Network Transport Layer: TCP

Qiao Xiang, Congming Gao, Qiang Su

https://sngroup.org.cn/courses/cnns-xmuf25/index.shtml

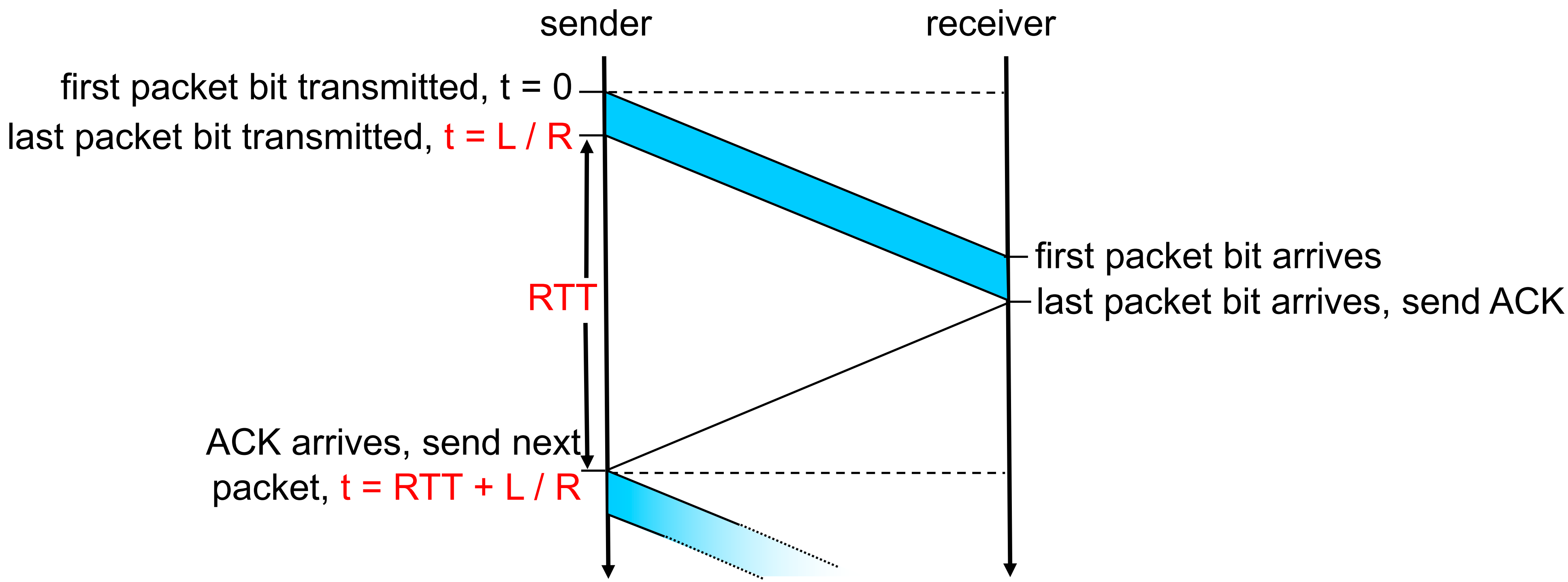
11/04/2025

This deck of slides are heavily based on CPSC 433/533 at Yale University, by courtesy of Dr. Y. Richard Yang.

Outline

- Admin and recap
- TCP

rdt3.0: Stop-and-Wait Performance

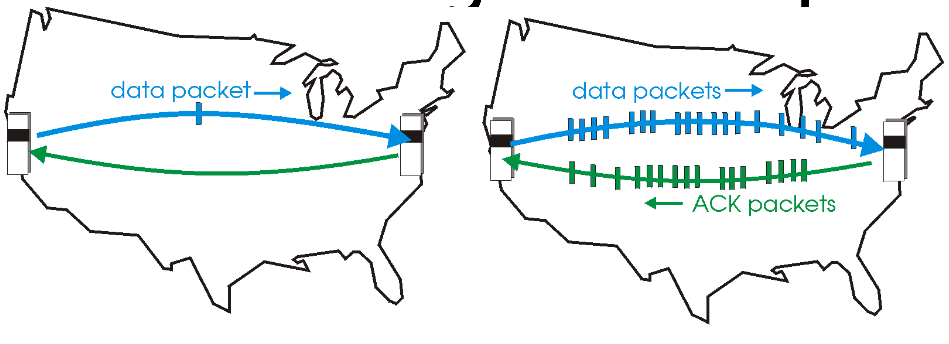


What is U_{sender}: utilization – fraction of time link busy sending?

Assume: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet

Recap: Reliable Transport

- Basic structure: sliding window protocols



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

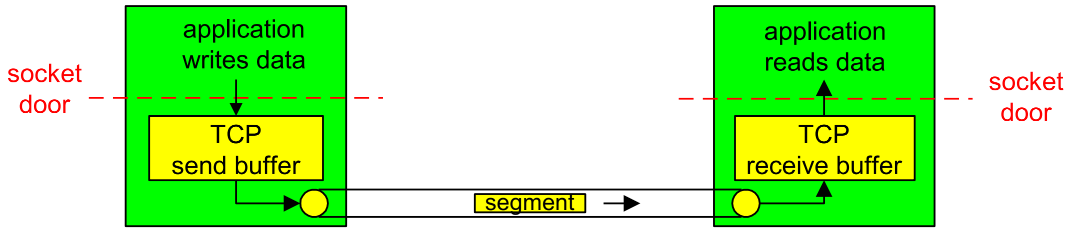
General technique: pipelining.

- Realization: GBN or SR

	Go-back-n	Selective Repeat
data bandwidth: sender to receiver (avg. number of times a pkt is transmitted)	Less efficient $\frac{1-p+pw}{1-p}$	More efficient $\frac{1}{1-p}$
ACK bandwidth (receiver to sender)	More efficient	Less efficient
Relationship between M (the number of seq#) and W (window size)	M > W	M ≥ 2W
Buffer size at receiver	1	W
Complexity	Simpler	More complex

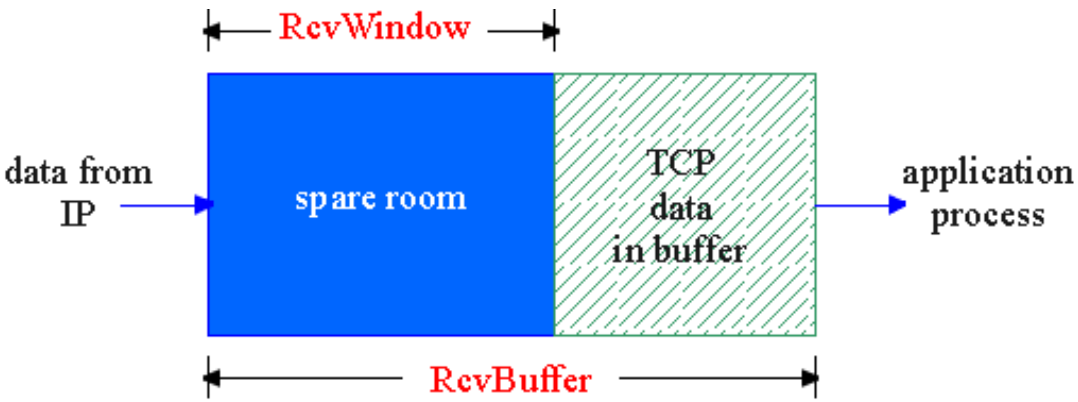
TCP Reliable Data Transfer

- **Connection-oriented:**
  - connection management
    - setup (exchange of control msgs) init's sender, receiver state before data exchange
    - close
- **Full duplex data:**
  - bi-directional data flow in same connection
- **A sliding window protocol**
  - a combination of go-back-n and selective repeat:
    - send & receive buffers
    - cumulative acks
    - TCP uses a single retransmission timer
    - do not retransmit all packets upon timeout



Flow Control

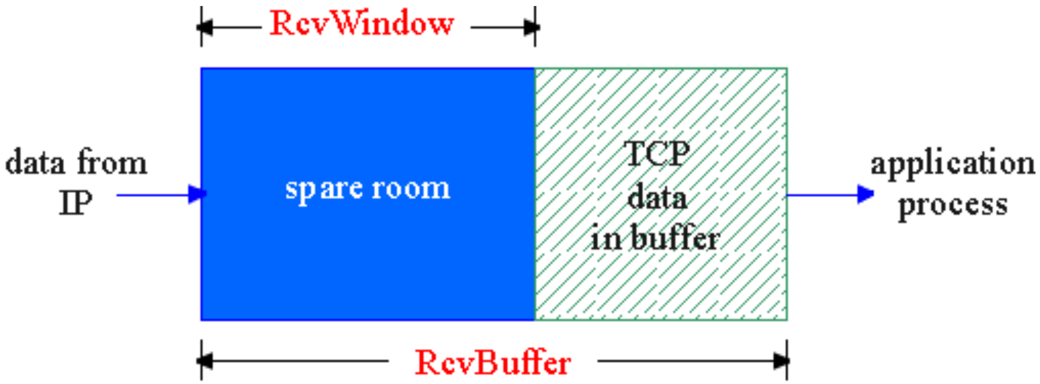
- receive side of a connection has a receive buffer:



- app process may be slow at reading from buffer

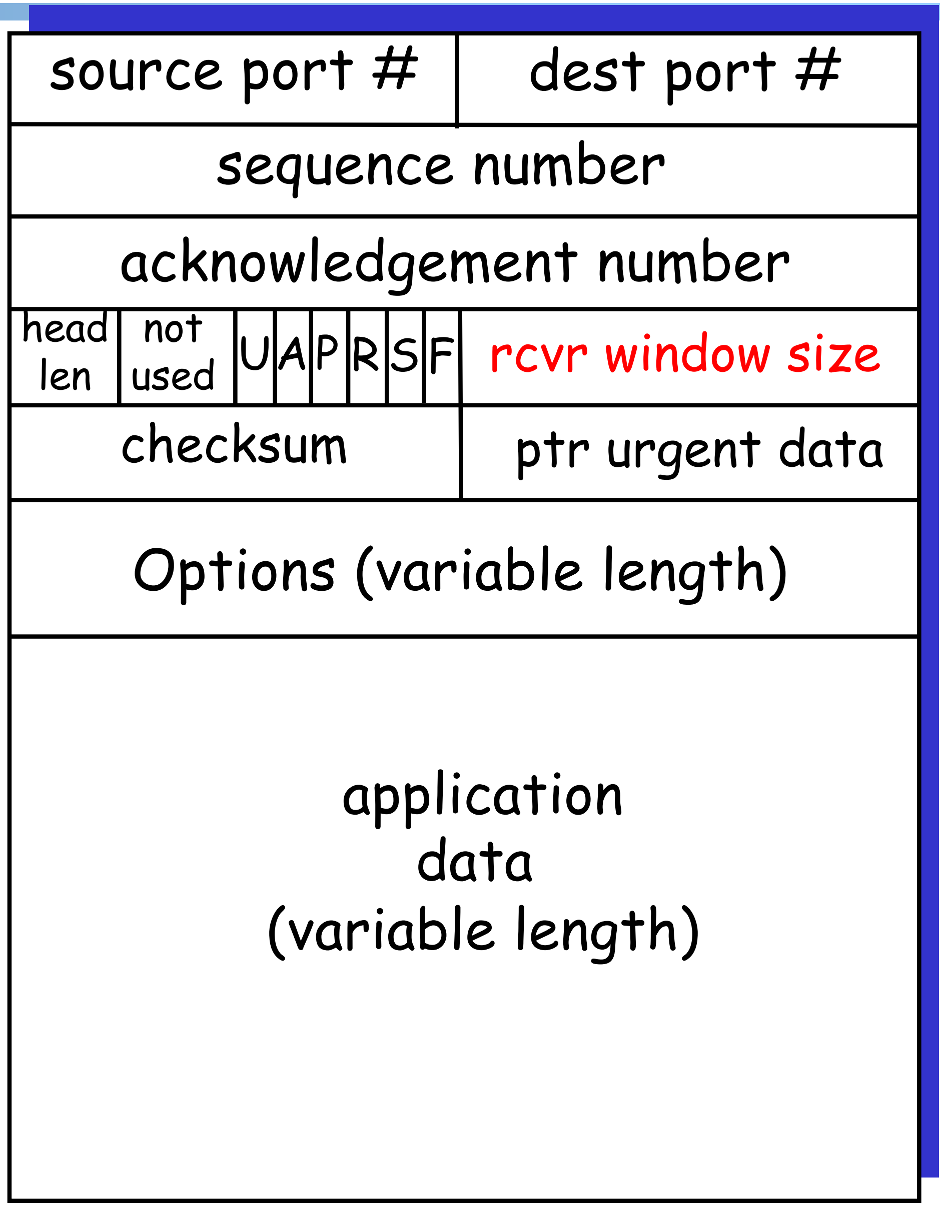
- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow Control: How it Works



- spare room in buffer

= **RcvWindow**



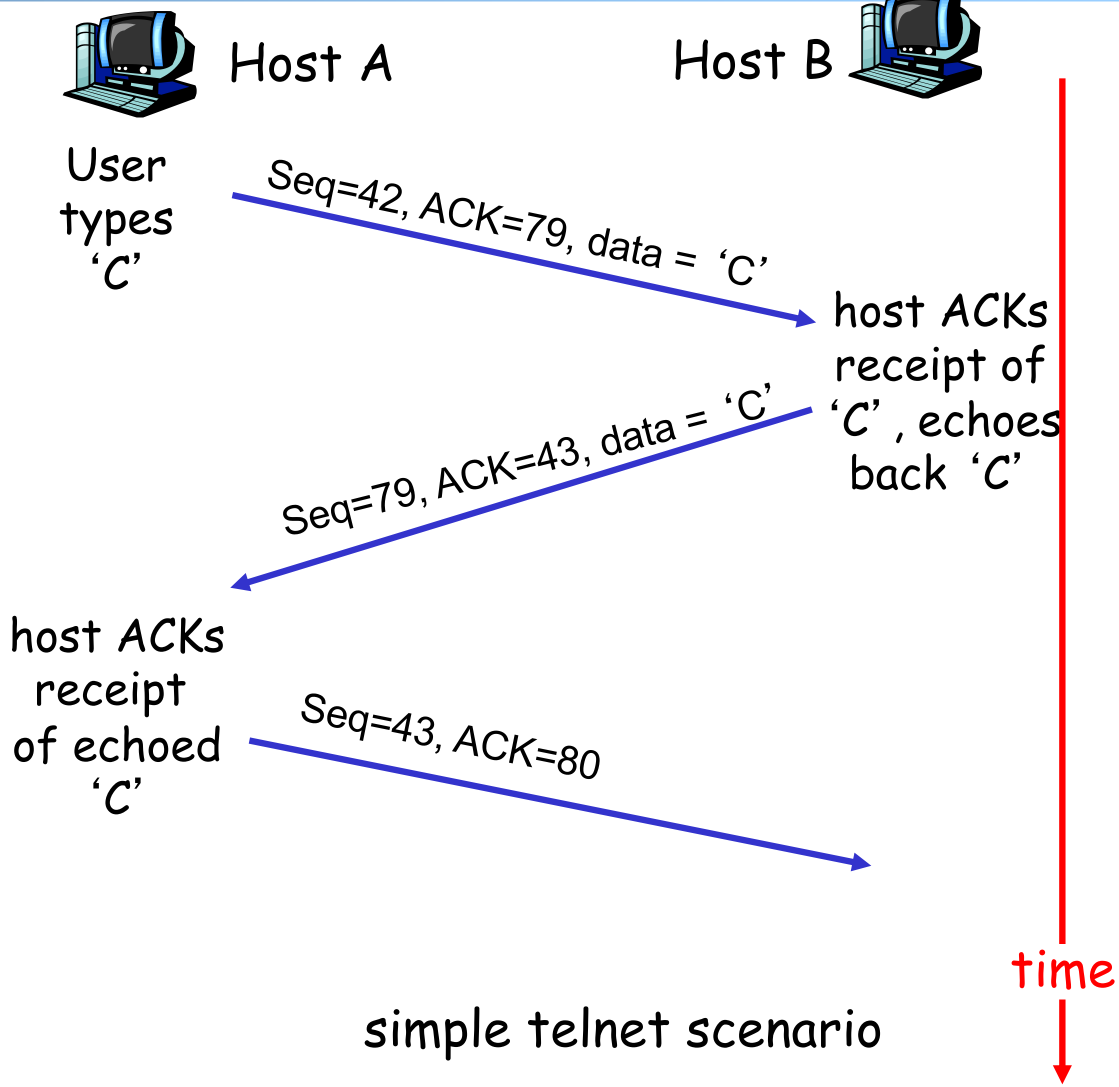
TCP Seq. #'s and ACKs

Seq. #'s:

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

ACKs:

- seq # of next byte expected from other side
- cumulative ACK in standard header
- selective ACK in options



simple telnet scenario

TCP Send/Ack Optimizations

- TCP includes many tune/optimizations, e.g.,
  - the "small-packet problem": sender sends a lot of small packets (e.g., telnet one char at a time)
    - Nagle's algorithm: do not send data if there is small amount of data in send buffer and there is an unack'd segment
  - the "ack inefficiency" problem: receiver sends too many ACKs, no chance of combing ACK with data
    - Delayed ack to reduce # of ACKs/combine ACK with reply

TCP Receiver 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

Outline

- Admin and Recap
- Reliable data transfer
  - perfect channel
  - channel with bit errors
  - channel with bit errors and losses
  - sliding window: reliability with throughput
- TCP reliability
  - data seq#, ack, buffering
  - *timeout realization*

TCP Reliable Data Transfer

- Basic structure: sliding window protocol
- Remaining issue: How to determine the "right" parameters?
  - timeout value?
  - sliding window size?

History

- Key parameters for TCP in mid-1980s
  - fixed window size W
  - timeout value = 2 RTT

- Network collapse in the mid-1980s
  - UCB ←→ LBL throughput dropped by 1000X !
- The intuition was that the collapse was caused by wrong parameters…

Timeout: Cost of Timeout Param

Why is good timeout value important?

- too short
  - premature timeout
  - unnecessary retransmissions; many duplicates

- too long
  - slow reaction to segment loss

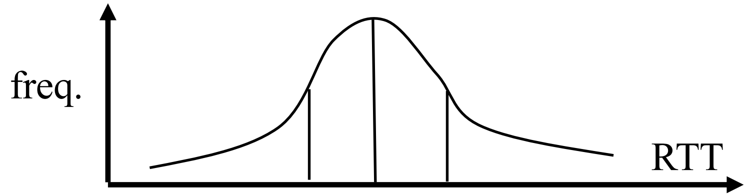
Q: Is it possible to set Timeout as a constant?

Q: Any problem w/ the early approach: Timeout = 2 RTT

Setting Timeout

Problem:

- Ideally, we set timeout = RTT,
  but RTT is not a fixed value
  =>
  using the average of **RTT** will generate
  many timeouts due to network variations
- Possibility: using the average/median of RTT
- Issue: this will generate many timeouts due to network variations



Solution:

- **Set Timeout RTO = avg +** "safety margin" based on variation

TCP approach:

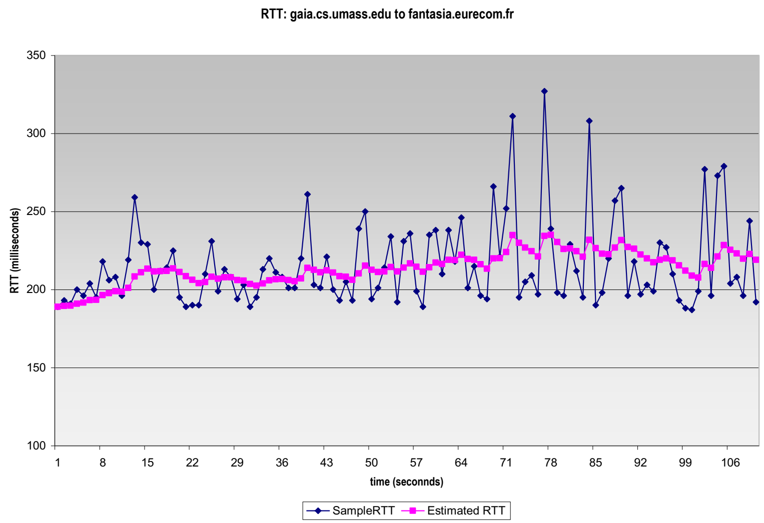
Timeout = EstRTT + 4 * DevRTT

Compute EstRTT and DevRTT

- Exponential weighted moving average (EWMA)
  - influence of past sample decreases exponentially fast

```
EstRTT = (1-alpha)*EstRTT + alpha*SampleRTT
```

