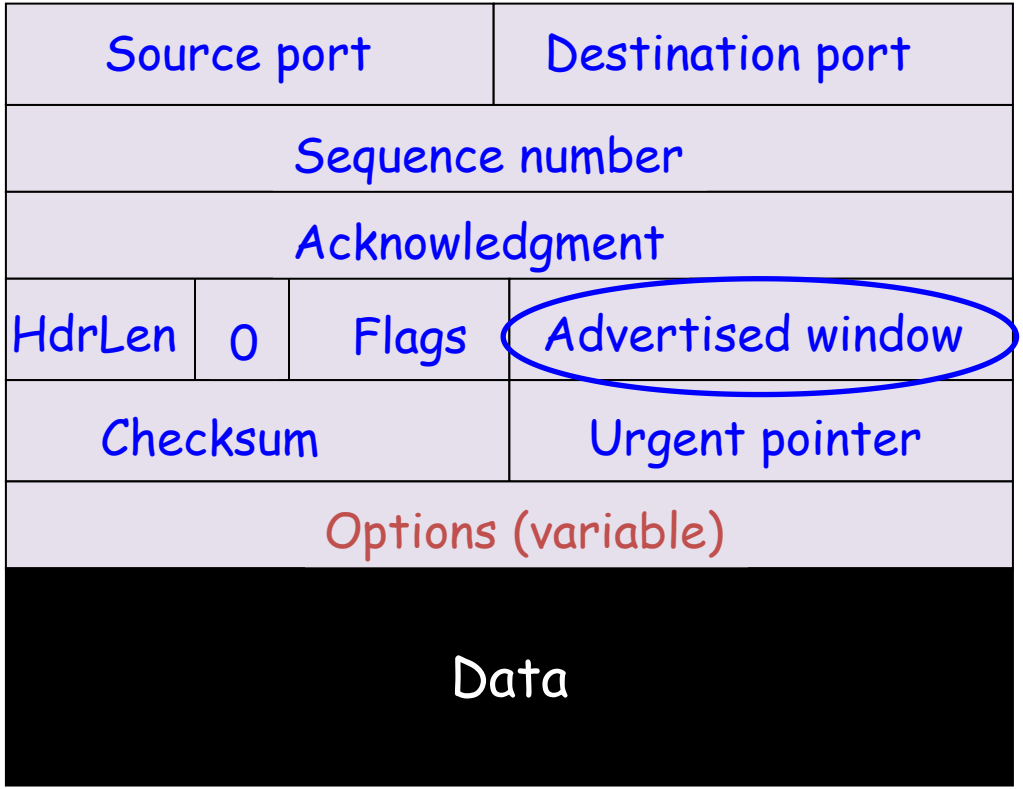
运输层

Outline

- TCP flow control
- TCP congestion control
- TCP congestion control wrap-up
- Router assisted congestion control

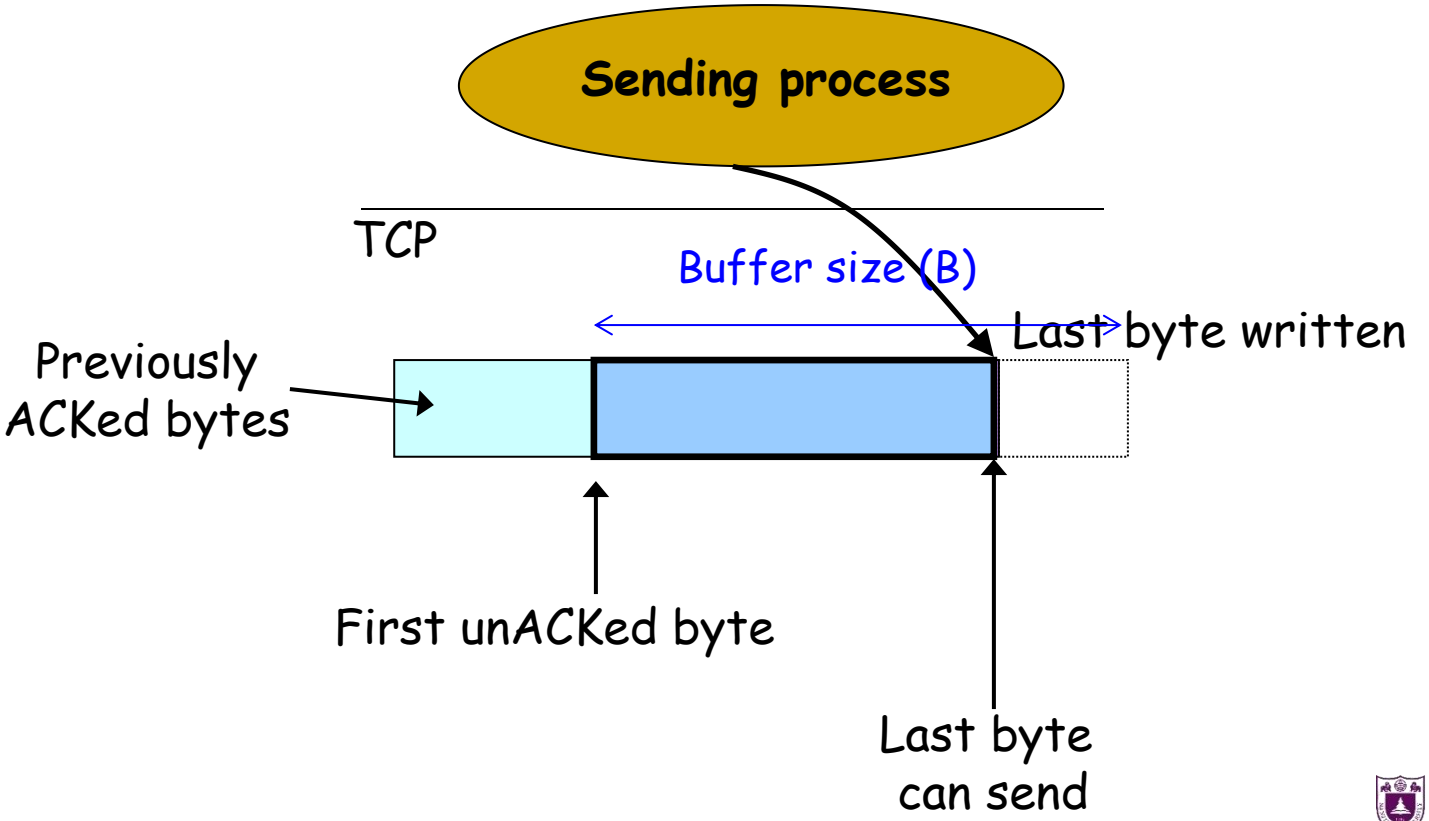
TCP header


Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

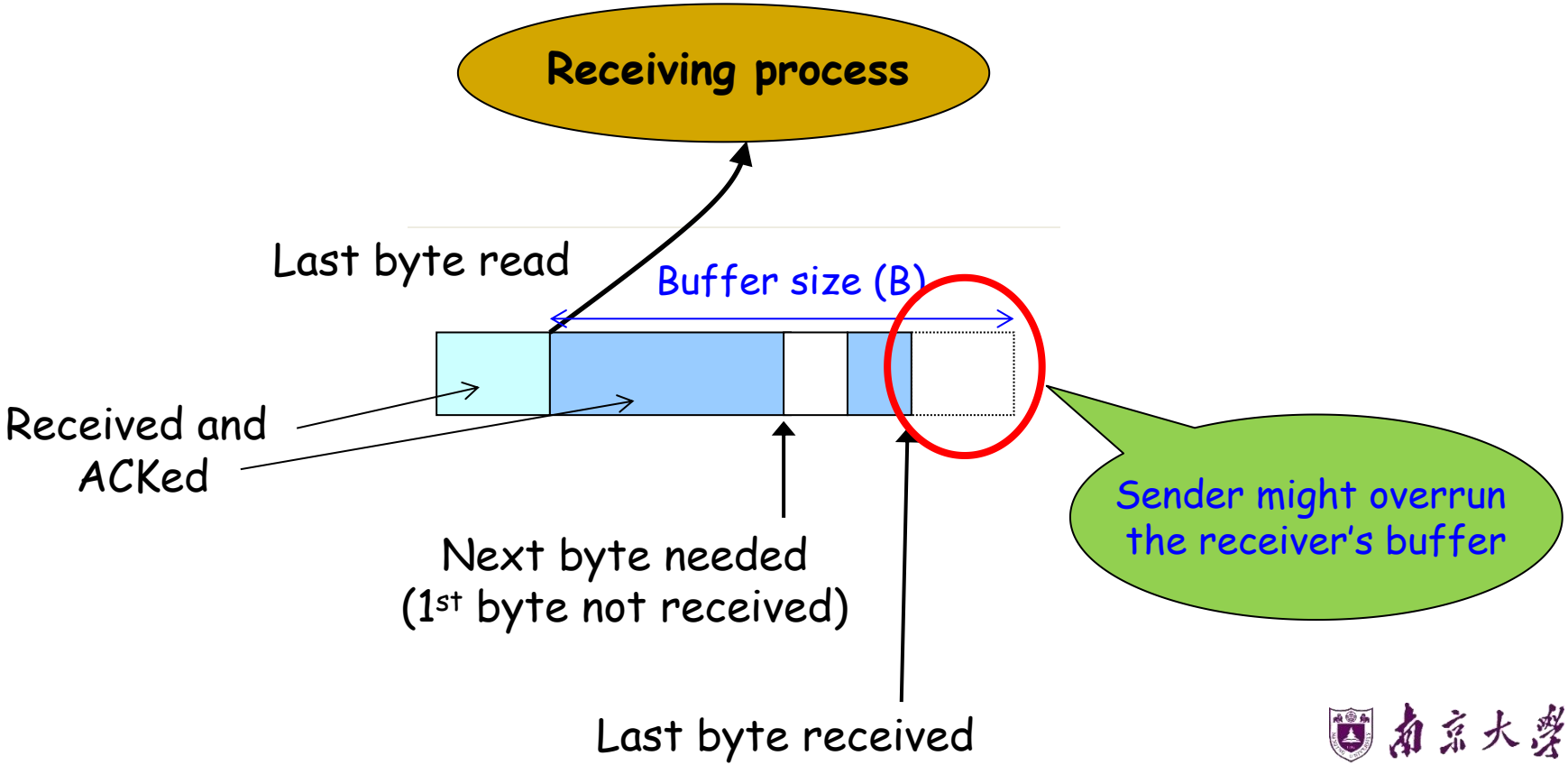
Recap: Sliding window

- Both sender and receiver maintain a window
- Left edge of window:
  - Sender: beginning of unacknowledged data
  - Receiver: beginning of expected data
    - ✓ First "hole" in received data
    - ✓ When sender gets ack, knows that receiver's window has moved
- Right edge: Left edge + constant
  - The constant is only limited by buffer size in the transport layer

Sliding window at sender



Sliding window at receiver



Fixed sliding window?

