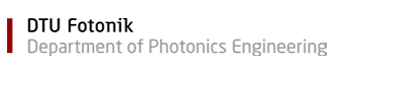
****

**Report**

23/09/2018

**Konstantinos Paraskevopoulos**

**S171743**

Contents

No table of contents entries found.

# **Abstract**

Transmission Control Protocol (TCP) is a quite old connection-oriented protocols invented back in 1938, but still used in a large-scale today’s networks. Motivated by the importance of TCP, in this project, there has been an effort to present and implement an architecture for the TCP byte-oriented stop and wait version. The approach used for generating the above-mention design is to divide the top-level architecture into 3 different entities. [Continue]

# **Introduction**

Transmission control protocol (TCP) invented by Robert Elliot Kahn in 1938 [Wikipedia bob kahn] introduces a mechanism that is able to setup and tear down a full duplex communication between an application that generates traffic and a server. The advantages that TCP offers in compare to the other most used layer 4 protocol (UDP) is that it is reliable as it guarantees the transmission and reception of error free packets and consists of a flow and congestion control mechanism. Since its invention, TCP has undergone many modifications and enhancements which aim exclusively to make TCP faster, and more reliable based on the evolving, bandwidth-hungry and aggregated speed that today’s applications are characterized.

Moreover, this project’s TCP application is based on the Stop and Wait ARQ protocol. However, while in the above-mention technique for managing the connection, the exploitation of 1 bit is enough in both the sequence and acknowledgement numbers in the TCP header format (as illustrated in figure 1), in our implementation, a byte-oriented approach was taken. The motivation for doing this is that the integration to the newest versions of TCP (e.g. Selective Repeat, Go-Back-N) is more feasible.

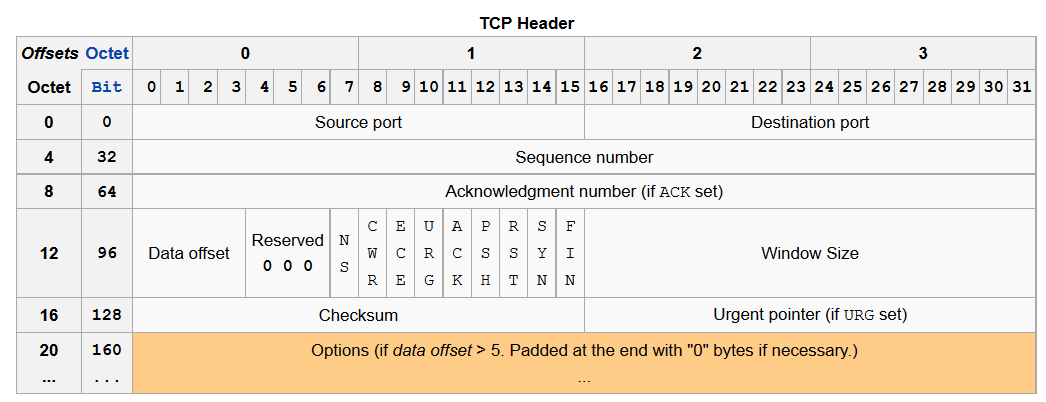
Having said that, it is also important to note that in our system, only TCP segments are processed, and they are compliant with the RFC defined format depicted in figure 1. Furthermore, due to the reason that the TCP checksum is dependent on the source and destination IP, 16 bytes have been added in front of the before mention format. Thus, in the next sections, when referring to packet, segment, or frame, they should all be considered as 16bytes concatenated with the standardized TCP format. Lastly, along with this format, up to 1480 bytes of data can be sent.

Figure 1: wikipedia image

# **Architecture**

# **TOE FSM entity**

In the following section, the central entity of the architecture is anatomized. An insight of its implementation can be gained by looking at figure x where a reduced TCP state diagram is illustrated. The states are named after [rfc] and the events that trigger the transition of states are also defined by the [rfc]. Bidirectional connection is achieved while parallel connections are not supported by the system design.

Moreover, the central entity is communicating through shared signals (interfaces) with the rest architecture except from buffers to ensure the smooth operation of the system. With regard to its operation tasks, it is responsible for controlling both incoming and outgoing traffic by setting and clearing the appropriate signals and construct headers.

