Project 2 – Implement Reliable Transport

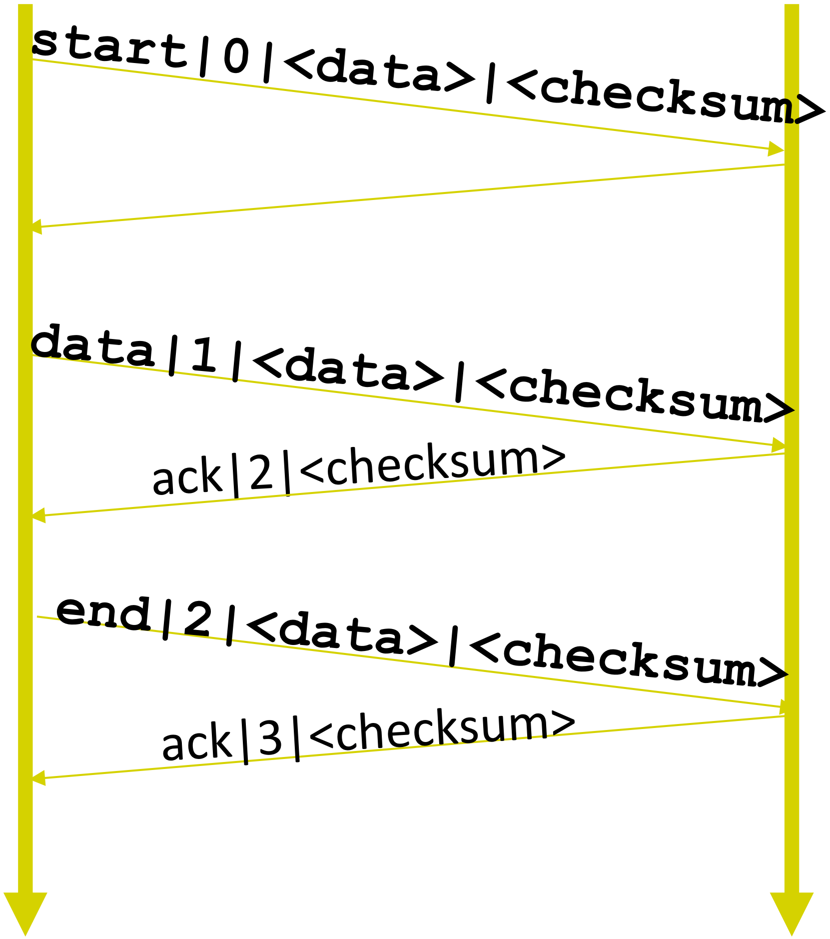
- Bears – TP: A simple reliable transport protocol based on GBN
  - Receiver code is provided
  - Only implement sender

- Basic requirements (85%) , deal with:
  - Loss, corruption and reordering
  - Duplication and delay

- Performance requirements (15%):
  - Fast retransmit
  - Selective acknowledgement

Protocol

- Packet types:
  - Start, data, ack, end, and sack
- Sliding window size: 5 packets
- Receiver returns cumulative acknowledgement



Sender

- The sender should be able to send a file to the receiver

  python Sender.py -f <input file>

- Implement a Go Back N based sender
- It should have a 500ms retransmission timeout
- It **must not** produce any console output

Test and Grading

We provide TestHarness.py for testing

- and a similar version of TestHarness.py is used for grading

Tips:

- Start your project early
- You may start with “Stop-and-Wait”
- Write your own test cases

Logistics

- GSIs: Peter, Radhika and Akshay
- Additional OH for help with the project – will be announced on Piazza
- These slides, Spec and code online midnight, today
- Due Nov 2, at noon

TCP: Congestion Control

CS 168, Fall 2014

Sylvia Ratnasamy

http://inst.eecs.berkeley.edu/~cs168

Material thanks to Ion Stoica, Scott Shenker, Jennifer Rexford, Nick McKeown, and many other colleagues

Administrivia

- HW2 due at midnight (not noon) on Oct 16
- Project#2 due on Nov 2 (not Oct 27)
- Next lecture: midterm review
- Today's material (CC) on the midterm?
  - Very basic concepts not details (~ up to slide#26)

Last lecture

- Flow control: adjusting the sending rate to keep from overwhelming a slow *receiver*

Today

- Congestion control: adjusting the sending rate to keep from overloading the *network*

Statistical Multiplexing → Congestion

- If two packets arrive at a router at the same time
  - Router will transmit one and buffer/drop the other
- Internet traffic is bursty
- If many packets arrive close in time
  - the router cannot keep up → gets congested
  - causes packet delays and drops

A few design considerations

…If you were starting with TCP?

- How do we know the network is congested?
- Who takes care of congestion?
  - *network, end hosts, both, …*
- How do we handle of congestion?

TCP’s approach

- **End hosts** adjust sending rate
- Based on **implicit feedback** from network

- Not the only approach
  - A consequence of history rather than planning

Some History: TCP in the 1980s

- Sending rate only limited by flow control
  - Dropped packets → senders (repeatedly!) retransmit
- Led to “congestion collapse” in Oct. 1986
  - Throughput on the NSF network dropped from 32Kbits/s to 40bits/sec
- “Fixed” by Van Jacobson’s development of TCP’s congestion control (CC) algorithms

Van Jacobson

- Leader of the networking research group at LBL
- Many contributions to the early TCP/IP stack
  - Most notably congestion control
- Creator of many widely used network tools
  - Traceroute, tcpdump, pathchar, Berkeley Packet Filter
- Later Chief Scientist at Cisco, now Fellow at PARC

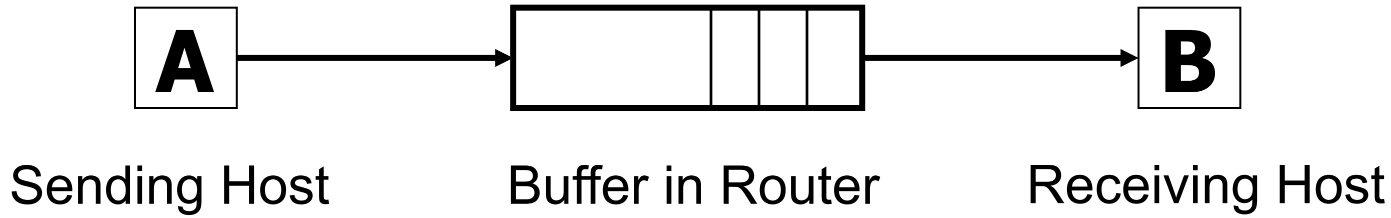
Jacobson's Approach

- Extend TCP's existing window-based protocol but adapt the window size in response to congestion
- A pragmatic and effective solution
  - required no upgrades to routers or applications!
  - patch of a few lines of code to TCP implementations
- Extensively researched and improved upon
  - Especially now with datacenters and cloud services

Three Issues to Consider

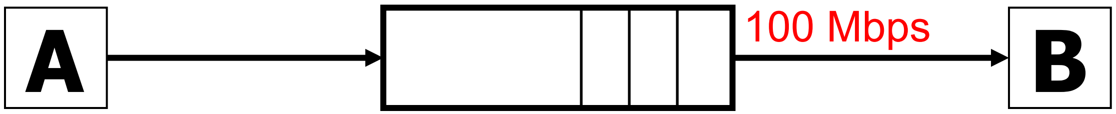
- Discovering the available (bottleneck) bandwidth
- Adjusting to variations in bandwidth
- Sharing bandwidth between flows

Abstract View



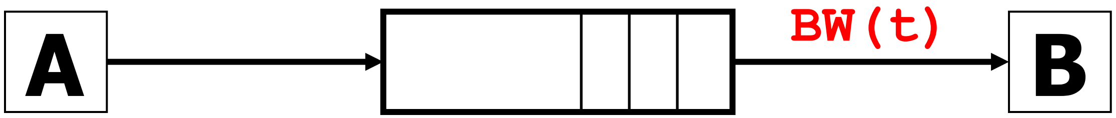
- Ignore internal structure of router and model it as a single queue for a particular input-output pair

Discovering available bandwidth



- Pick sending rate to match bottleneck bandwidth
  - Without any *a priori* knowledge
  - Could be gigabit link, could be a modem

Adjusting to variations in bandwidth

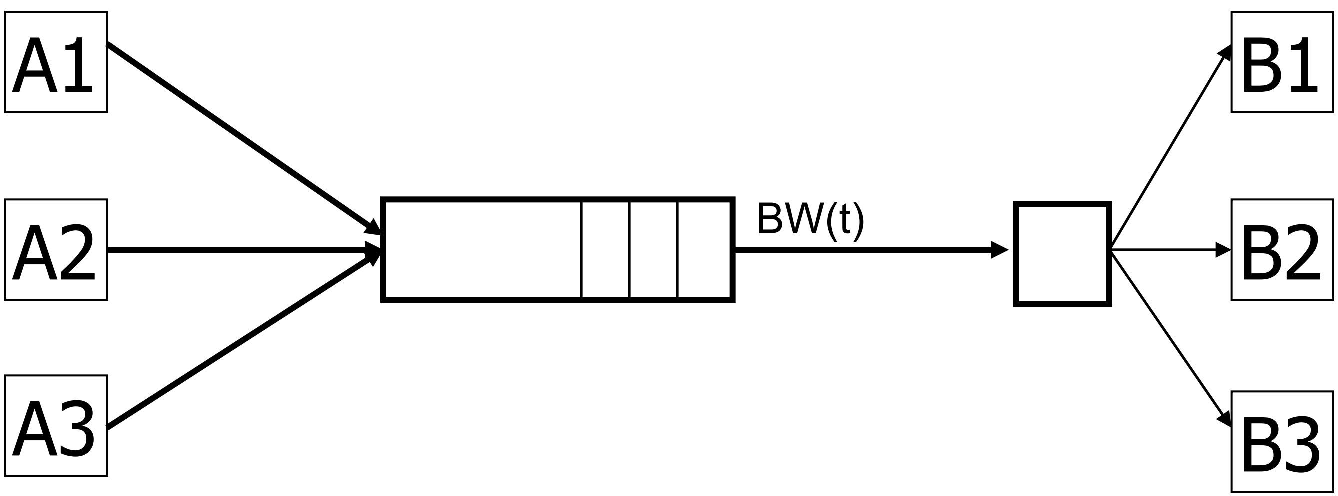


- Adjust rate to match instantaneous bandwidth
  - Assuming you have rough idea of bandwidth