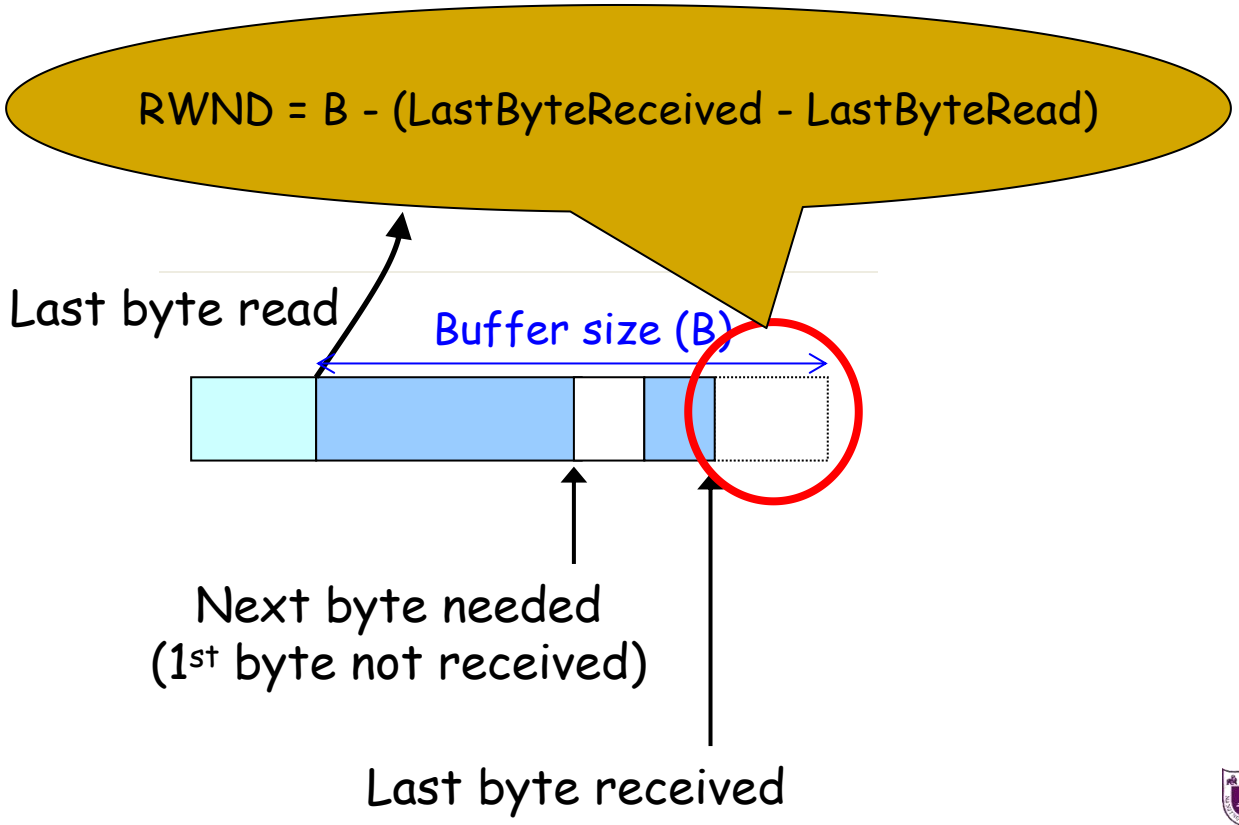
- Fixed sliding window
  - Works well on reliable direct links

- Problem:
  - Failure to receive ACK is taken as flow control indication
  - The receiver can achieve flow control by stop sending ACK, but the sender can not distinguish between lost segment and flow control

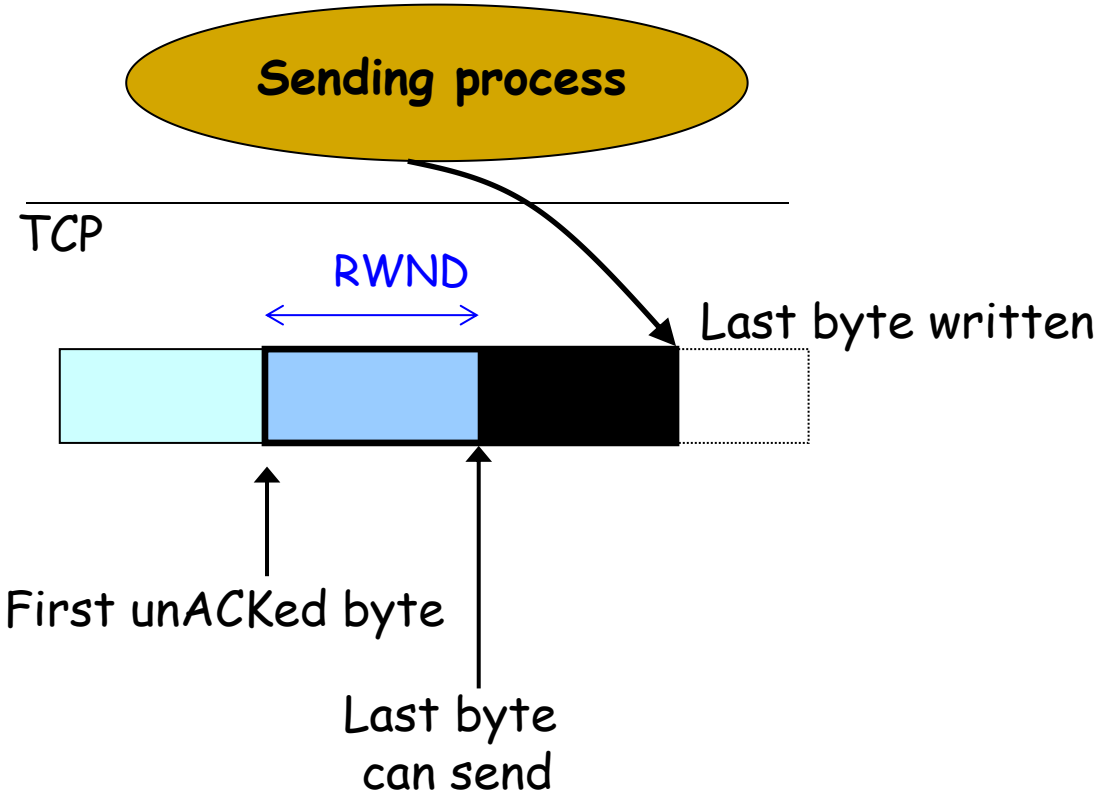
Solution: Credit Scheme

- Receiver advertises spare room (credits) using an "Advertised Window" (RWND) to prevent sender from overflowing its window
  - Receiver indicates value of RWND in ACKs
  - Sender ensures that the total number of bytes in flight <= RWND

Sliding window at receiver



Sliding window at sender

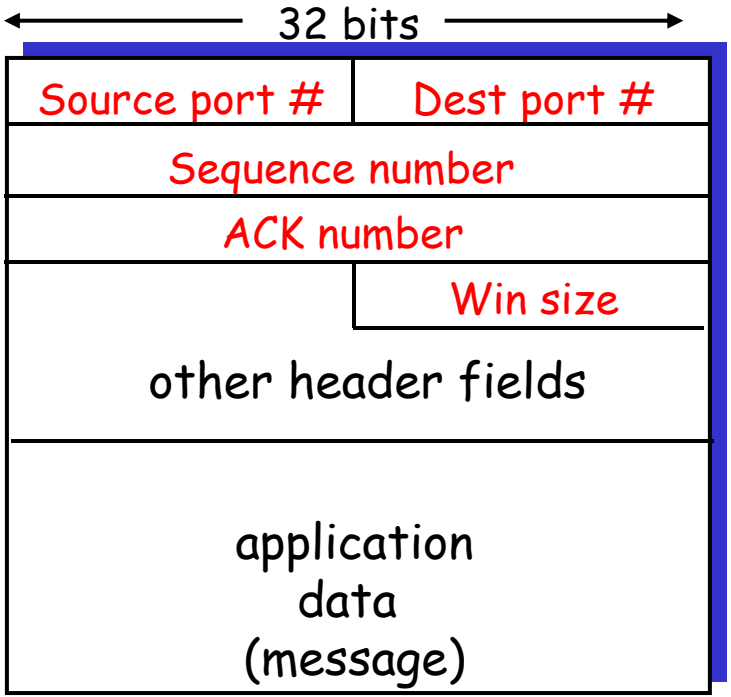


Sliding window with flow control

- Sender: window advances when new data ACK'd
- Receiver: window advances as receiving process consumes data
- Receiver advertises to the sender where the receiver window currently ends ("righthand edge")
  - Sender agrees not to exceed this amount
- UDP does not have flow control
  - Data can be lost due to buffer overflow

Benefit of Credit Scheme

- Greater control on Internet
- Decouples flow control from ACK
  - May ACK without granting credit
- Each octet has a sequence number
- Each transport segment has seq number, ack number and window size in header

