| TCP - Transmission Control Protocol (TCP Fast Transmit and Recovery) | | | | | |
|---|---|---|---|---|---|
| Client Node | | Internet | Server Node | | EventStudio System Designer 4.0 |
| Client | | Net | Server | | |
| Client App | Client Socket | Network | Server Socket | Server App | 29-Jul-07 07:37 (Page 1) |

This diagram was generated with EventStudio System Designer 4.0. (http://www.EventHelix.com/EventStudio)



**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)

create — Server Application creates a Socket

Closed — The Socket is created in Closed state

seq_num = 100 — Server sets the initial sequence number to 100

Passive_Open — 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

Listen — Socket transitions to the Listen state

create — 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.

**LEG: Client initiates TCP connection**

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
src = Client_Port,
dst = Server_Port,
seq_num = 0

SYN TCP segment is received by the server

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

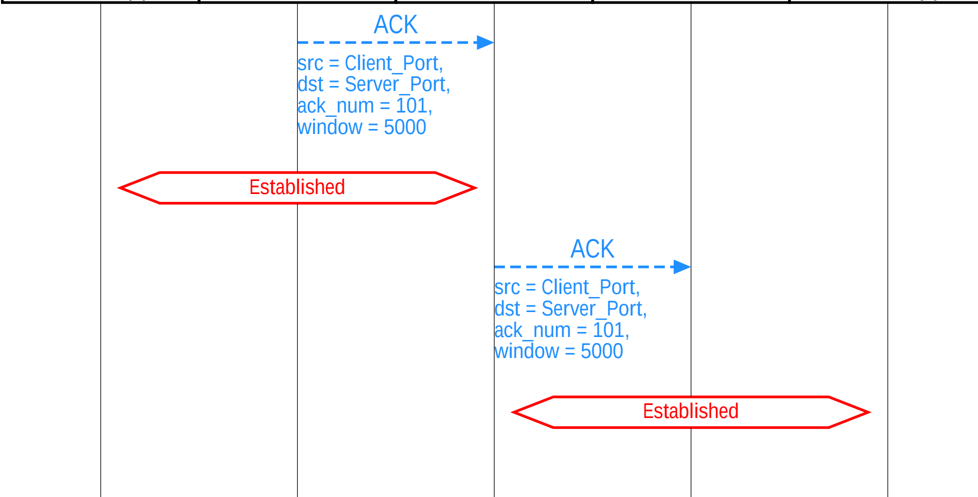
Server sets the SYN and the ACK bits in the TCP header. Server sends its initial sequence number as 100. Server also sets its window to 65535 bytes. i.e. Server has buffer space for 65535 bytes of data. Also note that the ack sequence numer is set to 1. This signifies that the server expects a next byte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

SYN Received — Now the server transitions to the SYN Received state

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

Client receives the SYN_ACK TCP segment

| TCP - Transmission Control Protocol (TCP Fast Transmit and Recovery) | | | | | |
|---|---|---|---|---|---|
| Client Node | | Internet | Server Node | | **EventStudio System Designer 4.0** |
| Client | | Net | Server | | 29-Jul-07 07:37 (Page 2) |
| Client App | Client Socket | Network | Server Socket | Server App | |




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.

