This sequence diagram was generated with EventStudio System Designer (http://www.EventHelix.com/EventStudio).

TCP Slow Start and Congestion Avoidance lower the data throughput drastically when segment loss is detected. Fast Retransmit and Fast Recovery have been designed to speed up the recovery of the connection, without compromising its congestion avoidance characteristics.

Fast Retransmit and Recovery detect a segment loss via duplicate acknowledgements. When a segment is lost, TCP at the receiver will keep sending ack segments indicating the next expected sequence number. This sequence number would correspond to the lost segment. If only one segment is lost, TCP will keep generating acks for the following segments. This will result in the transmitter getting duplicate acks (i.e. acks with the same ack sequence number)

Socket initialization

Server awaits client socket connections.

Client socket initialization

create

Client Socket

Client Application creates Socket

Closed

The socket is created in the Closed state

seq_num = 0

Initial sequence number is set to 0

Active_Open

Application wishes to communicate with a destination server using a TCP connection. The application opens a socket for the connection in active mode. In this mode, a TCP connection will be attempted with the server.
Typically, the client will use a well known port number to communicate with the remote Server. For example, HTTP uses port 80.

Client initiated three way handshake to establish a TCP connection

SYN

src = Client_Port,
dst = Server_Port,
seq_num = 0

Client sets the SYN bit in the TCP header to request a TCP connection. The sequence number field is set to 0. Since the SYN bit is set, this sequence number is used as the initial sequence number

SYN Sent

Socket transitions to the SYN Sent state

SYN+ACK

src = Server_Port,
dst = Client_Port,
seq_num = 100,
ack_num = 1,
window = 65535

Client receives the "SYN+ACK" TCP segment

ACK

src = Client_Port,
dst = Server_Port,
ack_num = 101,
window = 5000

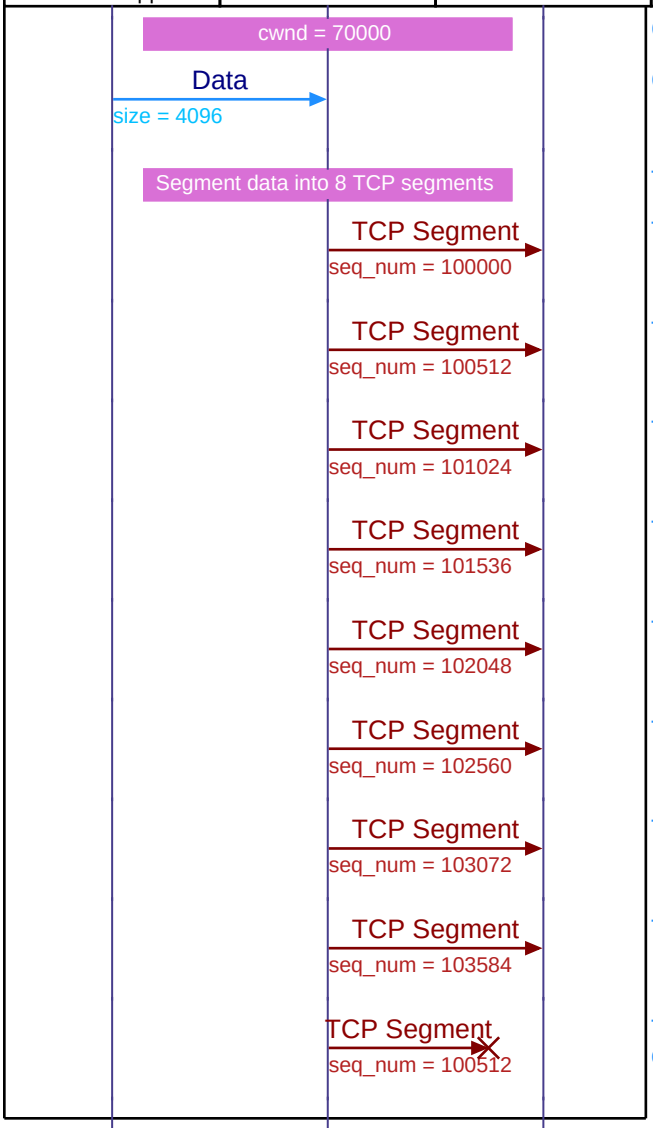
Client now acknowledges the first segment, thus completing the three way handshake. The receive window is set to 5000. Ack sequence number is set to 101, this means that the next expected sequence number is 101.