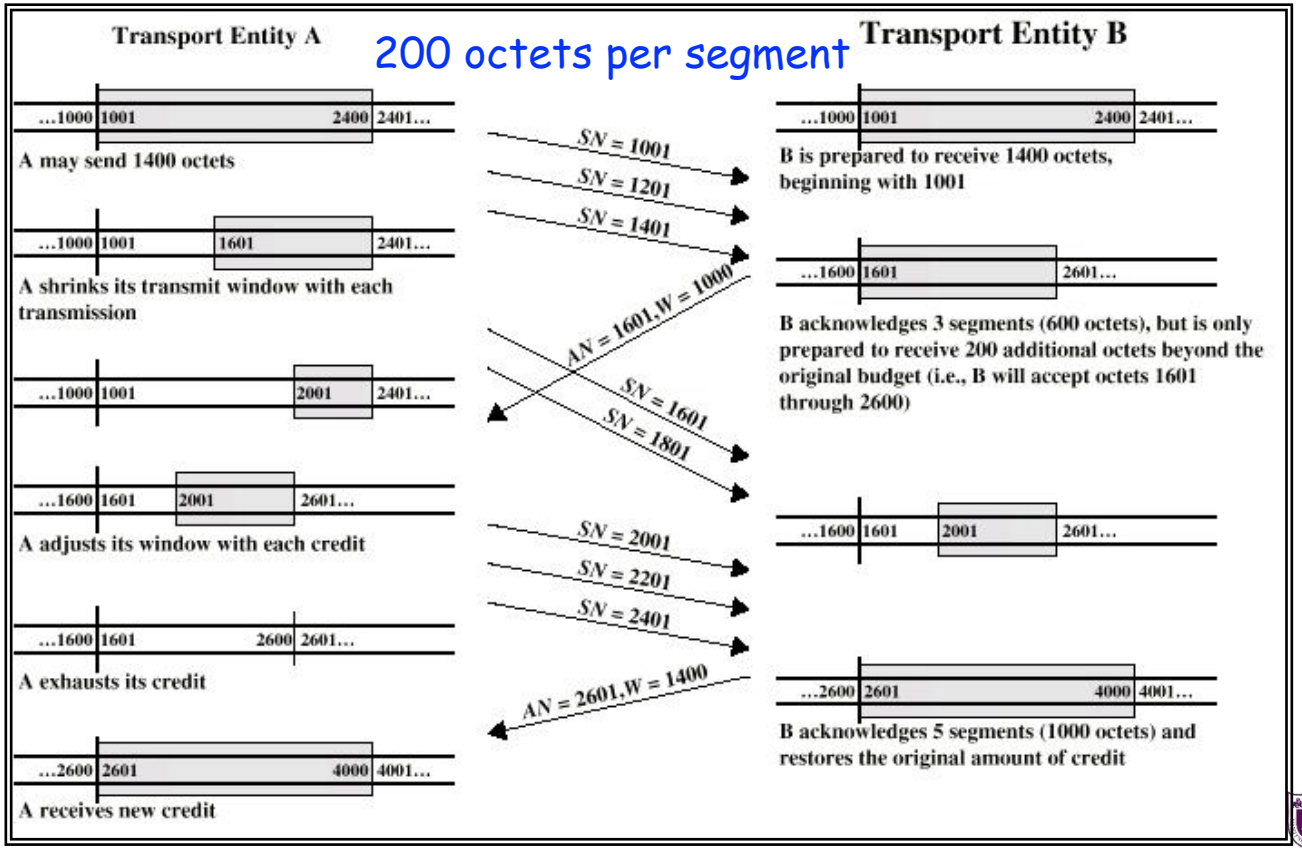


TCP segment format

Use of Header Fields

- When sending a segment
  - seq number (SN) is that of first octet in segment
  - ACK includes AN=i, W=j
- All octets through SN=i–1 acknowledged
  - Next expected octet is i
- Permission to send additional window of W=j octets
  - i.e. octets from i to i+j–1

Credit Allocation Procedure



Credit allocation flow control

Credit allocation flow control mechanism

- Suppose that the last octet of data received by B was octet number i-1, and that the last segment issued by B was (AN=i, W=j). Then
  - To increase credit to an amount k (k>j) when no additional data have arrived, B issues (AN=i, W=k)
  - To acknowledge an incoming segment containing m octets of data (m<j) without granting additional credit, B issues (AN=i+m, W=j-m)
- If an ACK/CREDIT segment is lost, little harm is done. Future acknowledgments will resynchronize the protocol.
- Further, if the sender times out and retransmits a data segment, it triggers a new acknowledgment.

- Credit allocation deadlock
  - B sends A: segment with AN=i, W=0 closing rcv-window
  - B sends A: AN=i, W=j to reopen, but this maybe lost
  - Now A thinks window is closed, B thinks it is open and wait
- Handle
  - Use window timer
  - If timer expires without any receiving, send something
  - Could be re-transmission of previous segment

Outline

- TCP flow control
- TCP congestion control
- TCP congestion control wrap-up
- Router assisted congestion control

A story: Congestion collapse in 1980s

- In 1981, TCP was standardized and widely deployed.
- No congestion control is considered.

- In October of 1986, the Internet had the first of what became a series of 'congestion collapses'. During this period, the data throughput from LBL to UC Berkeley (sites separated by 400 yards and two IMP hops) dropped from 32 Kbps to 40 bps. (Van Jacobson, Congestion Avoidance and Control)

- This open a new area of congestion control study.

Jacobson's fix to TCP

- 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

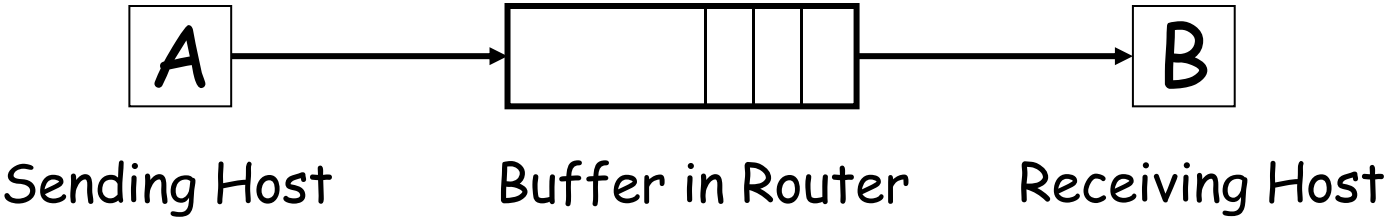
Key design considerations

- How do we know the network is congested?
  - Implicit and/or explicit signals from the network
- Who takes care of congestion?
  - End hosts (may receive some help from the network)
- How do we handle congestion?
  - Continuous adaptation

Three issues to consider

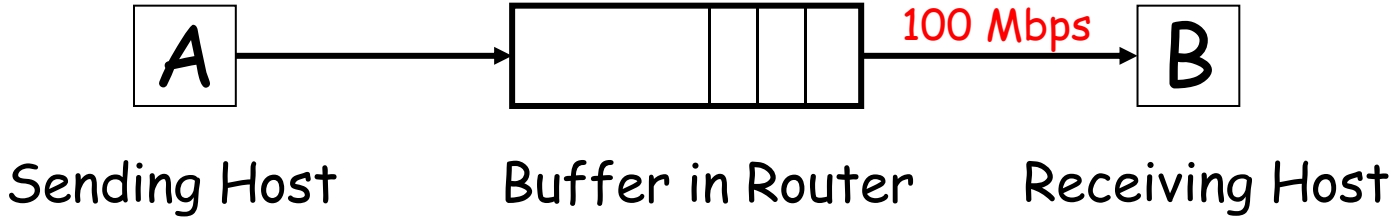
- Discovering the available (bottleneck) bandwidth
- Adjusting to variations in bandwidth
- Sharing bandwidth between flows

Abstract view



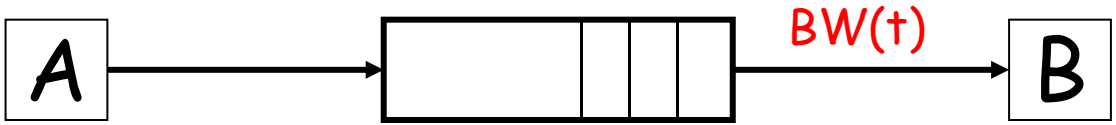
- Ignore internal structure of router and model it as a single queue for a particular input-output pair