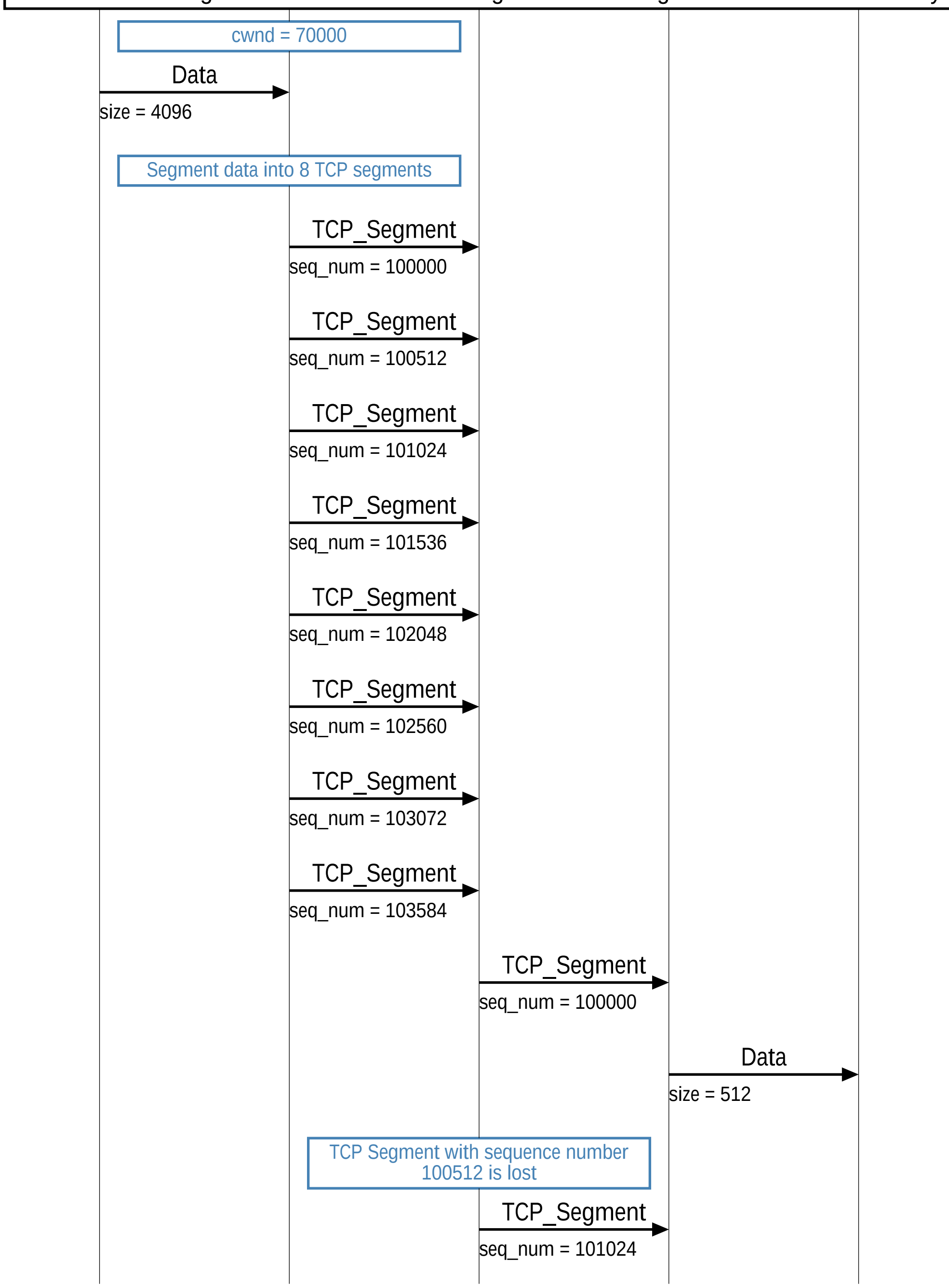
At this point, the client assumes that the TCP connection has been established

