CS519: Computer Networks

Lecture 5, Part 4: Mar 29, 2004

Transport: TCP congestion control

TCP performance

- We’ve seen how TCP “the protocol” works
  - Sequencing, receive window, connection setup and teardown
- And we’ve seen techniques to make it perform well
  - RTT estimation, sending big packets, compression, fast timeout

TCP congestion control

- Now lets finish the picture:
- How TCP avoids and controls congestion in the network
- Without this, TCP still won’t perform well…

What is congestion?

- Lets distinguish between a strict definition of congestion and a working definition of congestion
- Strictly:
  - Congestion occurs anytime more than one packet competes for the same link at the same time

Question:

- Do we want to prevent instances of multiple packets competing for the same link at the same time?

Answer:

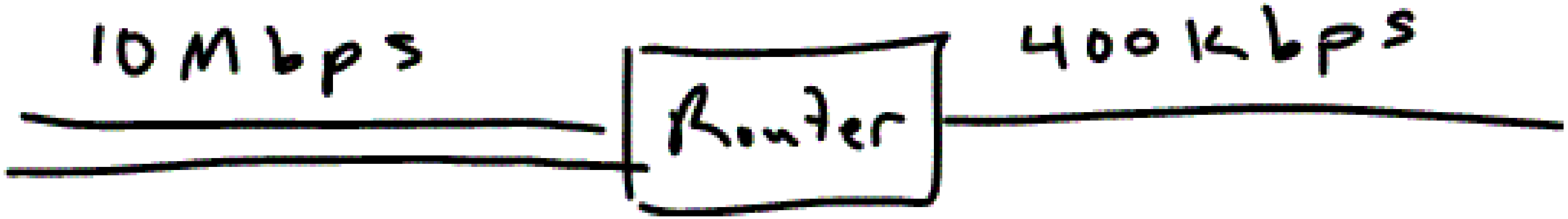
- No!
- Pure circuit networks avoid ever having two packets compete for the same link at the same time
  - (more or less)
- By reserving a fixed amount of bandwidth at each link for each connection
- But as we've already discussed, for bursty traffic, utilization is low!

Queues in switches

- Queues deal with congestion at packet timescales
  - Two packets arrive at the same time, one is queued behind the other
- Queues allow us to increase the utilization of links
  - At the expense of packet delay
- In this sense, packet timescale congestion is actually good!

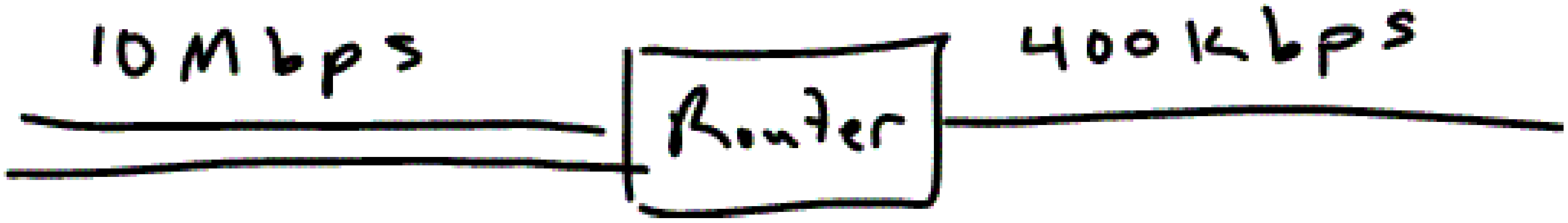
Delay and throughput

- With no queue, sender can never send at more than 400Kbps
- If sender bursty, then bursts are limited to 400Kbps,
    - with links unused during periods between bursts



Delay and throughput

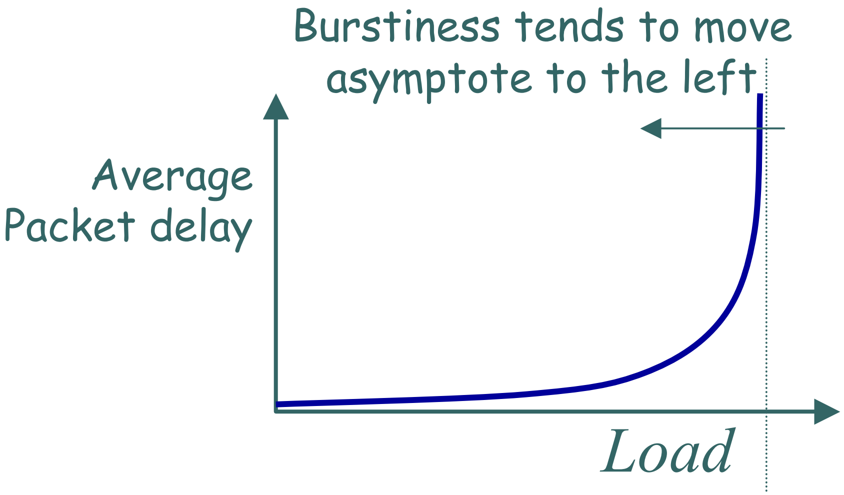
- With a queue, sender can burst at 10Mbps
- Burst will start to fill the queue
- After burst is over, queue empties into slow link
  - Link utilized during silent periods!



Load, delay and power

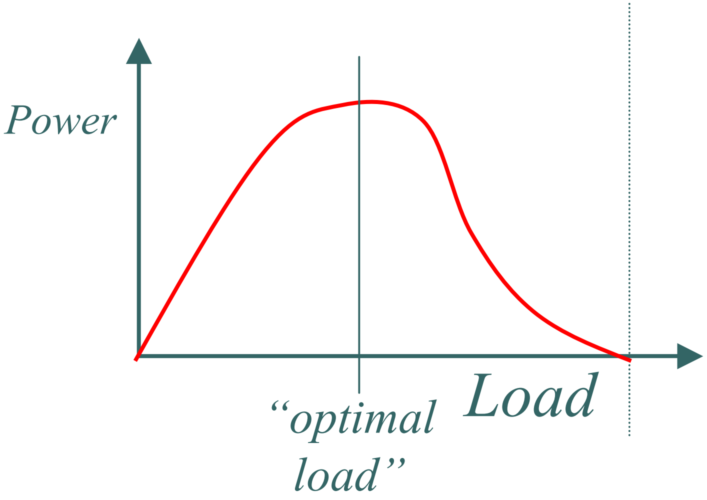
CS519

Typical behavior of queuing systems with random arrivals:



A simple metric of how well the network is performing:

$$Power = \frac{Load}{Delay}$$

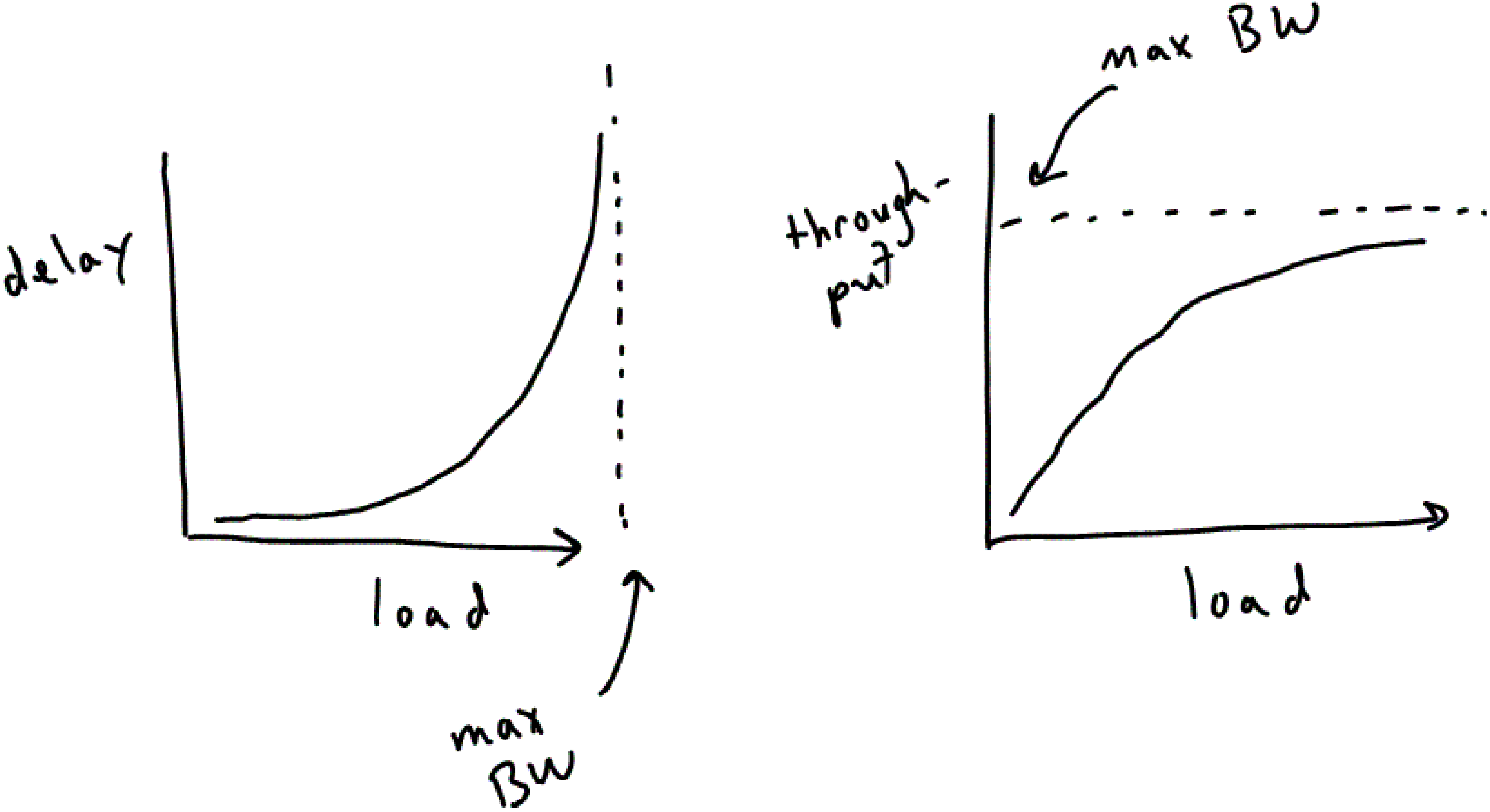


Slide from Nick McKeown, Stanford

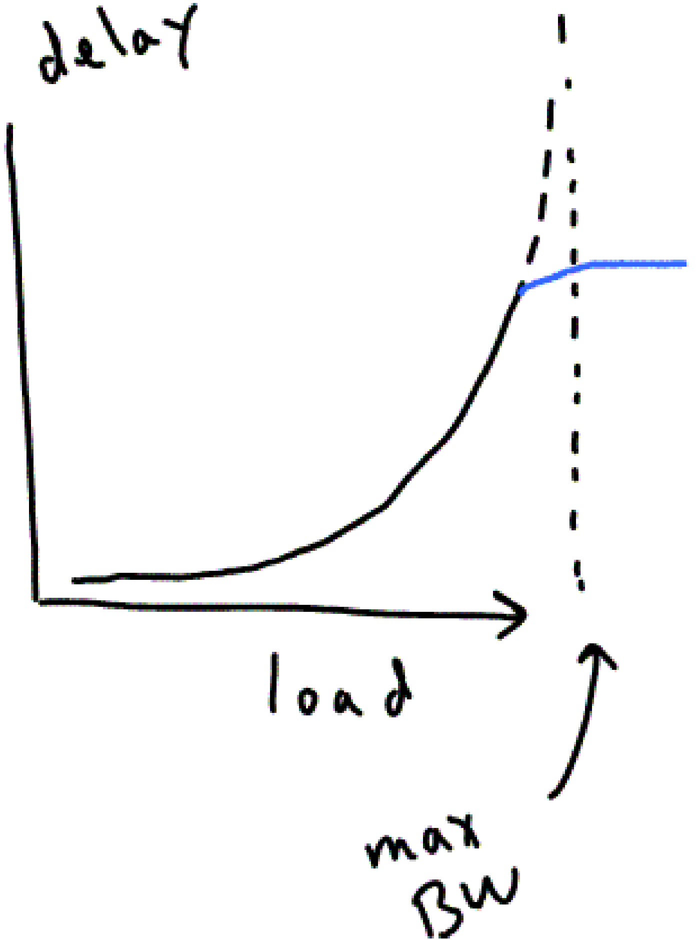
Our definition of congestion

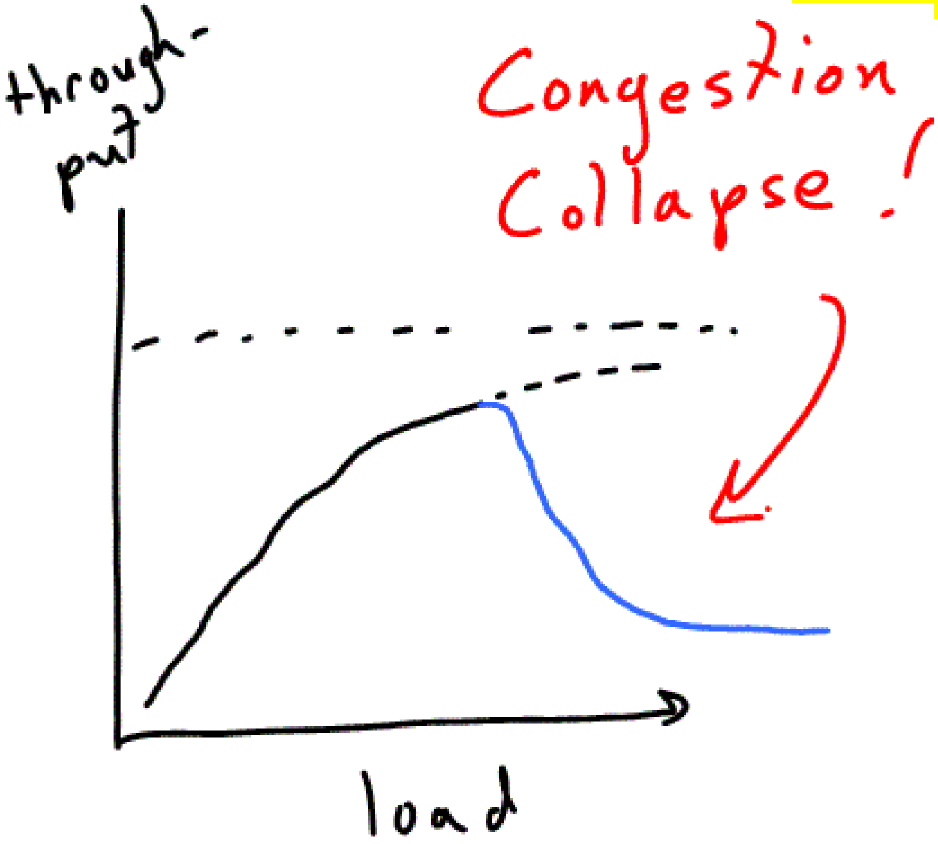
- Where network load is large enough that queues overflow and packets are lost
- We are also concerned with “congestion collapse”

