Transport Layer: TCP Congestion Control & Buffer Management

- Congestion Control
  - What is congestion? Impact of Congestion
  - Approaches to congestion control
- TCP Congestion Control
  - End-to-end based: implicit congestion inference/notification
  - Two Phases: slow start and congestion avoidance
  - CongWin, theshold, AIMD, triple duplicates and fast recovery
  - TCP Performance and Modeling; TCP Fairness Issues
- Router-Assisted Congestion Control and Buffer Management
  - RED (random early detection)
  - Fair queueing

Readings: Sections 6.1-6.4

What is Congestion?

- Informally: "too many sources sending too much data too fast for *network* to handle"
- Different from flow control!
- Manifestations:
  - Lost packets (buffer overflow at routers)
  - Long delays (queuing in router buffers)

Effects of Retransmission on Congestion

- Ideal case
  - Every packet delivered successfully until capacity
  - Beyond capacity: deliver packets at capacity rate
- Realistically
  - As offered load increases, more packets lost
    - More retransmissions → more traffic → more losses …
  - In face of loss, or long end-end delay
    - Retransmissions can make things worse
    - In other words, no new packets get sent!
  - Decreasing rate of transmission in face of congestion
    - Increases overall throughput (or rather "goodput") !

Congestion: Moral of the Story

- When losses occur
  - Back off, don't aggressively retransmit
    i.e., be a nice guy!
- Issue of fairness
  - "Social" versus "individual" good
  - What about greedy senders who don't back off?

Approaches towards Congestion Control

Two broad approaches towards congestion control:

End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed as loss, delay
- approach taken by TCP

Network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at

TCP Approach

- End to End congestion control:
  - How to limit, - How to predict, - What algorithm?
- Basic Ideas:
  - Each source "determines" network capacity for itself
  - Uses implicit feedback, adaptive congestion window
    - Packet loss is regarded as indication of network congestion!
  - ACKs pace transmission ("self-clocking")
- Challenges
  - Determining available capacity in the first place
  - Adjusting to changes in the available capacity
    - Available capacity depends on # of users and their traffic, which vary over time!

TCP Congestion Control

- Changes to TCP motivated by ARPANET congestion collapse
- Basic principles
  - AIMD
  - Packet conservation
  - Reaching steady state quickly
  - ACK clocking

TCP Congestion Control Basics

- "probing" for usable bandwidth:
  - ideally: transmit as fast as possible (**Congwin** as large as possible) without loss
  - *increase* **Congwin** until loss (congestion)
  - loss: *decrease* **Congwin**, then begin probing (increasing) again
- two "phases"
  - slow start
  - congestion avoidance
- important variables:
  - **Congwin**
  - **Congwin threshold:** defines threshold between slow start and congestion avoidance phases
- Q: how to adjust Congwin?

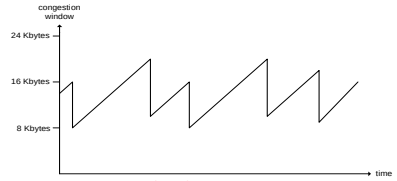
Additive Increase/Multiplicative Decrease

- Objective: Adjust to changes in available capacity
  - A state variable per connection: CongWin
    - Limit how much data source is in transit
  - MaxWin = MIN(RcvWindow, CongWin)
- Algorithm:
  - Increase CongWin when congestion goes down (no losses)
    - Increment CongWin by 1 pkt per RTT (linear increase)
  - Decrease CongWin when congestion goes up (timeout)
    - Divide CongWin by 2 (multiplicative decrease)

TCP AIMD

additive increase: increase **CongWin** by 1 MSS (max. seg. size) every RTT in the absence of loss events

multiplicative decrease: cut **CongWin** in half after loss event



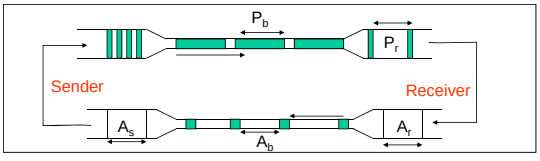
Long-lived TCP connection

Packet Conservation

- At equilibrium, inject packet into network only when one is removed
  - Sliding window (*not* rate controlled)
  - But still need to avoid sending burst of packets → would overflow links
    - Need to carefully pace out packets
    - Helps provide stability
- Need to eliminate spurious retransmissions
  - Accurate RTO estimation
  - Better loss recovery techniques (e.g., fast retransmit)

TCP Packet Pacing

- Congestion window helps to "pace" the transmission of data packets
- In steady state, a packet is sent when an ack is received
  - Data transmission remains smooth, once it is smooth
  - Self-clocking behavior



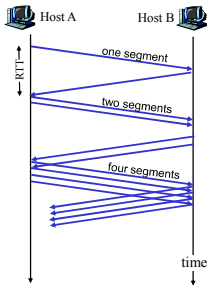
Why Slow Start?