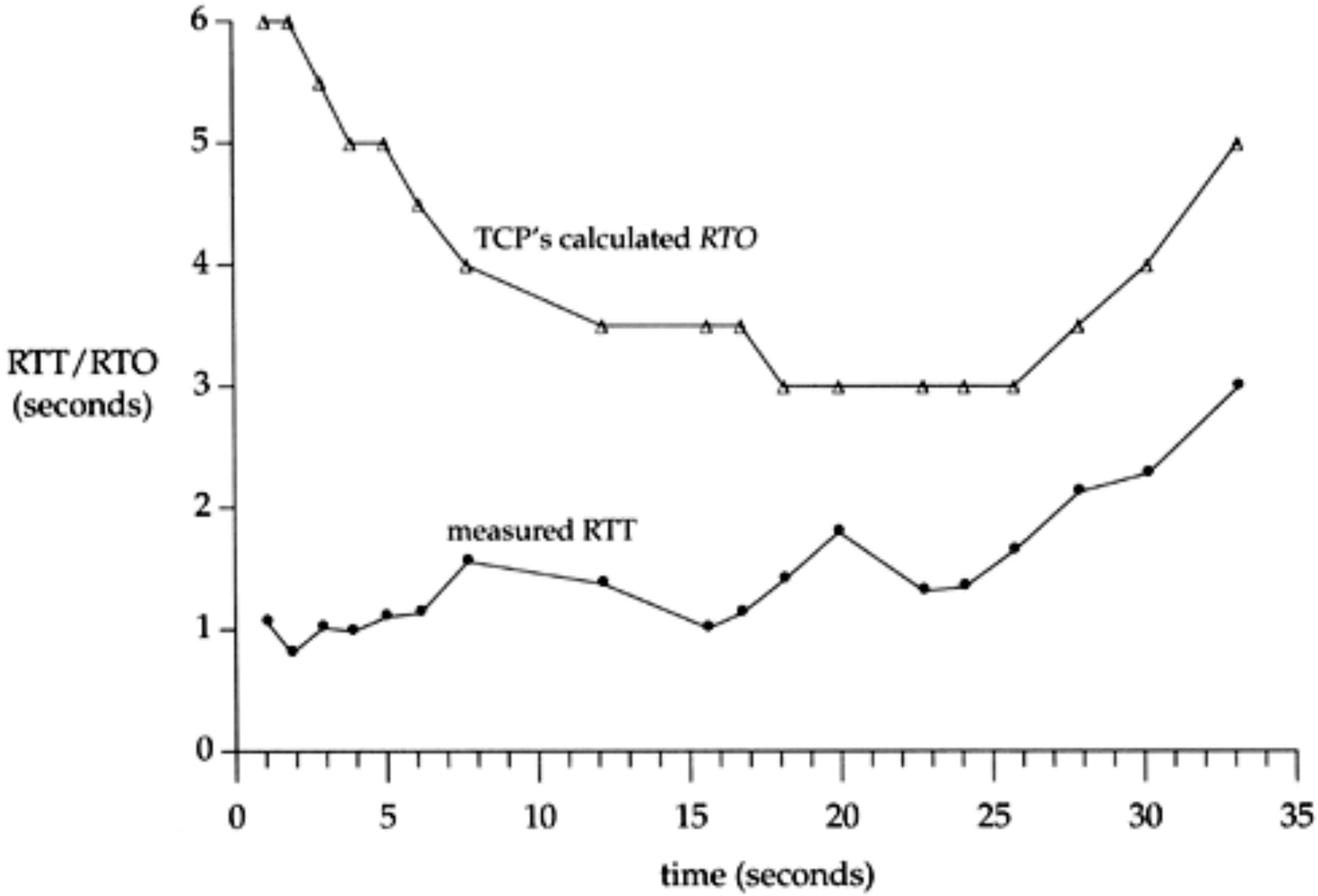
- **SampleRTT**: measured time from segment transmission until ACK receipt
- typical value: **alpha** = 0.125




```
DevRTT = (1-beta)*DevRTT + beta|SampleRTT-EstRTT|

(typically, beta = 0.25)
```

An Example TCP Session



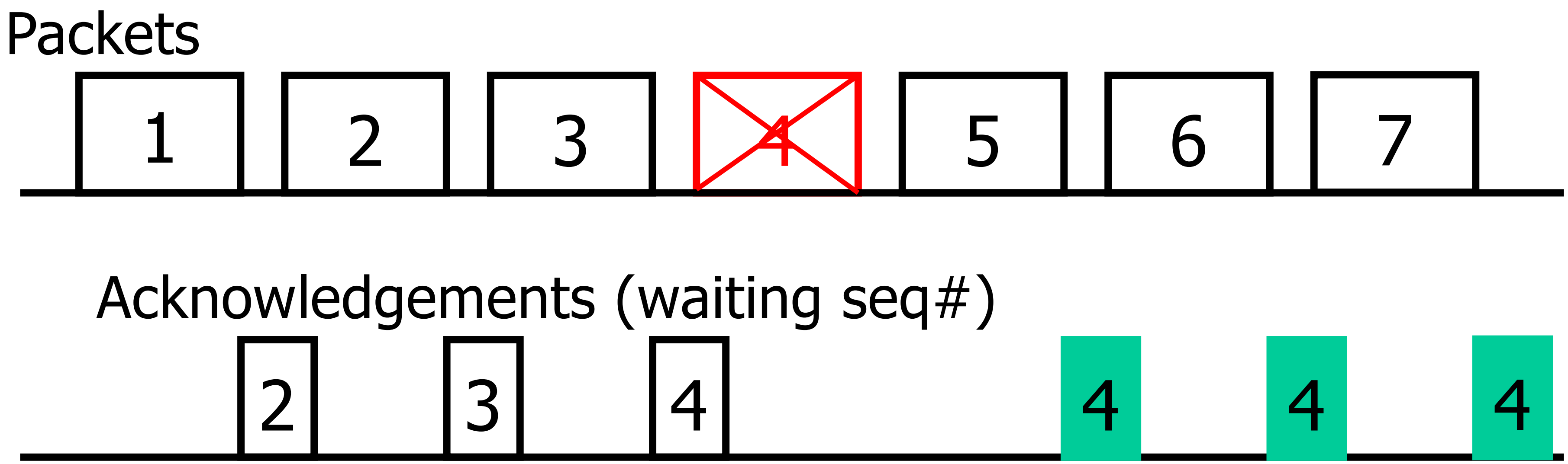
Fast Retransmit

- Issue: Timeout period often relatively long:
  - long delay before resending lost packet
- Question: Can we detect loss faster than RTT?

- 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:
  - resend segment before timer expires

Triple Duplicate Ack



Fast Retransmit:

```
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

TCP: reliable data transfer

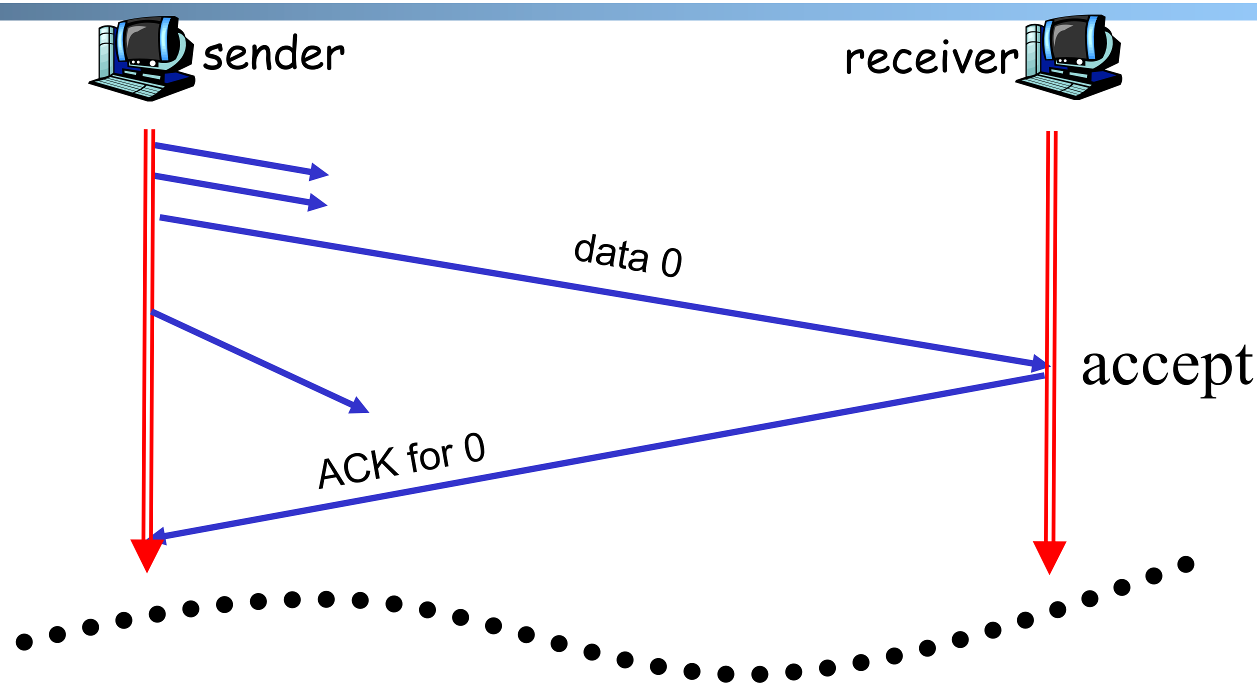
Simplified TCP sender

