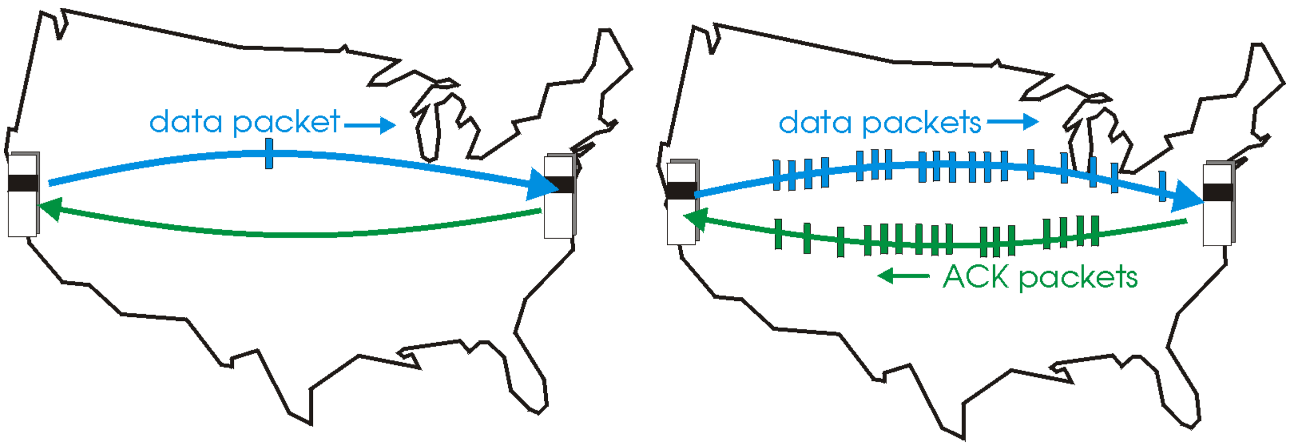
Transport Congestion Control

Recap: Sliding Window Protocols

- Basic idea: using pipelining: to make better use of link bandwidth, sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

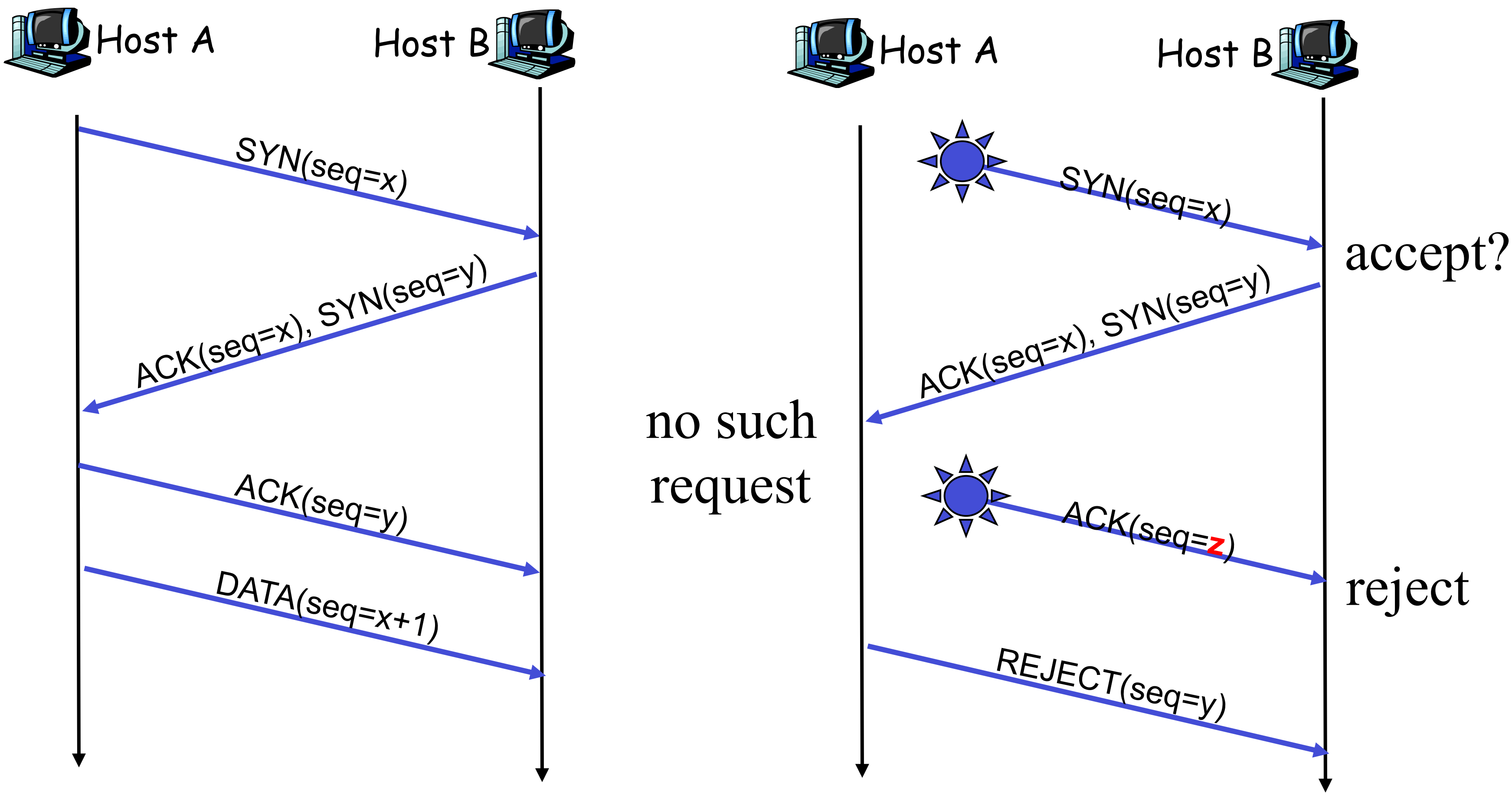


(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

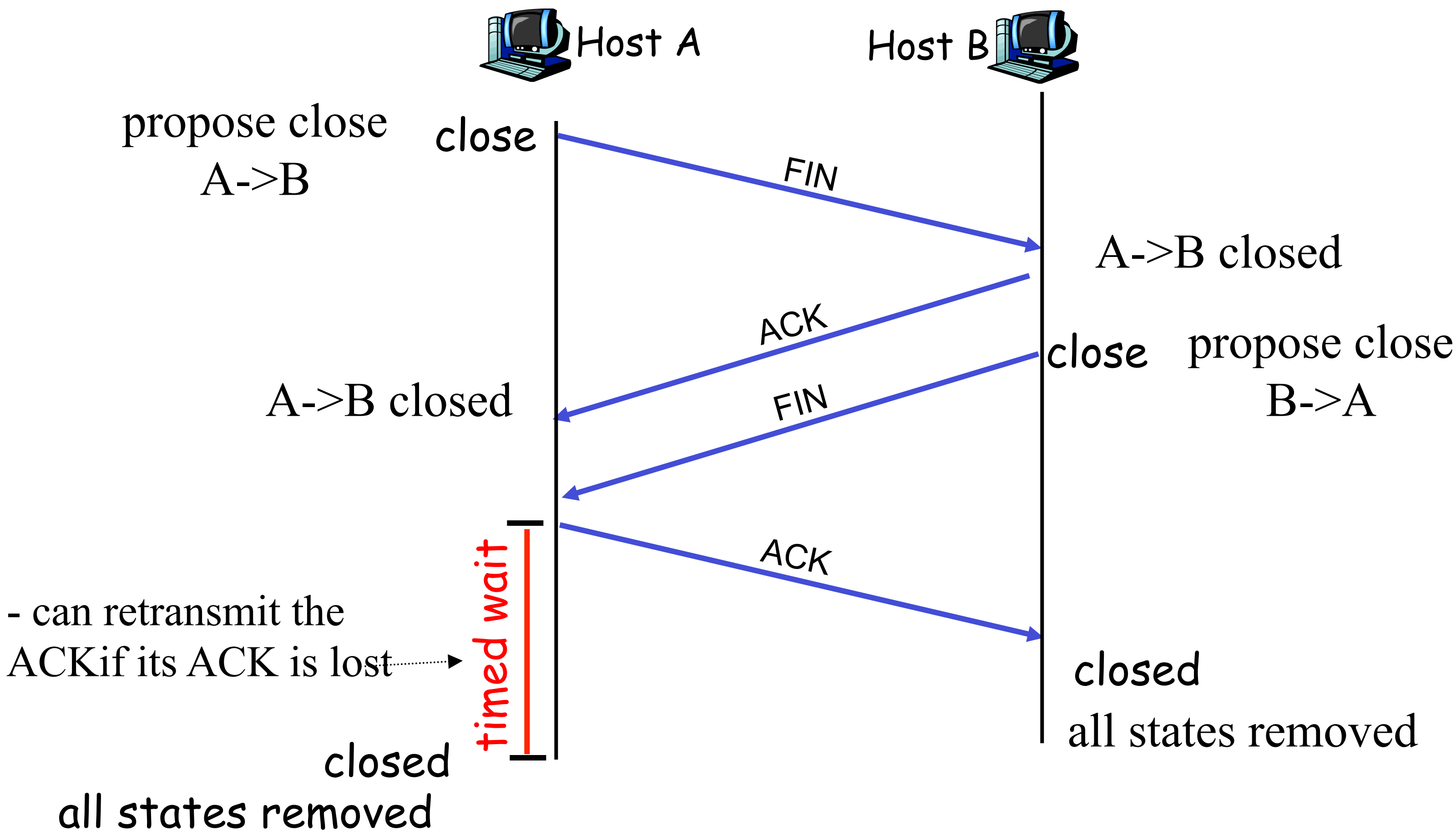
- Two generic forms of pipelined protocols: *go-Back-N, selective repeat*

Three Way Handshake (TWH) [Tomlinson 1975]

- To ensure that the other side does want to send a request



Four Way Teardown



A Summary of Questions

- How to improve the performance of rdt3.0?
  - Sliding window protocols
- What if there are duplication and reordering?
  - Network guarantee: max packet life time
  - Transport guarantee: not reuse a seq# before life time
  - Seq# management and connection management
- How to determine the "right" parameters?

Outline

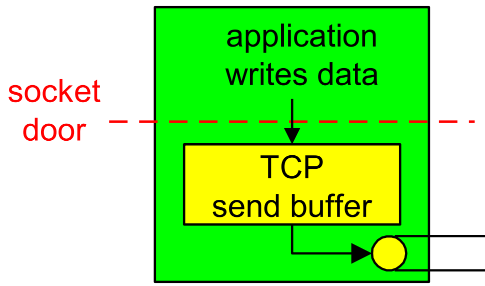
- Recap
- *TCP reliability*
- Introduction to congestion control

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

- Point-to-point: one sender, one receiver
- Reliable transport using sliding window protocol
- Flow controlled and congestion controlled

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
- **Retransmissions are triggered by**
  - Timeout events
  - Duplicate acks (fast retransmit)



