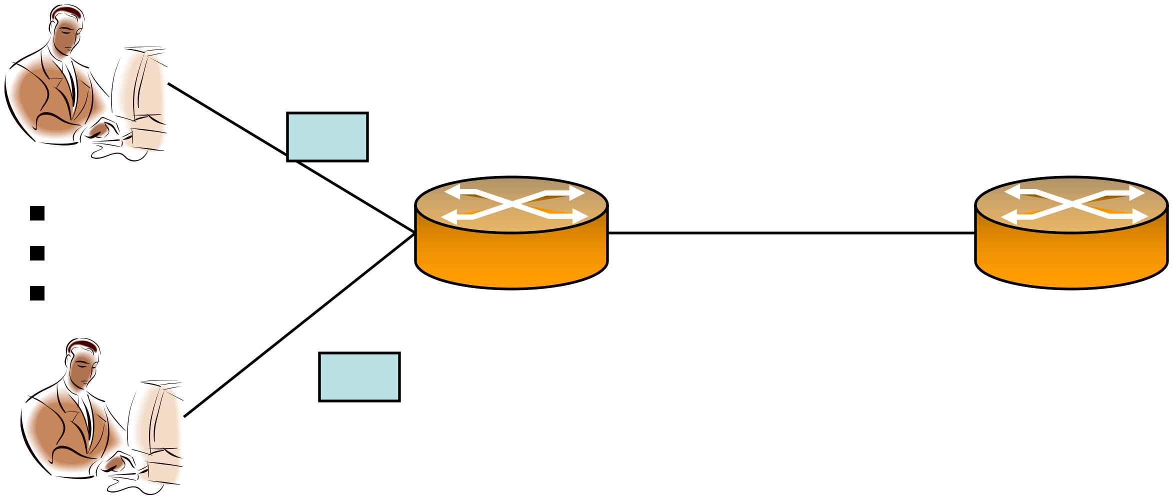
CompSci 514: Computer Networks
Lecture 5: Congestion Control

Xiaowei Yang

Outline

- Background on TCP congestion control
- The congestion control algorithm
- Theoretic background
- Macroscopic modeling

The Internet architecture



- Packet switching
- Statistical multiplexing
- Q: N users, and one network
  - How fast should each user send?
  - Why is it a difficult problem?

Background

- Original TCP (Cerf74)
  - Static window, no congestion control, no RTT estimation
  - In October of 86, the Internet had the first of what became a series of “congestion collapse”: increased load leads to decreased throughput
  - The NSFnet phase-I backbone dropped three orders of magnitude from its capacity of 32 kbit/s to 40 bit/s, and continued until end nodes started implementing Van Jacobson's congestion control between 1987 and 1988.

Possible explanations

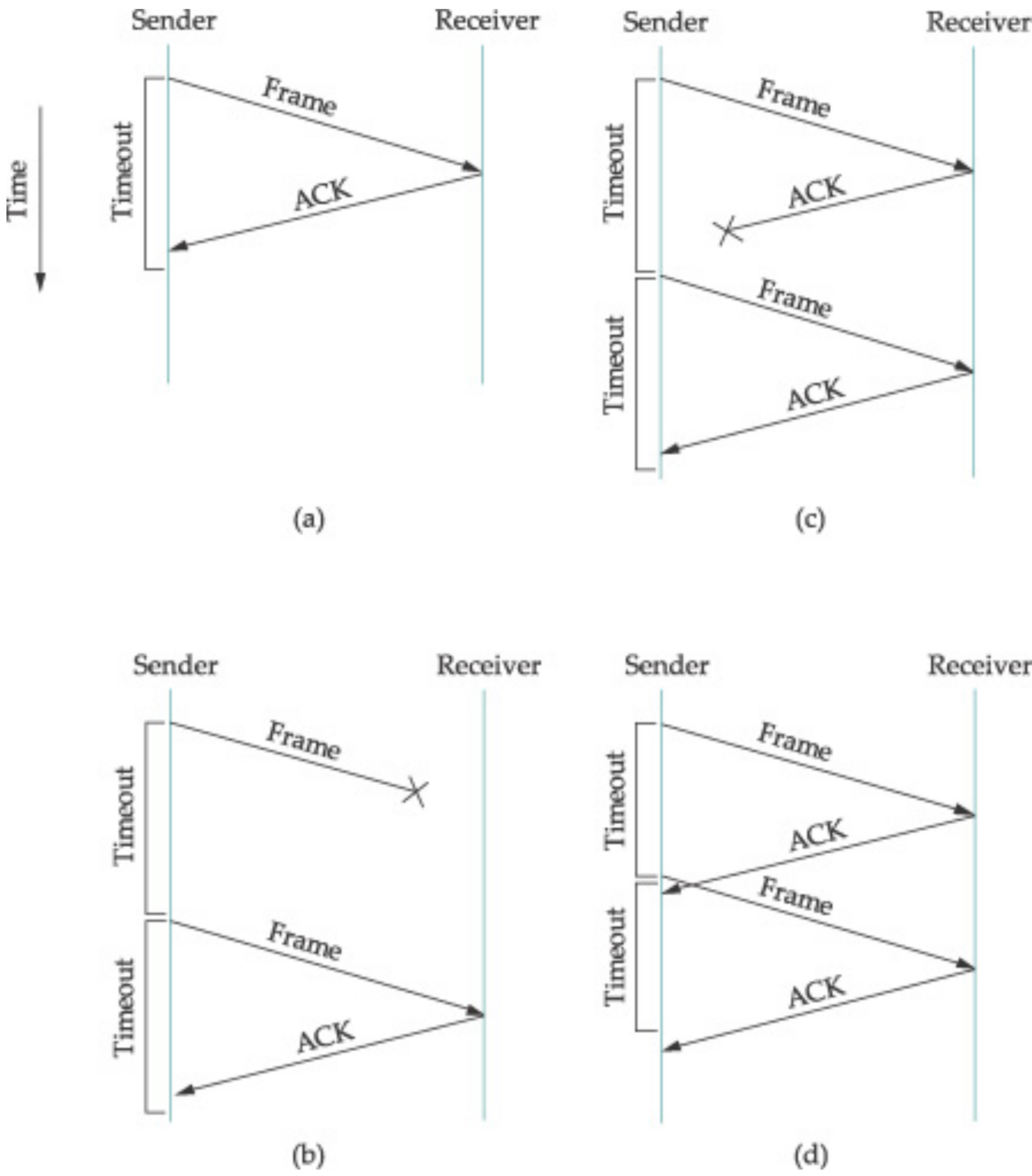
- Queueing delay increases
- TCP times out
- TCP retransmits too early, wasting the network’s bandwidth to retransmit the same packets already in transit and reducing useful throughput (goodput)