```
sendbase = initial_sequence number agreed by TWH
nextseqnum = initial_sequence number by TWH
loop (forever) {
  switch(event)
  event: data received from application above
        if (window allows send)
          create TCP segment with sequence number nextseqnum
          if (no timer) start timer
          pass segment to IP
          nextseqnum = nextseqnum + length(data)
        else put packet in buffer
   event: timer timeout for sendbase
      retransmit segment
      compute new timeout interval
      restart timer
   event: ACK received, with ACK field value of y
      if (y > sendbase) { /* cumulative ACK of all data up to y */
         cancel the timer for sendbase
         sendbase = y
         if (no timer and packet pending) start timer for new sendbase
         while (there are segments and window allow)
            sent a segment;
      }
      else { /* y==sendbase, duplicate ACK for already ACKed segment */
         increment number of duplicate ACKs received for y
         if (number of duplicate ACKS received for y == 3) {
            /* TCP fast retransmit */
           resend segment with sequence number y
           restart timer for segment y
         }
  }  /* end of loop forever */
```

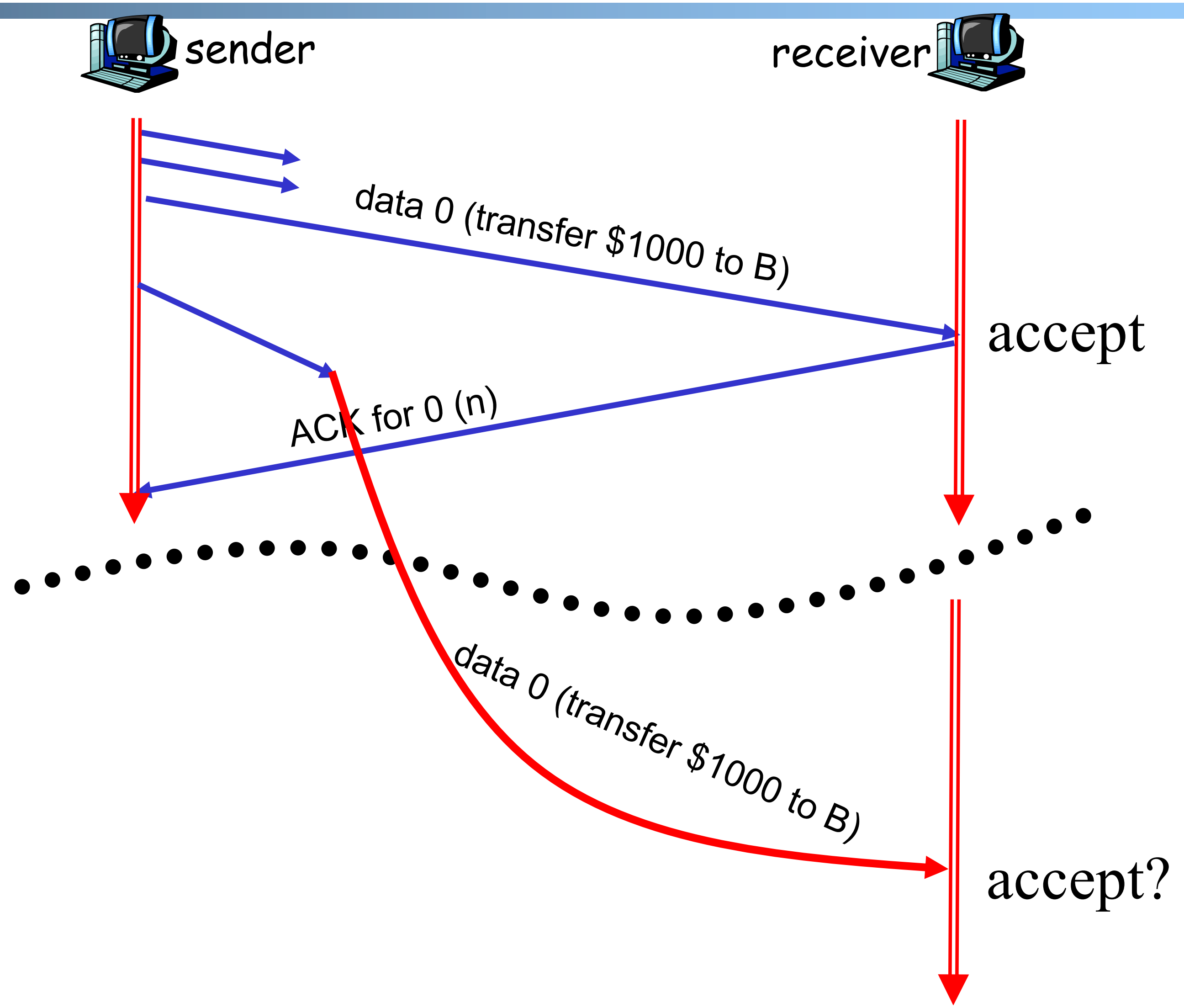
Outline

- Admin and Recap
- Reliable data transfer
  - perfect channel
  - channel with bit errors
  - channel with bit errors and losses
  - sliding window: reliability with throughput
- TCP reliability
