Principles of TCP congestion control

There are a huge number of issues related to TCP congestion control. In this course we focus only on the most basic aspects of congestion control.

First, notice that *Congestion Control* and *Flow Control* are different aspects of TCP data transfer. Flow control refers to limiting data sent, so as not to overwhelm the capacity of the receiving host. Congestion control, on the other hand, is related to changes in the intervening network parameters such as ambient data traffic, router capacities and physical link properties.

Secondly, congestion control is not part of core TCP specifications. Accordingly, TCP implementations handle congestion in different ways. TCPs are not even required to have congestion control algorithms implemented.

What is congestion?

When the transmission rate out of a router is lower than the incoming data rate, bits get queued up in the router buffers. When the buffer get full, all further incoming packets are dropped by the router TCP - *in this case no error messages are sent to the originator of the datagram*. The sender will have to realize this drop event, and retransmit the dropped packet. This retransmission should be at a rate that does not further overwhelm the congested routers.

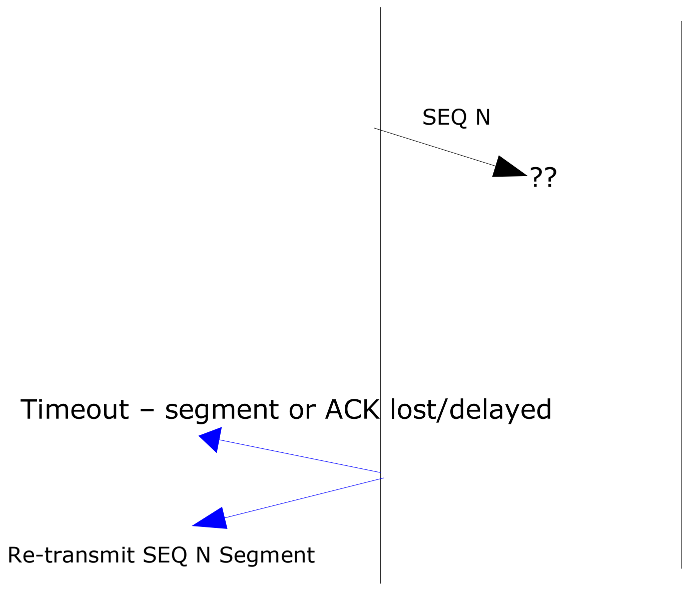
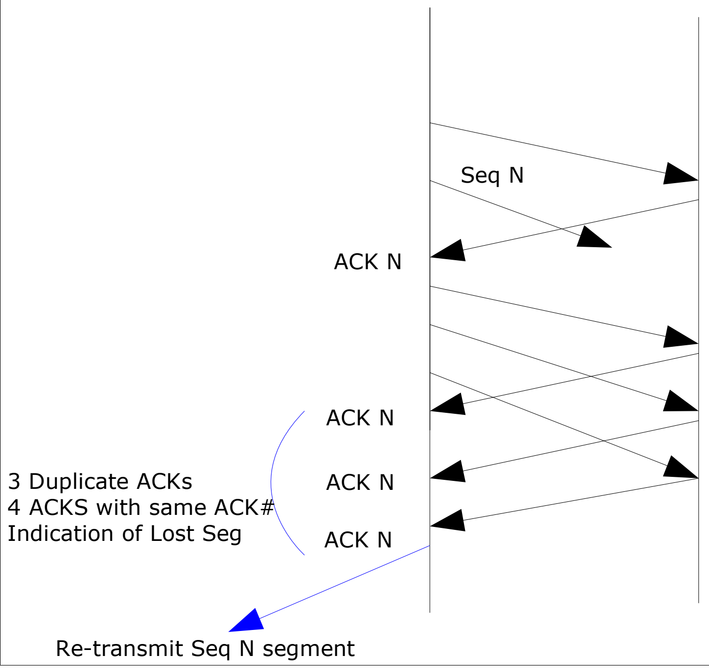
There are many complicating factors in congestion-control. For example, if a packet is dropped due to a checksum error, how can it not be confused with a drop due to congestion? Let us set these aside for the time being.

So the principal issues are

- How to detect congestion
- How to respond to congestion

There are network-assisted congestion detection mechanisms and end-to-end congestion detection mechanisms. TCP uses end-to-end congestion detection. This means that TCP senses congestion based on how segments and ACKs received at the end points of the link.

There are no fool-proof way to detect congestion without help from the network. So TCP resorts to ad-hoc methods which works most of the time.

