TCP Congestion Control and Performance

CSC 343·643

WAKE FOREST UNIVERSITY
Department of Computer Science

Fall 2013

TCP Congestion Control

Control TCP connections sharing the bandwidth of a congested link

- TCP congestion control is closed-loop and uses implicit feedback
- Transmission rate limited by the window size, w
  - Number of bytes that can be sent without ACKs
- General operation
  - Transmit as fast as possible as long as no segments are lost
  - TCP starts with a small w, and increases slowly over time
  - Once a segment is lost, w is reduced quickly
  - Process is repeated during the connection lifetime

Why is this implicit feedback, not explicit feedback?

TCP Throughput

- An important measure of TCP performance is throughput
  - Rate at which data transmitted from sender to receiver
- Throughput will depend on w
  - If w segments sent back-to-back, must wait RTT to send again
  - If w bytes every RTT seconds, the throughput (bytes/sec) is

$$\frac{w}{\mathsf{RTT}}$$

- Consider k TCP connections traversing a link that has capacity r
  - Assume **no** UDP connections use the link, and each TCP connection requires more bandwidth than r

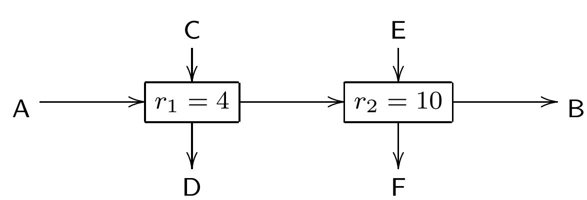
    Why no UDP?
  - Ideally, each connection should receive $\frac{r}{k}$

- Consider a connection that passes through n links
  - Assume link i has capacity r_i and k_i connections use the link
  - The maximum throughput for the connection would be

$$\min\left\{\frac{r_1}{k_1}, \frac{r_2}{k_2}, \ldots, \frac{r_n}{k_n}\right\}$$

  - This is called a **max-min fair** allocation

Given the connections $A \to B$, $C \to D$, and $E \to F$, what is the max-min allocation?



TCP Congestion Control Overview

- Each side of a TCP connection maintains
  - Send and receive buffers
  - Variables (`lastByteSent`, `rcvWindow`,...)
- There are two additional variables to maintain
  - `congWindow` - Another constraint on the amount sent
  - `threshold` - How quickly `congWindow` grows
- The amount of unACKed data that can be sent is

$$\texttt{lastByteSent} - \texttt{lastByteACKed} \leq \min\{\texttt{congWindow}, \texttt{rcvWindow}\}$$

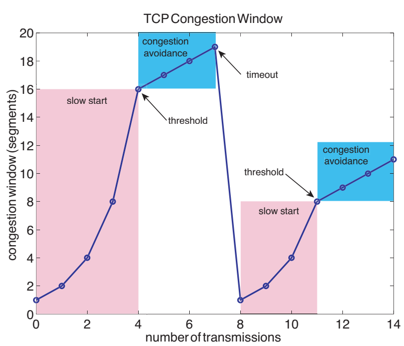
Therefore, amount that can be sent is bound by **two windows**
  - `rcvWindow`, determined by the receiver
  - `congWindow`, determined by congestion

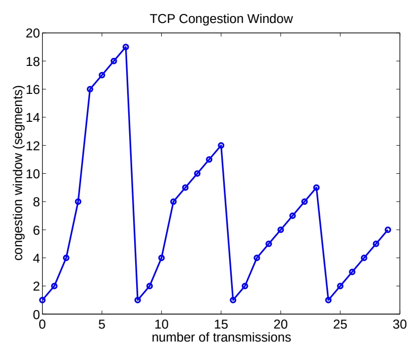
Example Operation

- Assume station A sends to station B using a TCP connection
  1. Station A takes one MSS from the send buffer and sends to B
     - Initially `congWindow` equals one MSS (bytes)
  2. If the segment is ACKed before the timeout, `congWindow` is increased by one MSS
  3. Station A sends two MSS to station B
  4. If these segments are ACKed before their timeout, `congWindow` increased by one MSS per ACKed segment
  5. Station A sends 4 MSS, ...
- The procedure continues as long as
  - `congWindow` $<$ `threshold`
  - ACKs arrive before timeout

- During the *initialization* `congWindow` increases **exponentially**
  - `congWindow` = 2 after 1 RTT
  - `congWindow` = 4 after 2 RTT
  - `congWindow` = 8 after 3 RTT
- This is called **slow start**, because the window size starts small (although increases quickly)
- Once `threshold` is passed, `congWindow` increases linearly
  - Increase `congWindow` by one MSS every RTT
  - This phase is called **congestion avoidance**

