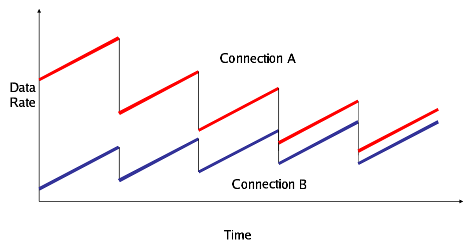
TCP Congestion Control

Mukul Goyal

Overview

- AIMD (Additive Increase Multiplicative Decrease)
- Slow Start and Congestion Avoidance
- Fast Retransmit/Fast Recovery
- Selective Acknowledgement (SACK)
- Forward Acknowledgement (FACK)
- TCP Vegas

Additive Increase Multiplicative Decrease (AIMD): Fair Distribution of Network Resources



TCP Variables

- The congestion window (cwnd) is a sender-side limit on the amount of data the sender can transmit into the network before receiving an acknowledgment (ACK)
- The receiver's advertised window (rwnd) is a receiver-side limit on the amount of outstanding data. The minimum of cwnd and rwnd governs data transmission.
- The slow start threshold (ssthresh), is used to determine whether the slow start or congestion avoidance algorithm is used to control data transmission. The slow start algorithm is used when cwnd < ssthresh, while the congestion avoidance algorithm is used when cwnd > ssthresh.
- SMSS: Sender's Maximum Segment Size

08/22/2003 12:17

Slow Start

- Beginning transmission into a network with unknown conditions requires TCP to slowly probe the network to determine the available capacity, in order to avoid congesting the network with an inappropriately large burst of data.
- Initial cwnd = 1 or 2 segments.
- The initial value of ssthresh MAY be arbitrarily high.

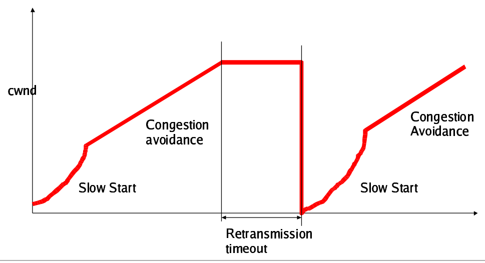
Slow Start and Congestion Avoidance

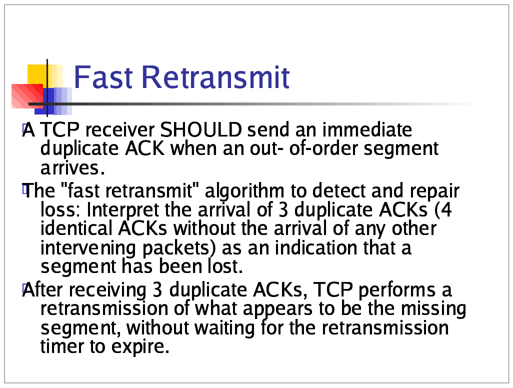
- During slow start, a TCP increments cwnd by at most SMSS bytes for each ACK received that acknowledges new data. Slow start ends when cwnd exceeds ssthresh or when congestion is observed.
- During congestion avoidance, cwnd is incremented by 1 full-sized segment per round-trip time (RTT). Congestion avoidance continues until congestion is detected.
- Slow start actually leads to exponential increase in cwnd. Cwnd doubles every RTT.
- The slow start algorithm is used at the beginning of a transfer, or after repairing loss detected by the retransmission timer.

Setting ssthresh and cwnd on Detecting Packet Loss Via Retransmission Timeout

- When a TCP sender detects segment loss using the retransmission timer, the value of ssthresh MUST be set as follows: ssthresh = max (FlightSize / 2, 2*SMSS)
- FlightSize is the amount of outstanding data in the network.
- Cwnd is set to 1 full-size segment.

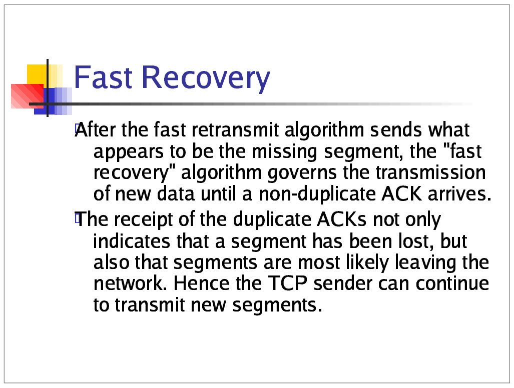
Congestion Control Via Retransmission Timeout





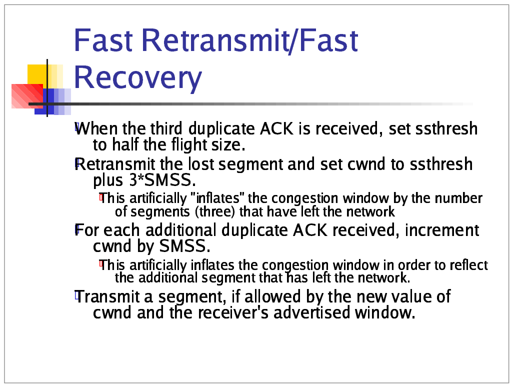
Fast Retransmit

- A TCP receiver SHOULD send an immediate duplicate ACK when an out- of-order segment arrives.
- The "fast retransmit" algorithm to detect and repair loss: Interpret the arrival of 3 duplicate ACKs (4 identical ACKs without the arrival of any other intervening packets) as an indication that a segment has been lost.
- After receiving 3 duplicate ACKs, TCP performs a retransmission of what appears to be the missing segment, without waiting for the retransmission timer to expire.