Review of TCP's sliding window algorithm

- A well-known algorithm in networking
- Used for
    - Reliable transmission
    - Flow control
    - Congestion control

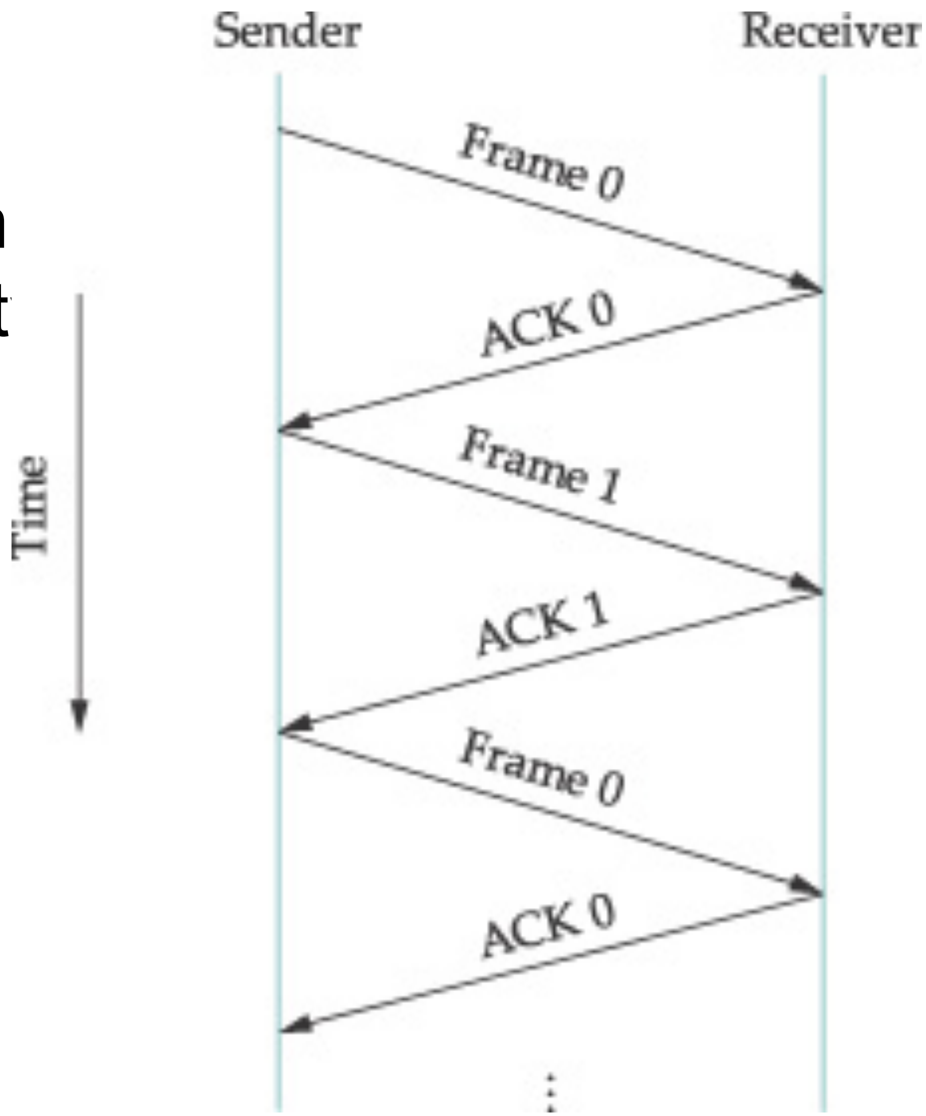
Stop-and-wait

- Send one frame, wait for an ack, and send the next
- Retransmit if times out
- Note in the last figure (d), there might be confusion: a new frame, or a duplicate?



