Study of TCP Tahoe

Submitted By: WASEEM ULLAH
Submitted To: Engr Majid Ashraf
Department of Electrical Engineering
University of Engineering & Technology Peshawar

Abstract:

The purpose of this attempt is to analyze tcp tahoe and discuss different technique of congestion control and avoidance mechanisms presented for TCP/IP protocols, in Tahoe. Congestion is bad for the overall performance in the network. It can cause Excessive delays of data reception time it can also waste precious bandwidth. So the mechanism for determining when a segment should be transmitted and when it will be re-transmitted and how the sender should behave when it encounters congestion and what pattern of transmissions should it follow to avoid congestion.

TCP Tahoe:

TCP Tahoe was introduced for the first time in 1988. This was the first TCP implementation to include the congestion control mechanisms and round-trip-timing enhancements proposed by Van Jacobson in his paper "Congestion Avoidance and Control". These new algorithms were introduced in response to congestive collapses which began occurring on the Internet in 1986 and caused throughput to drop in some cases by a factor of a thousand.

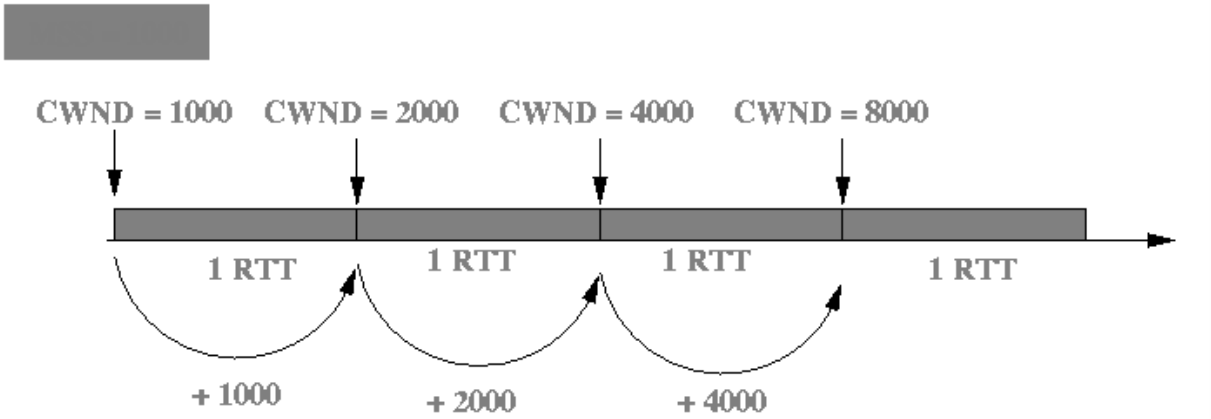
The goal of these mechanisms is to ensure that a TCP connection is able to reach a state of equilibrium and that the connection obeys the "conservation of packets principle" once it is in equilibrium. This principle states that once a connection has reached equilibrium, it should only transmit a packet on the network when it receives feedback indicating that a packet has left the network. I.e. if the connection is running at the available bandwidth capacity then a packet is not injected into the network unless a packet is taken out as well. TCP implements this principle by using the acknowledgements to clock outgoing packets because an acknowledgement means that a packet was taken off the wire by the receiver. It also maintains a congestion window CWD to reflect the network capacity. However there are certain issues, which need to be resolved to ensure this equilibrium.

1) Determination of the available bandwidth.
2) Ensuring that equilibrium is maintained.
3) How to react to congestion.

TCP Congestion control algorithm consisting of slow start phase & congestion avoidance phase.

