Computer Networks (CS425)

Instructor: Dr. Dheeraj Sanghi

Prev | Next | Index

Topics in TCP

TCP Congestion Control

If the receiver advertises a large window-size , larger than what the network en route can handle , then there will invariably be packet losses. So there will be re-transmissions as well . However , the sender cannot send all the packets for which ACK has not been received because this way it will be causing even more congestion in the network. Moreover , the sender at this point of time cannot be sure about how many packets have actually been lost . It might be that this is the only one that has been lost , and some following it have actually been received and buffered by the receiver. In that case , the sender will have unnecessarily sent a number of packets.

So the re-transmission of the packets also follows slow-start mechanism. However , we do indeed need to keep an upper bound on the size of the packets as it increases in slow start, to prevent it from increasing unbounded and causing congestion. This cap is put at half the value of the segment size at which packet loss started.

Congestion Window

We have already seen one bound on the size of the segments sent by the receiver-namely , the receiver window that the receiver advertises . However there could be a bottleneck created by some intermediate network that is getting clogged up. The net effect is that just having the receiver window is not enough. There should be some bound relating to the congestion of the network path - congestion window captures exactly this bound. Similar to receiver window, we have another window , the Congestion Window , and the maximum size of the segments sent are bounded by the minimum of the sizes of the two windows. E.g. If the receiver says "send 8K" (size of the receiver window) , but the sender knows that bursts of more than 4K (size of congestion window) clog the network up, then it sends 4K. On the other hand , if the congestion window was of size 32K , then the sender would send segments of maximum size 8K.

How do we calculate/manage the Congestion Window ?

The size of the congestion window is initialized to 1.For every ACK received , it is incremented by 1. Another field that we maintain is threshold which is

equal to half the size of the congestion window. Whenever a packet loss takes place, the congestion window is set to 1.Then we keep increasing the congestion window by 1 on every ACK received till we reach the threshold value. Thereafter, we increment the congestion window size by 1 after every round trip time.

Notice that TCP always tries to keep the flow rate slightly below the maximum value. So if the network traffic fluctuates slightly, then a lot of packets might be lost. Packet losses cause a terrible loss in throughput.

In all these schemes, we have been assuming that any packet loss occurs only due to network congestion. What happens if some packet loss occurs not due to some congestion but due to some random factors?

When a packet is lost, the congestion window size is set to 1. Then when we retransmit the packet, if we receive a cumulative ACK for a lot of subsequent packets, we can assume that the packet loss was not due to congestion, but because of some random factors. So we give up slow start and straightaway set the size of Congestion Window to the threshold value.

Silly Window Syndrome

This happens when the application supplying data to the sender does do in large chunks, but the application taking data from receiver (probably an interactive application) does it in very small chunks, say 1 byte at a time. The sender keeps advertising windows of size 1 byte each as the application consumes the bytes one at a time.

Clark's Solution to this problem

We try to prevent the sender from advertising very small windows. The sender should try to wait until it has accumulated enough space in the window to send a full segment or half the receiver's buffer size, which it can estimate from the pattern of window updates that it received in the past.

Another problem: What if the same behavior is shown by an interactive application at the sender's end ? That is , what if the sender keeps sending in segments of very small size?

Nagle's algorithm

when data comes to the sender one byte at a time , send the first byte and buffer all the remaining bytes till the outstanding byte is acknowledged. Then send all the buffered characters in one segment and start buffering again till they are acknowledged. It can help reduce the bandwidth usage for example when the user is typing quickly into a telnet connection and the network is slow .

Persistent Timer

Consider the following deadlock situation . The receiver sends an ACK with 0 sized window, telling the sender to wait. Later it send an ACK with non-zero window, but this ACK packet gets lost. Then both the receiver and the sender will be waiting for each other to do something. So we keep another timer. When this timer goes off, the sender transmits a probe packet to the sender with an ACK number that is old. The receiver responds with an ACK with updated window size and transmission resumes.

Now we look at the solution of the last two problems ,namely **Problem of Random Losses** and **Sequence Number Wrap Around**.

Problem of Random Losses

How do we know if a loss is a congestion related loss or random loss ?If our window size is very large then we cannot say that one packet loss is random loss.So we need to have some mechanism to find what packets are lost. Cumulative Acknowledgement is not a good idea for this.