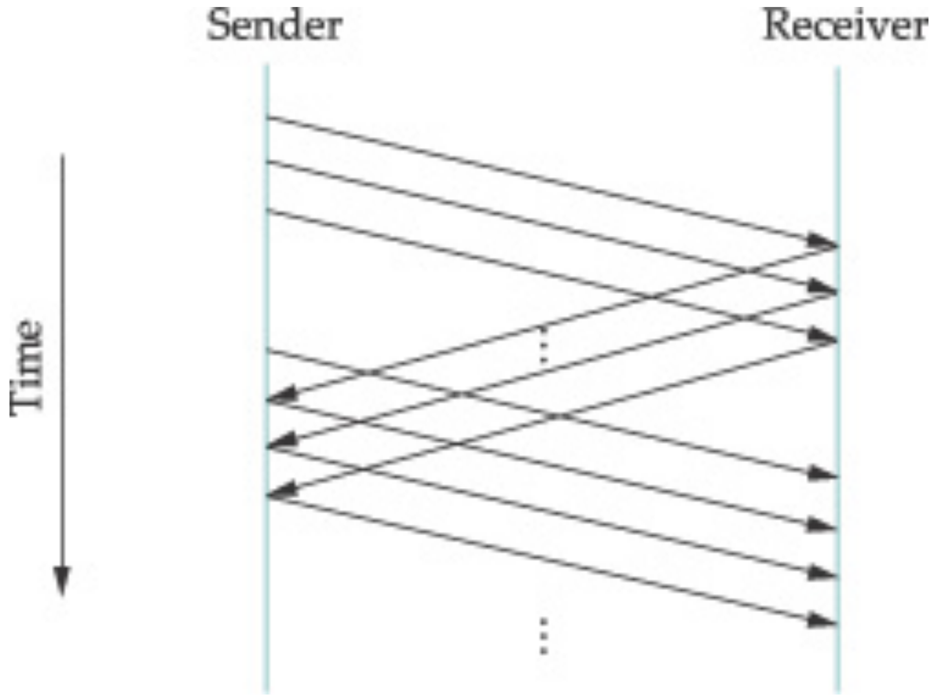
Sequence number

- Add a sequence number to each fram to avoid the ambiguit



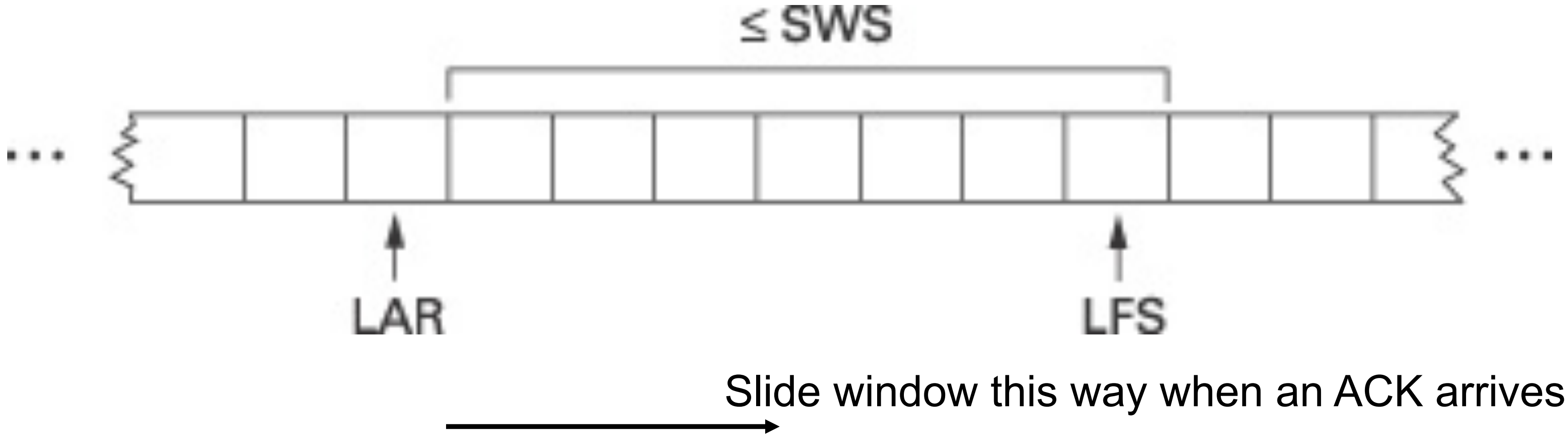
Sliding window

- Stop-and-wait: too slow
- Key idea: allowing multiple outstanding (unacked) frames to keep the pipe full



Sliding window on sender

- Assign a sequence number (SeqNum) to each frame
- Maintains three variables
  - Send Window Size (SWS)
  - Last Ack Received (LAR)
  - Last Frame Sent (LFS)
- Invariant: LFS – LAR ≤ SWS



Slide window this way when an ACK arrives

- Sender actions
  - When an ACK arrives, moves LAR to the right, opening the window to allow the sender to send more frames
  - If a frame times out before an ACK arrives, retransmit

Sliding window on receiver for flow control

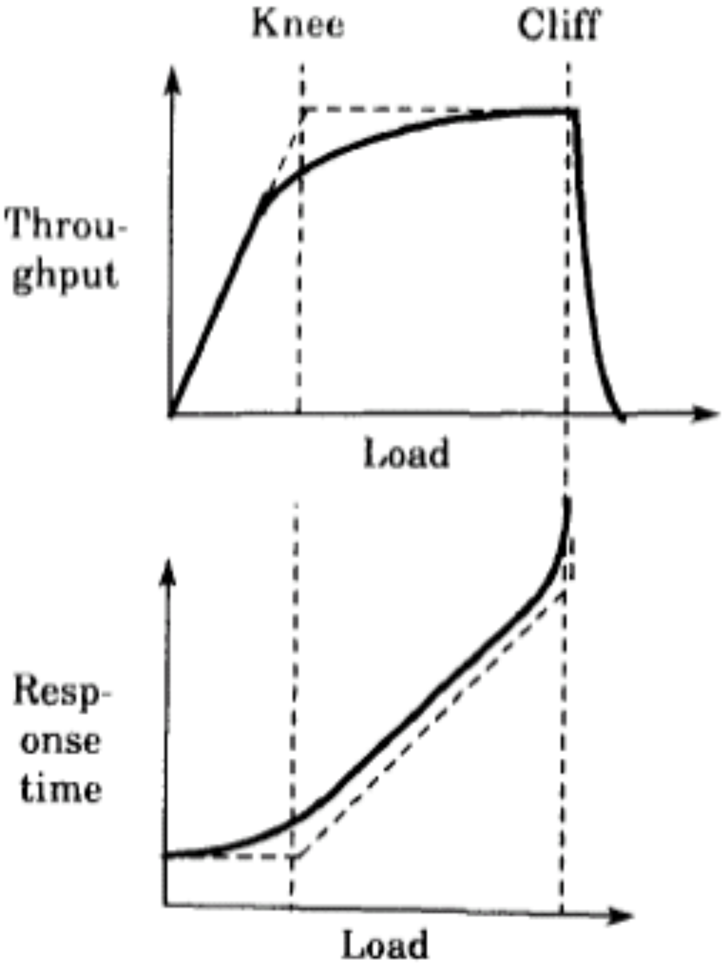
- Maintains three window variables
  - Receive Window Size (RWS)
  - Largest Acceptable Frame (LAF)
  - Last frame received (LFR)
- Invariant
  - LAF – LFR ≤ RWS

