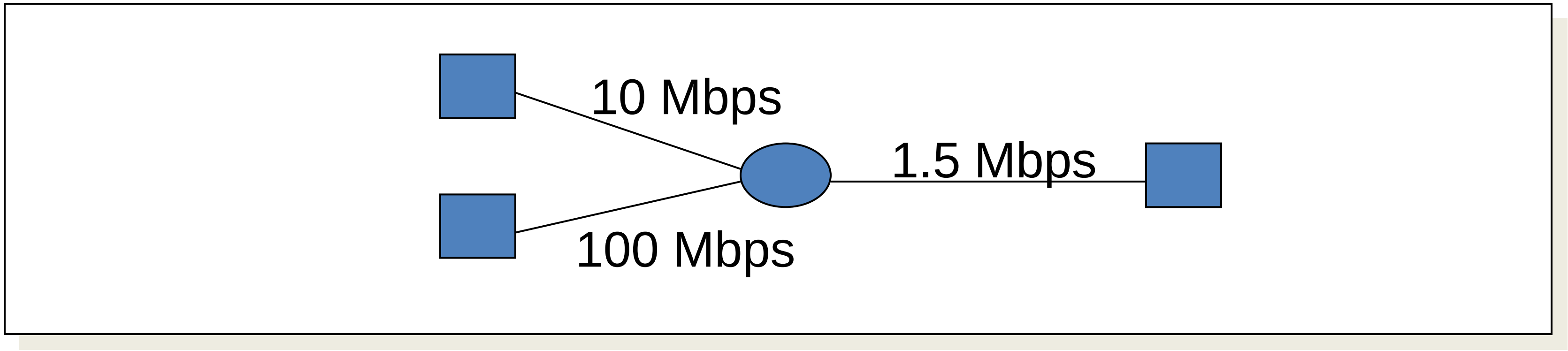
Congestion: Tragedy of Commons

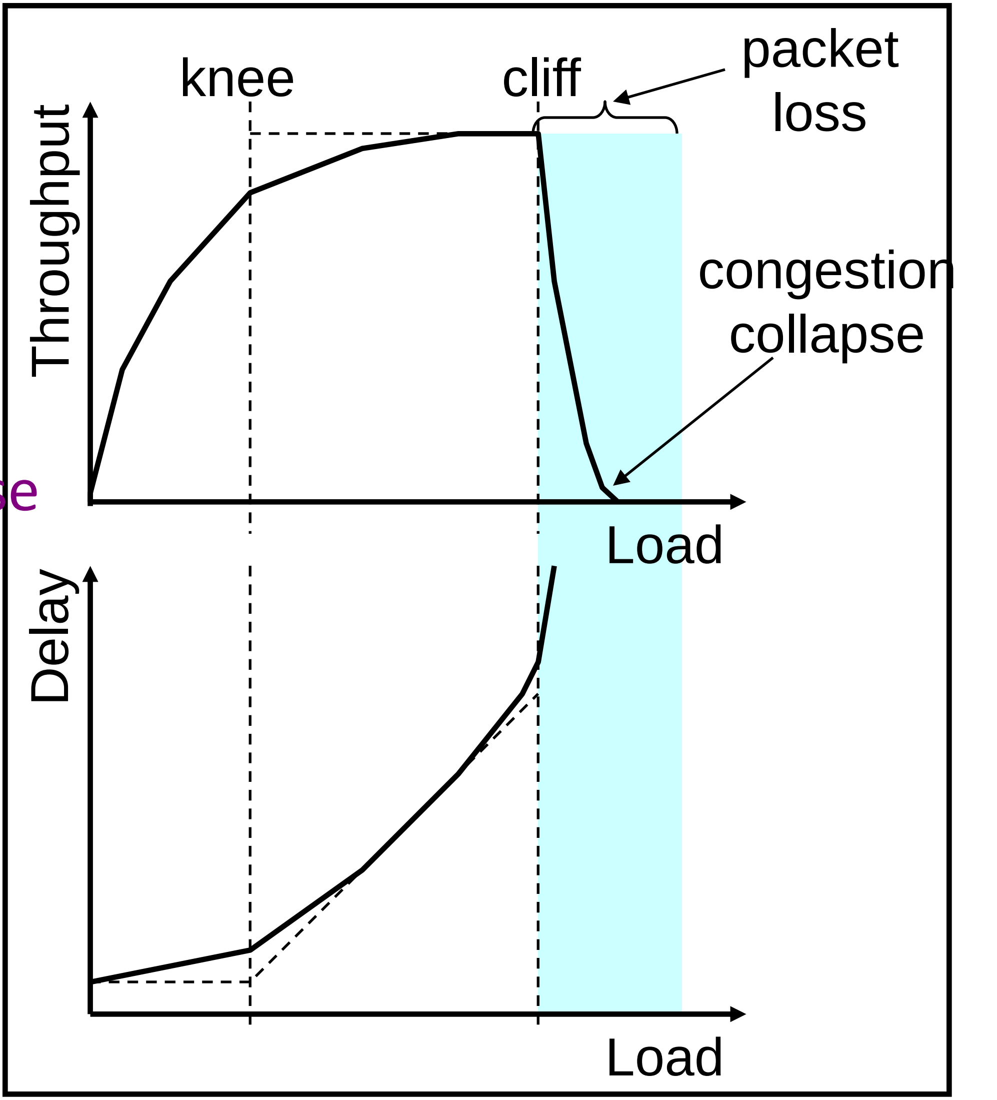


- Different sources compete for "common" or "shared" resources inside network.
  - Sources are unaware of current state of resource
  - Sources are unaware of each other
  - Source has self-interest. Assumes that increasing rate by N% will lead to N% increase in throughput!
  - Conflicts with collective interests: if all sources do this to drive the system to overload, throughput gain is NEGATIVE, and worsens rapidly with incremental overload => congestion collapse!!
  - Need "enlightened" self-interest!

Congestion: A Close-up View

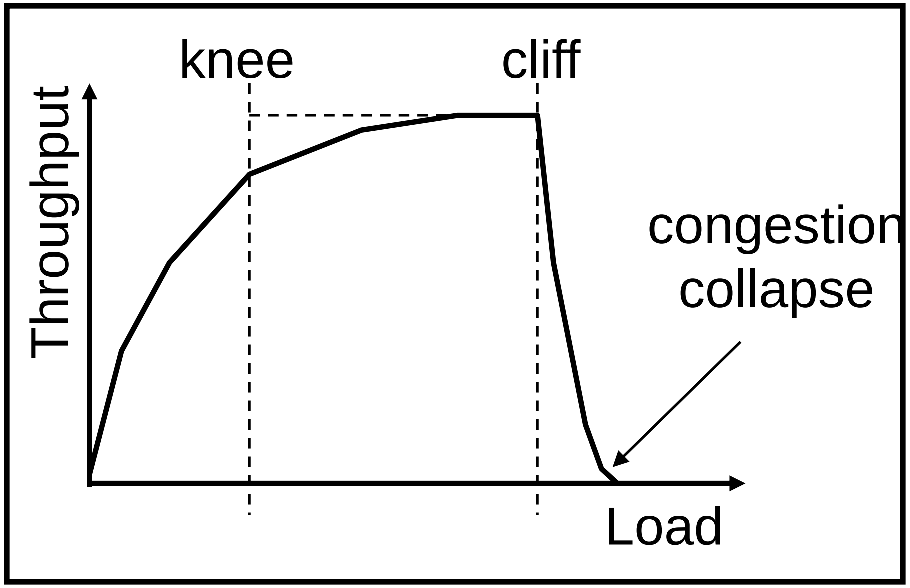
- knee – point after which
  - throughput increases very slowly
  - delay increases fast
- cliff – point after which
  - throughput starts to decrease very fast to zero (congestion collapse)
  - delay approaches infinity

- Note (in an M/M/1 queue)
  - delay = 1/(1 – utilization)



Congestion Control vs. Congestion Avoidance

- Congestion control
  - goal is to stay left of cliff
- Congestion avoidance
  - Goal is to stay left of knee
- Congestion collapse
  - Right of cliff:



Congestion Collapse

- Definition: *Increase in network load results in decrease of useful work done*
- Many possible causes
  - Spurious retransmissions of packets still in flight
  - Undelivered packets
    - Packets consume resources and are dropped elsewhere in network
  - Fragments
    - Mismatch of transmission and retransmission units
  - Control traffic
    - Large percentage of traffic is for control
  - Stale or unwanted packets
    - Packets that are delayed on long queues

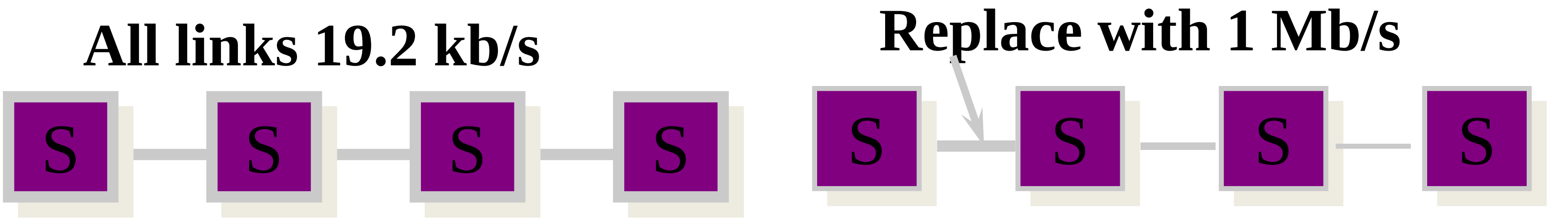
Static solutions…

- Q: Will the "congestion" problem be solved when:

a) Memory becomes cheap (infinite memory)?

Static solutions...