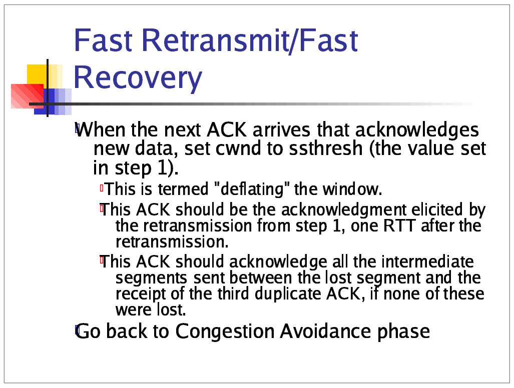
Fast Recovery

- After the fast retransmit algorithm sends what appears to be the missing segment, the "fast recovery" algorithm governs the transmission of new data until a non-duplicate ACK arrives.
- The receipt of the duplicate ACKs not only indicates that a segment has been lost, but also that segments are most likely leaving the network. Hence the TCP sender can continue to transmit new segments.



Fast Retransmit/Fast Recovery

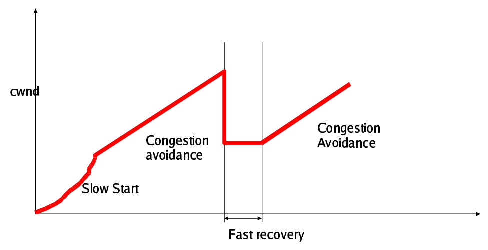
- When the third duplicate ACK is received, set ssthresh to half the flight size.
- Retransmit the lost segment and set cwnd to ssthresh plus 3*SMSS.
  - This artificially "inflates" the congestion window by the number of segments (three) that have left the network
- For each additional duplicate ACK received, increment cwnd by SMSS.
  - This artificially inflates the congestion window in order to reflect the additional segment that has left the network.
- Transmit a segment, if allowed by the new value of cwnd and the receiver's advertised window.



Fast Retransmit/Fast Recovery

- When the next ACK arrives that acknowledges new data, set cwnd to ssthresh (the value set in step 1).
  - This is termed "deflating" the window.
  - This ACK should be the acknowledgment elicited by the retransmission from step 1, one RTT after the retransmission.
  - This ACK should acknowledge all the intermediate segments sent between the lost segment and the receipt of the third duplicate ACK, if none of these were lost.
- Go back to Congestion Avoidance phase

Congestion Control Via Fast Retransmit/Fast Recovery



The Problem with Fast Retransmit/Recovery

- Fast retransmit/fast recovery is known to generally not recover very efficiently from multiple losses in a single flight of packets.
- Loss of 3 or more packets in a window typically results in retransmission timeout.

The NewReno Modification to TCP's Fast Recovery Algorithm

- In the case of multiple packets dropped from a single window of data, the first new information available to the sender comes when the sender receives an acknowledgement for the retransmitted packet (that is the packet retransmitted when Fast Retransmit was first entered).
- If there had been a single packet drop, then the acknowledgement for this packet will acknowledge all of the packets transmitted before Fast Retransmit was entered (in the absence of reordering). However, when there were multiple packet drops, then the acknowledgement for the retransmitted packet will acknowledge some but not all of the packets transmitted before the Fast Retransmit. We call this packet a **partial acknowledgment**.

The NewReno Algorithm

- When the third duplicate ACK is received and the sender is not already in the Fast Recovery procedure, set ssthresh to half the flight size
- Record the highest sequence number transmitted in the variable "recover".
- Retransmit the lost segment and set cwnd to ssthresh plus 3*MSS.
- For each additional duplicate ACK received, increment cwnd by MSS.

The NewReno Algorithm

- Transmit a segment, if allowed by the new value of cwnd and the receiver's advertised window.
- When an ACK arrives that acknowledges new data:
  - If this ACK acknowledges all of the data up to and including "recover", Set cwnd to ssthreshand exit the Fast Recovery procedure.
  - If this ACK does *not* acknowledge all of the data up to and including "recover", then this is a partial ACK. In this case, retransmit the first unacknowledged segment. **Do not exit the Fast Recovery procedure** (i.e., if any duplicate ACKs subsequently arrive, increment cwnd by MSS).