- When a frame with SeqNum arrives
  - – Discards it if out of window
    - Seq ≤ LFR or Seq > LAF
  - – If in window, decides what to ACK
    - Cumulative ack
    - Acks SeqNumToAck even if higher-numbered packets have been received
    - Sets LFR = SeqNumToAck-1, LAF = LFR + RWS
    - Updates SeqNumToAck

Drawbacks of static window sizes

- One TCP flow
  - MSS = 512 bit, Window size = 10, RTT = 100ms
- 10 TCP flows, 100 TCP flows, …

Design Goals



- Congestion avoidance: making the system operate around the knee to obtain low latency and high throughput
- Congestion control: making the system operate left to the cliff to avoid congestion collapse

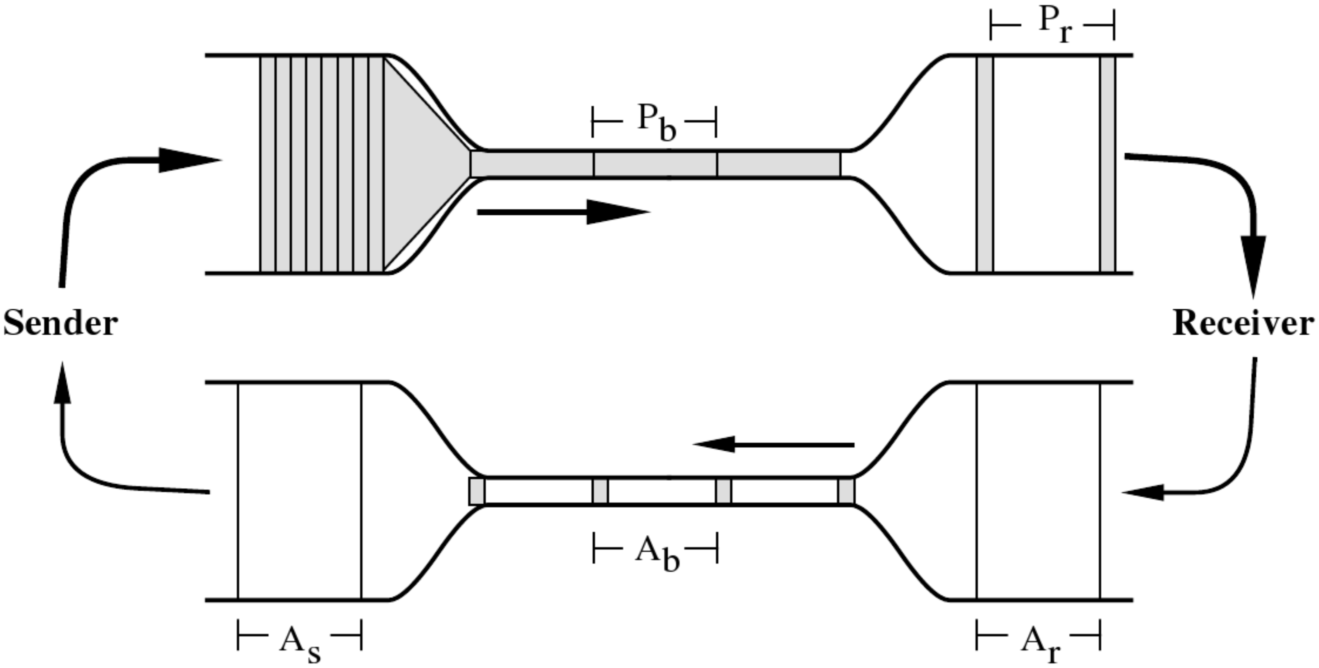
Key Improvements from the TCP88 paper

- RTT variance estimate
  - Old design: $RTT_{n+1} = \alpha\ RTT + (1- \alpha)\ RTT_n$
  - $RTO = \beta\ RTT_{n+1}$
- Exponential backoff
- Slow-start
- Dynamic window sizing
- Fast retransmit

Challenge

- Send at the “right” speed
  - Fast enough to keep the pipe full
  - But not to overrun the “pipe”
    - Drawback?
  - Share nicely with other senders

Key insight: packet conservation principle and self-clocking



- When pipe is full, the speed of ACK returns equals to the speed new packets should be injected into the network

Solution: Dynamic window sizing

- Sending speed: SWS / RTT

- → Adjusting SWS based on available bandwidth

- The sender has two *internal* parameters:
  - **Congestion Window** (**cwnd**)
  - **Slow-start threshold Value** (**ssthresh)**

- SWS is set to the minimum of (cwnd, receiver advertised win)

Two Modes of Congestion Control

1. Probing for the available bandwidth
   - **slow start** (cwnd < ssthresh)

2. Avoid overloading the network
   - **congestion avoidance** (cwnd >= ssthresh)

Slow Start

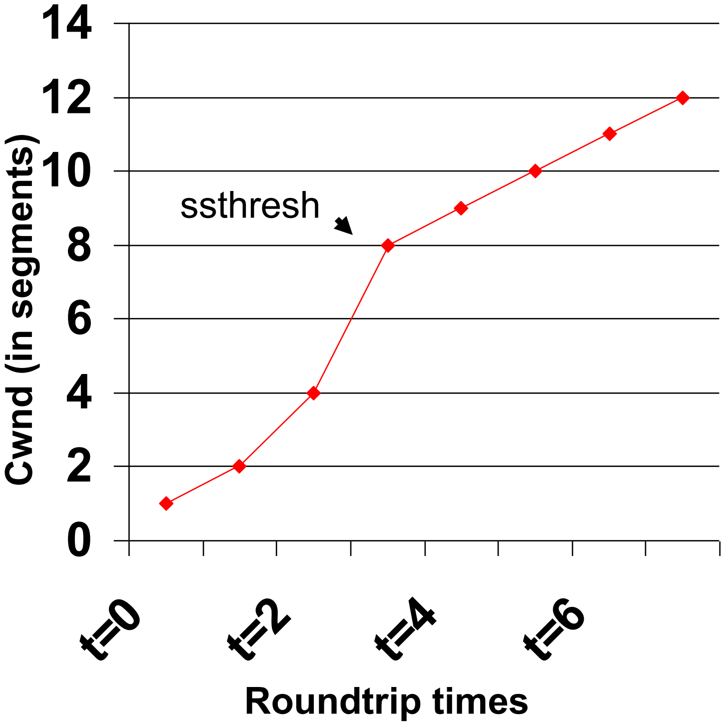
- Initial value: Set **cwnd = 1 MSS**
  - Modern TCP implementation may set initial cwnd to a much larger value
- When receiving an ACK, cwnd+= 1 MSS

