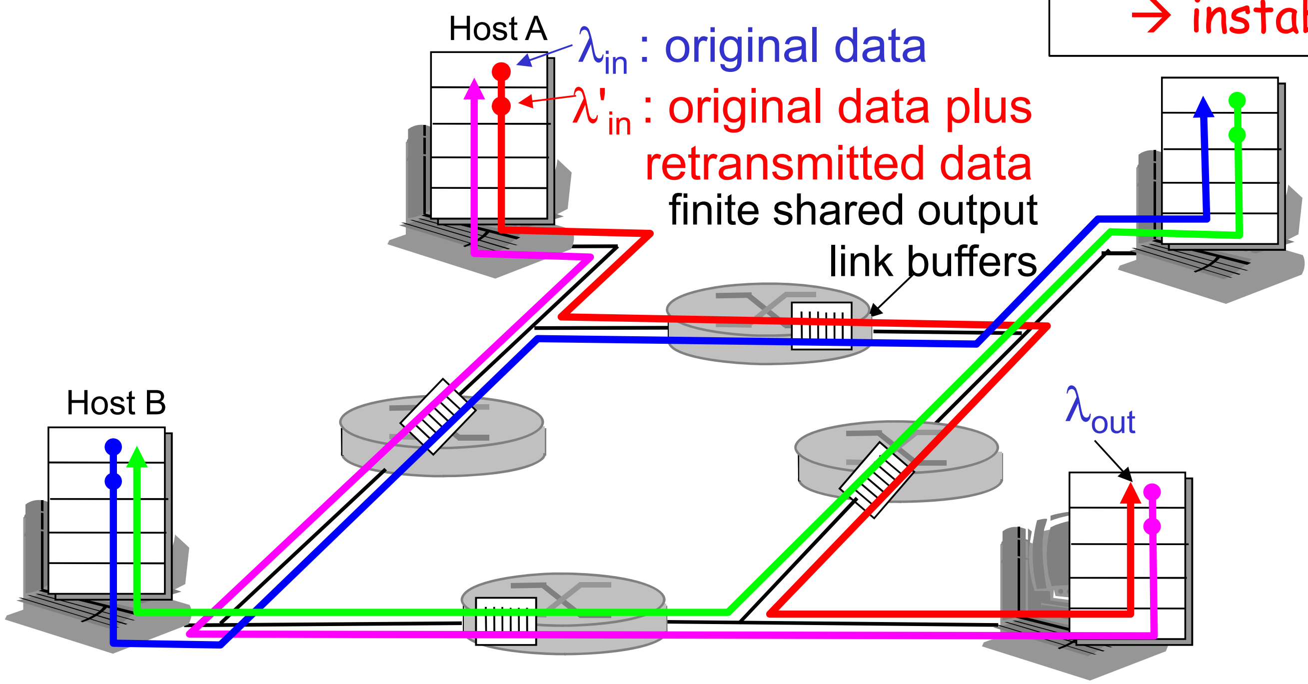
sender: keeps amount of transmitted, unACKed data less than most recently received `RcvWindow` **value**

Causes/costs of congestion: scenario

- four senders
- multi-hop paths
- Timeout & retransmit

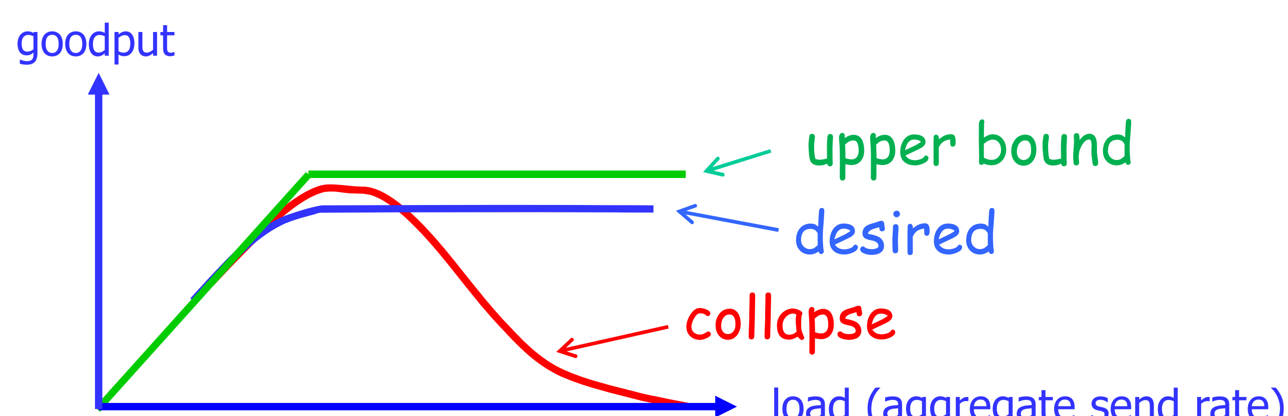
Q: what happens as λ_{in} and λ'_{in} increase at many senders?