Established

At this point, the client assumes that the TCP connection has been established

TCP Connection begins with slow start. The congestion window grows from an initial 512 bytes to 70000 bytes

Loss of a TCP segment



Congestion window has reached 70000 bytes

Client App transmits 4Kbytes of data

TCP segments data to 8 TCP segments (each segment is 512 bytes)

TCP segment (start sequence number = 100000) is transmitted

TCP segment (start sequence number = 100512) is transmitted

TCP segment (start sequence number = 101024) is transmitted

TCP segment (start sequence number = 101536) is transmitted

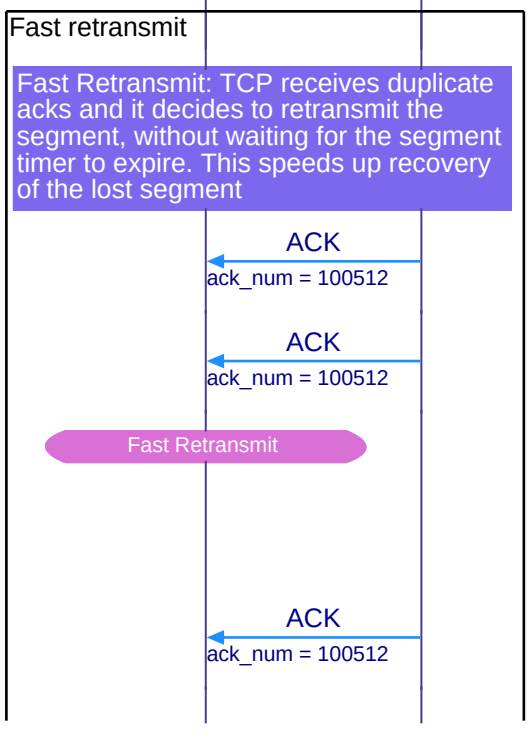
TCP segment (start sequence number = 102048) is transmitted

TCP segment (start sequence number = 102560) is transmitted

TCP segment (start sequence number = 103072) is transmitted

TCP segment (start sequence number = 103584) is transmitted

TCP segment (start sequence number = 100512) is lost due to congestion in the network.

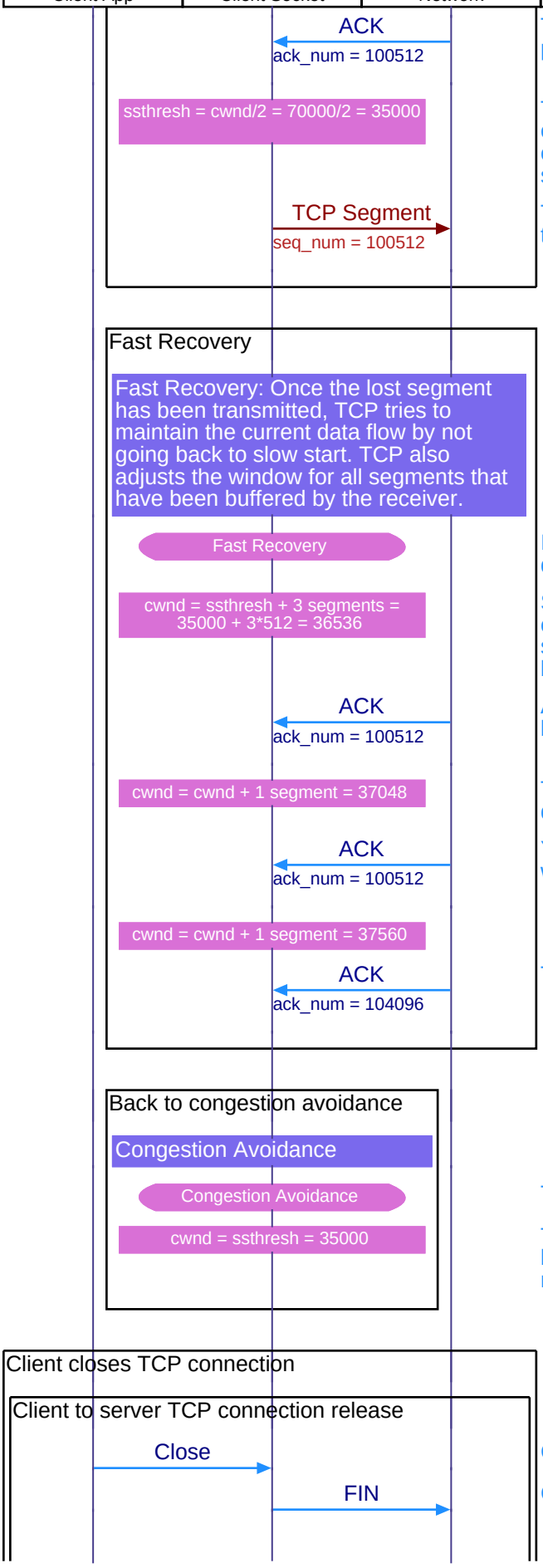


Client receives acknowledgement to the segment with starting sequence number 100512

