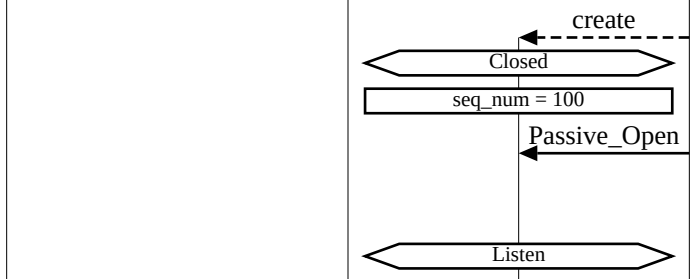
LEG: About Fast Retransmit and Fast Recovery

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)



Server Application creates a Socket

The Socket is created in Closed state

Server sets the initial sequence number to 100

Server application has initiated a passive open. In this mode, the socket does not attempt to establish a TCP connection. The socket listens for TCP connection request from clients

Socket transitions to the Listen state

Client Application creates Socket

The socket is created in the Closed state

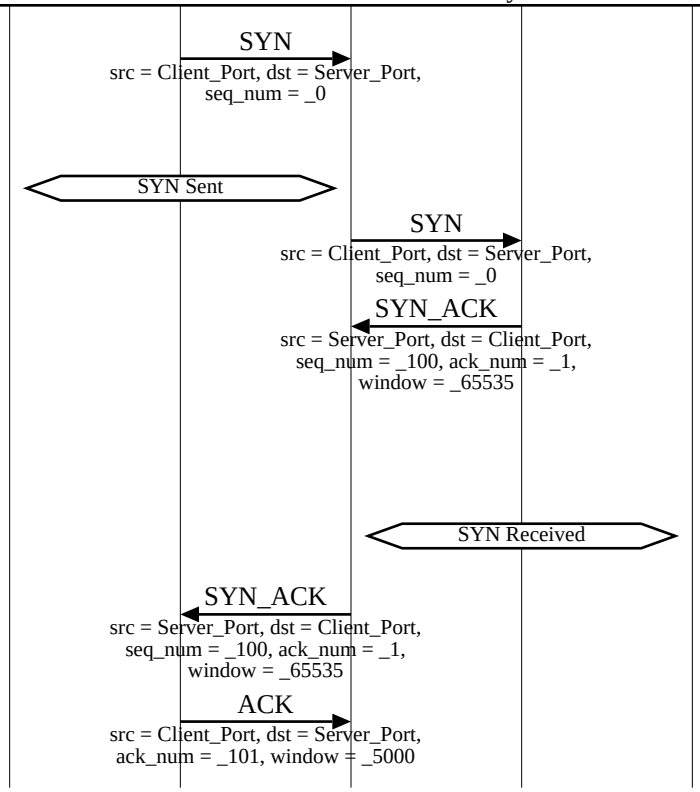
Initial sequence number is set to 0

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.