positive feedback → instability



Effect of Congestion

- W too big for many flows -> congestion
- Packet loss -> transmissions on links a packet has traversed prior to loss are wasted
- Congestion collapse due to too many retransmissions and too much wasted transmission capacity
- October 1986, Internet had its first congestion collapse



TCP Window Control

- **Receiver *flow control***
  - Avoid overloading receiver
  - rcvwindow: receiver's advertised window (also rwnd)
  - Receiver sends rcvwindow to sender
- **Network *congestion control***
  - Sender tries to avoid overloading network
  - It infers network congestion from "loss indications"
  - congwin: congestion window (also cwnd)
- **Sender sets W = min (congwin, rcvwindow)**

TCP Congestion Control

- end-to-end control (no network assistance)
- sender limits transmission:
  `LastByteSent-LastByteAcked` $\leq$ `CongWin`
- Roughly, the send buffer's

$$\text{throughput} \leq \frac{\text{CongWin}}{\text{RTT}} \text{ bytes/sec}$$

where CongWin is in bytes

How does sender determine CongWin?

- loss event = **timeout** *or* **3 duplicate acks**
- TCP sender reduces `CongWin` after a loss event

three mechanisms:

- slow start
- reduce to 1 segment after timeout event
- AIMD (additive increase multiplicative decrease)

Note: For now consider RcvWindow to be very large such that the send window size is equal to CongWin.

TCP Slow Start

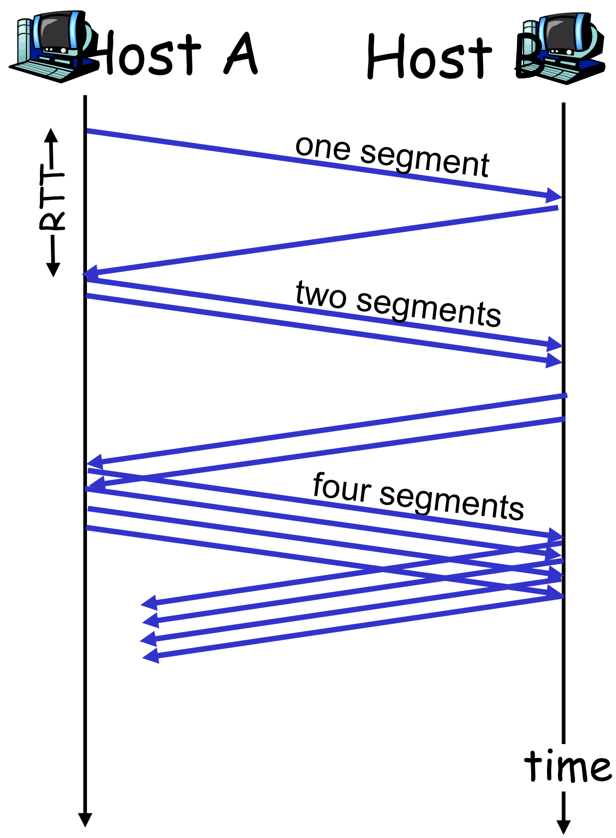
- Probing for usable bandwidth

- When connection begins, `CongWin` = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 2500 bytes/sec = 20 kbps

- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to a higher rate

TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event or "threshold"
  - double `CongWin` every RTT
  - done by incrementing `CongWin` by 1 MSS for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