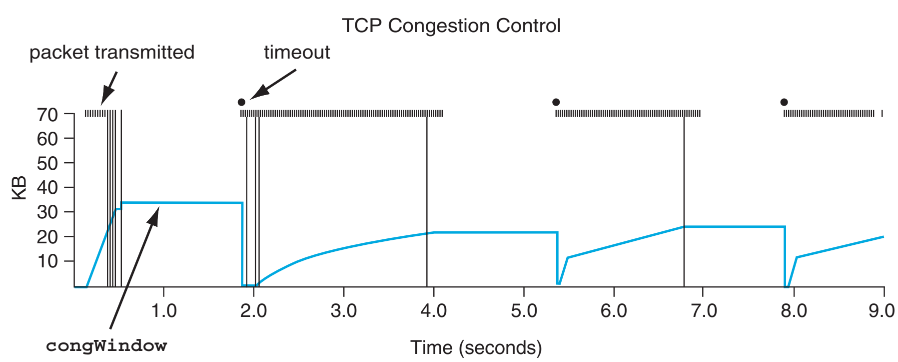
- A timeout indicates congestion
  - Set `threshold` = `threshold`/2 and `congWindow` = 1 (MSS)





TCP Tahoe

- The previous congestion mechanism is TCP **Tahoe**, rules are
  1. If congWindow $<$ threshold, **slow start** phase, congWindow grows exponentially
  2. If congWindow $>$ threshold, **congestion avoidance** phase, congWindow grows linearly
  3. Whenever a timeout occurs, threshold $=$ threshold$/2$ and congWindow $= 1$
- This process continues for the duration of the connection
  - However timeouts are lengthy, need to detect congestion earlier

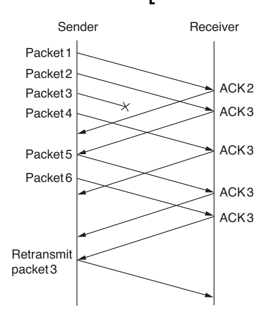


- Two other variations of Tahoe exist
  - **Reno** (implemented by most OS) and **Vegas**
  - Both attempt to improve on the performance of Tahoe

TCP Reno

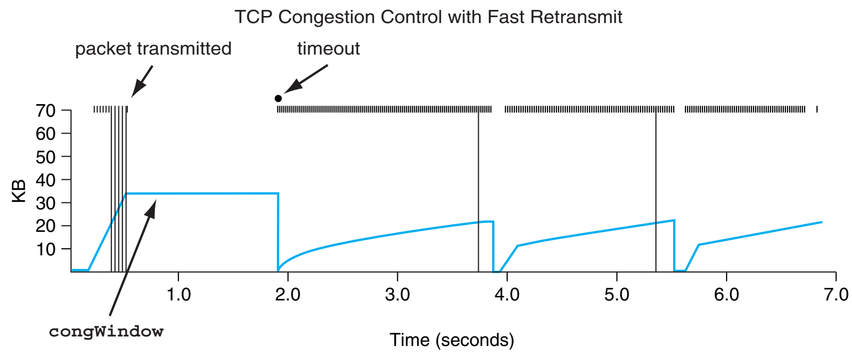
- Attempts to adjust to congestion before timeout
  - Use timeouts and duplicate ACKs to indicate congestion
- *Why would a sender receive duplicate ACKs?*
  - TCP does **not** allow NACKs
  - If a segment is received out-of-order, receiver re-ACKs the last in-order byte (duplicate ACKs)
  - N.B. This means other segments are being received, so congestion *may* exist
- If the sender receives 3 duplicate ACKs
  - This signals the sender to send the segment following the segment that has been ACKed three times

  - Hopefully this occurs before the missing segment timeout
  - This is called **fast retransmit** [RFC 2581]



- After **fast retransmit** following steps are taken in TCP Reno[a]
  - TCP moves to congestion avoidance phase
  - Set $\texttt{threshold} = \texttt{threshold}/2$
  - Set $\texttt{congWindow} = \texttt{congWindow}/2$
  - This is called **fast recovery**

[a]Simplification, details in RFC[2581]



- After a timeout, same action taken as Tahoe
- Generally speaking (no slow start) TCP increases `congWindow` by 1 MSS per RTT, then halves the value once path is congested
  – Called **Additive Increase Multiplicative Decrease** (AIMD)