Discovering available bandwidth



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

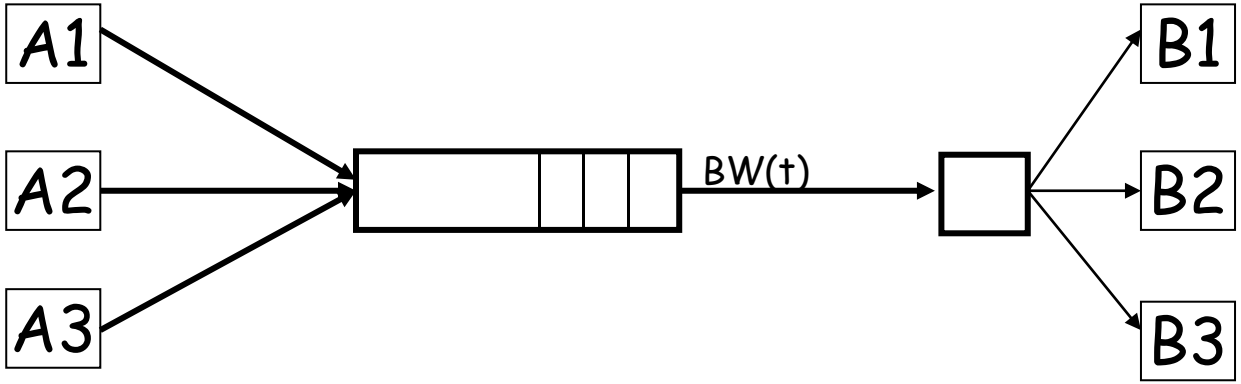
Adjusting to variations in bandwidth



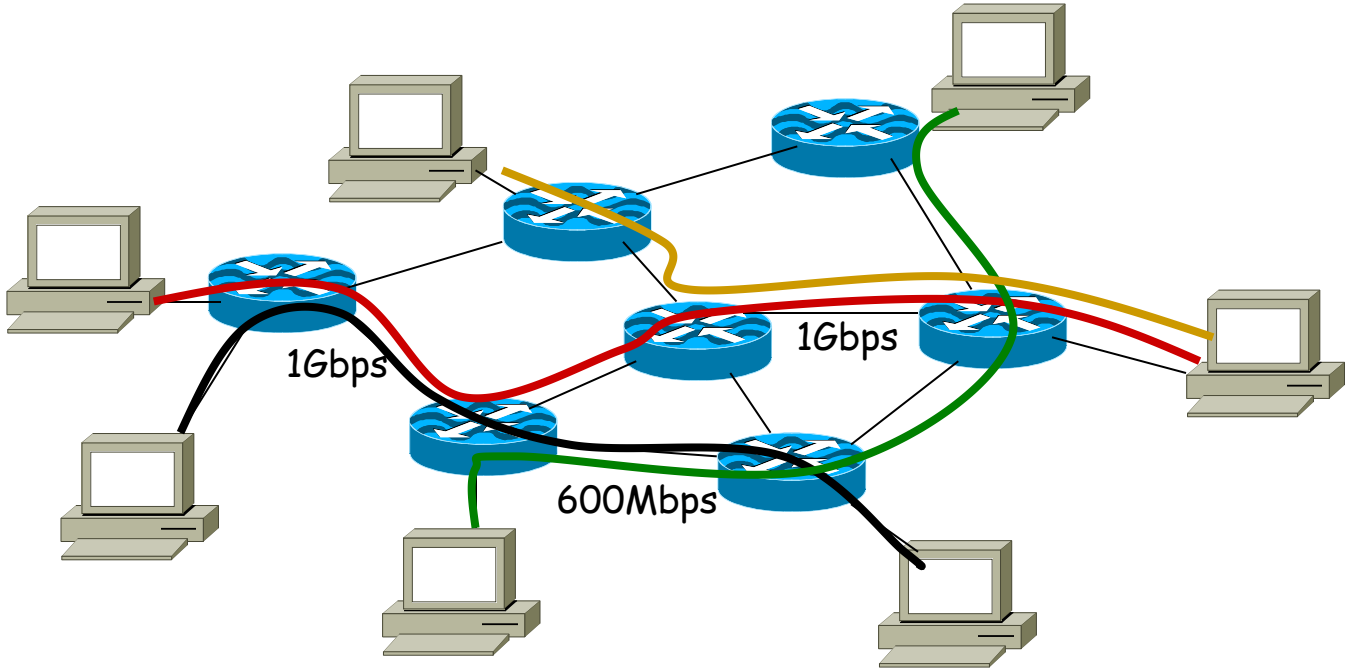
- Adjust rate to match instantaneous bandwidth
  - Assuming you have rough idea of bandwidth

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

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

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 end hosts 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 are 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 (Slow Start)
  - Adjusting to bandwidth variations (Congestion Avoidance: AIMD)

Bandwidth discovery with "Slow Start"

- Goal: estimate available bandwidth
  - Start slow (for safety)
  - 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

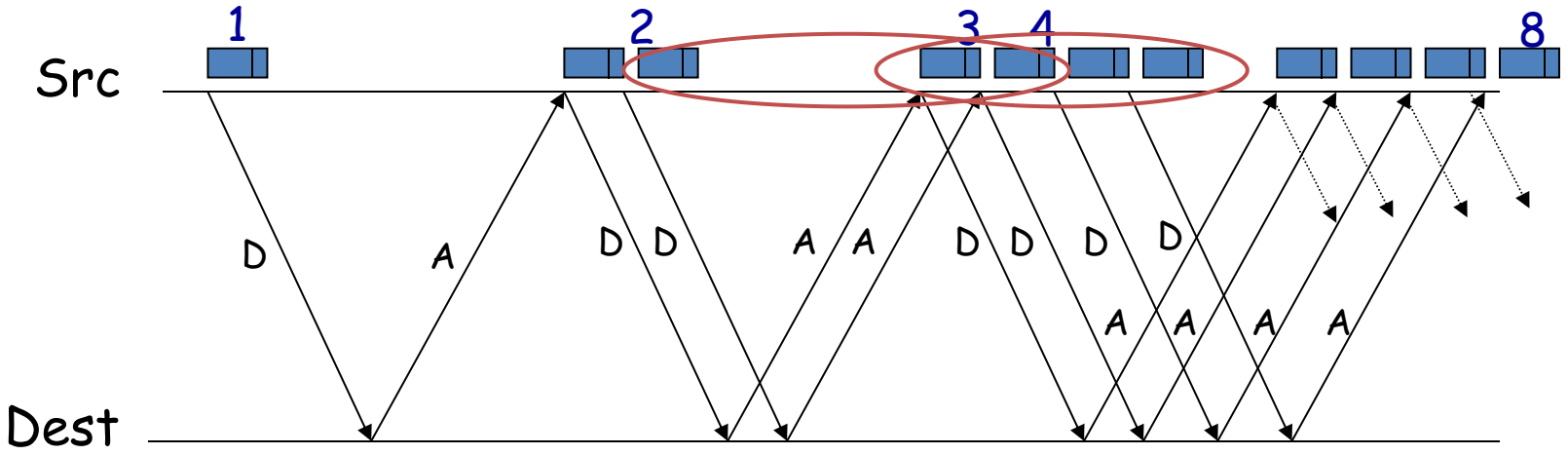
Slow Start in action

- For each RTT: double CWND
  - i.e., for each ACK, CWND += 1

Linear increase per ACK(CWND+1) ➔ exponential increase per RTT (2*CWND)

Slow Start in action

- For each RTT: double CWND
  - i.e., for each ACK, CWND += 1



When does Slow Start stop?

- Slow Start gives an estimate of available bandwidth
  - At some point, there will be loss
- Introduce a "slow start threshold" (ssthresh)
  - Initialized to a large value
- If CWND > ssthresh, stop Slow Start

Adjusting to varying bandwidth

- CWND > ssthresh
  - Stop rapid growth and focus on maintenance
- 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)

AIMD

- Additive increase: when CWND> ssthresh
  - For each ACK, CWND = CWND+ 1/CWND
  - CWND is increased by one only if all segments in a CWND have been acknowledged
- Multiplicative decrease
- On 3 duplicate ACKs (packet loss event)
  - ssthresh = CWND/2
  - CWND= ssthresh
  - Enter Congestion Avoidance: cwnd increases by 1 (linearly instead of exponentially) after each RTT
- On timeout event
  - ssthresh = CWND/2
  - CWND = 1
  - Initiate Slow Start

Illustration of Window

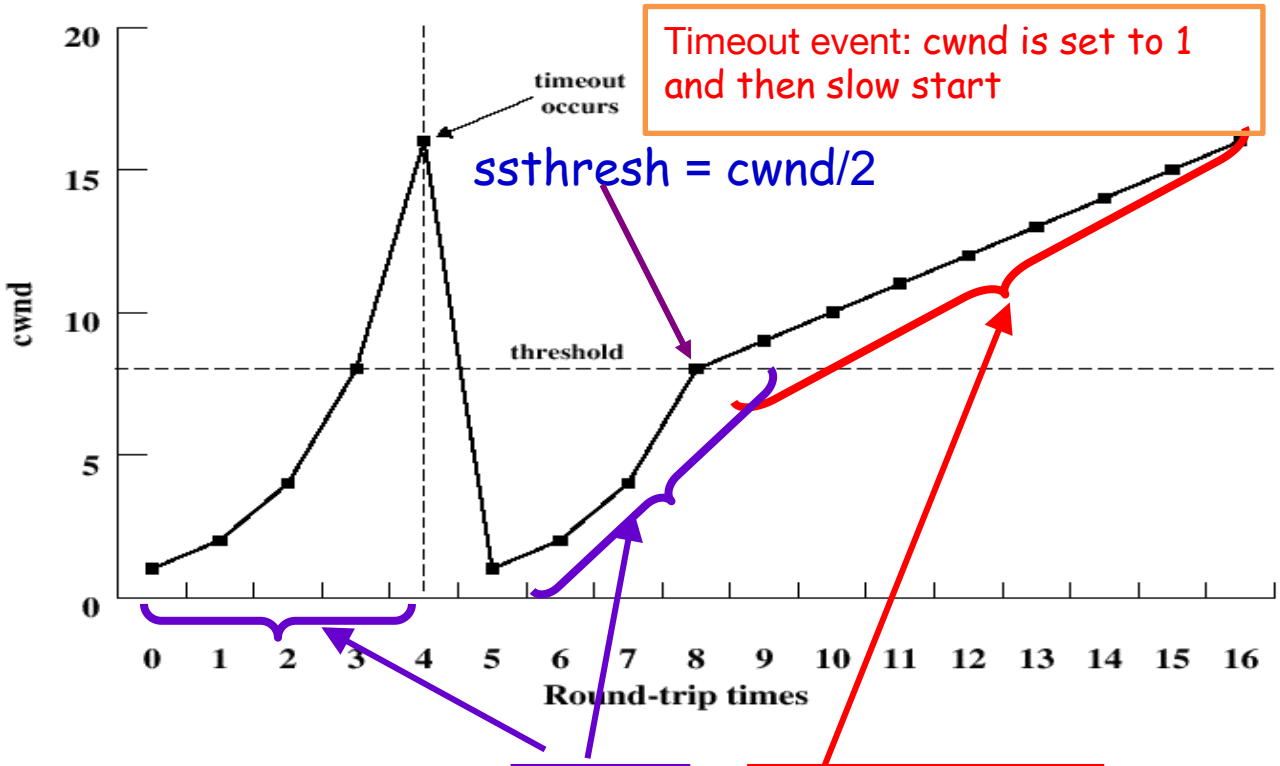
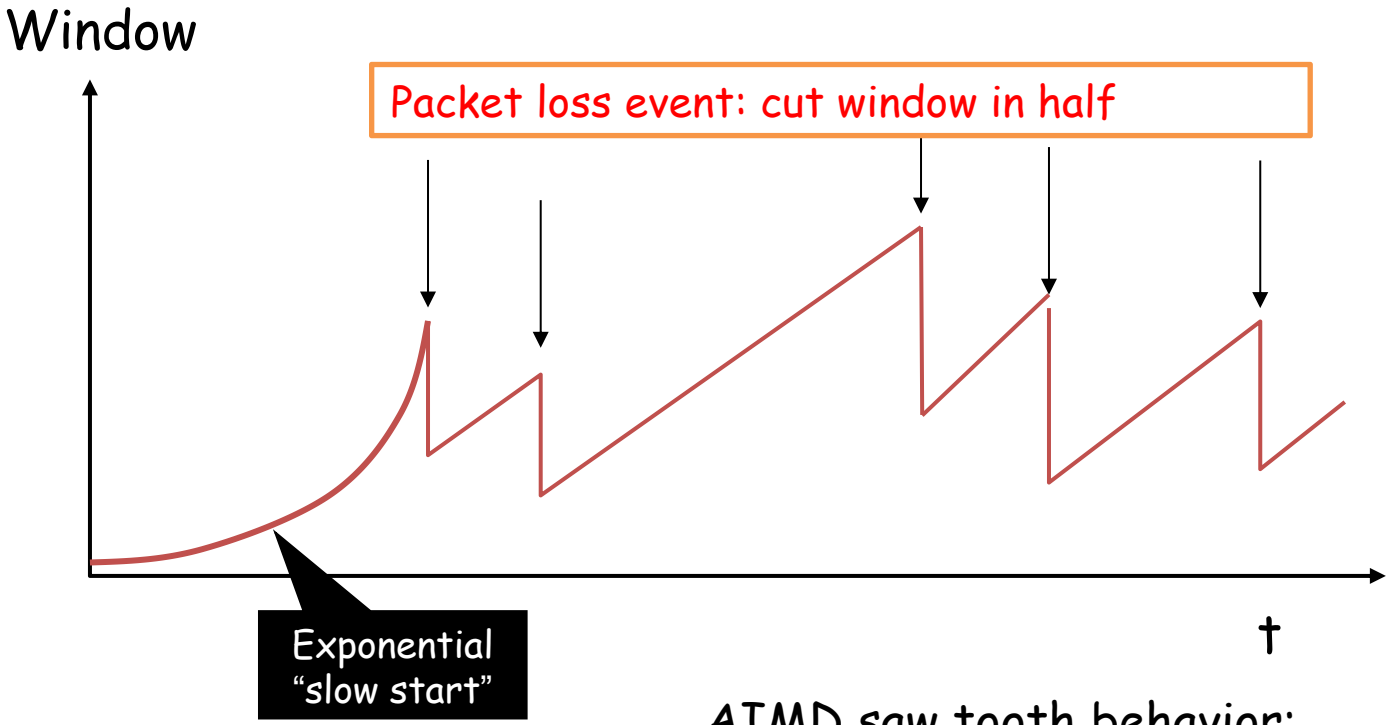