We consider a simplified TCP sender: ignore flow control, congestion control

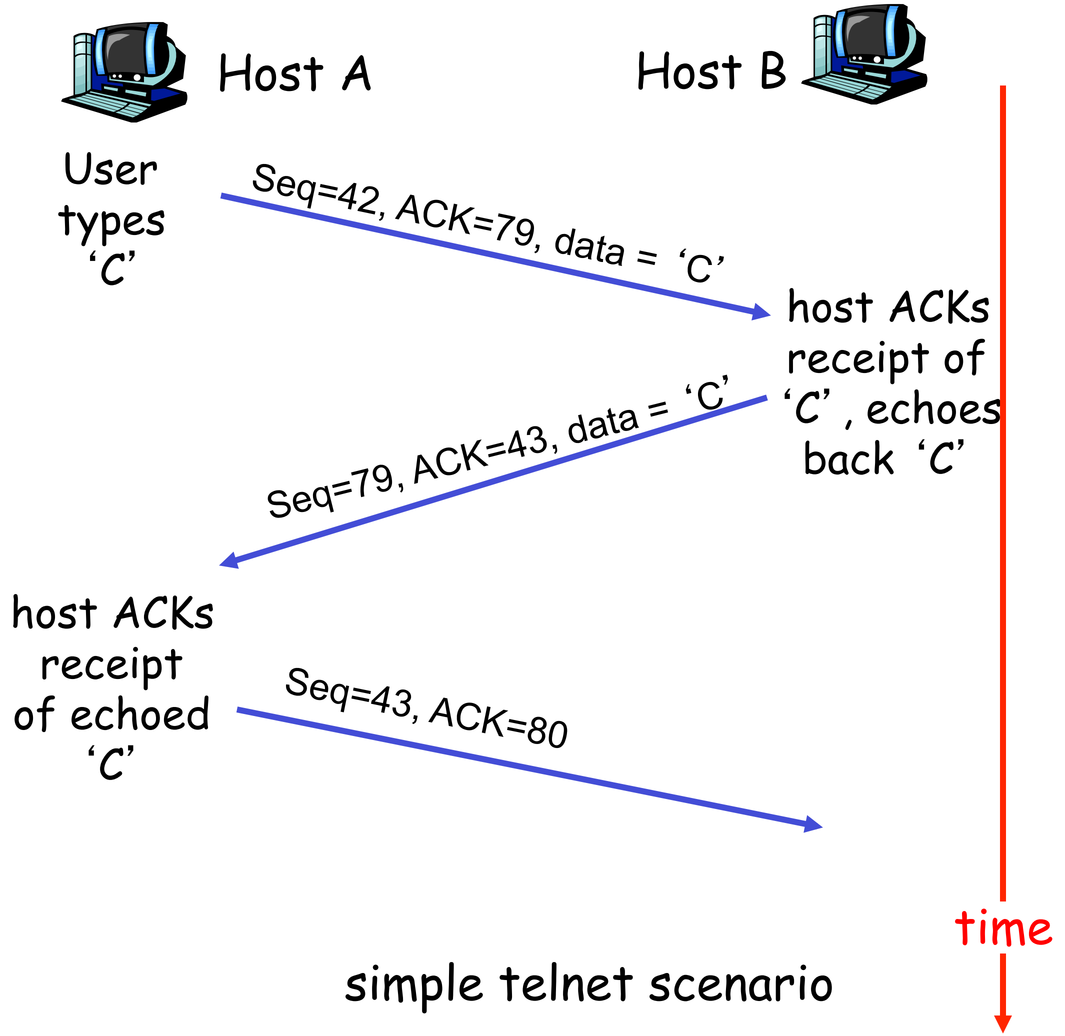
TCP Seq. #'s and ACKs

Seq. #'s:

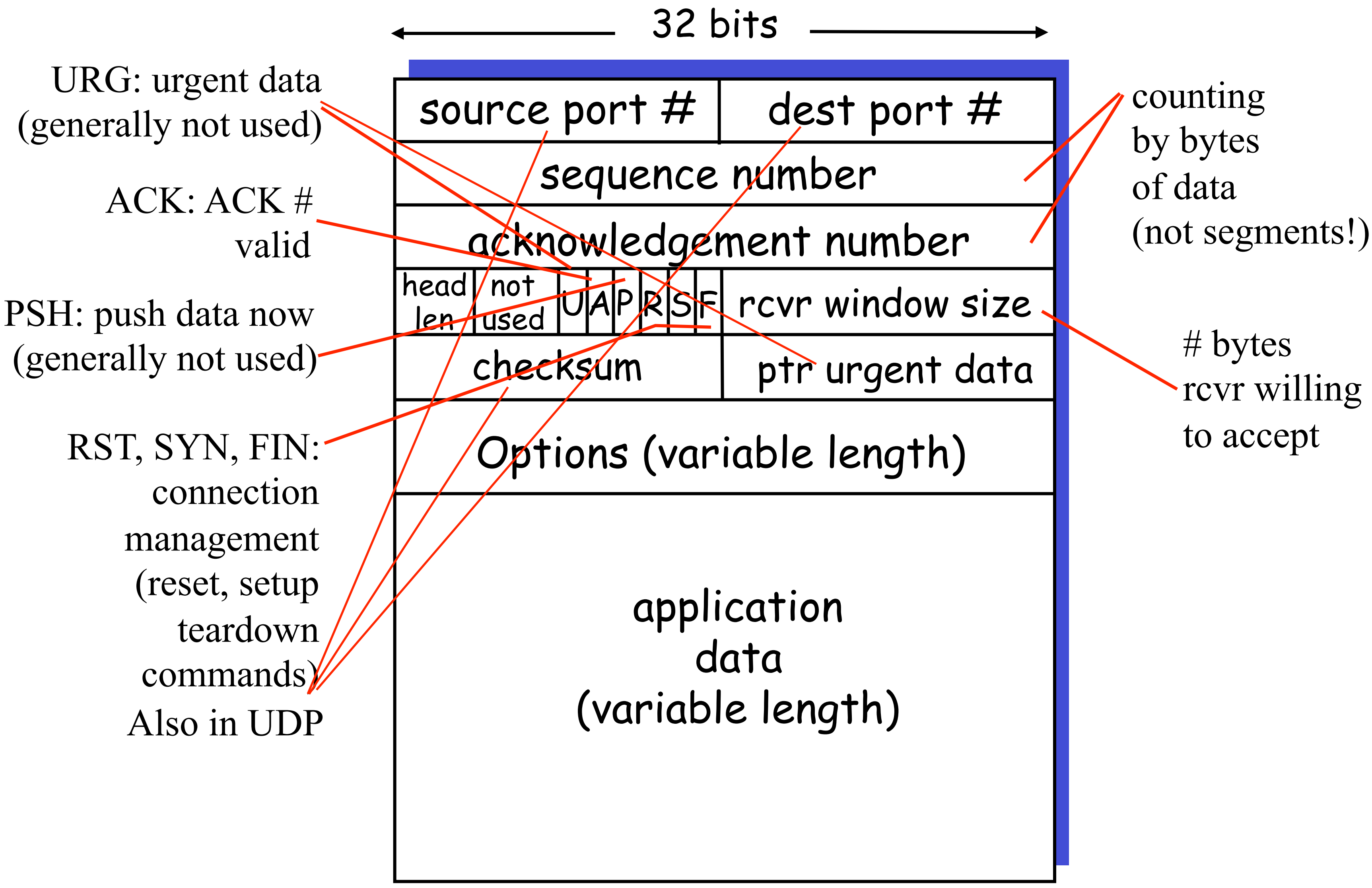
- Byte stream "number" of first byte in segment's data

ACKs:

- *Seq #* of next byte expected from other side
- Cumulative ACK



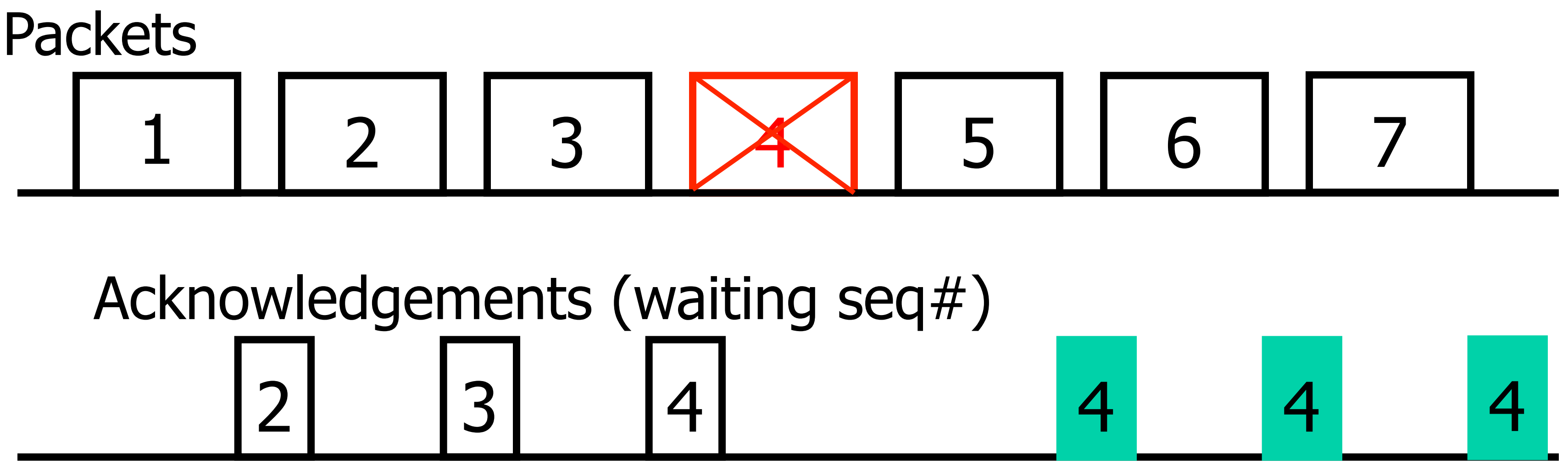
TCP Segment Structure



Fast Retransmit

- Timeout 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:
  - 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

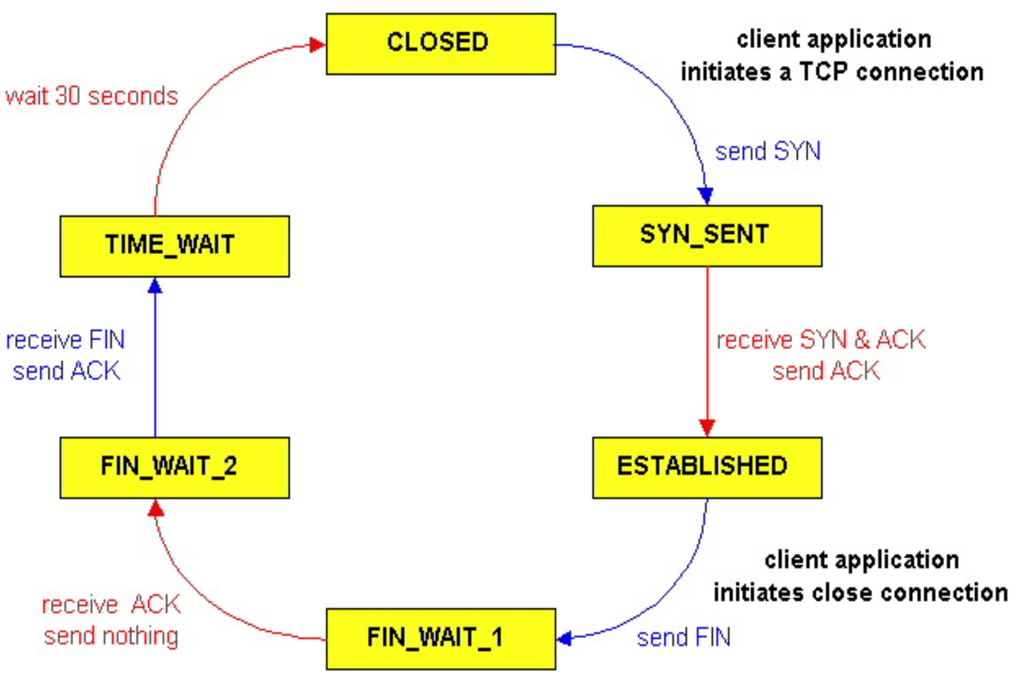
```
sendbase = initial_sequence number agreed by TWH
nextseqnum = initial_sequence number by TWH
loop (forever) {
  switch(event)
  event: data received from application above
         if (window allow send)
           create TCP segment with sequence number nextseqnum
           if (no timer) start 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 */
```