LEG: Client initiates TCP connection

Client initiated three way handshake to establish a TCP connection



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

Socket transitions to the SYN Sent state

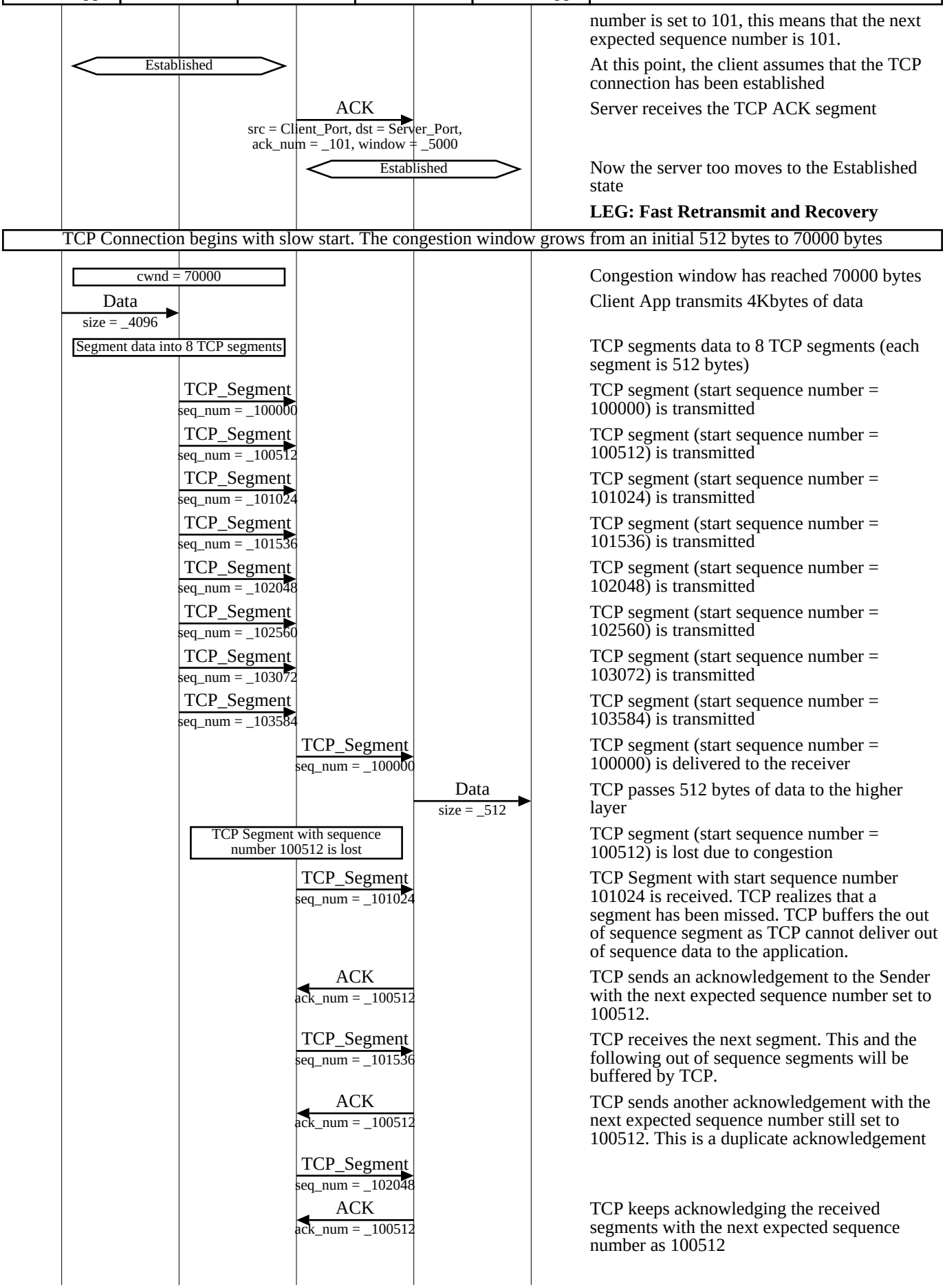
SYN TCP segment is received by the server

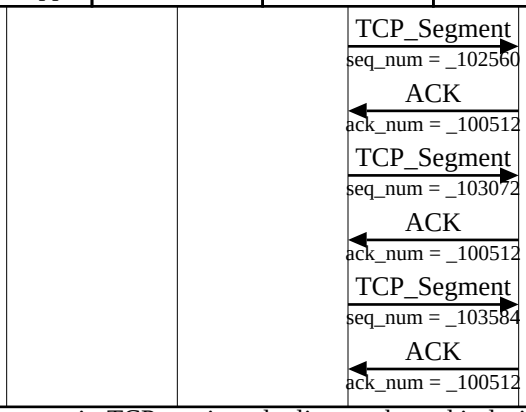
Server sets the SYN and the ACK bits in the TCP header. Server sends its initial sequence number as 100. Server also sets its window to 65535 bytes. i.e. Server has buffer space for 65535 bytes of data. Also note that the ack sequence numer is set to 1. This signifies that the server expects a next byte sequence number of 1