- Objective
  - Determine the available capacity in the first place
    - Should work both for a CDPD (10s of Kbps or less) and for supercomputer links (10 Gbps and growing)
- Idea:
  - Begin with congestion window = 1 MSS
  - Double congestion window each RTT
    - Increment by 1 MSS for each ack
- Exponential growth, but slower than "one blast"
- Used when
  - first starting connection
  - connection goes dead waiting for a timeout

TCP Slowstart

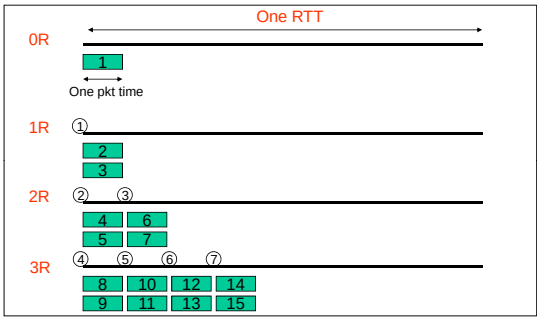
Slowstart algorithm

initialize: Congwin = 1
for (each segment ACKed)
  Congwin++
until (loss event OR
  CongWin > threshold)

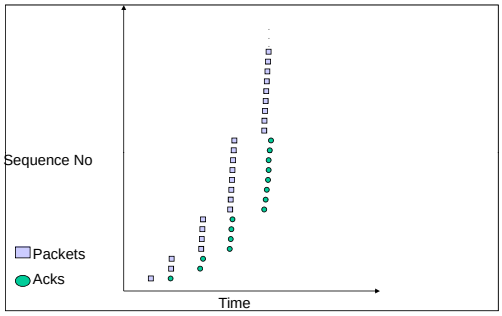


- exponential increase (per RTT) in window size (not so slow!)
- loss event: timeout (Tahoe TCP) and/or three duplicate ACKs (Reno TCP)

Slow Start Example

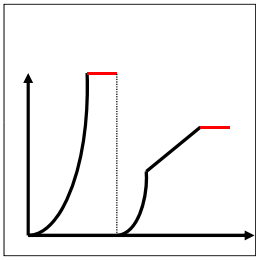


Slow Start Sequence Plot



Slow Start Packet Pacing

- How do we get this clocking behavior to start?
  - Initialize cwnd = 1
  - Upon receipt of every ack, cwnd = cwnd + 1
- Implications
  - Window actually increases to W in RTT * $\log_2(W)$
  - Can overshoot window and cause packet loss



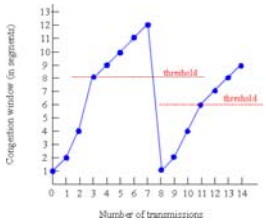
Congestion Avoidance: Basic Ideas

- If loss occurs when cwnd = W
  - Network can handle 0.5W ~ W segments
  - Set cwnd to 0.5W (multiplicative decrease)
- Upon receiving ACK
  - Increase cwnd by (1 packet)/cwnd
    - What is 1 packet? → 1 MSS worth of bytes
    - After cwnd packets have passed by → approximately increase of 1 MSS
- Implements AIMD with a "twist":
  - When timeout occurs, use a threshold parameter, and set it to 0.5W, and then return to slow start
  - Want to be more "conservative", as (long) timeout may cause us to lose self-clocking, i.e., "rate" we should inject packets into the network

TCP Congestion Avoidance (without Fast Recovery)