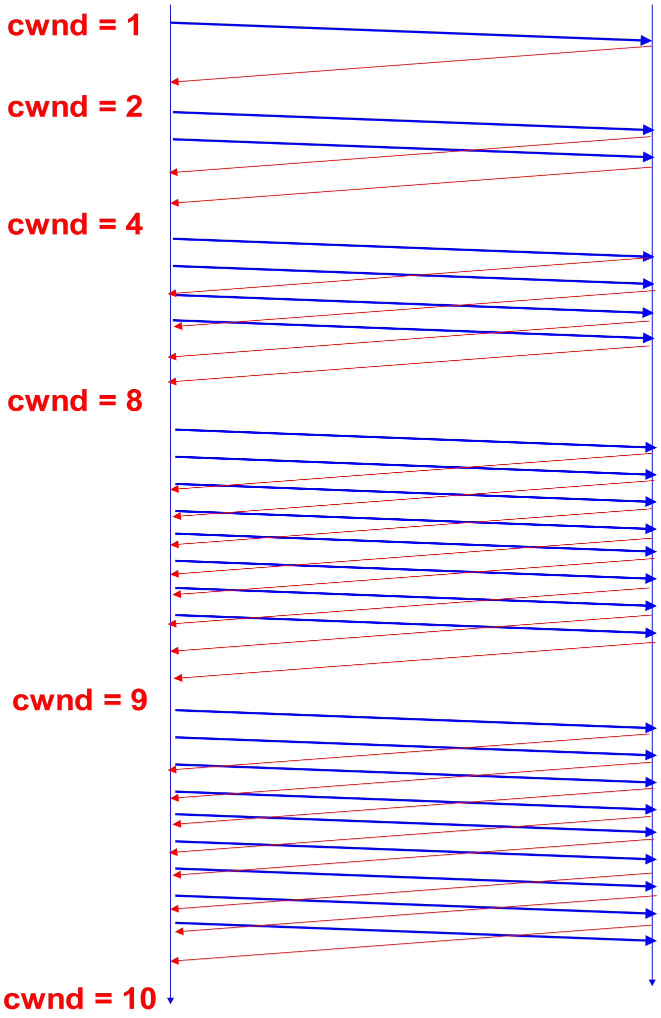
Congestion Avoidance

- If cwnd >= ssthresh then each time an ACK is received, increment cwnd as follows:
  - cwnd += MSS * (MSS / cwnd) (cwnd measured in bytes)
- So *cwnd* is increased by one MSS only if all *cwnd*/MSS segments have been acknowledged.

Example of Slow Start/Congestion Avoidance

Assume *ssthresh = 8 MSS*





Congestion detection

- What would happen if a sender keeps increasing cwnd?
  - Packet loss

- TCP uses packet loss as a congestion signal

- Loss detection
  1. Receipt of a duplicate ACK (cumulative ACK)
  2. Timeout of a retransmission timer

Reaction to Congestion

- Reduce cwnd

- Timeout: severe congestion
  - cwnd is reset to one MSS:

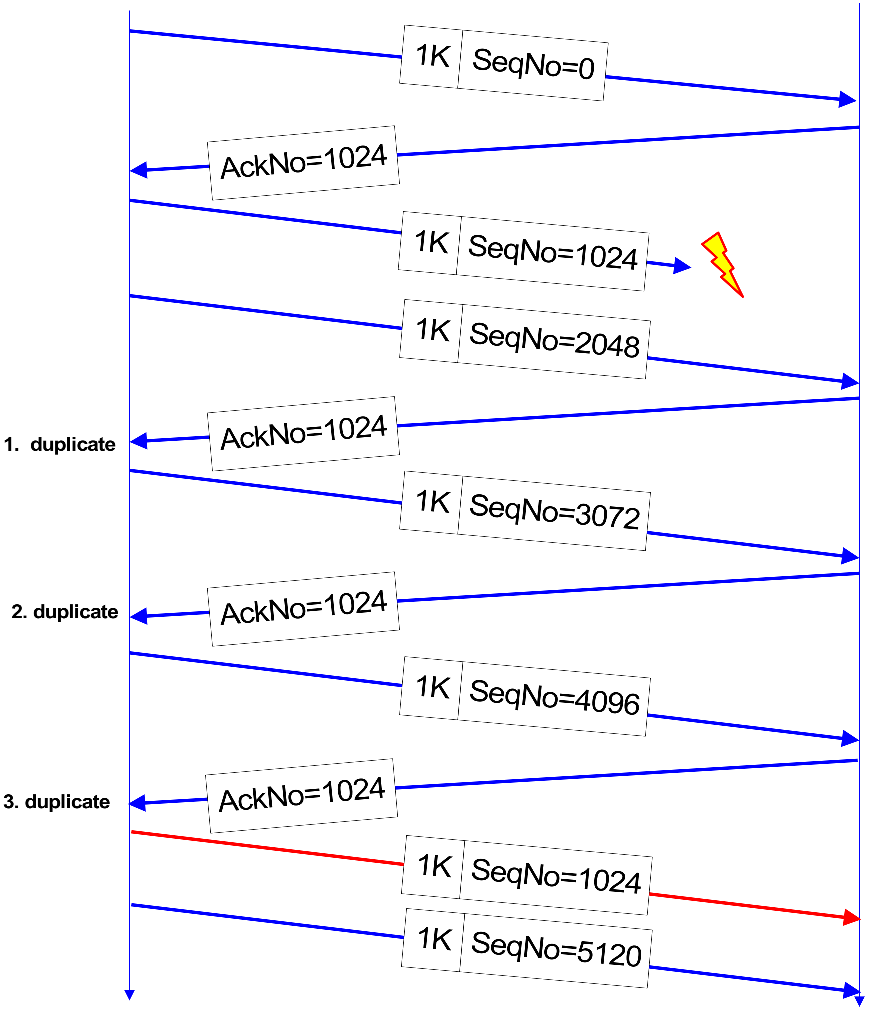
    cwnd = 1 MSS
  - ssthresh is set to half of the current size of the congestion window:

    ssthressh = cwnd / 2
  - entering slow-start

Reaction to Congestion

- Duplicate ACKs: not so congested (why?)
- Fast retransmit
  - Three duplicate ACKs indicate a packet loss
  - Retransmit without timeout

Duplicate ACK example



Reaction to congestion: Fast Recovery

- Avoiding slow start (changed from TCP88)
  - ssthresh = cwnd/2
  - cwnd = cwnd+3MSS
  - Increase cwnd by one MSS for each additional duplicate ACK

- When ACK arrives that acknowledges “new data,” set:

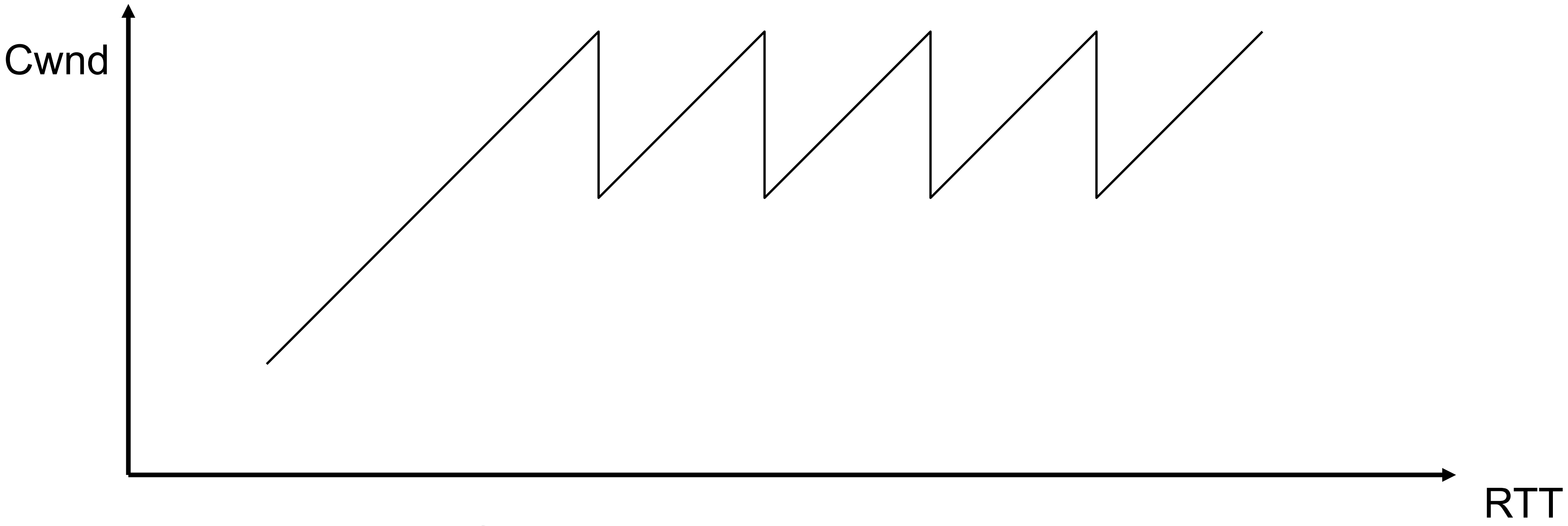
  cwnd=ssthresh

  enter congestion avoidance

Flavors of TCP Congestion Control

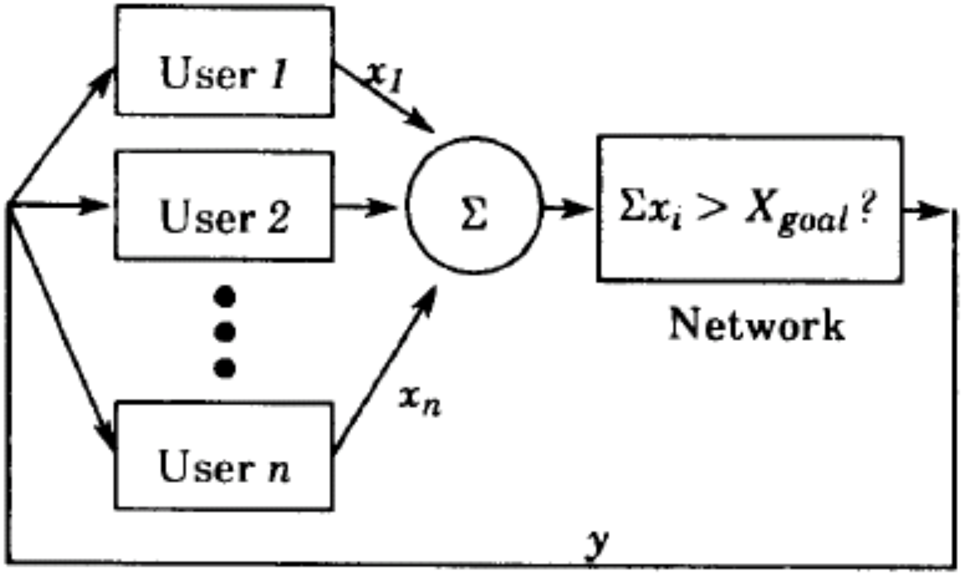
- **TCP Tahoe** (1988, FreeBSD 4.3 Tahoe)
  - Slow Start
  - Congestion Avoidance
  - Fast Retransmit
- **TCP Reno** (1990, FreeBSD 4.3 Reno)
  - Fast Recovery
  - Modern TCP implementation
- **New Reno** (1996)
- **SACK** (1996)
- **TCP CUBIC** (Current linux and window TCP)