- *b) Links become cheap (high speed links)?*

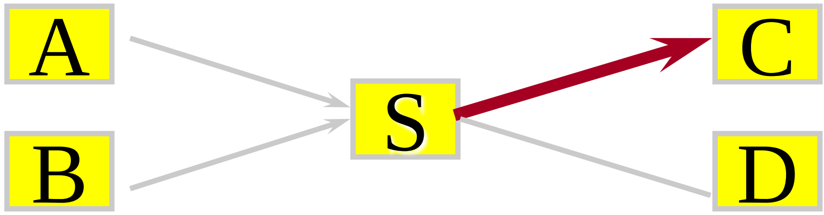


File Transfer time = ***5 mins***

File Transfer Time = ***7 hours***

Static solutions (Continued)

- c) *Processors* become cheap (fast routers & switches)



high-speed

Two models of congestion control

- 1. End-to-end model:
  - End-systems is ultimately the source of “demand”
  - End-system must robustly estimate the timing and degree of congestion and reduce its demand appropriately
  - Must trust other end hosts to do right thing
  - Intermediate nodes relied upon to send timely and appropriate penalty indications (eg: packet loss rate) during congestion
    - Enhanced routers could send more accurate congestion signals, and help end-system avoid other side-effects in the control process (eg: early packet marks instead of late packet drops)

Two models of congestion control…

- 2. Network-based model:
  - A) All end-systems cannot be trusted and/or
  - B) The network node has more control over isolation/scheduling of flows
  - Assumes network nodes can be trusted.
  - Each network node implements isolation and fairness mechanisms (eg: scheduling, buffer management)
- Problems:
  - <u>Partial soln:</u> if flows don't back off, each flow has congestion collapse, i.e. lousy throughput during overload
  - Significant complexity in network nodes
  - If some routers do not support this complexity, congestion still exists

Goals of Congestion Control

- To guarantee stable operation of packet networks
  - Sub-goal: avoid congestion collapse

- To keep networks working in an efficient status
  - Eg: high throughput, low loss, low delay, and high utilization

- To provide fair allocations of network bandwidth among competing flows in steady state
  - For some value of “fair” ☺

Basic Control Model

- Let’s assume window-based operation
- Reduce window when congestion is perceived
  - How is congestion signaled?
    - Either mark or drop packets
  - When is a router congested?
    - Drop tail queues – when queue is full
    - Average queue length – at some threshold
- Increase window otherwise
  - Probe for available bandwidth – how?

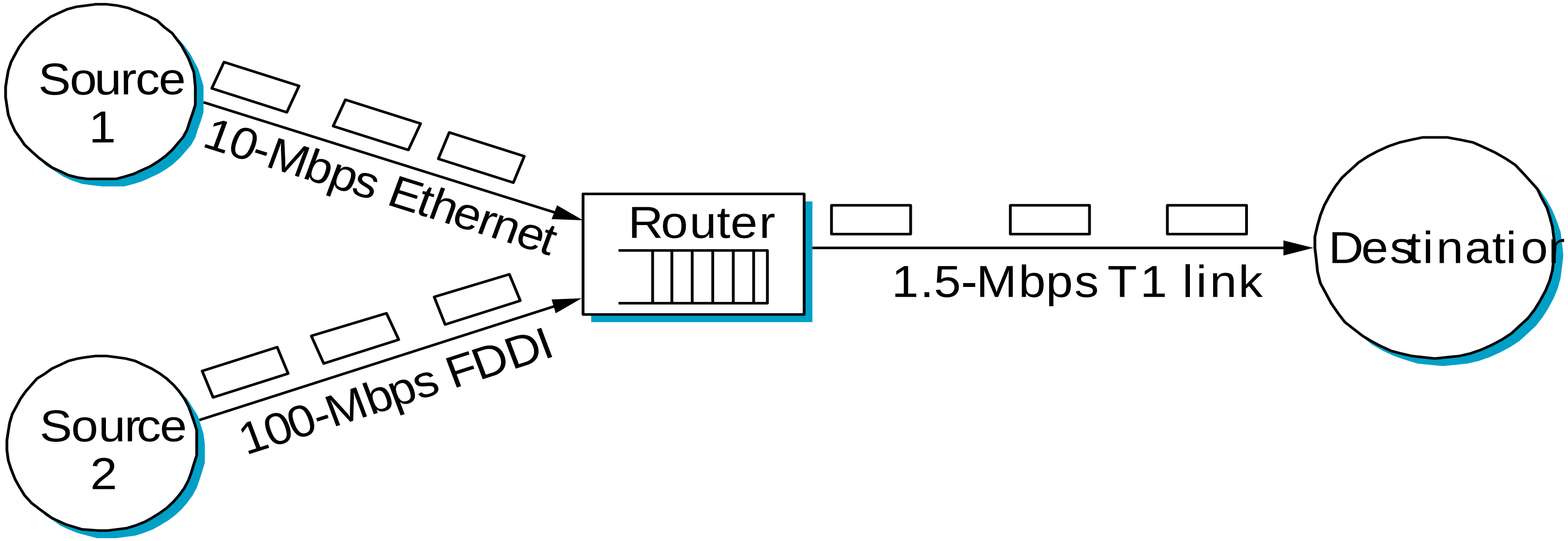
Congestion Control
and
Resource Allocation

Issues

Two sides of the same coin

– pre-allocate resources so at to avoid congestion

– control congestion if (and when) is occurs



Two points of implementation

– hosts at the edges of the network (transport protocol)

– routers inside the network (queuing discipline)

Underlying service model

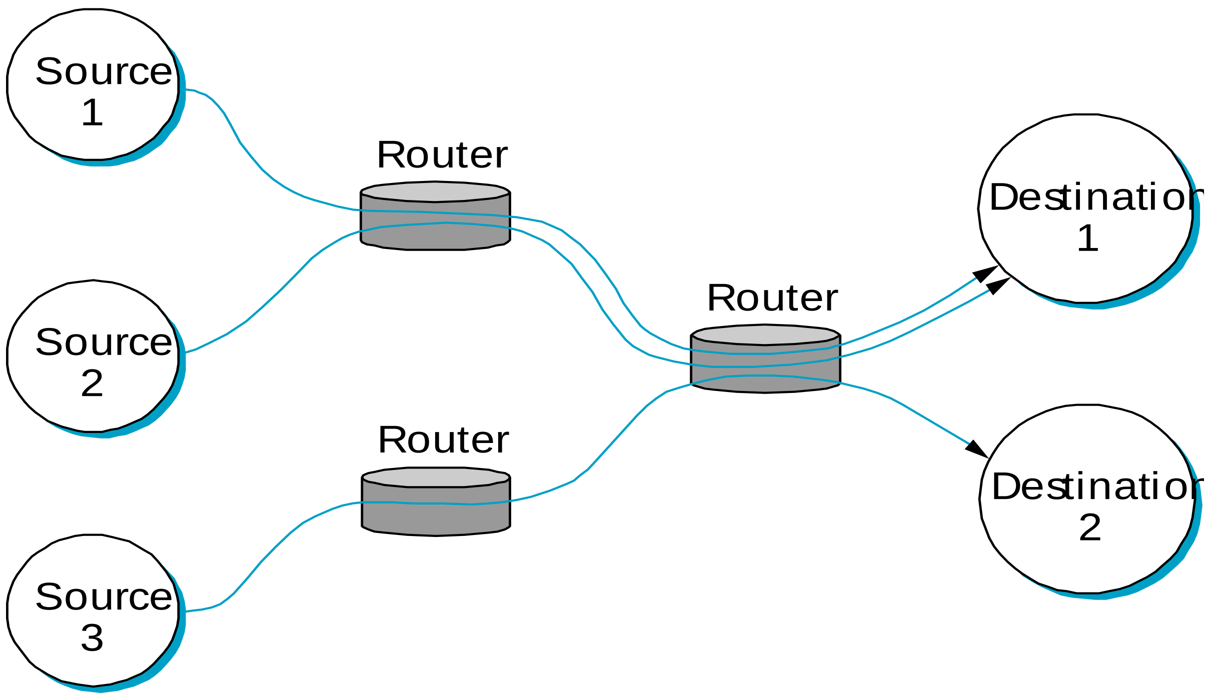
– best-effort connectionless services

Framework

Connectionless flows

–sequence of packets sent between source/destination pair

– maintain *soft state* at the routers



Classification on congestion control Techniques

--router-centric versus host-centric

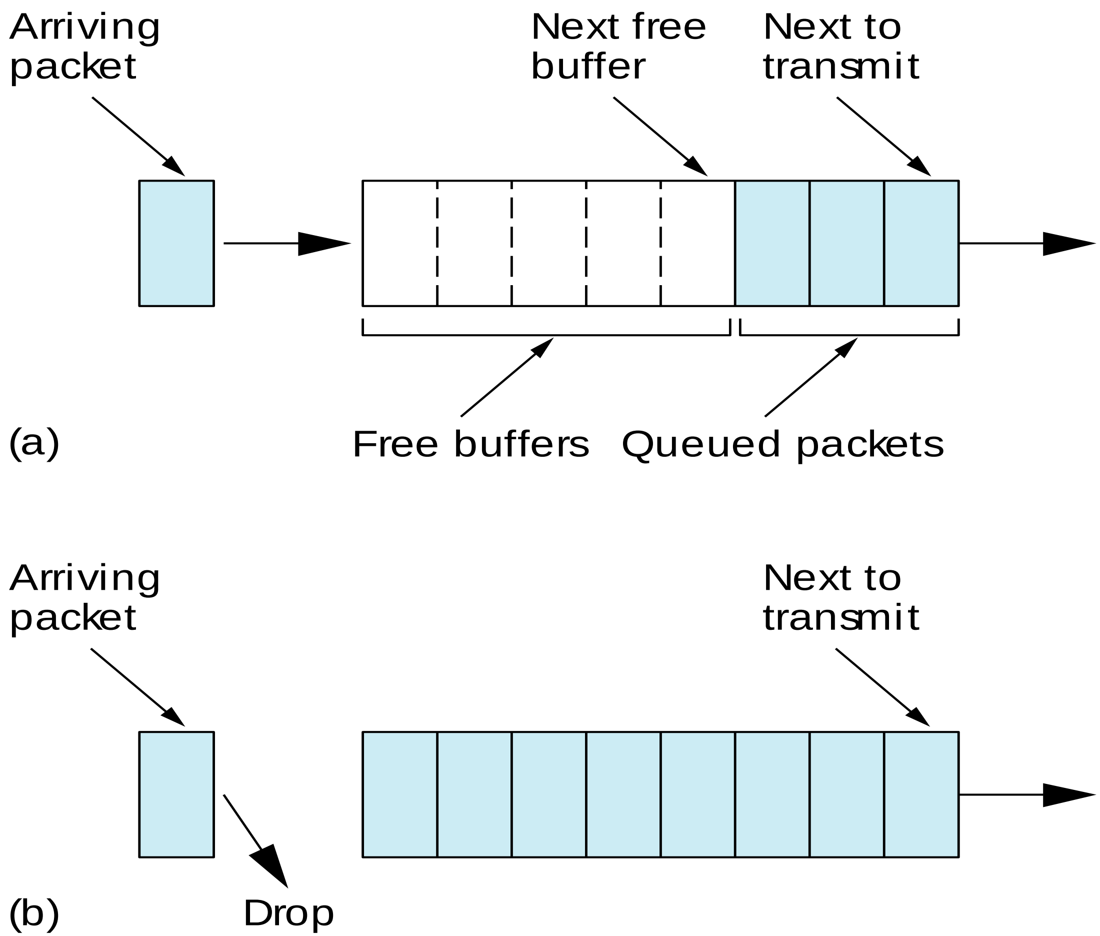
– reservation-based versus feedback-based