Congestion Avoidance

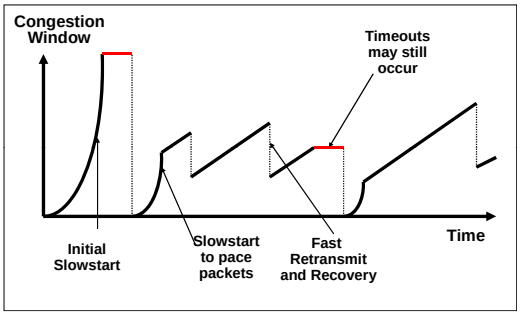
```
/* slowstart is over        */
/* Congwin > threshold */
Until (loss event) {
  every W segments ACKed:
     Congwin++
  }
Threshold: = Congwin/2
Congwin = 1
perform slowstart
```



Fast Retransmit/Fast Recovery

- Coarse-grain TCP timeouts lead to idle periods
- Fast Retransmit
  - Use duplicate acks to trigger retransmission
  - Retransmit after three duplicate acks
- After "triple duplicate ACKs", Fast Recovery
  - Remove slow start phase
  - Go directly to half the last successful CongWin
    - Ack clocking rate is same as before loss

TCP Saw Tooth Behavior



TCP Congestion Control: Recap

- end-end control (no network assistance)
- sender limits transmission: `LastByteSent-LastByteAcked ≤ CongWin`
- Roughly,

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- **CongWin** is dynamic, function of perceived network congestion

How does sender perceive congestion?

- loss event = timeout *or* 3 duplicate ACKs
- TCP sender reduces rate (**CongWin**) after loss event

three mechanisms:

- AIMD
- slow start
- conservative after timeout events

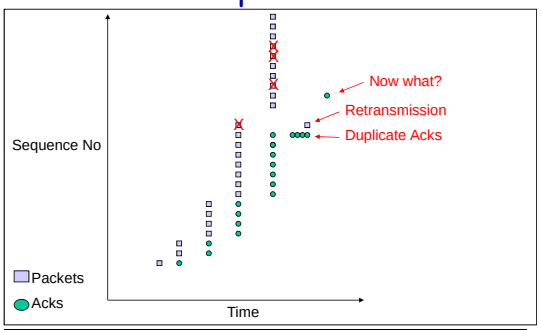
TCP Congestion Control: Recap (cont'd)

- When **CongWin** is below **threshold**, sender in slow-start phase, window grows exponentially.
- When **CongWin** is above **Threshold**, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACKs occurs, **threshold** set to **CongWin/2**, and **CongWin** set to **threshold**.
- When timeout occurs, **threshold** set to **CongWin/2**, and **CongWin** is set to 1 MSS.

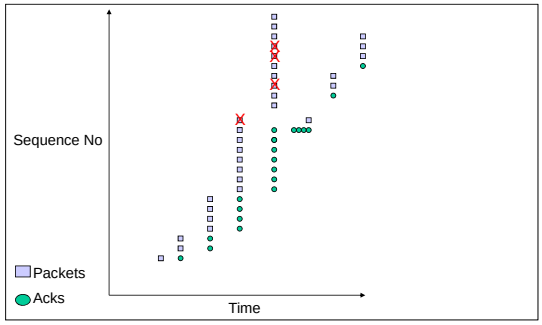
TCP Variations

- Tahoe, Reno, NewReno, Vegas
- TCP Tahoe (distributed with 4.3BSD Unix)
  - Original implementation of Van Jacobson's mechanisms (VJ paper)
  - Includes:
    - Slow start
    - Congestion avoidance
    - Fast retransmit

Multiple Losses



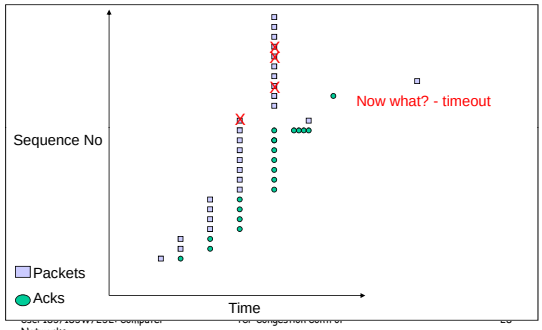
Tahoe



TCP Reno (1990)

- All mechanisms in Tahoe
- Addition of fast-recovery
  - Opening up congestion window after fast retransmit
- Delayed acks
- With multiple losses, Reno typically timeouts because it does not see duplicate acknowledgements