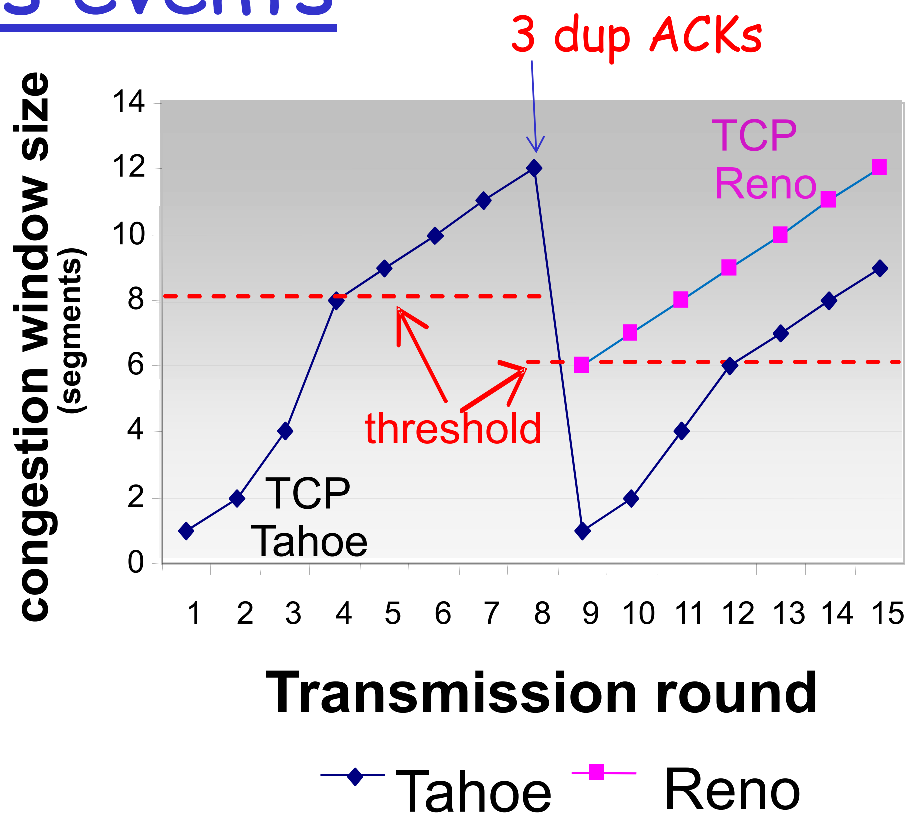
Congestion avoidance state & responses to loss events

Q: If no loss, when should the exponential increase switch to linear?

A: When `CongWin` gets to current value of threshold

Implementation:

- For initial slow start, threshold is set to a large value (e.g., 64 Kbytes)
- Subsequently, threshold is variable
- *At a loss event, threshold is set to 1/2 of CongWin just before loss event*



Note: For simplicity, CongWin is in number of segments in the above graph.

Rationale for Reno's Fast Recovery

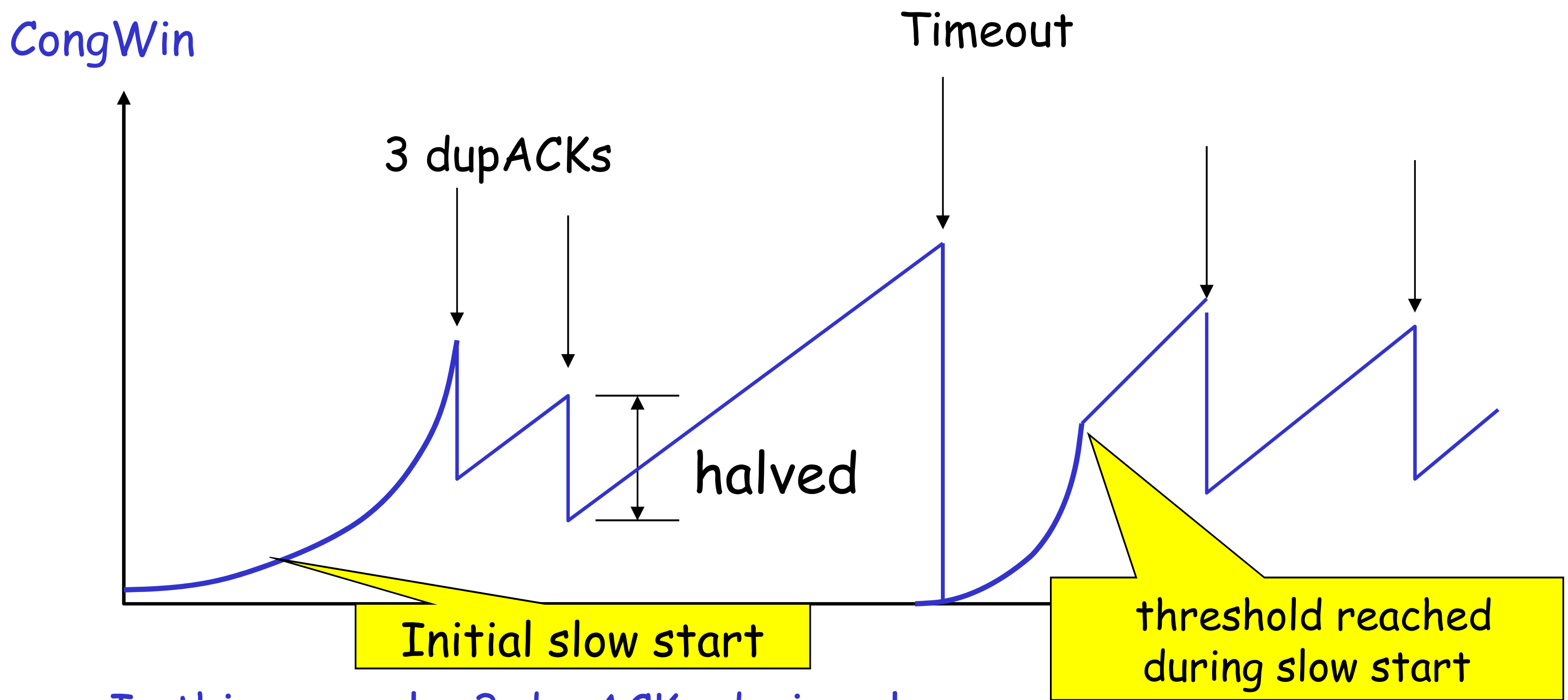
- 3 dup ACKs indicate network capable of delivering some segments