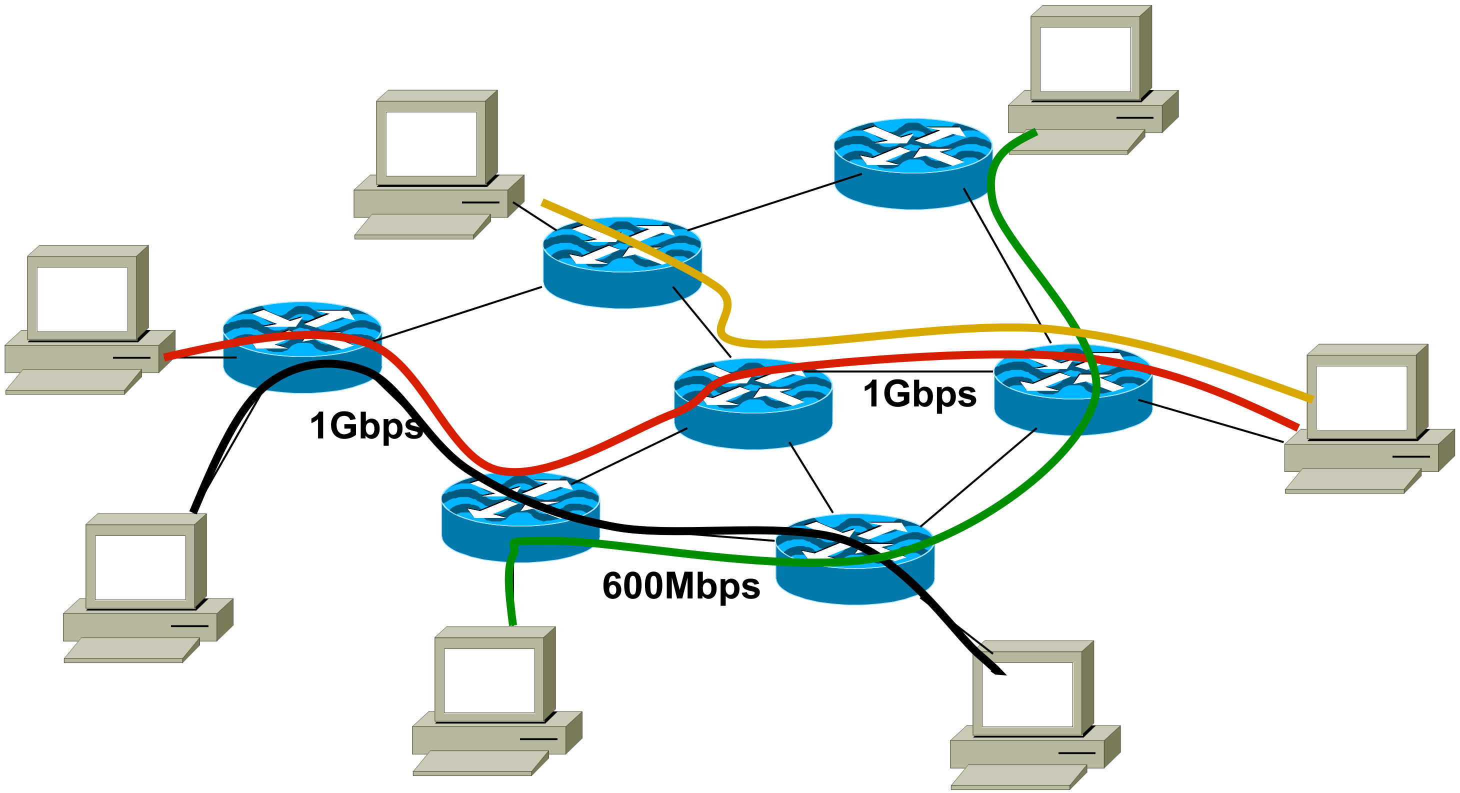
Multiple flows and sharing bandwidth

Two Issues:

- Adjust total sending rate to match bandwidth
- Allocation of bandwidth between flows



Reality



Congestion control is a resource allocation problem involving many flows, many links, and complicated global dynamics

Possible Approaches

(0) Send without care

- Many packet drops

Possible Approaches

(0) Send without care

(1) Reservations

- Pre-arrange bandwidth allocations
- Requires negotiation before sending packets
- Low utilization

Possible Approaches

(0) Send without care

(1) Reservations

(2) Pricing

- Don’t drop packets for the high-bidders
- Requires payment model

Possible Approaches

(0) Send without care

(1) Reservations

(2) Pricing

(3) Dynamic Adjustment

- Hosts infer level of congestion; adjust
- Network reports congestion level to hosts; hosts adjust
- Combinations of the above
- Simple to implement but suboptimal, messy dynamics

Possible Approaches

(0) Send without care

(1) Reservations

(2) Pricing

(3) Dynamic Adjustment

All three techniques have their place

- Generality of dynamic adjustment has proven powerful
- Doesn't presume business model, traffic characteristics, application requirements
- But does assume good citizenship!

TCP's Approach in a Nutshell

- TCP connection has window
  - Controls number of packets in flight
- Sending rate: ~Window/RTT
- Vary window size to control sending rate

All These Windows…

- Congestion Window: CWND
  - How many bytes can be sent without overflowing routers
  - Computed by the sender using congestion control algorithm
- Flow control window: AdvertisedWindow (RWND)
  - How many bytes can be sent without overflowing receiver's buffers
  - Determined by the receiver and reported to the sender
- Sender-side window = minimum{CWND, RWND}
  - Assume for this lecture that RWND >> CWND

Note

- This lecture will talk about CWND in units of MSS
  - (Recall MSS: Maximum Segment Size, the amount of payload data in a TCP packet)
  - This is only for pedagogical purposes
- Keep in mind that real implementations maintain CWND in bytes

Two Basic Questions

- How does the sender detect congestion?
- How does the sender adjust its sending rate?
  - To address three issues
    - Finding available bottleneck bandwidth
    - Adjusting to bandwidth variations
    - Sharing bandwidth

Detecting Congestion

- Packet delays
  - Tricky: noisy signal (delay often varies considerably)

- Routers tell endhosts when they're congested

- Packet loss
  - Fail-safe signal that TCP already has to detect
  - Complication: non-congestive loss (e.g., checksum errors)

Not All Losses the Same

- Duplicate ACKs: isolated loss
  - Still getting ACKs
- Timeout: much more serious
  - Not enough dupacks
  - Must have suffered several losses
- Will adjust rate differently for each case

Rate Adjustment

- Basic structure:
  - Upon receipt of ACK (of new data): increase rate
  - Upon detection of loss: decrease rate

- How we increase/decrease the rate depends on the phase of congestion control we're in:
  - Discovering available bottleneck bandwidth *vs.*
  - Adjusting to bandwidth variations

Bandwidth Discovery with Slow Start

- Goal: estimate available bandwidth
  - start slow (for safety)
  - but ramp up quickly (for efficiency)
- Consider
  - RTT = 100ms, MSS=1000bytes
  - Window size to fill 1Mbps of BW = 12.5 packets
  - Window size to fill 1Gbps = 12,500 packets
  - Either is possible!

"Slow Start" Phase

- Sender starts at a slow rate but increases **exponentially** until first loss
- Start with a small congestion window
  - Initially, CWND = 1
  - So, initial sending rate is MSS/RTT
- Double the CWND for each RTT with no loss