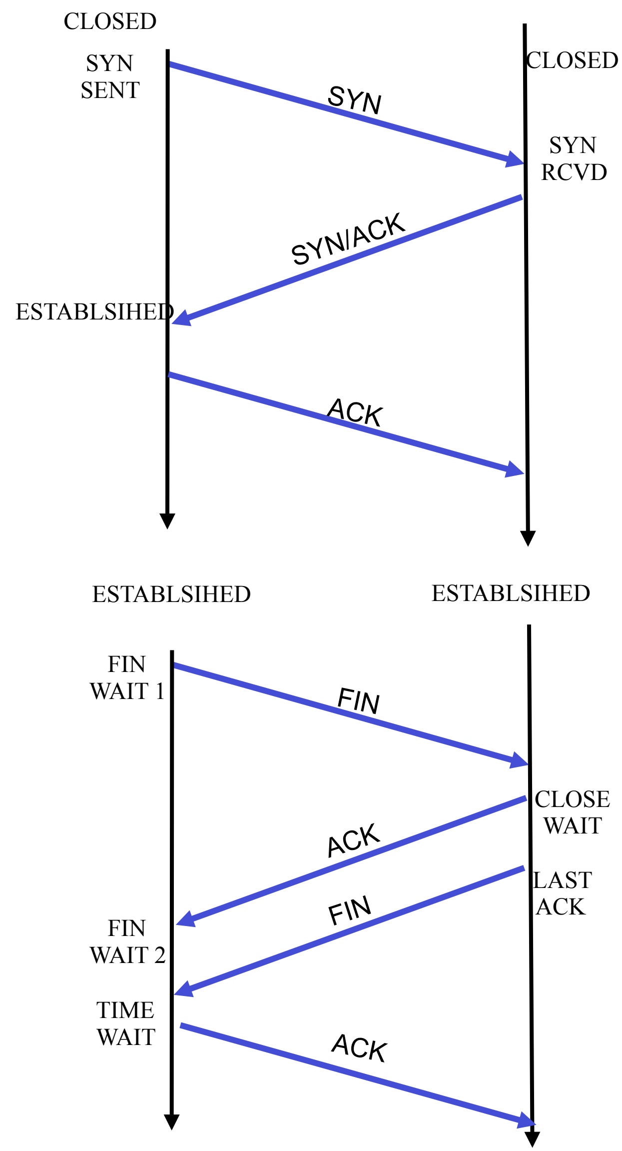
TCP Receiver ACK Generation [RFC 1122, RFC 2581]