Scenario - 1  SEQ N ?? Timeout – segment or ACK lost/delayed Re-transmit SEQ N Segment	A time-out event where an ACK for a segment was not received might be a sign of congestion – in this case the segment is re-transmitted immediately.
Scenario – 2  Seq N ACK N ACK N ACK N ACK N 3 Duplicate ACKs 4 ACKS with same ACK# Indication of Lost Seg Re-transmit Seq N segment	Receipt of three or more duplicate ACKs (same ACK numbers) is generally taken as an indication that a packet was lost, even though some packets are getting through. Again, this is considered a signal of congestion.

The main TCP congestion control mechanism is based on a parameter called *Congestion Window*, **cwind**. This parameter determines the *rate* at which TCP sends *data out.* The amount of unacknowledged data in the network, (LastByteSent-LastByteAcked), is supposed to be lass than the *cwind* and the RecvWindow (receive window at the receiver).

$(\text{LastByteSent-LastByteAcked}) \leq \min\{cwind, \text{RecvWindow}\}$

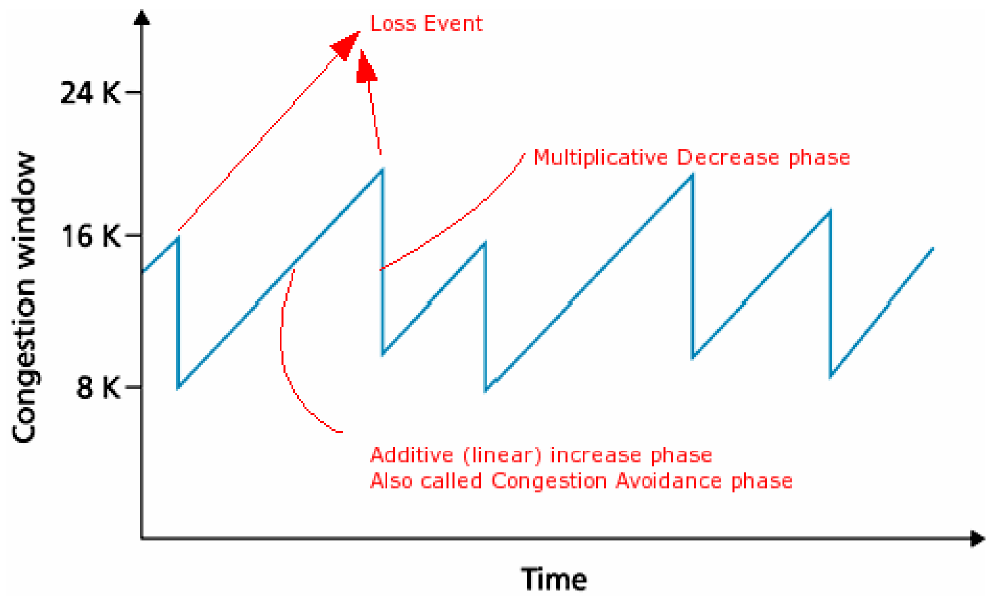
In practice the receive windows are quite large, so that in effect, we can say that

$(\text{LastByteSent-LastByteAcked}) \leq \text{cwind}$

In the beginning of a send cycle, a sender can send up to 1 cwind bytes of data, which is acknowledged in 1 RTT (Round Trip Time). Thus the average send rate is $\frac{cwind}{RTT}$ bytes/sec. Thus reducing or enlarging *cwind* generally decreases or increases the sending rate at the sender. The basic idea behind congestion control is to reduce the sending rate, by reducing the *cwind* value, when a loss event occurs.