Tahoe suggest that whenever tcp connection start or restart (after packet loss) it should go through a procedure called slow start. The reason for this slow start according to him was that the initial burst will overwhelm the channel and the connection might be lost. So in slow start a sender first send one congestion window then after receiving an acknowledgment from receiver congestion window is increased by one and in this manner conj wind increase by 1 after every ack. (This increase in conj wind will be upto some threshold value)

In the slow start phase, the CWND will increase as follows:



It first set CWND to one MSS (i.e., one packet) Whenever TCP receives new ACK packet, then *increases* CWND by *MSS* initially (at time 0), CWND = 1000 at time RTT (round trip time), CWND = 2000
At time 2 RTT, CWND = 4000 At time 3 RTT (not in figure), CWND = 8000And so on...
When we plot CWND over time, CWND will double after each RTT time when plotted out, the CWND increases exponentially with time
The slow start procedure is used in the following 2 situations, first When a TCP connection is first establish (first start transmitting) In this case, SSThresHold is set to (equal to) AWS. Second when TCP has detected a packet loss, in this case, SSThresHold will be set to CWND/2 (this is done by the congestion avoidance algorithm prior to entering in the slow start process) This slow start epoch can ended by 2 events:Ended normally when CWND becomes > SSThresh, In this case, TCP will enter the congestion control phase.

Ended abnormally if TCP detects a packet loss during the slow start phase In this case, TCP will first sets SSThresh = CNWD/2 (CWND is the transmit window size that TCP was using at the time that the packet loss was detected) Then TCP re-start the slow start procedure I.e., set CWND = 1 and increase CWND by 1 for every new ACK

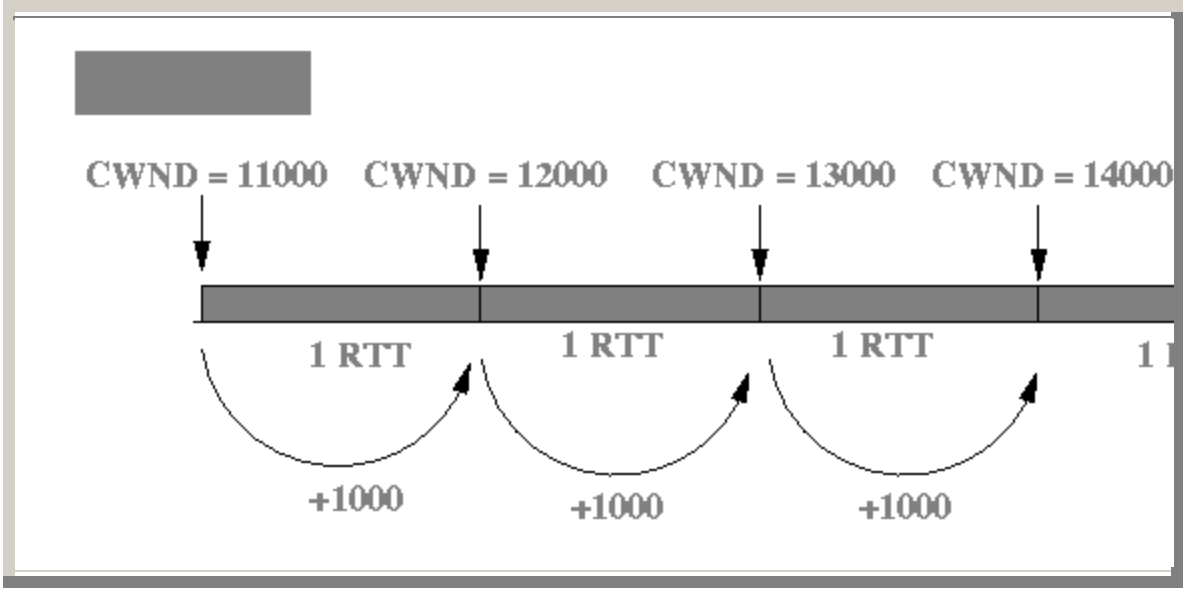
A new ACK message is an ACK that has a higher ACK number than the receive sequence number If TCP receives a *duplicate* ACK (i.e., an ACK message that has a ACK number that is less or equal to the receive sequence number) the *CWND* variable is unchanged.