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

TCP Connection Management

TCP lifecycle: init SYN/FIN

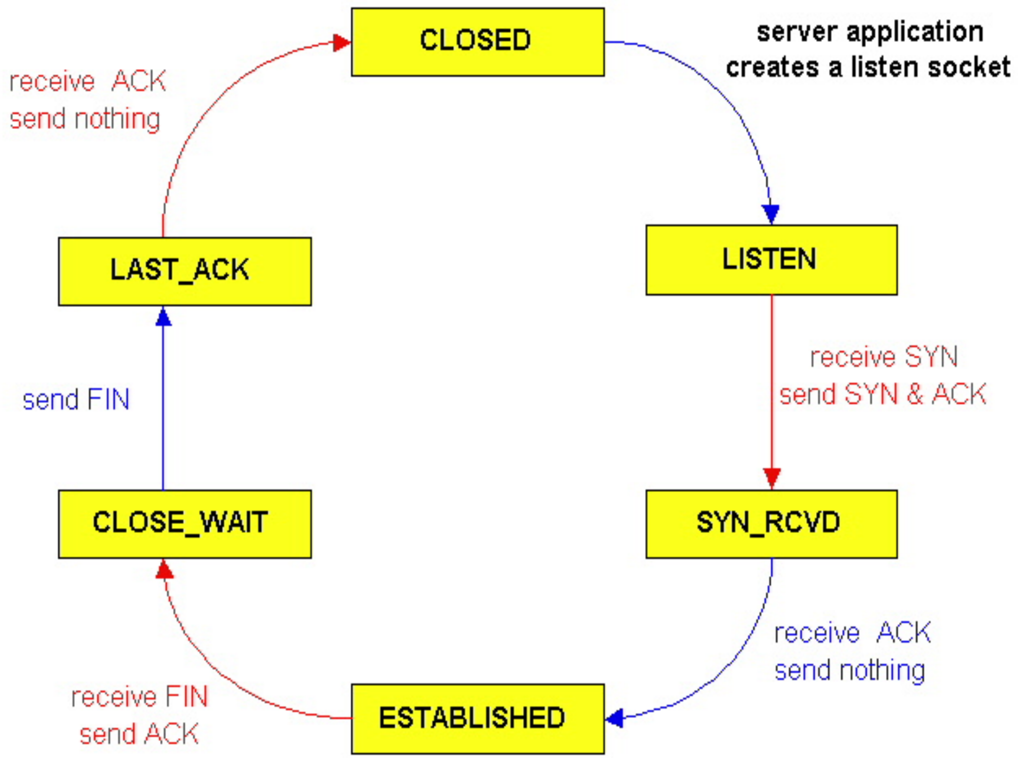


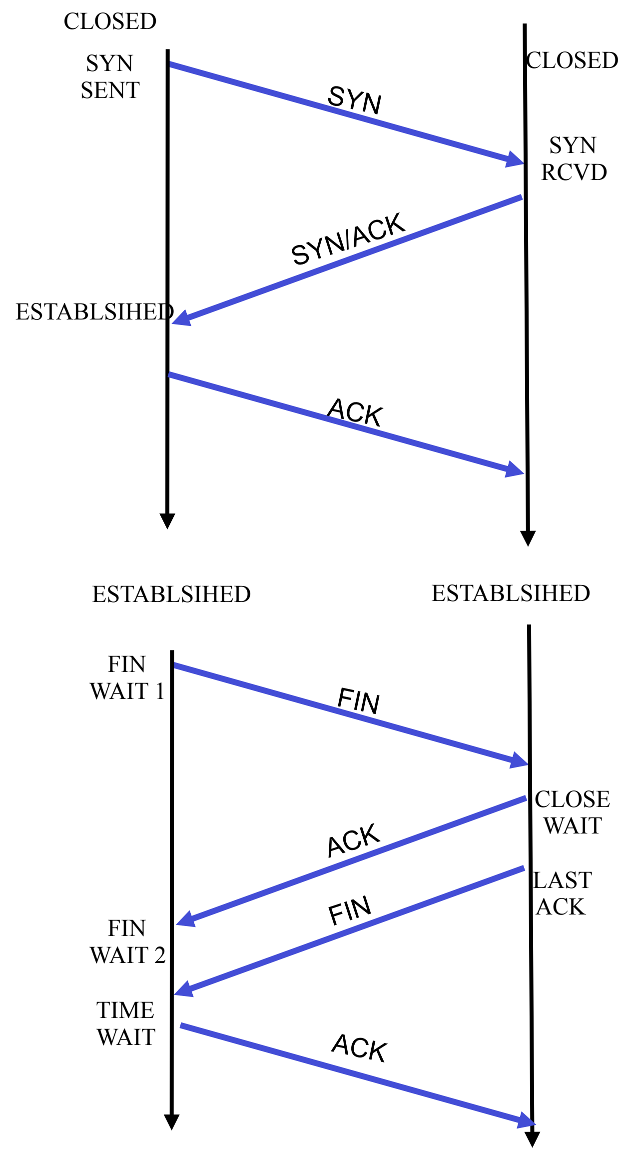


http://dsd.lbl.gov/TCP-tuning/ip-sysctl-2.6.txt

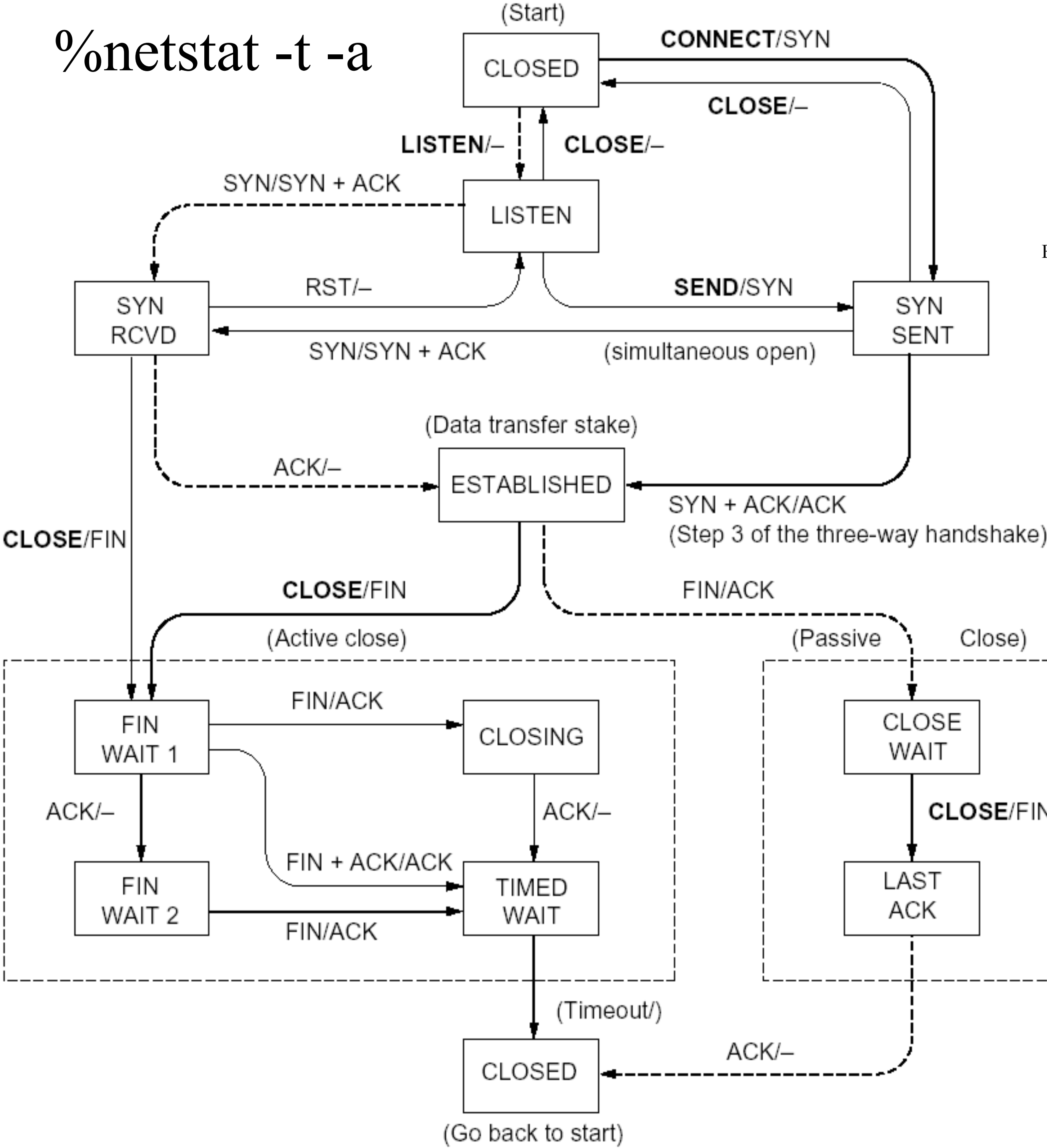
TCP Connection Management

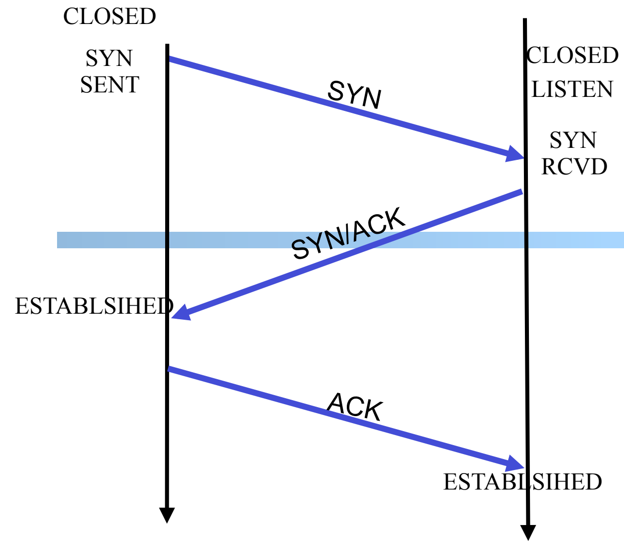
TCP lifecycle: wait for SYN/FIN

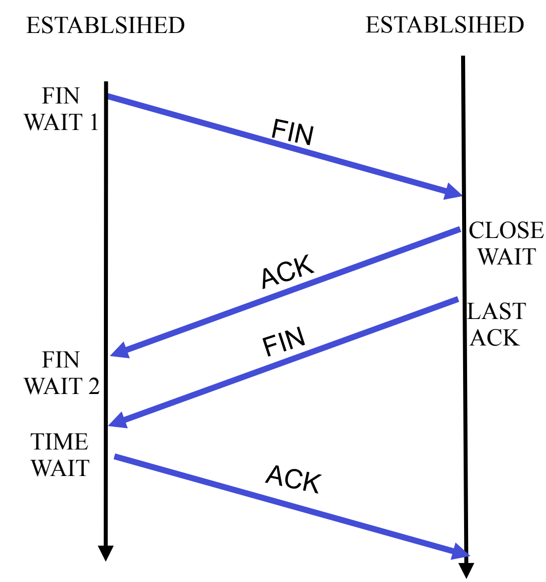




%netstat -t -a

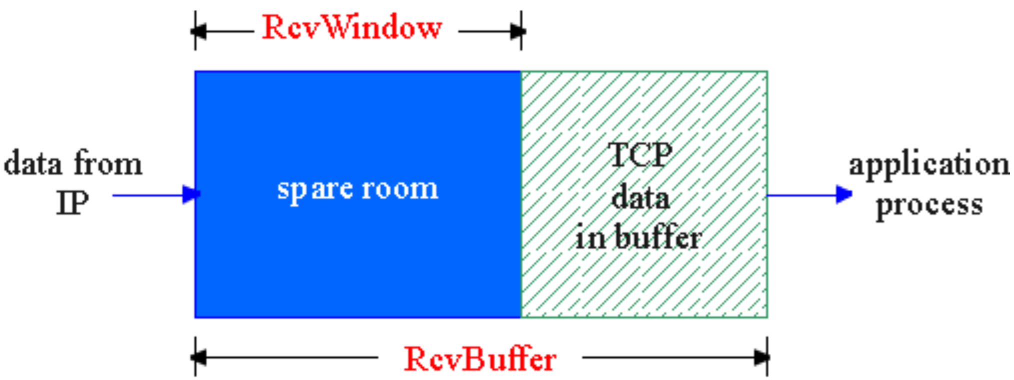

(Start)
CLOSED
CONNECT/SYN
CLOSE/–
LISTEN/–
CLOSE/–
SYN/SYN + ACK
LISTEN
RST/–
SEND/SYN
SYN RCVD
SYN SENT
SYN/SYN + ACK
(simultaneous open)
(Data transfer stake)
ACK/–
ESTABLISHED
SYN + ACK/ACK
(Step 3 of the three-way handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
(Active close)
(Passive
Close)
FIN WAIT 1
FIN/ACK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

Flow Control

- Receive side of a connection has a receive buffer:



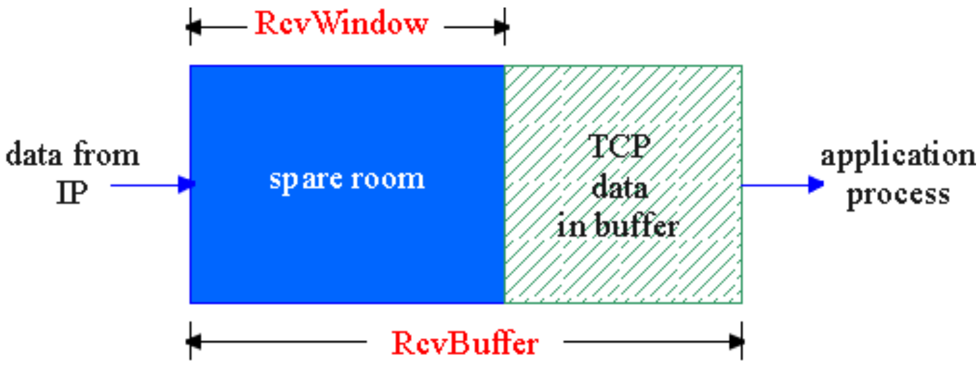
- App. process may be slow at reading from buffer

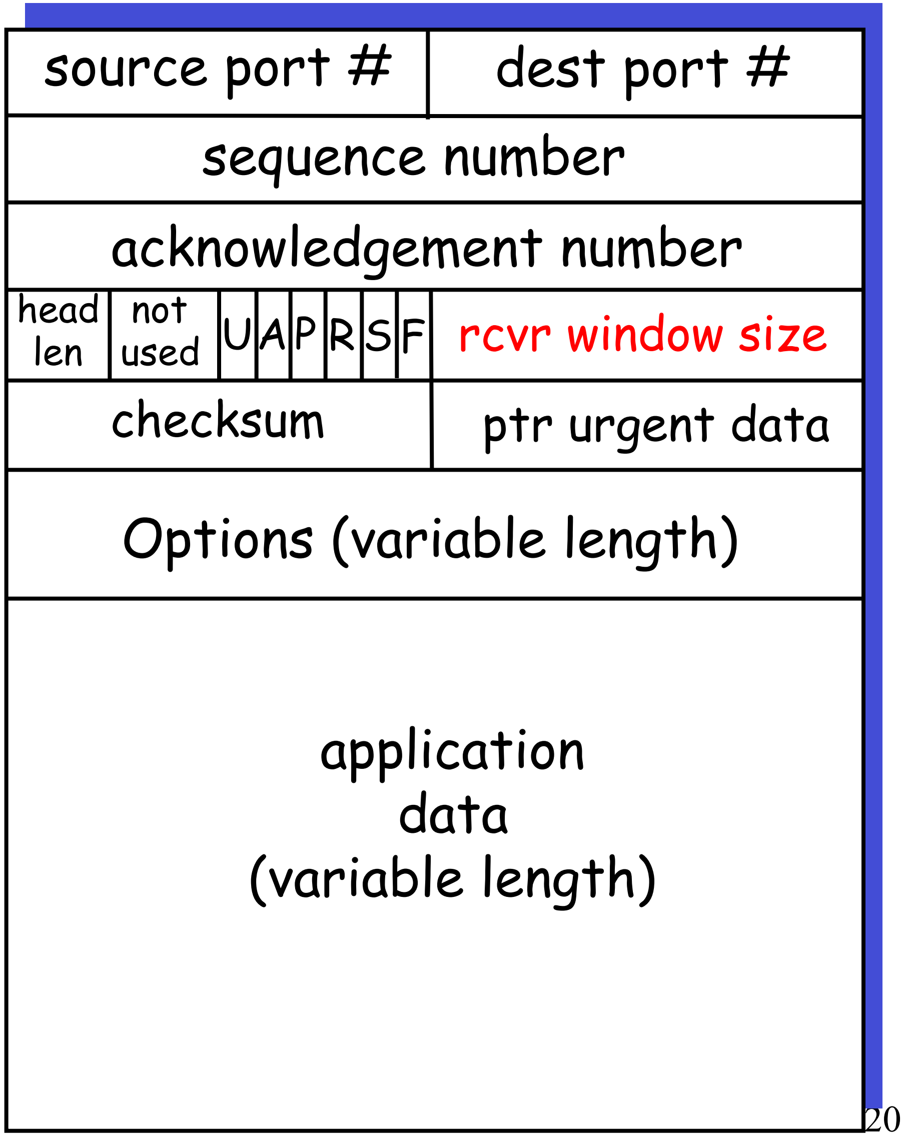
flow control

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





- spare room in buffer

= **RcvWindow**

A Summary of Questions

- How to improve the performance of rdt3.0?
  - Sliding window protocols
- What if there are duplication and reordering?
  - Network guarantee: max packet life time
  - Transport guarantee: not reuse a seq# before life time
  - Seq# management and connection management

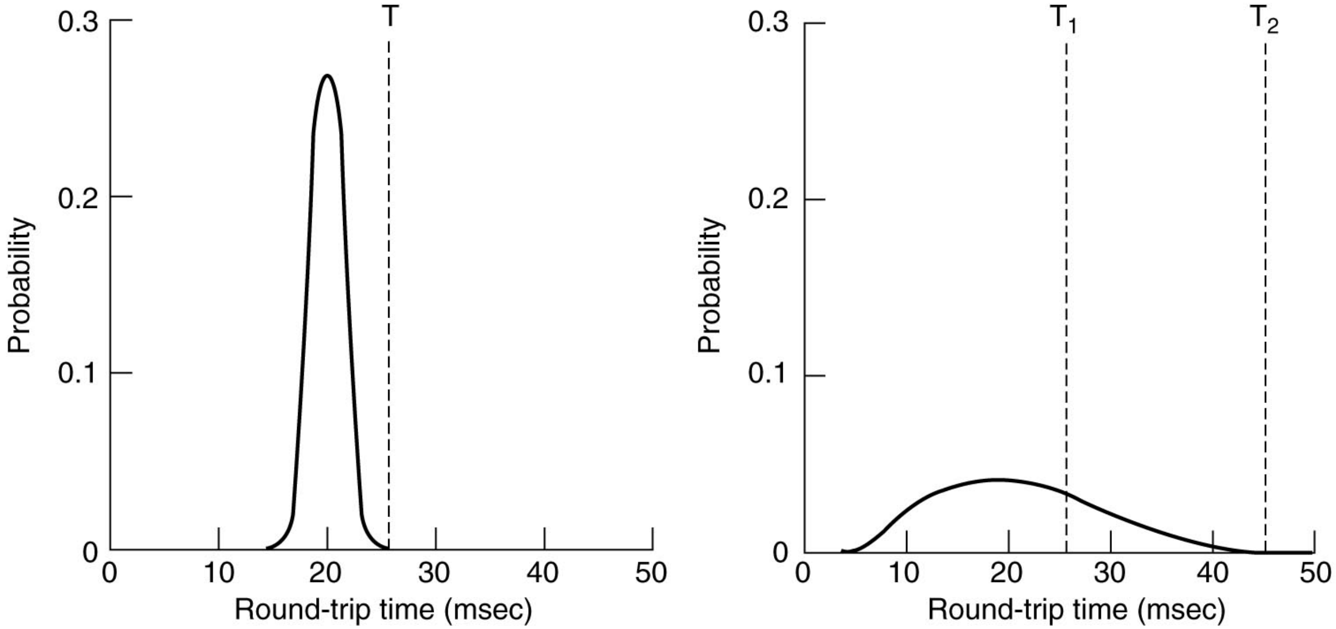
- How to determine the "right" parameters?

Timeout

Q: how to set timeout value?

- Too short: premature timeout
  - Unnecessary retransmissions; many duplicates
- Too long: slow reaction to segment loss

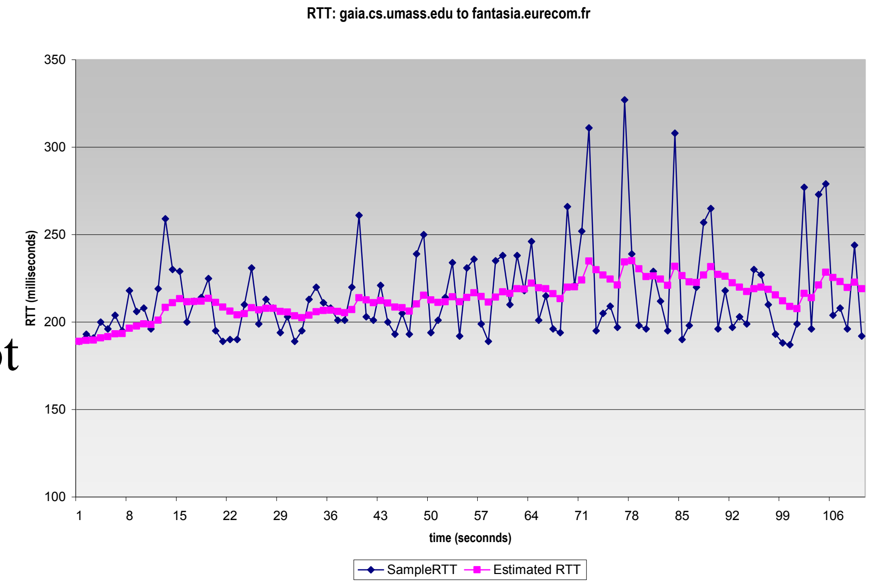
High-level Idea



Set timeout = average + safe margin

Estimating Round Trip Time

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



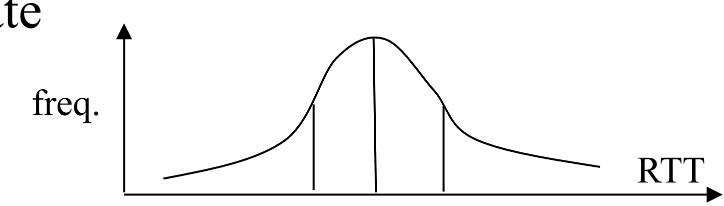

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: $\alpha = 0.125$

Setting Timeout

Problem:

- Using the average of **SampleRTT** will generate many timeouts due to network variations



Solution:

- **EstimtedRTT** plus "safety margin"
  - Large variation in **EstimatedRTT ->** larger safety margin

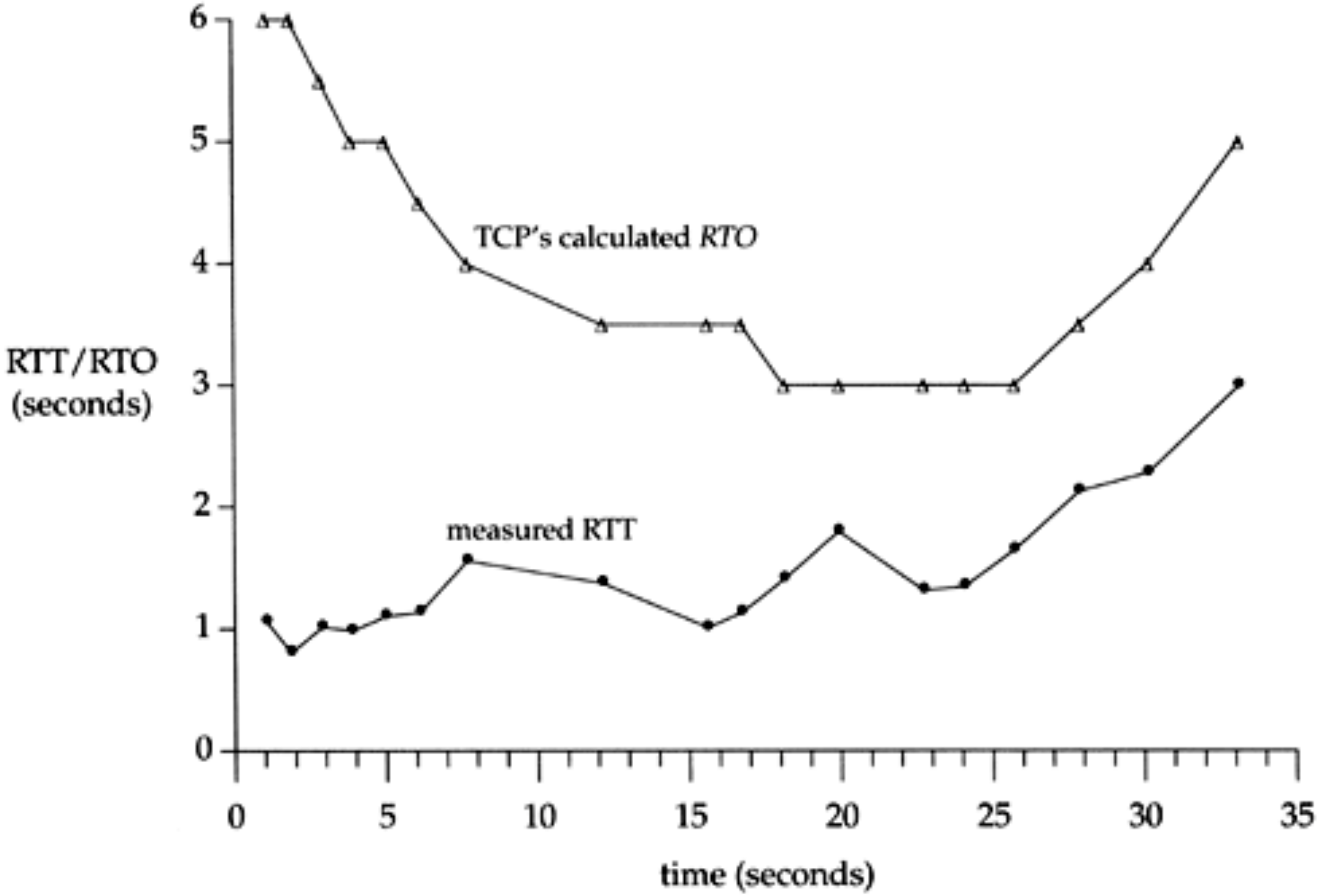
```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

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

An Example TCP Session



Outline

- Recap
- TCP reliability
- *Introduction to congestion control*

Principles of Congestion Control

Big picture:

- How to determine a flow's sending rate?

Congestion:

- Informally: "too many sources sending too much data too fast for the *network* to handle"
- Different from flow control !
- Manifestations:
  - Lost packets (buffer overflow at routers)
  - Wasted bandwidth
  - Long delays (queueing in router buffers)
- A top-10 problem !

History

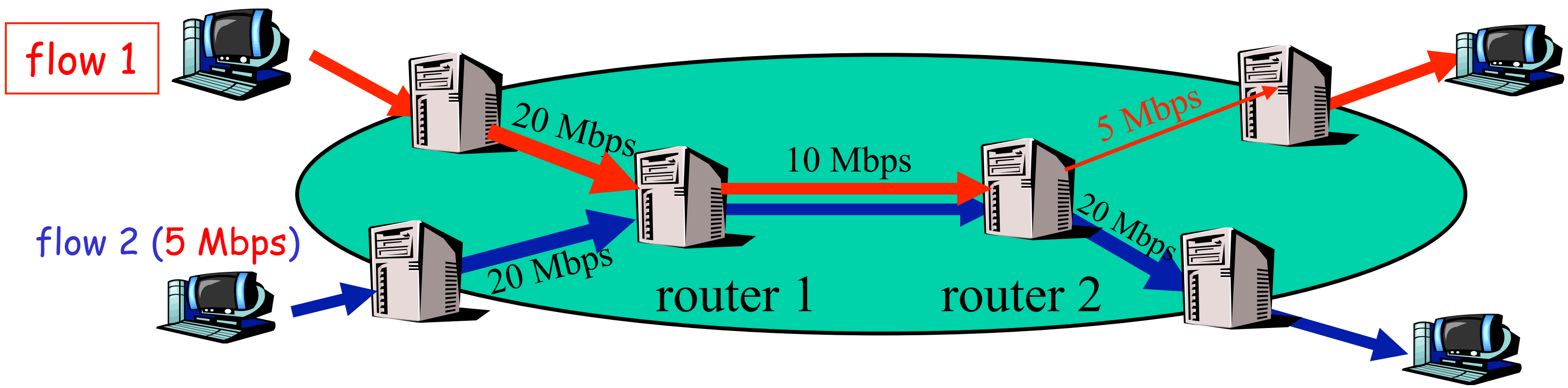
- TCP congestion control in mid-1980s
  - Fixed window size W
  - Timeout value = 2 RTT

- Congestion collapse in the mid-1980s
  - UCB ←→ LBL throughput dropped by 1000X !

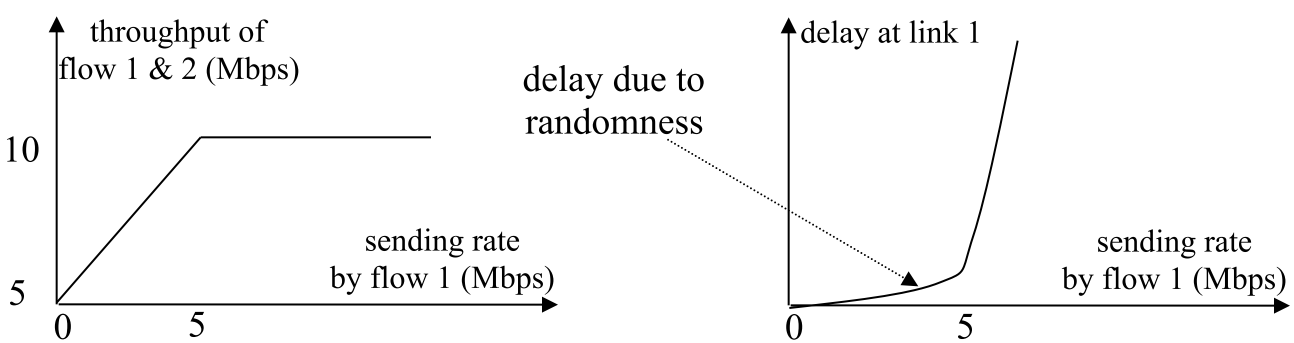
Some General Questions

- How can congestion happen?
- What is congestion control?
- Why is congestion control difficult?

Cause/Cost of Congestion: Single Bottleneck

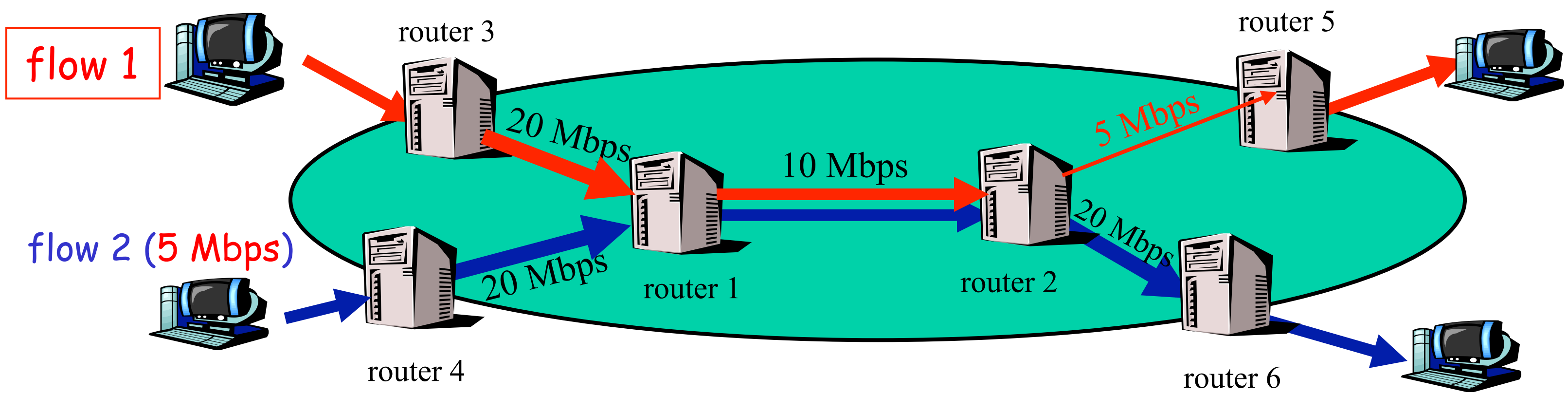


- Flow 2 has a fixed sending rate of 5 Mbps
- We vary the sending rate of flow 1 from 0 to 20 Mbps
- Assume