- timeout occurring before 3 dup ACKs is "more alarming"

- After 3 dup ACKs:
  - `CongWin` is cut in half (*multiplicative decrease*)
  - window then grows linearly (*additive increase*)
- But after timeout event:
  - `CongWin` is set to 1 MSS instead;
  - window then grows exponentially to threshold, then grows linearly

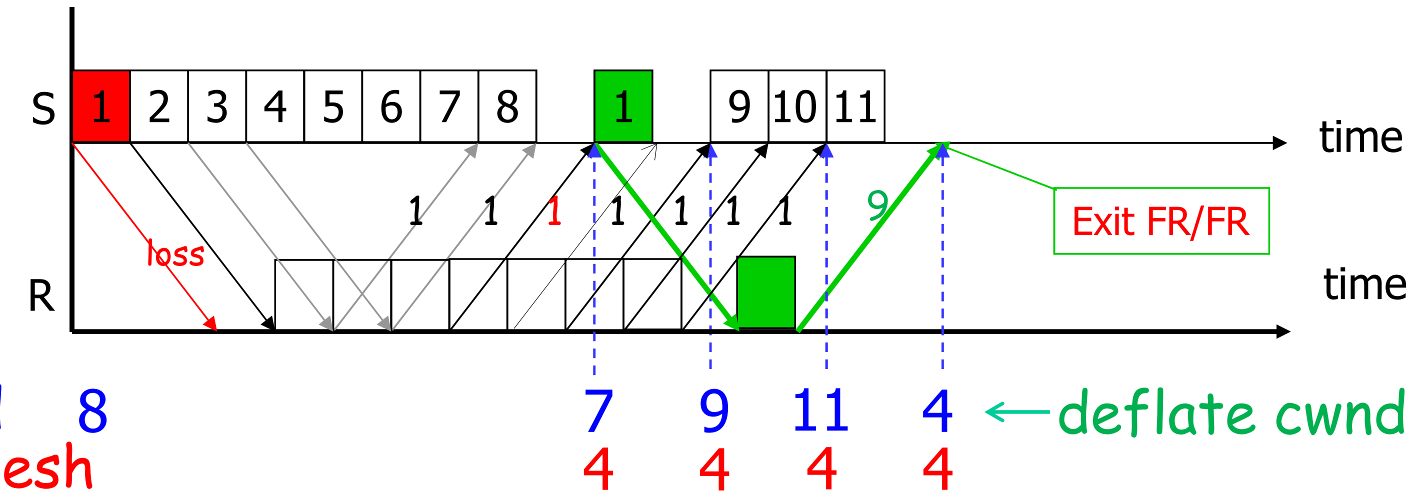
Additive Increase Multiplicative Decrease (AIMD)

TCP Reno (example scenario)