The second mechanism added by TCP Tahoe is an improved method for round-trip-time estimation. The earliest versions of TCP used a smoothed round-trip-time estimator which was a Simple low-pass filters with a constant RTT variance factor of two. The problem with this Algorithm is that it adapts slowly to large changes in round-trip-times and this can cause the RTT to be underestimated. This in turn will cause the retransmission timer to expire even though the original packet has not been lost. The packet will then be sent a second time with the first packet still in the network. This wastes bandwidth and can lead to congestion because it violates the Conservation of packets requirement. The new RTT algorithm proposed by Jacobson uses a simple estimate of the round-trip-time variance rather than a constant beta factor of two. This allows it to more rapidly adjust to large timing changes and prevents erroneous timer expirations. Jacobson also showed that for proper stability an exponential backoff should be used when a retransmission timer expires and the packet is resent.

The third mechanism added by TCP Tahoe is Congestion-Avoidance. The purpose of this Algorithm is to ensure that a sender cuts its throughput in half when a loss occurs since the loss is assumed to be due to congestion. Although Congestion-Avoidance is a separate concept from Slow-Start, they are implemented as a single procedure in practice. When a loss occurs, a Threshold variable called ssthresh is set to half of the congestion window or receiver's window whichever is smaller. Then the congestion window is set to one to initiate Slow-Start. The connection stays in the Slow-Start phase and increases the congestion window by one packet for every ACK received until the congestion window reaches ssthresh. Then the connection enters the Congestion-Avoidance phase in which it increases the congestion window by one packet for each full window of data successfully transmitted and acknowledged This approach results in exponential decreases and linear increases.

TCP enters the congestion avoidance phase when the slow start phase terminates normally (i.e., CWND > SSThresHold). TCP increases CWND by one packet (= MSS bytes) when no packet drops have been detected for RTT sec (one epoch), A question arises when does TCP start using the Congestion Avoidance algorithm. Its answer is TCP starts to use Congestion Avoidance, when TCP has successfully completed the *Slow Start* phase I.e. when CWND ≥ SSThresh A second common question which arises is when TCP stops using the Congestion Avoidance algorithm, TCP stop to use Congestion Avoidance:
When TCP detects a packet loss event (TCP times out waiting for an ACK) A packet drop means congestion

Graphically:



It is difficult to constantly monitor packet drops for a given period of time but It is easier by far to increase CWND each time you receive a new ACK message Notice that if TCP is transmitting maximum size packets, and the congestion window is CWND, then there are approximately CWND/MSS packets sent using the transmit window

Practical implementation of the linear increase method:

We want to increase CWND by approximately MSS in RTT second

Fact:

- The are approximately CWND/MSS packets in transition towards the receiver
- If no packets are dropped, the sender will receive approximately CWND/MSS new acknowledgements
- Suppose we increase CWND by x each time we receive a new ACK
- Then after RTT sec (or CWND/MSS acks), the new value of CWND will be:

CWND + (CWND/MSS × x)

If we want CWND to increase by MSS in RTT sec (or CWND/MSS acks), then we must use:

CWND + (CWND/MSS * x) = CWND + MSS

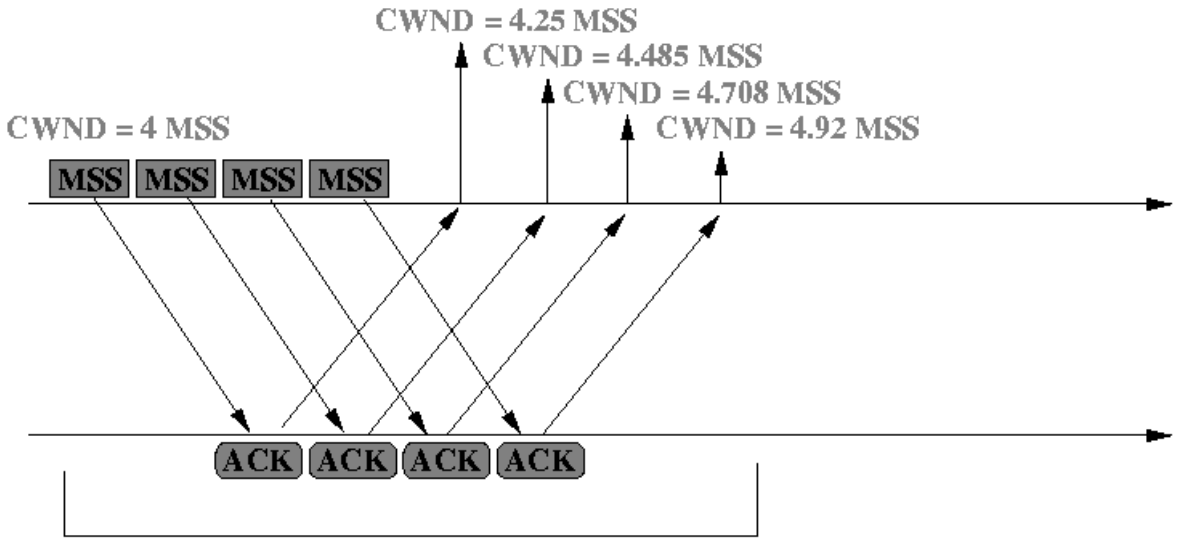
(CWND/MSS * x) = MSS

$$x = \frac{MSS * MSS}{CWND}$$

Conclusion:

If we want to increase CWND by MSS in RTT time, we can approximately achives this by increasing CWND by MSS^2/CWND each time TCP receives a new acknowledgement

Example of TCP operation in the congestion avoidance phase: using CWND = 4 × MSS



The congestion window size (CWND) increases APPROXIMATELY by 1 after approximately 1 RTT (Linear Increase)

Notes:

TCP has sent out 4 packets (each containing MSS bytes) to the receiver.
If there is no congestion (no packet drops), the sender will receive 4 new ACK packets in approximately RTT time

When the first ACK packet is received, TCP updates CWND as follows:

CWND = CWND + MSS * MSS/CWND // CWND = 4 MSS
= 4 MSS + MSS * MSS/(4 MSS)
= 4 MSS + MSS * 1/4

= 4.25 MSS

When the second ACK packet is received, TCP updates CWND as follows:

CWND = CWND + MSS * MSS/CWND // CWND = 4.25 MSS
= 4.25 MSS + MSS * MSS/(4.25 MSS)
= 4.25 MSS + MSS * 1/4.25

= 4.485 MSS

When the third ACK packet is received, TCP updates CWND as follows:

CWND = CWND + MSS * MSS/CWND // CWND = 4.485 MSS
= 4.485 MSS + MSS * MSS/(4.485 MSS)
= 4.485 MSS + MSS * 1/4.485

= 4.708 MSS

Finally, when the fourth (and final) ACK packet is received, TCP updates CWND as follows:

CWND = CWND + MSS * MSS/CWND // CWND = 4.708 MSS

= 4.708 MSS + MSS * MSS/(4.708 MSS)
= 4.708 MSS + MSS * 1/4.708

= 4.92 MSS

Conclusion:

The value of CWND is increased by 0.92 × MSS

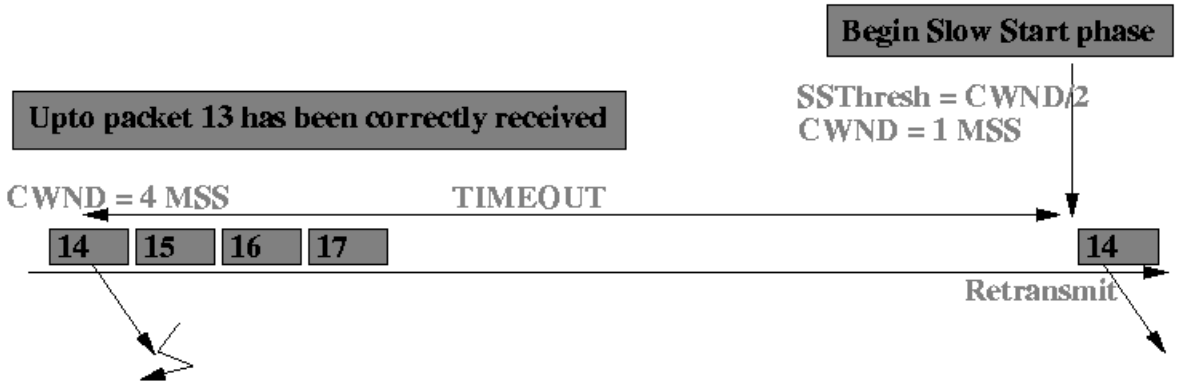
TCP start using the Congestion Avoidance algorithm When TCP has successfully completed the Slow Start phase I.e. when CWND ≥ SSThresh