Load, delay, and throughput: what's wrong with this picture??



Queue's aren't infinite, packets get dropped





Why congestion collapse?

- Lost packets leads to retransmissions
- Retransmissions add to load, resulting in more lost packets
- Packets may go several hops before being dropped
  - Using up resources along the way
- Note congestion collapse doesn’t occur where there are no retransmissions

TCP was causing congestion collapse

- In the late 1980's---Internet was becoming unusable!
- Solution attributed to Van Jacobsen
- Problem was that the network did not signal the host when there was congestion
  - ICMP source quench wasn't widely implemented

TCP congestion control

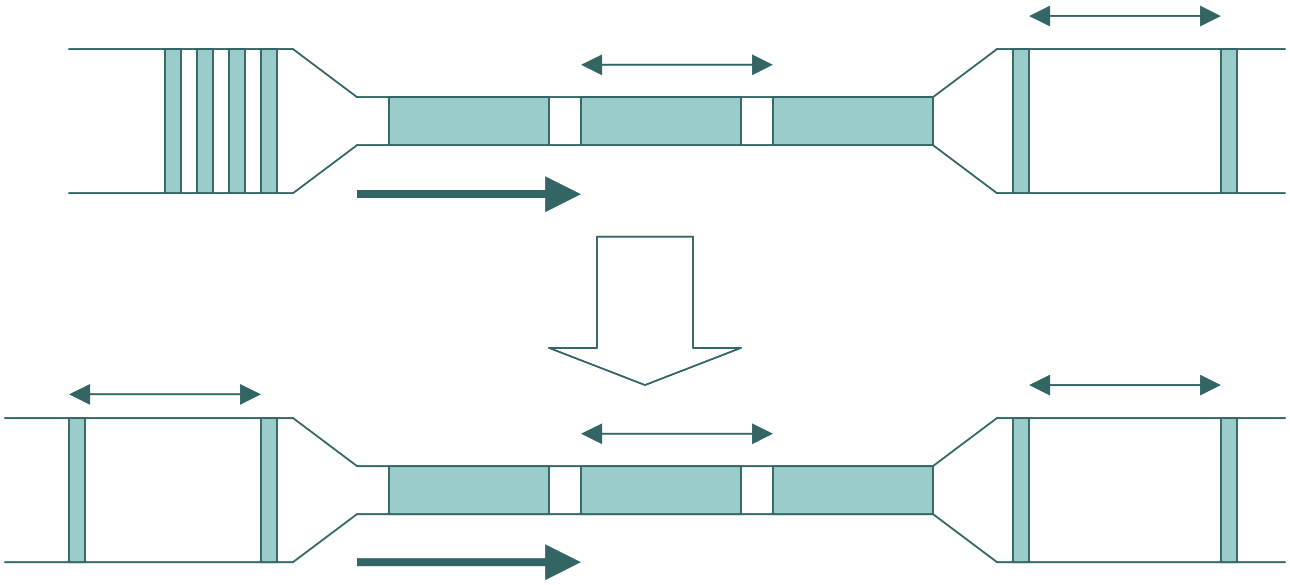
- Basic idea:
  - TCP gently “probes” the network to determine its capacity
  - Uses dropped packets as a sign of congestion
  - Backs off when congestion sensed

TCP congestion control goals

- First and foremost, prevent congestion collapse
- Also, fairly apportion resources
  - Each TCP flow gets an equal amount of the link bandwidth
- While achieving good performance
  - Keep the pipe full, but not too full!

Ideal TCP behavior

- Bottleneck bandwidth determines inter-packet spacing
  - Sender should space packets

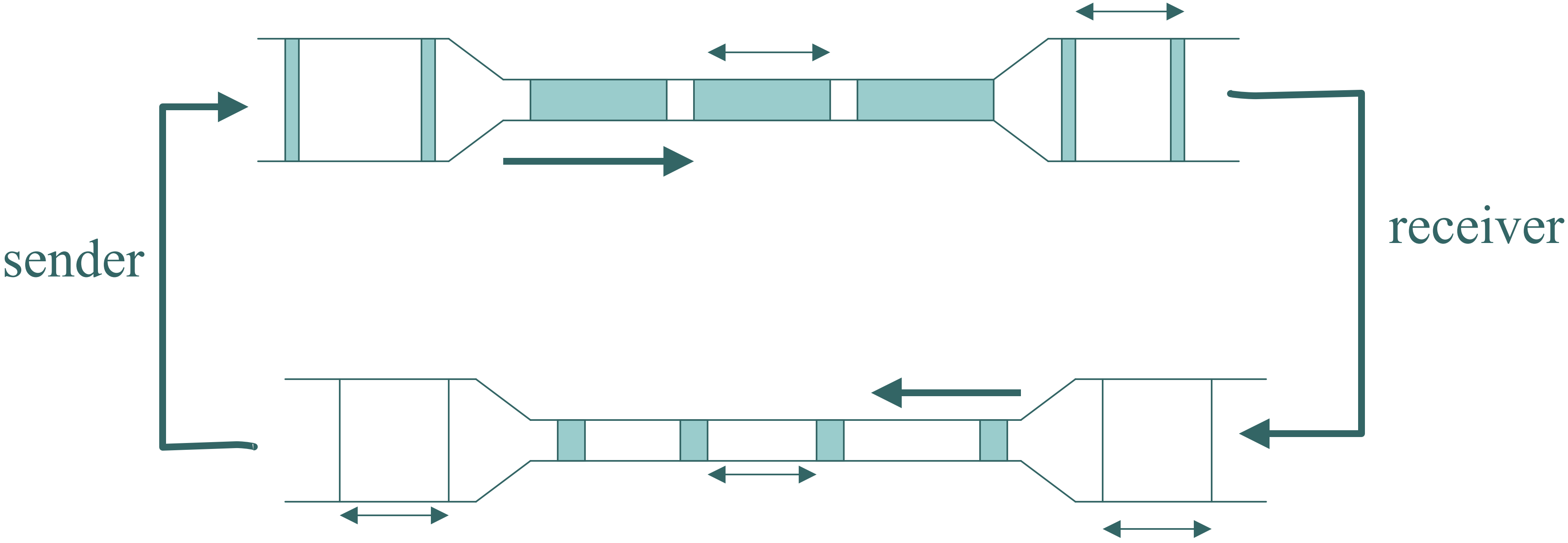


How can TCP sender space packets properly?