According to the fsm in figure x, we can have two modes, the passive open and active open indicated by the application interface that sets the signal i\_active\_mode ‘0’ and ‘1’ respectively. This along with the ‘start’ signal forces our system to move either to LISTEN or SYN\_SENT state.

## **Active open**

In the active open scenario, the application resided in our system behaves as client and wants to initialize a connection. Thus, the following steps are taking place for the 3-way handshake:

1. Application passes to central entity the source and destination ip and port, subsequently the central entity constructs a SYN packet, commands the Tx engine to forward it and moves to the SYN\_SENT state.
2. In SYN\_SENT state if an ack timer expires, it triggers the machine to instruct the resend of the SYN packet again while the timeout timer if expires forces the system to close the connection.
3. If a SYN ACK is received, the system transitions to the ESTABLISHMENT state and its ready to start sending data.

## **Passive open**

1. When application sets the i\_active\_mode low, it drives the system to move to the LISTEN state while it provides the ip and port that the system should listen.
2. If a valid packet is received with the SYN flag high, the systems transitions to SYN\_RCVD state after instructing the Tx engine to forward an ACK response
3. In SYN\_RCVD, timers could force the system to behave as in active open mode, while when a valid ACK packet is received it will move to ESTABLISHMENT state.

## **Establishment State**

Both modes behave the same when in this state. More specifically, there is outgoing traffic from our system and incoming traffic through the network interface. Therefore, we could have one of the following scenarios:

1. A valid packet has arrived that either needs to be acked and its payload to be forwarded to our application or it is an acknowlegde of a packet already sent by our system, resulting to updating the sequence number.
2. The application located in our system wants to transmit data by setting the ‘last’ signal to high.
3. Both 1 and 2 are taking place at the same time.

## **4-way handshake termination**

Either side can initiate the termination handshake. However, the transition from ESTABLISHMENT to CLOSE\_WAIT or FIN\_WAIT\_1 is pertaining to who is commencing the termination.

Having said this, if our system receives a FIN packet, it will move to CLOSE\_WAIT state while at the same time, it will construct an ACK packet and inform the application residing in our system to close. From CLOSE\_WAIT state, when the application is ready to close, it will set the i\_open state to zero and the system will transition to LAST\_ACK by sending a FIN packet. Finally, when the last ACK is received the connection will terminate.

On the other hand, if our system’s application decides to close when in ESTABLISHMENT state, a FIN packet is constructed before transition to FIN\_WAIT\_1 state. While in FIN\_WAIT\_1 state and apart from timers there are the following three ways for triggering our system to change state:

1. An ACK of the previously sent FIN has been received forcing the system to move to FIN\_WAIT\_2.
2. A FIN packet has been received, which implies that before the application, that resides on the other side of the network interface, receives our FIN, it decided almost “simultaneously” to close the connection. As a response, an ACK packet is sent by our system while transition to CLOSING state.
3. A FIN, ACK packet is received by the external application, that leads to a 3-way handshake termination and moves the system to TIME\_WAIT state.

Lastly when in CLOSING or FIN\_WAIT\_2 state, the system waits for an ACK and FIN respectively to drive the system into TIME\_WAIT where the connection is terminated, after a timer expires.

A close up of a map

Description automatically generated

A close up of a map

Description automatically generated

As denoted before, the main part of the TOE fsm is to construct headers and acknowledgements and instruct the engines to forward data or discard it (Rx engine). Therefore, for testing the intentional behavior, a testcase has been generated which can be found in appendix section and from which, we can extract the information depicted in figure x regarding constructed packets.