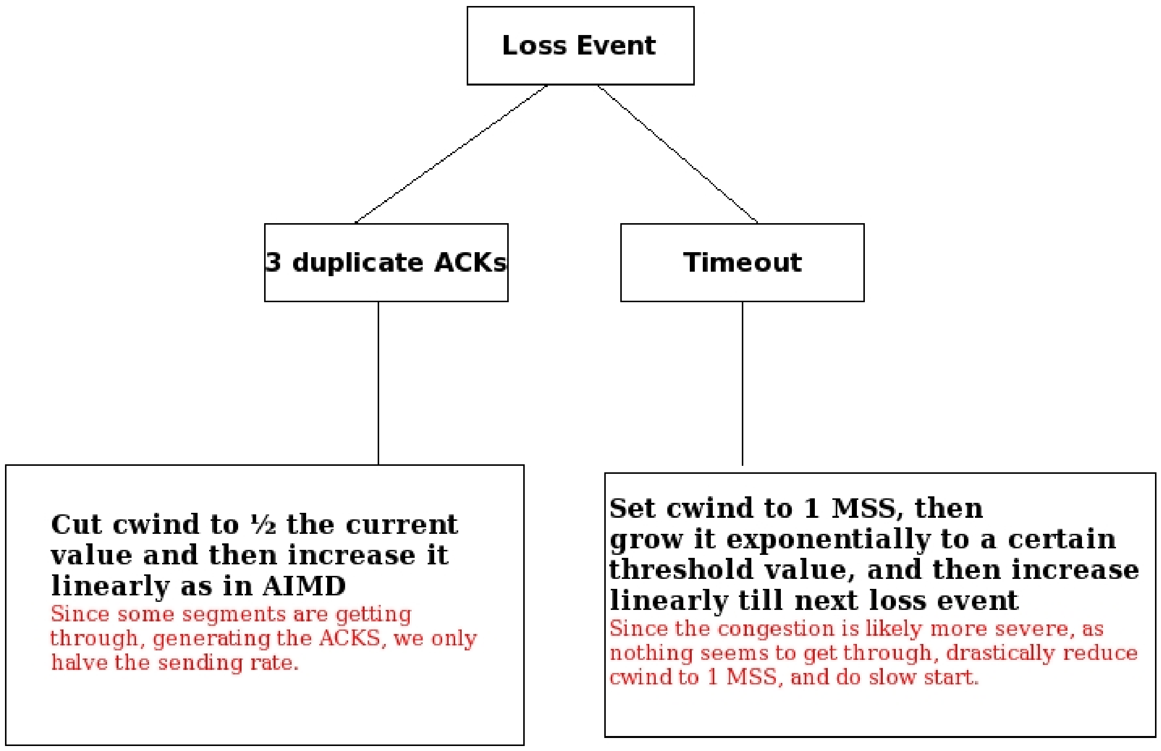
There are a few well-known algorithms generally used for handling congestion. We will look at each of these in turn. Many modifications of these mechanisms have been proposed, but we consider the most-often implemented ones.

AIMD(Additive Increase Multiplicative Decrease). TCP starts by setting its cwind to conservative values (such as 1 MSS), and if it perceives no congestion, linearly increases the cwind value, thereby increasing the send rate. When a loss event is perceived, via timeout or multiple duplicate ACKs), the cwind is cut into half the current value, repeating the cycle again.



TCP Slow Start. In the above scheme(AIMD), during the linear increase phase of the *cwind,* there may be higher bandwidth available which is not being used. An alternative mechanism proposes starting with very low sending rate (slow start), then rapid (exponential) increasing in the sending rate till congestion is detected. Specifically, TCP starts by setting cwind to 1 and sending 1 segment of size 1 MSS into the network. When an ACK of this is received, the cwind is increased by 1 MSS (to a value of 2 MSS) and sends 2 MSS bytes. If these two are ACKed before a loss event, the cwind is increased by 2 MSS (to a value of 4 MSS) and four segments are sent. Thus the sending rate is rapidly (exponentially) increased till a loss occurs.

When a loss event occurs, TCPs can react in many different ways. One scheme is as follows:


Loss Event
3 duplicate ACKs
Timeout
Cut cwind to ½ the current value and then increase it linearly as in AIMD
Since some segments are getting through, generating the ACKS, we only halve the sending rate.
Set cwind to 1 MSS, then grow it exponentially to a certain threshold value, and then increase linearly till next loss event
Since the congestion is likely more severe, as nothing seems to get through, drastically reduce cwind to 1 MSS, and do slow start.