TCP Vegas

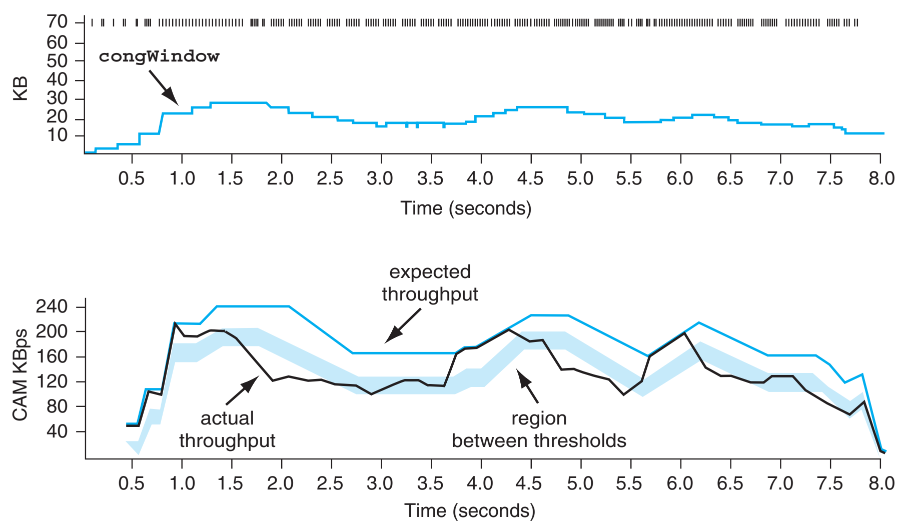
- Measures throughput, as well as duplicate ACKs and timeouts
- In Tahoe and Reno the congestion avoidance mechanism increases `congWindow` over time, decreases if there is a loss
- Vegas computes the *expected throughput* of the connection as

$$\texttt{expectedThru} = \frac{\texttt{congWindow}}{\min\left\{\texttt{sampleRTT}\right\}}$$

- The actual throughput (`actualThru`) is also calculated
  – Periodically measure the time to send a segment
- Vegas compares the two throughputs and adjusts `congWindow`

- Let, $\delta = \texttt{expectedThru} - \texttt{actualThru}$
- In addition define two thresholds α and β where $\alpha < \beta$
- The difference in throughput is compared to the thresholds
  - if $\delta < \alpha$, increase `congWindow` linearly
  - if $\delta > \beta$, decrease `congWindow` linearly (avoid congestion)
  - if $\alpha < \delta < \beta$, `congWindow` remains the same
- Still performs multiplicative for segment loss
  - Vegas attempts to avoid this situation

TCP Vegas Congestion-Avoidance

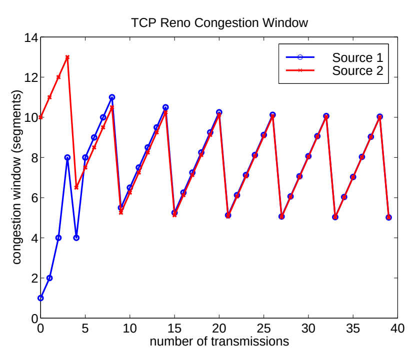


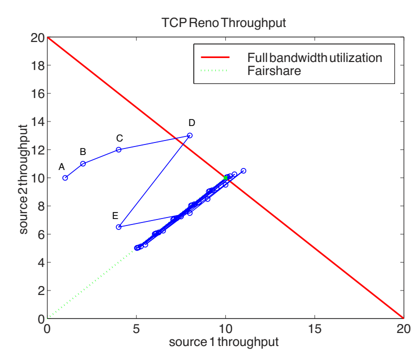
Recent TCP Congestion Control Algorithms

- Internet speeds are increasing and distances are longer
  - Characterized by large bandwidth and delay products (total number of packets in-flight)
  - The best performance requires a large congestion window
- Consider a short TCP connection
  - Tahoe and Reno may increase the window too slowly...
- Several *high-speed* TCP variants have been proposed
  - TCP CUBIC increases the window size aggressively when the window is far from the saturation point, and the slowly when it is close to the saturation point
  - Current default for Linux

TCP Congestion Fairness

- Share *bottleneck* link bandwidth *fairly* among TCP users
- *Do the AIMD algorithms achieve this goal?*
  - *What if users start with different window sizes?*
  - TCP does provide an equal share among competing users





- We have made several assumptions in this analysis
  - Only TCP connections traverse the link
  - All connections have the same RTT
  - Only one TCP connection per source-destination pair

  How does UDP traffic change fairness?