Server receives the TCP ACK segment

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




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

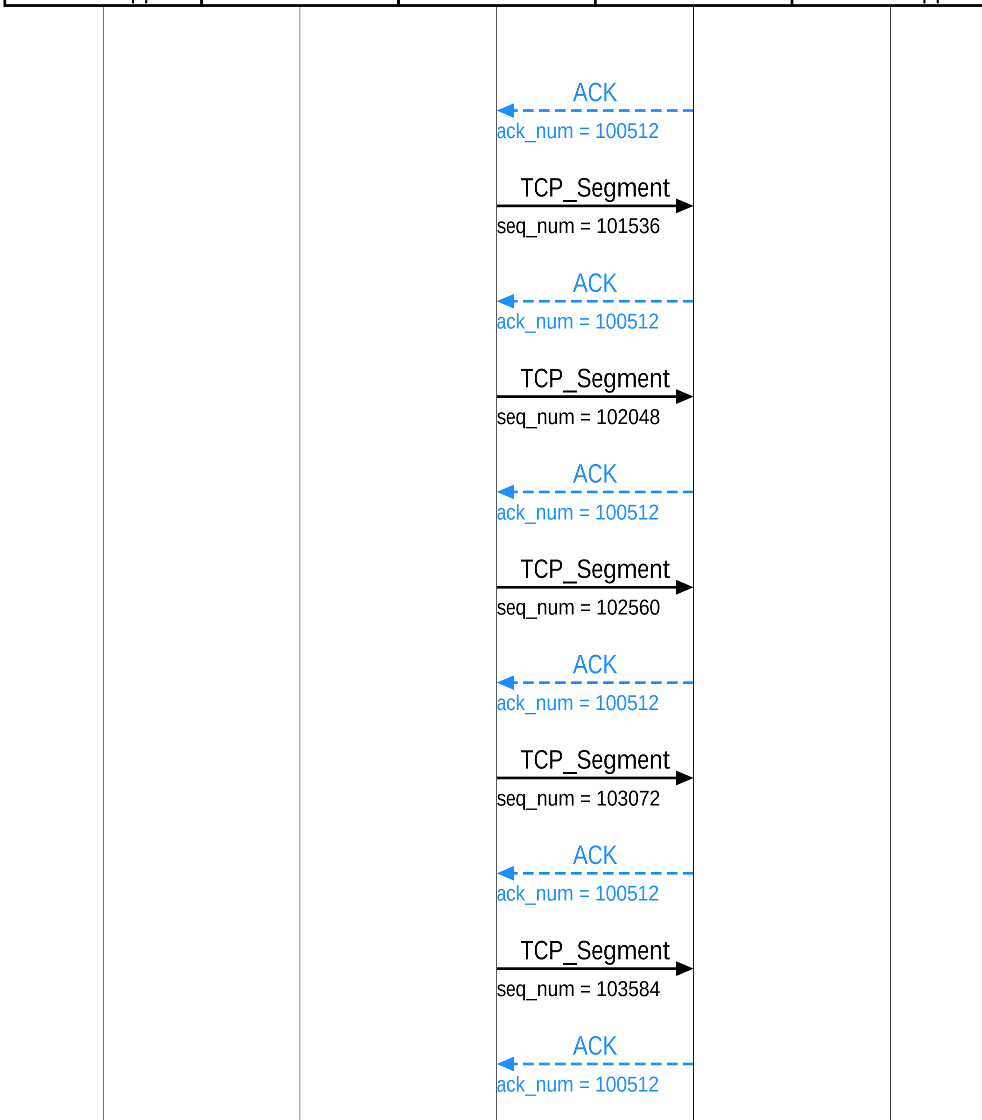
TCP segment (start sequence number = 100000) is delivered to the receiver

TCP passes 512 bytes of data to the higher layer

TCP segment (start sequence number = 100512) is lost due to congestion

TCP Segment with start sequence number 101024 is received. TCP realizes that a segment has been missed. TCP buffers the out of

| TCP - Transmission Control Protocol (TCP Fast Transmit and Recovery) | | | | | |
|---|---|---|---|---|---|
| Client Node | | Internet | Server Node | | EventStudio System Designer 4.0 |
| Client | | Net | Server | | 29-Jul-07 07:37 (Page 3) |
| Client App | Client Socket | Network | Server Socket | Server App | |




sequence segment as TCP cannot deliver out of sequence data to the application.

