Network Transport Layer: TCP Reliability; Transport Congestion Control; TCP/Reno CC

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs433/

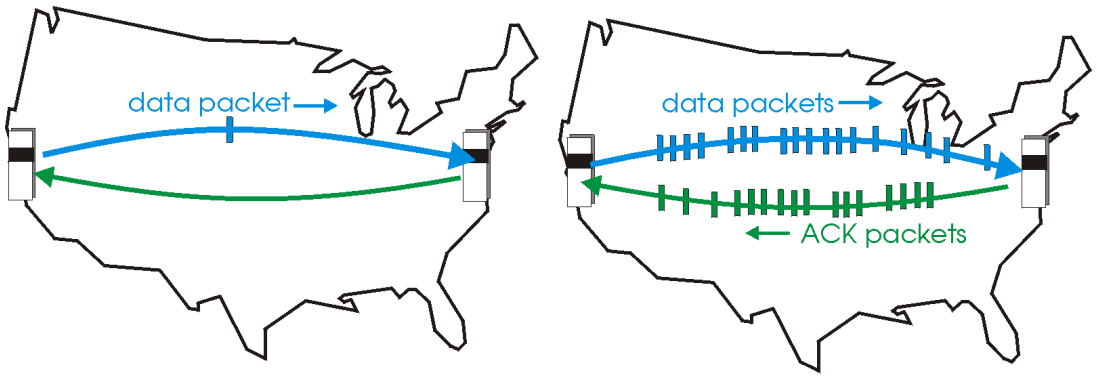
11/8/2018

Admin.

- Programming assignment 4 Part 1 discussion instructor slots:
  - Nov. 9: 3:30-5:00 pm
  - Nov. 11: 3:00-4:30 pm
  - Nov. 12: 2:00-3:00 pm
  - Nov. 13: 2:30-3:30 pm (regular office hour)

Recap: Reliable Transport Basic Structure

- Basic structure: sliding window protocols



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

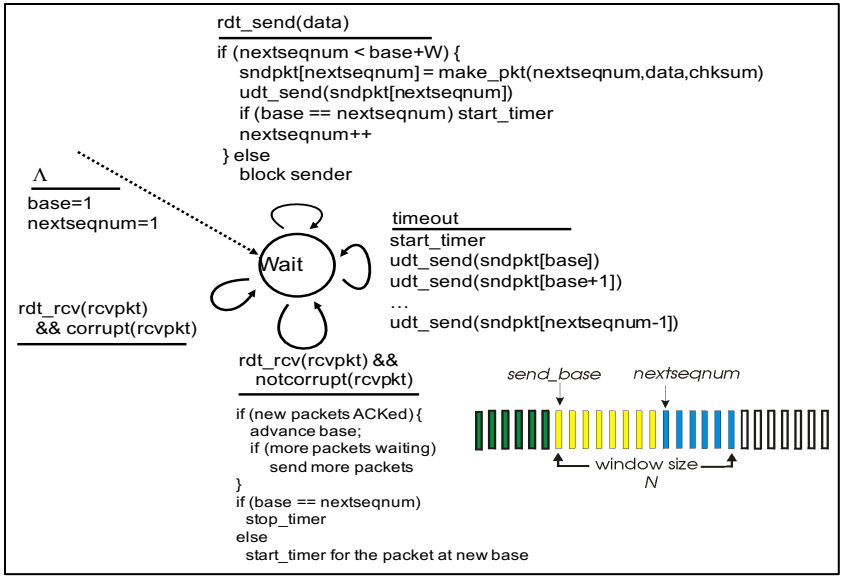
(b) a pipelined protocol in operation

An instance of generic computer science technique: pipelining.

- Benefit: when window size is bandwidth-delay product, it achieves non-stop transmission until ack comes back, fully utilizing link

Recap: Sliding Window Realizations

- Two major types: Go-Back-n (GBN) and Selective-Repeat (SR)



GBN: sender

sender

data from above :

- unACKed packets is less than window size W, send; otherwise block app.

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+W-1]:

- mark pkt n as received
- update sendbase to the first packet unACKed

receiver

pkt n in [rcvbase, rcvbase+W-1]

- send ACK(n)
- if (out-of-order)
  mark and buffer pkt n
  else /*in-order*/
  deliver any in-order packets

otherwise:

- ignore

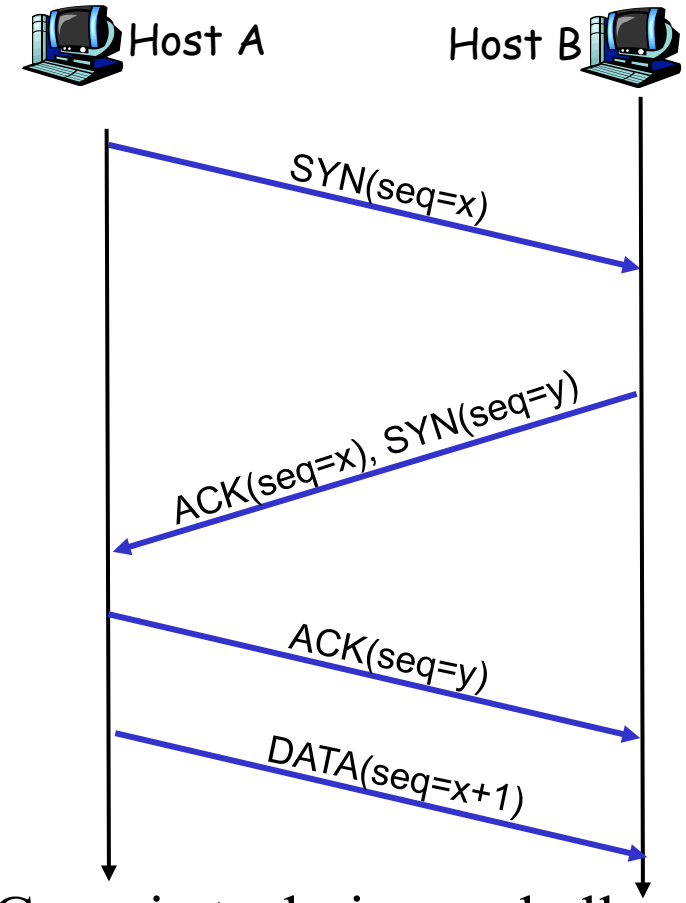
SR: sender/receiver

Recap: Sliding Window Realizations

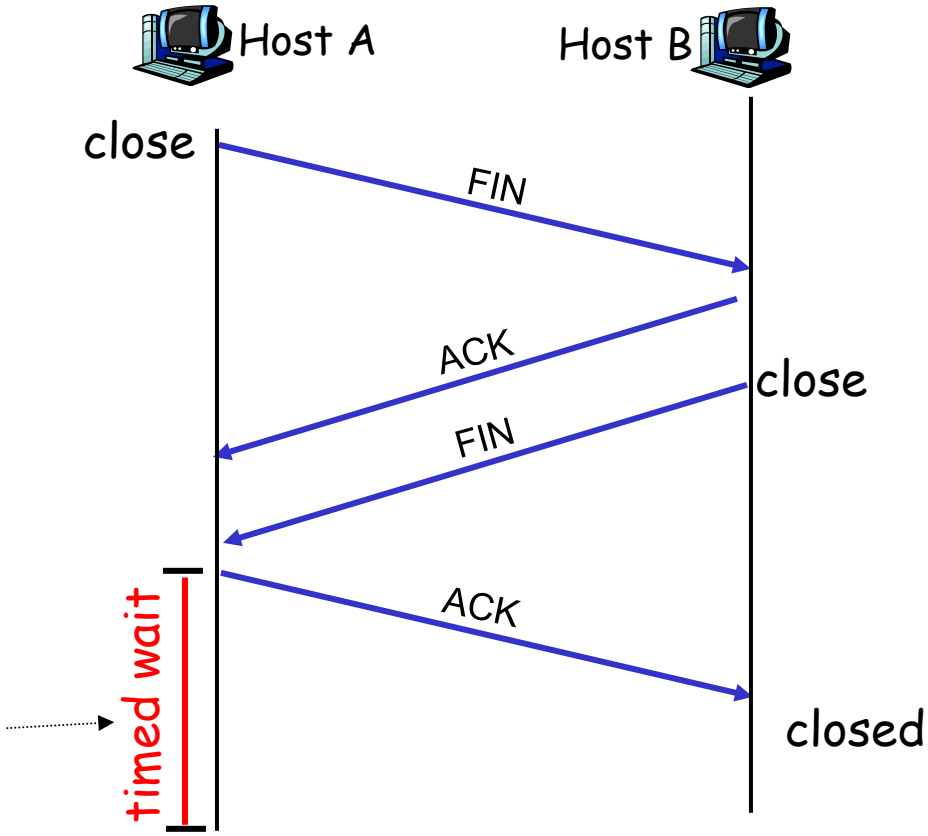
- GBN and SR comparison
  - Go-Back-n (GBN): sequential receiving at receiver
  - Selective-Repeat (SR): W parallel receiving at receiver

	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

Recap: Transport Connection Management



Generic technique: challenge to verify freshness.



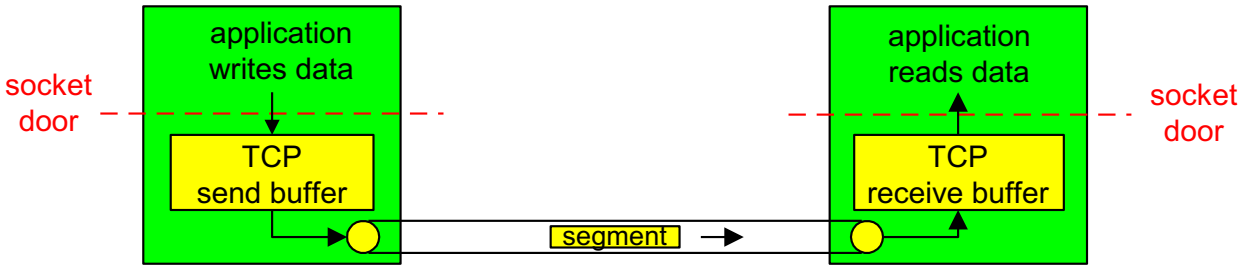
Generic technique: Timeout to realize ”solve” infeasible problem.

Outline

- Admin and recap
- Transport reliability
  - sliding window protocols
  - connection management
  - TCP reliability