  Do the different TCP congestion algorithms change fairness?

  Suppose you needed more bandwidth (than your fair share) for TCP, how could this be done?

Macroscopic Description of TCP Dynamics

- Consider a sending a large file using TCP
- Lets ignore the slow start phase, assume the window grows linearly and once a loss is detected, the window is halved
  - This yields a *saw-tooth* behavior
- The rate at which TCP sends is a function of
  - The window w and the current RTT
  - The rate is w/RTT
- During this time TCP increases w until a loss occurs, denote this window size as w_l

- Assuming RTT and w_l remain approximately constant[a], transmission rate ranges

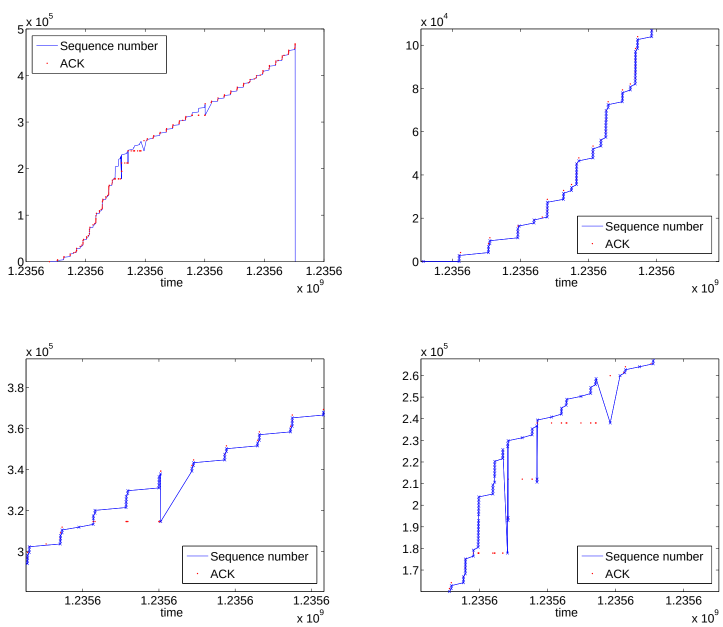
$$\left[\frac{\frac{1}{2} \times w_l}{\mathsf{RTT}}, \frac{w_l}{\mathsf{RTT}}\right]$$

- The average between these two extremes is

$$\frac{\frac{3}{4} \times w_l}{\mathsf{RTT}}$$

[a]Extremely simplified analysis.

TCP Window and Loss



TCP Congestion Control Weaknesses

1. Assumes any loss is an indication of congestion
   - Loss may be due to temporary router error
   - Wireless networks have higher loss rate due to transmission technology, sending more/less segments does not change performance
2. Assumes congestion is due to own window size
   - A malicious user could force others to reduce their rate
   - No differentiation
3. Detects congestion using loss, with no room for error
   - In steady state buffers are typically full
   - A short burst will cause everyone to reduce rate
4. Assumes all users experience same loss probability

TCP Window Size

- The TCP header defines the window field as 16 bits
  - The maximum window is 64KB

    Is this the flow or congestion window?
- This may be problematic for high speed links
  - T3 link (44.735 Mbps) takes only 12 msec to exhaust window
  - However, RTT maybe greater (50 msec across US)
  - Sender is idle ≈ 0.75 of the time
- A **window scale** option was proposed in RFC 1323
  - Negotiate larger window (during handshake)
  - Interpret window differently, specifically the number of bytes each bit represents (currently count bytes)
  - Maximum is $1,073,725,440$ bytes

Sending Data Using TCP

Consider a `telnet` session, where the user is using `vi`

- In the worse case
  - When a character arrives to the send buffer, new TCP segment (21 bytes) created given to IP (41 byte datagram)
  - Receiver side, TCP immediately sends 40 byte ACK

    Why is the ACK 40 bytes?
  - When `vi` processes character, it sends it back to sender (echo) which requires another 41 byte IP datagram
  - As a result, 162 bytes sent for each character
- One solution is to delay ACKs and rely on cumulative ACKs
  - Reduces the number of ACKs
  - Does not reduce sending overhead