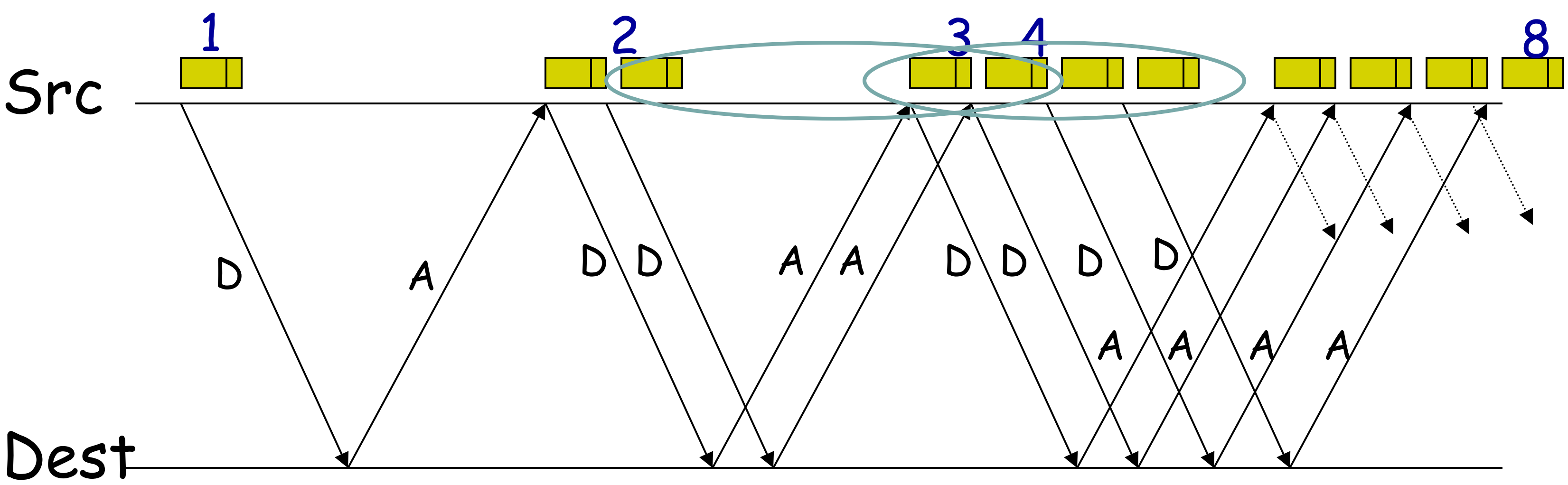
Slow Start in Action

- For each RTT: double CWND
- Simpler implementation: for each ACK, CWND += 1

Linear increase per <u>ACK</u> `(CWND+1)` → exponential increase per <u>RTT</u> `(2xCWND)`

Slow Start in Action

- For each RTT: double CWND
- Simpler implementation: for each ACK, CWND += 1



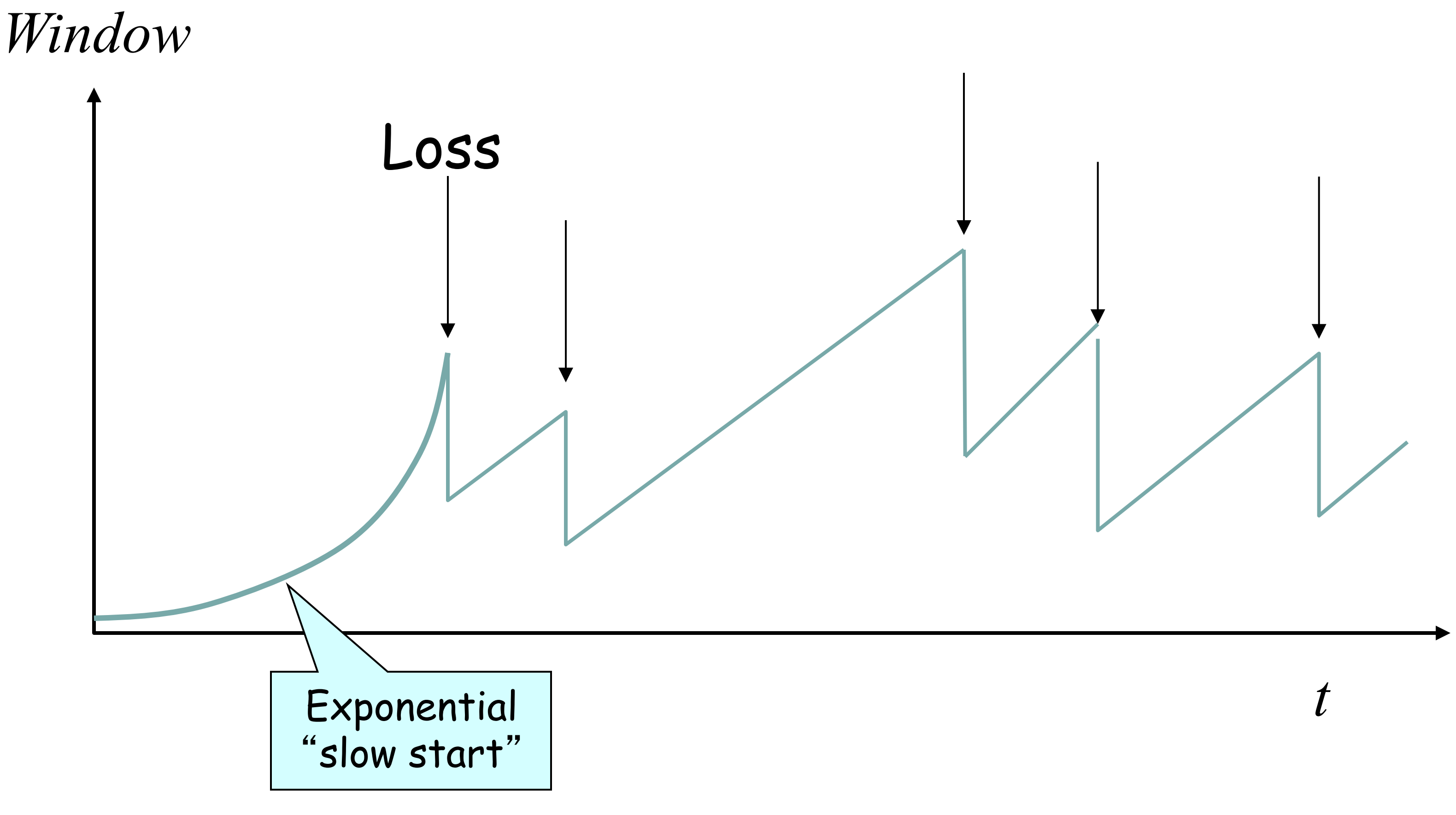
Adjusting to Varying Bandwidth

- Slow start gave an estimate of available bandwidth
- Now, want to track variations in this available bandwidth, oscillating around its current value
  - Repeated probing (rate increase) and backoff (decrease)
- TCP uses: “Additive Increase Multiplicative Decrease” (AIMD)
  - We’ll see why shortly…

AIMD

- Additive increase
  - Window grows by one MSS for every RTT with no loss
  - For each successful RTT, CWND = CWND + 1
  - Simple implementation:
    - for each ACK, CWND = CWND+ 1/CWND
- Multiplicative decrease
  - On loss of packet, divide congestion window in **half**
  - On loss, CWND = CWND/2

Leads to the TCP "Sawtooth"



Slow-Start vs. AIMD

- When does a sender stop Slow-Start and start Additive Increase?
- Introduce a “slow start threshold” (ssthresh)
  - Initialized to a large value
  - On timeout, ssthresh = CWND/2
- When CWND = ssthresh, sender switches from slow-start to AIMD-style increase

Why AIMD?

Recall: Three Issues

- Discovering the available (bottleneck) bandwidth
  - Slow Start
- Adjusting to variations in bandwidth
  - AIMD
- Sharing bandwidth between flows

Goals for bandwidth sharing

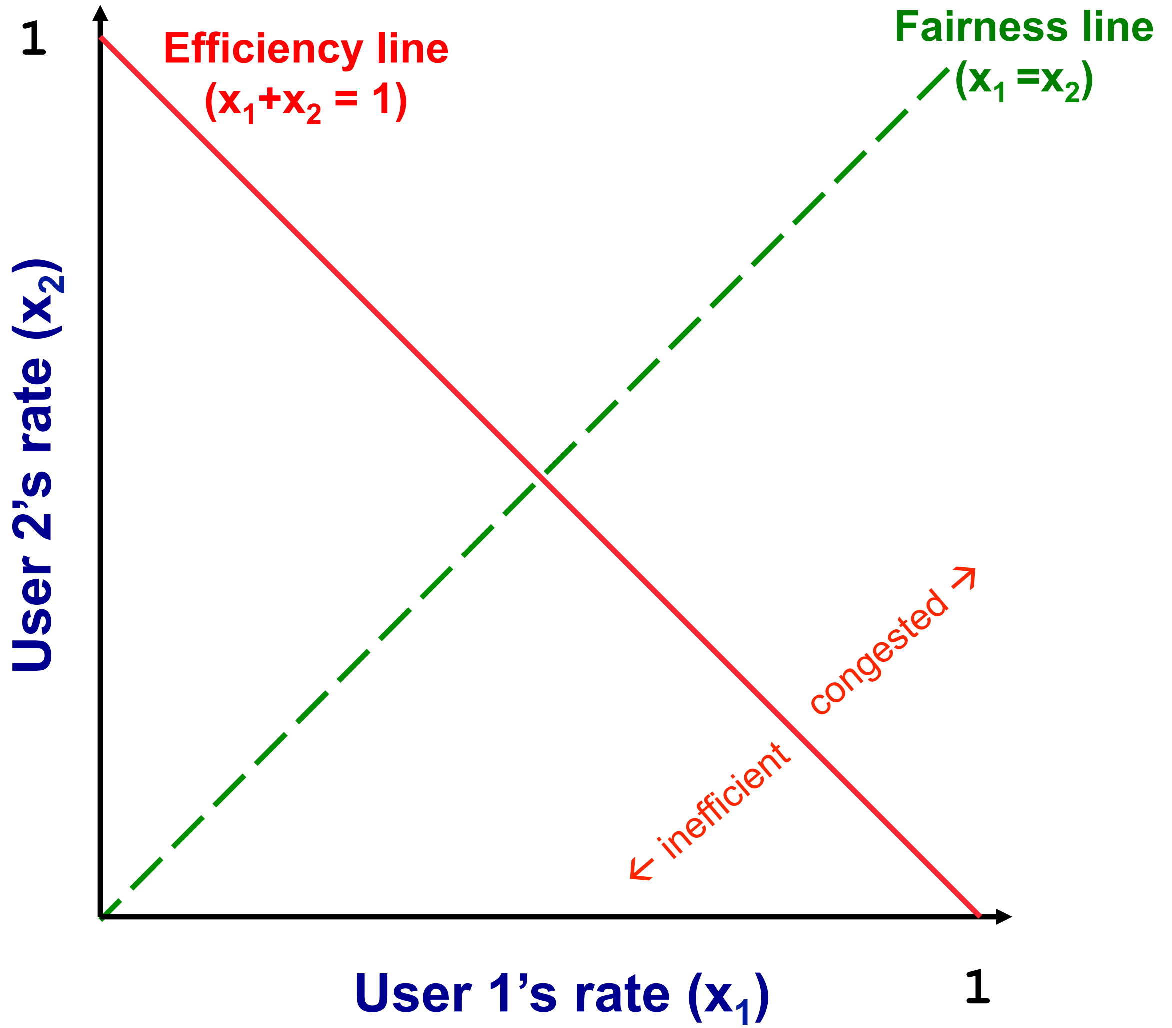
- Efficiency: High utilization of link bandwidth
- Fairness: Each flow gets equal share

Why AIMD?