TCP Selective Acknowledgment (SACK)

- NewReno is not the complete solution to the poor performance when multiple packets are lost from one window of data.
- With the limited information available from cumulative acknowledgments, a TCP sender can only learn about a single lost packet per round trip time.
  - New Reno recovers from N losses in N round-trip times.
- A Selective Acknowledgment (SACK) mechanism, combined with a selective repeat retransmission policy, can help to overcome these limitations.
- The receiving TCP sends back SACK packets to the sender informing the sender of data that has been received. The sender can then retransmit only the missing data segments.

TCP Selective Acknowledgment (SACK)

- The SACK option is to be sent by a data receiver to inform the data sender of non-contiguous blocks of data that have been received and queued.
- Each contiguous block of data queued at the data receiver is defined in the SACK option by:
  - Left Edge of Block: This is the first sequence number of this block.
  - Right Edge of Block: This is the sequence number immediately following the last sequence number of this block.

How Many Blocks can be reported in a SACK Option

- A SACK option that specifies n blocks will have a length of 8*n+2 bytes, so the 40 bytes available for TCP options can specify a maximum of 4 blocks.
- It is expected that SACK will often be used in conjunction with the Timestamp option, which takes an additional 10 bytes (plus two bytes of padding); thus a maximum of 3 SACK blocks will be allowed in this case.

Sender Response to a SACK Option

- When an acknowledgment segment arrives containing a SACK option, the data sender will turn on the SACKed bits for segments that have been selectively acknowledged.
- After the SACKed bit is turned on (as the result of processing a received SACK option), the data sender will skip that segment during any later retransmission.
- Any segment that has the SACKed bit turned off and is less than the highest SACKed segment is available for retransmission.

FACK: Using SACK Information for Congestion Control

- TCP SACK option allows the receiver to specify additional information which the sender may use to better handle recovery from lost data.
- SACK information can be used to accurately retransmit lost segments.
- SACK information can also be used to refine TCP congestion control during recovery.

FACK: Using SACK Information for Congestion Control

- Use data in SACK blocks to precisely compute the amount of data outstanding in the network (awnd).
- Data is assumed to have left the network when either
  - it has been ACKed or SACKed, or
  - some segment following it has been SACKed

FACK Contd.

- Forward-most SACKed data is stored in a new variable, snd.fack. During non-recovery states,
  - snd.fack = snd.una
- If there have been no retransmissions, the amount of data in the network at any time is given by:
  - awnd = snd.nxt - snd.fack

FACK Contd.: Accounting for Retransmissions

- The state variable retran_data is increased when data is retransmitted
- retran_data is decreased when
  - Data is SACKed
- In general, then:
  - awnd = snd.nxt - snd.fack + retran_data
- While awnd < cwnd
  - SendSomeThing()

TCP Vegas

- TCP Reno: Reactive Congestion Control
- TCP Reno uses the loss of segments as a signal that there is congestion in the network. It has no mechanism to detect the incipient congestion – before losses occur – so that losses can be prevented.
- Proactive Congestion Control:
  - Increase in round trip delay is a symbol of incipient congestion.

TCP Vegas

- TCP Vegas detects incipient congestion by comparing the measured throughput rate with expected throughput rate.
- The Vegas idea: The number of bytes in transit is directly proportional to the expected throughput, and therefore as the window size increases – causing the bytes in transit to increase – the throughput of the connection should also increase.

TCP Vegas

- Measure and control the amount of extra data this connection has in transit.
  - The extra data – the data that would not have been sent if the bandwidth used by the connection exactly matched the available bandwidth in the network.
- If the connection is sending too much extra data, it is causing congestion.
- If it sends too little extra data, it may not take advantage of the extra bandwidth that recently became available in the network.

TCP Vegas

- BaseRTT: The RTT value when there is no congestion. Usually taken to be the minimum of all observed RTTs.
- Expected throughput = congestion window size/BaseRTT.
- Actual throughput = actual number of bytes sent during the round trip time of a particular packet/the round trip time

TCP Vegas

- Let diff = expected throughput – actual throughput
  - Diff has to be greater than 0 since expected = cwnd/min(rtt) and actual = cwnd/(a sample of rtt).
- Let A and B be two thresholds such that A < B
- If diff < A, increment cwnd by a segment
- If diff > B, decrement cwnd by a segment
- If A < diff < B, don't change the cwnd
- A and B set according to the number of extra packets acceptable in the network. E.g. A = 2 packets/min(RTT), B = 4 packets/min(RTT)

TCP Vegas During Slow Start

- If the ssthresh is set too high, TCP Reno is bound to lose half the cwnd worth of packets sooner or later.
  - Since the cwnd is doubled every rtt.
- That many loses will lead to expensive retransmission timeout.
- Vegas Solution: Double the cwnd every other RTT. In the meanwhile, apply the vegas congestion avoidance. Get out of slow start when **Expected – Actual** > **C** (another threshold).