The Sawtooth behavior of TCP



- For every ACK received
  - Cwnd += 1/cwnd
- For every packet lost
  - Cwnd /= 2

Why does it work? [Chiu-Jain]



- A feedback control system
- The network uses feedback y to adjust users’ load ∑x_i

Goals of Congestion Avoidance

- Efficiency: the closeness of the total load on the resource ot its knee
- Fairness:

$$F(x) = \frac{(\Sigma x_i)^2}{n(\Sigma x_i^2)}.$$

  - When all x_i’s are equal, F(x) = 1
  - When all x_i’s are zero but x_j = 1, F(x) = 1/n

- Distributedness
  - A centralized scheme requires complete knowledge of the state of the system
- Convergence
  - The system approach the goal state from any starting state

Metrics to measure convergence

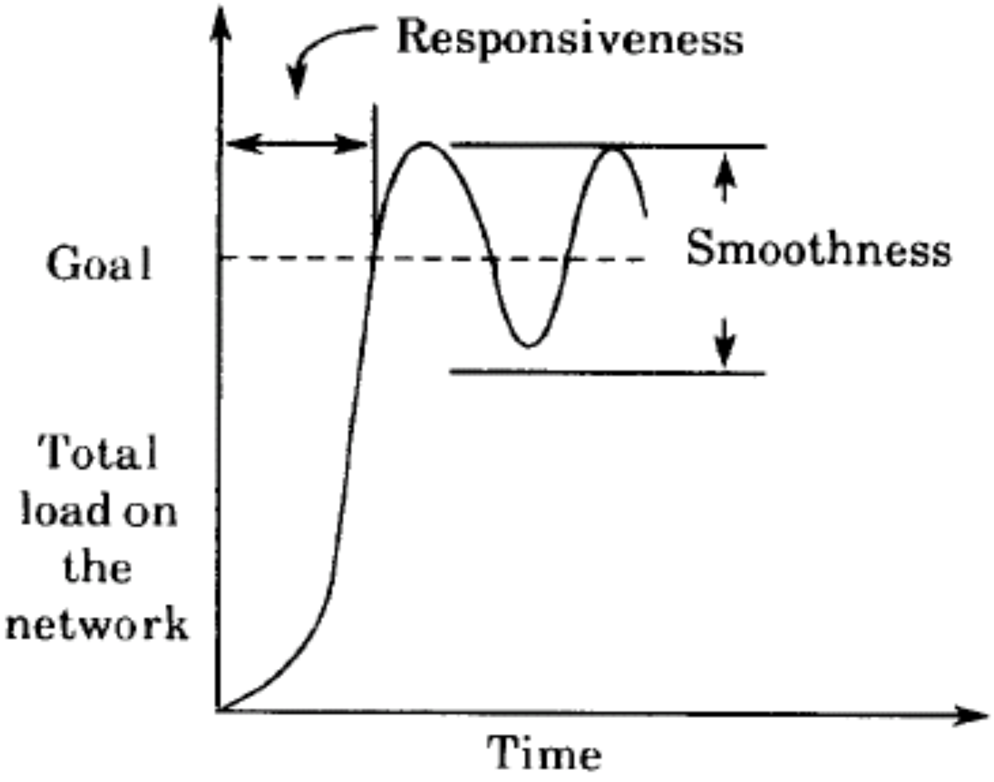


Fig. 3. Responsiveness and smoothness.