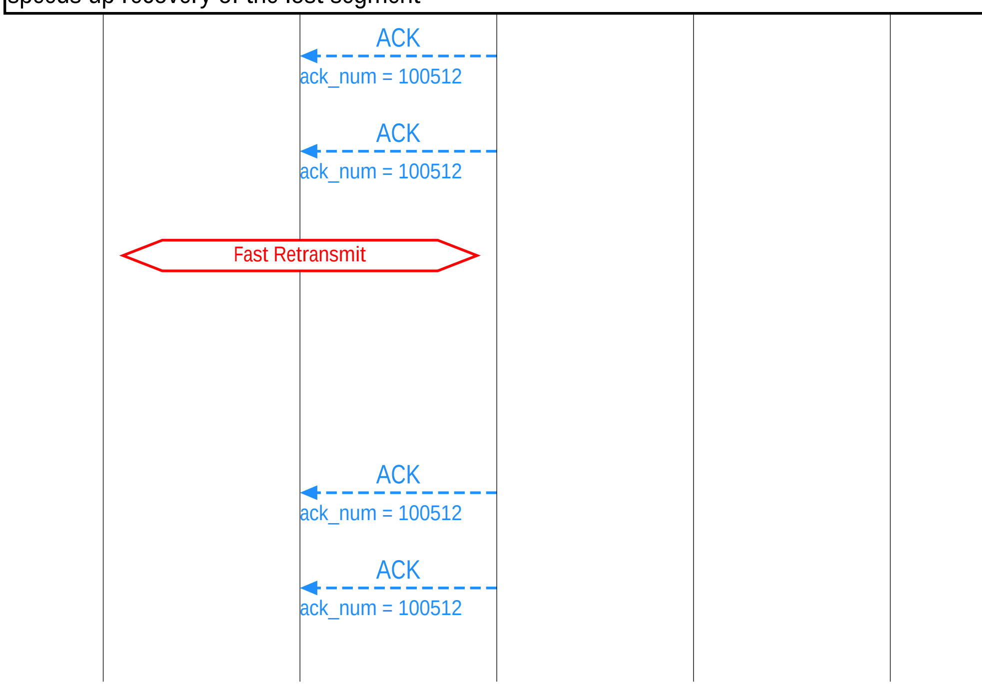
TCP sends an acknowledgement to the Sender with the next expected sequence number set to 100512.

TCP receives the next segment. This and the following out of sequence segments will be buffered by TCP.

TCP sends another acknowledgement with the next expected sequence number still set to 100512. This is a duplicate acknowledgement

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 - Transmission Control Protocol (TCP Fast Transmit and Recovery)**

| Client Node | | Internet | Server Node | | **EventStudio System Designer 4.0** |
|---|---|---|---|---|---|
| Client | | Net | Server | | 29-Jul-07 07:37 (Page 4) |
| Client App | Client Socket | Network | Server Socket | Server App | |

ssthresh = cwnd/2 = 70000/2 = 35000

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_Segment (Client Socket → Network)
seq_num = 100512

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.

Fast Recovery

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

cwnd = ssthresh + 3 segments = 35000 + 3*512 = 36536

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.

ACK (Network → Client Socket)
ack_num = 100512

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

cwnd = cwnd + 1 segment = 37048

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

ACK (Network → Client Socket)
ack_num = 100512

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

cwnd = cwnd + 1 segment = 37560

TCP_Segment (Network → Server Socket)
seq_num = 100512

Finally, the retransmitted segment is delivered to the server

Data (Server Socket → Server App)
size = 3584

Now TCP can pass the just received missing segment and all the buffered segments to the application layer

ACK (Server Socket → Network)
ack_num = 104096

Now TCP acknowledges all the segments that it had buffered

ACK (Network → Client Socket)
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 App → Client Socket)

Client application wishes to release the TCP connection

FIN (Client Socket → Network)

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 (Network → Server Socket)