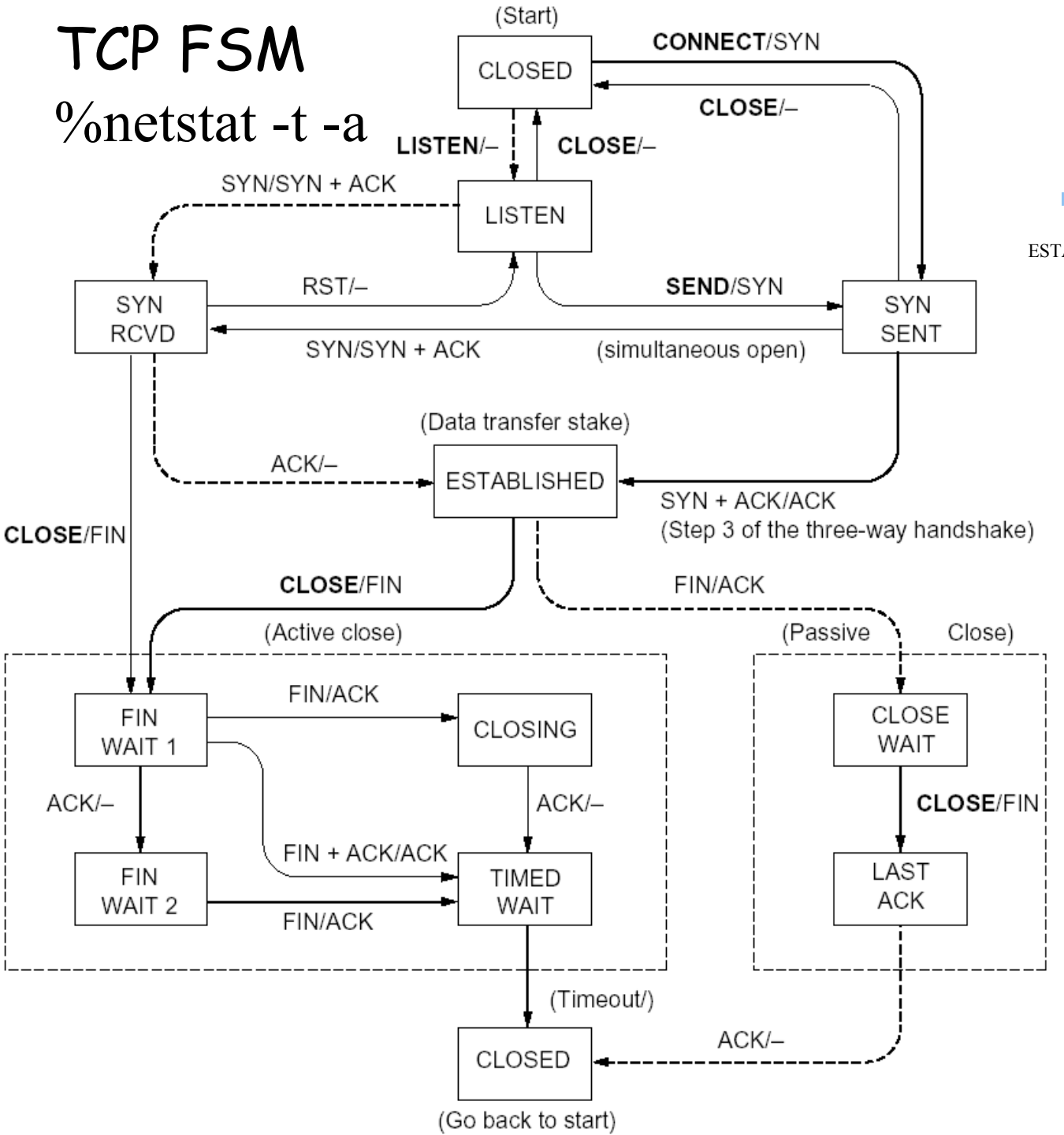
TCP Reliable Data Transfer

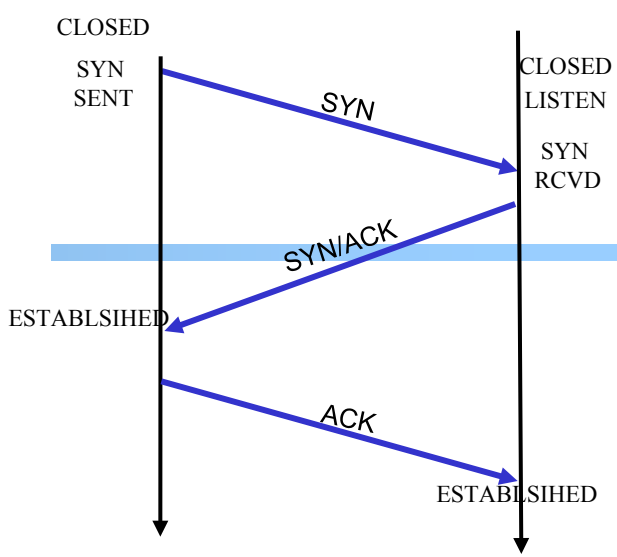
- A sliding window protocol
  - a combination of go-back-n and selective repeat:
    - send & receive buffers (flow control helps manage buffer size)
    - cumulative acks
    - uses a single retransmission timer
    - do not retransmit all packets upon timeout
- Connection managed
  - setup (exchange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