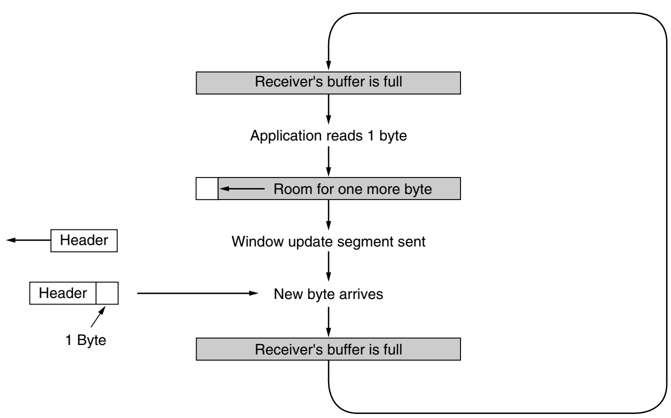
Nagle's Algorithm

- Nagle's algorithm is an another solution to the preceding problem
- When data comes one character at-a-time, into sending buffer
  - Send first character, buffer remaining until ACK received
  - Hopefully, a larger segment can be sent once ACK received

  What type of application does not work well with the algorithm?

Silly Window Syndrome

- Problem occurs when data is passed to sending entity in large blocks, but receiver application only reads one byte at-a-time
- Consider the following situation
  1. One byte read by receiver, TCP sends window update
     - Advertised window size is one byte
  2. Sender sends one byte
  3. Receiver ACKs, advertises window size is zero
  4. Application reads one byte...

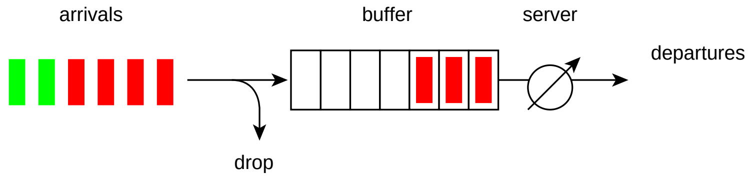


- Solution, prevent the receiver from advertising a window of 1 byte
  - Instead, wait until advertised window equals one MSS

Is Nagle's algorithm and the solution to silly window syndrome complements, or the same solution?

Early Drop

- We have described a router as a simple buffer and server system



  - Buffer stores packets from different connections (aggregated)
  - Buffer has finite space, if full arriving packets are dropped
  - This dropping pattern is called **tail dropping**
  - Possible one connection would lose more packets than others

    Which connection is this?

- Other packet-drop strategies are possible

  What is a dropping strategy for different classes of traffic?

- Early drop schedulers drop packets even if the buffer is not full
  - Suitable for networks where loss is used to adjust rates
  - Seek to distribute loss across all connections
  - Cooperative sources see lower delays
  - Uncooperative sources see severe packet loss
  - Encourage slower rate earlier, before multiple losses occur
- Two forms of early drop
  - Early random drop
  - Random early detection

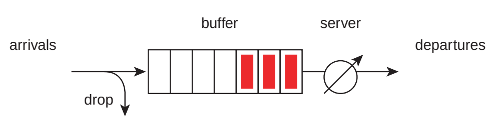
  What is an alternative to dropping packets early?

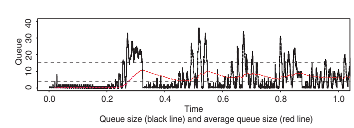
Early Random Drop

- Router monitors queue length
- Whenever the aggregate queue length reaches a threshold
  - This is an *instantaneous* measurement
  - Router drops each arriving packet with a given *drop probability*
  - For TCP segment (packet) loss represents congestion
- General idea misbehaving sources transmit more packets
  - More likely to drop misbehaving source packet than cooperative
- Target misbehaving source, without affecting cooperative sources

Random Early Detection

- Random Early Detection (RED) monitors queue length



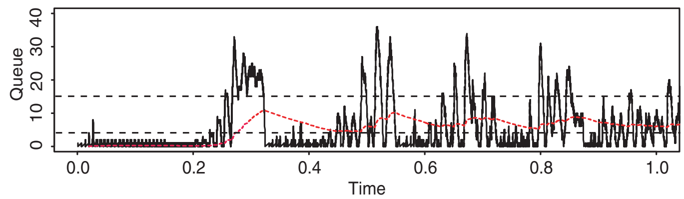


  - At thresholds, drop packets with a probability
  - The queue length measurement is an **average**
- Let `avgLength` be the average length of the queue

$$\texttt{avgLength} = (1-\alpha) \times \texttt{avgLength} + \alpha \times \texttt{sampleLength}$$

where $0 < \alpha < 1$ and `sampleLength` is the sampled queue length

- Due to the bursty nature of Internet traffic, an average is better
  - Queues can become full, then empty very quickly
  - The average *filters-out* any short-term congestion
  - However, the average will follow the long-term congestion
  - Therefore, the algorithm for computing `avgLength` determines the degree of burstiness allowed