- Any ideas?

How can TCP sender space packets properly?

- Simple solution: use returned ACKs to clock packets out!



Ideal TCP behavior

- Get the pipe full
- Once full, use return ACKs to clock out new packets

- Now the question is, how to you know when the pipe is full???

Answer:

- You don’t know when the pipe is full!
- You only know when it is too full!
  - When there is a packet loss
  - Actually, more recent work challenges this…
- So, what TCP does is slowly fill the pipe until it is too full, then drain the pipe some and start filling again . . .

TCP congestion control

- Sender maintains two windows:
  - The advertised receive window we learned about
  - A congestion window (cwnd)
- The actual window is the minimum of the two:
  - Window = min{Advertized window, cwnd}
- In other words, send at the rate of the slowest component: network or receiver

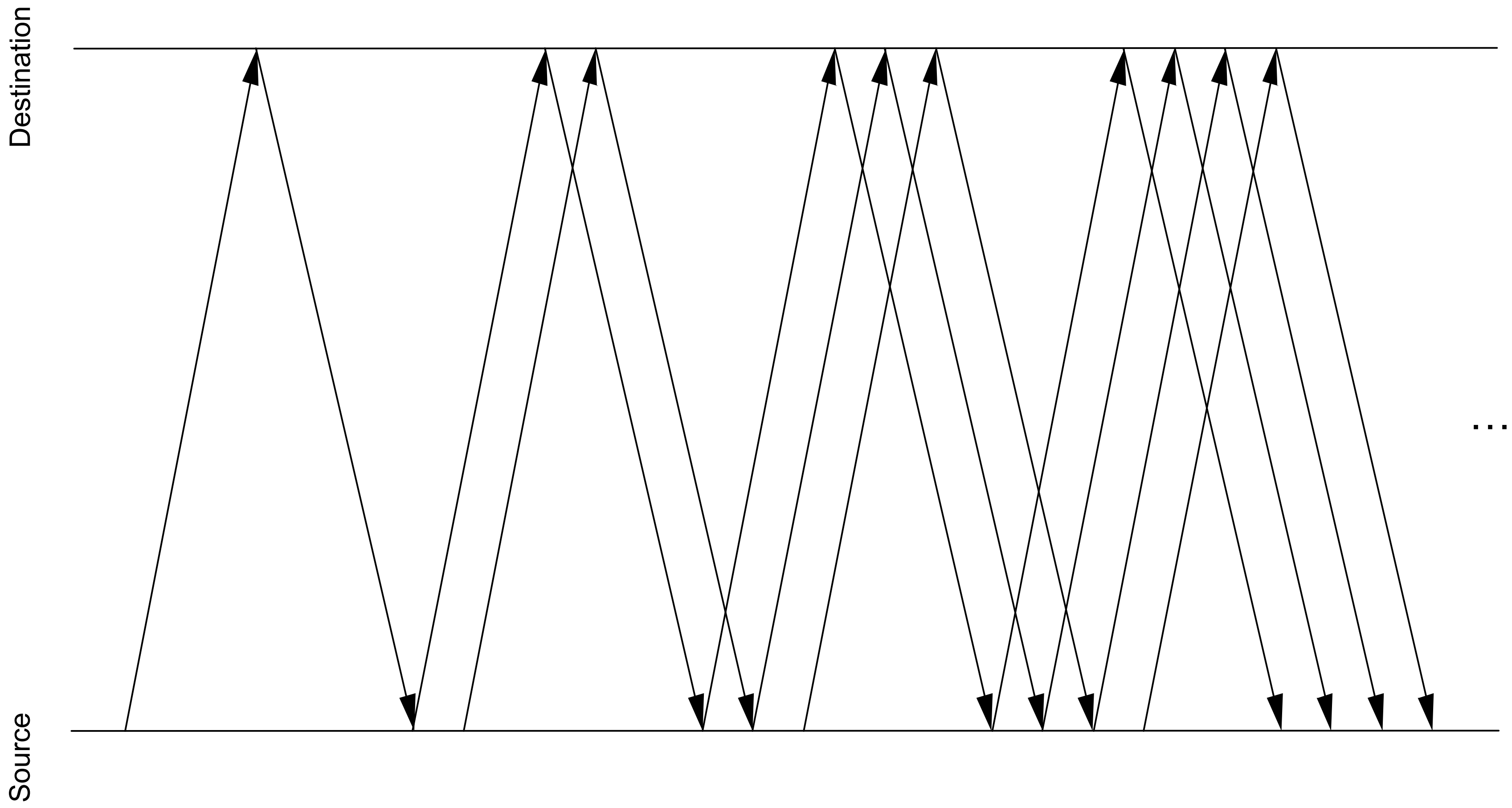
Setting the congestion window (cwnd)

- Increase cwnd conservatively
- Decrease cwnd aggressively
  - When loss detected, cut in half!
  - Multiplicative decrease
- Cwnd increase has two phases:
  - Additive phase (when pipe is full)
  - Multiplicative phase (when pipe is empty)
    - Called “slow start”!