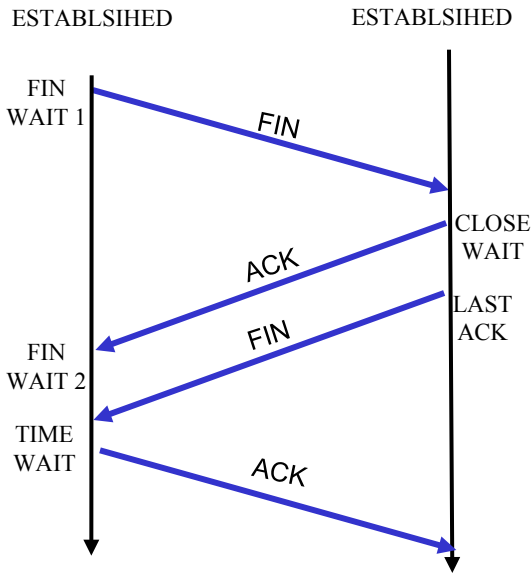


TCP FSM

%netstat -t -a

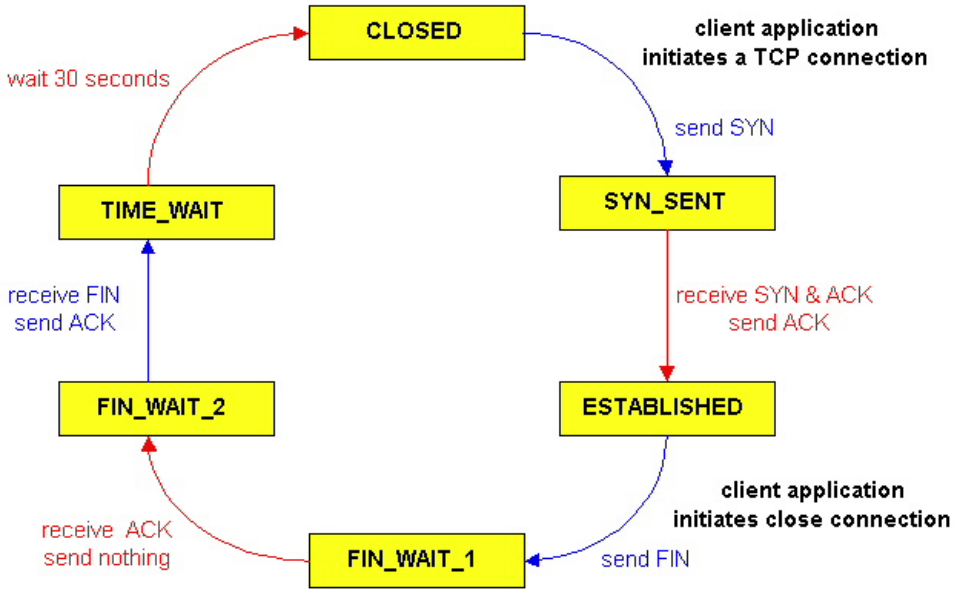


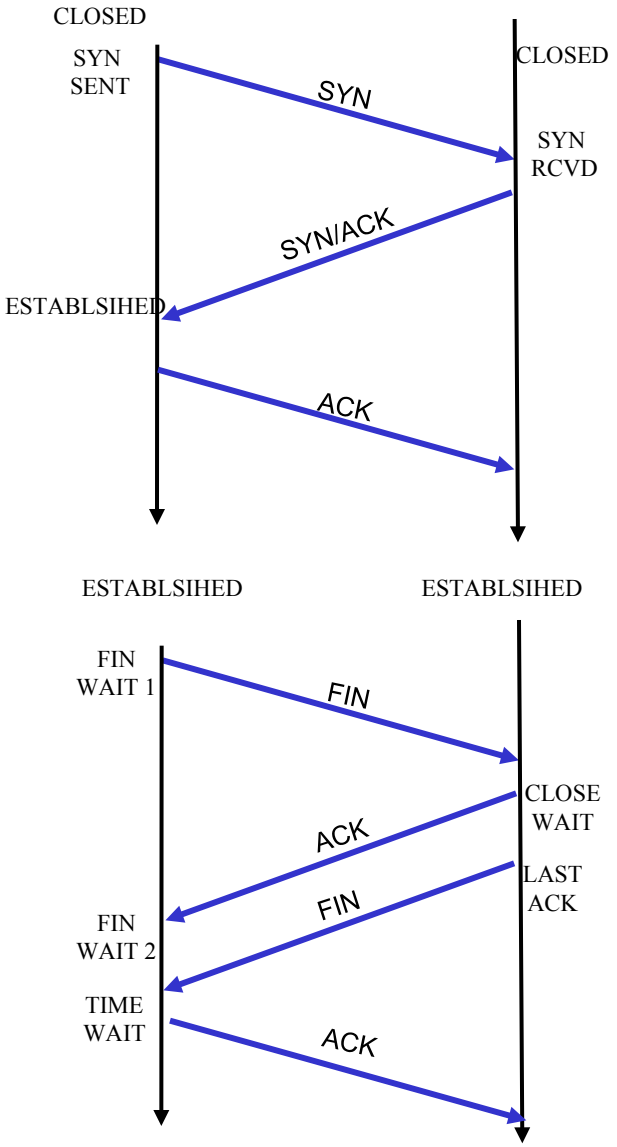




Example Transition (Client Connect + Client Close): Client

TCP lifecycle: init SYN/FIN

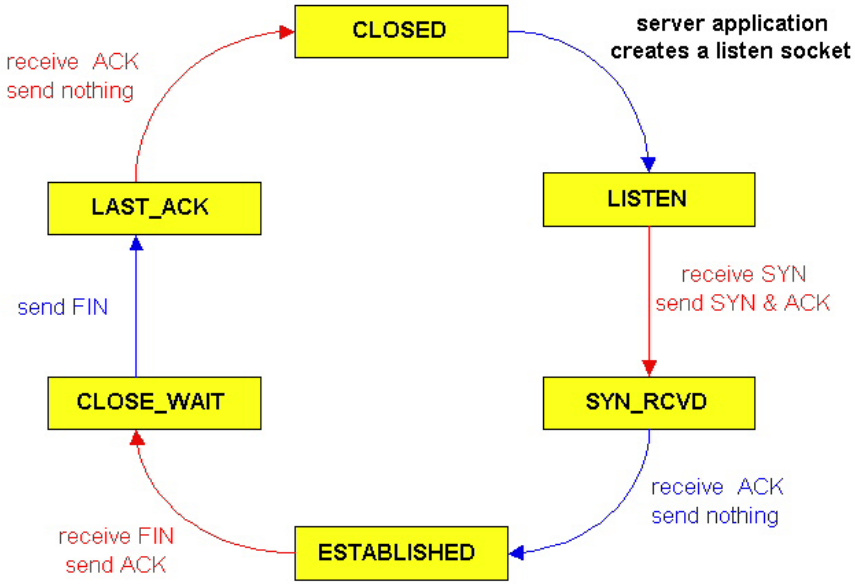


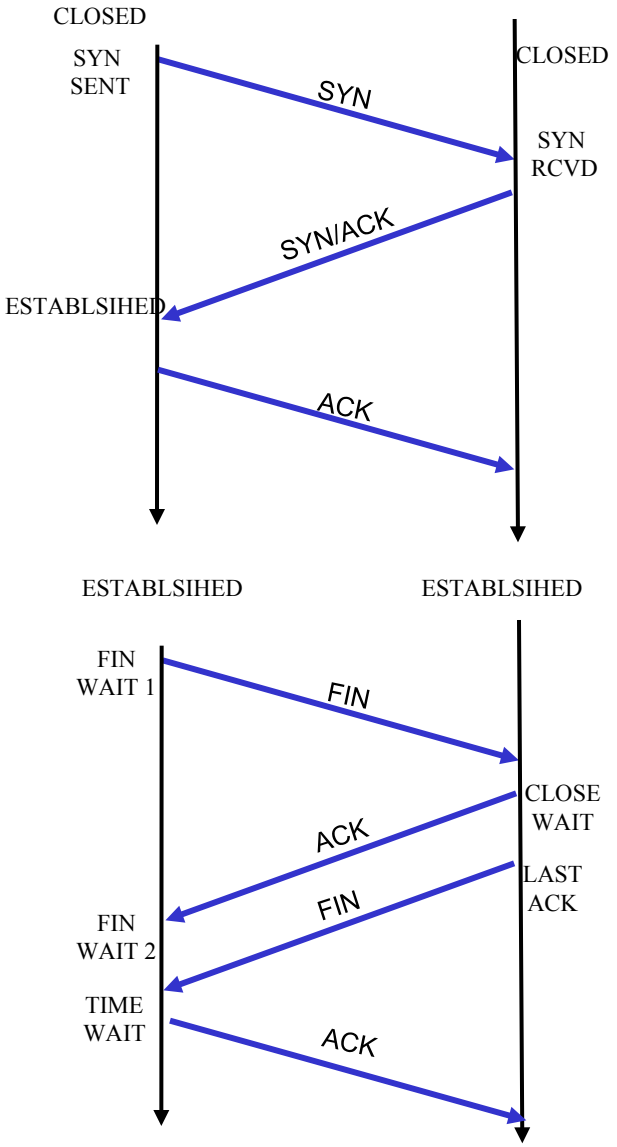


http://dsd.lbl.gov/TCP-tuning/ip-sysctl-2.6.txt

Example Transition (Client Connect + Client Close): Server

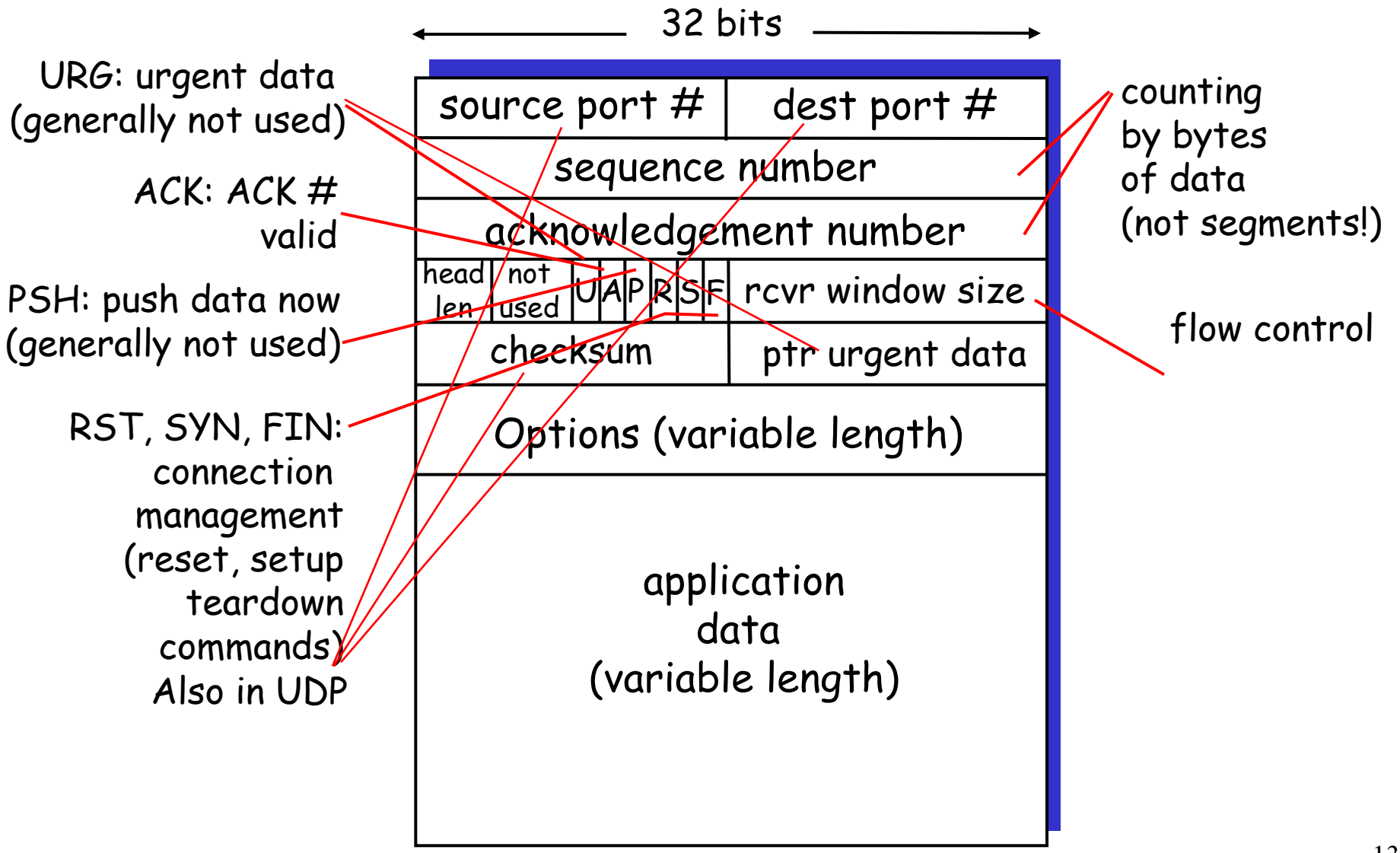
TCP lifecycle: wait for SYN/FIN





Exercise: Wireshark Capture TCP

TCP Reliability Basic: TCP Segment Structure



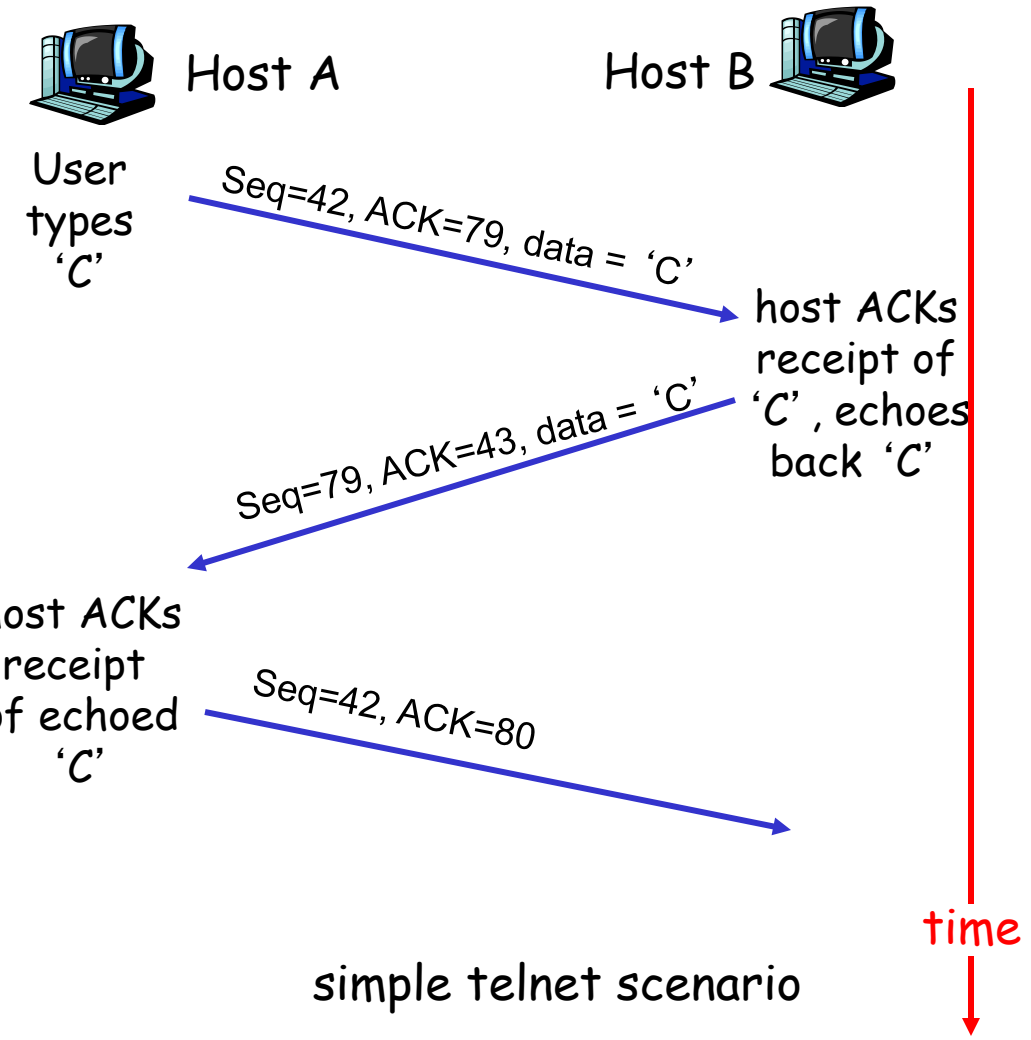
TCP Reliability Basic: TCP Seq. #'s and ACKs

Seq. #'s:

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

ACKs:

- seq # of next byte expected from other side
- cumulative ACK



simple telnet scenario

TCP Reliability Optimizations

- Multiple efforts to tune/optimize basic sliding window protocol, 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 [not required in Assignment 4]
  - 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 [not required in Assignment 4]
  - the loss detection optimization problem
    - Adaptive retransmission timeout (RTO) estimation, Fast retransmit before RTO [Required]

Offline Read: TCP Ack Generation Rules

[RFC 1122, RFC 2581]

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

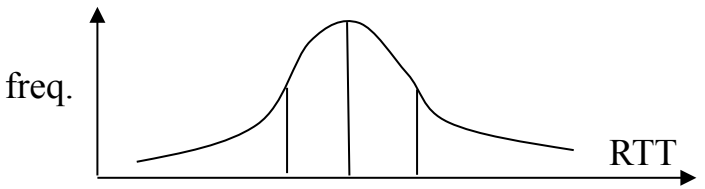
TCP Timeout Estimation

- Why is good timeout value important
  - Why is "too short" bad?
    - premature timeout
    - unnecessary retransmissions; many duplicates
  - Why is "too long" bad?
    - slow reaction to segment loss
- Ideal timeout and how to estimate?

TCP Retransmission Timeout (RTO) Alg

Problem:

- Ideal timeout = RTT, but RTT is not a fixed value
- Possibility: using the average of RTT, but this will generate many timeouts due to network variations



TCP solution:

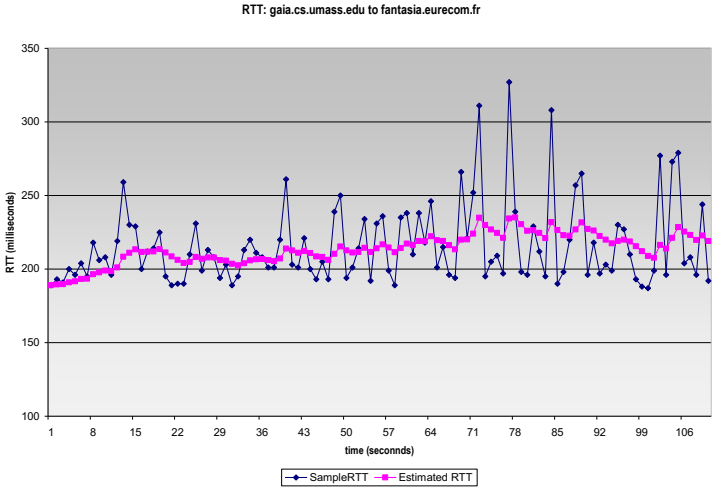
```
Timeout = EstRTT + 4 * DevRTT
```

TCP EstRTT and DevRTT Computation

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

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

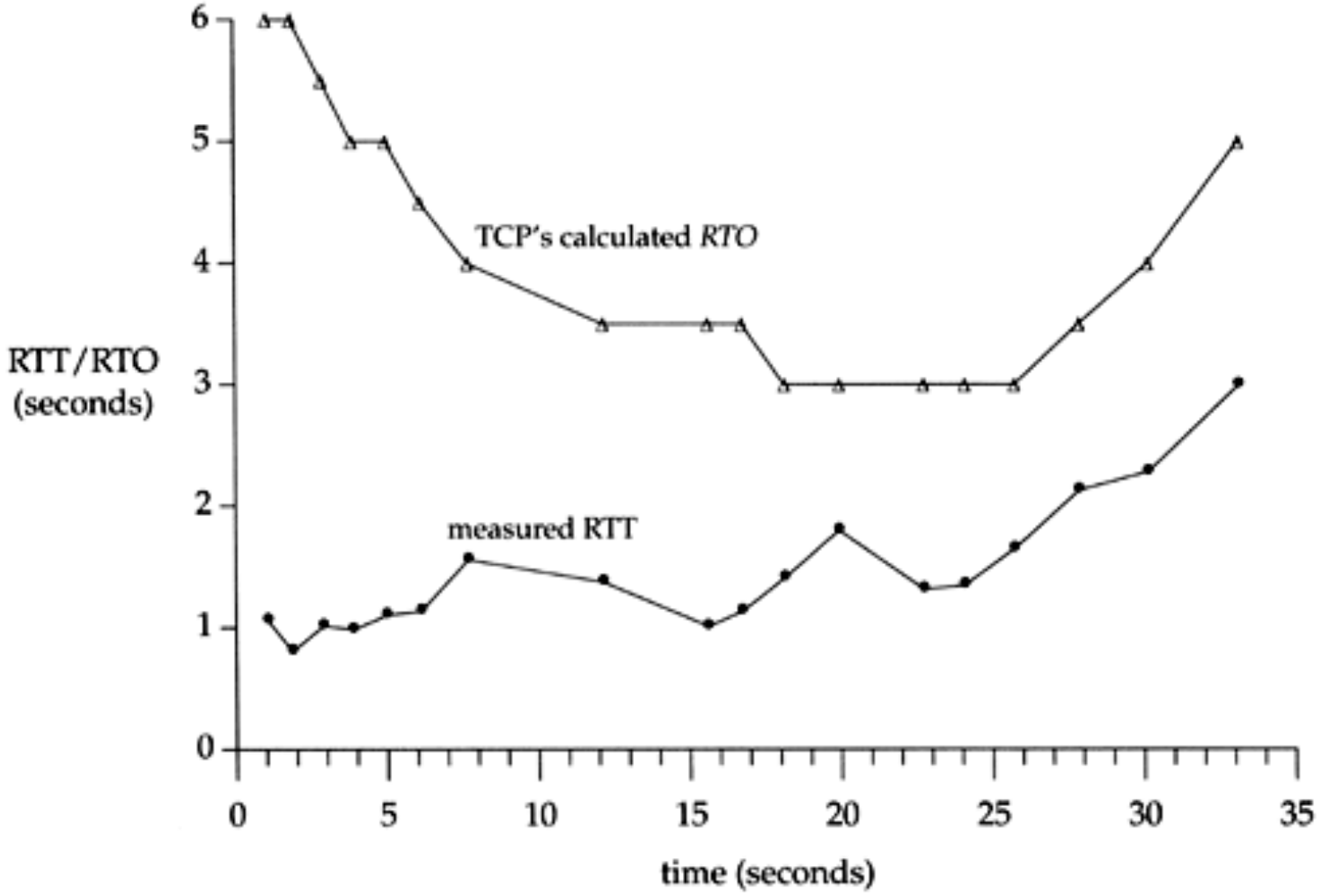
- **SampleRTT**: measured time from segment transmission until ACK receipt
- typical value: **alpha** = 0.125