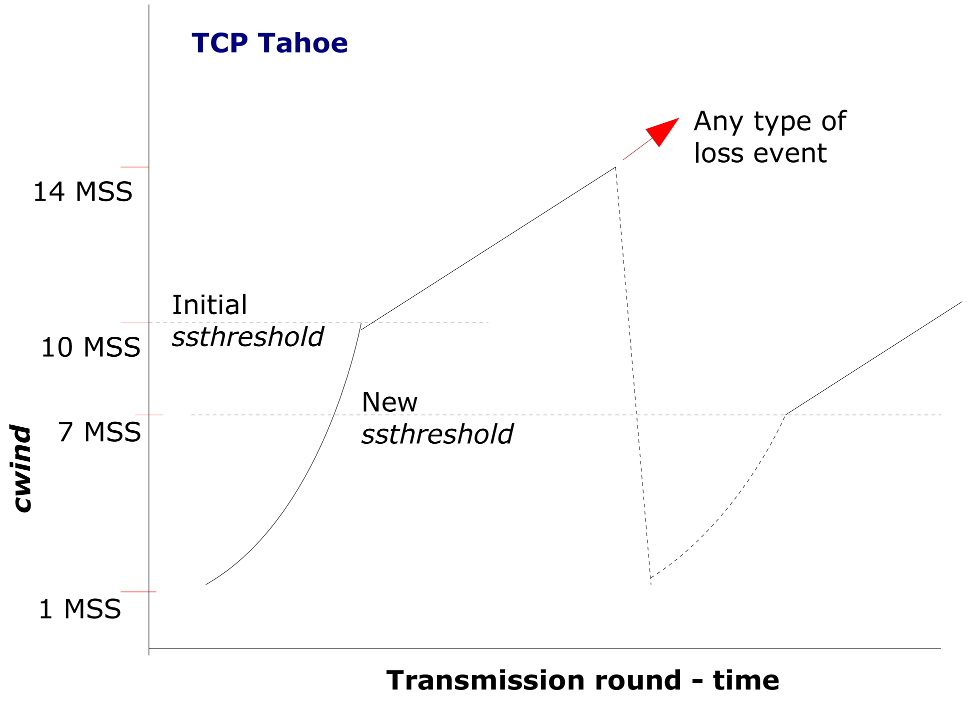
Implementation.
The implementation of this algorithm is done with another parameter, called Slow Start Threshold, *ssthreshold.* The idea is that when *cwind* is below *ssthreshold,* sender is in slow-start phase, and the *cwind* can grow exponentially. When the *cwind* is above

ssthreshold, sender is in congestion-avoidance, and *cwind* grows linearly.

Initially *ssthreshold* is set to a reasonably high value. When a loss event due to three duplicate ACKs occurs, *ssthreshold* is set to ½ the current *cwind* value, *cwind* is then set to *ssthreshold* and TCP goes into congestion-avoidance (linear increase of send rate) phase. When a loss event due to timeout occurs *ssthreshold* is set to ½ the current *cwind* value, *cwind* is then set to 1 MSS

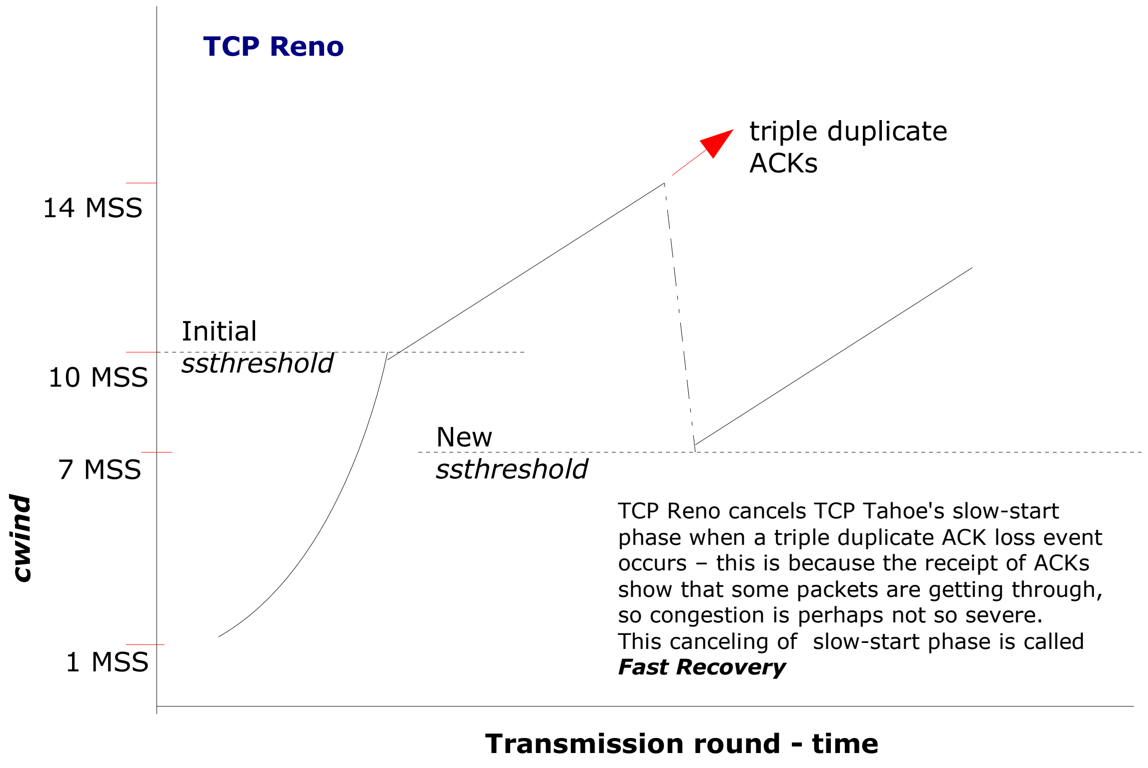
Evolution of Congestion Control algorithms for TCP

The earliest Congestion algorithm implemented in 4.3 BSD[1], called **TCP Tahoe**, incorporates most of the mechanisms discussed above. In particular TCP Tahoe cuts the *cwind* to 1 MSS when any type of congestion is sensed, followed by slow start, till cwind reaches *ssthreshold*, and then does congestion-avoidance till next loss event. The figure below graphically shows the behavior of TCP Tahoe.