Solutions

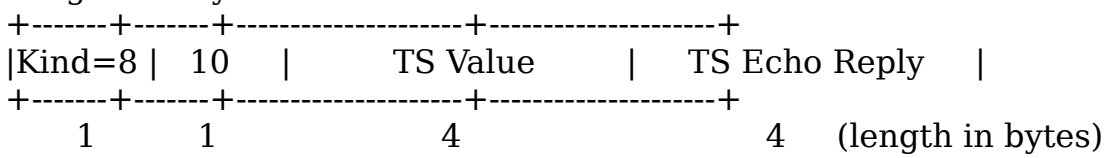
Selective Acknowledgement

We need a selective acknowledgement but that creates a problem in TCP because we use byte sequence numbers .So what we we do is that we send the sequence number and the length. We may have to send a large number of such Selective Acknowledgements which will increase the overhead So whenever we get out of sequence packets we send the information a few time not in all the packets anyway. So we cannot rely on Selective Acknowledgement anyway. If we have 32 bit sequence number and 32 bit length,then already we will have too much of overhead .One proposal is to use 16 bit length field. If we have very small gaps then we will think that random losses are there and we need to fill them .If large gaps are there we assume that congestion is there and we need to slow down.

TCP Timestamps Option

TCP is a symmetric protocol, allowing data to be sent at any time in either direction, and therefore timestamp echoing may occur in either direction. For simplicity and symmetry, we specify that timestamps always be sent and echoed in both directions. For efficiency, we combine the timestamp and timestamp reply fields into a single TCP Timestamps Option.

Kind: 8
Length: 10 bytes



The Timestamps option carries two four-byte timestamp fields. The Timestamp Value field (TSval) contains the current value of the timestamp clock of the TCP sending the option. The Timestamp Echo Reply field (TSecr) is only valid if the **ACK** bit is set in the TCP header; if it is valid, it echos a times- tamp value that was sent by the remote **TCP** in the TSval field of a Timestamps option. When TSecr is not valid, its value must be zero. The TSecr value will generally be the time stamp for the last in-sequence packet received.

Example:
Sequence of packet send : 1 (t1) 2 (t2) 3 (t3) 4 (t4) 5 (t5) 6 (t6)
sequence of packets received: 1 2 4 3 5 6
time stamp copied in **ACK:** t1 t2 t3

PAWS: Protect Against Wrapped Sequence Numbers

PAWS operates within a single TCP connection, using state that is saved in the connection control block. PAWS uses the same TCP Timestamps option as the RTTM mechanism described earlier, and assumes that every received TCP segment (including data and ACK segments) contains a timestamp **SEG.TSval** whose values are monotone non-decreasing in time. The basic idea is that a segment can be discarded as an old duplicate if it is received with a timestamp **SEG.TSval** less than some timestamp recently received on this connection.

In both the PAWS and the RTTM mechanism, the "timestamps" are 32- bit unsigned integers in a modular 32-bit space. Thus, "less than" is defined the same way it is for TCP sequence numbers, and the same implementation techniques apply. If s and t are timestamp values, s < t if 0 < (t - s) < 2**31, computed in unsigned 32-bit arithmetic.

The choice of incoming timestamps to be saved for this comparison must guarantee a value that is monotone increasing. For example, we might save the timestamp from the segment that last advanced the left edge of the receive window, i.e., the most recent in- sequence segment. Instead, we choose the value **TS.Recent** for the RTTM mechanism, since using a common value for both PAWS and RTTM simplifies the implementation of both. **TS.Recent** differs from the timestamp from the last in-sequence segment only in the case of delayed ACKs, and therefore by less than one window. Either choice will therefore protect against sequence number wrap-around.

RTTM was specified in a symmetrical manner, so that TSval timestamps are carried in both data and ACK segments and are echoed in TSecr fields carried in returning ACK or data segments. PAWS submits all incoming segments to the same test, and therefore protects against duplicate ACK segments as well as data segments. (An alternative un-symmetric algorithm would protect against old duplicate ACKs: the sender of data would reject

incoming ACK segments whose TSecr values were less than the TSecr saved from the last segment whose ACK field advanced the left edge of the send window. This algorithm was deemed to lack economy of mechanism and symmetry.)

TSval timestamps sent on {SYN} and {SYN,ACK} segments are used to initialize PAWS. PAWS protects against old duplicate non-SYN segments, and duplicate SYN segments received while there is a synchronized connection. Duplicate {SYN} and {SYN,ACK} segments received when there is no connection will be discarded by the normal 3-way handshake and sequence number checks of TCP.

Header Prediction

As we want to know that from which TCP connection this packet belongs. So for each new packet we have to match the header of each packet to the database that will take a lot of time so what we do is we first compare this header with the header of last received packet and on an average this will reduce the work. Assuming that this packet is from the same TCP connection from where we have got the last one (locality principal).

---

back to top
Prev| Next | Index

acing="0" style="border-collapse: collapse" bordercolor="#111111" width="100%" id="AutoNumber11">