In this example, 3 dupACKs during slow start before reaching initial threshold

Example: FR/FR entry and exit



- Above scenario: Packet 1 is lost, packets 2, 3, and 4 are received; 3 dupACKs with seq. no. 1 returned
- Fast retransmit
  - Retransmit packet 1 upon 3 dupACKs
- Fast recovery (in steps)
  - Inflate cwnd with #dupACKs such that new packets 9, 10, and 11 can be sent while repairing loss

FR/FR (in more detail)

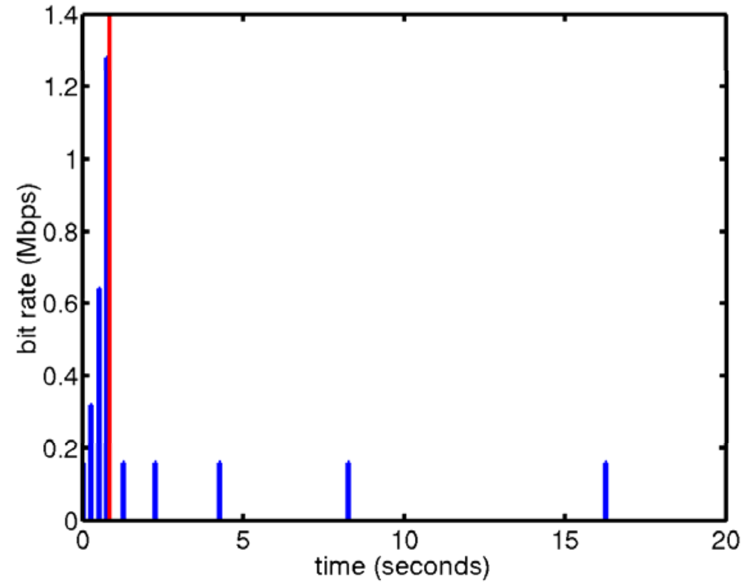
- Enter FR/FR after 3 dupACKs
  - Set ssthresh $\leftarrow$ max(flightsize/2, 2)
  - Retransmit lost packet
  - Set cwnd $\leftarrow$ ssthresh + #dupACKs (window inflation)
  - Wait till W=min(rwnd, cwnd) is large enough; transmit new packet(s)
  - On non-dup ACK (1 RTT later), set cwnd $\leftarrow$ ssthresh (window deflation)
- Enter Congestion Avoidance

Summary: TCP Congestion Control (Reno)

- When `CongWin` is below `Threshold`, sender in slow-start phase, window grows exponentially (*until loss event or exceeding threshold*).
- When `CongWin` is above `Threshold`, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, `Threshold` set to `CongWin/2` and `CongWin` set to `Threshold` (*also fast retransmit*)
- When timeout occurs, `Threshold` set to `CongWin/2` and `CongWin` is set to 1 MSS.

Successive Timeouts

- When there is another timeout, double the timeout value
- Keep doing so for each additional loss-retransmission
  - Exponential backoff up to max timeout value equal to 64 times initial timeout value



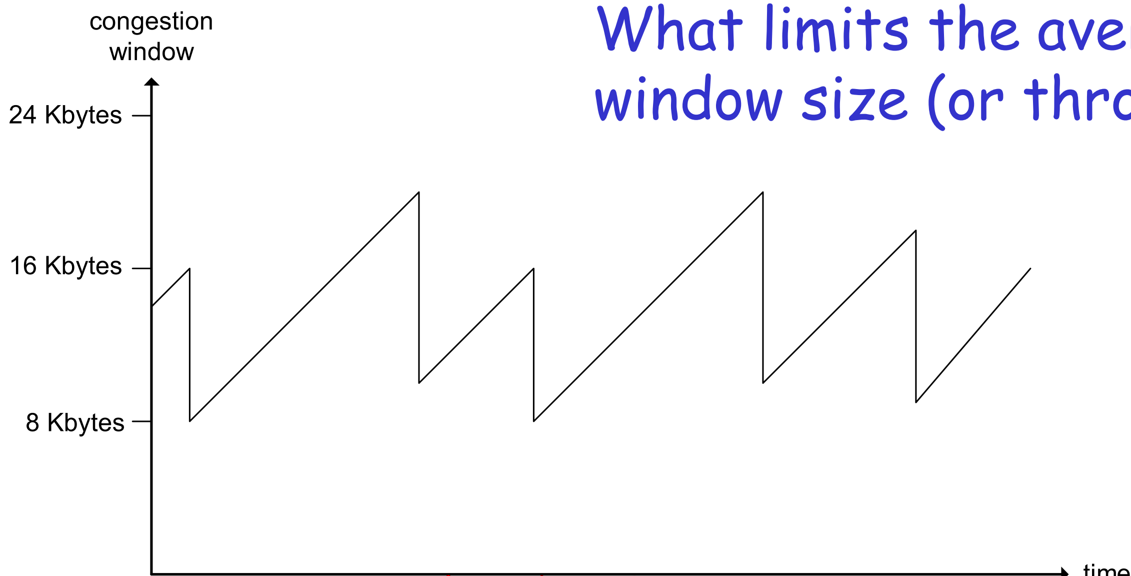
(There are other variations.)