  - data seq#, ack, buffering
  - timeout realization
  - *connection management*

Why Connection Setup/When to Accept (Safely Deliver) First Packet?



Why Connection Setup/When to Accept (Safely Deliver) First Packet?



24

Transport "Safe-Setup" Principle

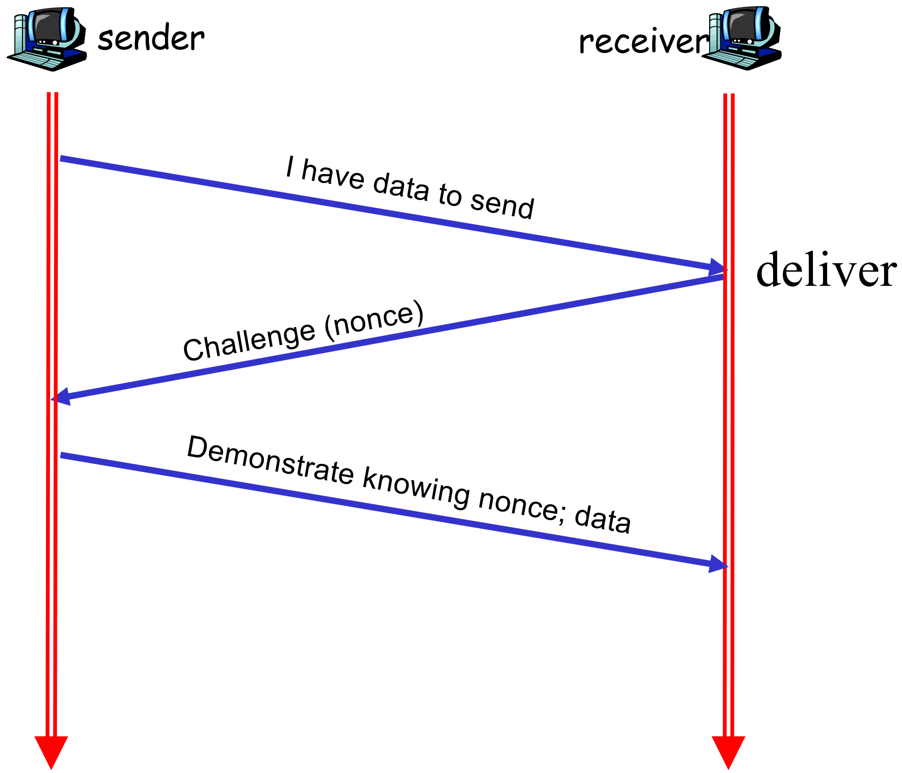
- A general safety principle for a receiver R to accept a message from a sender S is the general "authentication" principle, which consists of two conditions:

Transport authentication principle:
- [p1] Receiver can be sure that what Sender says is **fresh**
- [p2] Receiver receives something that ***only*** Sender can say

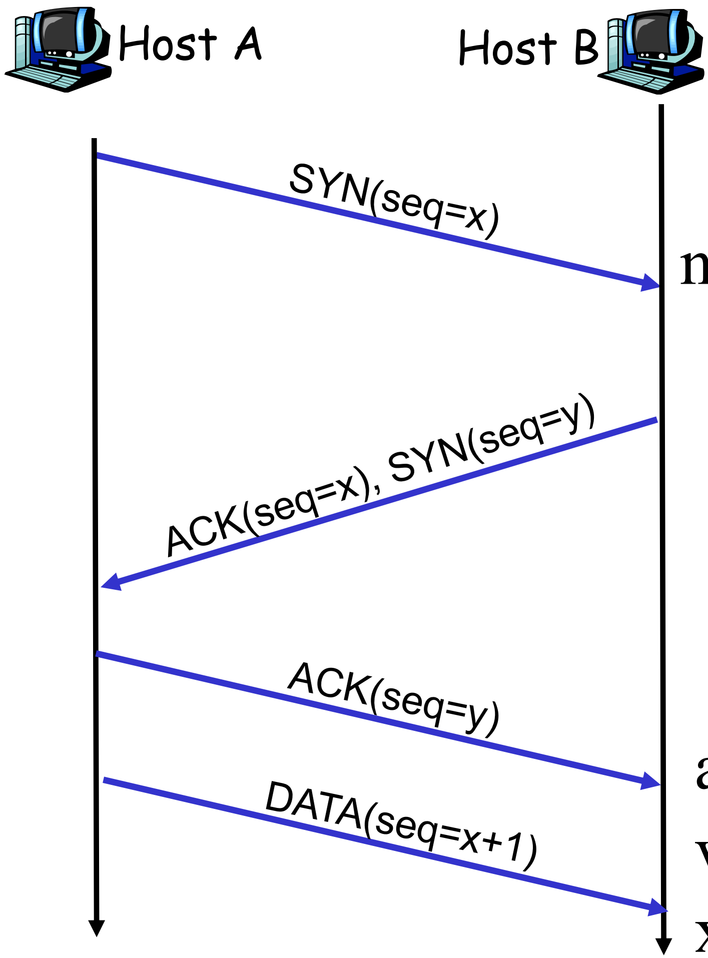
We first assume a secure setting: no malicious attacks.

Exercise: Techniques to allow a receiver to check for freshness (e.g., add a time stamp)?

Generic Challenge-Response Structure Checking Freshness



Three Way Handshake (TWH) [Tomlinson 1975]



notify initial seq#. Accept?

think of y as a challenge

accept data only after verified y is bounced back x is the init. seq

SYN: indicates connection setup

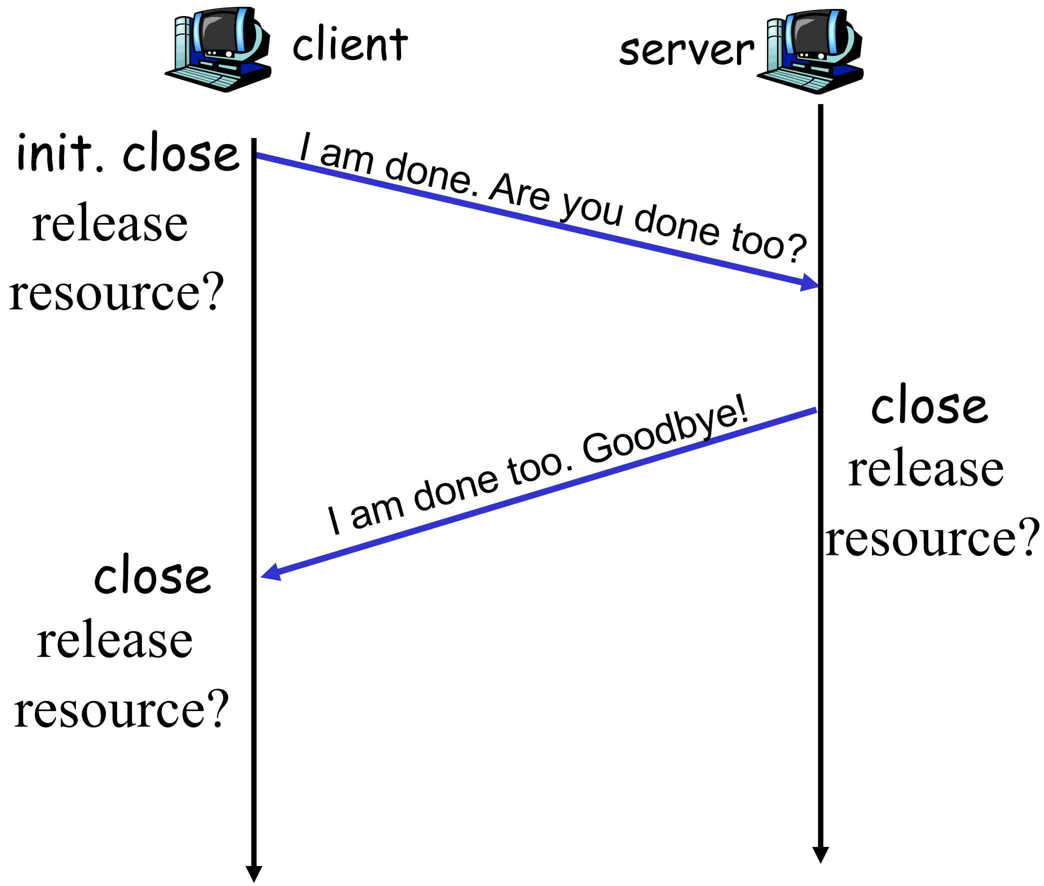
Make "Challenge y" Robust

- To avoid that "SYNC ACK y" comes from reordering and duplication
  - for each connection (sender-receiver pair), ensuring that two identically numbered packets are never outstanding at the same time
    - network bounds the life time of each packet
    - a sender will not reuse a seq# before it is sure that all packets with the seq# are purged from the network