– window-based versus rate-based

Queuing Discipline

First-In-First-Out (FIFO)

– does not discriminate between traffic sources



Queuing Discipline

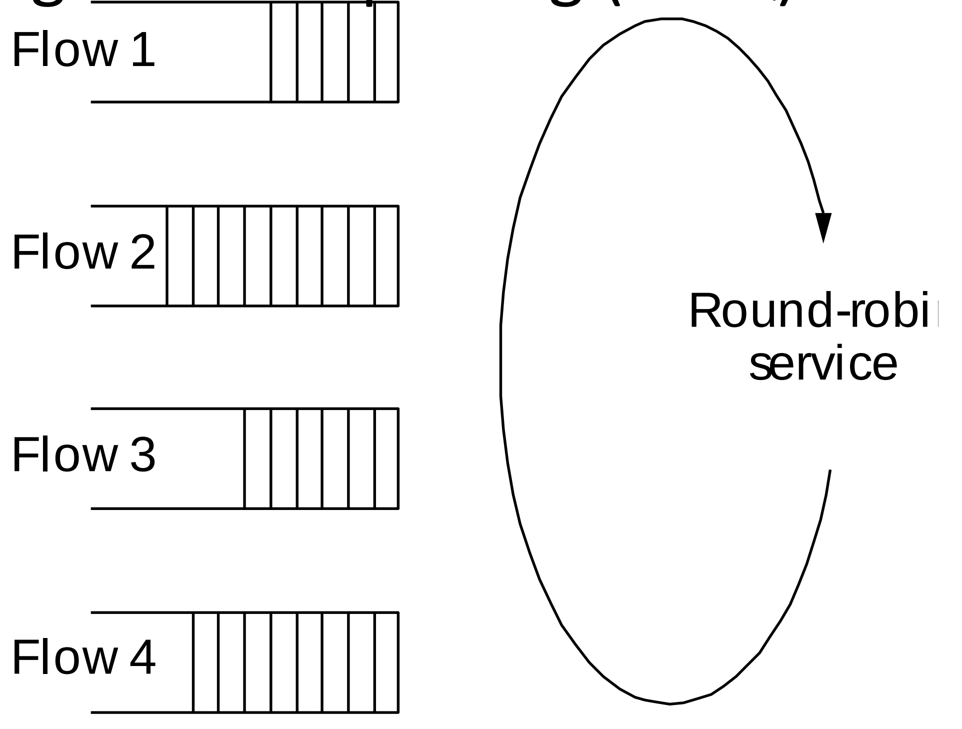
Fair Queuing (FQ)

– explicitly segregates traffic based on flows

– ensures no flow captures more than its share of capacity

– variation: weighted fair queuing (WFQ)

• Problem?



FQ Algorithm

Suppose clock ticks each time a bit is transmitted

- Let *Pi* denote the length of packet *i*
- Let *Si* denote the time when start to transmit packet *i*
- Let *Fi* denote the time when finish transmitting packet *i*

Fi = *Si* + *Pi*

When does router start transmitting packet *i*?

– if before router finished packet *i* - 1 from this flow, then immediately after last bit of *i* - 1 (*Fi-1*)

– if no current packets for this flow, then start

transmitting when arrives (call this *Ai*)

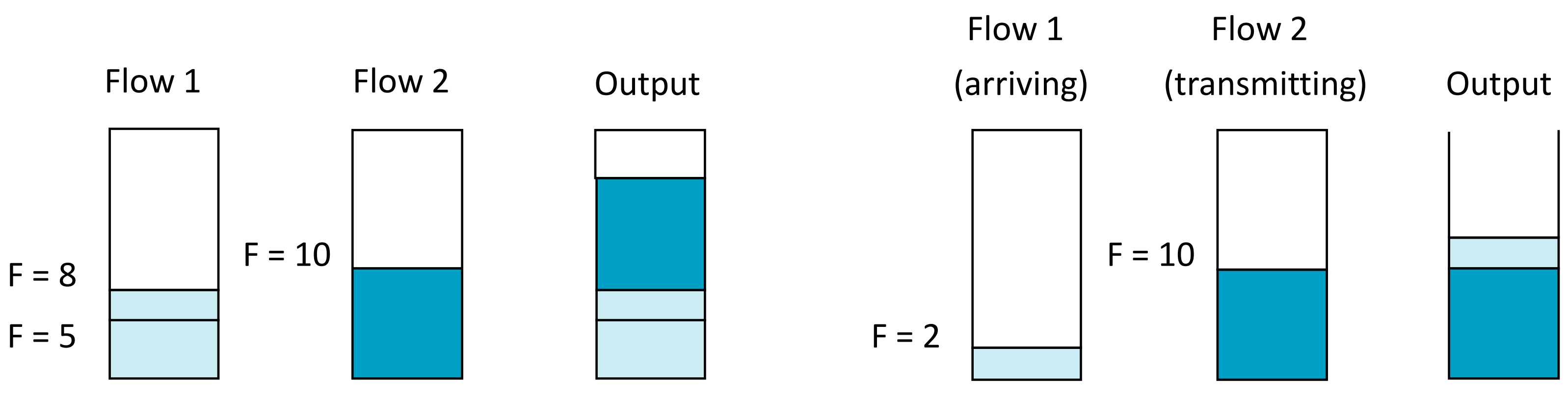
Thus: *Fi* = *MAX* (*Fi* - 1, *Ai*) + *Pi*

FQ Algorithm (cont)

For multiple flows

- – calculate *Fi* for each packet that arrives on each flow
- – treat all *Fi*'s as timestamps
- – next packet to transmit is one with lowest timestamp

- Not perfect: can't preempt current packet
- Example



(a)

(b)

TCP Congestion Control

- Idea
  - assumes best-effort network (FIFO or FQ routers)
  - each source determines network capacity for itself
  - uses implicit feedback
  - ACKs pace transmission (*self-clocking*)
- Challenge
  - determining the available capacity in the first place
  - adjusting to changes in the available capacity

Additive Increase/Multiplicative Decrease

Objective: adjust to changes in the available capacity

- New state variable per connection: **CongestionWindow**
- limits how much data source has in transit

MaxWin = MIN(CongestionWindow, AdvertisedWindow)

EffWin = MaxWin - (LastByteSent - LastByteAcked)

Idea:

– increase **CongestionWindow** when congestion goes down

– decrease **CongestionWindow** when congestion goes up

AIMD (cont)

Question: how does the source determine whether or not the network is congested?

Answer: a timeout occurs

- timeout signals that a packet was lost
- packets are seldom lost due to transmission error
- lost packet implies congestion

AIMD (cont)

Algorithm

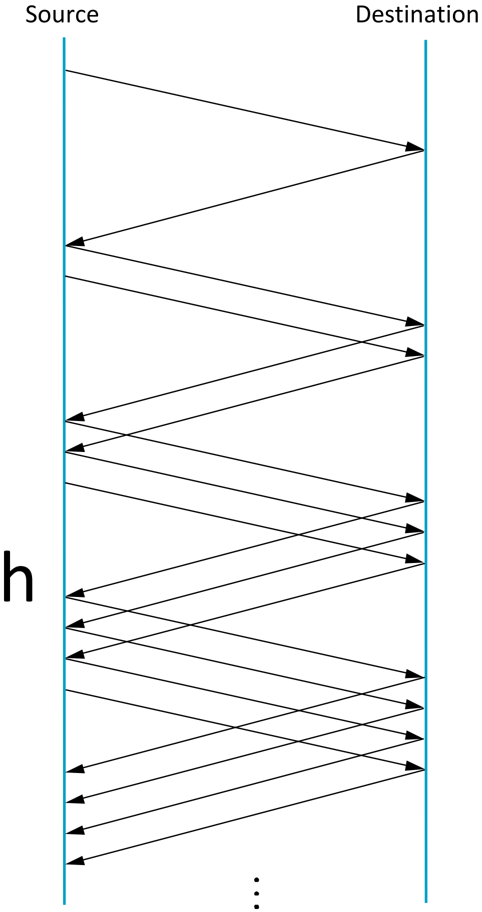
– increment **CongestionWindow** by one packet per RTT (*linear increase*)

– divide **CongestionWindow** by two whenever a timeout occurs (*multiplicative decrease*)