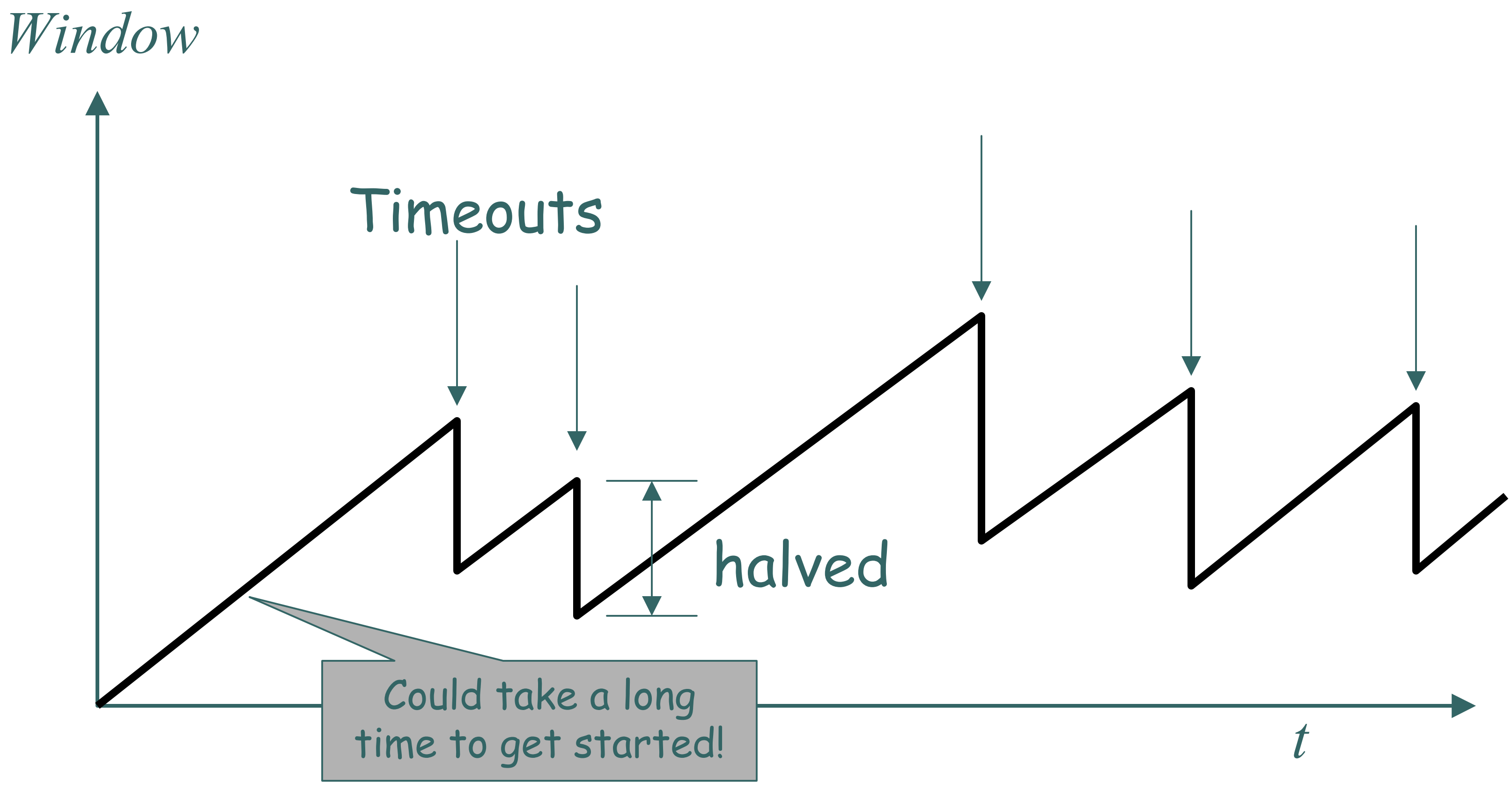
AIMD: Additive Increase Multiplicative Decrease

- Used when pipe is full
- Every RTT, add one “packet” to the cwnd
  - Actually, one MSS worth of bytes
  - Since multiple ACKs per RTT, a fraction of MSS added per ACK
- If loss detected (timeout or duplicate ACKs), decrease cwnd by half

Additive Increase



The famous AIMD sawtooth

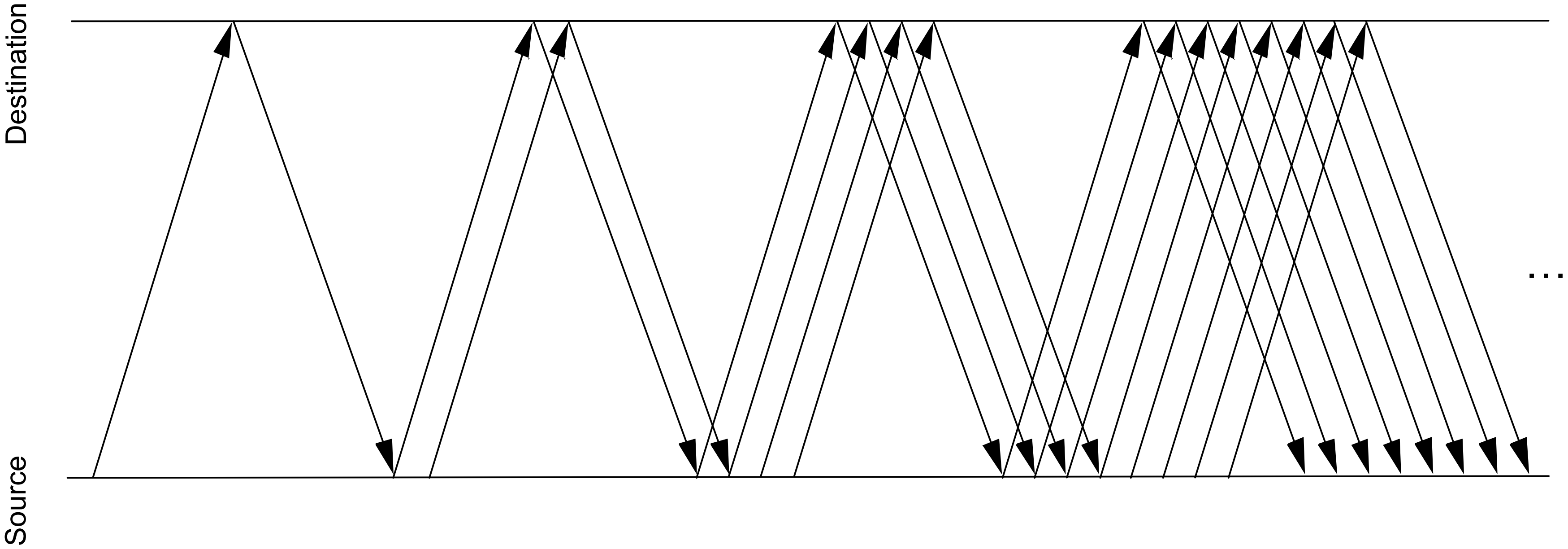


Could take a long time to get started!

“Slow start”

- Additive increase takes too long to fill pipe when pipe is empty
  - i.e. at the beginning of a connection
- During slow start, double the cwnd every RTT
  - Increase the cwnd for every ACK received