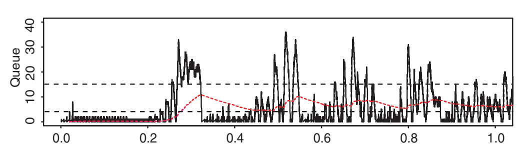


Queue size (black line) and average queue size (red line)

- Two additional variables, `minThreshold` and `maxThreshold`
  - Want the average queue length to be between these two values

- On every packet arrival, the following rules are applied

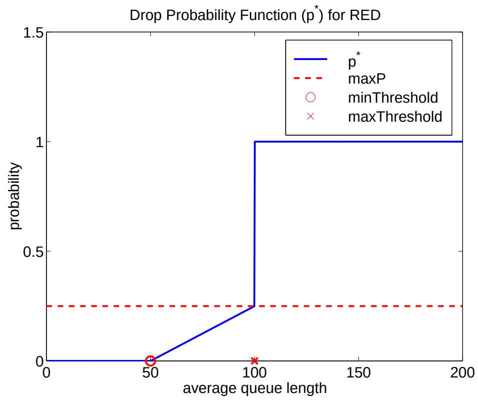
if `avgLength` < `minThreshold`
  queue packet
else if `minThreshold` < `avgLength` < `maxThreshold`
  calculate probability p
  drop packet with probability p
else
  drop packet



Queue size (black line) and average queue size (red line)

Figure 1: N.B. The variables `minThreshold` and `maxThreshold` compared against the queue length **average**

- The following graphs depicts a *drop probability*, **similar** to RED



Will RED ever have tail-drop behavior?

- The actual computation of p is slightly more complex

- Designers of RED wanted drops to be more widely distributed
  - Prevent clusters of drops (which would be from one source)
  - Better distribute drops over time
- Based on RED variables the the value of p is

$$p^* = \texttt{maxP} \times \frac{\texttt{avgLength} - \texttt{minThreshold}}{\texttt{maxThreshold} - \texttt{minThreshold}}$$

$$p = \frac{p^*}{1 - n \times p^*}$$

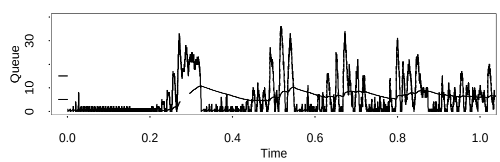
where n is the number of packets queued (not dropped) while `avgLength` has been between the two thresholds

  - The idea is for every packet that is not dropped, (while `avgLength` is between thresholds) probability increases

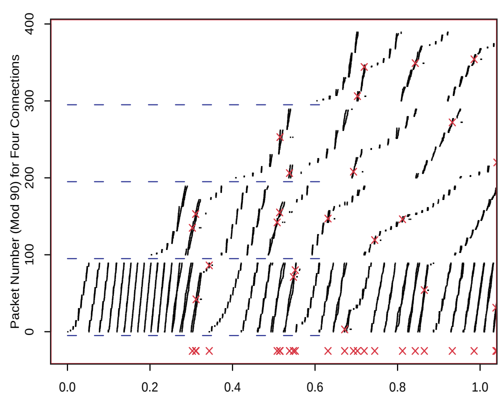
RED Performance

- Random nature of RED yields an interesting property
  - RED drops packets randomly
  - Will drop a connections packets roughly proportional to the connections allocation
- RED can yield *approximately* fair allocations
  - However, the allocations are not precise
- How are RED variable values determined
  - Given a traffic load that can be modeled using a stochastic process, the optimal RED variables can be determined
  - Typically maximize *power* (ratio of throughput to delay)
  - However, the analysis depends on the characterization of traffic, which may not be realistic

RED Performance



Queue size (solid line) and average queue size (dashed line).



A simulation with four FTP connections with staggered start times.

Internet Transport and Application Layers

Application	Application Layer Protocol	Transport Layer Protocol
Electronic mail	SMTP [RFC 821]	TCP
Remote terminal access	`telnet` [RFC 854]	TCP
Web	`http` [RFC 2616]	TCP
File transfer	`ftp` [RFC 959]	TCP
Remote file server	NFS	UDP or TCP
Streaming multimedia		UDP
Internet telephony		UDP