TCP stop using the Congestion Avoidance algorithm When TCP detects a packet loss event because A packet drop means congestion

When TCP times out:

- Set: SSThresh = CWND/2
- Enter the slow start phase

Example:



Notes:

The sender has CWND = 4 × MSS

Suppose packet 14 is lost

The ACK message for packet 14 will not be received and the sender wills time out

Result of the time out: CP sets SSThresh = CWND/2 = 2 × MSS.

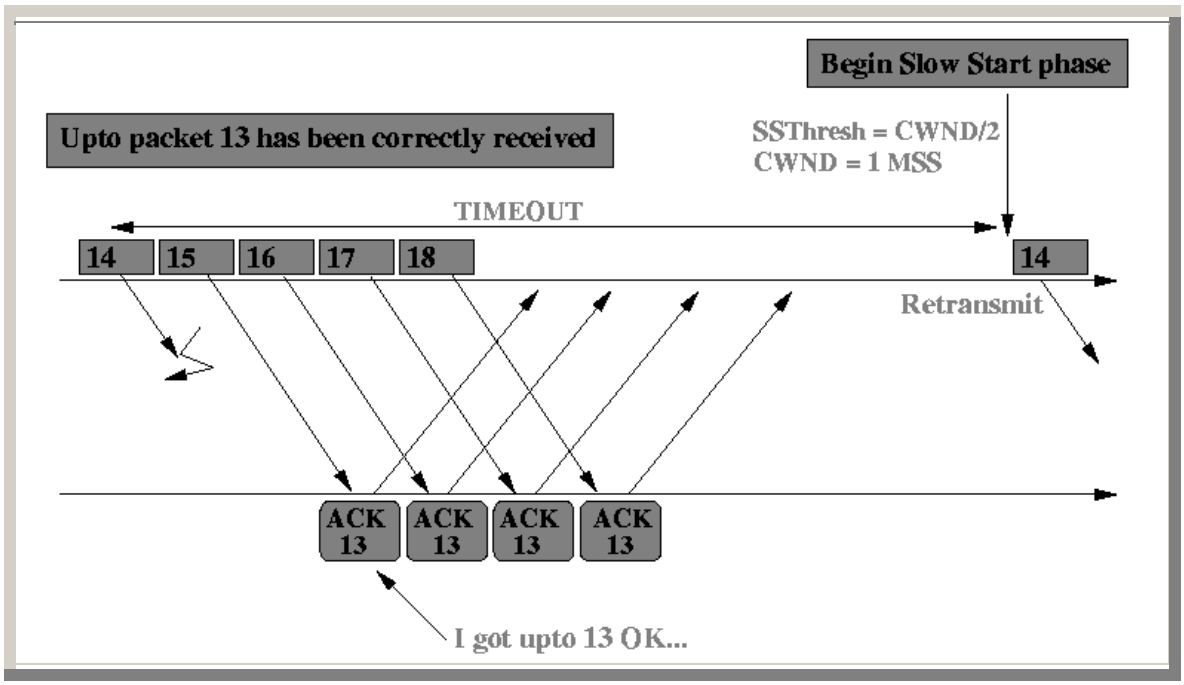
(This is the new "safe" operation level)...