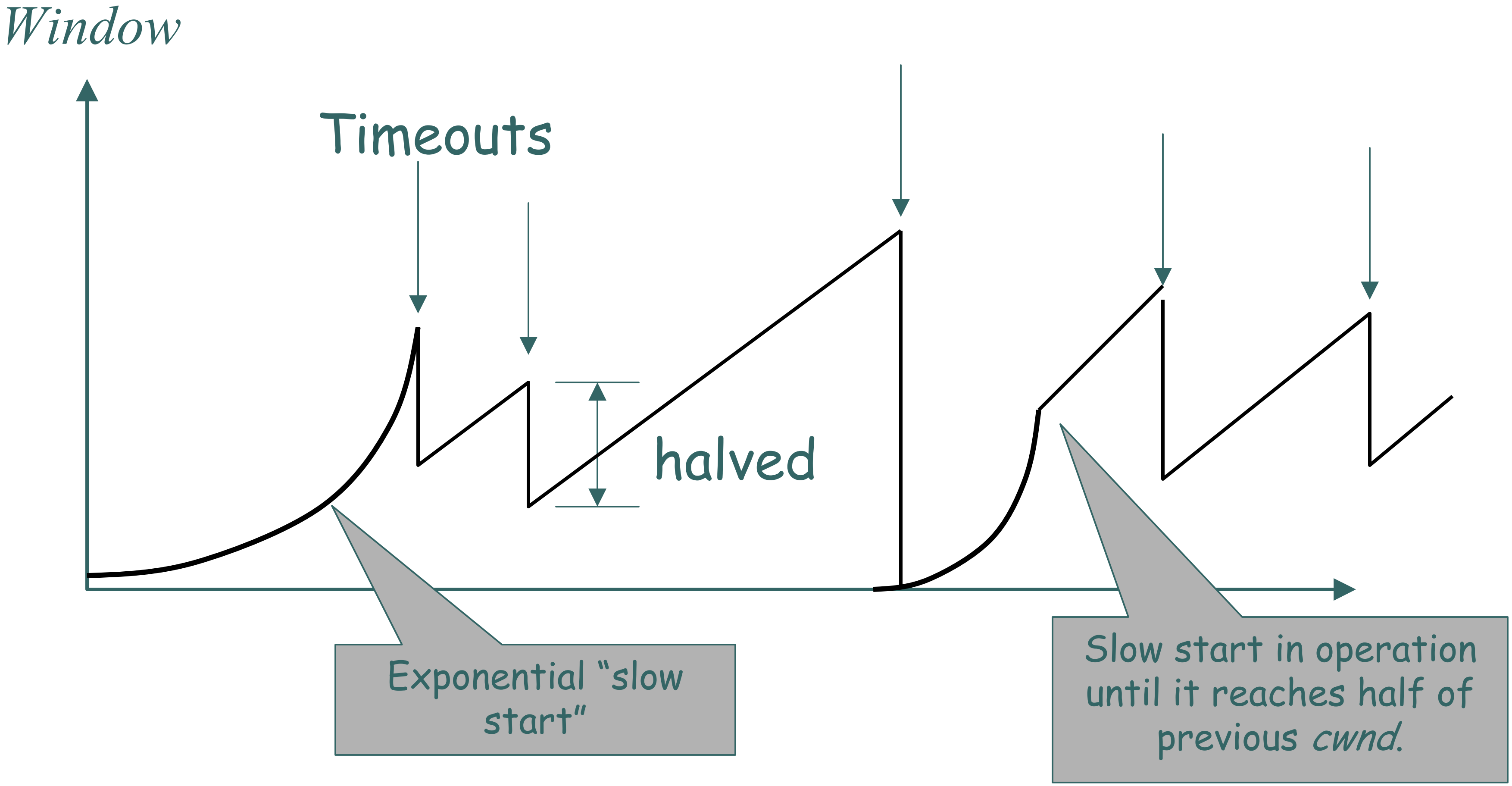
Slow start



Two reasons for an empty pipe

- Beginning of the connection
  - In this case, do “slow start” until packet loss
- Restart after a “stalled connection”
  - If timeout, then the pipe is empty
  - In this case, we remember the previous cwnd
  - Do slow start until cwnd reaches 1/2 the previous cwnd, then do additive