In practice: increment a little for each ACK

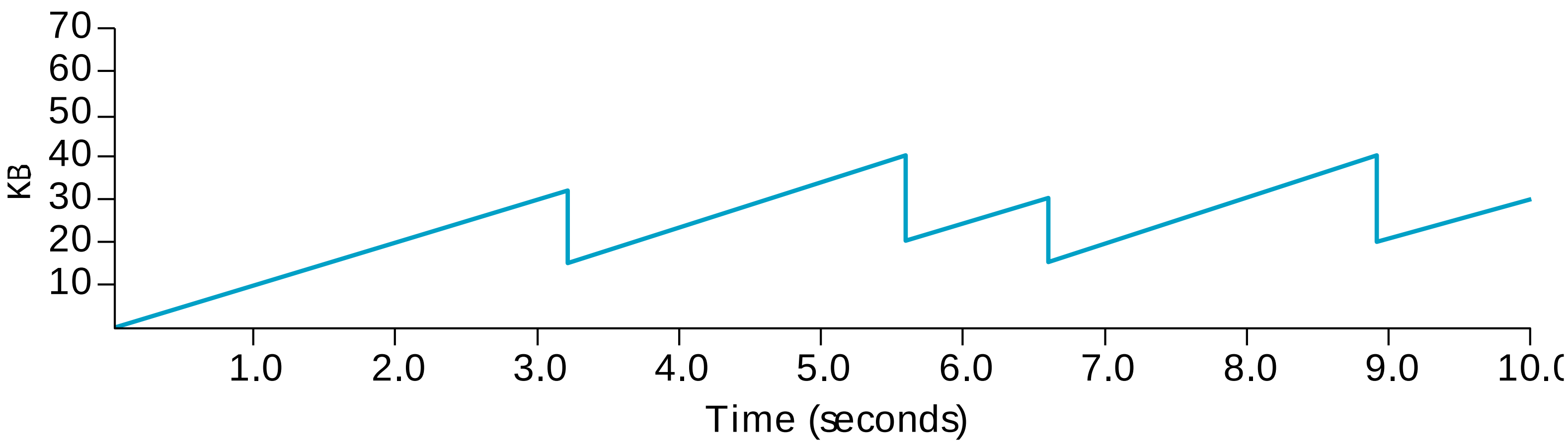
Increment = (MSS * MSS)/CongestionWindow

CongestionWindow += Increment



AIMD (cont)

Trace: sawtooth behavior



Slow Start

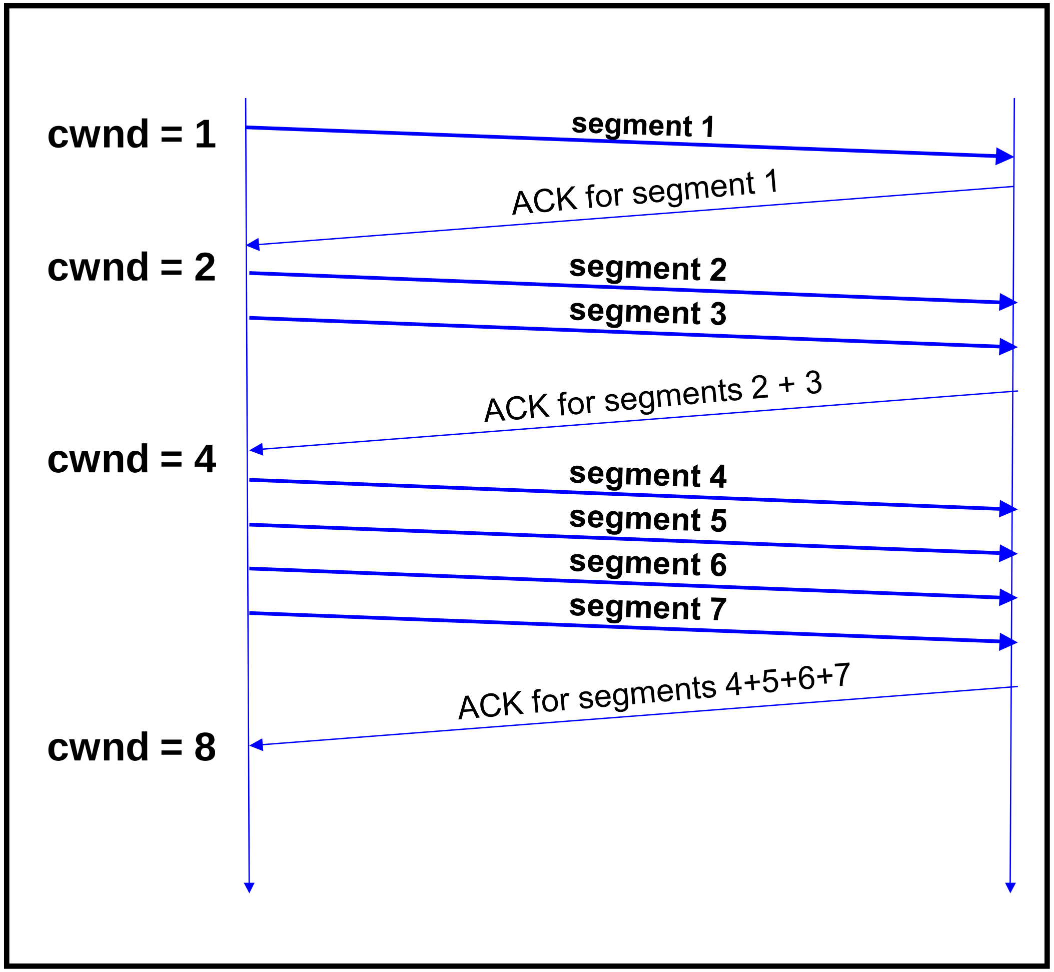
Objective: determine the available capacity in the first

- Idea:
  - begin with **CongestionWindow** = 1 packet
  - double **CongestionWindow** each RTT (increment by 1 packet for each ACK)