Note: red line in figure denotes first timeout

AIMD in steady state (when no timeout)

additive increase:
increase `CongWin` by 1 MSS every RTT in the absence of any loss event: probing

multiplicative decrease:
cut `CongWin` in half after loss event (3 dup acks)



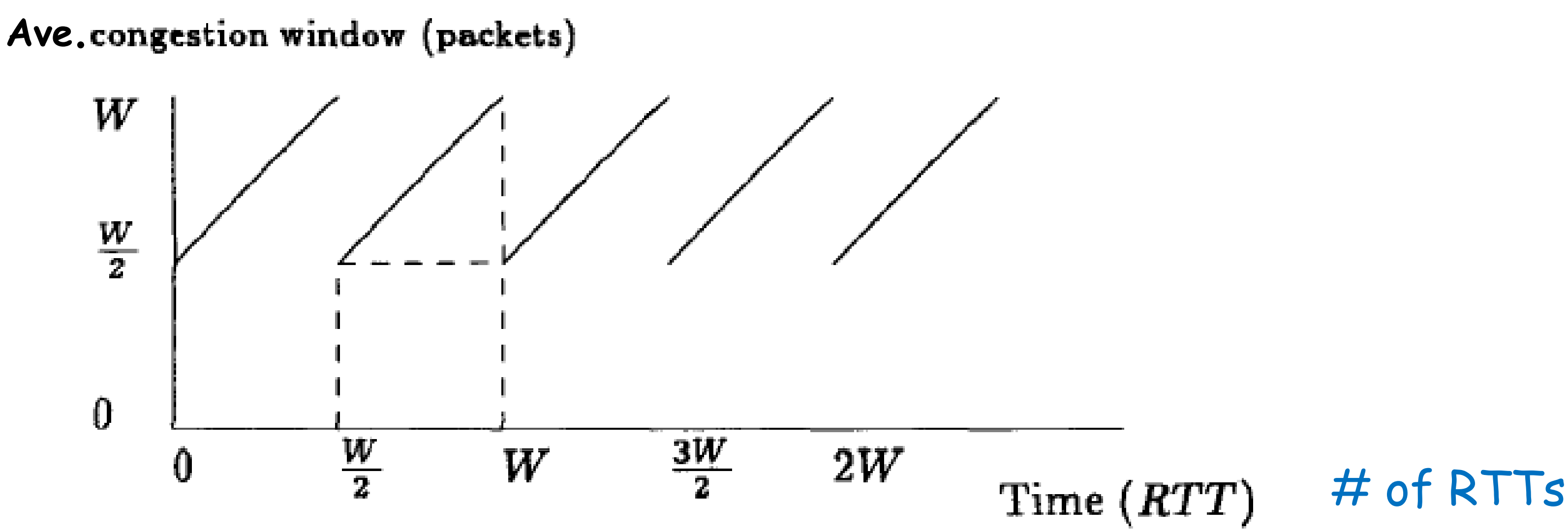
Long-lived TCP connection

What limits the average window size (or throughput)?

First approximation

M. Mathis, et al., "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm," *ACM Computer Communicatons Review*, 27(3), 1997.

- No slow-start, no timeout, long-lived TCP connection
- Independent identically distributed "periods"
- Three dupACKs are received in a round with probability p



Geometric Distribution

Independent trials - a trial fails with probability p

Ave. no. of transmissions to get first "failure"

$$\boxed{\bar{n} \;=\; \sum_{i=1}^{\infty} i b_i = \sum_{i=1}^{\infty} i(1-p)^{i-1}p}$$

$$\boxed{= \; p\sum_{i=1}^{\infty} i(1-p)^{i-1}}$$

$$\boxed{= \; -p\frac{d}{dp}\sum_{i=1}^{\infty}(1-p)^{i} = -p\frac{d}{dp}\sum_{i=0}^{\infty}(1-p)^{i}}$$

$$\boxed{= \; -p\frac{d}{dp}\frac{1}{1-1+p} = p\frac{1}{p^2}}$$

$$= \; 1/p$$

Ave. no. of trials to get first "success" is

1/(1-p)

First approximation (cont.)