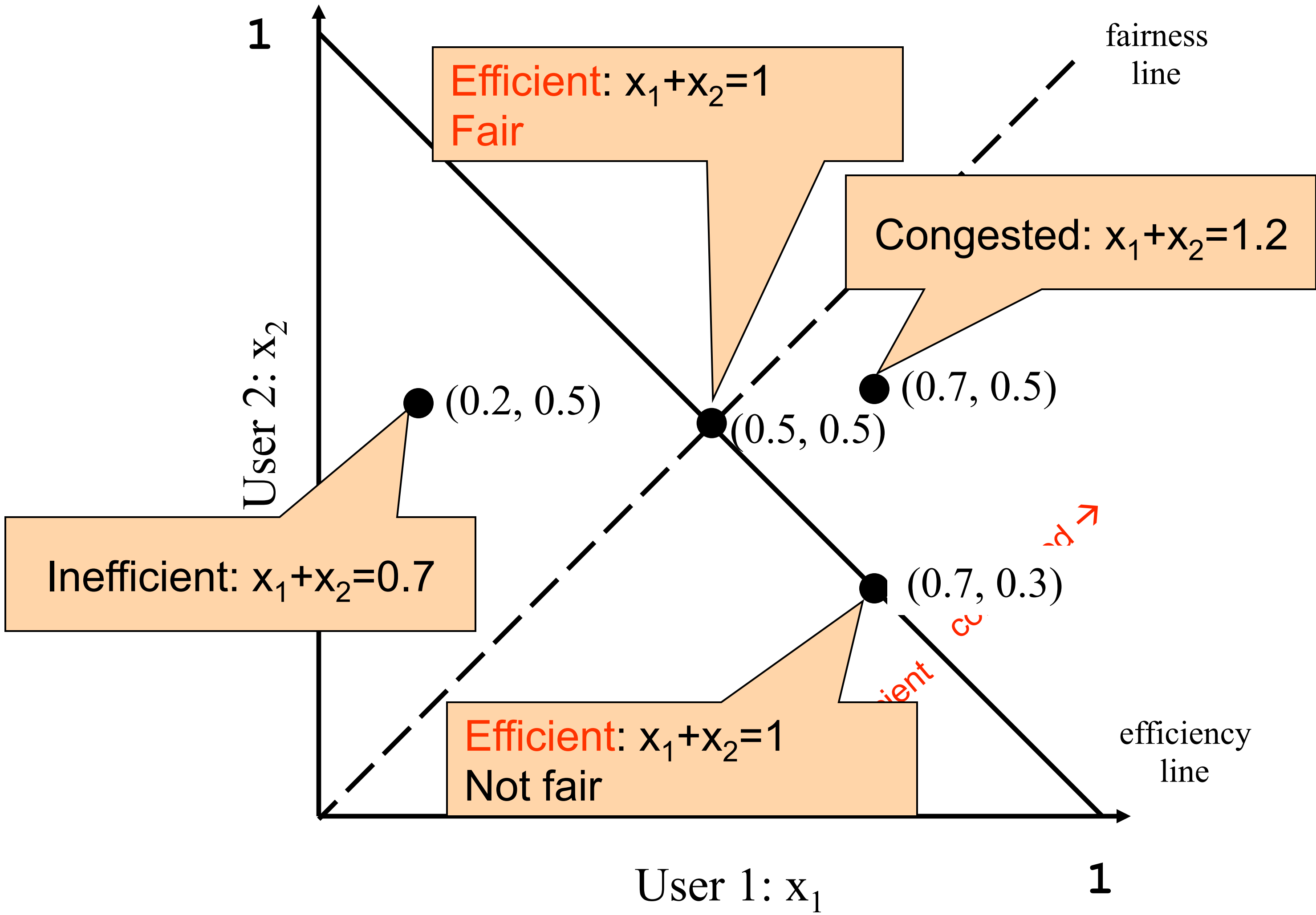
- Some rate adjustment options: Every RTT, we can
  - Multiplicative increase or decrease: CWND→ a*CWND
  - Additive increase or decrease: CWND→ CWND + b
- Four alternatives:
  - AIAD: gentle increase, gentle decrease
  - AIMD: gentle increase, drastic decrease
  - MIAD: drastic increase, gentle decrease
  - MIMD: drastic increase and decrease

Simple Model of Congestion Control

- Two users
  - rates x_1 and x_2
- Congestion when $x_1+x_2 > 1$
- Unused capacity when $x_1+x_2 < 1$
- Fair when $x_1 = x_2$



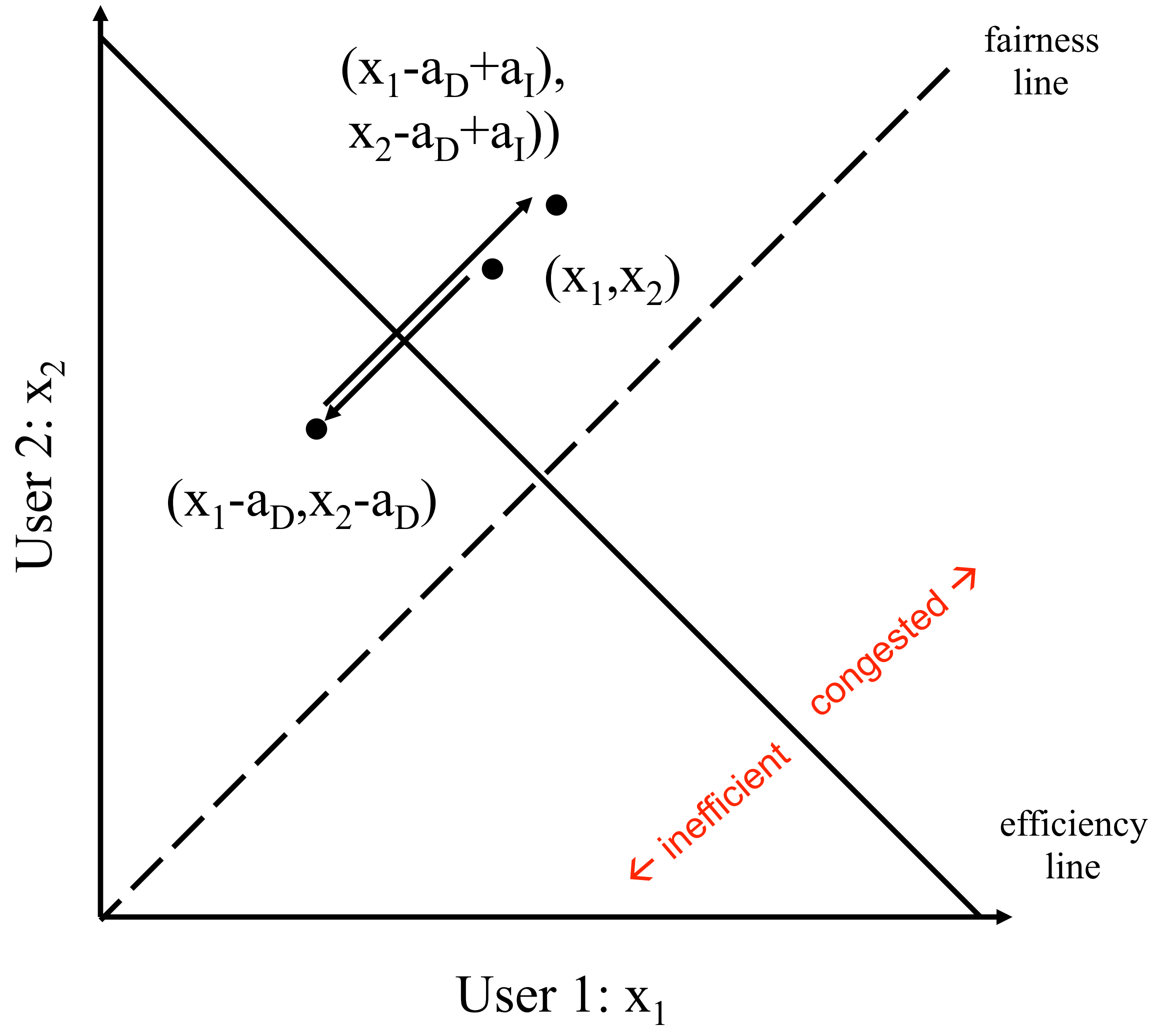
Example



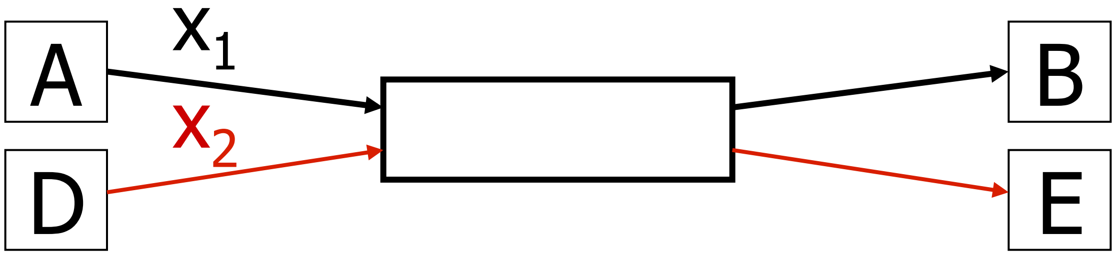
1
fairness line
Efficient: $x_1+x_2=1$
Fair
Congested: $x_1+x_2=1.2$
User 2: x_2
(0.2, 0.5)
(0.5, 0.5)
(0.7, 0.5)
Inefficient: $x_1+x_2=0.7$
(0.7, 0.3)
Efficient: $x_1+x_2=1$
Not fair
efficiency line
User 1: x_1
1