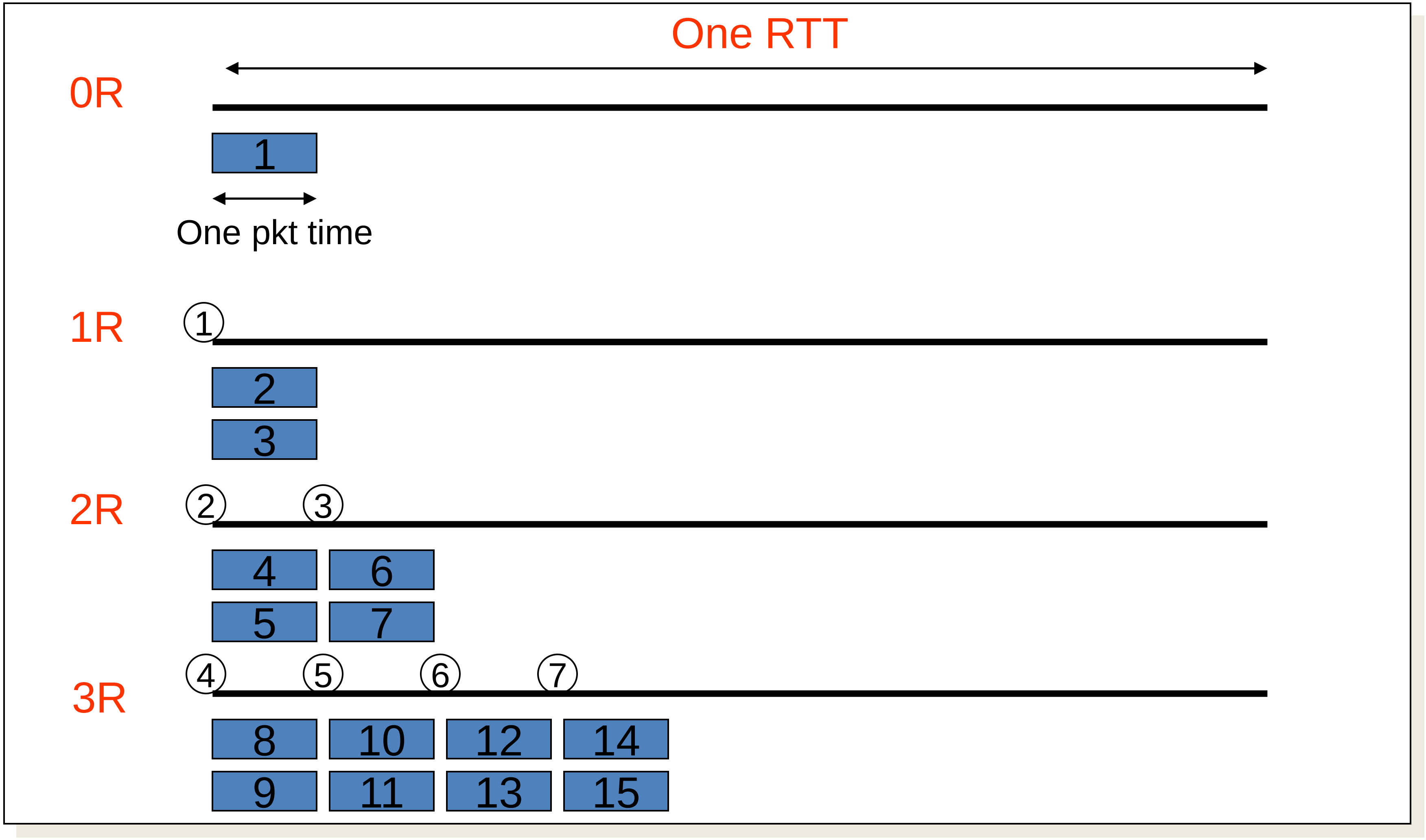
Slow Start Example

- The congestion window size grows very rapidly

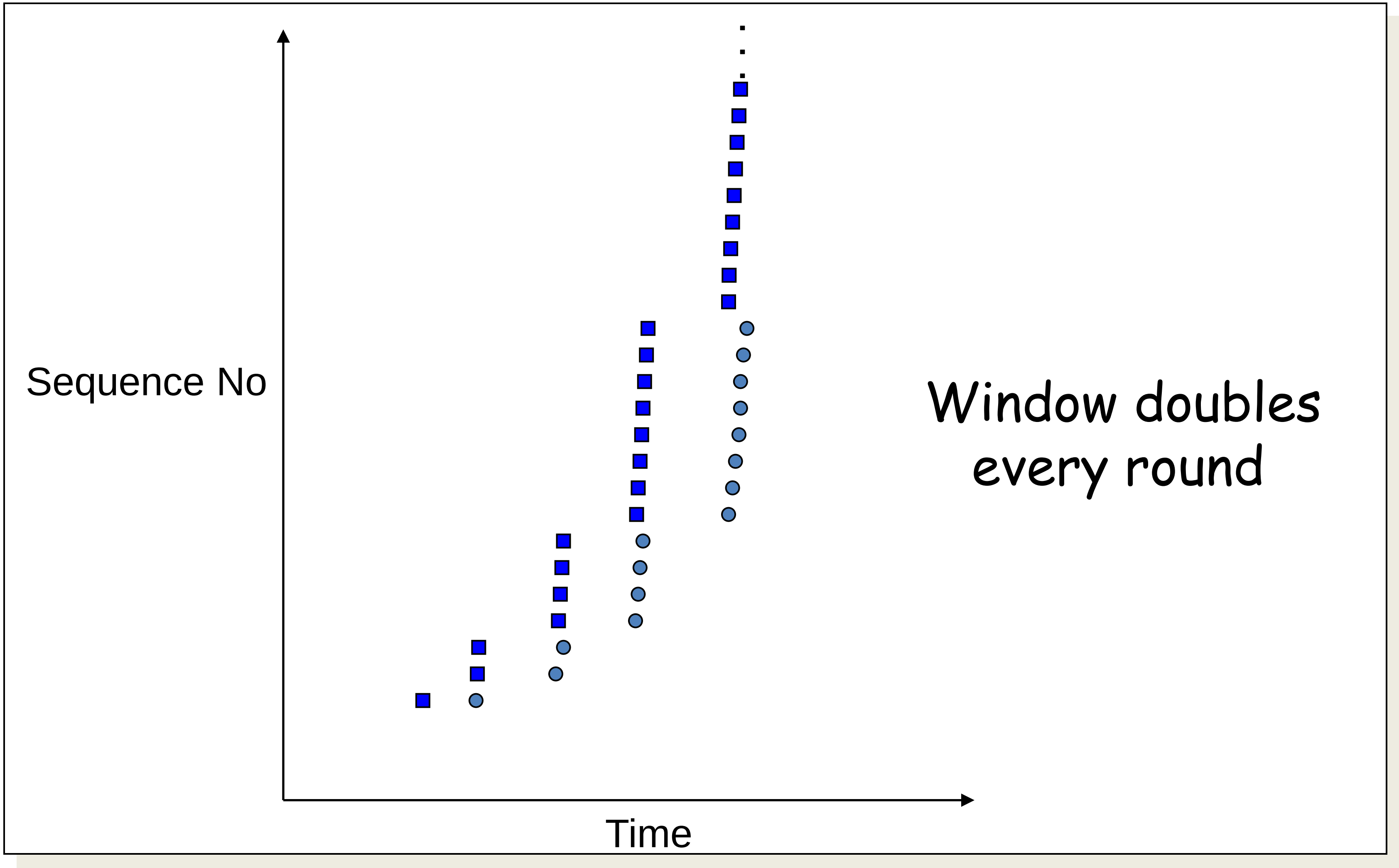
- TCP slows down the increase of *cwnd* when ***cwnd >= ssthresh***



Slow Start Example



Slow Start Sequence Plot



Congestion Avoidance

- Goal: maintain operating point at the left of the cliff:
- How?
  - additive increase: starting from the rough estimate (ssthresh), slowly increase cwnd to probe for additional available bandwidth
  - multiplicative decrease: cut congestion window size aggressively if a loss is detected.

Congestion Avoidance

- Slow down “Slow Start”

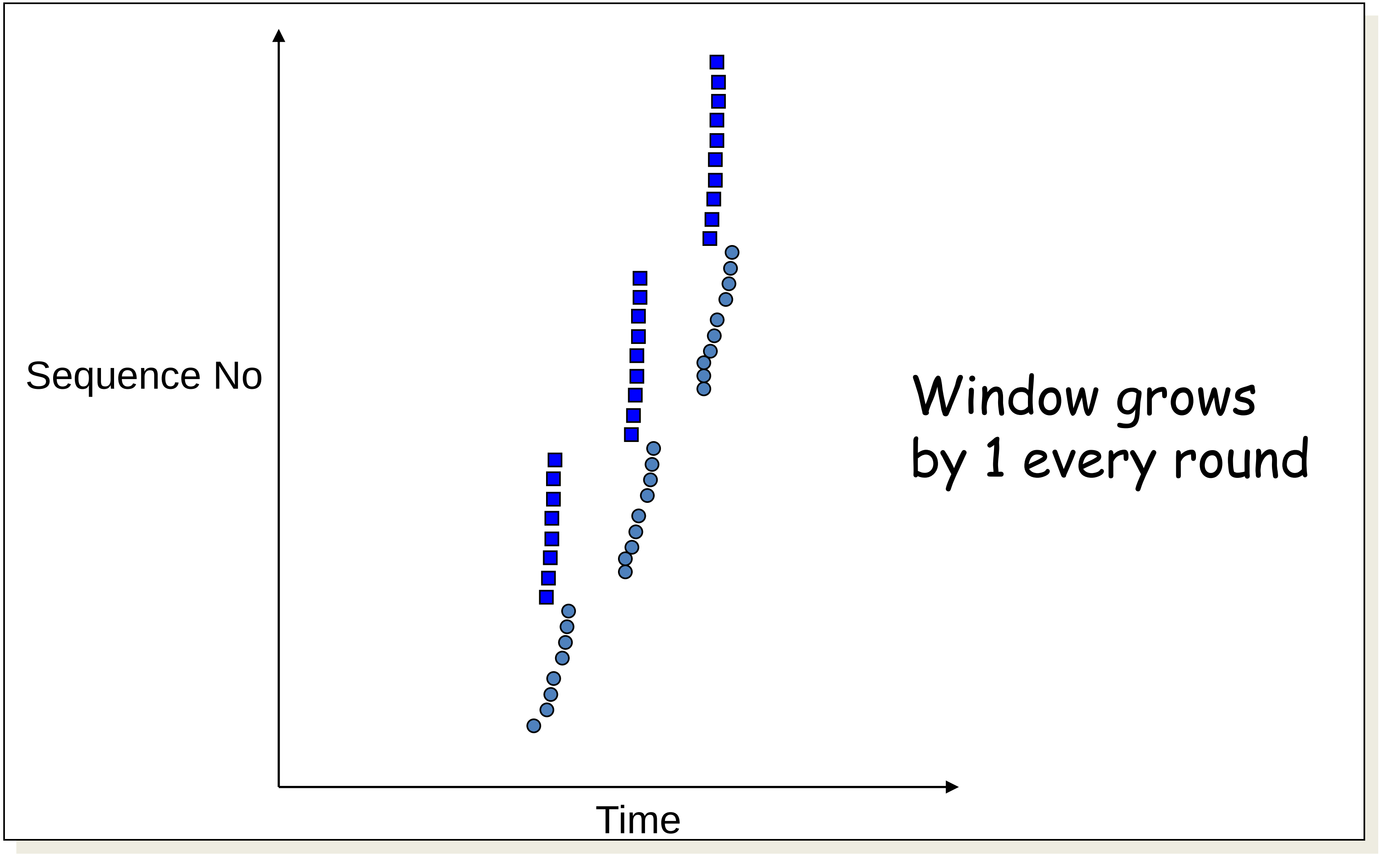
- **If** *cwnd > ssthresh* **then**

  each time a segment is acknowledged
  increment *cwnd* by *1/cwnd*

  i.e. (*cwnd += 1/cwnd*).

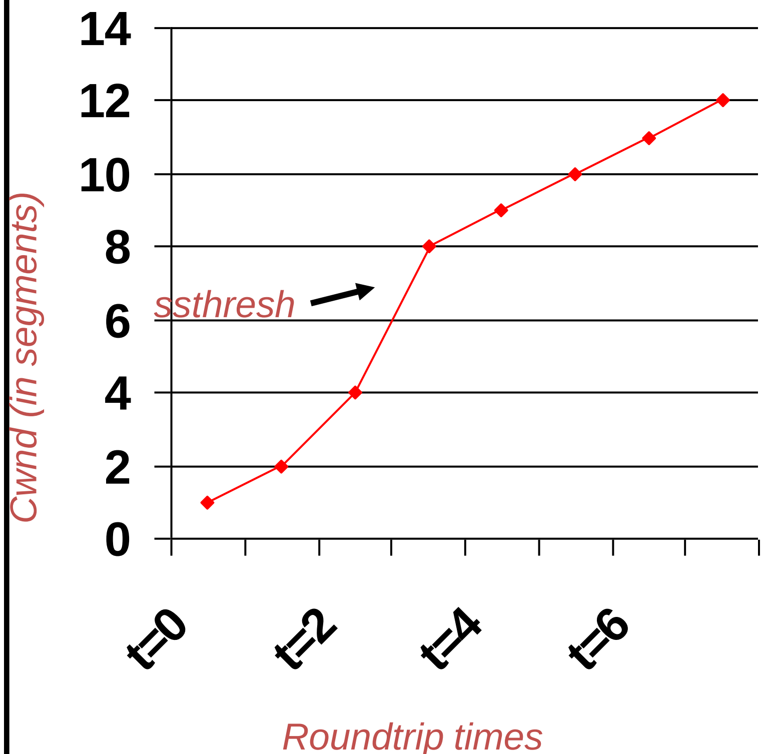
- So *cwnd* is increased by one only if all segments have been acknowledged.