    - no retransmission
    - the link from router 1 to router 2 has infinite buffer
    - throughput: e2e packets delivered in unit time



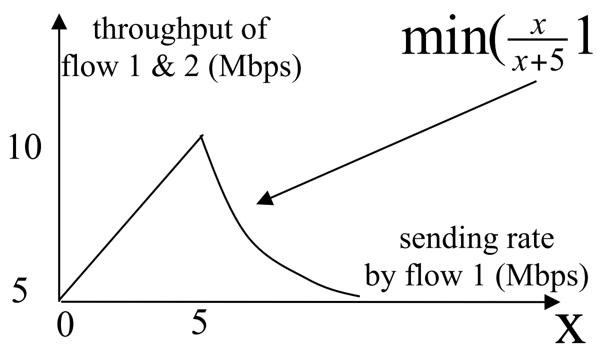
- large delays when congested

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

$$\min\left(\frac{x}{x+5}10, 5\right) + \frac{5}{x+5}10$$



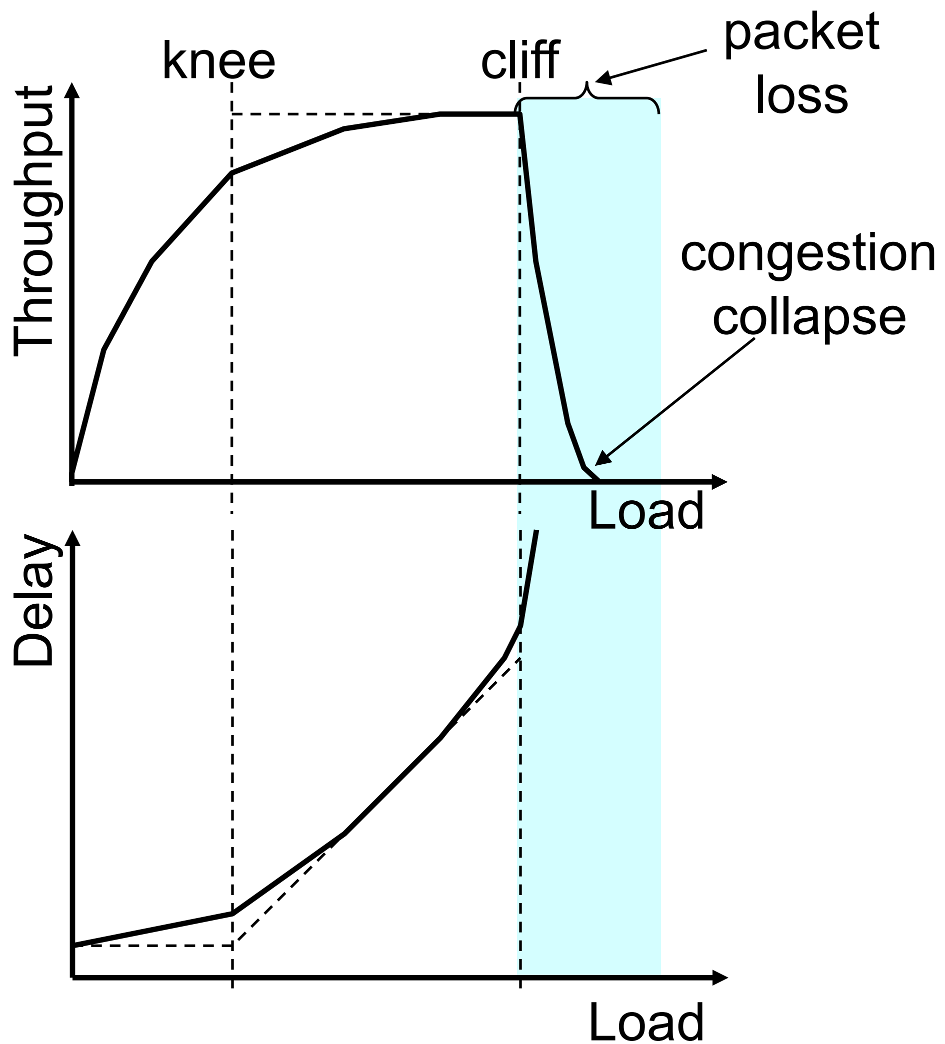
- when packet dropped at the link from router 2 to router 5, the upstream transmission from router 1 to router 2 used for that packet was wasted!

What if retransmission?

Summary: The Cost of Congestion

- 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

- Recap
- *Transport congestion control*
  - What is congestion
  - *Congestion control*

Implicit vs. Explicit

Implicit:

- Congestion inferred by end systems through observed loss, delay

Explicit:

- Routers provide feedback to end systems
  - Explicit rate sender should send at
  - Single bit indicating congestion (SNA, DECbit, TCP ECN, ATM)

Rate-based vs. Window-based

Rate-based:

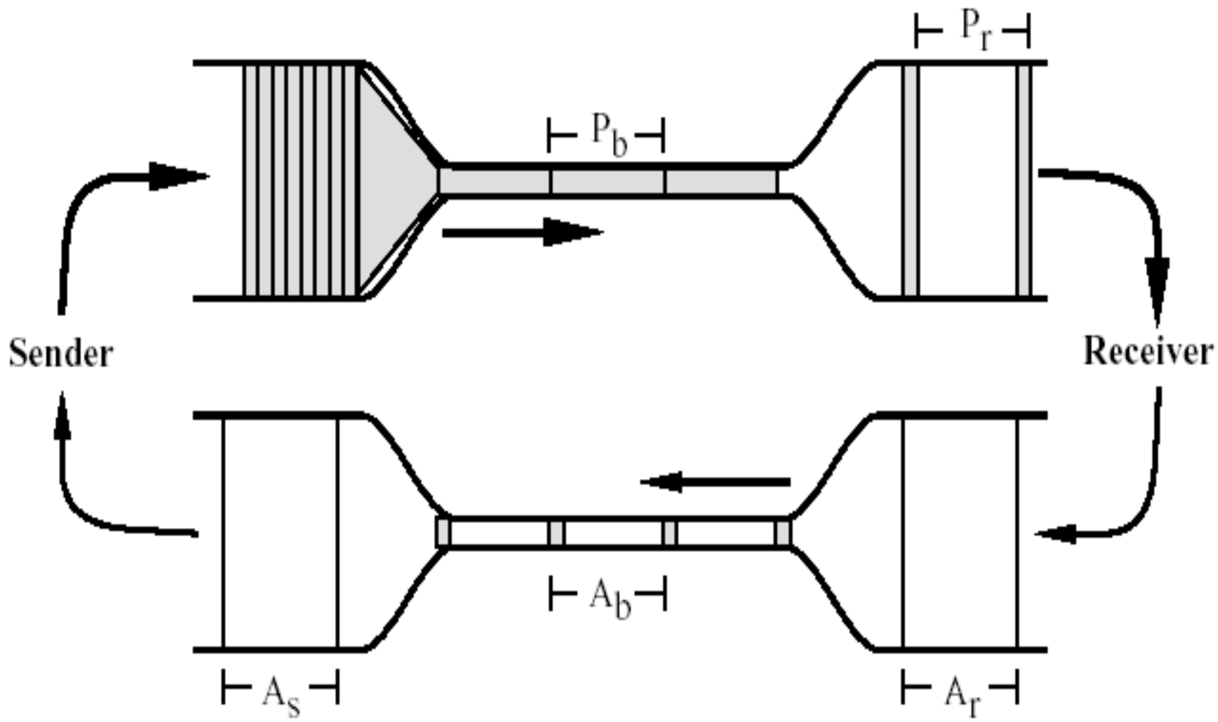
- Congestion control by explicitly controlling the sending rate of a flow, e.g. set sending rate to 128Kbps
- Example: ATM

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

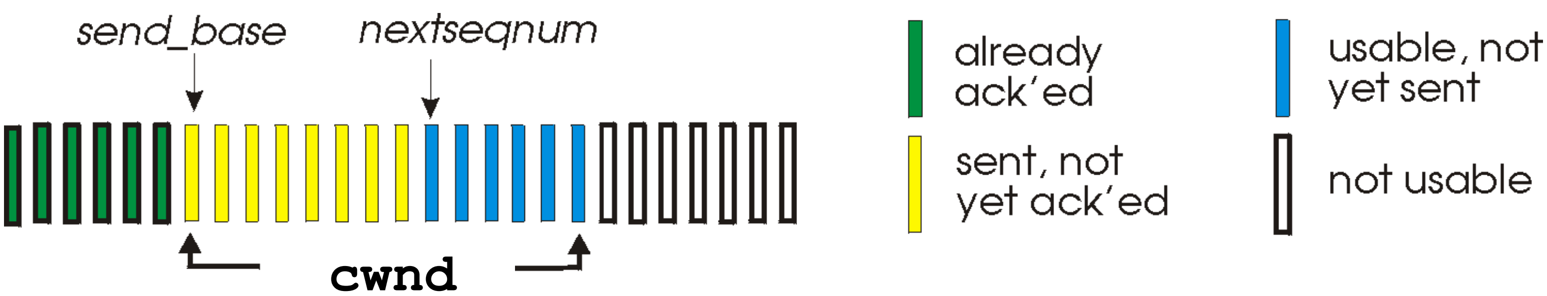
Window-based Congestion Control



- Window-based congestion control is self-clocking: considers flow conservation, and adjusts to RTT variation automatically