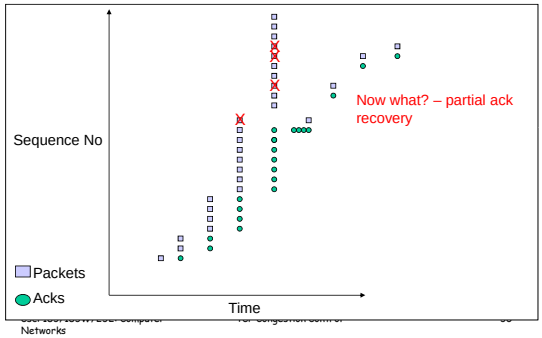
Reno



NewReno

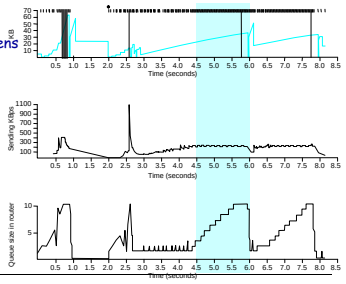
- The ack that arrives after retransmission (partial ack) could indicate that a second loss occurred
- When does NewReno timeout?
  - When there are fewer than three dupacks for first loss
  - When partial ack is lost
- How fast does it recover losses?
  - One per RTT

NewReno



TCP Vegas

- Idea: source watches for some sign that router's queue is building up and congestion will happen too; e.g.,
  - RTT grows
  - sending rate flattens



Algorithm

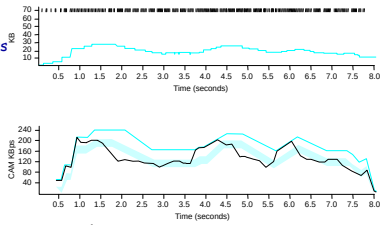
- Let **BaseRTT** be the minimum of all measured RTTs (commonly the RTT of the first packet)
- If not overflowing the connection, then

  ExpectRate = CongestionWindow/BaseRTT
- Source calculates sending rate (**ActualRate**) once per RTT
- Source compares **ActualRate** with **ExpectRate**

```
Diff = ExpectedRate - ActualRate
if Diff < α
    increase CongestionWindow linearly
else if Diff > β
    decrease CongestionWindow linearly
else
    leave CongestionWindow unchanged
```


Algorithm (cont)

- Parameters
  - α = 1 packet
  - β = 3 packets



- Even faster retransmit
  - keep fine-grained timestamps for each packet
  - check for timeout on first duplicate ACK

Changing Workloads

- New applications are changing the way TCP is used
- 1980's Internet
  - Telnet & FTP → long lived flows
  - Well behaved end hosts
  - Homogenous end host capabilities
  - Simple symmetric routing
- 2000's Internet
  - Web & more Web → large number of short xfers
  - Wild west – everyone is playing games to get bandwidth
  - Cell phones and toasters on the Internet
  - Policy routing

Short Transfers

- Fast retransmission needs at least a window of 4 packets
  - To detect reordering
- Short transfer performance is limited by slow start → RTT

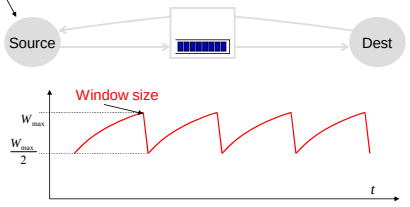
Short Transfers

- Start with a larger initial window
- What is a safe value?
  - Large initial window = min (4*MSS, max (2*MSS, 4380 bytes)) [rfc2414]
    - Not a standard yet
  - Enables fast retransmission
  - Only used in initial slow start not in any subsequent slow start

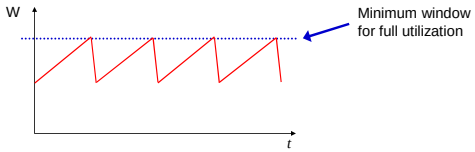
Impact of TCP Congestion Control on TCP Performance

Rule for adjusting W
- If an ACK is received: W ← W+1/W
- If a packet is lost: W ← W/2

Only W packets may be outstanding

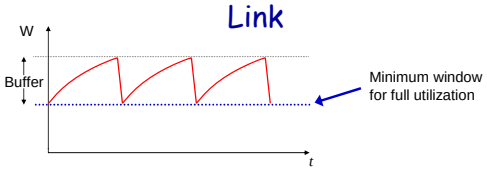


A Single TCP Flow over a Link with no Buffer



- The router can't fully utilize the link
  - If the window is too small, link is not full
  - If the link is full, next window increase causes drop
  - With no buffer it still achieves 75% utilization