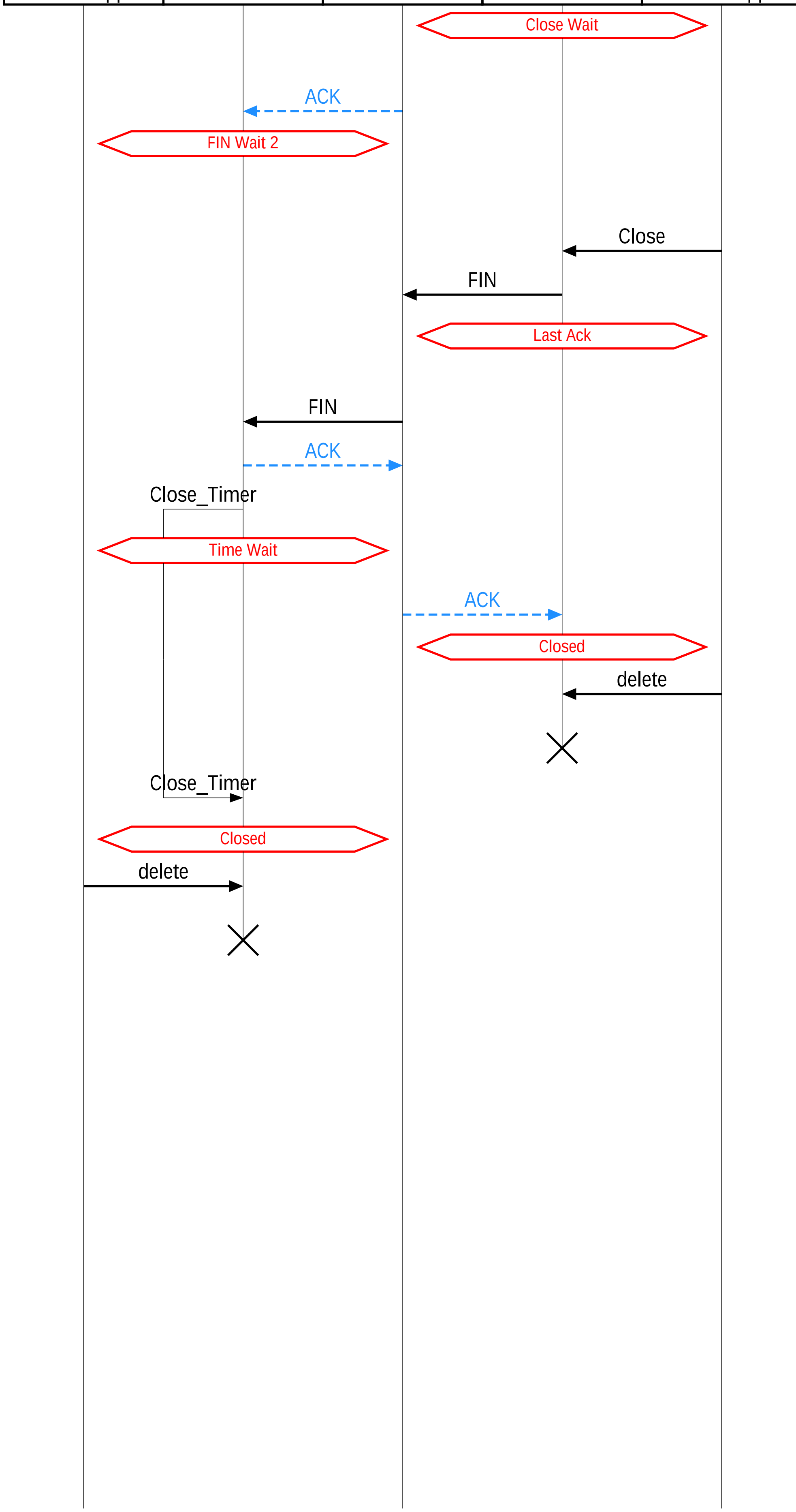
Server receives the FIN

ACK (Server Socket → Network)

Server responds back with ACK to acknowledge the FIN

**TCP - Transmission Control Protocol (TCP Fast Transmit and Recovery)**

| Client Node | | Internet | Server Node | | **EventStudio System Designer 4.0** |
|---|---|---|---|---|---|
| Client | | Net | Server | | 29-Jul-07 07:37 (Page 5) |
| Client App | Client Socket | Network | Server Socket | Server App | |




Server changes state to Close Wait. In this state the server waits for the server application to close the connection

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

Server application closes the TCP connection

FIN is sent out to the client to close the connection

Server changes state to Last Ack. In this state the last acknowledgement from the client will be received

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

Server receives the ACK

Server moves the connection to closed state

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