Slow start



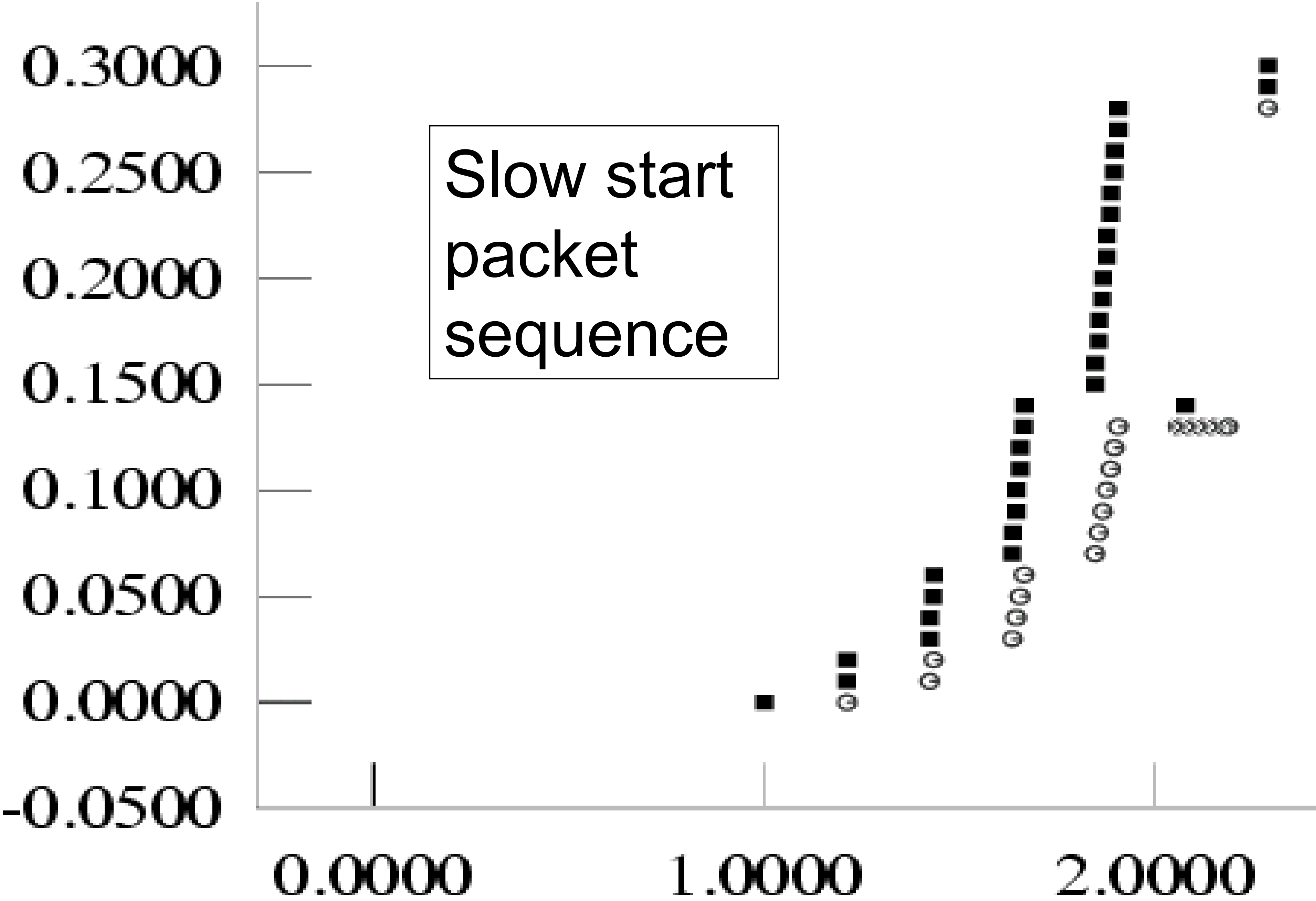
Window

Timeouts

halved

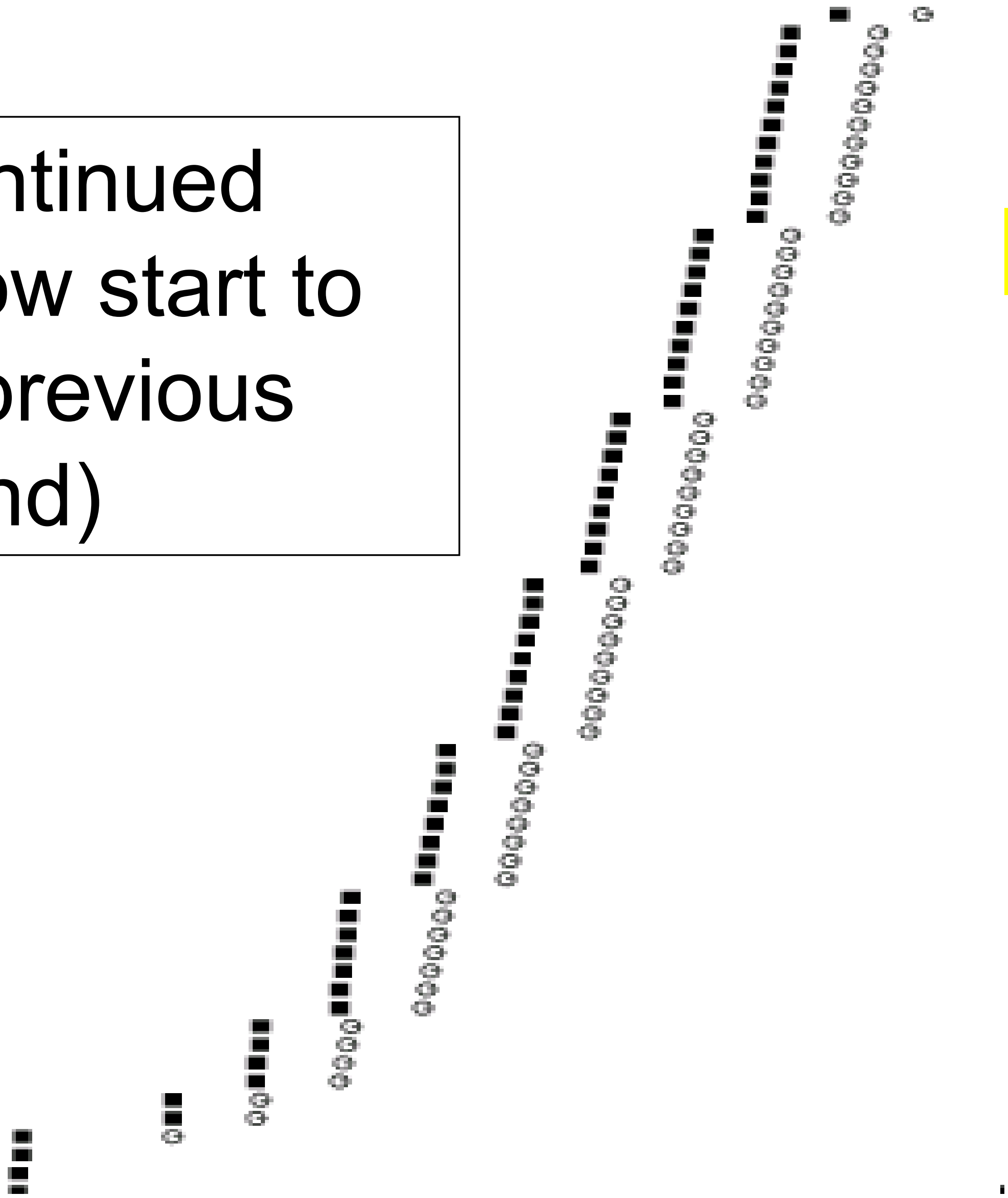
Exponential "slow start"

Slow start in operation until it reaches half of previous *cwnd*.



0.3000
0.2500
0.2000
0.1500
0.1000
0.0500
0.0000
-0.0500
Slow start packet sequence
0.0000
1.0000
2.0000

Continued (slow start to ½ previous cwnd)



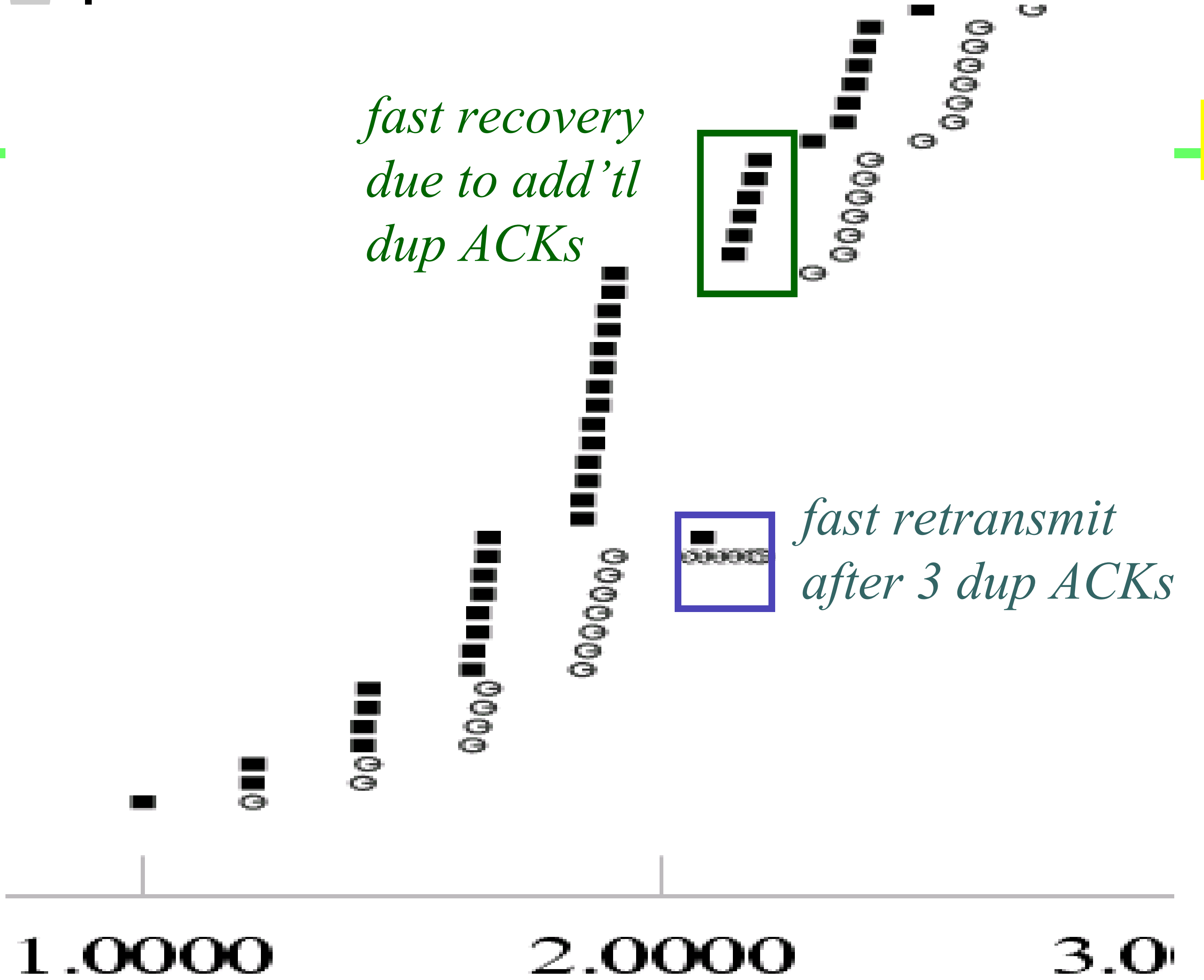
Fast Recovery

- Recall fast retransmit
    - Retransmit after three duplicate ACKs (don’t wait for a timeout)
- We can also use the duplicate ACKs to avoid dropping all the way back to slow start
- This is called fast recovery (always implemented as part of fast retransmit)