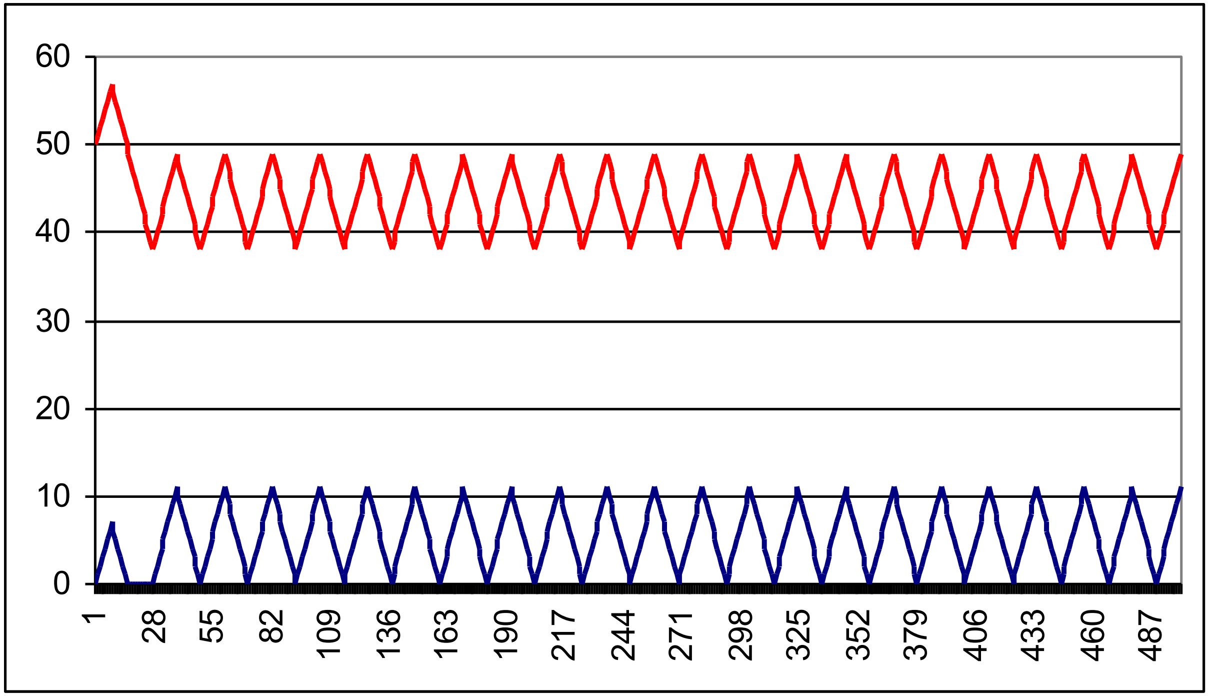
AIAD

- Increase: $x + a_I$
- Decrease: $x - a_D$
- Does not converge to fairness



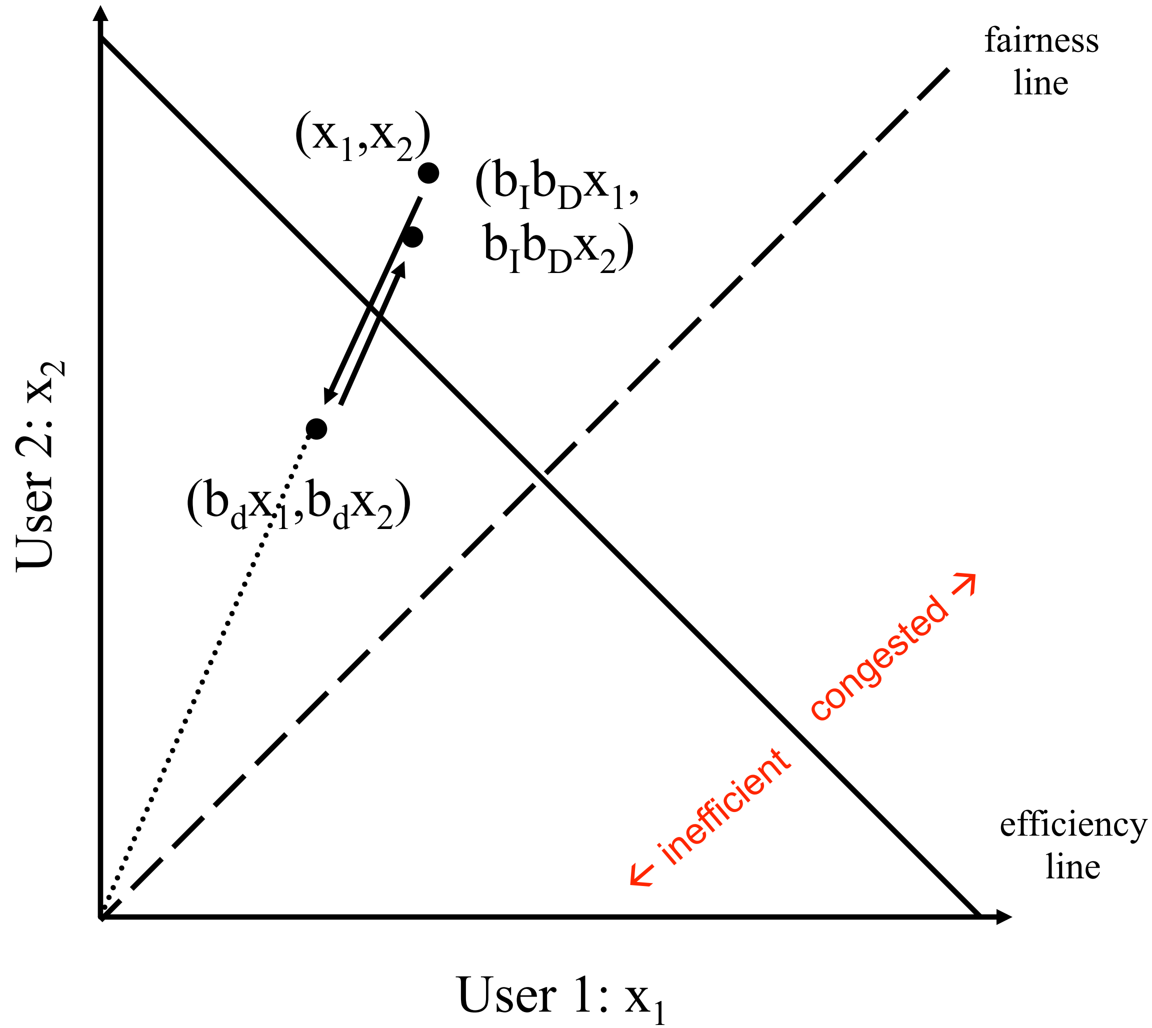
AIAD Sharing Dynamics





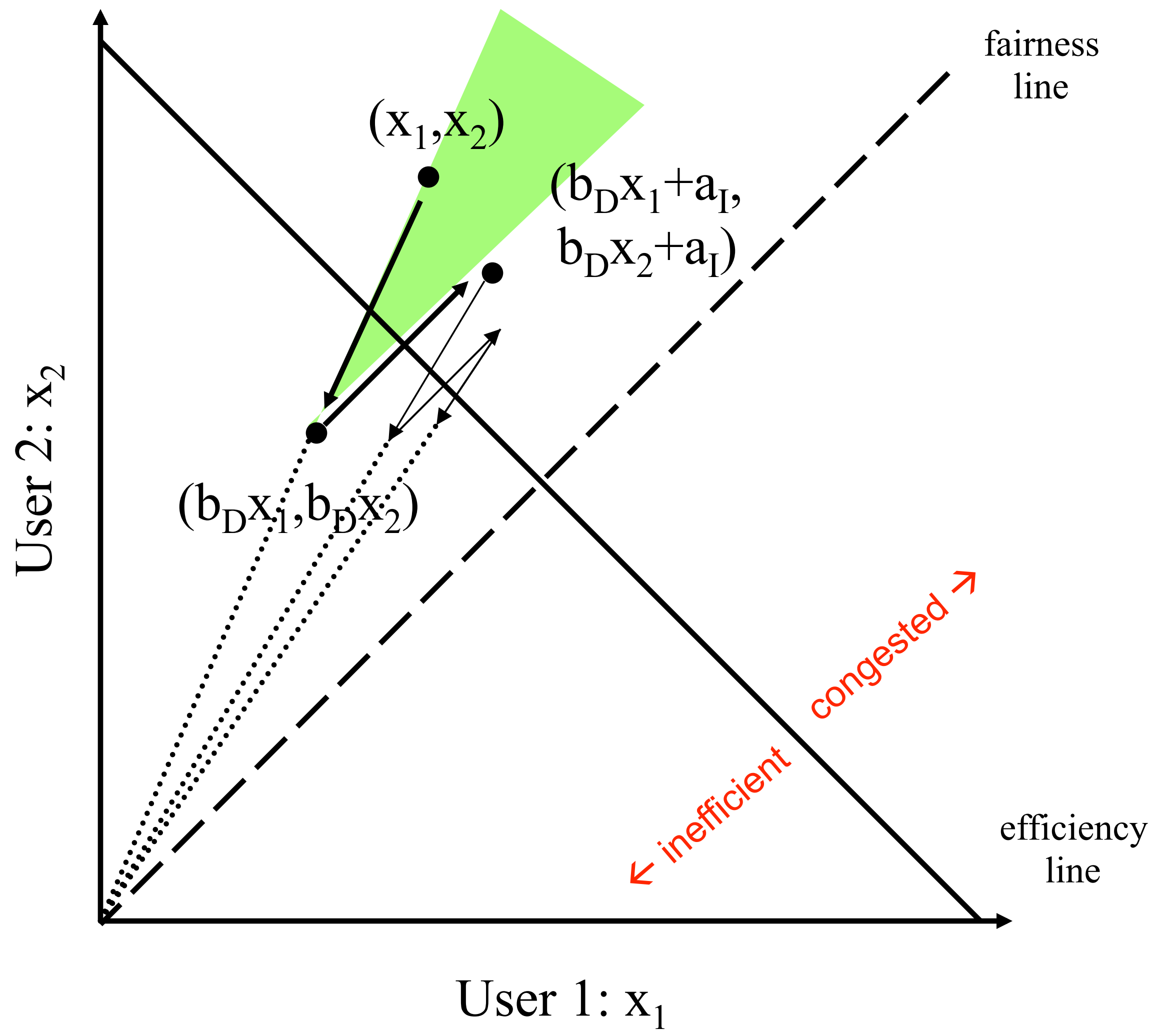
MIMD

- Increase: $x*b_I$
- Decrease: $x*b_D$
- Does not converge to fairness

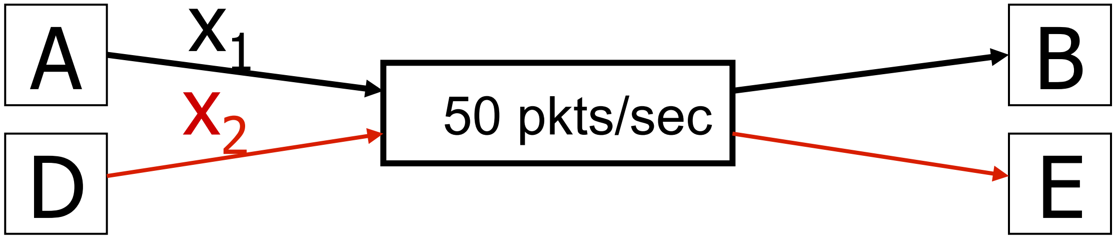


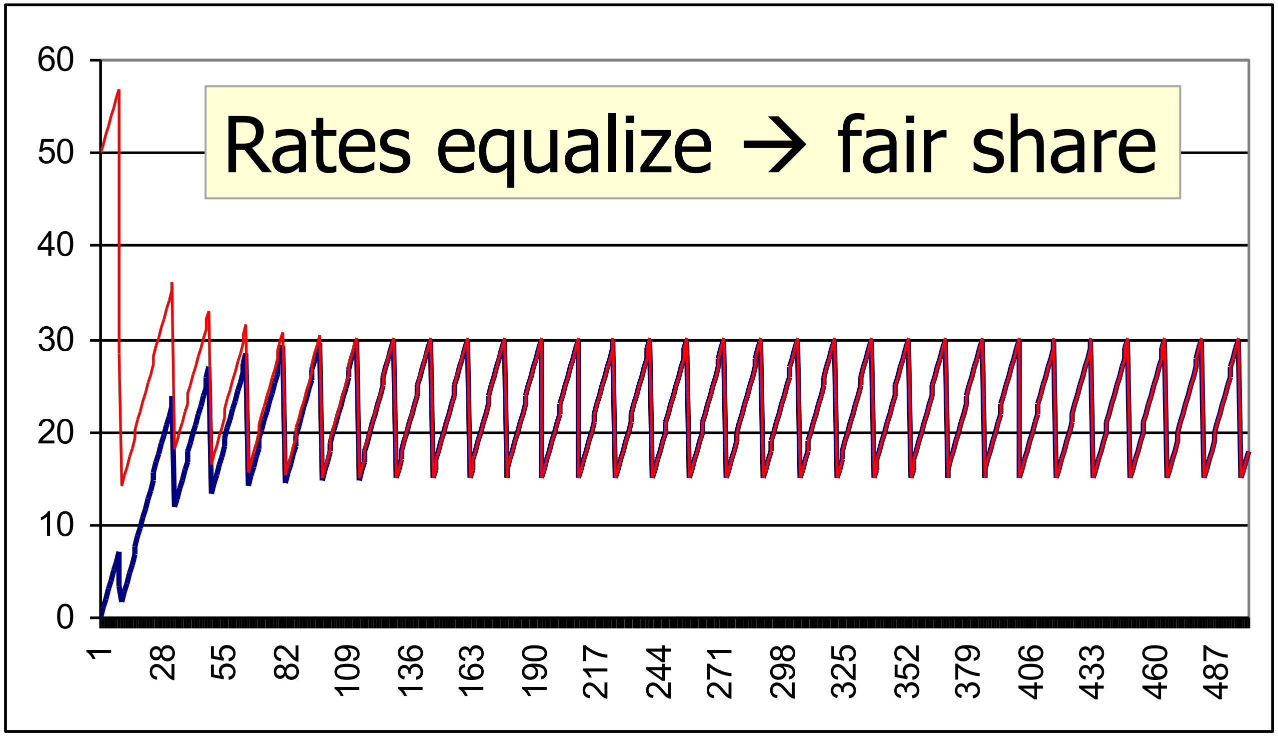
AIMD

- Increase: $x+a_I$
- Decrease: $x*b_D$
- Converges to fairness



AIMD Sharing Dynamics





TCP Congestion Control Details

Implementation

- State at sender
  - CWND (initialized to a small constant)
  - ssthresh (initialized to a large constant)
  - [Also dupACKcount and timer, as before]

- Events
  - ACK (new data)
  - dupACK (duplicate ACK for old data)
  - Timeout

Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

- *CWND packets per RTT*
- *Hence after one RTT with no drops:*
  CWND = 2xCWND

Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

Slow start phase

- Else
  - CWND = CWND + 1/CWND

- *CWND packets per RTT*
- *Hence after one RTT with no drops:*
  CWND = CWND + 1

"Congestion Avoidance" phase (additive increase)