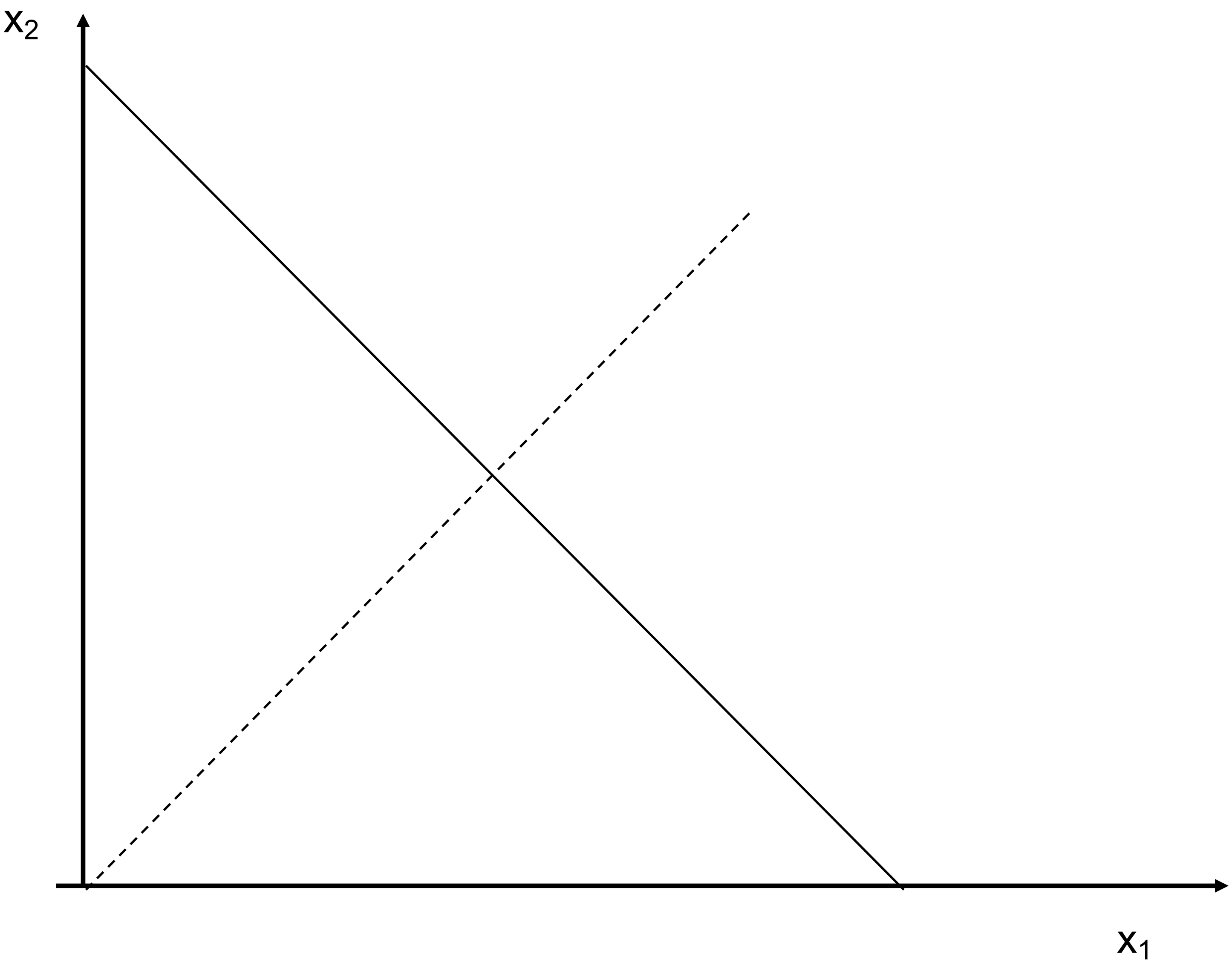
- Responsiveness
- Smoothness

Model the system as a linear control system

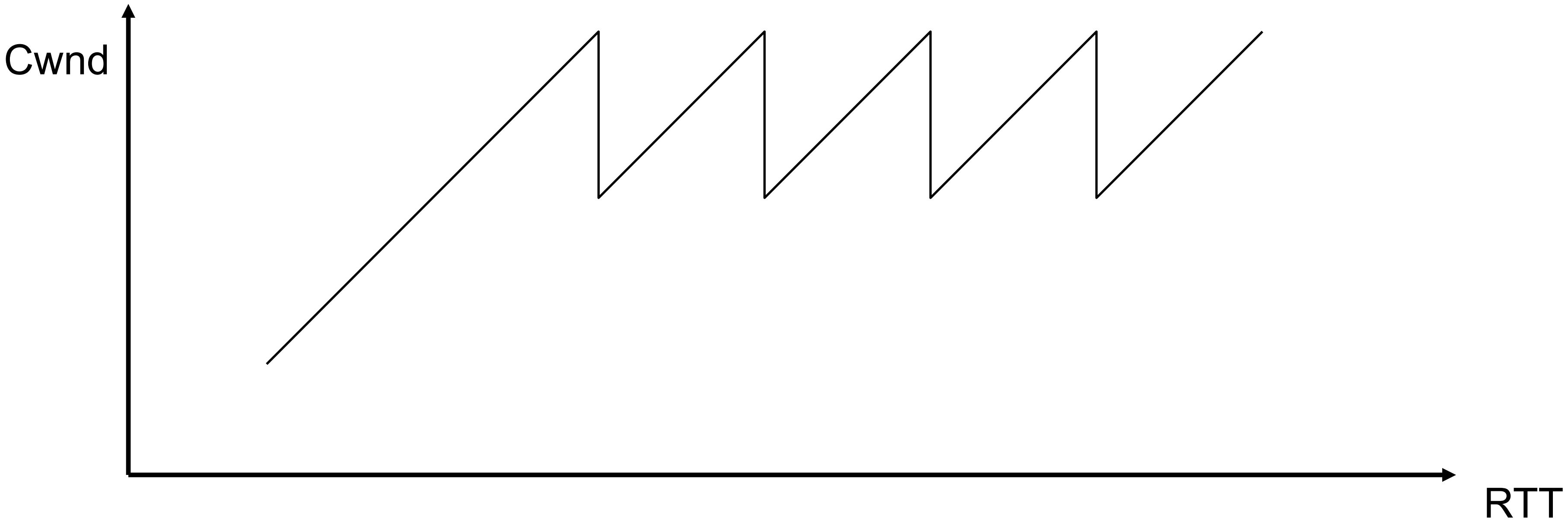
$$x_i(t+1) = \begin{cases} a_{\mathrm{I}} + b_{\mathrm{I}} x_i(t) & \text{if } y(t) = 0 \Rightarrow \text{Increase,} \\ a_{\mathrm{D}} + b_{\mathrm{D}} x_i(t) & \text{if } y(t) = 1 \Rightarrow \text{Decrease.} \end{cases}$$

- Four sample types of controls
- AIAD, AIMD, MIAD, MIMD

Phase plot



TCP congestion control is AIMD



- Problems:
  - – Each source has to probe for its bandwidth
  - – Congestion occurs first before TCP backs off
  - – Unfair: long RTT flows obtain smaller bandwidth shares

Macroscopic behavior of TCP

- Throughput is inversely proportional to RTT:

$$\frac{\sqrt{1.5} \bullet MSS}{RTT \bullet \sqrt{p}}$$

- In a steady state, total packets sent in one sawtooth cycle:
  - $S = w + (w+1) + \ldots (w+w) = 3/2\ w^2$
- the maximum window size is determined by the loss rate
  - $1/S = p$
  - $w = \frac{1}{\sqrt{1.5p}}$
- The length of one cycle: w * RTT
- Average throughput: 3/2 w * MSS / RTT

Conclusion

- Congestion control is one of the fundamental issues in networking
- TCP congestion control algorithm
- The AIMD algorithm
- TCP macroscopic behavior model