Then TCP enters the slow start procedure:

- Set CWND = 1 x MSS (i.e., 1 packet worth of data) and
- increases CWND at an expontial rate towards SSThresh (the "safe" level")

Fast Retransmit: detecting packet drop using *duplicate ACKs*

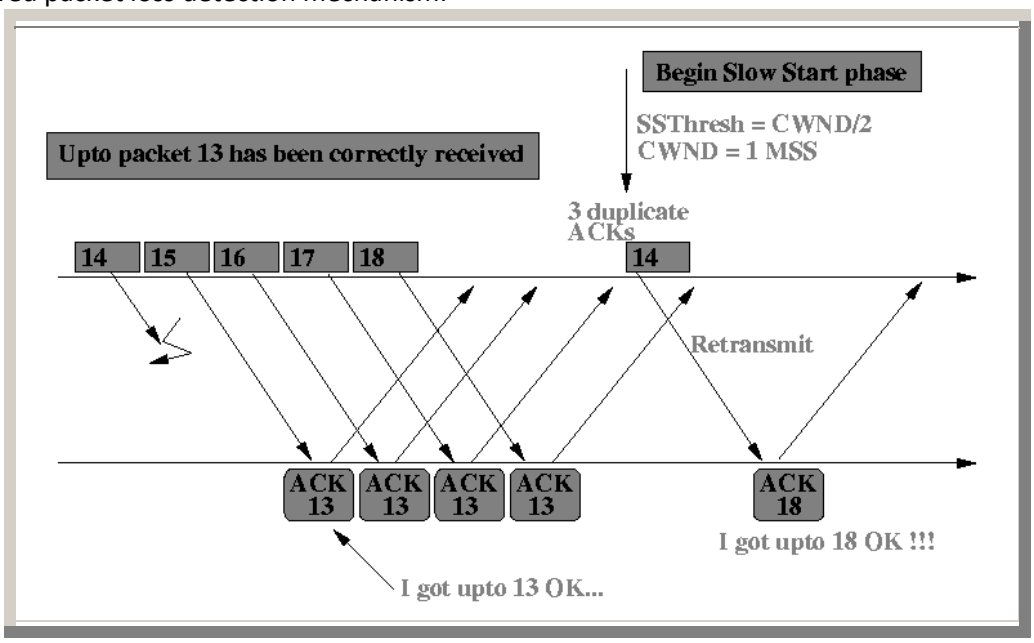
Notice that when a packet is dropped, the packets that *follow* the dropped packet will trigger duplicate ACK messages from the receiver:



We see 4 duplicate ACKs in the above example
Caveat: *not* every *duplicate* ACK is the result of packet loss
Avoid *misinterpretating* duplicate ACK as packet loss:
To eliminate *false* loss indications from duplicate ACK, TCP will conclude packet loss *only* after receive 3 duplicate ACKs

(So TCP received a total of 4 identical ACK packets),

Improved packet loss detection mechanism:



After receiving 3 duplicate ACKs, TCP concludes that a packet has been loosed

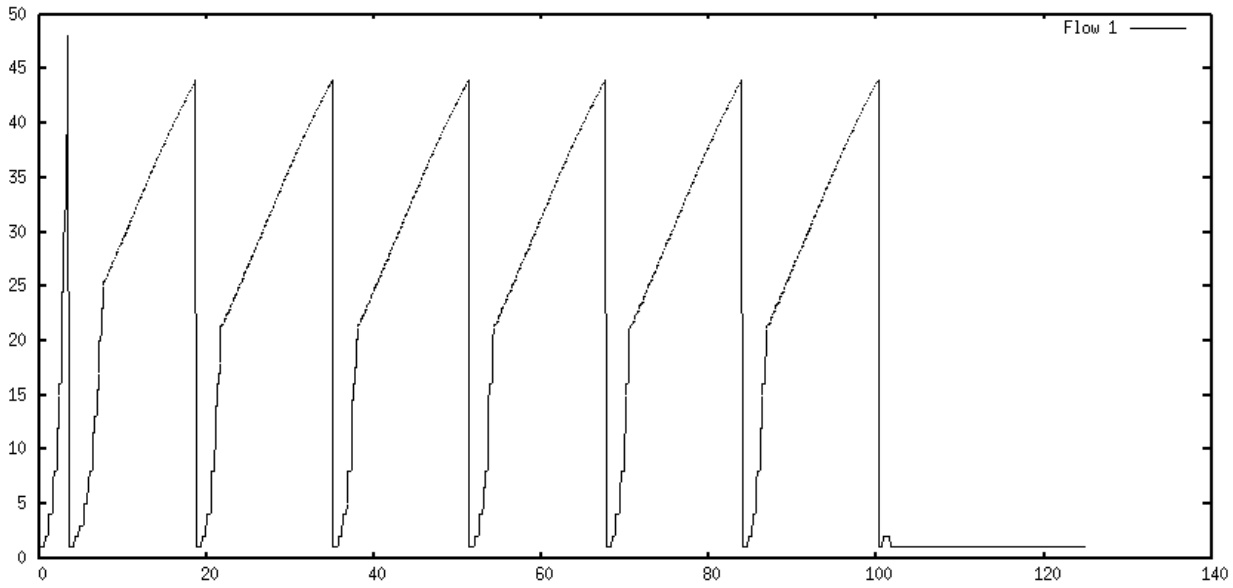
TCP retransmits the loosed packet *immediately* without waiting for timeout !
This retransmission technique is called fast retransmit