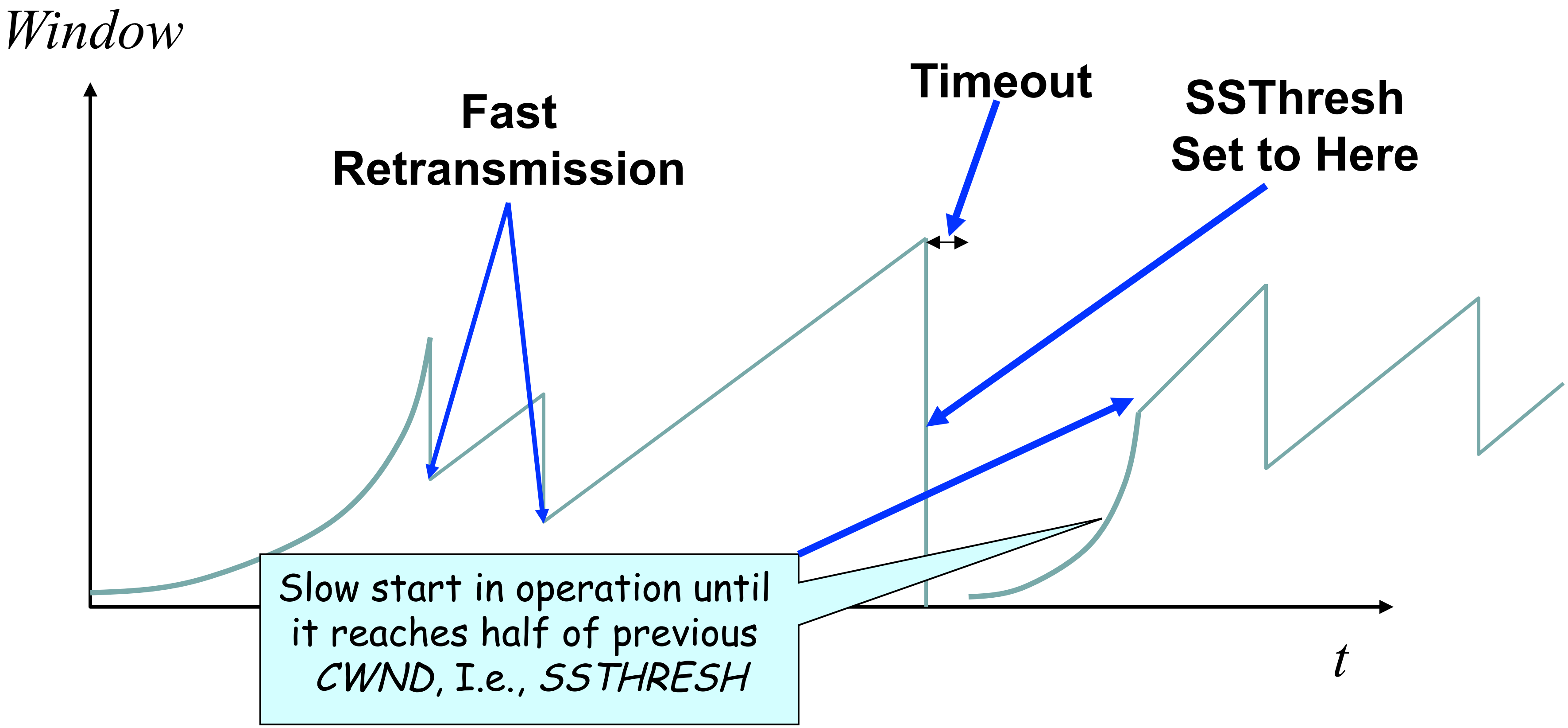
Event: TimeOut

- On Timeout
  - ssthresh ← CWND/2
  - CWND ← 1

Event: dupACK

- dupACKcount ++
- If dupACKcount = 3 /* fast retransmit */
  - ssthresh = CWND/2
  - CWND = CWND/2

Example



Slow-start restart: Go back to CWND = 1 MSS, but take advantage of knowing the previous value of CWND

One Final Phase: Fast Recovery

- The problem: congestion avoidance too slow in recovering from an isolated loss

Example

- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101
- 10 packets [101, 102, 103,…, 110] are in flight
  - Packet 101 is dropped
  - What ACKs do they generate?
  - And how does the sender respond?

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)
- ACK 111 (due to 101) ← only now can we transmit new packets
- Plus no packets in flight so ACK "clocking" (to increase CWND) stalls for another RTT

Solution: Fast Recovery

Idea: Grant the sender temporary "credit" for each dupACK so as to keep packets in flight

- If dupACKcount = 3
  - ssthresh = cwnd/2
  - cwnd = ssthresh + 3
- While in fast recovery
  - cwnd = cwnd + 1 for each additional duplicate ACK
- Exit fast recovery after receiving new ACK
  - set cwnd = ssthresh

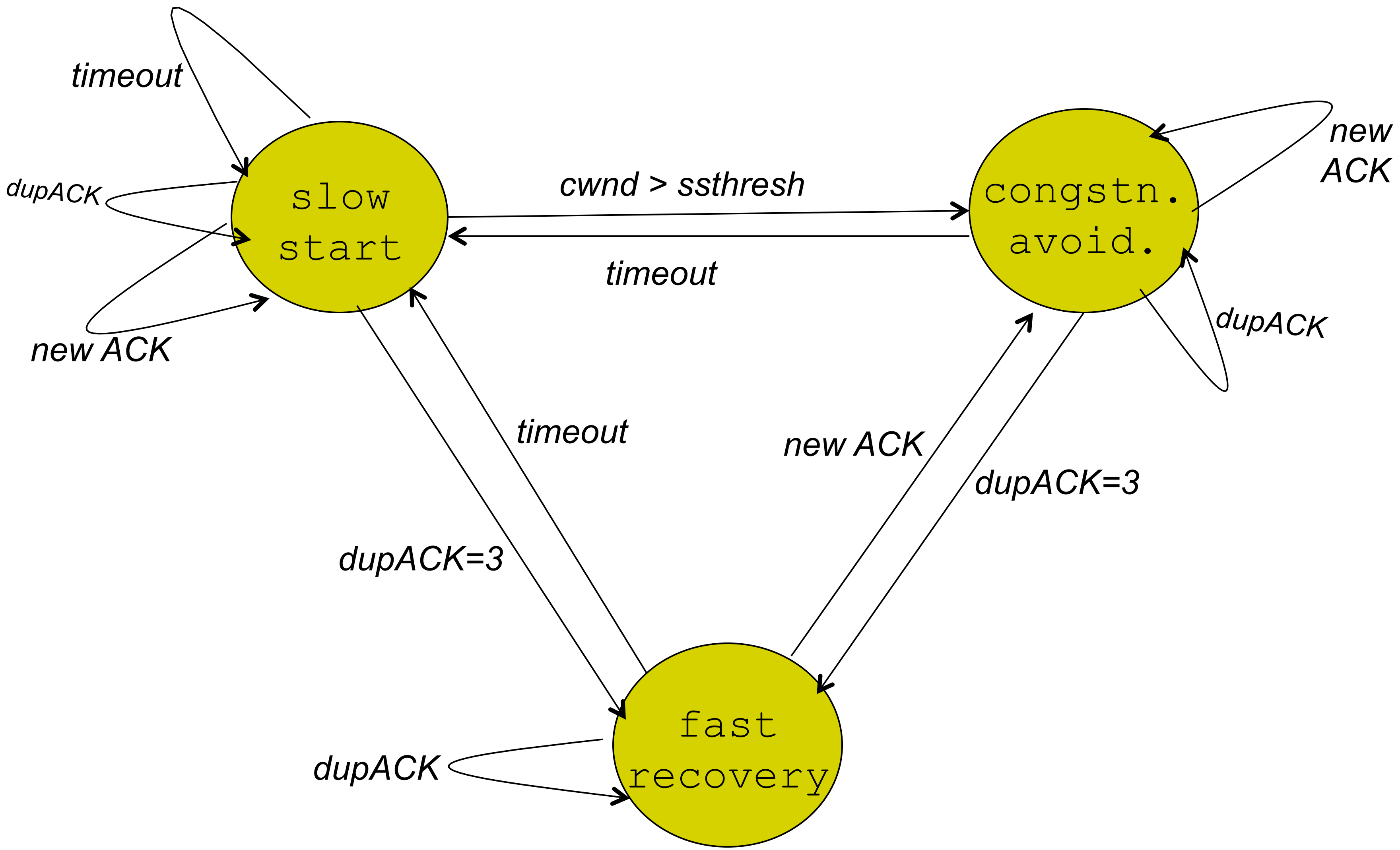
Example

- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101
- 10 packets [101, 102, 103,…, 110] are in flight
  - Packet 101 is dropped