- Average number of packets delivered in one period (area under one saw-tooth)

$$\left(\frac{W}{2}\right)^2 + \frac{1}{2}\left(\frac{W}{2}\right)^2 = \frac{3}{8}W^2$$

- Average number of packets sent per period is 1/p
- Equate the two and solve for W, we get $W = \sqrt{\frac{8}{3p}}$

send rate (in packets/sec)

$$= \frac{\text{no. of packets/period}}{\text{time per period}} = \boxed{\frac{\frac{3}{8}W^2}{RTT\left(\frac{W}{2}\right)}}$$

$$= \boxed{\frac{1/p}{RTT\left(\sqrt{\frac{2}{3p}}\right)}} = \boxed{\frac{1}{RTT}\sqrt{\frac{3}{2p}}}$$

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send *duplicate ACK*, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

Receiver implements Delayed ACKs

- Receiver sends one ACK for every two packets received -> each saw-tooth is WxRTT wide
  -> area under a saw-tooth is $\dfrac{3W^2}{4} = \dfrac{1}{p}$
- Send rate is $\dfrac{1/p}{RTT \cdot W} = \dfrac{1/p}{RTT \cdot \sqrt{4/(3p)}} = \dfrac{1}{RTT}\sqrt{\dfrac{3}{4p}}$
- One ACK for every *b* packets received -> send rate is

$$\boxed{\frac{1}{RTT}\sqrt{\frac{3}{2bp}}}$$

Challenges in the future

- TCP average throughput (approximate) in terms of loss rate, *p*

$$\frac{1.22 \cdot MSS}{RTT\sqrt{p}} \qquad \text{for } b = 1$$

- Example: 1500-byte segments, 100ms RTT, to get 10 Gbps throughput, loss rate needs to be very low

  $p = 2 \times 10^{-10}$

- New versions of TCP needed for connections with large delay-bandwidth product
  - E.g., data center networks (local, global)

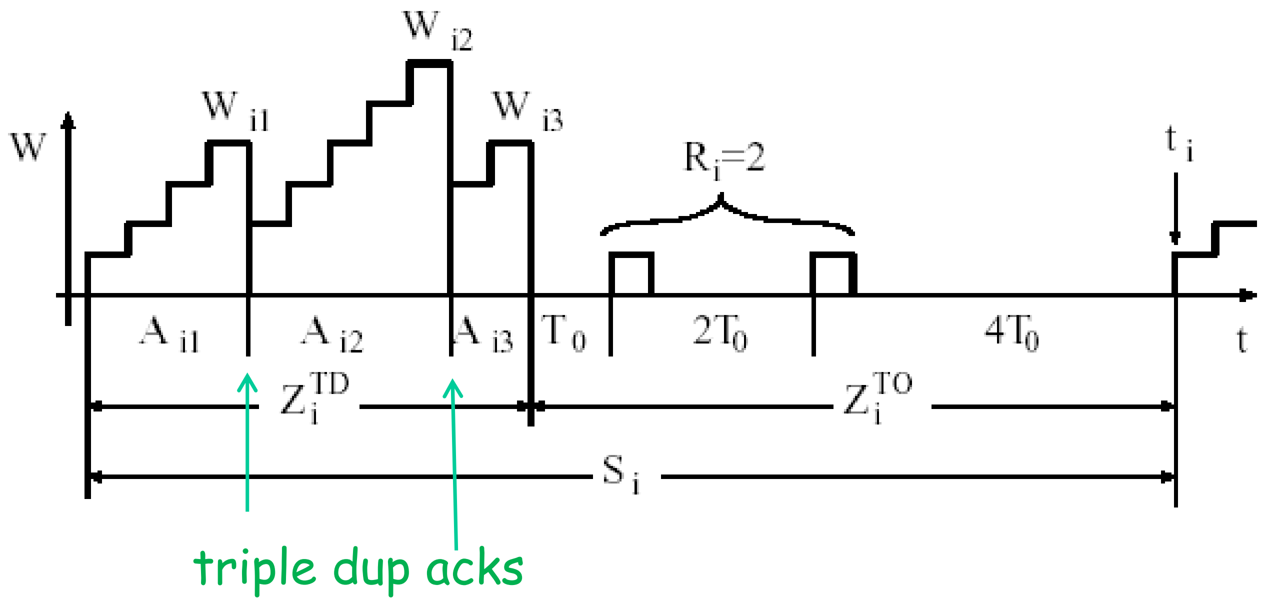
A more detailed model

Reference:
J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation," *Proceedings ACM SIGCOMM*, 1998.