Sliding Window Congestion Control

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



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

$$\boxed{\text{throughput} \approx \frac{W * MSS}{RTT} \text{ Bytes/sec}}$$

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

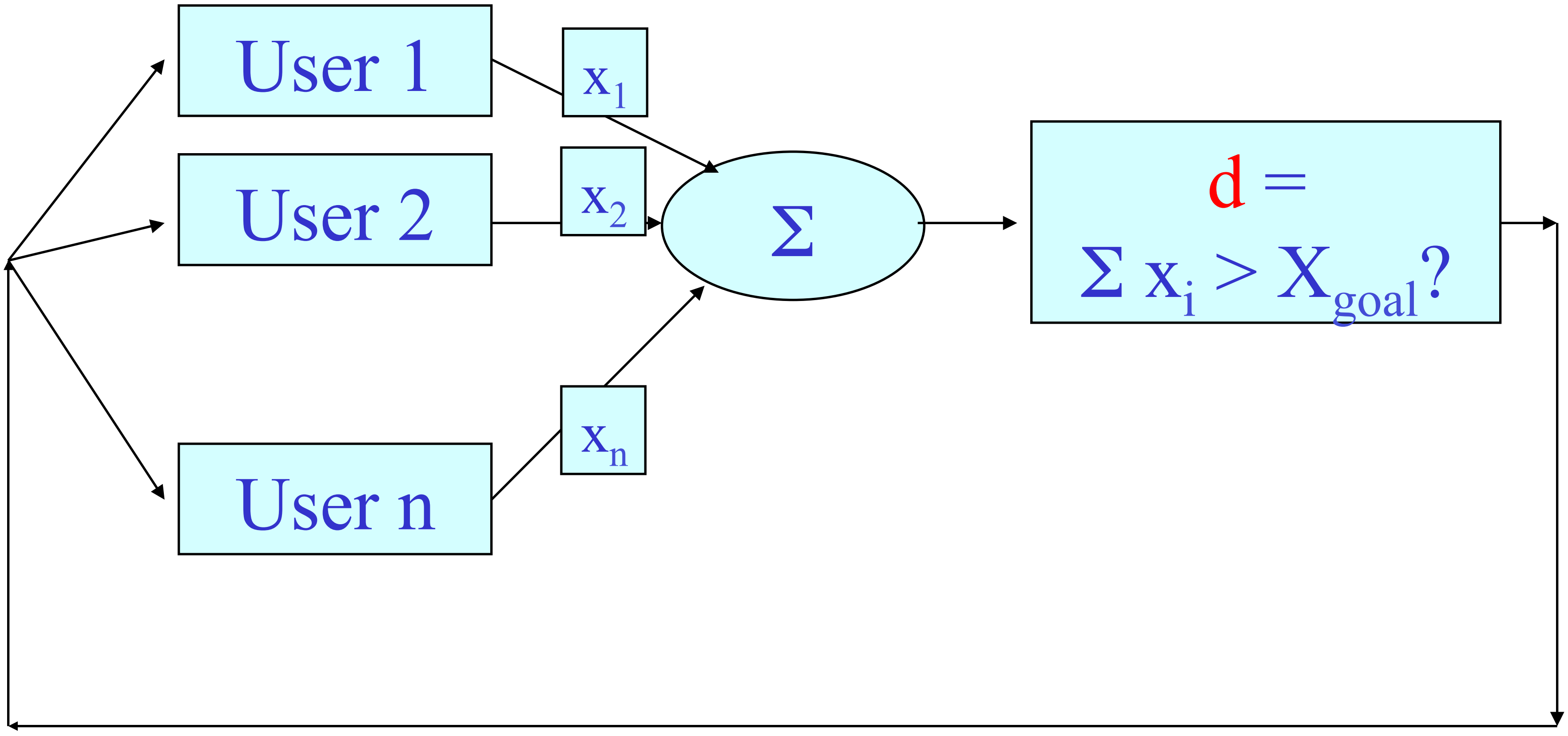
The Desired Properties of a Congestion Control Scheme

- Efficiency: close to full utilization but low delay
  - Fast convergence after disturbance, low oscillations

- Fairness (resource sharing)

- Distributedness (no central knowledge for scalability)

A Simple Model



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

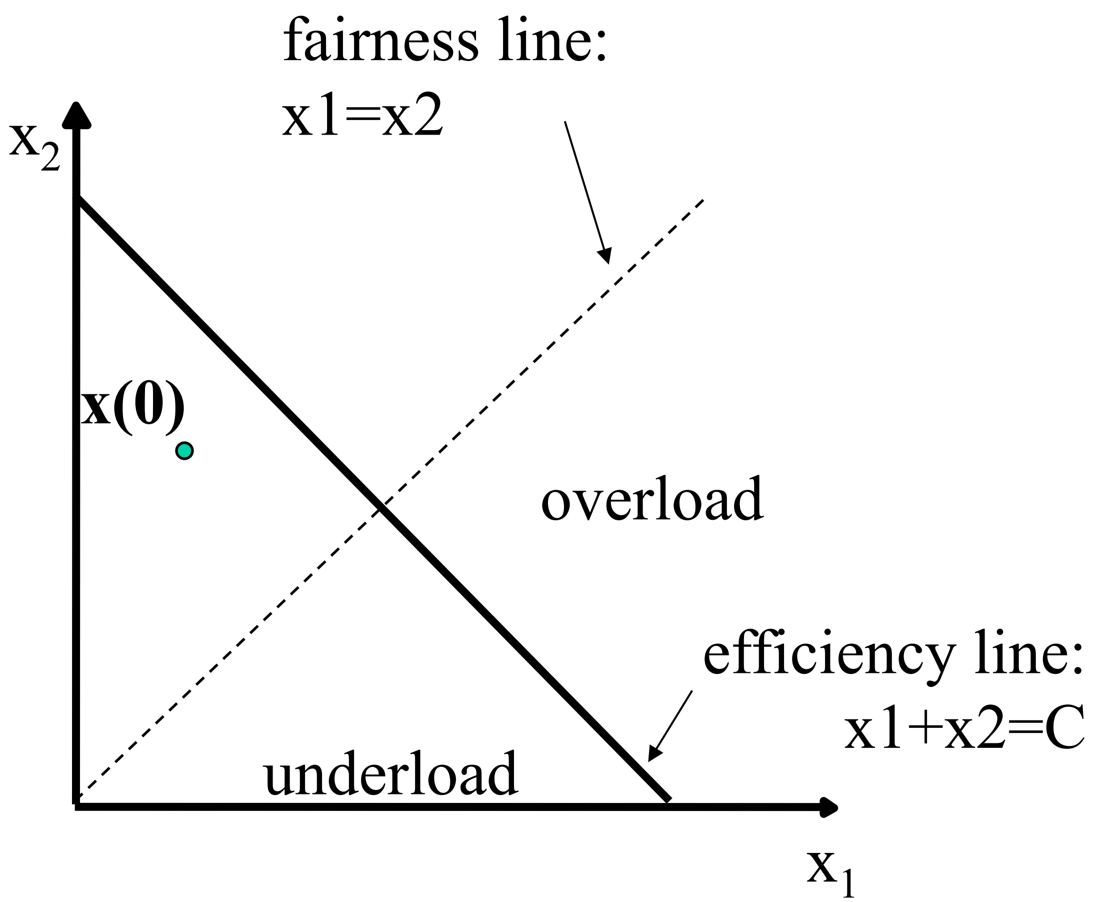
Linear Control

- Proposed by Chiu and Jain (1988)
- The simplest control strategy

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

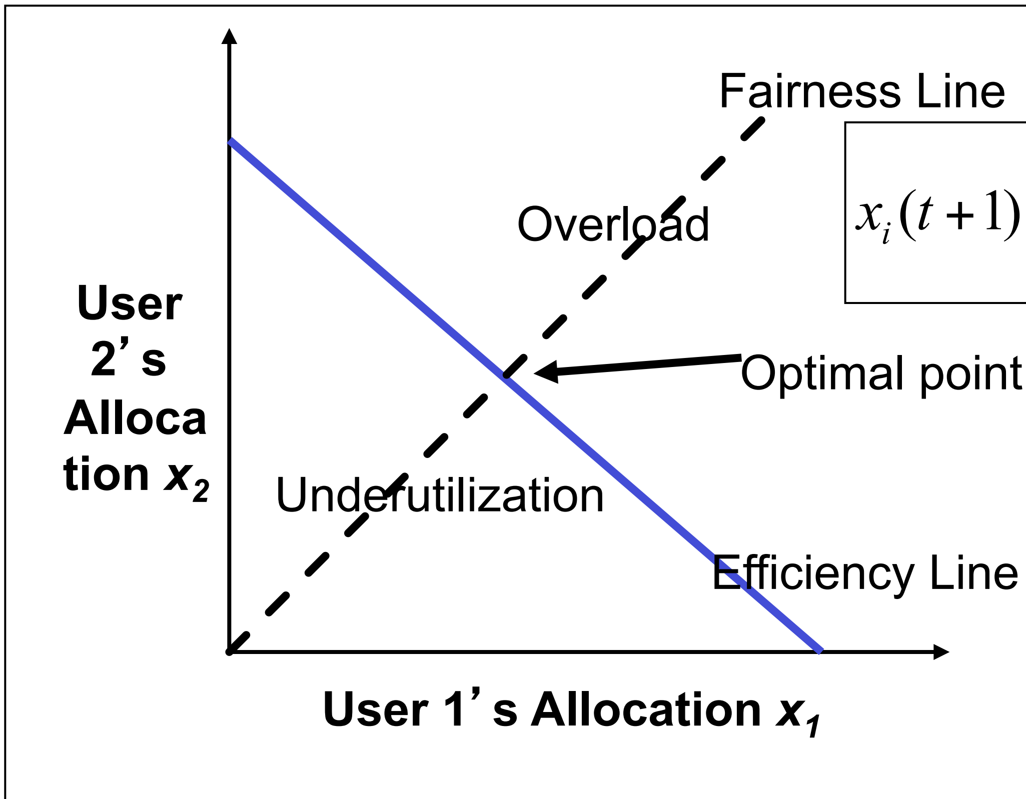
Discussion: values of the parameters?

State Space of Two Flows



State Space of Two Flows

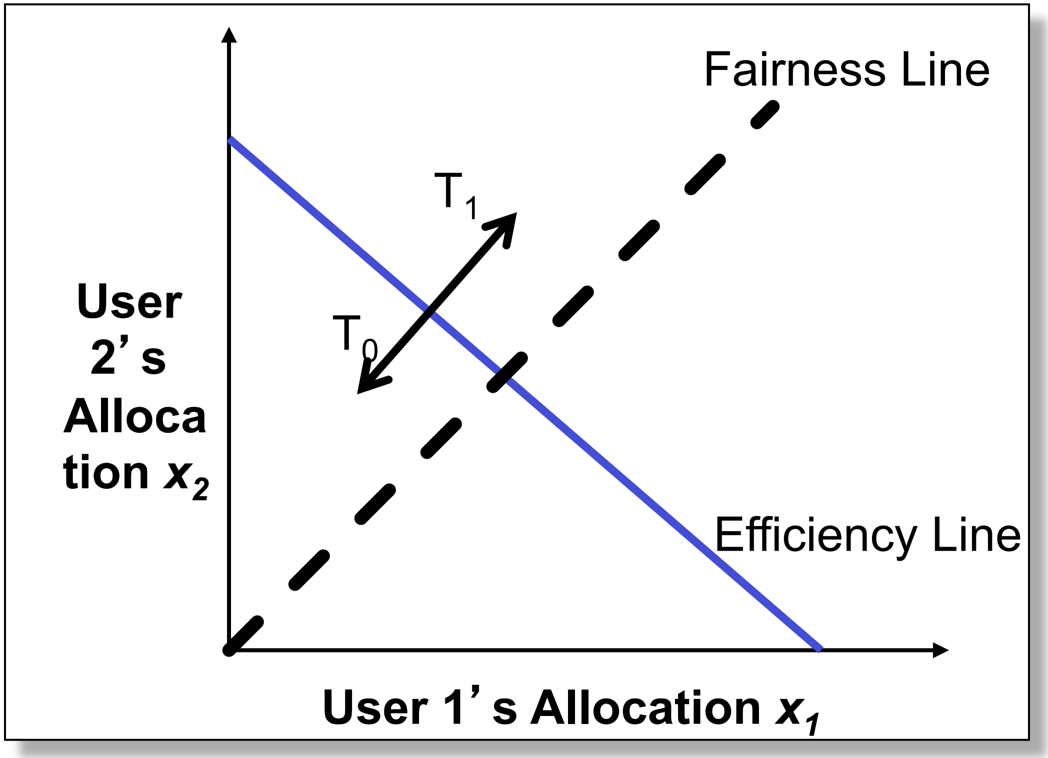
- What are desirable properties?
- What if flows are not equal?



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

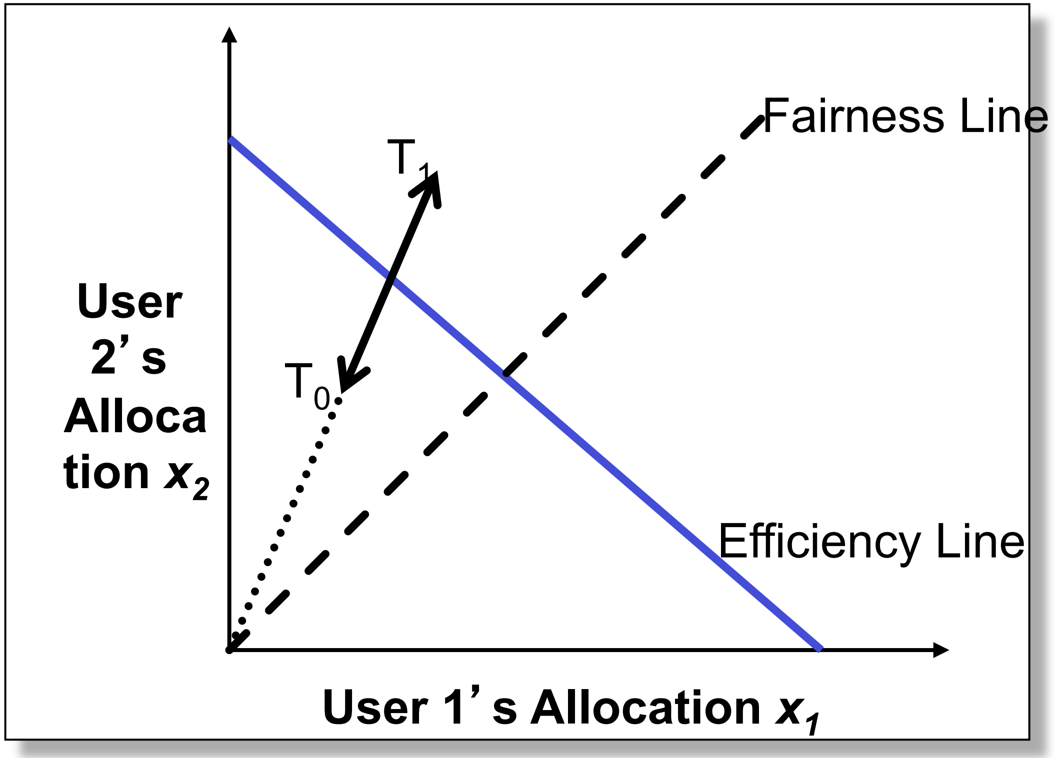
Additive Increase/Decrease

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



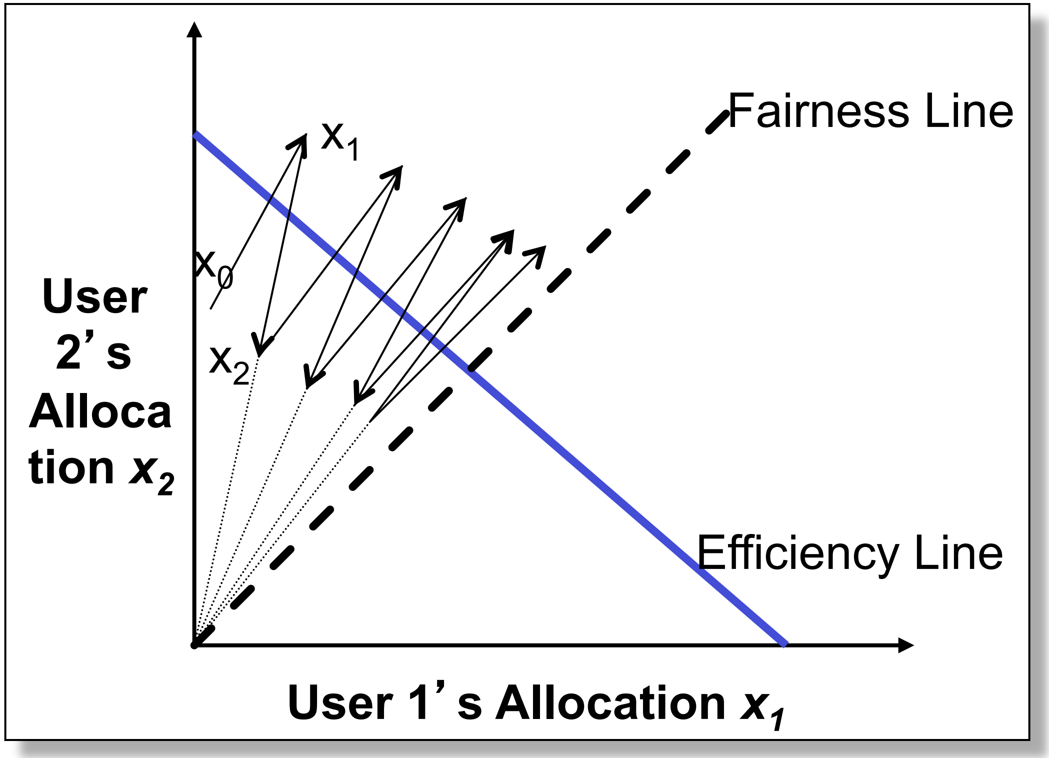
Multiplicative Increase/ Decrease

- Both X1 and X2 increase by the same factor over time
  - Extension from origin – constant fairness