Now the server transitions to the SYN Received state

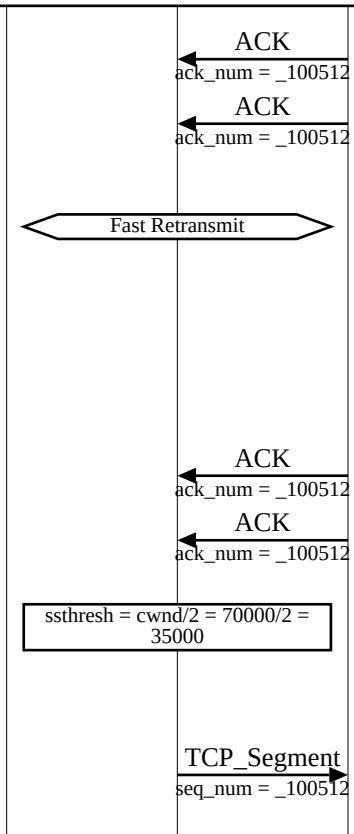
Client receives the SYN_ACK TCP segment

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
ACK
src = Client_Port, dst = Server_Port, ack_num = _101, window = _5000
Server receives the TCP ACK segment
Established
Now the server too moves to the Established state
LEG: Fast Retransmit and Recovery
TCP Connection begins with slow start. The congestion window grows from an initial 512 bytes to 70000 bytes
cwnd = 70000
Congestion window has reached 70000 bytes
Data
size = _4096
Client App transmits 4Kbytes of data
Segment data into 8 TCP segments
TCP segments data to 8 TCP segments (each segment is 512 bytes)
TCP_Segment
seq_num = _100000
TCP segment (start sequence number = 100000) is transmitted
TCP_Segment
seq_num = _100512
TCP segment (start sequence number = 100512) is transmitted
TCP_Segment
seq_num = _101024
TCP segment (start sequence number = 101024) is transmitted
TCP_Segment
seq_num = _101536
TCP segment (start sequence number = 101536) is transmitted
TCP_Segment
seq_num = _102048
TCP segment (start sequence number = 102048) is transmitted
TCP_Segment
seq_num = _102560
TCP segment (start sequence number = 102560) is transmitted
TCP_Segment
seq_num = _103072
TCP segment (start sequence number = 103072) is transmitted
TCP_Segment
seq_num = _103584
TCP segment (start sequence number = 103584) is transmitted
TCP_Segment
seq_num = _100000
TCP segment (start sequence number = 100000) is delivered to the receiver
Data
size = _512
TCP passes 512 bytes of data to the higher layer
TCP Segment with sequence number 100512 is lost
TCP segment (start sequence number = 100512) is lost due to congestion
TCP_Segment
seq_num = _101024
TCP Segment with start sequence number 101024 is received. TCP realizes that a segment has been missed. TCP buffers the out of sequence segment as TCP cannot deliver out of sequence data to the application.
ACK
ack_num = _100512
TCP sends an acknowledgement to the Sender with the next expected sequence number set to 100512.
TCP_Segment
seq_num = _101536
TCP receives the next segment. This and the following out of sequence segments will be buffered by TCP.
ACK
ack_num = _100512
TCP sends another acknowledgement with the next expected sequence number still set to 100512. This is a duplicate acknowledgement
TCP_Segment
seq_num = _102048
ACK
ack_num = _100512
TCP keeps acknowledging the received segments with the next expected sequence number as 100512



Fast Retransmit: TCP receives duplicate acks and it decides to retransmit the segment, without waiting for the segment timer to expire. This speeds up recovery of the lost segment



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.

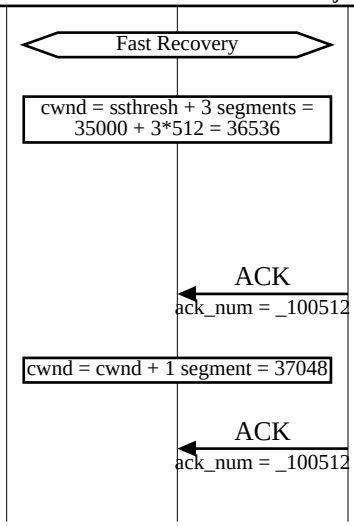
Second duplicate ack is received

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

Fast Recovery: Once the lost segment has been transmitted, TCP tries to maintain the current data flow by not going back to slow start. TCP also adjusts the window for all segments that have been buffered by the receiver.