(It retransmits a loss packet faster than using time out)

Furthermore, TCP will enter the *Slow Start* phase (because a packet loss had occurred)
TCP Tahoe

- The TCP algorithm that uses:
- Slow start
- Congestion avoidance
- Fast Retransmit

is known as TCP Tahoe



we can see the operation of TCP Tahoe clearly from the above figure:

- At approximately time 0, TCP Tahoe starts and it is in the slow start mode, the congestion window size increases exponentially
- At approximately time 5, packet loss is detected.
  TCP marks SSThresh = 25 (approximately) and begins another slow start
- When it reaches CWND = 25 (approximately), the CWND increases linearly - here TCP Tahoe enters the congestion avoidance mode
- At approximately time 19, TCP Tahoe detects packet loss and begins a slow start.

  SSThresHold is approximately 22.

- TCP begins another slow start and so on.

The final mechanism introduced in TCP Tahoe was Fast-Retransmission. If a packet arrives out-of-order (i.e. before the packet ahead of it in the sender's transmission sequence) it means that either it took a faster path or the packet before it was lost. The Fast-Retransmission algorithm requires the receiver to immediately send an ACK when an out-of-order packet is received.

Advantages of Tahoe:

Tahoe has the most basic congestion and loss mechanisms of all the modern TCP Implementations. However, the Fast-Retransmit algorithm introduced with TCP Tahoe may provide the single most significant performance improvement when packets are lost due to congestion. Fast-Retransmit is often able to detect packet losses in a matter of several milliseconds on a LAN and within roughly a hundred milliseconds on a WAN. Without Fast- Retransmit, the sender is forced to wait for a retransmission timer to expire. Since these timers are designed to provide a relatively loose upper bound, they often have values on the order of several seconds for even the fastest of LANs. Therefore, Fast-Retransmit can save on the order of several seconds each time a loss occurs and this translates into enormous gains in throughput.

Disadvantages of Tahoe:

While Fast-Retransmit makes Tahoe perform drastically better than a TCP implementation whose sole means of loss detection is retransmission timers, it obtains significantly less than optimal performance on high delay-bandwidth connections because of its initiation of Slow-Start (which TCP Reno avoids). Also, in the case of multiple losses within a single window, it is possible that the sender will retransmit packets which have already been received.

REFERENCE:

John Kristoff's Overview of TCP (Fundamental concepts behind TCP and how it is used to transport data between two endpoints)
http://www.apps.ietf.org/rfc
http://en.wikipedia.org/wiki/Transmission_Control_Protocol
http://www.apps.ietf.org/rfc/rfc2581.html
http://www.apps.ietf.org/rfc/rfc2001.html
http://en.wikipedia.org/wiki/Nagle%27s_algorithm
http://www.mathcs.emory.edu/~cheung/Courses/455/Syllabus/A1-congestion/tcp2.html
V. Jackobson and M. J. Karels, “Congestion Avoidance and Control”, Proceedings of SIGCOMM ’88, August 1988.
H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, “A Comparison of Mechanisms for Improving TCP Performance over Wireless Links”, Proceedings of ACM SIGCOMM ’96, August 1996.