First duplicate ack is received. TCP does not know if this ack has been duplicated due to out of sequence delivery of segments or the duplicate ack is caused by lost segment.

At this point TCP moves to the fast retransmit state. TCP will look for duplicate acks to decide if a segment needs to be retransmitted
Note: TCP segments sent by the sender can be delivered out of sequence to the receiver. This can also result in duplicate acks. Thus TCP waits for 3 duplicate acks before concluding that a segment has been missed.

Second duplicate ack is received



Third duplicate ack is received. TCP now assumes that duplicate acks point to a segment that has been lost

TCP uses the current congestion window to mark the point of congestion. It saves the slow start threshold as half of the current congestion window size. If current cwnd is less than 4 segments, cwnd is set to 2 segments

TCP retransmits the missing segment i.e. the segment corresponding to the ack sequence number in the duplicate acks

In "Fast Recovery" state, TCP's main objective is to maintain the current data stream data flow.

Since TCP started taking action on the third duplicate ack, it sets the congestion window to ssthresh + 3 segment. This halfs the TCP window size and compensates for the TCP segments that have already been buffered by the receiver.

Another duplicate ack is received. This means that the receiver has buffered one more segment

TCP again inflates the congestion window to compensate for the delivered segment

Yet another ack is received, this will further inflate the congestion window

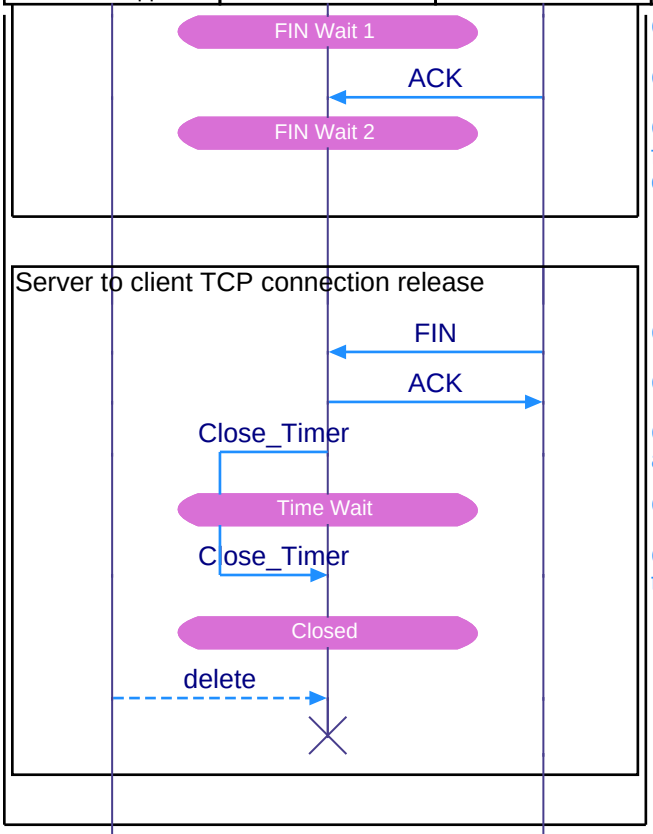
The cummulative TCP ack is delivered to the client

The connection has moved back to the congestion avoidance state.

TCP takes a congestion avoidance action and sets the segment size back to the slow start threshold. The TCP window will now increase by a maximum of one segment per round trip

Client application wishes to release the TCP connection

Client sends a TCP segment with the FIN bit set in the TCP header



Client changes state to FIN Wait 1 state

Client receives the ACK

Client changes state to FIN Wait 2. In this state, the TCP connection from the client to server is closed. Client now waits close of TCP connection from the server end

Client receives FIN

Client sends ACK

Client starts a timer to handle scenarios where the last ack has been lost and server resends FIN

Client waits in Time Wait state to handle a FIN retry

Close timer has expired. Thus the client end connection can be closed too.