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

(typically, beta = 0.25)
```

An Example TCP Session



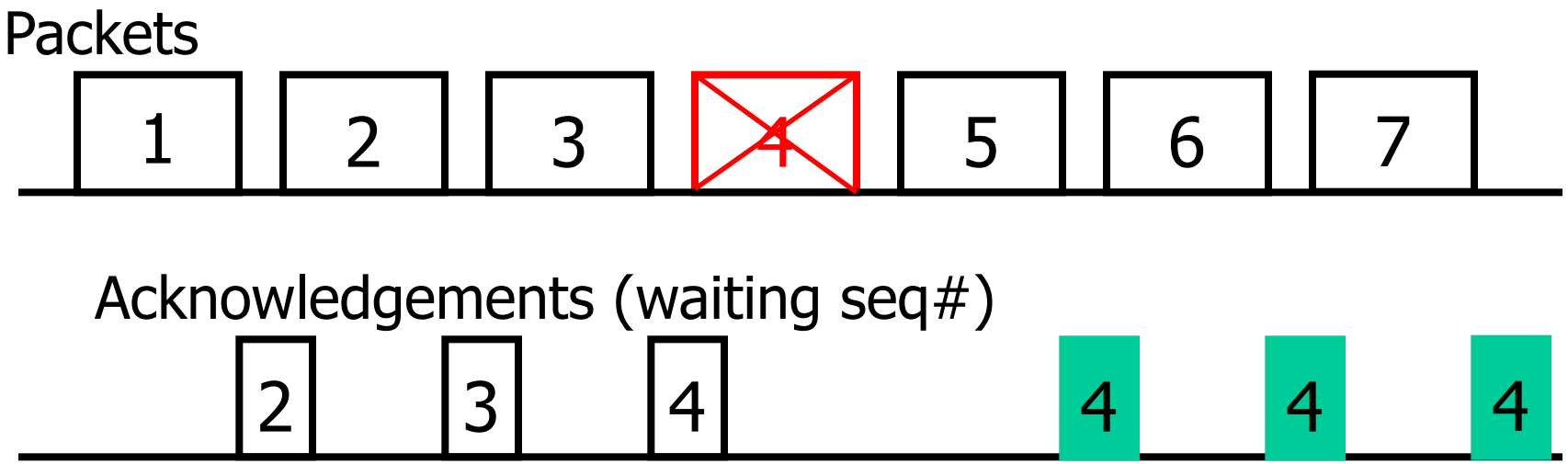
TCP Fast Retransmit

- Problem: Adding RTT variance into RTO may lead to longer than RTT delay.

- 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

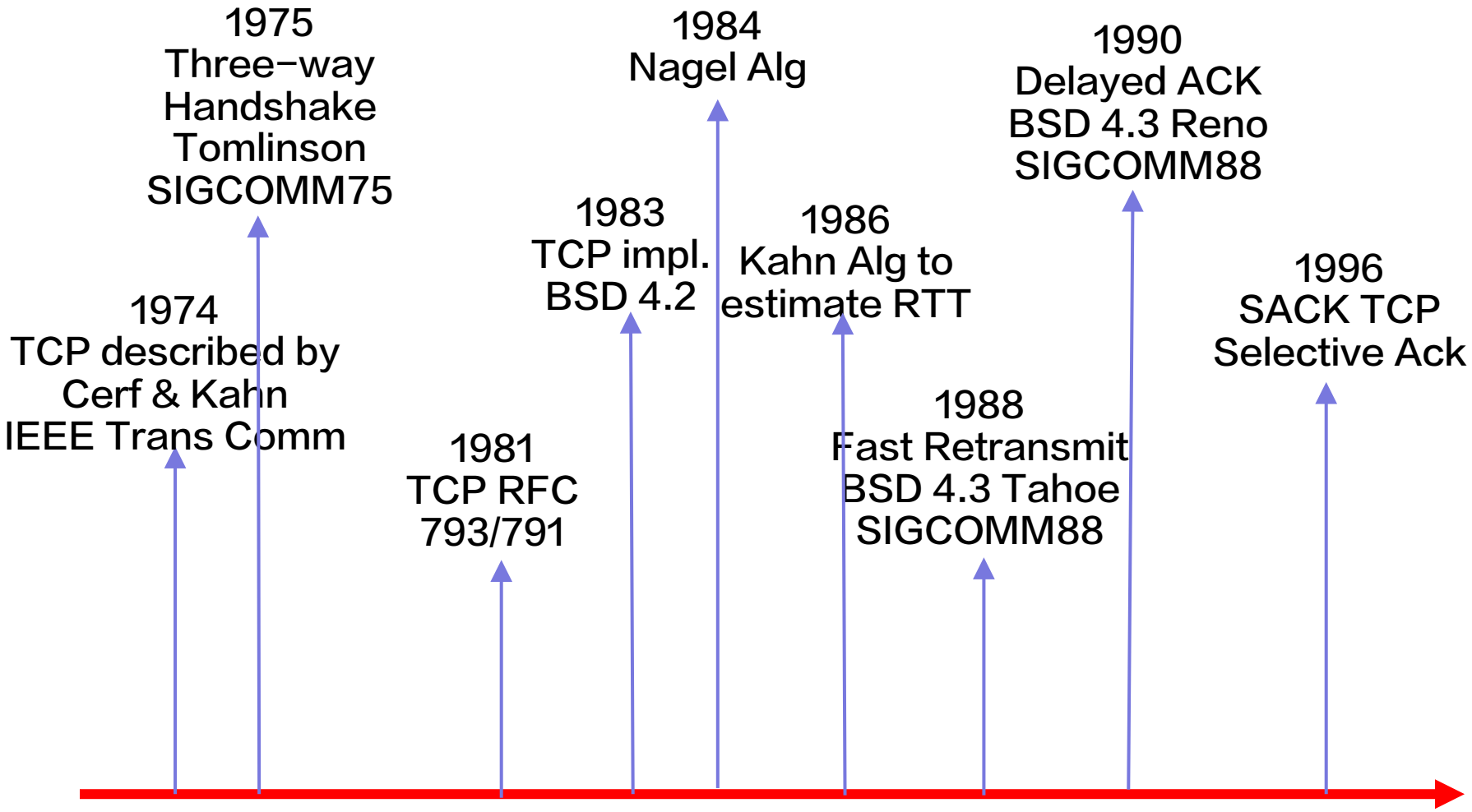
fast retransmit

TCP: reliable data transfer

Simplified TCP sender

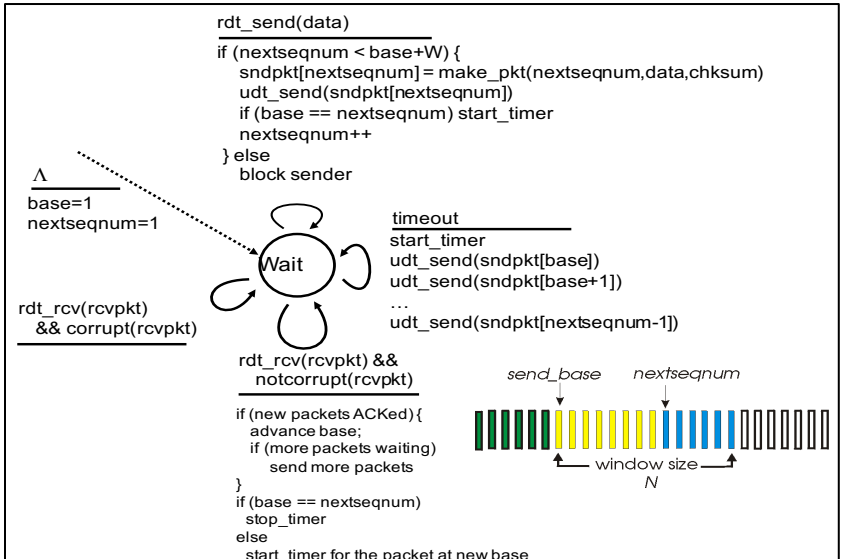
```
00  sendbase = initial_sequence number agreed by TWH
01   nextseqnum = initial_sequence number by TWH
02    loop (forever) {
03      switch(event)
04      event: data received from application above
05                if (window allows send)
06                   create TCP segment with sequence number nextseqnum
06                   if (no timer) start timer
07                   pass segment to IP
08                   nextseqnum = nextseqnum + length(data)
                  else put packet in buffer
09       event: timer timeout for sendbase
10            retransmit segment
11            compute new timeout interval
12            restart timer
13       event: ACK received, with ACK field value of y
14            if (y > sendbase) { /* cumulative ACK of all data up to y */
15                 cancel the timer for sendbase
16                 sendbase = y
17                 if (no timer and packet pending) start timer for new sendbase
17                 while (there are segments and window allow)
18                      sent a segment;
18            }
19            else { /* y==sendbase, duplicate ACK for already ACKed segment */
20                 increment number of duplicate ACKs received for y
21                 if (number of duplicate ACKS received for y == 3) {
22                      /* TCP fast retransmit */
23                    resend segment with sequence number y
24                    restart timer for segment y
25                 }
26       }  /* end of loop forever */
```

TCP Developed in Time



Summary: Transport Reliability

- Basic structure: sliding window protocols
- Basic analytical techniques: joint state machine, traces, state invariants
- Realization example: TCP
- Remaining issue: How to determine the "right" parameters?



sender

data from above :

- unACKed packets is less than window size W, send; otherwise block app.

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+W-1]:

- mark pkt n as received
- update sendbase to the first packet unACKed

receiver

pkt n in [rcvbase, rcvbase+W-1]

- send ACK(n)
- if (out-of-order)
  mark and buffer pkt n