Figure 17.14 Illustration of Slow Start and Congestion Avoidance

Leads to the TCP "Sawtooth"



AIMD saw tooth behavior:
probing for bandwidth

Why AIMD?

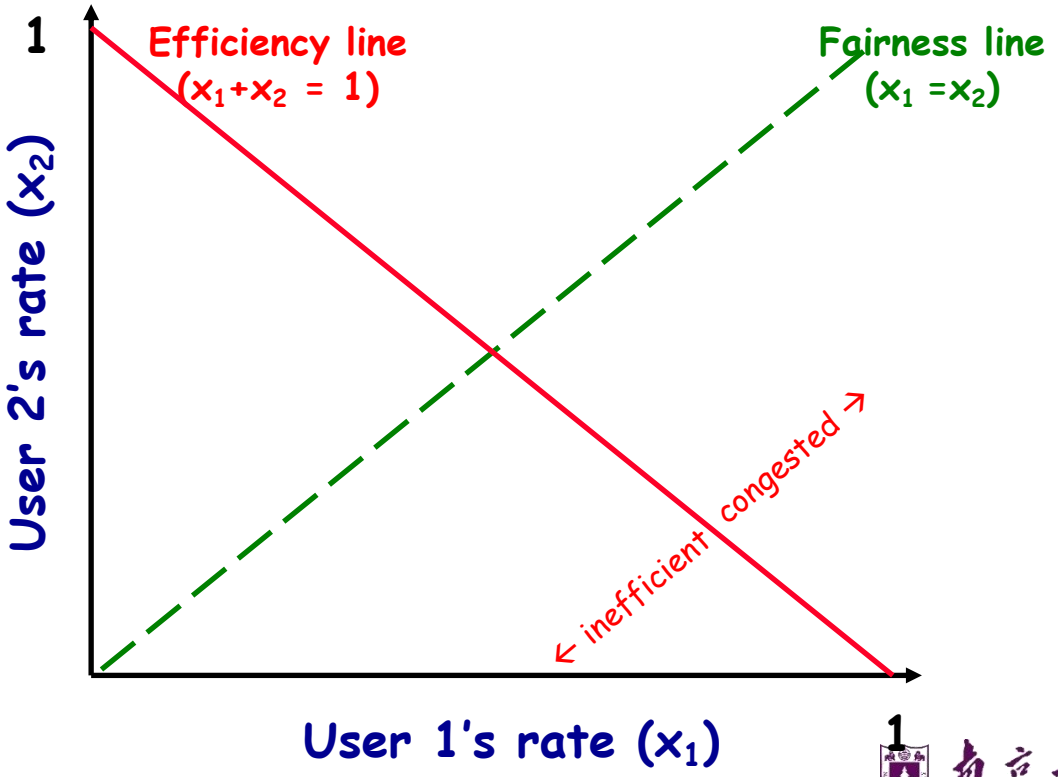
- Recall the three issues
  - Finding available bottleneck bandwidth
  - Adjusting to bandwidth variations
  - Sharing bandwidth
- Two goals for bandwidth sharing
  - Efficiency: High utilization of link bandwidth
  - Fairness: Each flow gets equal share

Why AIMD?

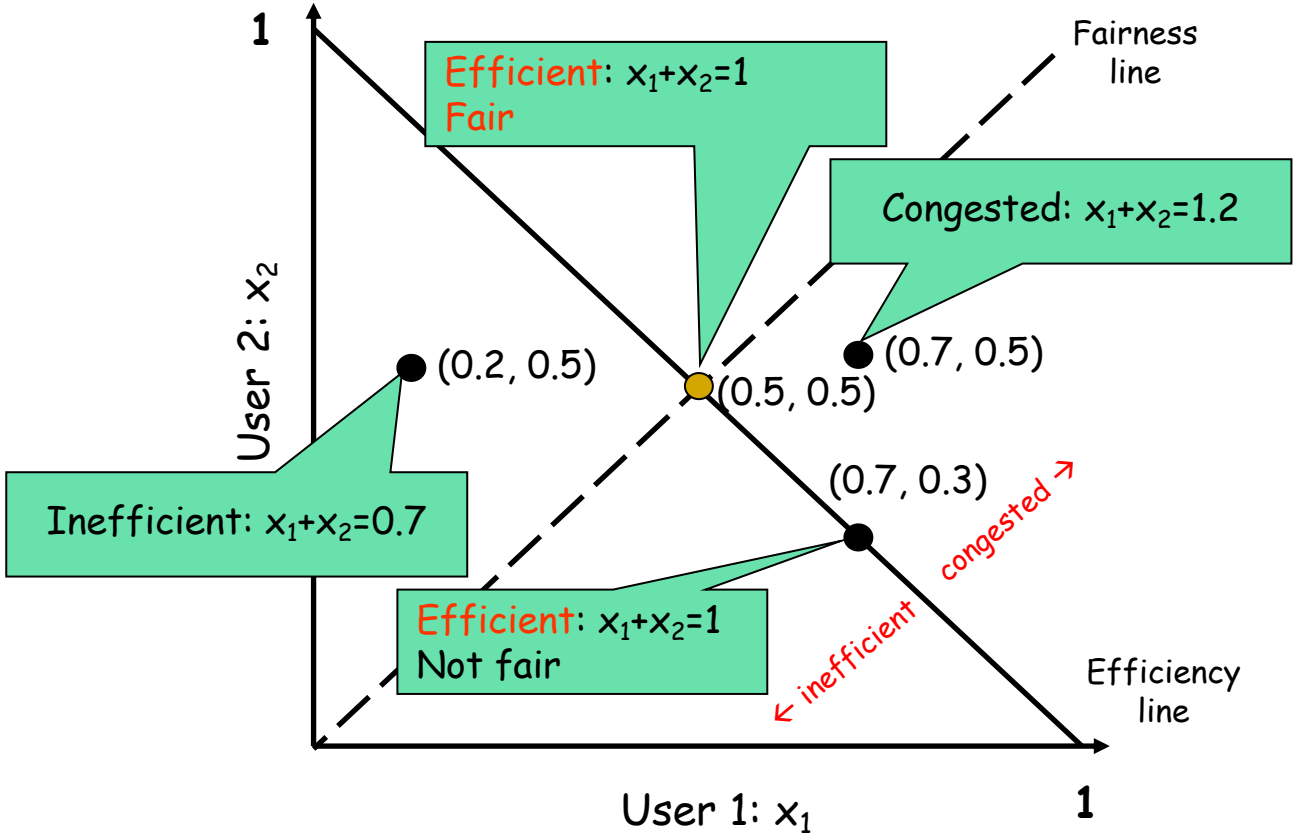
- Every RTT, we can do
  - Multiplicative increase or decrease: CWND→ a*CWND
  - Additive increase or decrease: CWND→ CWND + b
- Four alternatives:
  - AIAD: gentle increase, gentle decrease
  - AIMD: gentle increase, drastic decrease
  - MIAD: drastic increase, gentle decrease
  - MIMD: drastic increase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

Simple model of congestion control

- Two users
  - rates x1 and x2
- Congestion when x1+x2 > 1
- Unused capacity when x1+x2 < 1
- Fair when x1 =x2