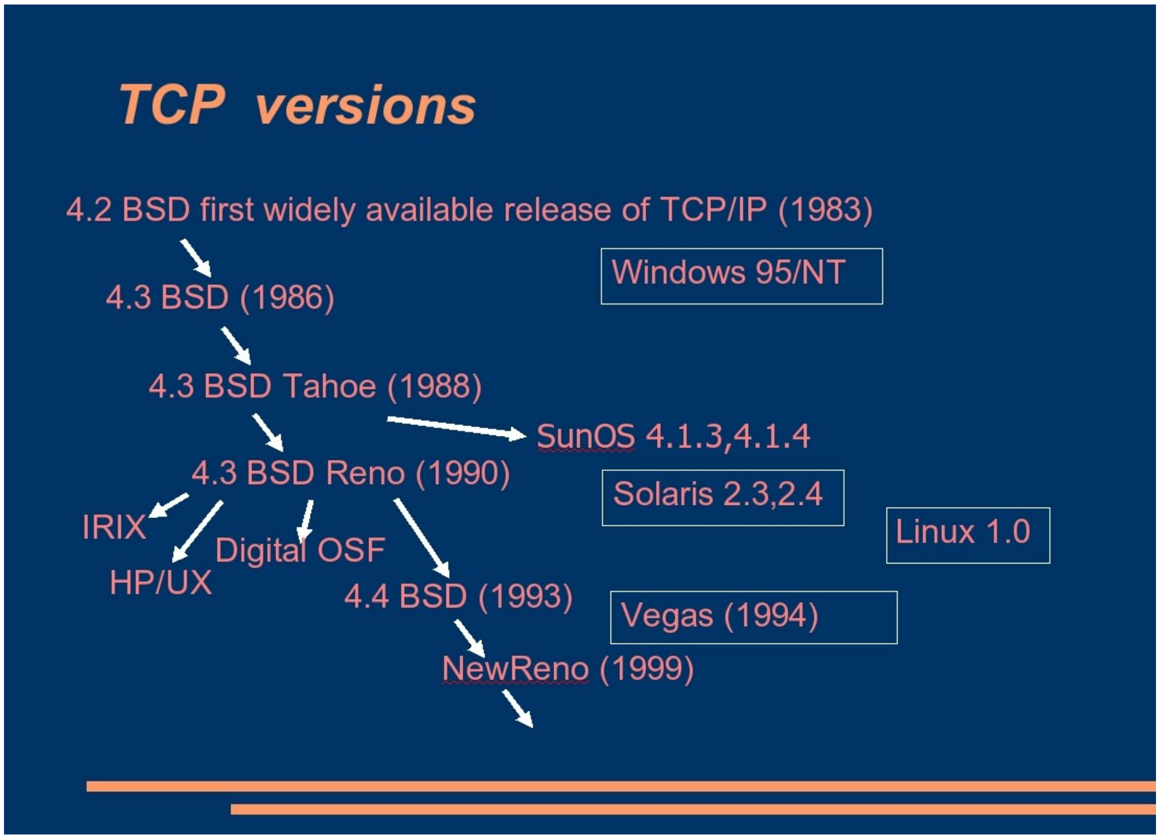
1 See http://www.oreilly.com/catalog/opensources/book/kirkmck.html for a history of Berkeley UNIX and BSD. Also see http://www.levenez.com/unix/history.html for a more complete UNIX history.

4.3 BSD, 1990, called **TCP Reno**, implements a *Fast re-transmit/Fast recovery* algorithm described in RFC2581. If three duplicate ACKs are received (that means 4 consecutive identical ACK #s received), signaling a lost packet but probably less severe network congestion, TCP retransmits the packet even before retransmission timeout is reached. This is called *fast re-transmit*. Then, instead of setting *cwind* to 1 MSS as in TCP Tahoe, it is set to ½ the *cwind* value at which loss event occurred, and congestion avoidance takes over.



A variation on TCP Reno, called NewReno is described in RFC2582.

The next figure summarizes various TCP flavors.



TCP versions