A Single TCP Flow over a Buffered Link



- With sufficient buffering we achieve full link utilization
  - The window is always above the critical threshold
  - Buffer absorbs changes in window size
    - Buffer Size = Height of TCP Sawtooth
    - Minimum buffer size needed is 2T*C
  - This is the origin of the rule-of-thumb

TCP Performance in Real World

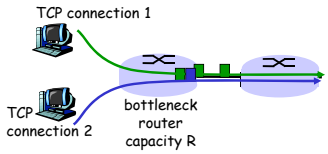
- In the real world, router queues play important role
  - Window is proportional to rate * RTT
    - But, RTT changes as well the window
  - "Optimal" Window Size (to fill links)
    - = propagation RTT * bottleneck bandwidth
    - If window is larger, packets sit in queue on bottleneck link

TCP Performance vs. Buffer Size

- If we have a large router queue → can get 100% utilization
  - But router queues can cause large delays
- How big does the queue need to be?
  - Windows vary from W → W/2
    - Must make sure that link is always full
    - W/2 > RTT * BW
    - W = RTT * BW + Qsize
    - Therefore, Qsize > RTT * BW
  - Large buffer can ensure 100% utilization
  - But large buffer will also introduce delay in the congestion feedback loop, slowing source's reaction to network congestion!

TCP Fairness

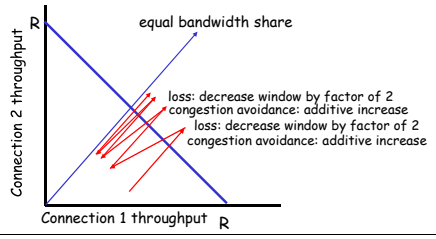
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why Is AIMD Fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



TCP Fairness

- BW proportional to 1/RTT?
- Do flows sharing a bottleneck get the same bandwidth?
  - NO!
- TCP is RTT fair
  - If flows share a bottleneck and have the same RTTs then they get same bandwidth
  - Otherwise, in inverse proportion to the RTT

TCP Fairness Issues

- Multiple TCP flows sharing the same bottleneck link do not necessarily get the same bandwidth.
  - Factors such as roundtrip time, small differences in timeouts, and start time, … affect how bandwidth is shared
  - The bandwidth ratio typically does stabilize
- Users can grab more bandwidth by using parallel flows.
  - Each flow gets a share of the bandwidth to the user gets more bandwidth than users who use only a single flow

TCP (Summary)

- General loss recovery
  - Stop and wait
  - Selective repeat
- TCP sliding window flow control
- TCP state machine
- TCP loss recovery
  - Timeout-based
    - RTT estimation
  - Fast retransmit
  - Selective acknowledgements

TCP (Summary)

- Congestion collapse
  - Definition & causes
- Congestion control
  - Why AIMD?
  - Slow start & congestion avoidance modes
  - ACK clocking
  - Packet conservation
- TCP performance modeling
  - How does TCP fully utilize a link?
    - Role of router buffers

Well Behaved vs. Wild West

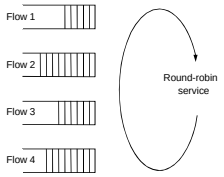
- How to ensure hosts/applications do proper congestion control?
- Who can we trust?
  - Only routers that we control
  - Can we ask routers to keep track of each flow
    - Per flow information at routers tends to be expensive
    - Fair-queuing later

Dealing with Greedy Senders

- Scheduling and dropping policies at routers
- First-in-first-out (FIFO) with tail drop
  - Greedy sender (in particular, UDP users) can capture large share of capacity
- Solutions?
  - Fair Queuing
    - Separate queue for each flow
    - Schedule them in a round-robin fashion
    - When a flow's queue fills up, only its packets are dropped
    - Insulates well-behaved from ill-behaved flows
  - Random Early Detection (RED) Router randomly drops packets w/ some prob., when queue becomes large!
    - Hopefully, greedy guys likely get dropped more frequently!