Fast Retransmit and Recovery

- If we get 3 duplicate acks for segment N
  - Retransmit segment N
  - Set ssthresh to 0.5*cwnd
  - Set cwnd to ssthresh + 3
- For every subsequent duplicate ack
  - Increase cwnd by 1 segment
- When new ack received
  - Reset cwnd to ssthresh (resume congestion avoidance)

Fast Recovery Example



fast recovery due to add’tl dup ACKs

fast retransmit after 3 dup ACKs

TCP performance again…

- TCP performs poorly if the pipe empties
- The pipe empties if a timeout occurs
- A timeout occurs if not enough packets were sent after the lost packet to trigger fast retransmit
- Unfortunately, drop-tail is likely to drop the last packets of a burst!
  - Drop-tail is router drop policy that drops all packets that overflow the queue

Random Early Detection (RED)

- Modifies the router drop policy to make TCP perform better
- Drop occasional packets before the queue is full, to avoid dropping many packets from a burst
- Select packet to drop randomly, so that a given burst will have only a single packet dropped
  - And later packets passed to trigger fast retransmission

RED

- Queue has two thresholds, min and max
- If queue below min, don’t drop
- If queue above max, drop all received packets
- If queue between min and max, drop received packet with some probability
  - Increase probability with time from last drop

But why drop at all???? Congestion Avoidance

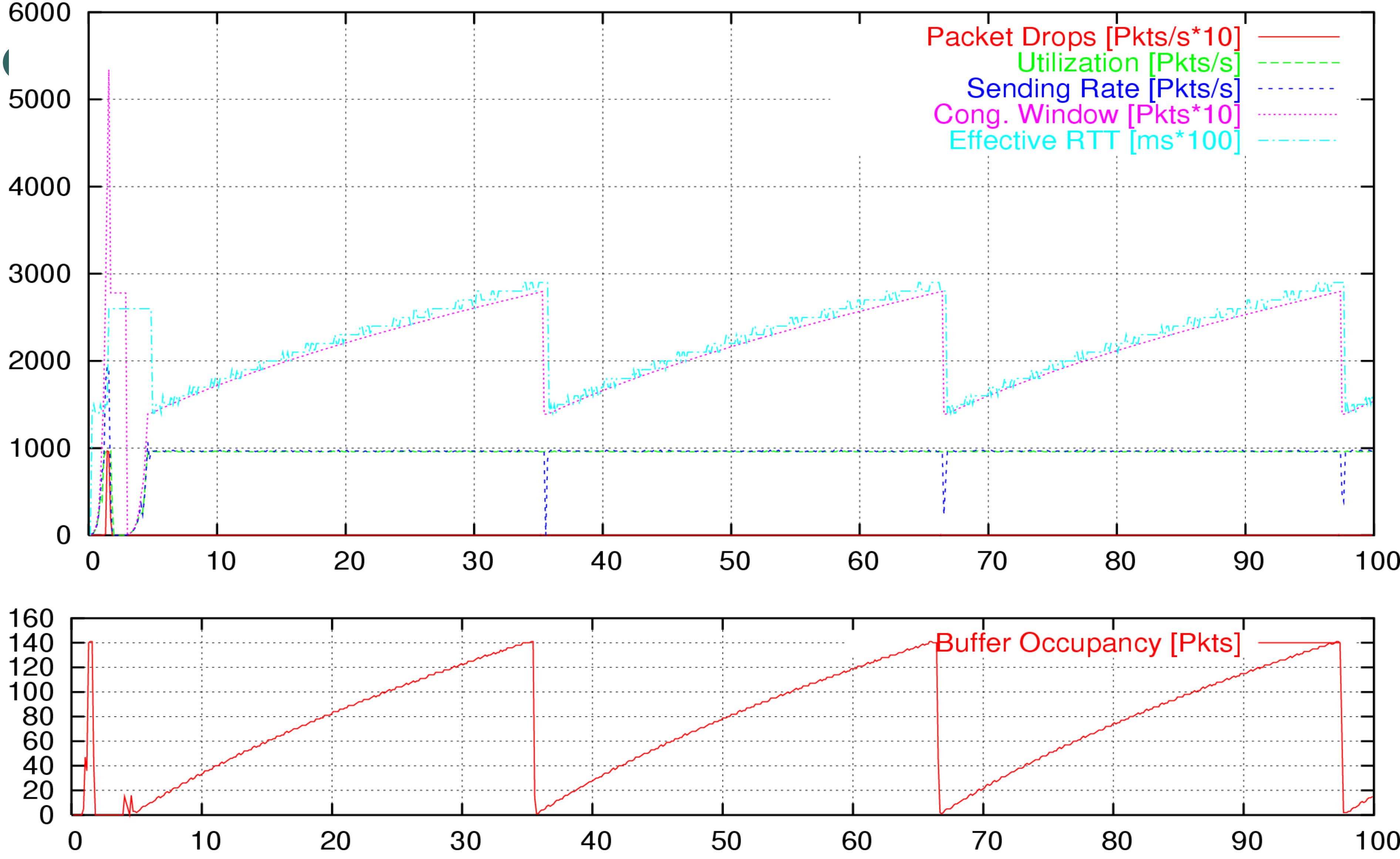
- Dropping not so bad for a long file transfer
- But can be noticeable for interactive applications
- Would be nice to avoid dropping at all
- Two ways to avoid dropping:
    - Explicit Congestion Notification from routers
    - Detect increasing queues before a drop occurs

ECN (Explicit Congestion Notification)

- DECNet had this back in the 80s!
  - “DEC bit”
- Router sets a bit in the packet if queues are above a threshold
  - Receiver echoes bit back to sender
- ECN is an IETF standard for use in conjunction with RED (RFC3168)
  - Two IP TOS bits defined for this
  - Plus new flags defined for TCP

Source-based congestion avoidance

- A router's queue starts to fill when the outgoing link capacity is reached
- When this happens, the sender will see:
  - Constant throughput (because capacity has been reached)
  - Increasing RTT (because of increasing wait in router queue)
- Sender can look for this, and back-off before packets are dropped!



TCPSIM: Time evolution of a TCP flow#(RTT 142ms, BW 8000kb, buffer 142 pkts of 1000 bytes)

Additional TCP issues

- TCP assumes that a timeout is the result of a lost packet due to congestion
- But, on many wireless links, timeouts occur because of temporary bad reception
- We don’t want the sender to back-off!
- Often a TCP-aware box placed at the Internet-wireless interface can trick the sender into not backing off

Additional TCP issues

- TCP performs poorly on very large delay X bandwidth pipes
- Takes too long to fill the pipe (slow start)
  - Performance dominated by RTT
- In this case, would like routers to tell the sender what cwnd to use from the start!
  - XTP

TCP status

- Most TCP is still slow-start, AIMD, fast-retransmit/fast-recovery
- RED implemented, but rarely turned on
- TCP issues remain as wireless is more pervasive, and as pipes get fatter and longer