  else /*in-order*/
  deliver any in-order packets

otherwise:

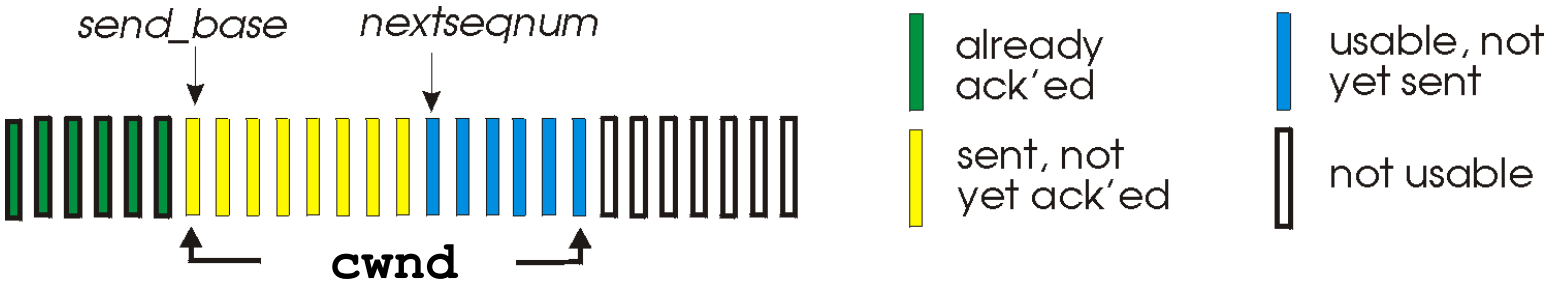
- ignore

History

- Network collapse in the mid-1980s
  - UCB ←→ LBL throughput dropped by 1000X !
- The intuition was that the collapse was caused by wrong parameters…
  - Key parameters for TCP in mid-1980s
    - fixed window size W
    - timeout value = 2 RTT (already discussed)

Key Parameter: Sliding Window Size

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



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

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

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

Outline

- Admin and recap
- Transport reliability
  - sliding window protocols
  - connection management
  - TCP reliability
- Transport congestion control

Some General Questions

Big picture question:

- How to determine a flow's sending rate?

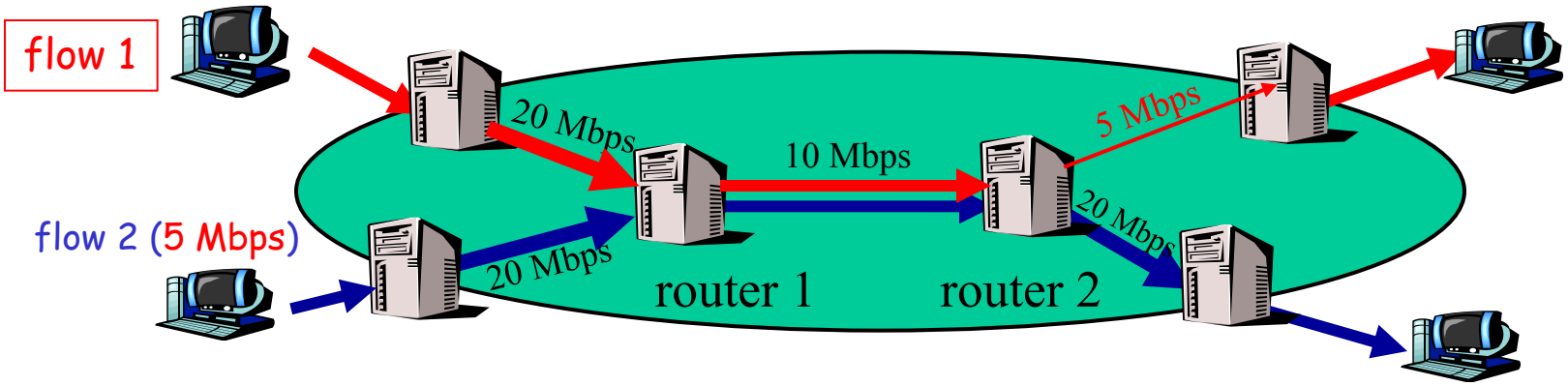
For better understanding, we need to look at a few basic questions:

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

Outline

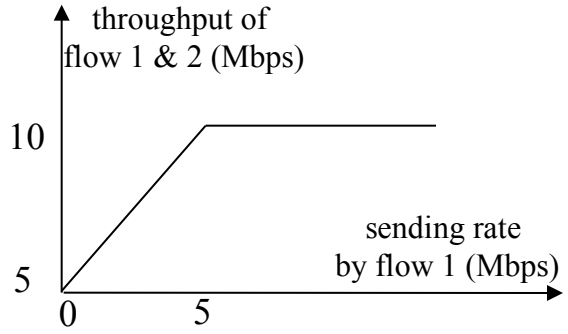
- Admin and recap
- Transport reliability
  - sliding window protocols
  - connection management
  - TCP reliability
- Transport congestion control
  - what is congestion (cost of congestion)

Cause/Cost of Congestion: Single Bottleneck

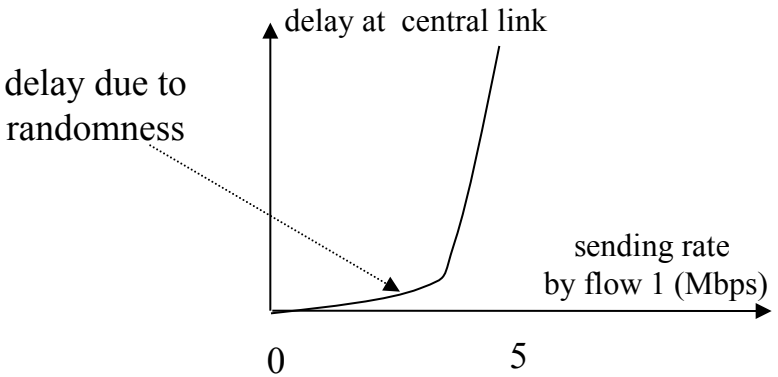


- Flow 2 has a fixed sending rate of 5 Mbps
- We vary the sending rate of flow 1 from 0 to 20 Mbps
- Assume
  - no retransmission; link from router 1 to router 2 has infinite buffer

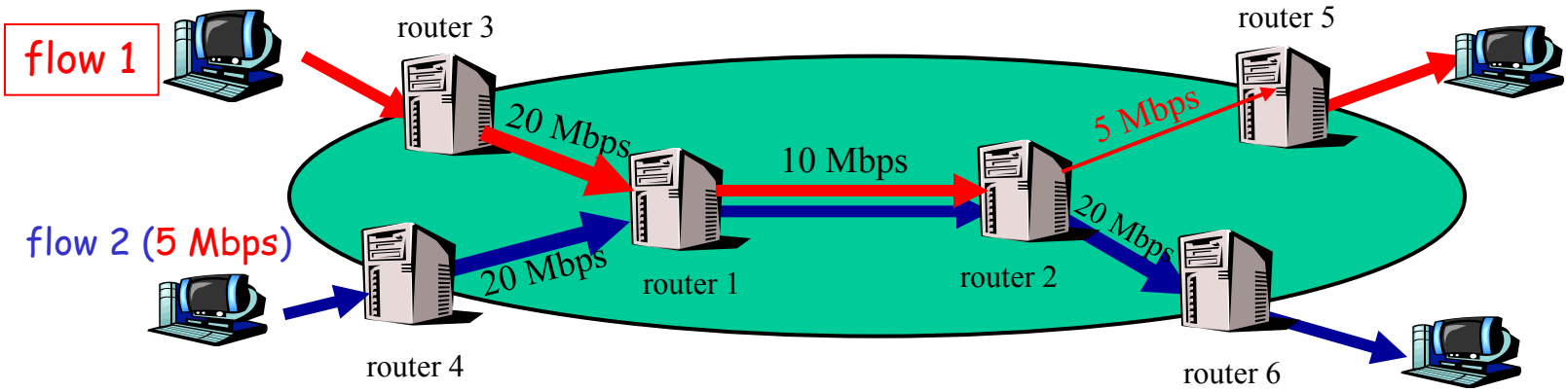
throughput: e2e packets delivered in unit time



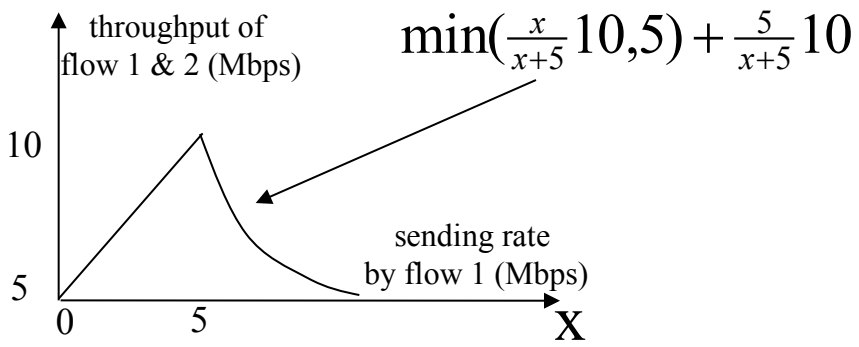
Delay?



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

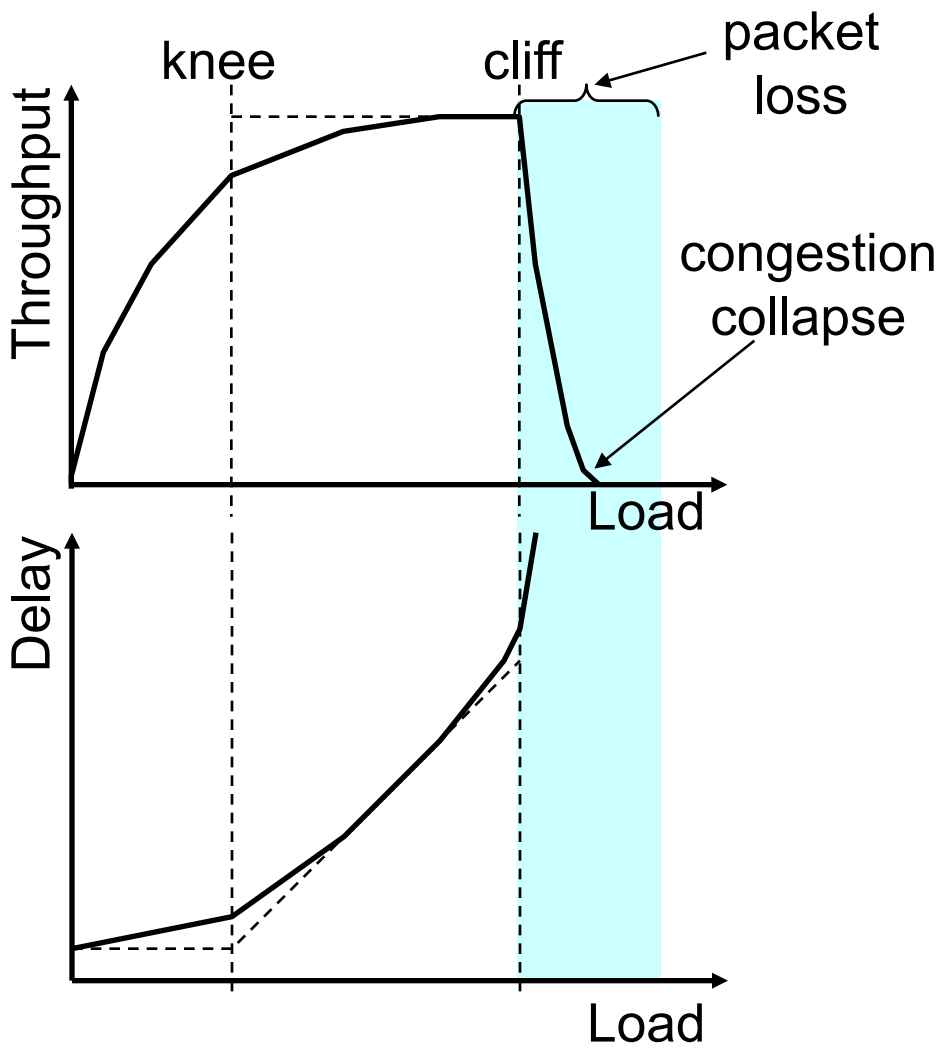


- Zombie packet: a packet dropped at the link from router 2 to router 5; the upstream transmission from router 1 to router 2 used for that packet was wasted!

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



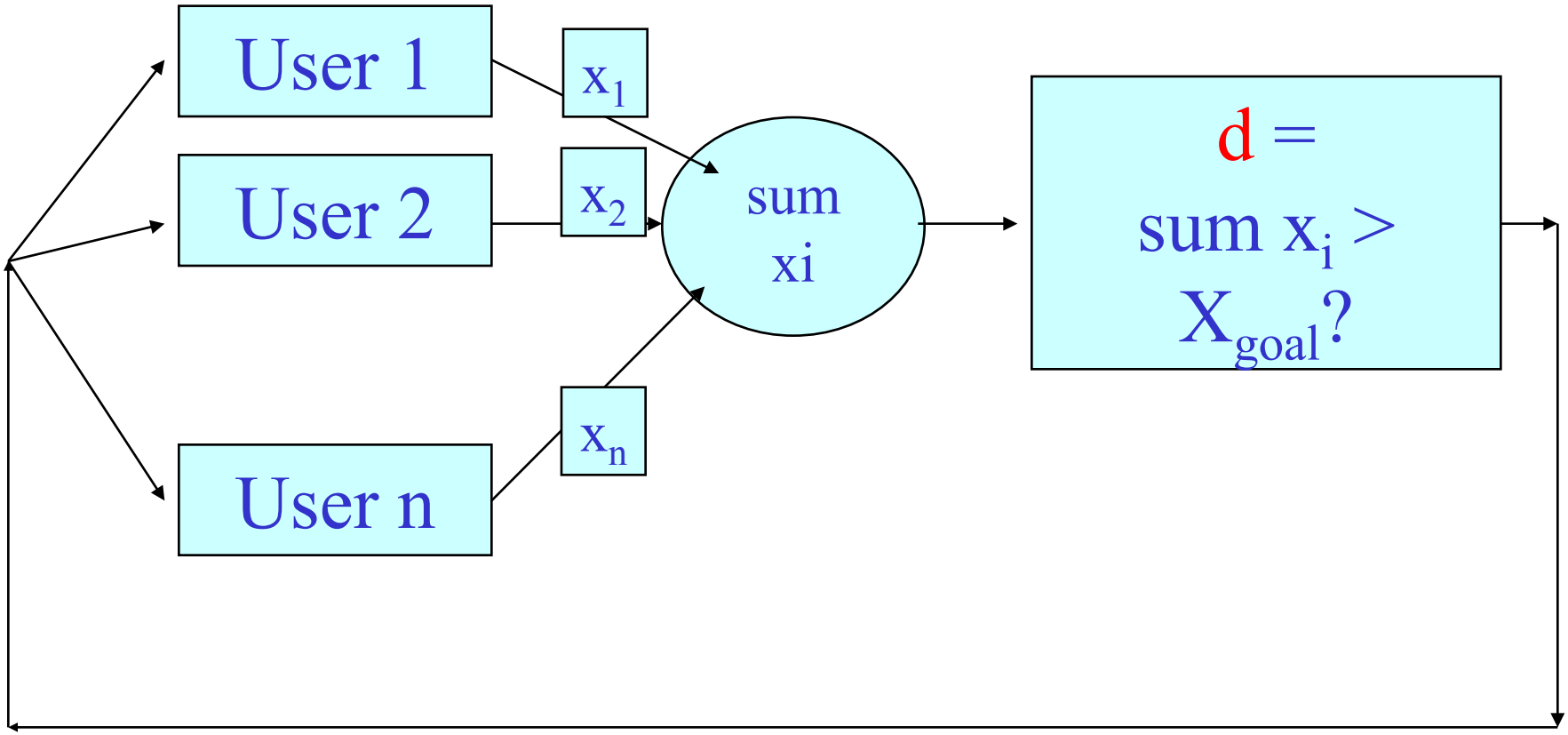
Outline

- Admin and recap
- Transport reliability
  - sliding window protocols
  - connection management
  - TCP reliability
- Transport congestion control
  - what is congestion (cost of congestion)
  - basic congestion control alg.

The Desired Properties of a Congestion Control Scheme

- Efficiency: close to full utilization but low delay
  - fast convergence after disturbance
- Fairness (resource sharing)
- Distributedness (no central knowledge for scalability)

Derive CC: A Simple Model