    - seq. number space should be large enough to not limit transmission rate

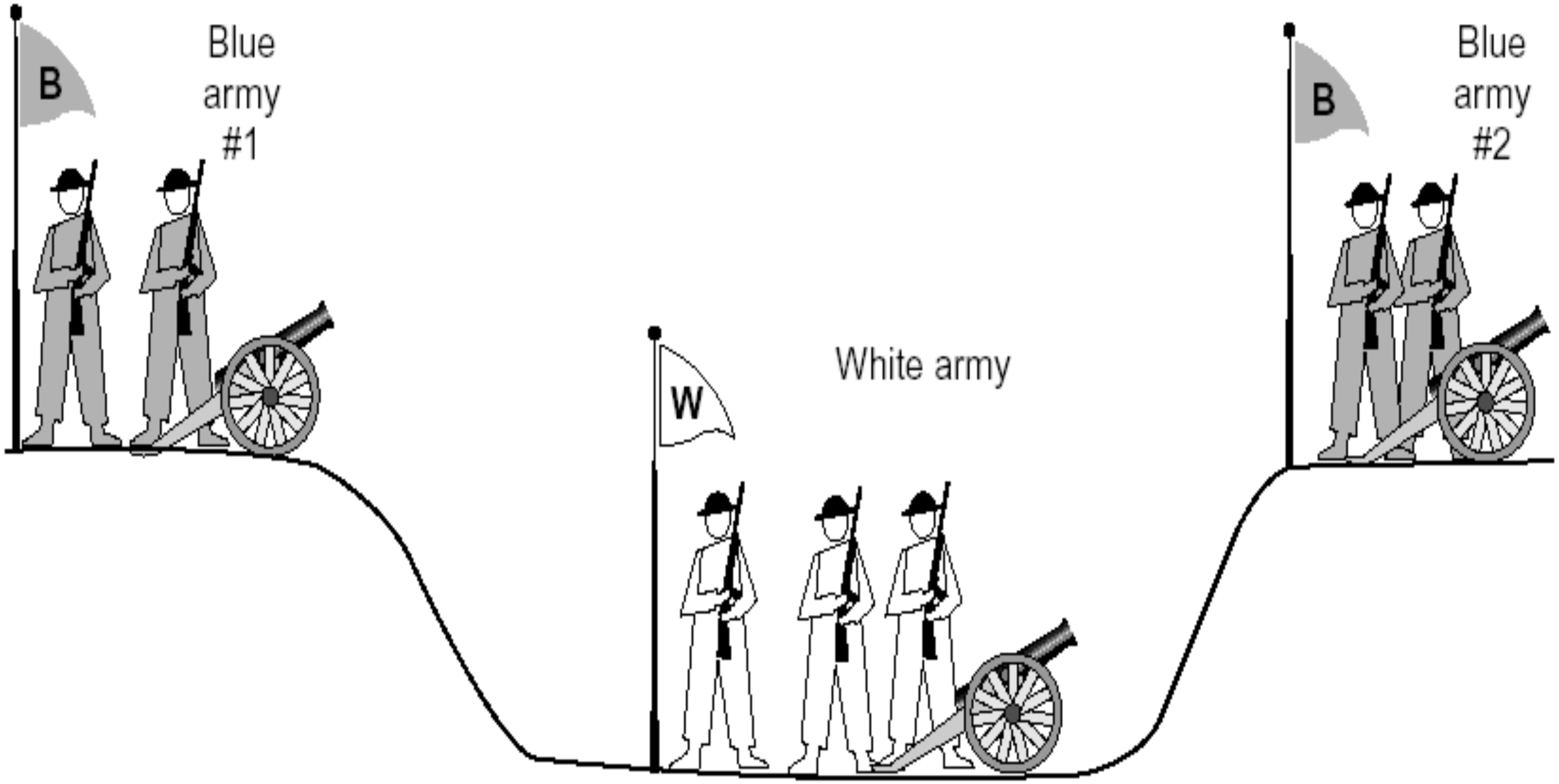
- Increasingly move to cryptographic challenge and response

Connection Close

- Why connection close?
  - so that each side can release resource and remove state about the connection (do not want dangling socket)



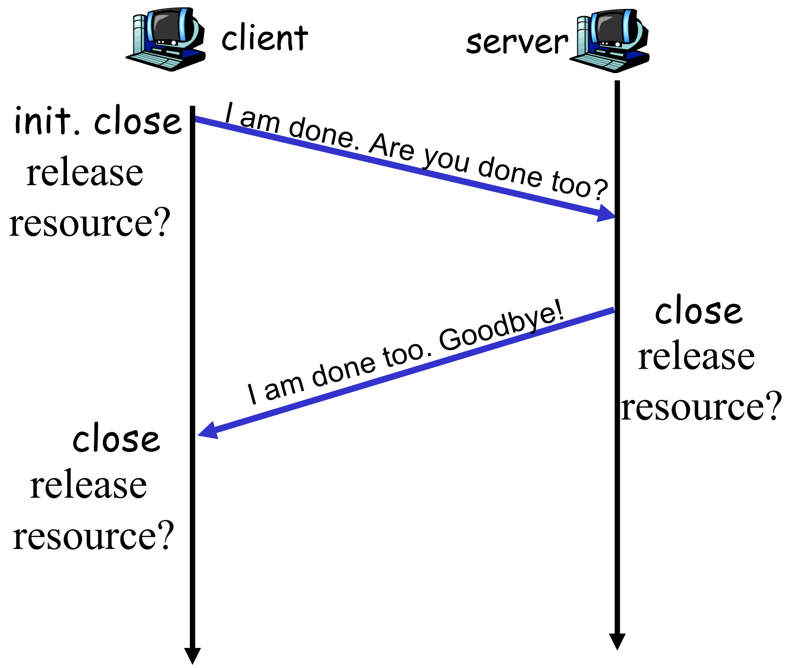
General Case: The Two-Army Problem

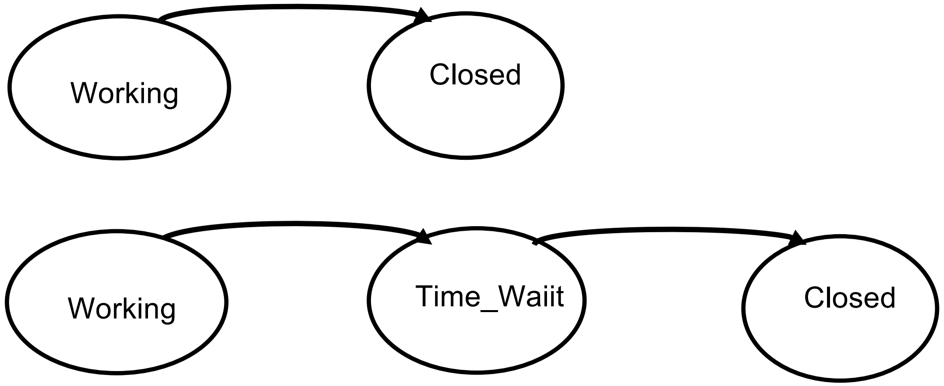


The gray (blue) armies need to agree on whether or not they will attack the white army. They achieve agreement by sending messengers to the other side. If they both agree, attack; otherwise, no. Note that a messenger can be captured!

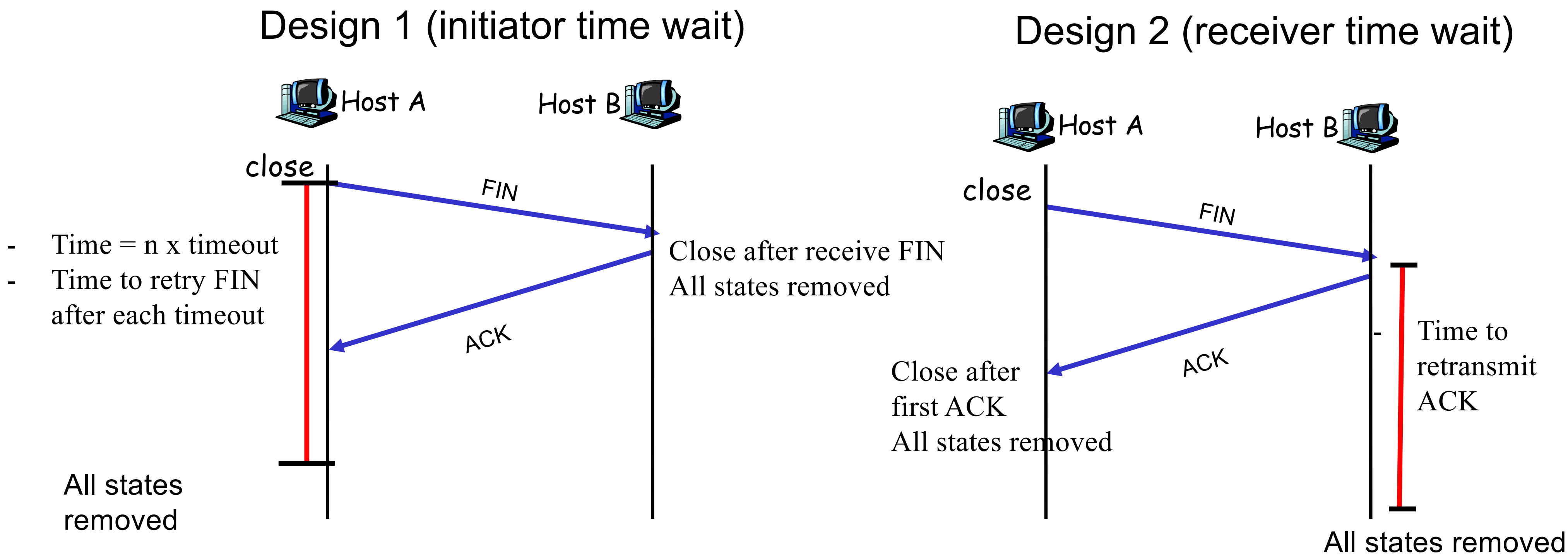
Time_Wait

- Generic technique: Timeout to "solve" infeasible problem
  - Instead of message-driven state transition, use a timeout based transition; use timeout to handle error cases

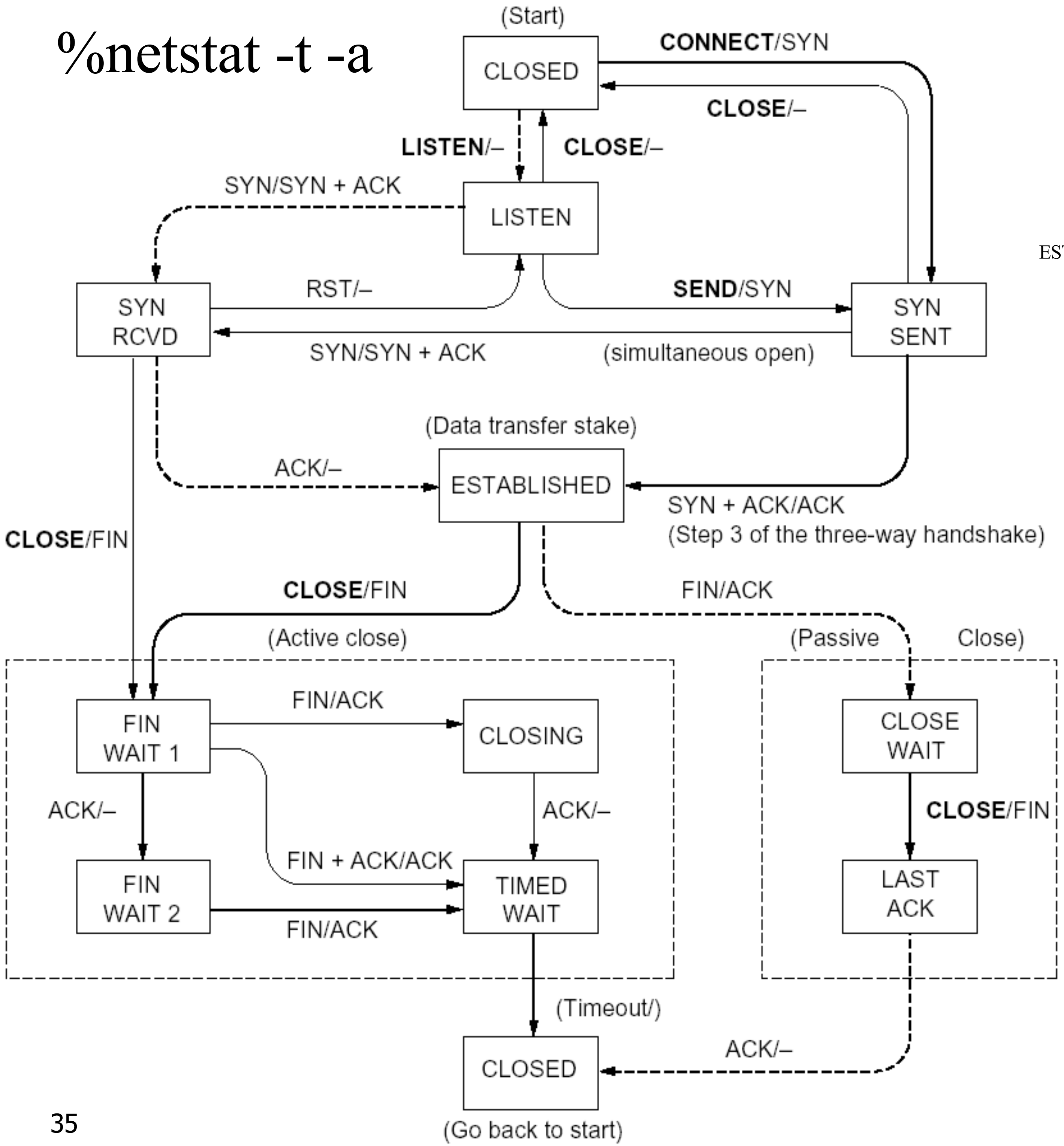




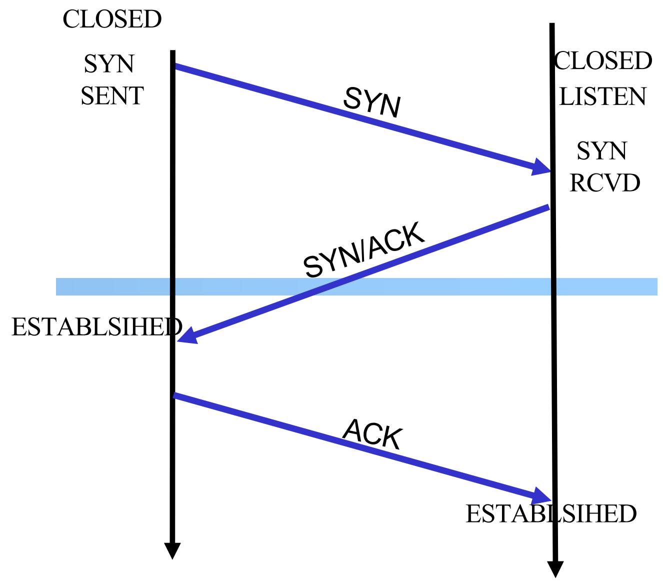
Time_Wait Design Options



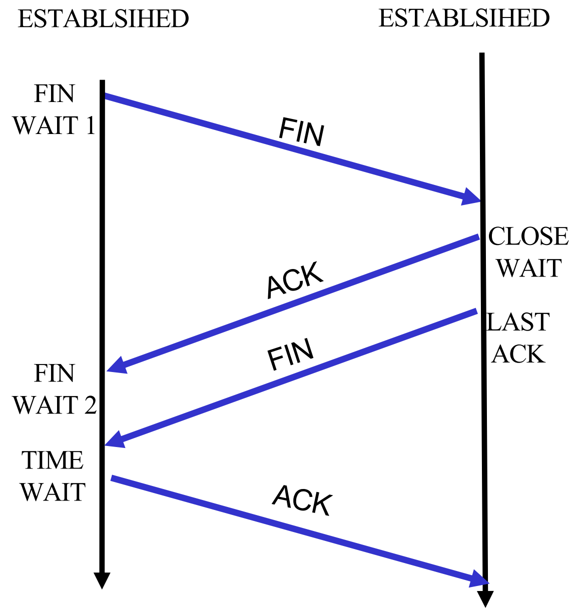
%netstat -t -a



(Start)
CLOSED
CONNECT/SYN
CLOSE/–
LISTEN/–
CLOSE/–
SYN/SYN + ACK
LISTEN
RST/–
SEND/SYN
SYN RCVD
SYN SENT
SYN/SYN + ACK
(simultaneous open)
(Data transfer stake)
ACK/–
ESTABLISHED
SYN + ACK/ACK
(Step 3 of the three-way handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
(Active close)
(Passive
Close)
FIN WAIT 1
FIN/ACK
CLOSING
CLOSE WAIT
ACK/–
ACK/–
CLOSE/FIN
FIN + ACK/ACK
FIN WAIT 2