What is the Right Choice?

- Constraints limit us to AIMD
  - Can have multiplicative term in increase (MAIMD)
  - AIMD moves towards optimal point



TCP and linear controls

- Upon congestion:
  - $w(t+1) = a*w(t) \qquad 0 < a < 1$
- While probing
  - $w(t+1) = w(t) + b \qquad 0 < b \ll wmax$
- TCP sets a = 1/2, b = 1 (packet)

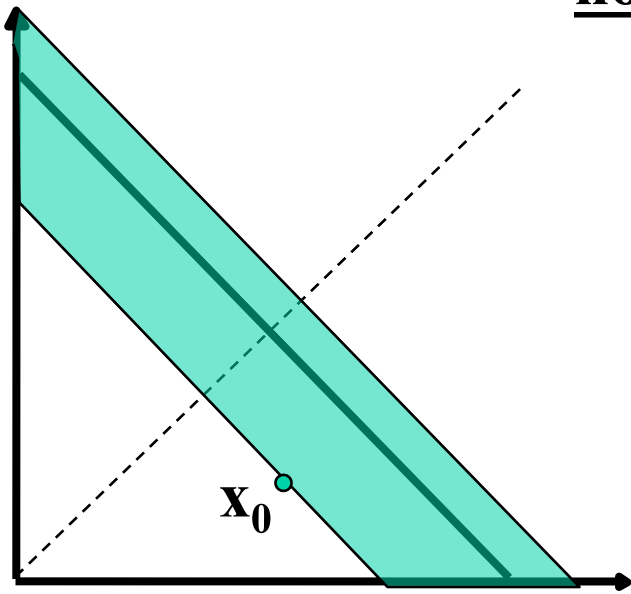
Implication: Congestion (overload) Case

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

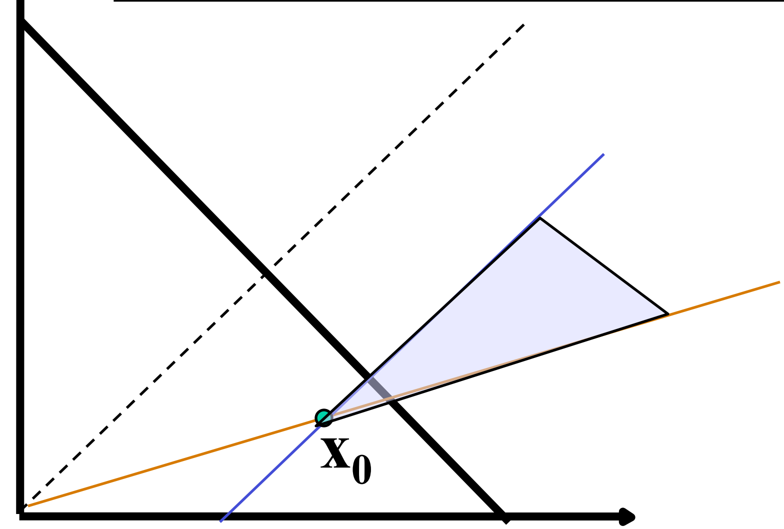
$$x_i(t+1) = \begin{cases} a_I + b_I x_i(t) & \text{if } \mathrm{d(t) = no\ cong.} \\ b_D x_i(t) & \text{if } \mathrm{d(t) = cong.} \end{cases}$$

no-congestion

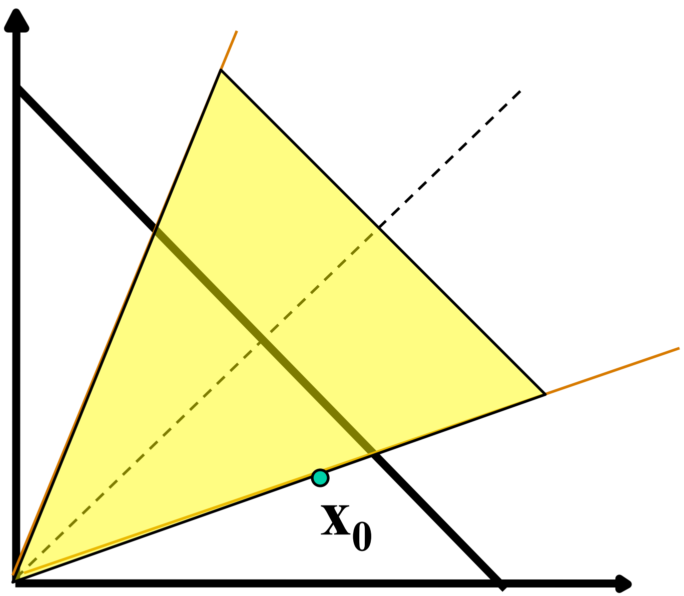
$$x_i(t+1) = \begin{cases} a_I + b_I x_i(t) & \text{if } d(t) = \text{no cong.} \\ a_D + b_D x_i(t) & \text{if } d(t) = \text{cong.} \end{cases}$$



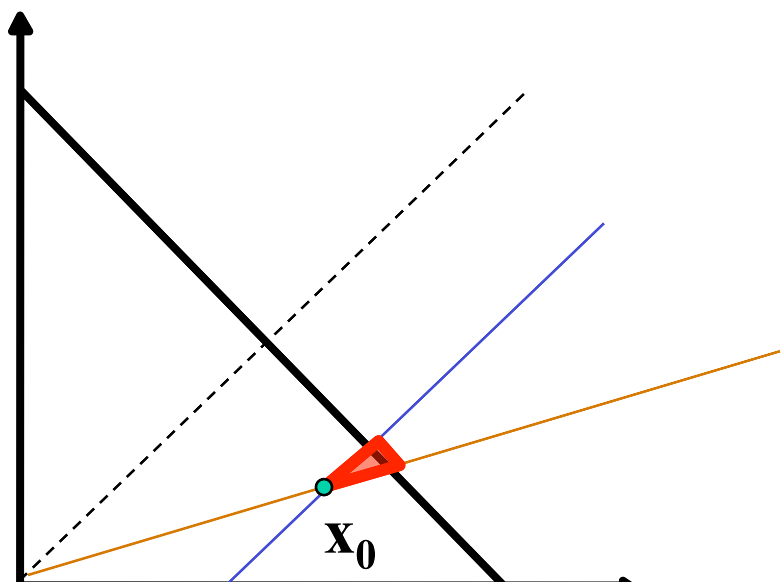
efficiency



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.,
  - $a_I > 0$, $b_I > 1$

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

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

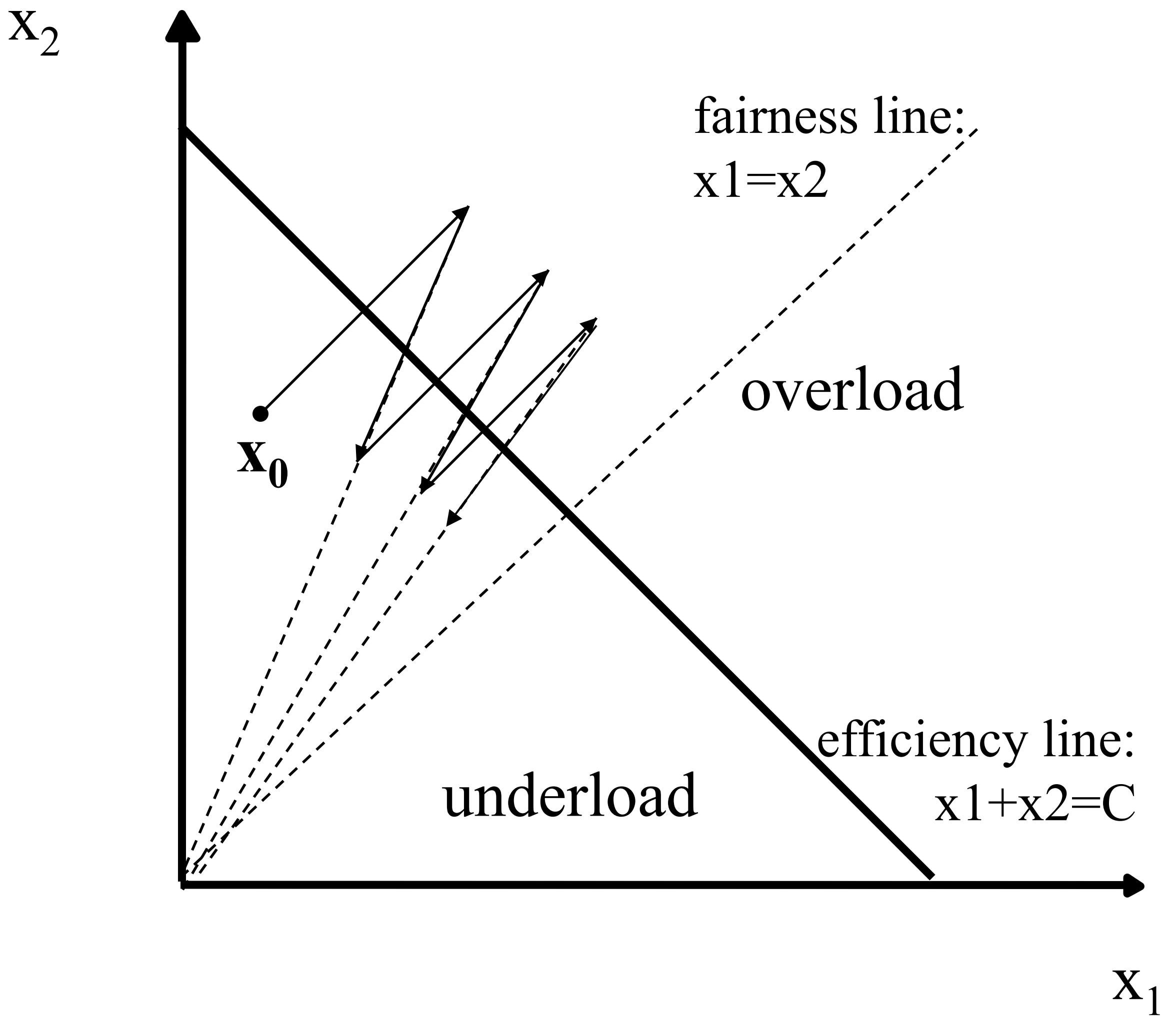
Four Special Cases

	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



Another Look

- Consider the difference or ratio of the rates of two flows
  - AIAD: difference does not change
  - MIAD: ratio does not change
  - MIMD: difference becomes bigger
  - AIMD: difference does not change