Flows observe congestion signal d, and locally take actions to adjust rates.

Linear Control

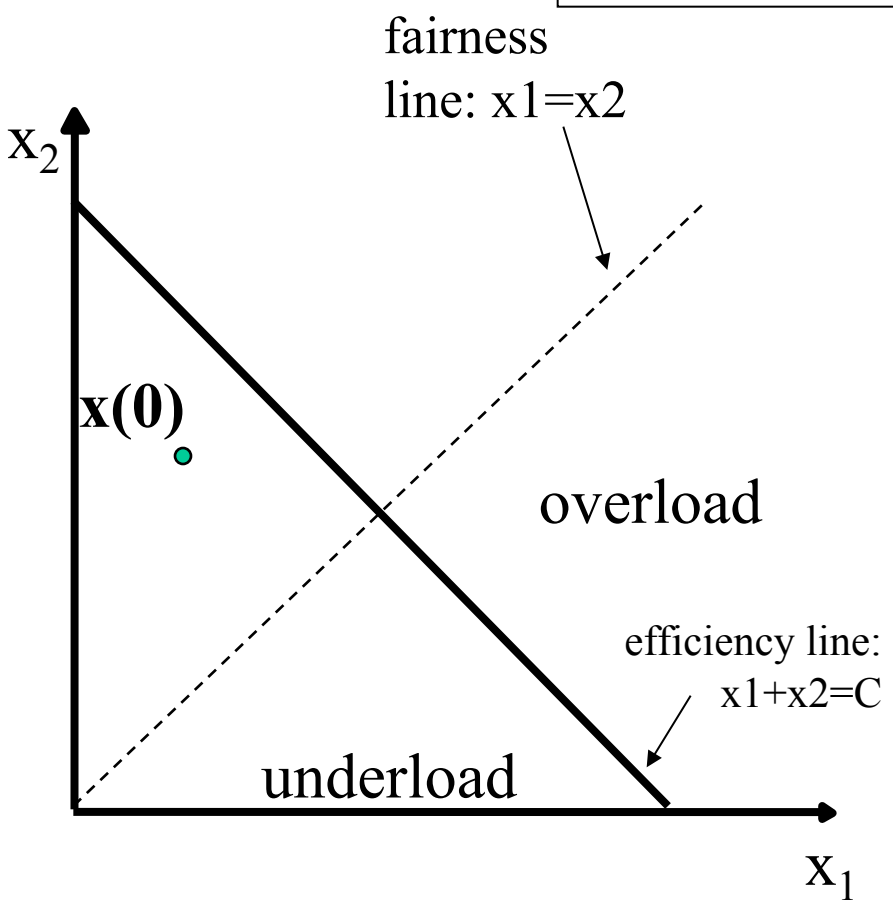
- Proposed by Chiu and Jain (1988)
- Considers the simplest class of control strategy

$$x_i(t+1) = \begin{cases} a_I + b_I x_i(t) & \text{if d(t) = no cong.} \\ a_D + b_D x_i(t) & \text{if d(t) = cong.} \end{cases}$$

Discussion: values of the parameters?

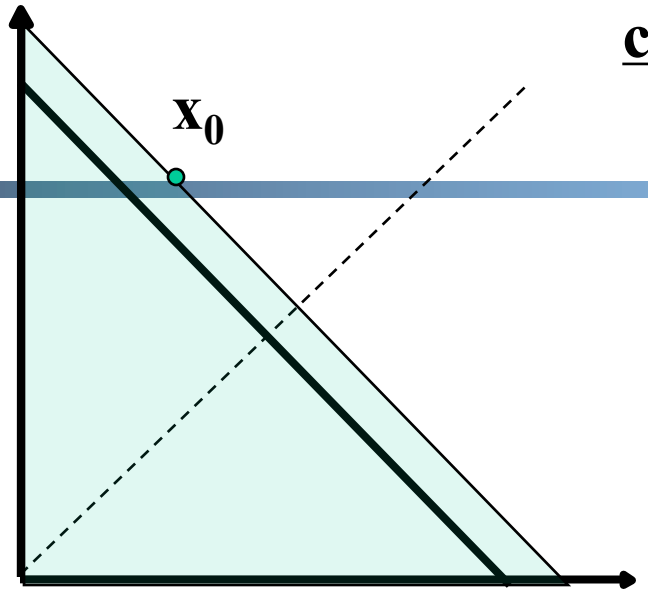
State Space of Two Flows

$$x_i(t+1) = \begin{cases} a_I + b_I x_i(t) & \text{if d(t) = no cong.} \\ a_D + b_D x_i(t) & \text{if d(t) = cong.} \end{cases}$$

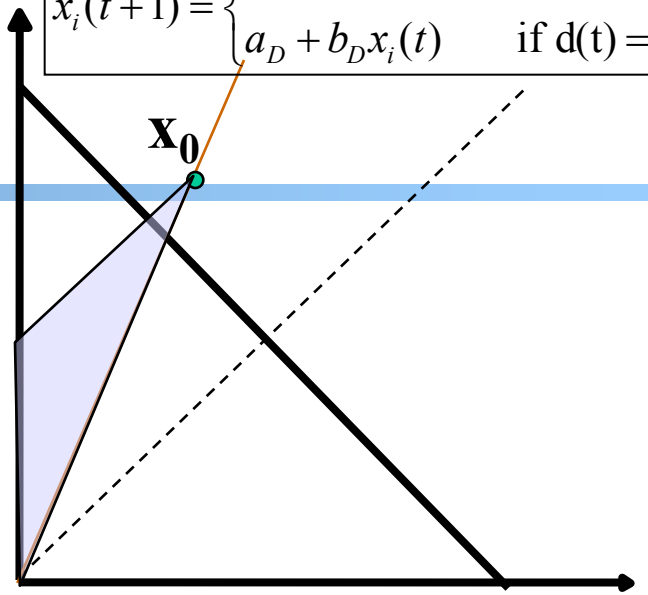


congestion

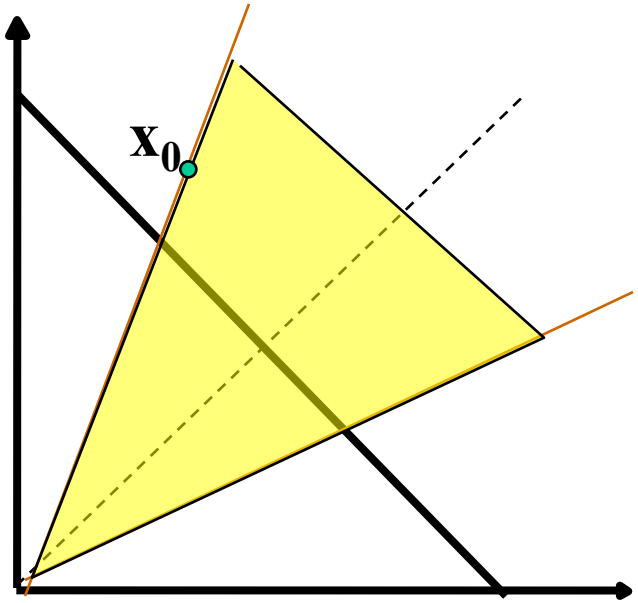
$$x_i(t+1) = \begin{cases} a_I + b_I x_i(t) & \text{if d(t) = no cong.} \\ a_D + b_D x_i(t) & \text{if d(t) = cong.} \end{cases}$$



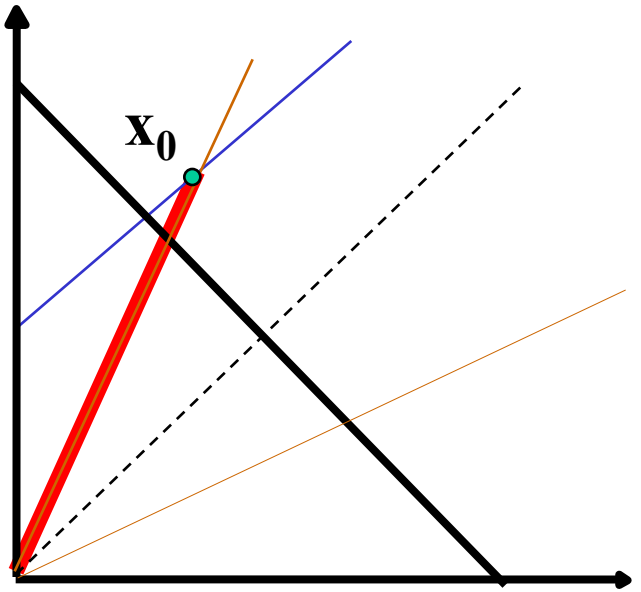
efficiency



efficiency: distributed linear rule



fairness



intersection

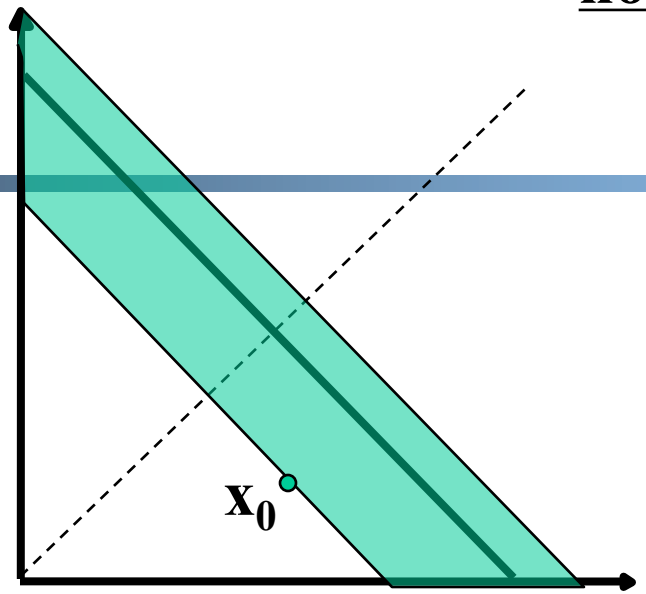
Implication: Congestion (overload) Case

- In order to get closer to efficiency and fairness after each update, decreasing of rate must be multiplicative decrease (MD)
  - $a_D = 0$
  - $b_D < 1$

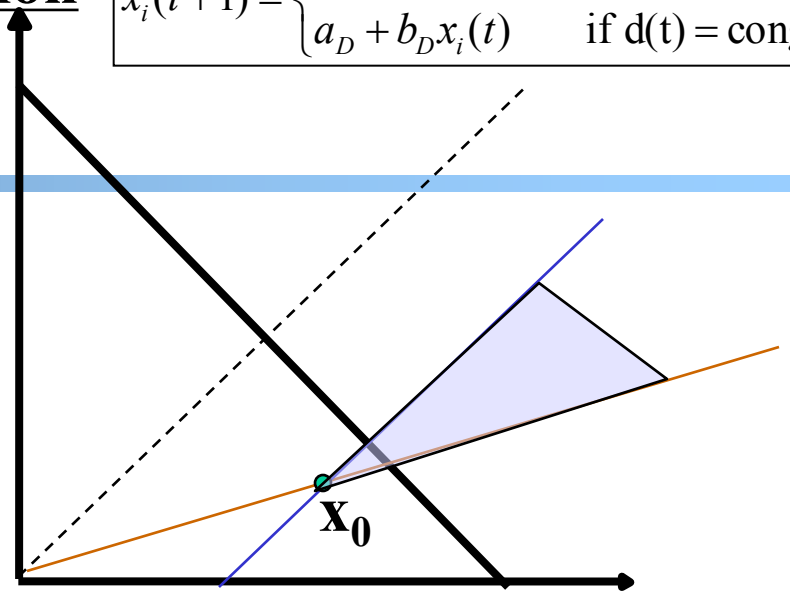
$$x_i(t+1) = \begin{cases} a_I + b_I x_i(t) & \text{if d(t) = no cong.} \\ b_D x_i(t) & \text{if d(t) = cong.} \end{cases}$$

no-congestion

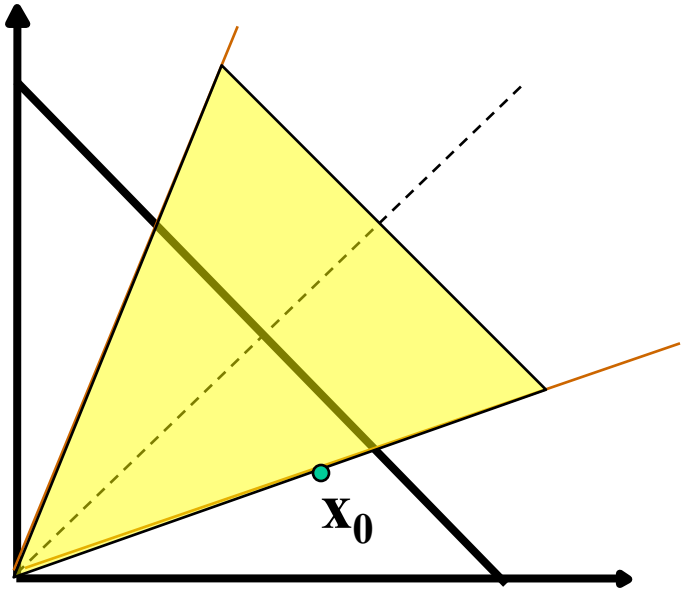
$$x_i(t+1) = \begin{cases} a_I + b_I x_i(t) & \text{if d(t) = no cong.} \\ a_D + b_D x_i(t) & \text{if d(t) = cong.} \end{cases}$$



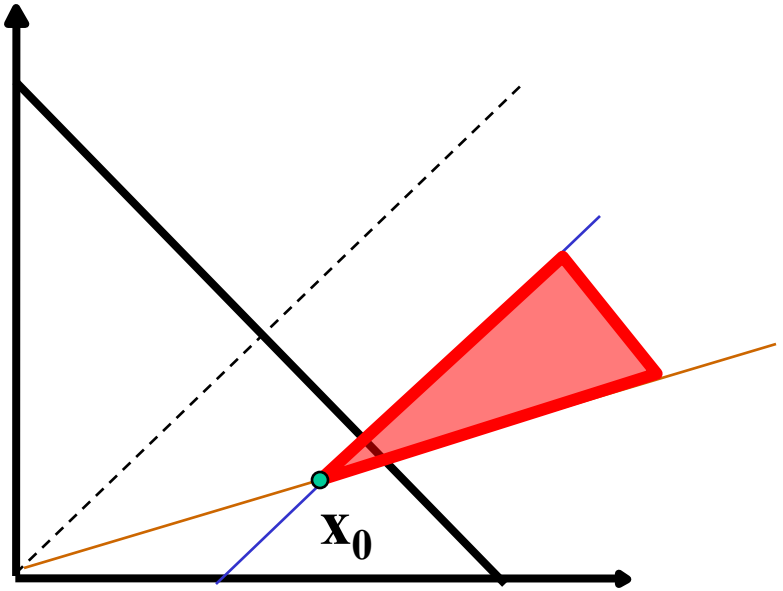
efficiency



efficiency: distributed linear rule



fairness



convergence

Implication: No Congestion Case

- In order to get closer to efficiency and fairness after each update, additive and multiplicative increasing (AMI), i.e.,
  - $a_I > 0$, $b_I > 1$

$$x_i(t+1) = \begin{cases} a_I + b_I x_i(t) & \text{if d(t) = no cong.} \\ b_D x_i(t) & \text{if d(t) = cong.} \end{cases}$$

- Simply additive increase gives better improvement in fairness (i.e., getting closer to the fairness line)
- Multiplicative increase may grow faster

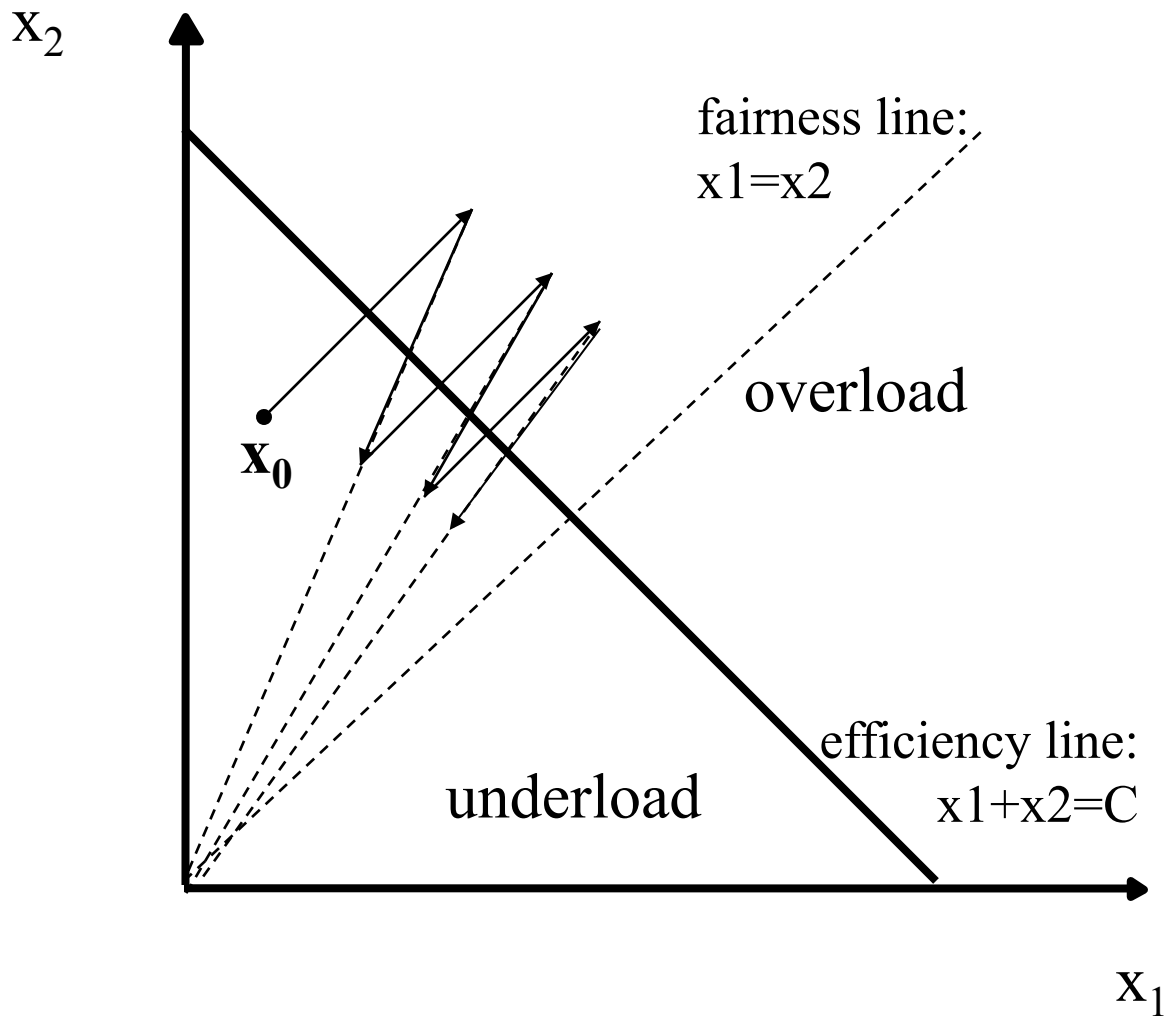
Intuition: State Trace Analysis of Four Special Cases

	Additive Decrease	Multiplicative Decrease
Additive Increase	AIAD ($b_I=b_D=1$)	AIMD ($b_I=1$, $a_D=0$)
Multiplicative Increase	MIAD ($a_I=0$, $b_I>1$, $b_D=1$)	MIMD ($a_I=a_D=0$)

$$x_i(t+1) = \begin{cases} a_I + b_I x_i(t) & \text{if d(t) = no cong.} \\ a_D + b_D x_i(t) & \text{if d(t) = cong.} \end{cases}$$

Discussion: state transition trace.