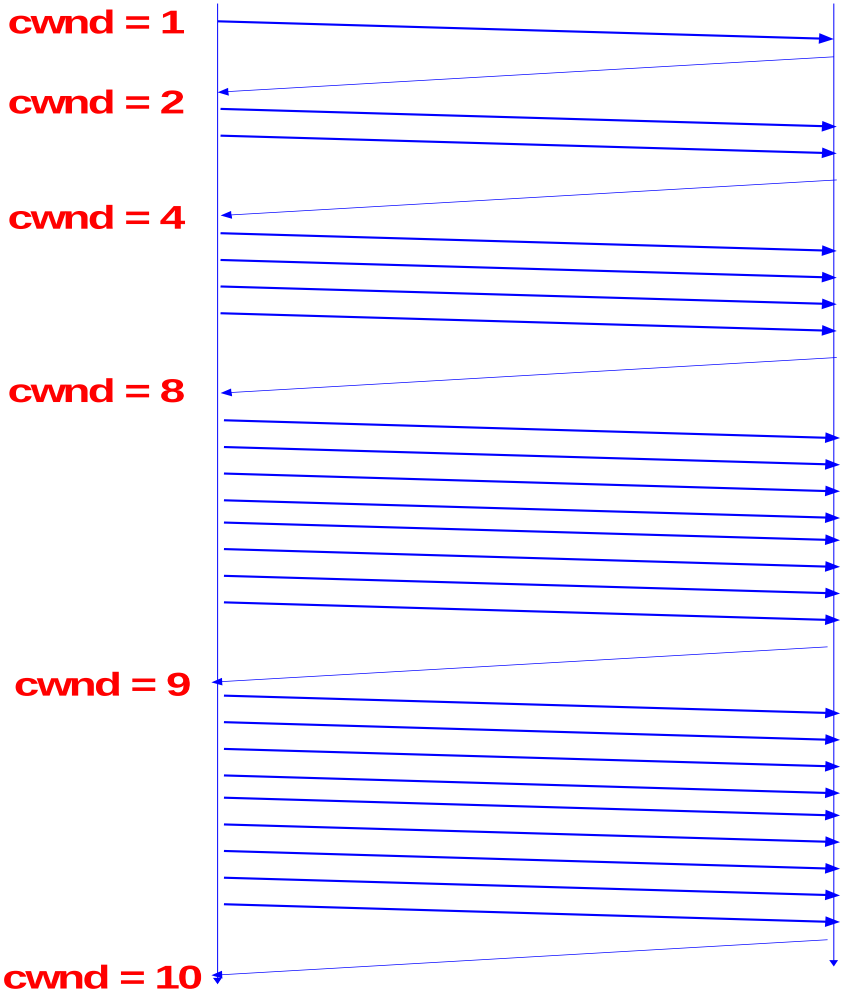
Congestion Avoidance Sequence Plot



Slow Start/Congestion Avoidance Eg.

- Assume that *ssthresh = 8*



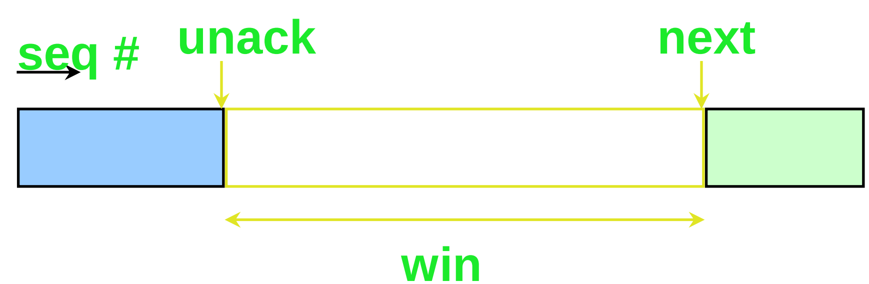


Putting Everything Together: TCP Pseudo-code

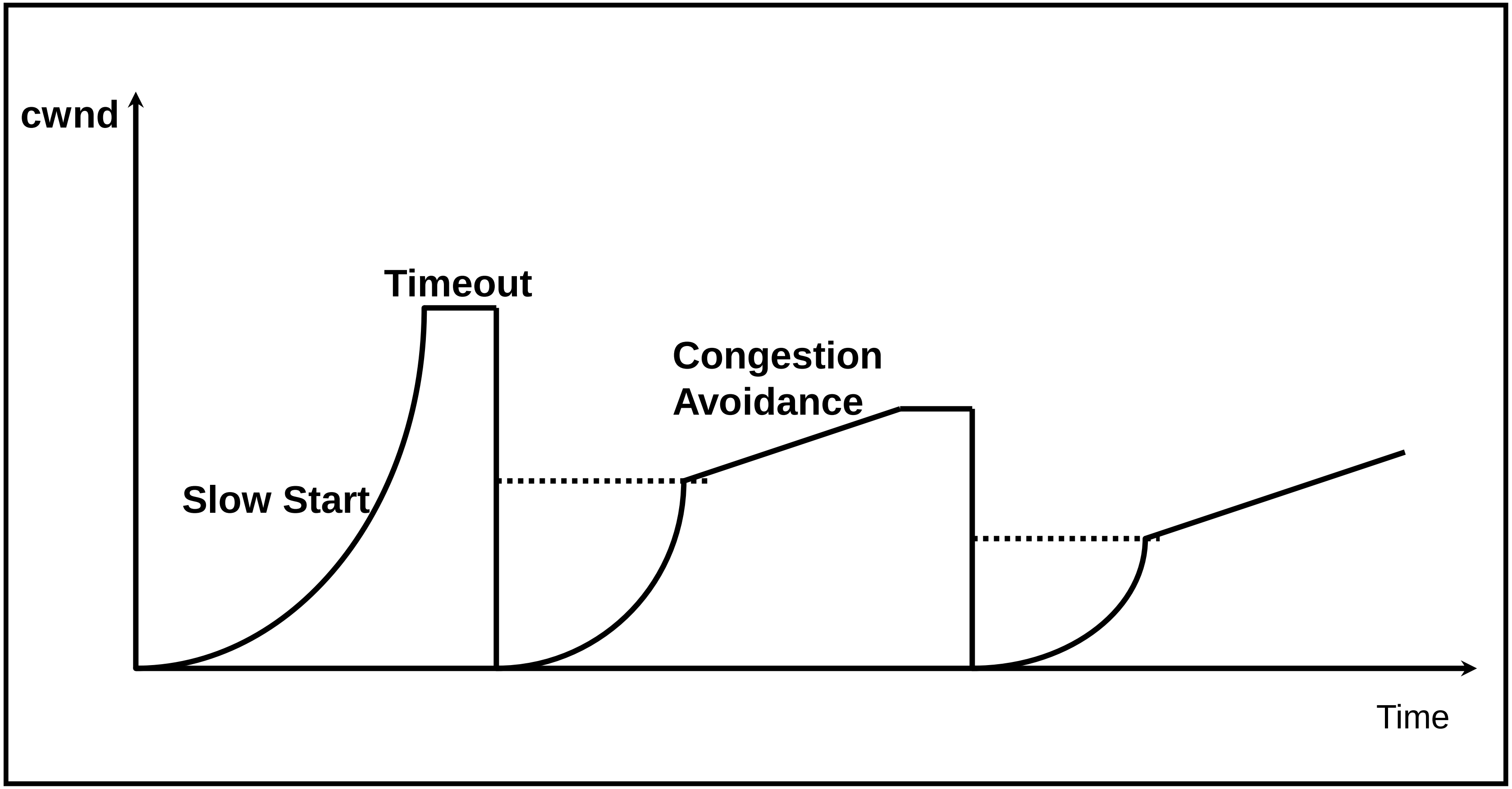
```
Initially:
    cwnd = 1;
    ssthresh = infinite;
New ack received:
    if (cwnd < ssthresh)
        /* Slow Start*/
        cwnd = cwnd + 1;
    else
        /* Congestion Avoidance */
        cwnd = cwnd + 1/cwnd;
Timeout: (loss detection)
    /* Multiplicative decrease */
    ssthresh = win/2;
    cwnd = 1;
```

```
while (next < unack + win)
    transmit next packet;

where   win = min(cwnd,
                  flow_win);
```



The big picture



Slow Start (cont)

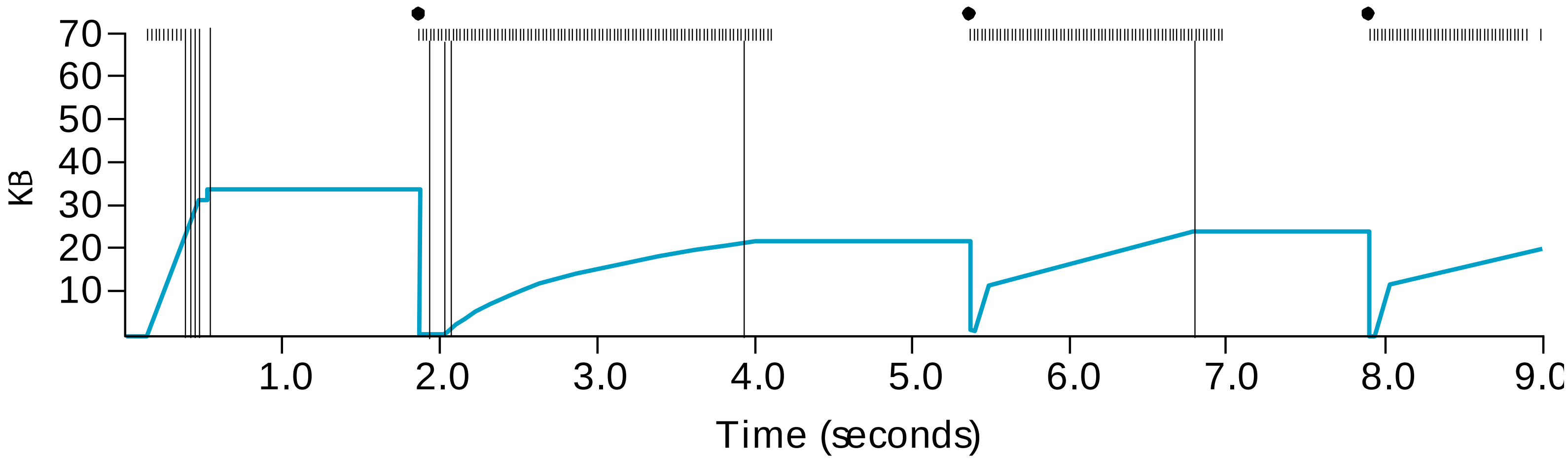
Exponential growth, but slower than all at once