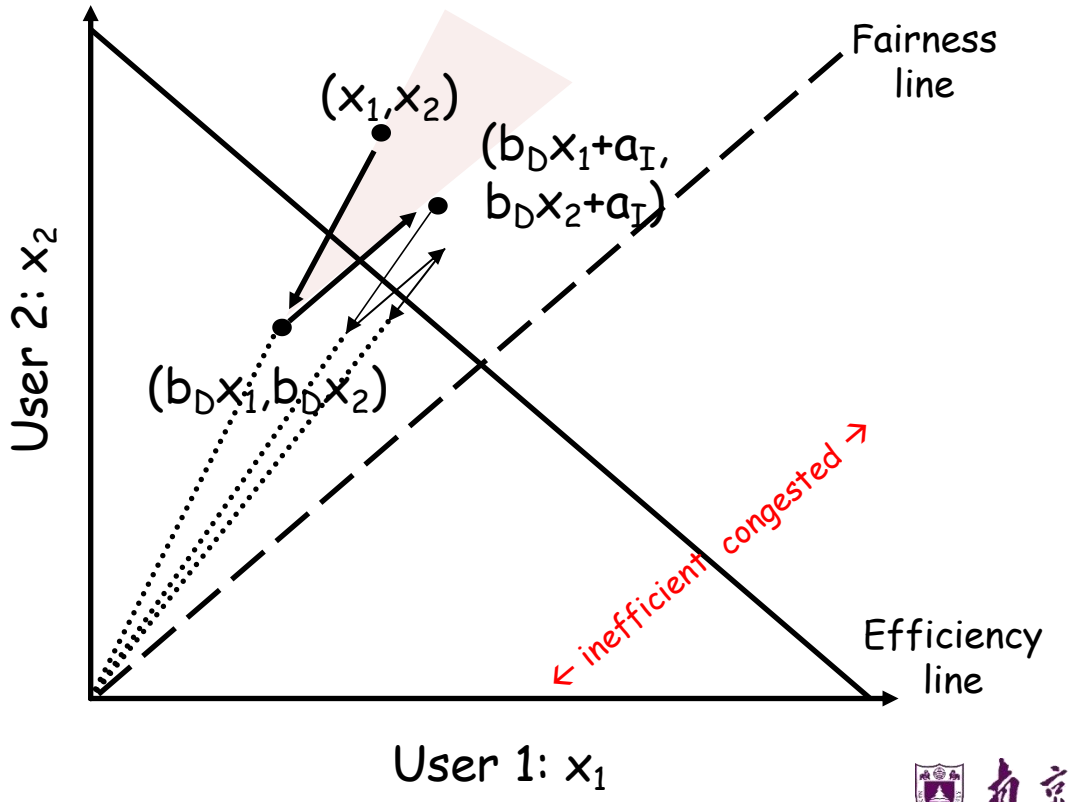


Example

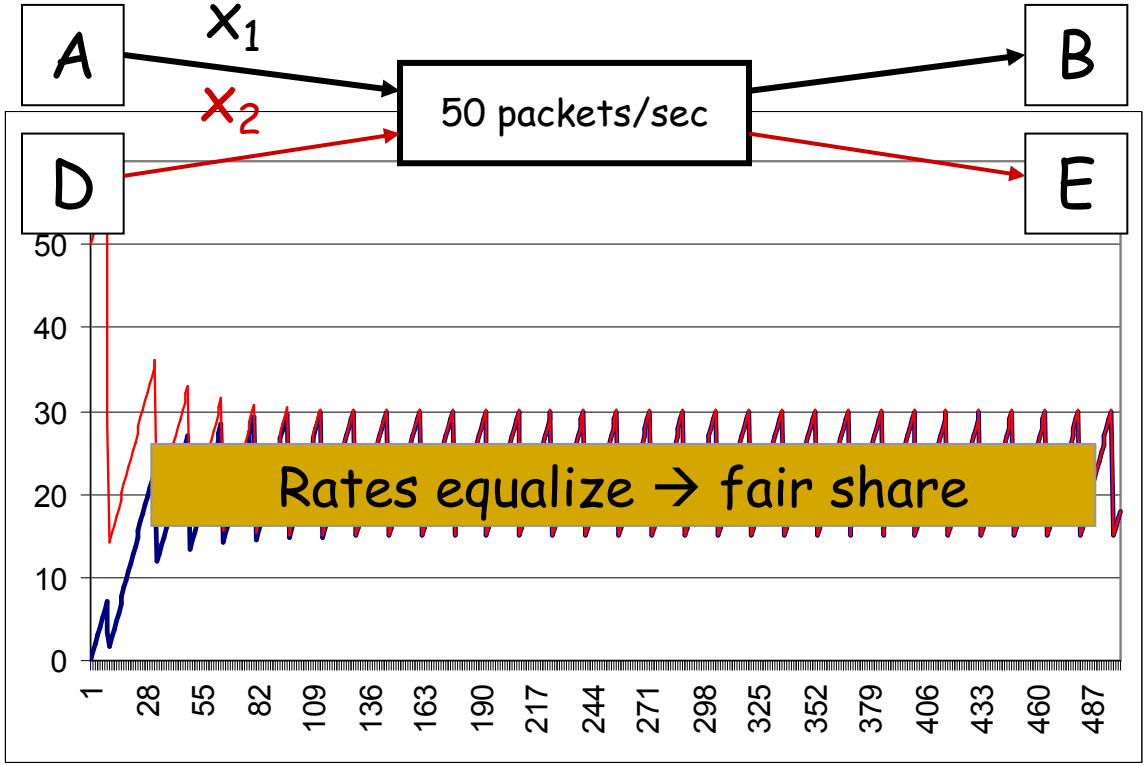


AIMD

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



AIMD Sharing Dynamics



Outline

- TCP flow control
- TCP congestion control
- TCP congestion control wrap-up
- Router assisted congestion control

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 dupACK
- Exit fast recovery after receiving new ACK
  - set CWND = ssthresh

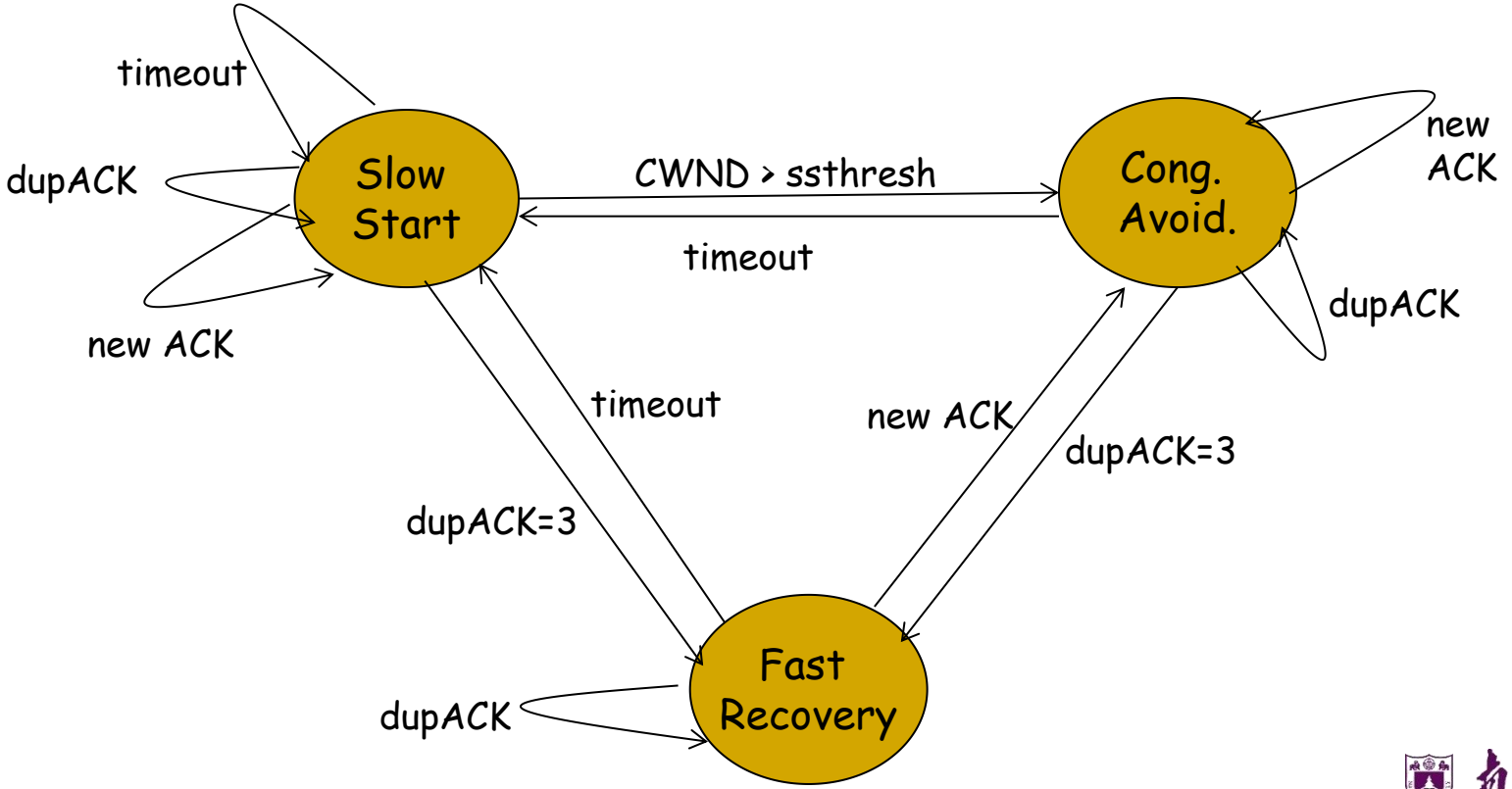
Example

- 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

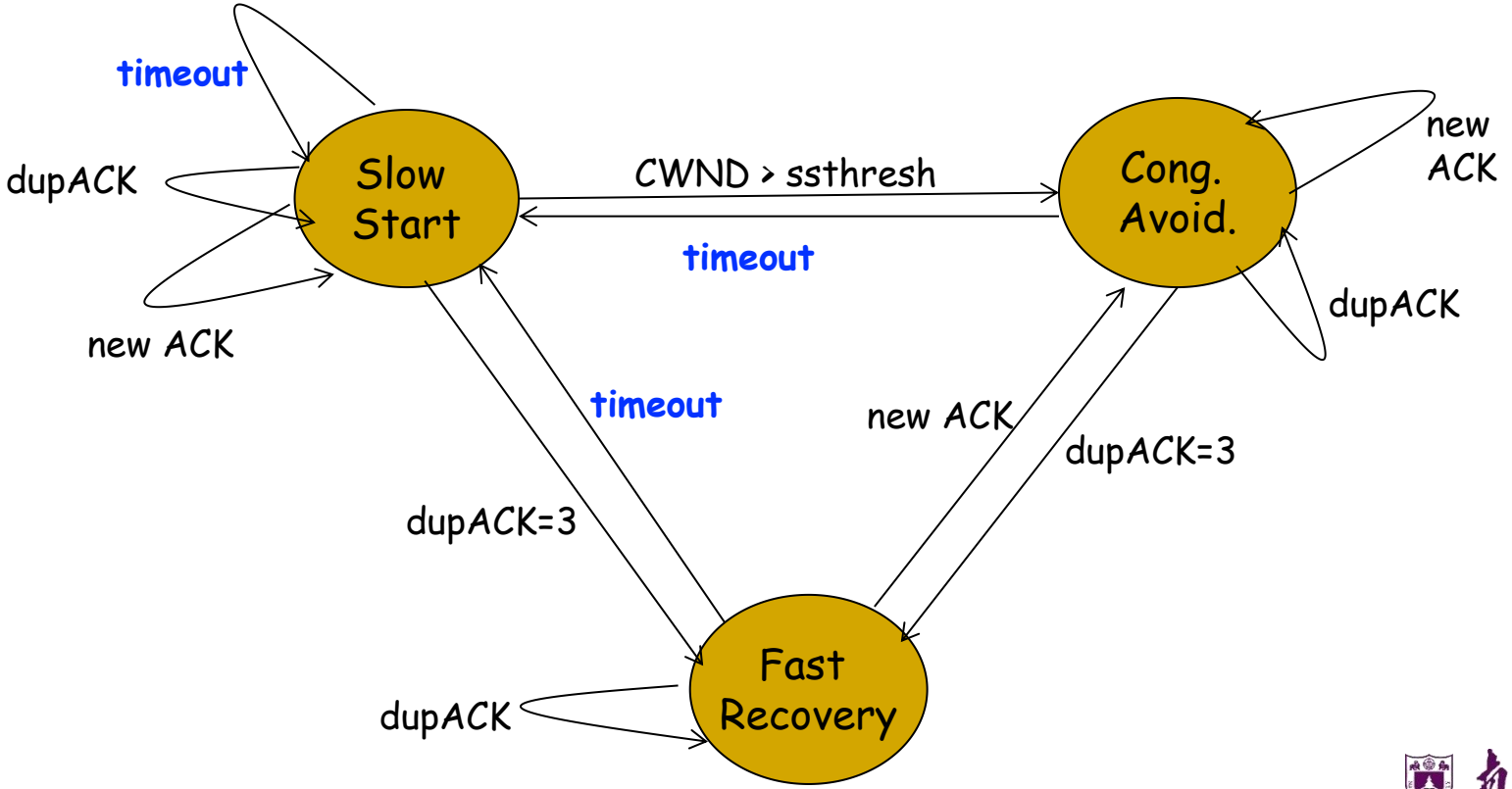
Timeline: [101, 102, ..., 110]