AIMD: State Transition Trace



Intuition: Another Look

- Consider the difference or ratio of the rates of two flows
  - AIAD
  - MIMD
  - MIAD
  - AIMD

Mapping A(M)I-MD to Protocol

- Question to look at: How do we apply the A(M)I-MD algorithm?

$$x_i(t+1) = \begin{cases} a_I + x_i(t) & \text{if d(t) = no cong.} \\ b_D x_i(t) & \text{if d(t) = cong.} \end{cases}$$

Rate-based vs. Window-based

Rate-based:

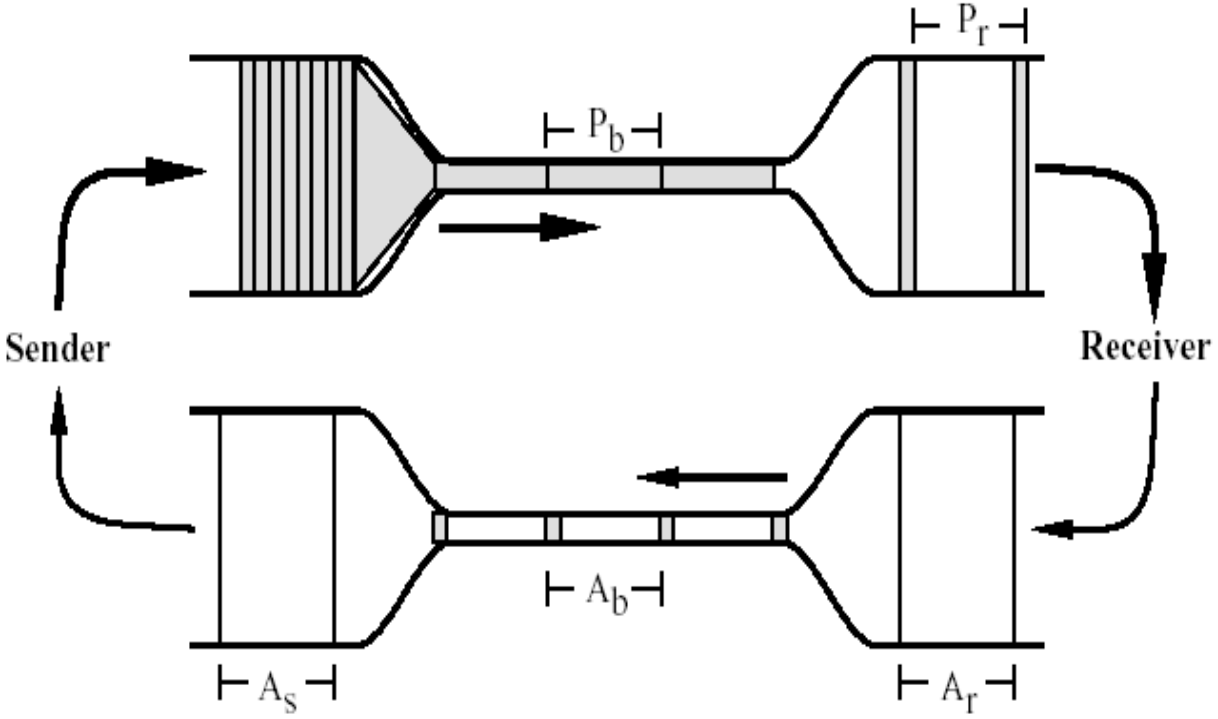
- Congestion control by explicitly controlling the sending rate of a flow, e.g., set sending rate to 128Kbps
- Example: ATM

Window-based:

- Congestion control by controlling the window size of a transport scheme, e.g., set window size to 64KBytes
- Example: TCP

Discussion: rate-based vs. window-based

Window-based Congestion Control



- Window-based congestion control is self-clocking: considers flow conservation, and adjusts to RTT variation automatically.
- Hence, for better safety, more designs use window-based design.

Mapping A(M)I-MD to Protocol

- Driving question: How do we apply the A(M)I-MD algorithm for sliding window protocols?

$$x_i(t+1) = \begin{cases} a_I + x_i(t) & \text{if d(t) = no cong.} \\ b_D x_i(t) & \text{if d(t) = cong.} \end{cases}$$

Outline

- Admin and recap
- Transport reliability
  - sliding window protocols
  - connection management
  - TCP reliability
- Transport congestion control
  - what is congestion (cost of congestion)
  - basic congestion control alg.
  - TCP/reno congestion control

TCP Congestion Control

- Closed-loop, end-to-end, window-based congestion control designed by Van Jacobson in late 1980s, based on the AIMD
- Worked in a large range of bandwidth values: the bandwidth of the Internet has increased by more than 200,000 times
- Many versions
  - TCP/Tahoe: this is a less optimized version
  - TCP/Reno: many OSs today implement Reno type congestion control

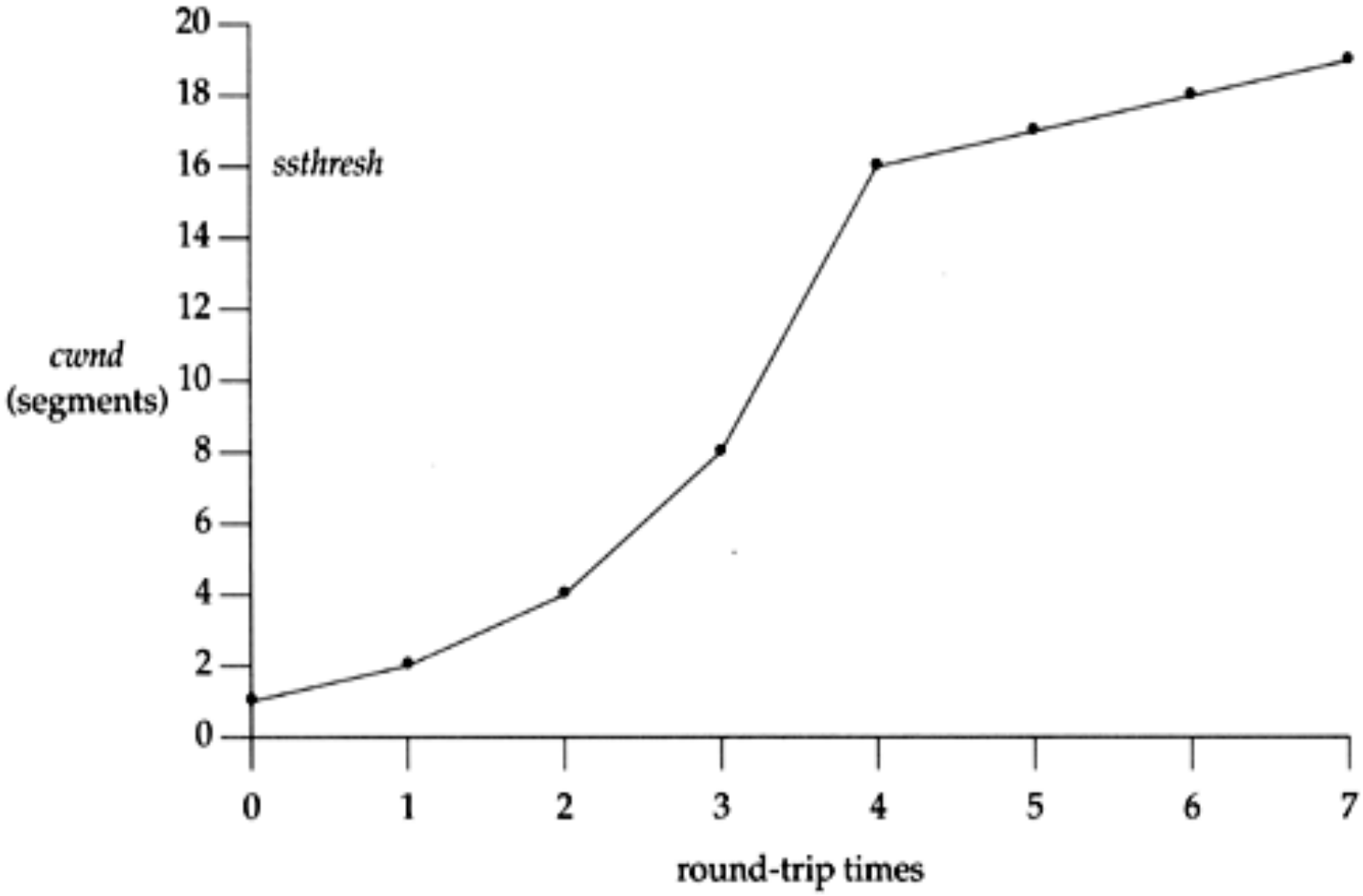
For more details: see TCP/IP illustrated; or read http://lxr.linux.no/source/net/ipv4/tcp_input.c for linux implementation

Basic Structure

- Two "phases"
  - MI: slow-start
  - AIMD: congestion avoidance

- Important variables:
  - **`cwnd`**: congestion window size
  - **`ssthresh`**: threshold between the slow-start phase and the congestion avoidance phase

Visualization of the Two Phases



MI: Slow Start

- Algorithm: MI
  - double *cwnd* every RTT until network congested
- Goal: getting to equilibrium gradually but quickly, to get a rough estimate of the optimal of *cwnd*

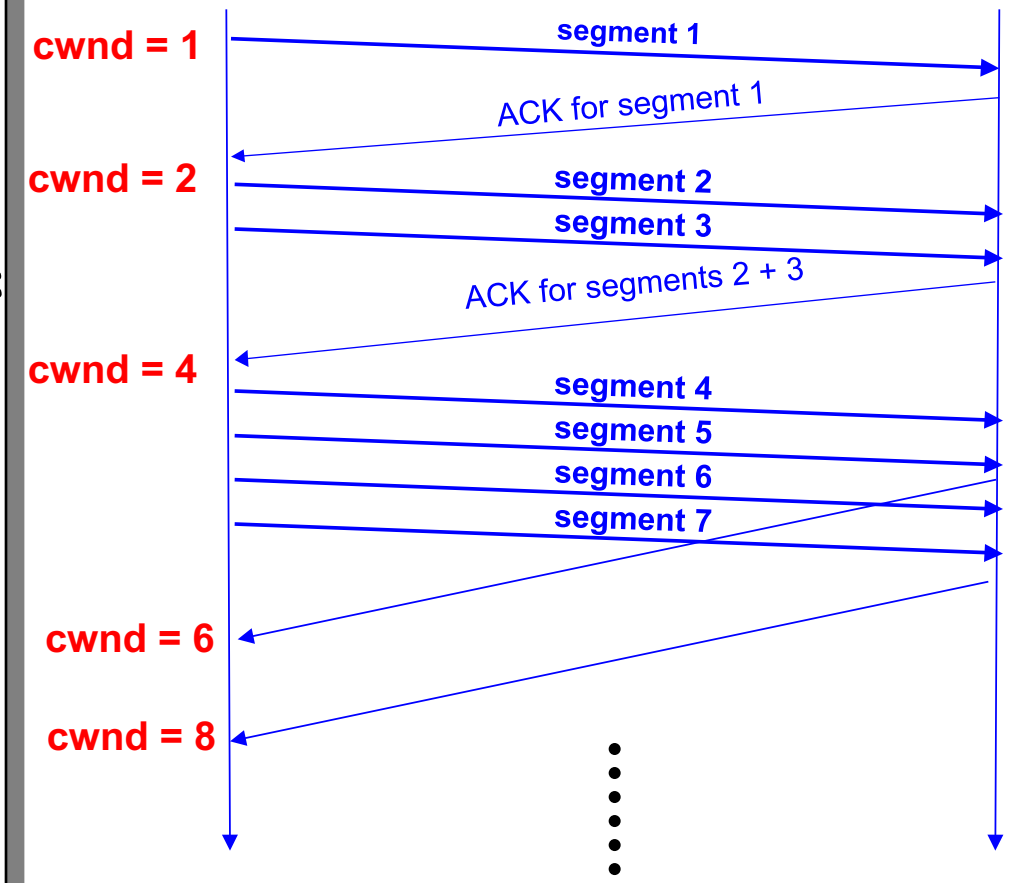
MI: Slow-start

Initially:

```
cwnd = 1;
ssthresh = infinite (e.g., 64K);
```

For each newly ACKed segment:

```
if (cwnd < ssthresh)
    /* MI: slow start*/
    cwnd = cwnd + 1;
```



AIMD: Congestion Avoidance

- Algorithm: AIMD
  - increases window by 1 per round-trip time
  - cuts window size
    - to half when detecting congestion by 3DUP
    - to 1 if timeout
    - if already timeout, doubles timeout
- Goal: Maintains equilibrium and reacts around equilibrium

TCP/Reno Full Alg

Initially:

```
    cwnd = 1;
    ssthresh = infinite (e.g., 64K);
```

For each newly ACKed segment:

```
    if (cwnd < ssthresh)        // slow start: MI
        cwnd = cwnd + 1;
    else
                                // congestion avoidance; AI
        cwnd += 1/cwnd;
```

Triple-duplicate ACKs:

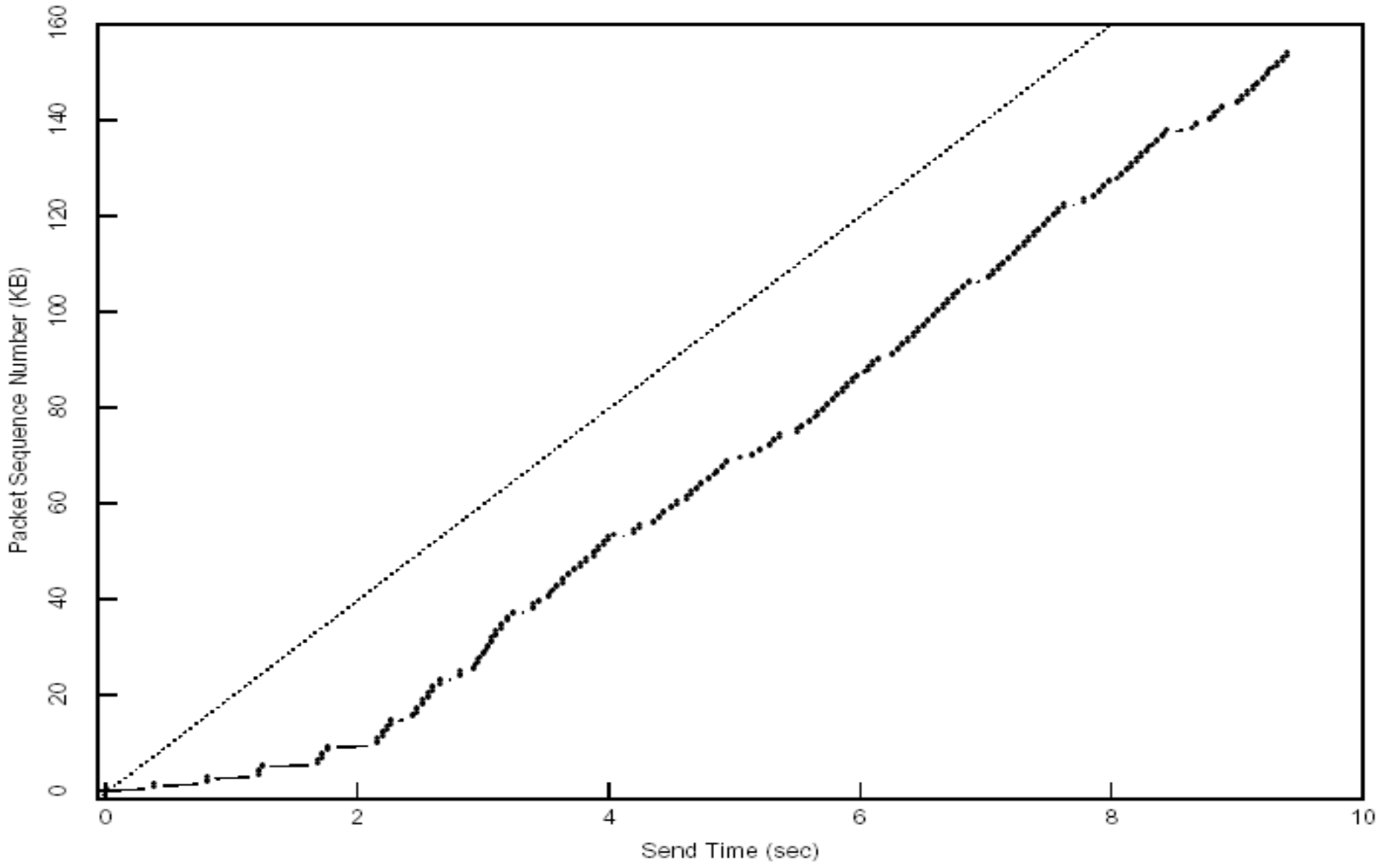
```
                                // MD
    cwnd = ssthresh = cwnd/2;
```

Timeout:

```
    ssthresh = cwnd/2;          // reset
    cwnd = 1;
```

(if already timed out, double timeout value; this is called exponential backoff)

Typical TCP/Reno Transmission Behavior



See [Jac89]