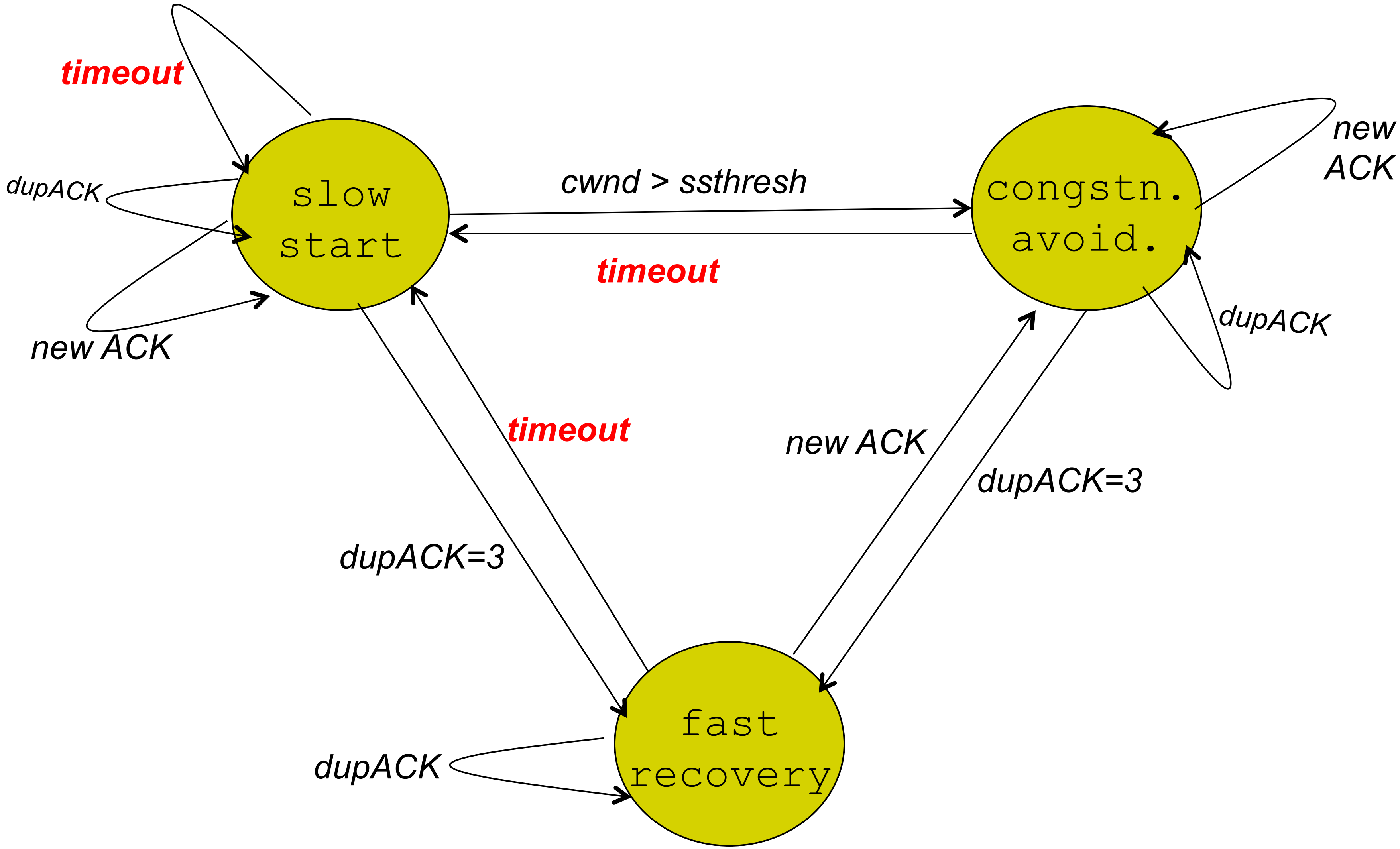
Timeline

- ACK 101 (due to 102) cwnd=10 dup#1
- ACK 101 (due to 103) cwnd=10 dup#2
- ACK 101 (due to 104) cwnd=10 dup#3
- REXMIT 101 ssthresh=5 cwnd= 8 (5+3)
- ACK 101 (due to 105) cwnd= 9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)
- ACK 101 (due to 109) cwnd=13 (xmit 113)
- ACK 101 (due to 110) cwnd=14 (xmit 114)
- ACK 111 (due to 101) cwnd = 5 (xmit 115) ← exiting fast recovery
- Packets 111-114 already in flight
- ACK 112 (due to 111) cwnd = 5 + 1/5 ← back in congestion avoidance

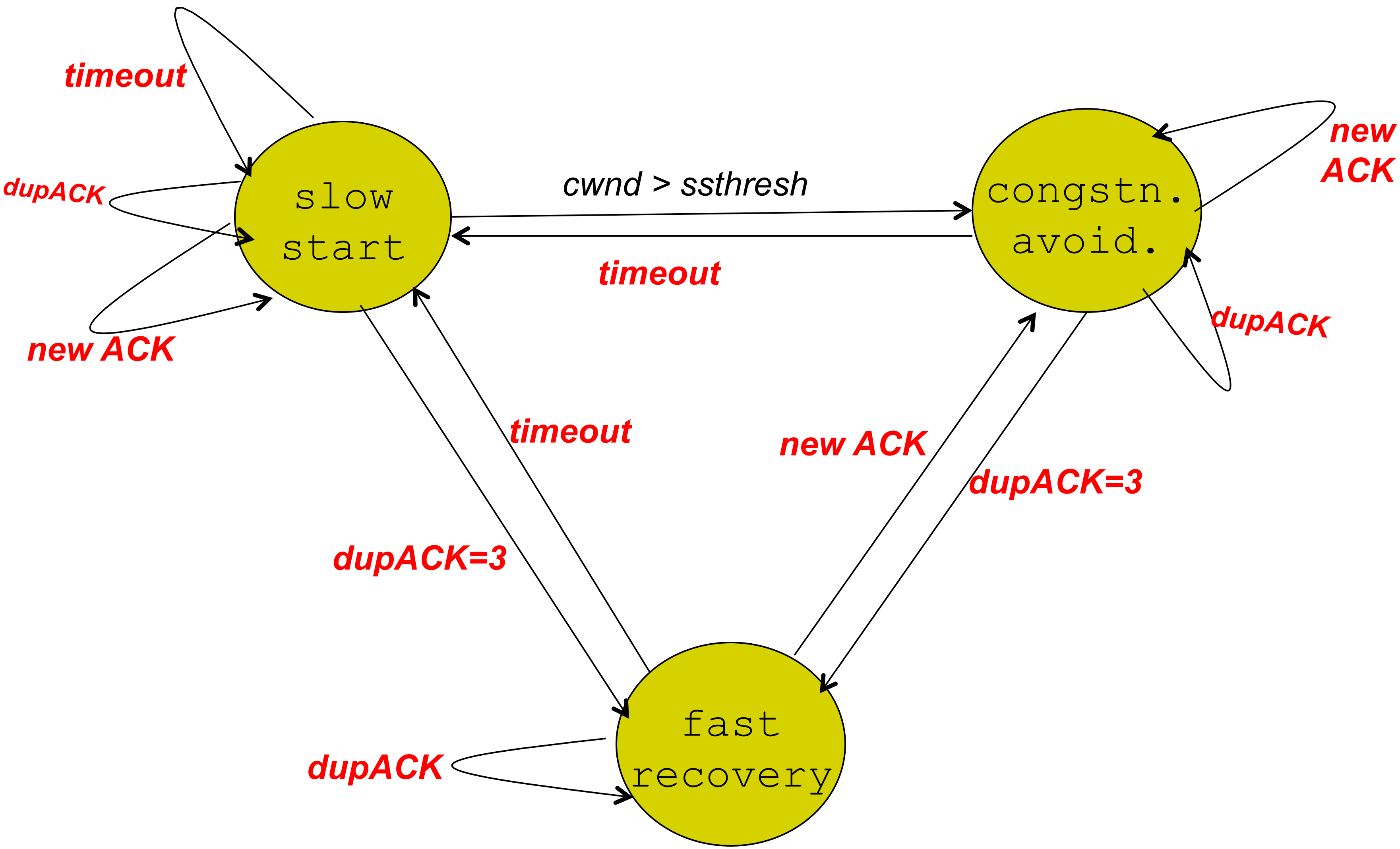
TCP State Machine



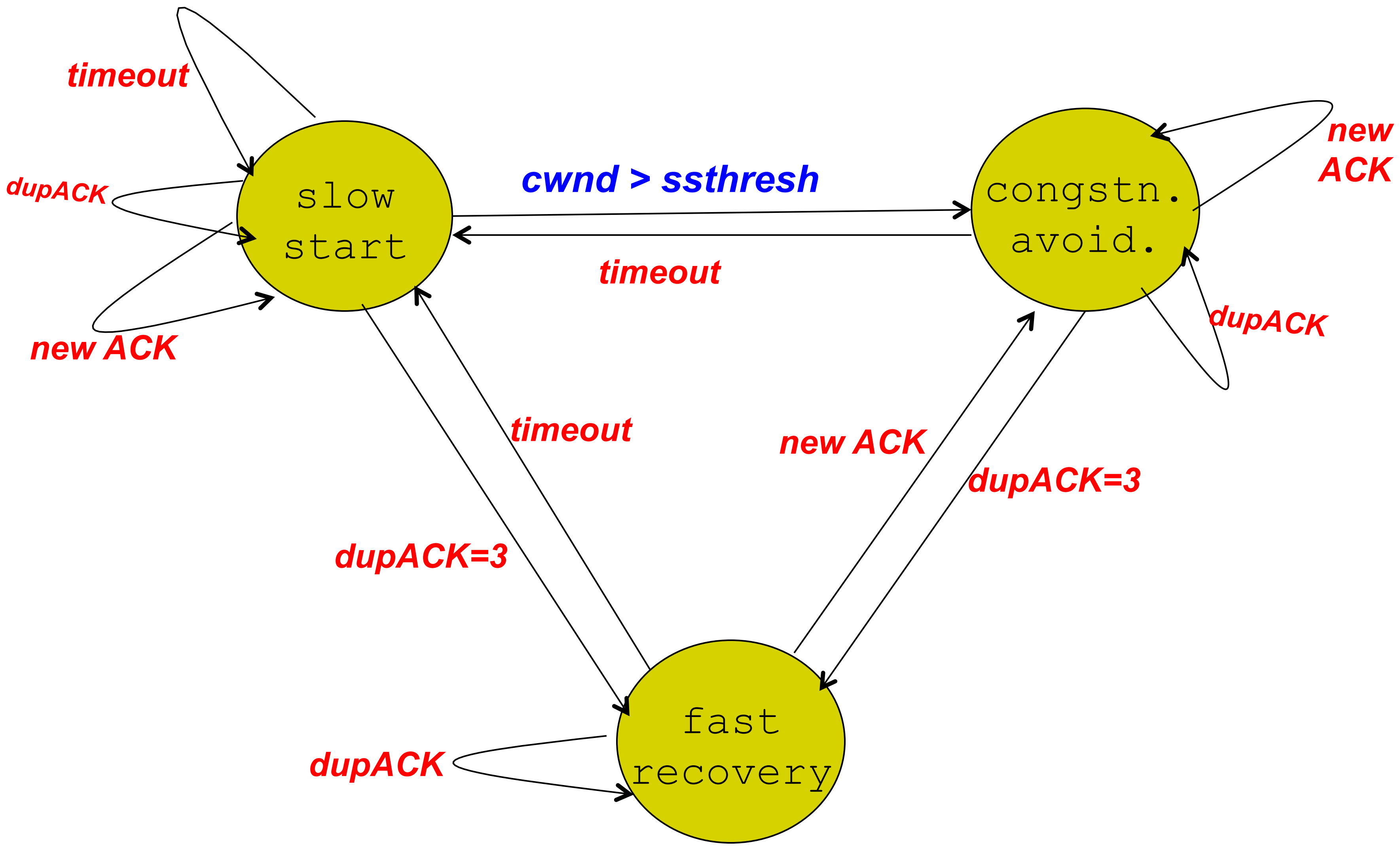
TCP State Machine



TCP State Machine



TCP State Machine



TCP Flavors

- TCP-Tahoe
  - CWND =1 on triple dupACK
- TCP-Reno
  - CWND =1 on timeout
  - CWND = CWND/2 on triple dupack
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - incorporates selective acknowledgements

Our default assumption