TIMED WAIT
LAST ACK
FIN/ACK
(Timeout/)
ACK/–
CLOSED
(Go back to start)



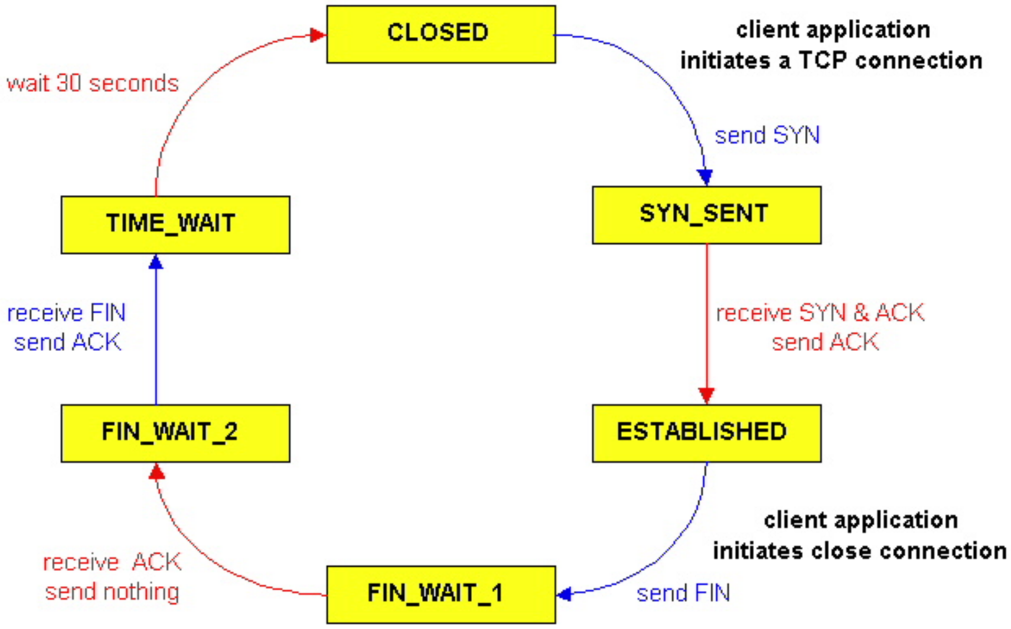
CLOSED
SYN SENT
CLOSED
LISTEN
SYN
SYN RCVD
SYN/ACK
ESTABLSIHED
ACK
ESTABLSIHED

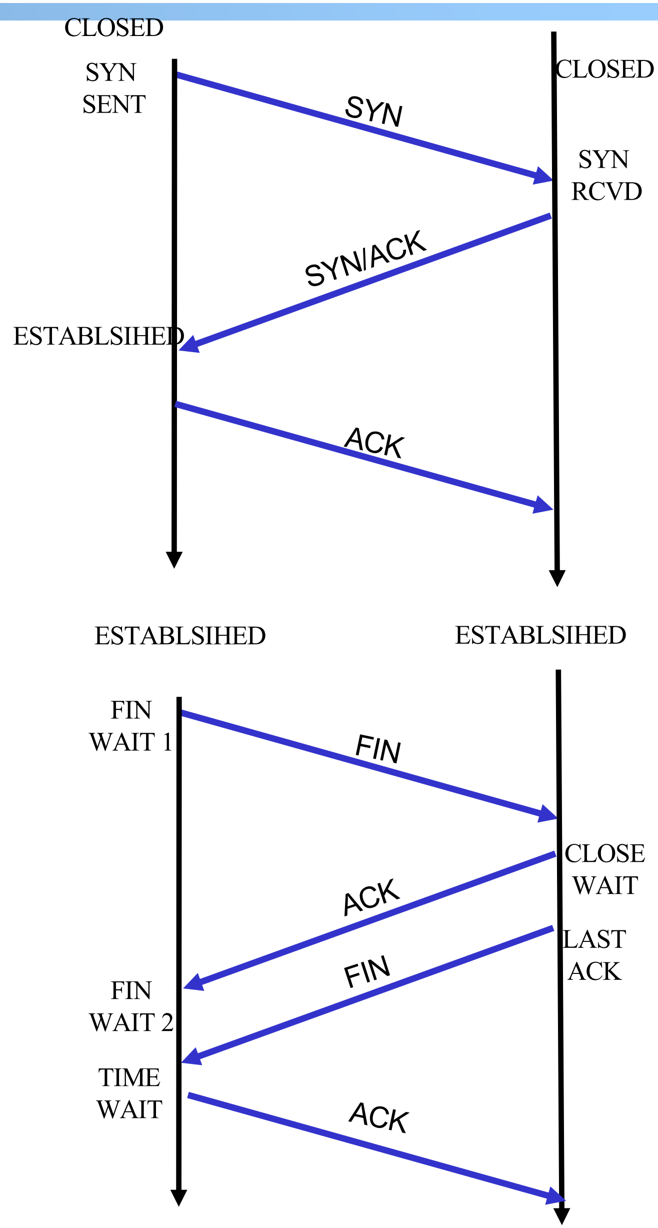


ESTABLSIHED
ESTABLSIHED
FIN WAIT 1
FIN
CLOSE WAIT
ACK
LAST ACK
FIN
FIN WAIT 2
TIME WAIT
ACK

TCP Connection Management

TCP lifecycle: init SYN/FIN

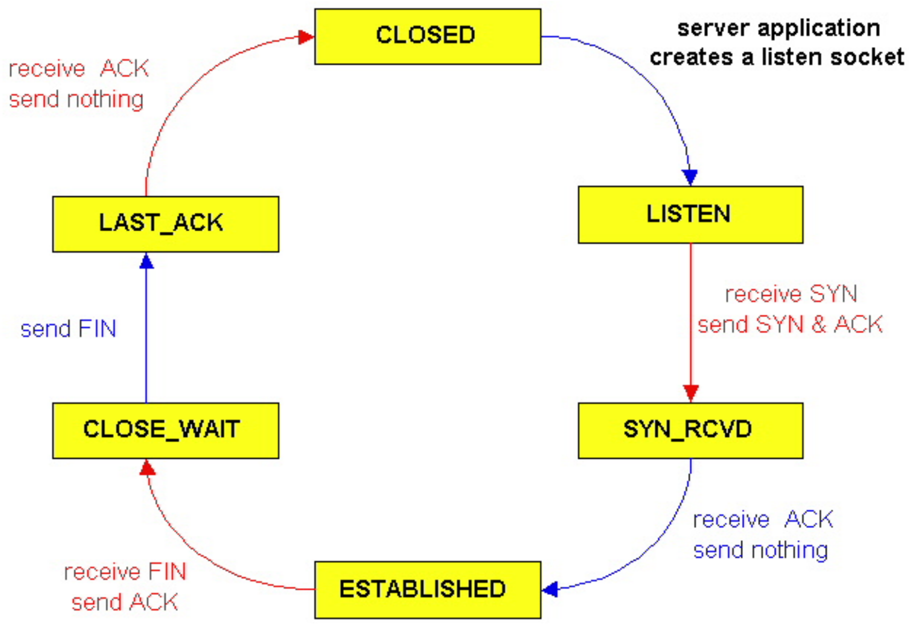


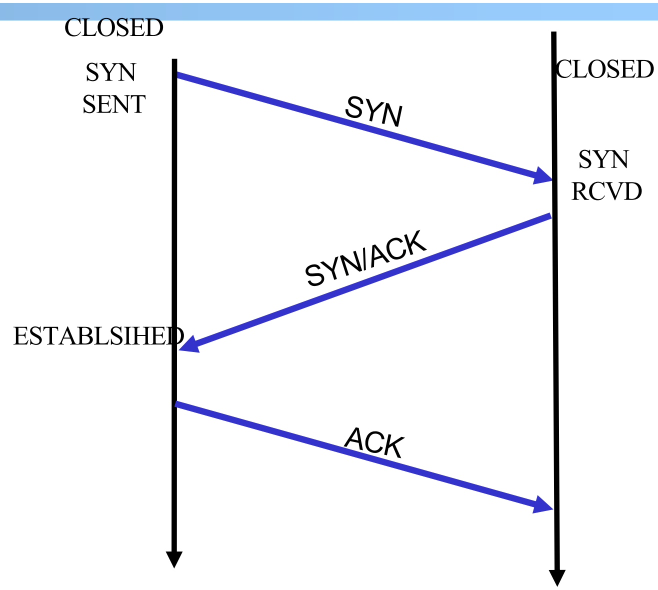


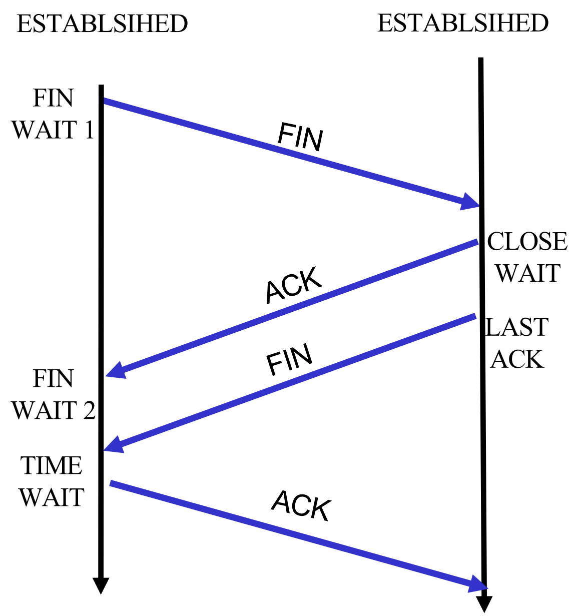
http://dsd.lbl.gov/TCP-tuning/ip-sysctl-2.6.txt

TCP Connection Management

TCP lifecycle: wait for SYN/FIN





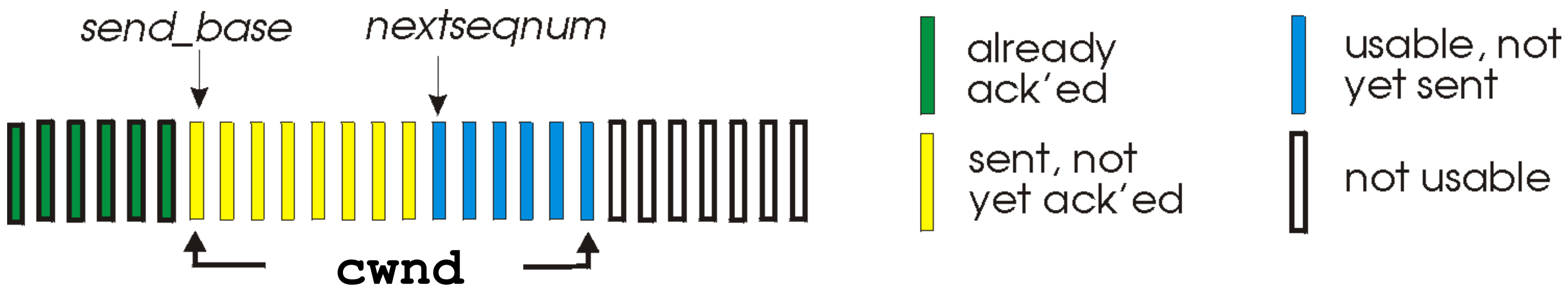


A Summary of Questions

- Basic structure: sliding window protocols
- How to determine the "right" parameters?
  - ✓ timeout: mean + variation
  - sliding window size?

Sliding Window Size Function: Rate Control

- Transmission rate determined by congestion window size, **cwnd**, over segments:



- cwnd segments, each with MSS bytes sent in one RTT:

$$\boxed{\text{Rate} = \frac{\text{cwnd} * \text{MSS}}{\text{RTT}} \text{ Bytes/sec}}$$

Assume W is small enough. Ignore small details. MSS: Minimum Segment Size

Some General Questions

Big picture question:

- How to determine a flow's sending rate?

For better understanding, we need to look at a few basic questions:

- What is congestion (cost of congestion)?
- Why are desired properties of congestion control?

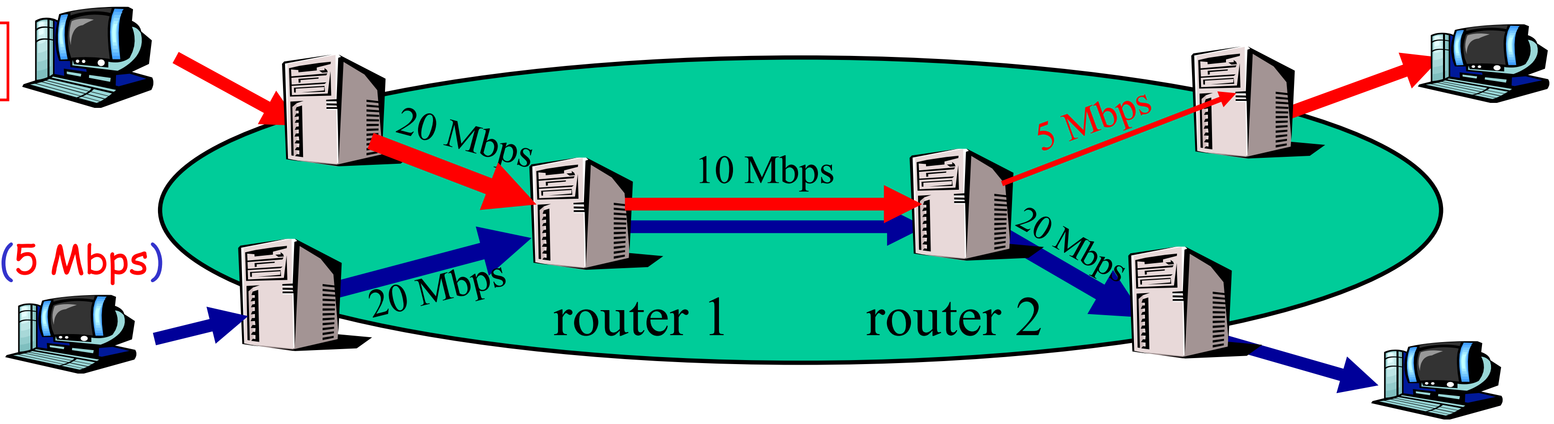
Roadmap