- 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
- RETRANSMIT 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 cong. avoidance

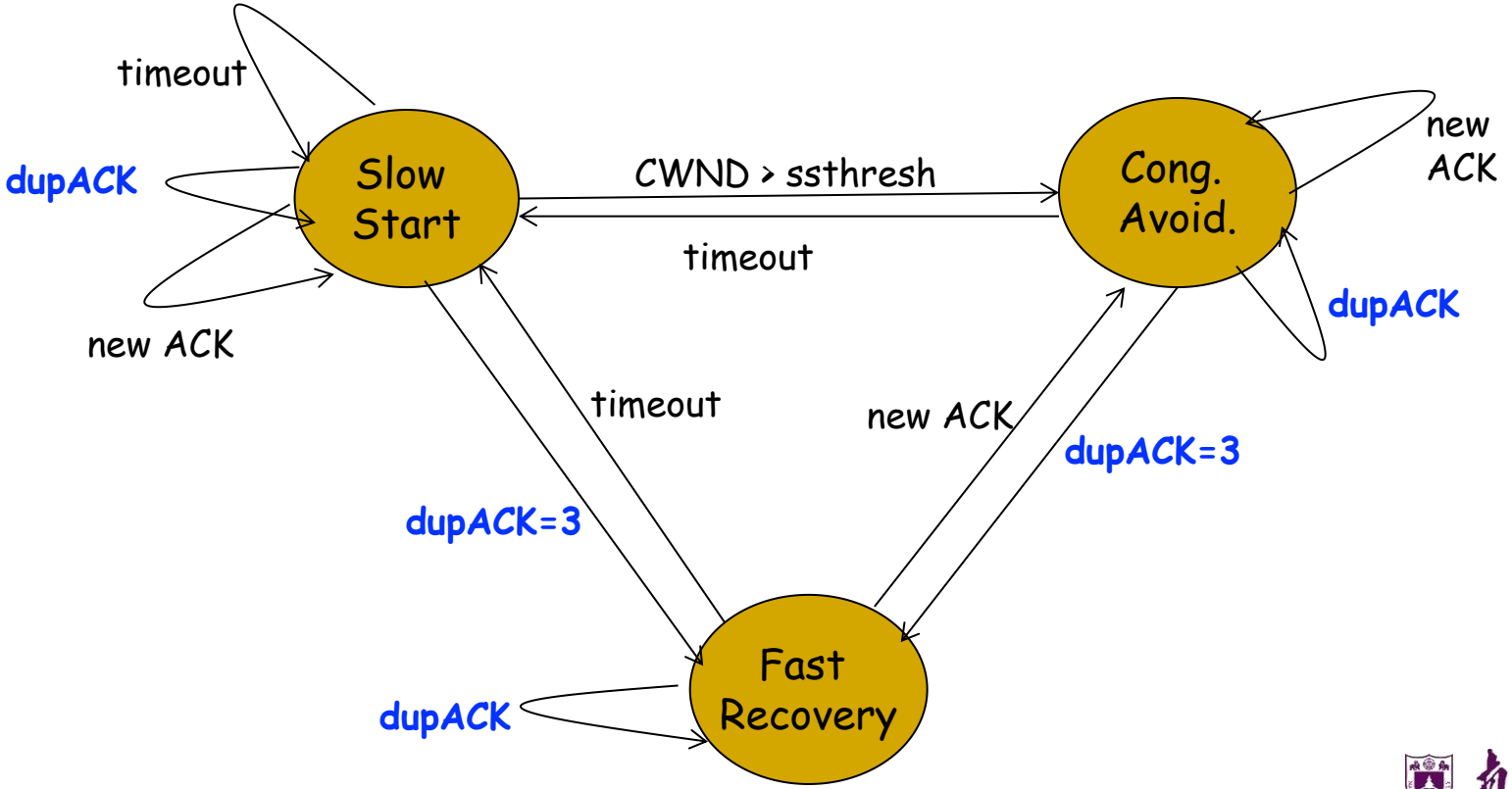
TCP state machine



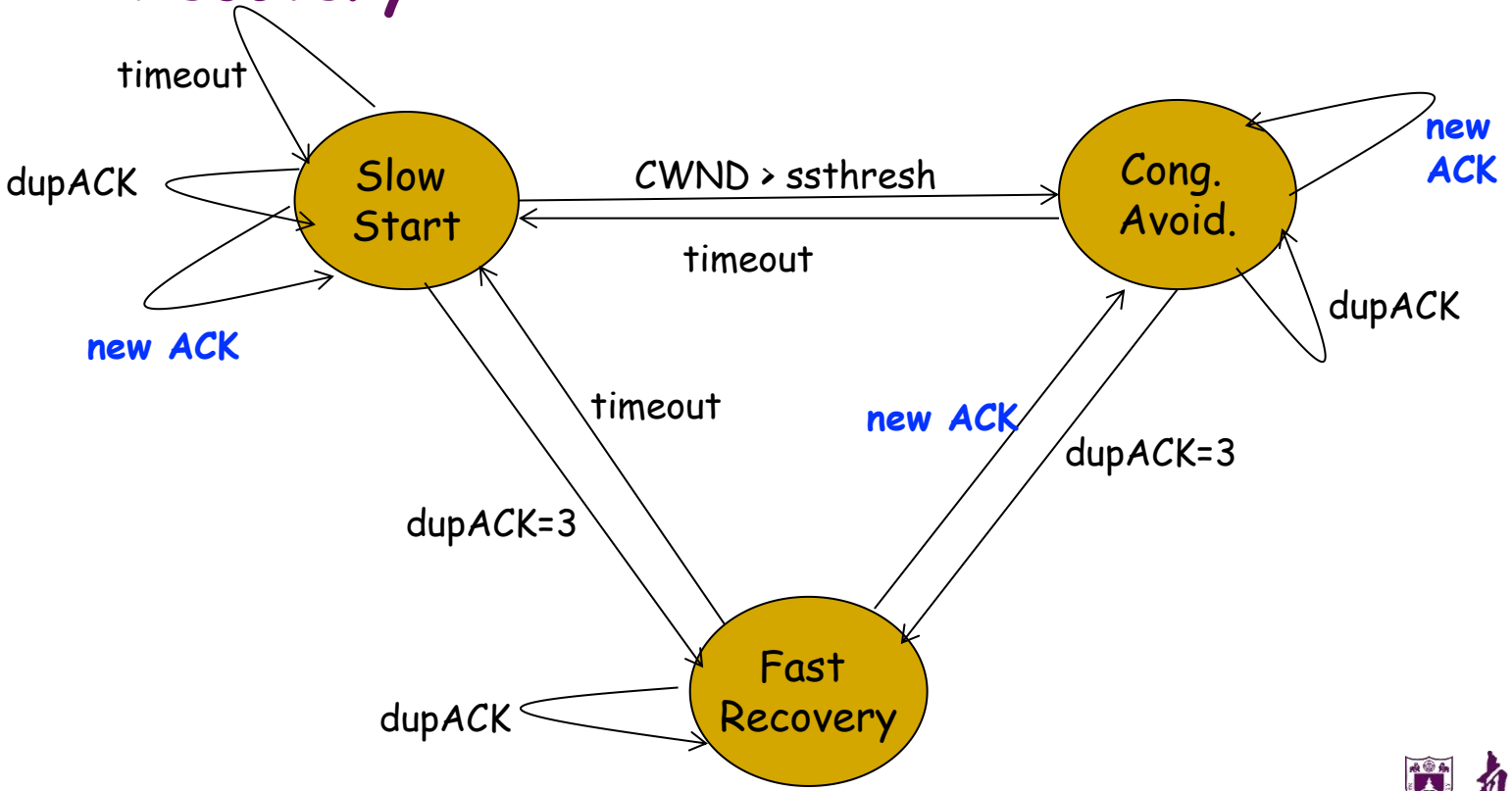
Timeouts ➔ Slow Start



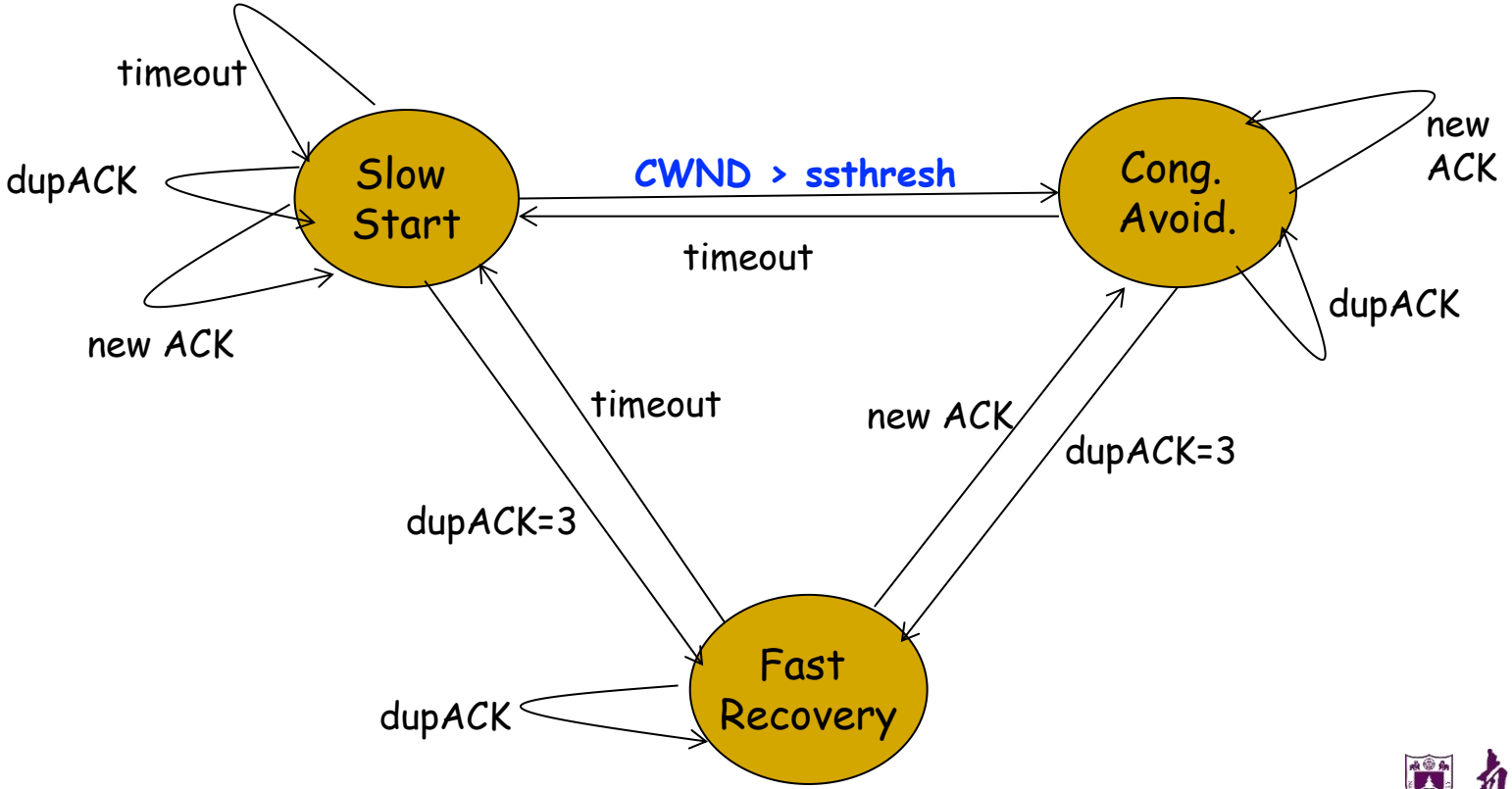
dupACKs ➔ Fast Recovery



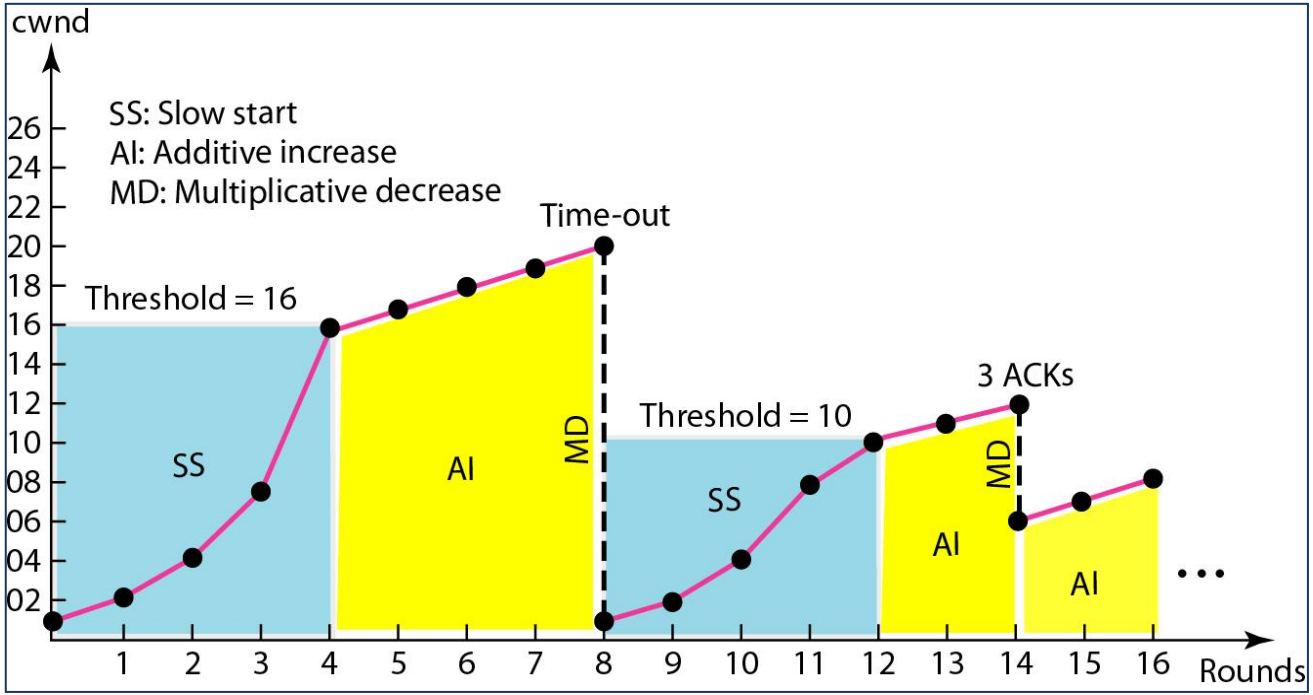
New ACK changes state ONLY from Fast Recovery



TCP state machine



Timeout and Dup-ack



TCP flavors

- TCP-Tahoe
  - CWND =1 on 3 dupACKs
- TCP-Reno
  - CWND =1 on timeout
  - CWND = CWND/2 on 3 dupACKs
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - Incorporates selective acknowledgements

Our default assumption

How can they coexist?

- All follow the same principle
  - Increase CWND on good news
  - Decrease CWND on bad news

Outline

- TCP flow control
- TCP congestion control
- TCP congestion control wrap-up
- Router assisted congestion control

Recap: TCP problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows complete before discovering available capacity
- AIMD impractical for high speed links
- Saw tooth discovery too choppy for some apps