Motivation

- Previous formulas not so accurate when loss rates are high
- TCP traces show that there are more loss indications due to timeouts (TO) than due to triple dupACKs (TD)

AIMD with Timeouts

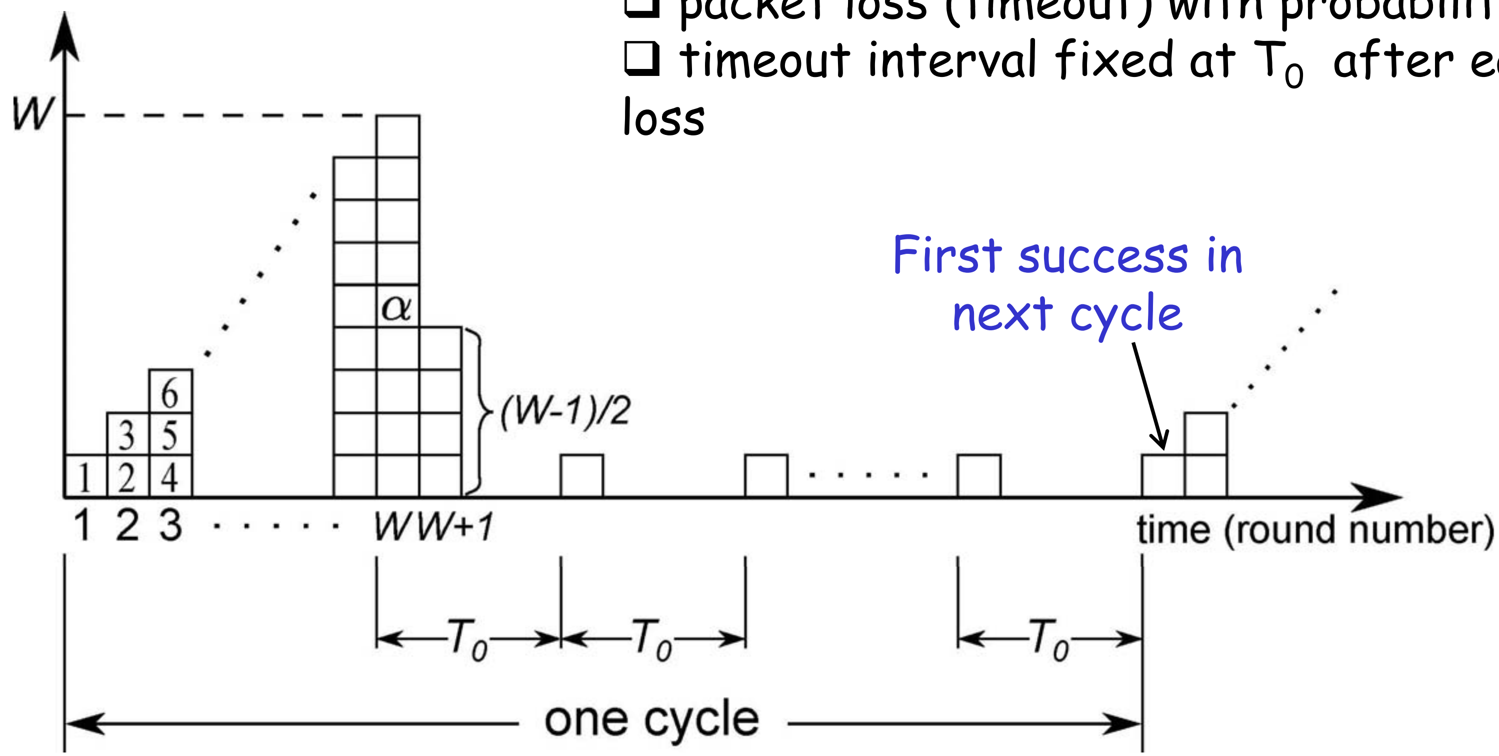


- No slow start
- b = 1 (no delayed ack)

Problem 3 in HW #2

Simplified:

- no triple duplicate Acks
- packet loss (timeout) with probability p
- timeout interval fixed at T_0 after each loss



The End