A screenshot of a cell phone

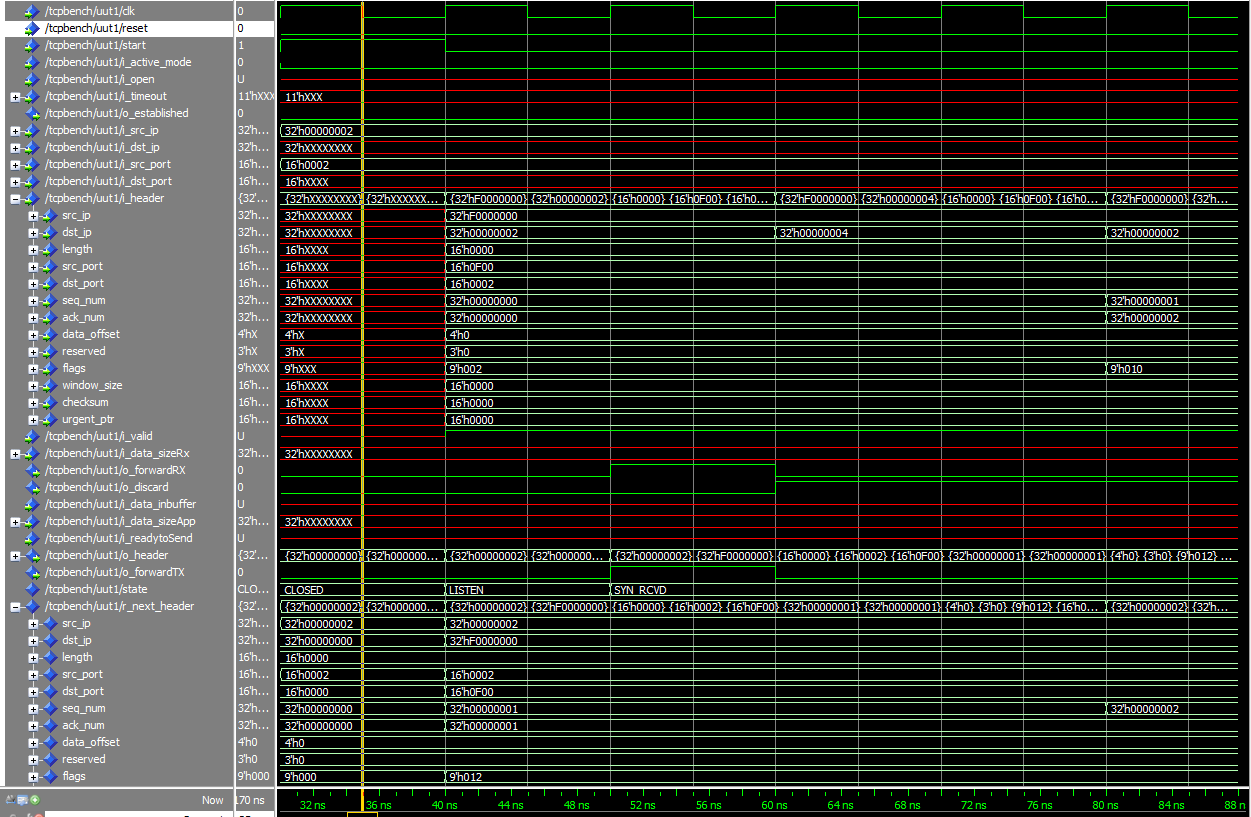
Description automatically generated

As can be seen when the internal application starts the communication after the required information has been fed to the TOE fsm by the application (source and dest ip, port), a SYN packet is constructed for start the 3-way handshake and forwarded to the network interface. Moreover, if at a given time both sides decide to transmit data and assuming our system receives data before the ack of the packet that it has sent, it will update the ack field and send the corresponding acknowledge response for the incoming data while it will wait until it receives the acknowledge of the packet that it has sent to update its sequence number.