- Unfair under heterogeneous RTTs
- Tight coupling with reliability mechanisms
- End-hosts can cheat

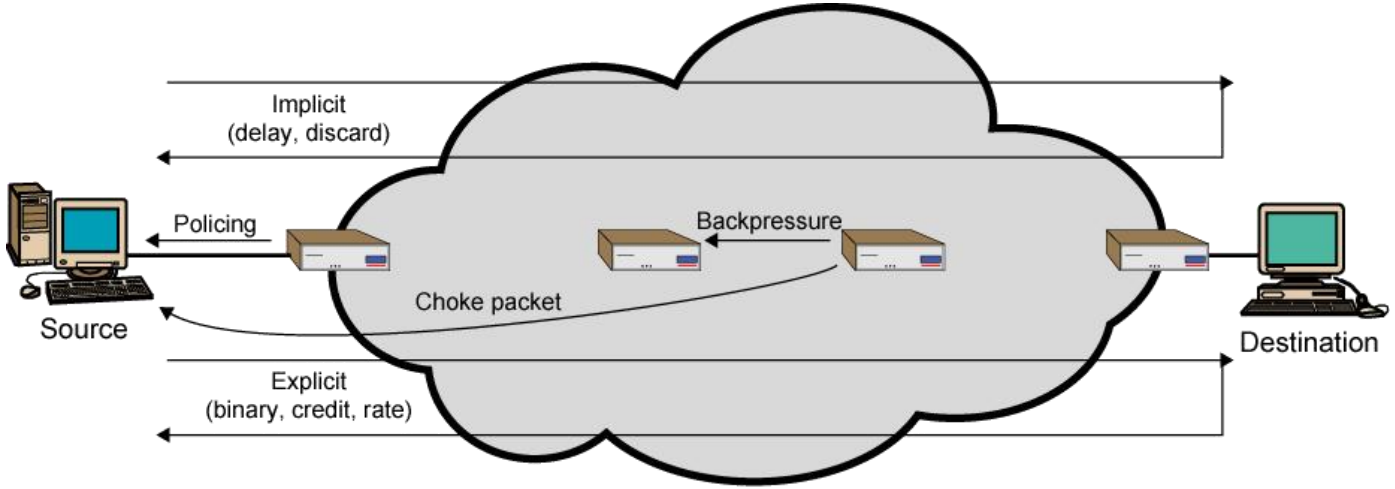
Routers tell endpoints if they're congested

Routers tell endpoints what rate to send at

Routers enforce fair sharing

Could fix many of these with some help from routers!

Mechanisms for Congestion Control



- Choke Packet
- Backpressure
- Warning bit
- Random early discard
- Fair Queuing (FQ)

- 抑制分组
- 反压
- 警告位
- 随机早期丢弃
- 公平队列

Warning Bit

Explicit Congestion Notification (ECN)

- Single bit in packet header; set by congested routers
  - If data packet has bit set, then ACK has ECN bit set
- Many options for when routers set the bit
  - Tradeoff between (link) utilization and (packet) delay
- Congestion semantics can be exactly like that of drop
  - i.e., end-host reacts as though it saw a drop

ECN

- Advantages:
  - Don't confuse corruption with congestion; recovery w/ rate adjustment
  - Can serve as an early indicator of congestion to avoid delays
  - Easy (easier) to incrementally deploy
    - ✓ Today: defined in RFC 3168 using ToS/DSCP bits in the IP header
    - ✓ Common in datacenters

提问

Q & A