Used…

– when first starting connection

– when connection goes dead waiting for timeout

- Trace



- Problem: lose up to half a **CongestionWindow**'s worth of data

Packet Loss Detection: Timeout Avoidance

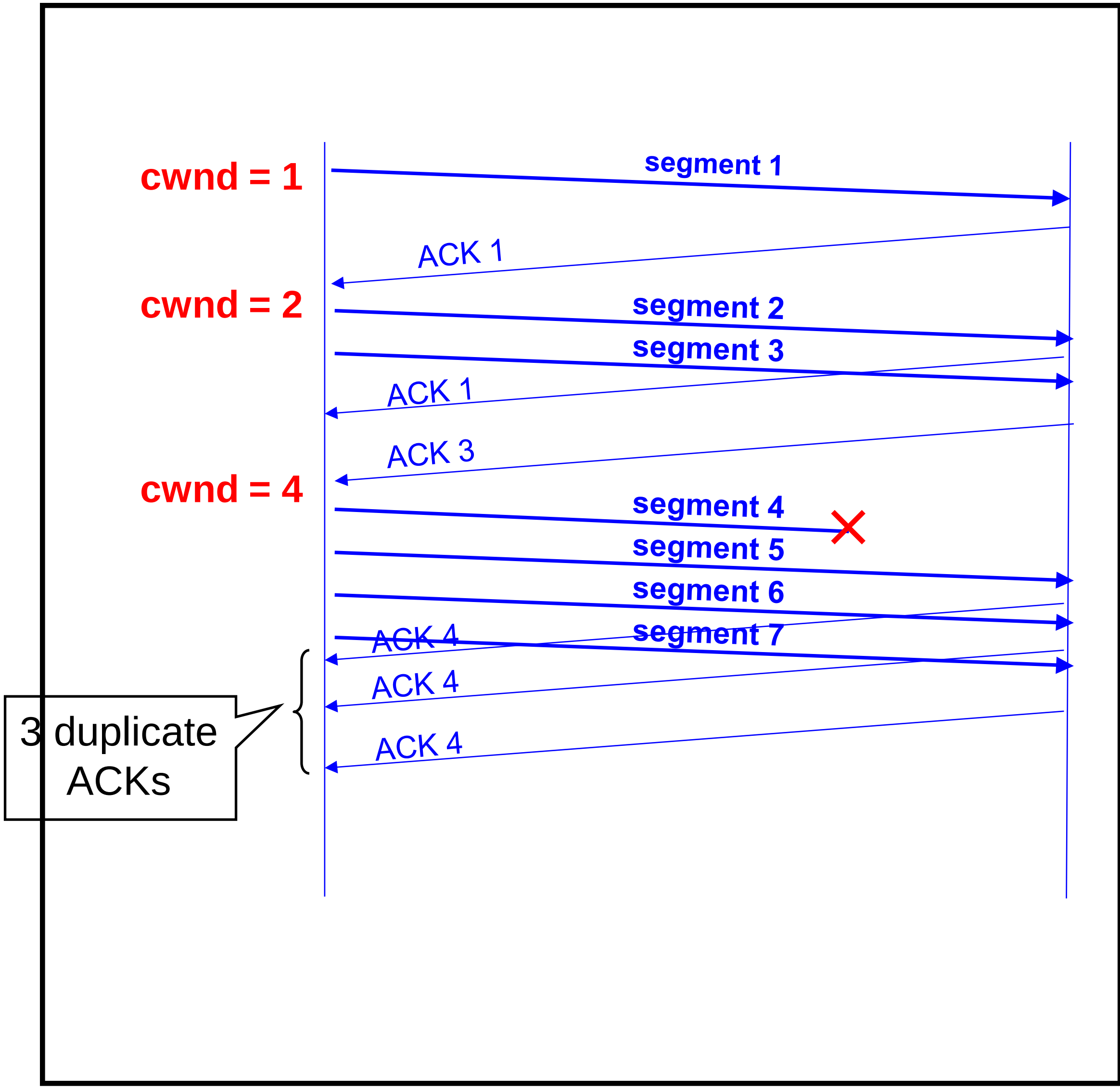
- Wait for Retransmission Time Out (RTO)
- What's the problem with this?
  - **Because RTO is a performance killer**

- In BSD TCP implementation, RTO is usually more than 1 second
  - the granularity of RTT estimate is 500 ms
  - retransmission timeout is at least two times of RTT

- Solution: Don't wait for RTO to expire
  - Use alternate mechanism for loss detection
  - Fall back to RTO only if these alternate mechanisms <u>fail</u>.

Fast Retransmit

- Resend a segment after 3 duplicate ACKs
  - Recall: a duplicate ACK means that an out-of sequence segment was received
- Notes:
  - duplicate ACKs due packet reordering!
  - if window is small don't get duplicate ACKs!

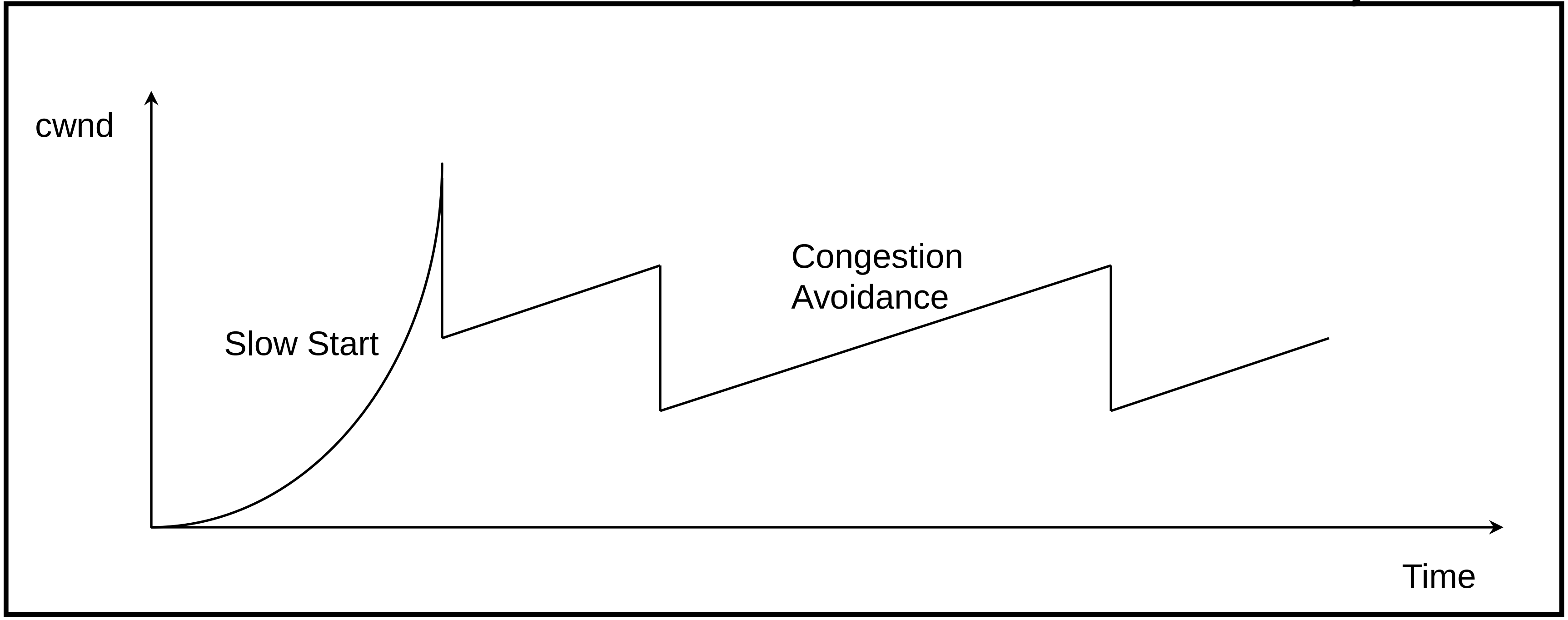


Fast Recovery (Simplified)

- After a fast-retransmit set *cwnd* to *ssthresh/2*
  - i.e., don't reset *cwnd* to 1
- But when RTO expires still do *cwnd* = 1

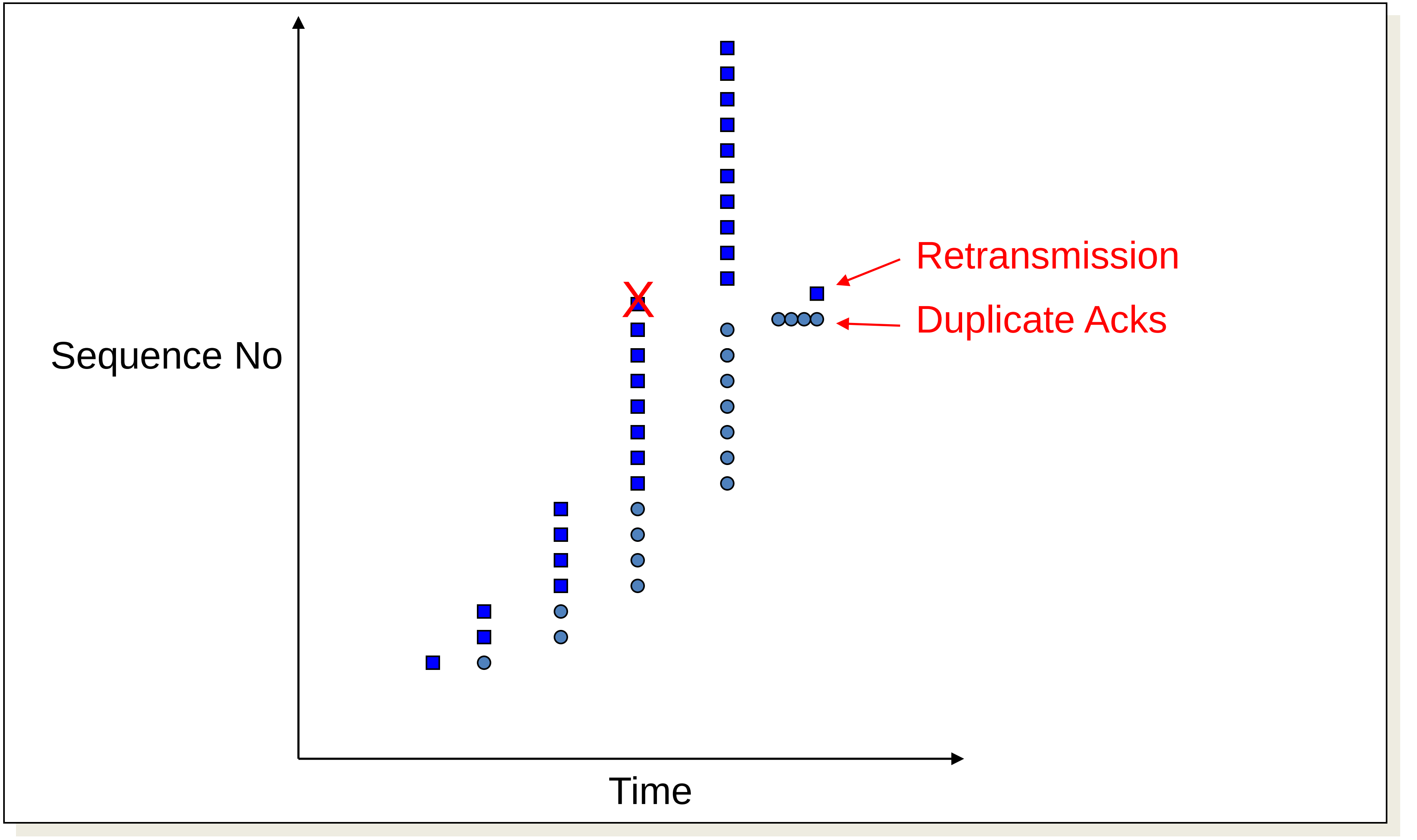
- Fast Retransmit and Fast Recovery → implemented by TCP Reno; most widely used version of TCP today

Fast Retransmit and Fast Recovery



- Retransmit after 3 duplicated acks
  - prevent expensive timeouts
- No need to slow start again
- At steady state, *cwnd* oscillates around the optimal window size.