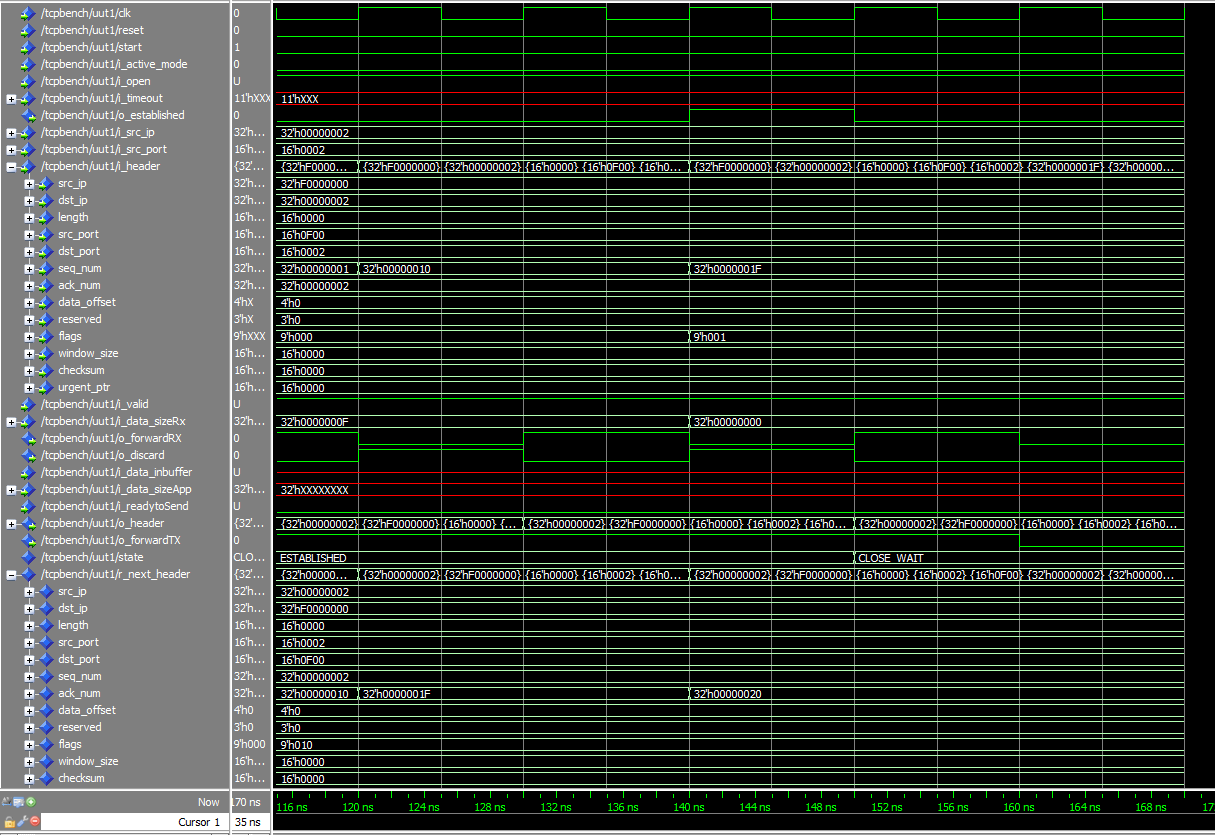
# **Appendix A**

## **Toe fsm**

### **1st Testcase**

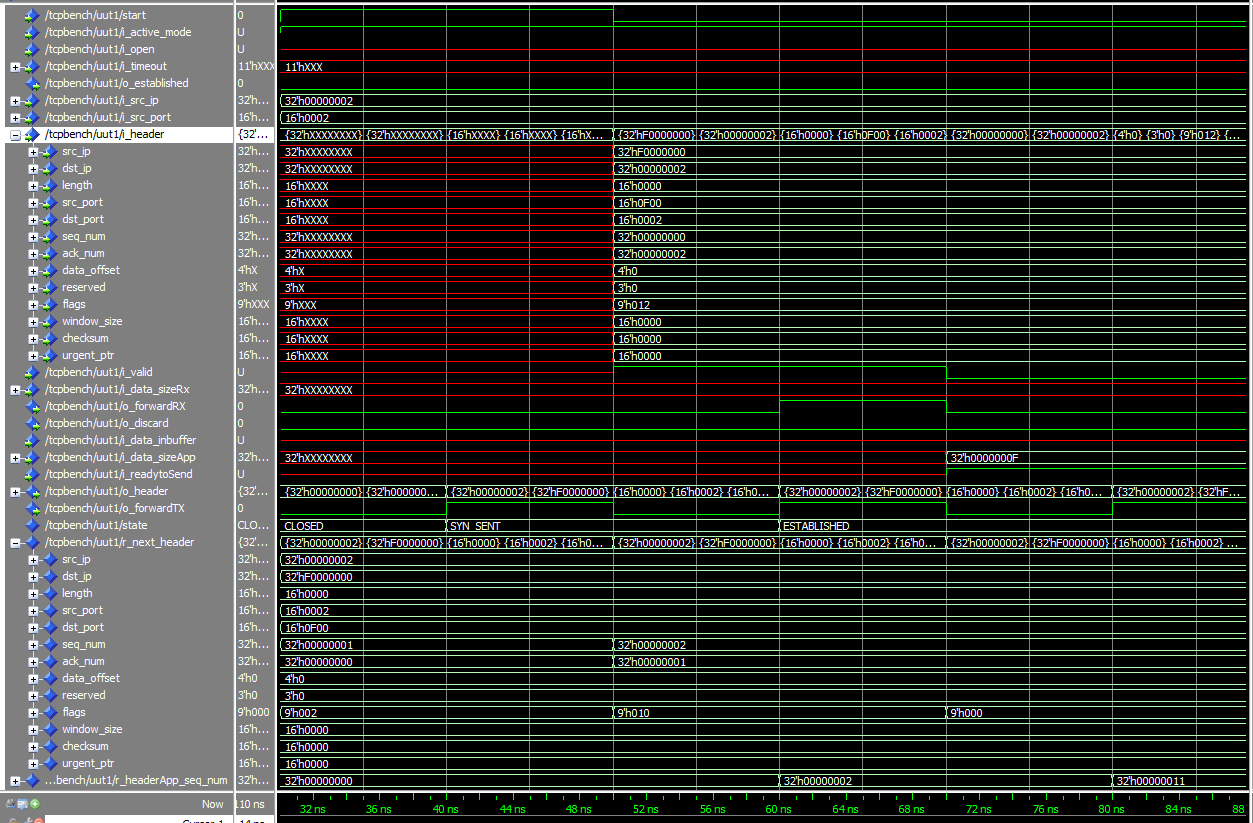
In the first testcase, only the internal application is sending data, while the external application is acknowledging these packets. In the following figures taken by ModelSim, it can be observed how the Toe fsm is constructing packets and how the acknowledgements are triggering our system to update its registers and act appropriately.