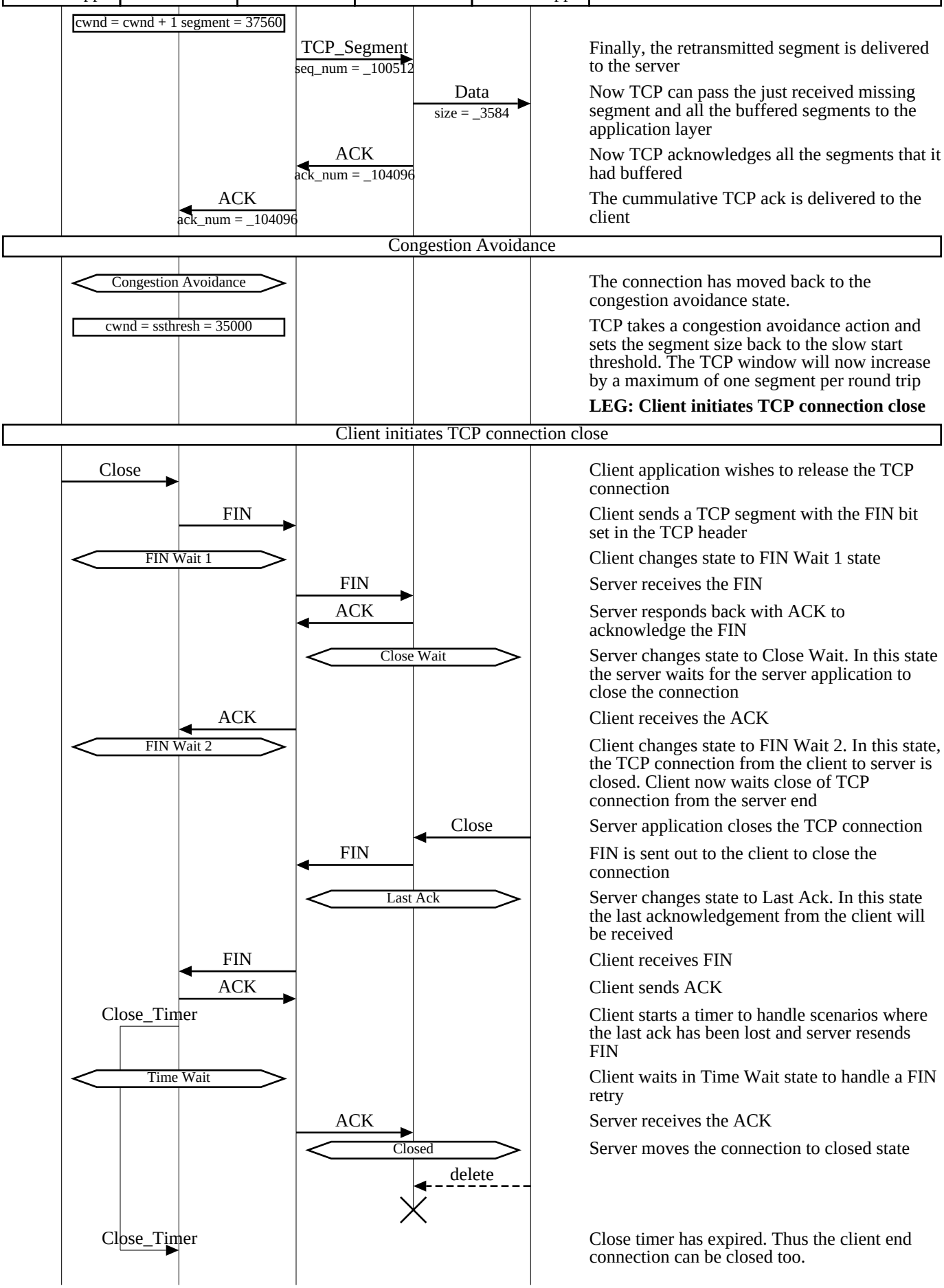
In "Fast Recovery" state, TCPs main objective is to maintain the current data stream data flow.

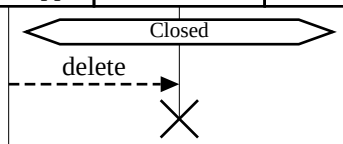
Since TCP started taking action on the third duplicate ack, it sets the congestion window to ssthresh + 3 segment. This halfs the TCP window size and compensates for the TCP segments that have already been buffered by the receiver.

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


cwnd = cwnd + 1 segment = 37560
TCP_Segment
seq_num = _100512
Finally, the retransmitted segment is delivered to the server
Data
size = _3584
Now TCP can pass the just received missing segment and all the buffered segments to the application layer
ACK
ack_num = _104096
Now TCP acknowledges all the segments that it had buffered
ACK
ack_num = _104096
The cummulative TCP ack is delivered to the client
Congestion Avoidance
Congestion Avoidance
The connection has moved back to the congestion avoidance state.
cwnd = ssthresh = 35000
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
LEG: Client initiates TCP connection close
Client initiates TCP connection close
Close
Client application wishes to release the TCP connection
FIN
Client sends a TCP segment with the FIN bit set in the TCP header
FIN Wait 1
Client changes state to FIN Wait 1 state
FIN
Server receives the FIN
ACK
Server responds back with ACK to acknowledge the FIN
Close Wait
Server changes state to Close Wait. In this state the server waits for the server application to close the connection
ACK
Client receives the ACK
FIN Wait 2
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
Close
Server application closes the TCP connection
FIN
FIN is sent out to the client to close the connection
Last Ack
Server changes state to Last Ack. In this state the last acknowledgement from the client will be received
FIN
Client receives FIN
ACK
Client sends ACK
Close_Timer
Client starts a timer to handle scenarios where the last ack has been lost and server resends FIN
Time Wait
Client waits in Time Wait state to handle a FIN retry
ACK
Server receives the ACK
Closed
Server moves the connection to closed state
delete
Close_Timer
Close timer has expired. Thus the client end connection can be closed too.


Closed
delete