Fast Retransmit



Congestion Avoidance

TCP's strategy

- control congestion once it happens
- repeatedly increase load in an effort to find the point at which congestion occurs, and then back off

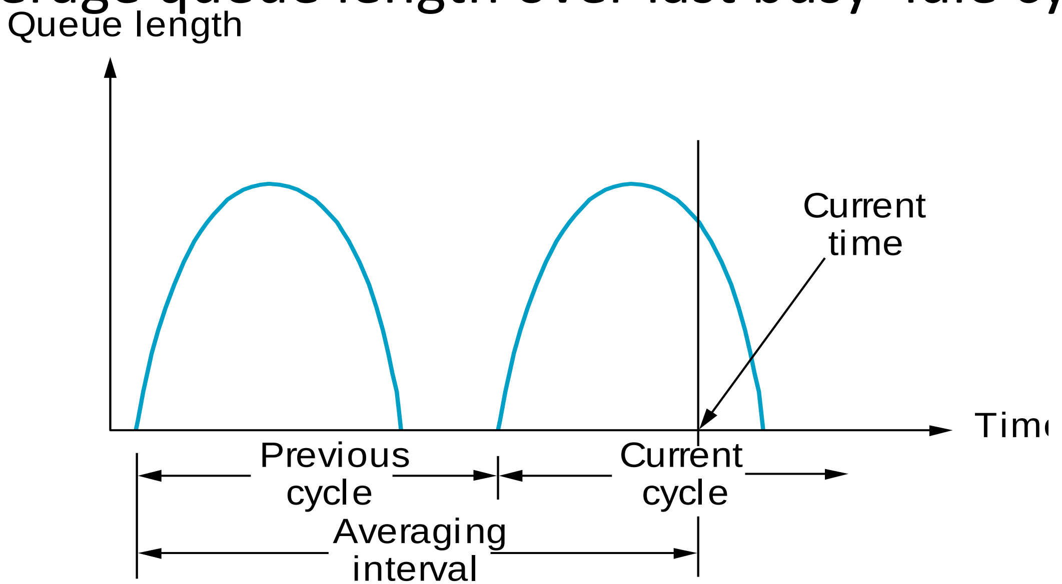
- Alternative strategy
  - predict when congestion is about to happen
  - reduce rate before packets start being discarded
  - call this congestion *avoidance*, instead of congestion *control*
- Approach
  - router-centric: DECbit and RED Gateways

DECbit

Add binary congestion but to each packet header

Router

– monitors average queue length over last busy+idle cycle



– set congestion bit if average queue length > 1

– attempts to balance throughout against delay

End Hosts

- Destination echoes bit back to source
  - Source records how many packets resulted in set bit
  - If less than 50% of last window's worth had bit set
  - increase **CongestionWindow** by 1 packet
- If 50% or more of last window's worth had bit set
  - decrease **CongestionWindow** by 0.875 times

Random Early Detection (RED)

Notification is implicit

- just drop the packet (TCP will timeout)
- could make explicit by marking the packet

- Early random drop
  - rather than wait for queue to become full, drop each arriving packet with some *drop probability* whenever the queue length exceeds some *drop level*

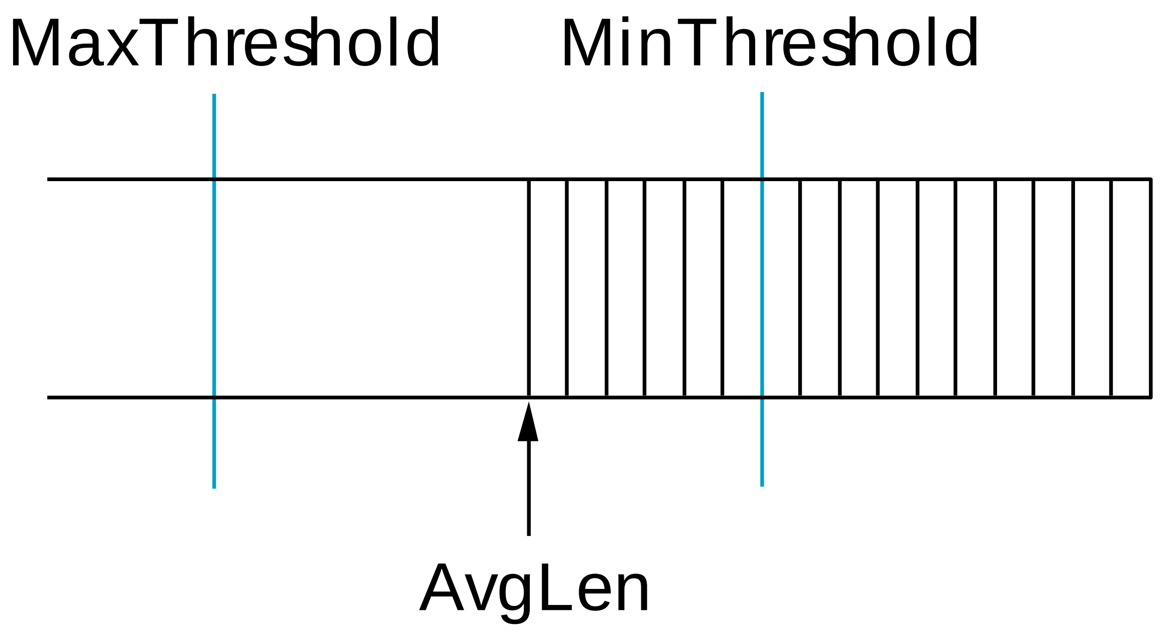
RED Details

Compute average queue length

AvgLen = (1 - Weight) * AvgLen + Weight * SampleLen

0 < **Weight** < 1 (usually 0.002)

SampleLen is queue length each time a packet arrives



RED Details (cont)

Two queue length thresholds

if AvgLen <= MinThreshold

then enqueue the packet

if MinThreshold < AvgLen < MaxThreshold

then

calculate probability P

drop arriving packet with probability P

if MaxThreshold <= AvgLen

then drop arriving packet

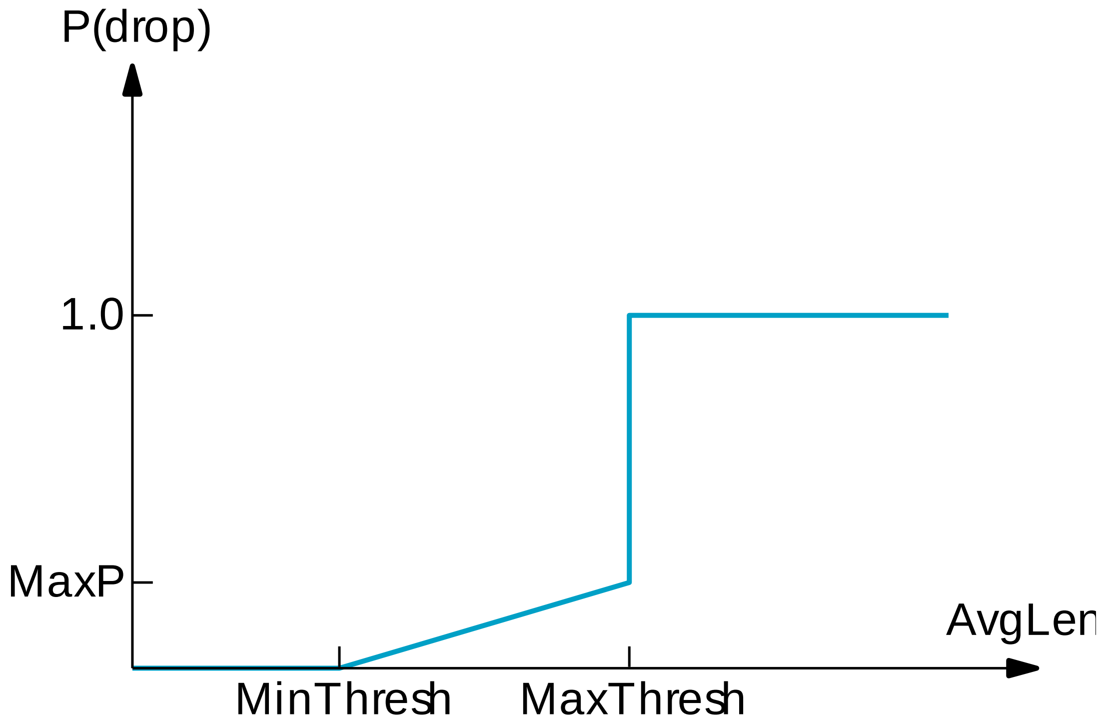
RED Details (cont)

Computing probability P

TempP = MaxP * (AvgLen - MinThreshold)/(MaxThreshold - MinThreshold)

P = TempP/(1 - count * TempP)

- Drop Probability Curve



TCP Congestion Control Summary

- *Sliding* window limited by receiver window.
- Dynamic ***windows***: slow start (exponential rise), congestion avoidance (additive rise), multiplicative decrease.
  - Ack clocking
- Adaptive ***timeout***: need mean RTT & deviation
- Timer backoff and Karn's algo during retransmission
- Go-back-N or Selective ***retransmission***
- Cumulative and Selective ***acknowledgements***
- *Timeout avoidance:* Fast Retransmit