Queuing Discipline

- First-In-First-Out (FIFO)
  - does not discriminate between traffic sources
- Fair Queuing (FQ)
  - explicitly segregates traffic based on flows
  - ensures no flow captures more than its share of capacity
  - variation: weighted fair queuing (WFQ)

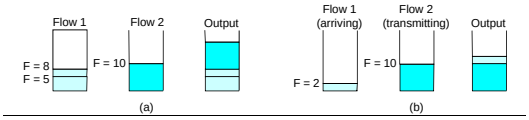


FQ Algorithm – Single Flow

- Suppose clock ticks each time a bit is transmitted
- Let P_i denote the length of packet i
- Let S_i denote the time when start to transmit packet i
- Let F_i denote the time when finish transmitting packet i
- $F_i = S_i + P_i$?
- When does router start transmitting packet i?
  - if before router finished packet $i-1$ from this flow, then immediately after last bit of $i-1$ (F_{i-1})
  - if no current packets for this flow, then start transmitting when arrives (call this A_i)
- Thus: $F_i = MAX(F_{i-1}, A_i) + P_i$

FQ Algorithm (cont)

- For multiple flows
  - calculate F_i for each packet that arrives on each flow
  - treat all F_i's as timestamps
  - next packet to transmit is one with lowest timestamp
- Not perfect: can't preempt current packet
- Example

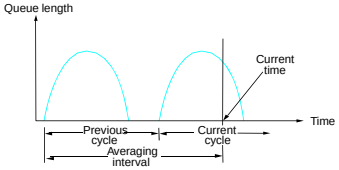


Congestion Avoidance

- 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*
- Two possibilities
  - router-centric: DECbit and RED Gateways
  - host-centric: TCP Vegas

DECbit

- Add binary congestion bit to each packet header
- Router
  - monitors average queue length over last busy+idle cycle, plus current busy cycle



  - set congestion bit if average queue length > 1
  - attempt 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*

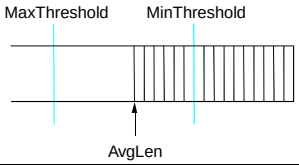
RED Details

- 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

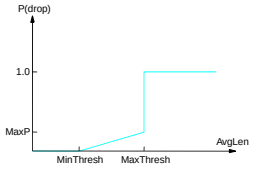
```
if AvgLen <= MinThreshold then
   enqueue the packet
if MinThreshold < AvgLen < MaxThreshold then
   calculate probability P
   drop arriving packet with probability P
if MaxThreshold <= AvgLen then
   drop arriving packet
```


RED Details (cont)

- Computing probability P

```
TempP = MaxP * (AvgLen - MinThreshold)/
  (MaxThreshold - MinThreshold)
P = TempP/(1 - count * TempP)
```

- Drop Probability Curve



Tuning RED

- Probability of dropping a particular flow's packet(s) is roughly proportional to the share of the bandwidth that flow is currently getting
- **`MaxP`** is typically set to 0.02, meaning that when the average queue size is halfway between the two thresholds, the gateway drops roughly one out of 50 packets.
- If traffic id bursty, then **`MinThreshold`** should be sufficiently large to allow link utilization to be maintained at an acceptably high level
- Difference between two thresholds should be larger than the typical increase in the calculated average queue length in one RTT; setting **`MaxThreshold`** to twice **`MinThreshold`** is reasonable for traffic on today's Internet

Congestion Control: Summary

- Causes/Costs of Congestion
  - On loss, back off, don't aggressively retransmit
- TCP Congestion Control
  - Implicit, host-centric, window-based
  - Slow start and congestion avoidance phases
  - Additive increase, multiplicative decrease
- Queuing Disciplines and Route-Assisted
  - FIFO, Fair queuing, DECBIT, RED

Transport Layer: Summary

- Transport Layer Services
  - Issues to address
  - Multiplexing and Demultiplexing
- UDP: Unreliable, Connectionless
- TCP: Reliable, Connection-Oriented
  - Connection Management: 3-way handshake, closing connection
  - Reliable Data Transfer Protocols:
    - Stop&Wait, Go-Back-N, Selective Repeat
    - Performance (or Efficiency) of Protocols
  - Estimation of Round Trip Time
- TCP Flow Control: receiver window advertisement
- Congestion Control: congestion window
  - AIMD, Slow Start, Fast Retransmit/Fast Recovery
  - Fairness Issue