- What is congestion
- The basic CC alg
- TCP/reno CC
- TCP/Vegas
- A unifying view of TCP/Reno and TCP/Vegas
- Network wide resource allocation
  - Framework
  - Axiom derivation of network-wide objective function
  - Derive distributed algorithm

Outline

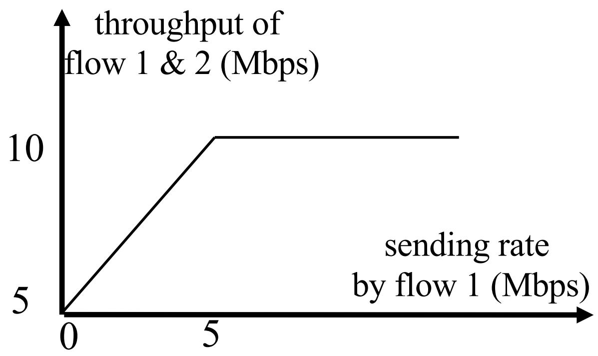
- Admin and recap
- TCP Reliability
- Transport congestion control
  - *what is congestion (cost of congestion)*

Cause/Cost of Congestion: Single Bottleneck

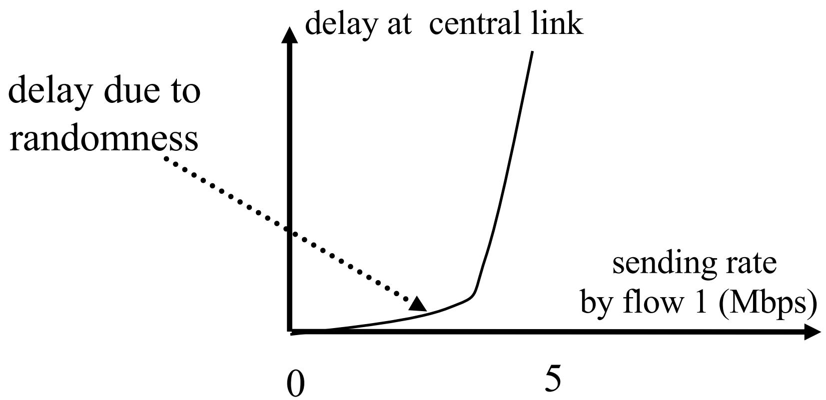


- Flow 2 has a fixed sending rate of 5 Mbps
- We vary the sending rate of flow 1 from 0 to 20 Mbps
- Assume
  - no retransmission; link from router 1 to router 2 has infinite buffer

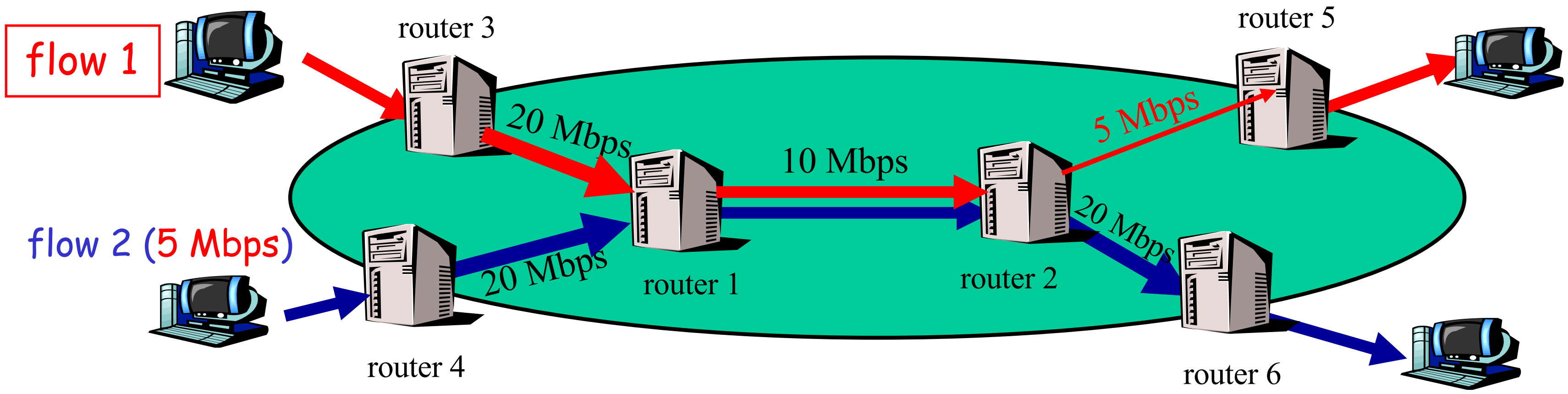
throughput: e2e packets delivered in unit time



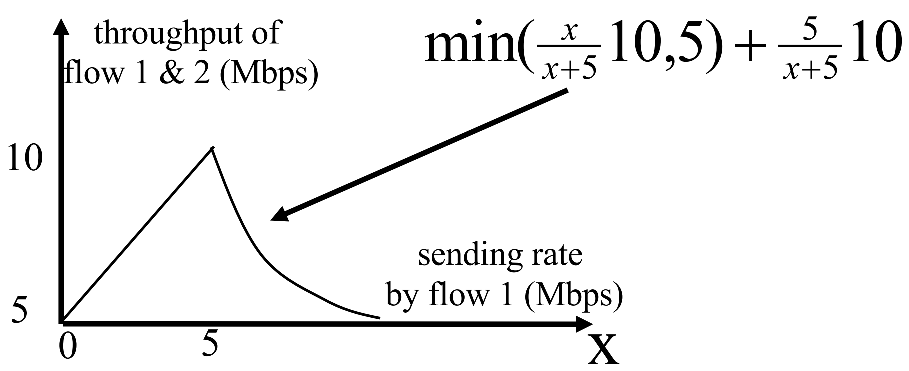
Delay?



Cause/Cost of Congestion: Single Bottleneck



- Assume
  - no retransmission
  - the link from router 1 to router 2 has finite buffer
  - throughput: e2e packets delivered in unit time

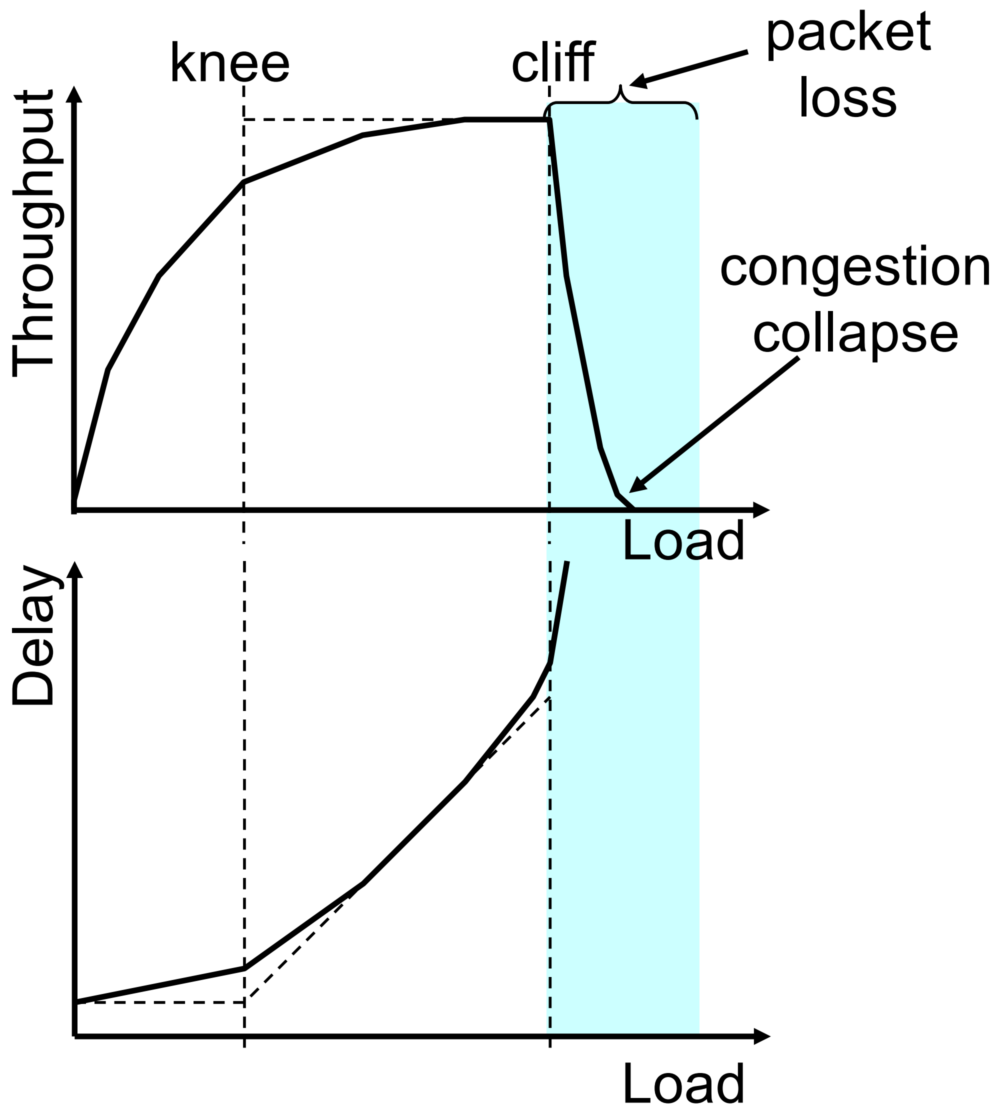


- Zombie packet: a packet dropped at the link from router 2 to router 5; the upstream transmission from router 1 to router 2 used for that packet was wasted!

Summary: The Cost of Congestion

When sources sending rate too high for the *network* to handle":

- Packet loss =>
  - wasted upstream bandwidth when a pkt is discarded at downstream
  - wasted bandwidth due to retransmission (a pkt goes through a link multiple times)



- High delay