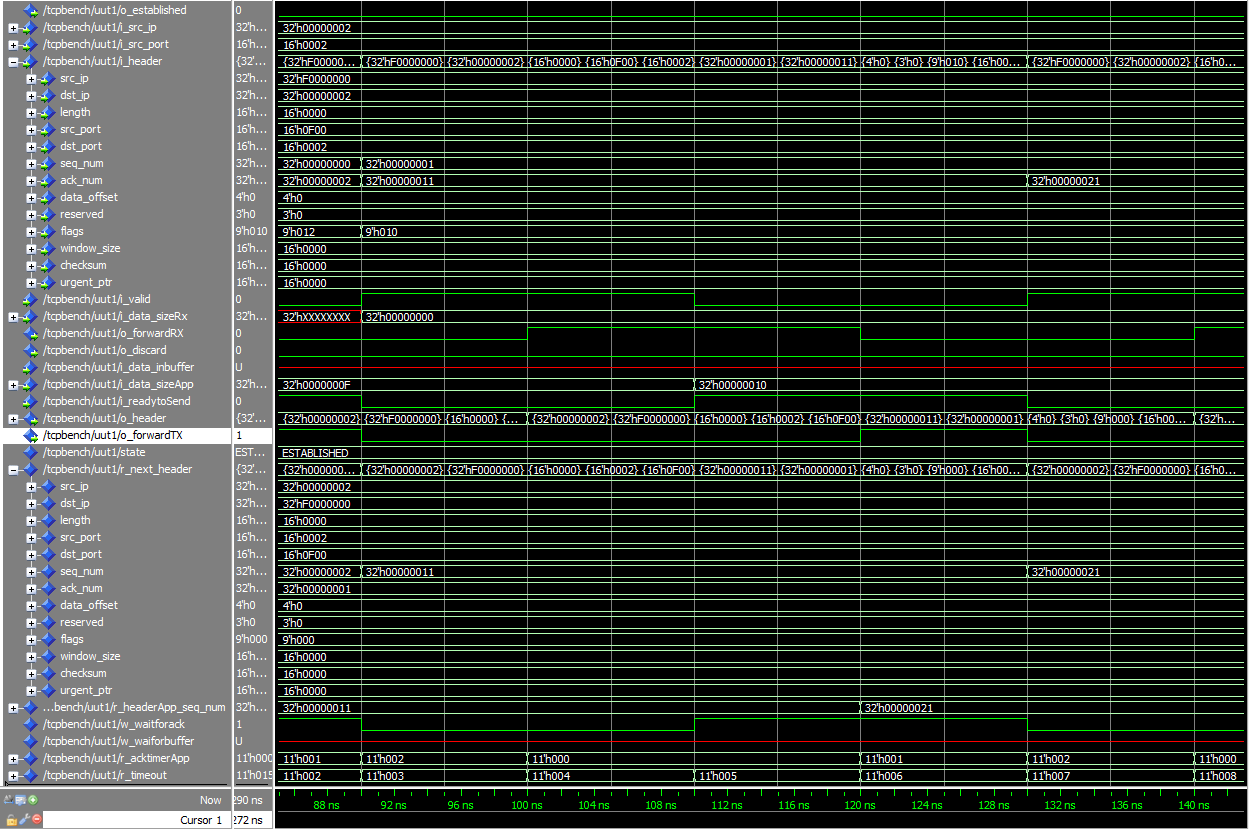
### 

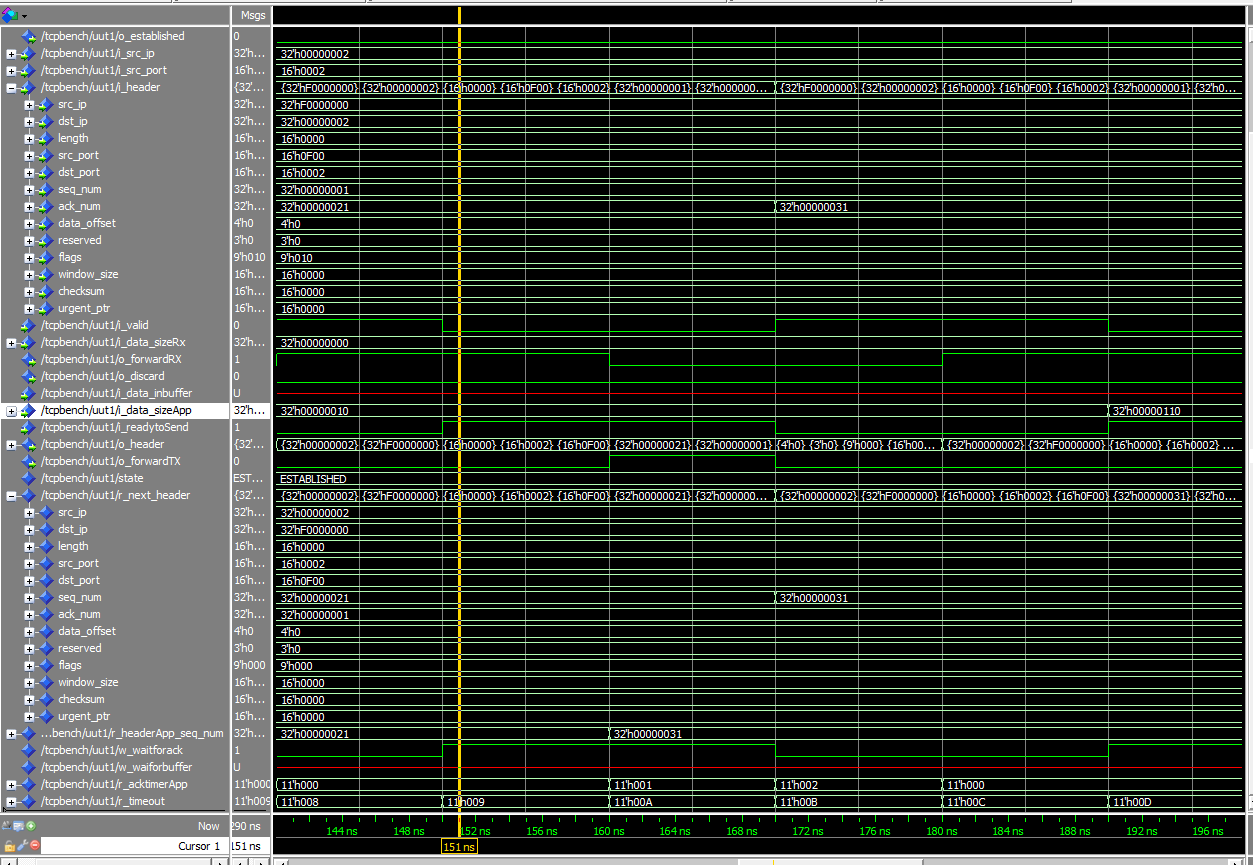


### **2st Testcase**

Moving to the next testcase, the reverse behavior has been simulated and reported namely, the external application is sending data while our system acknowledges these packets.







### **3rd Testcase**

The following testcase considers bidirectional traffic, where both internal and external applications are transmitting data. The figures depict the behavior of the Toe fsm regarding how it reacts to incoming as well as outgoing traffic.

