Some Aspects about a Self-Testing Solution for Implementing the TCP/IP Protocol

Daniela E. Popescu, Daniel Filipaș, Szabolcs Szilágyi
* Department of Computer Science, Faculty of Electrotechnics and Informatics, University of Oradea, Str. Universității 1, 410087 Oradea, Romania Phone: (+40) 259-408204,
E-Mail: depopescu@uoradea.ro, dfilipas@uoradea.ro, sszilagy@uoradea.ro;

Abstract TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications. The paper focuses on comparing the efficiencies of realizing a hardware implementation for the TCP protocol. In order to estimate the advantages of such a hardware implementation we show in this paper the comparative results of the classical implementation with that of a self-testing implementation of TCP.

1. Introduction

The Transmission Control Protocol (TCP) is intended to be a host-to-host protocol in common use in multiple networks; it is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks.
TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications. The TCP provides for reliable inter-process communication between pairs of processes in host computers attached to distinct but interconnected computer communication networks. In principle, the TCP should be able to operate above a wide spectrum of communication systems ranging from hard-wired connections to packet-switched or circuit-switched networks.

The TCP fits into a layered protocol architecture just above a basic Internet Protocol which provides a way for the TCP to send and receive variable-length segments of information enclosed in internet datagram "envelopes". The internet datagram provides a means for addressing source and destination TCPs in different networks. The internet protocol also deals with any fragmentation or reassembly of the TCP segments required to achieve transport and delivery through multiple networks and interconnecting gateways.

2. TCP Operational Overview and the TCP Finite State Machine (FSM)

A finite state machine (FSM) attempts to describe a protocol or algorithm by considering it like a virtual “machine” that progresses through a series of stages of operation in response to various happenings. A FSM describes the protocol by explaining all the different states the protocol can be in, the events that can occur in each state, what actions are taken in response to the events and what transitions happen as a result. The protocol usually starts in a particular beginning state when it is first run. It then follows a sequence of steps to get it into a regular operating state, and moves to other states in response to particular types of input or other circumstances. The state machine is called finite because there are only a limited number of states.
In the case of TCP, the finite state machine can be considered to describe the “life stages” of a connection. Each connection between one TCP device and another begins in a null state where there is no connection, and then proceeds through a series of states until a connection is established. It remains in that state until something occurs to cause the connection to be closed again, at which point it proceeds through another sequence of transitional states and returns to the closed state.

The full description of the states, events and transitions in a TCP connection is lengthy and complicated, since that would cover much of the entire TCP standard.

<table>
<thead>
<tr>
<th>State</th>
<th>State Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED</td>
<td>This is the default state that each connection starts in before the process of establishing it begins. The state is called “fictional” in the standard. The reason is that this state represents the situation where there is no connection between devices—it either hasn't been created yet, or has just been destroyed. If that makes sense.</td>
</tr>
<tr>
<td>LISTEN</td>
<td>A device (normally a server) is waiting to receive a synchronize (SYN) message from a client. It has not yet sent its own SYN message.</td>
</tr>
<tr>
<td>SYN-SENT</td>
<td>The device (normally a client) has sent a synchronize (SYN) message and is waiting for a matching SYN from the other device (usually a server).</td>
</tr>
<tr>
<td>SYN-RECEIVED</td>
<td>The device has both received a SYN (connection request) from its partner and sent its own SYN. It is now waiting for an ACK to its SYN to finish connection setup.</td>
</tr>
<tr>
<td>ESTABLISHED</td>
<td>The “steady state” of an open TCP connection. Data can be exchanged freely once both devices in the connection enter this state. This will continue until the connection is closed for one reason or another.</td>
</tr>
<tr>
<td>CLOSE-WAIT</td>
<td>The device has received a close request (FIN) from the other device. It must now wait for the application on the local device to acknowledge this request and generate a matching request.</td>
</tr>
<tr>
<td>LAST-ACK</td>
<td>A device that has already received a close request and acknowledged it, has sent its own FIN and is waiting for an ACK to this request.</td>
</tr>
<tr>
<td>FIN-WAIT-1</td>
<td>A device in this state is waiting for an ACK for a FIN it has sent, or is waiting for a connection termination request from the other device.</td>
</tr>
<tr>
<td>FIN-WAIT-2</td>
<td>A device in this state has received an ACK for its request to terminate the connection and is now waiting for a matching FIN from the other device.</td>
</tr>
<tr>
<td>CLOSING</td>
<td>The device has received a FIN from the other device and sent an ACK for it, but not yet received an ACK for its own FIN message.</td>
</tr>
<tr>
<td>TIME-WAIT</td>
<td>The device has now received a FIN from the other device and acknowledged it, and sent its own FIN and received an ACK for it. We are done, except for waiting to ensure the ACK is received and prevent potential overlap with new connections.</td>
</tr>
</tbody>
</table>

In Table 1, briefly describes each of the TCP states in a TCP connection, and also describes the main events that occur in each state, and what actions and transitions occur as a result. For brevity, three abbreviations are used for three types of message that control transitions between states, which correspond to the TCP header flags that are set to indicate a message is serving that function. These are:

- **SYN**: A synchronize message, used to initiate and establish a connection. It is so named since one of its functions is to synchronizes sequence numbers between devices.
- **FIN**: A finish message, which is a TCP segment with the FIN bit set, indicating that a device wants to terminate the connection.
• **ACK**: An acknowledgment, indicating receipt of a message such as a SYN or a FIN.

A TCP connection is always initiated with the 3-way handshake, which establishes and negotiates the actual connection over which data will be sent. The whole session is begun with a SYN packet, then a SYN/ACK packet and finally an ACK packet to acknowledge the whole session establishment. At this point the connection is established and able to start sending data. The FSM states are illustrated in Table 1. The FSM is illustrated in Figure 1 which you may find easier for seeing how state transitions occur.

![Figure 1: The TCP Finite State Machine (FSM)](image)

It's important to remember that this state machine is followed for each connection. This means at any given time TCP may be in one state for one connection to socket X, while in another for its connection to socket Y. Also, the typical movement between states for the two processes in a particular connection is not symmetric, because the roles of the devices are not symmetric: one
device initiates a connection, the other responds; one device starts termination, the other replies.

There is also an alternate path taken for connection establishment and termination if both devices initiate simultaneously (which is unusual, but can happen). Thus, for example, at the start of connection establishment, the two devices will take different routes to get to ESTABLISHED: one device (the server usually) will pass through the LISTEN state while the other (the client) will go through SYN-SENT. Similarly, one device will initiate connection termination and take the path through FIN-WAIT-1 to get back to CLOSED; the other will go through CLOSE-WAIT and LAST-ACK. However, if both try to open at once, they each proceed through SYN-SENT and SYN RECEIVED, and if both try to close at once, they go through FIN-WAIT-1, CLOSING and TIME-WAIT roughly simultaneously.

3. The Hardware Implementation Of The TCP Protocol

The TCP state machine implementation using ALTERA MAX+PLUS II is shown below (Figure 2). The matrix from Figure 3 contains a timing analysis on signal propagation from inputs to outputs, showing the time for the shortest and longest paths between them. We will use the longest path (the higher value) in our computation.

In our timing analysis, we used the following values:

- Clock period = 7.7 ns (the optimal value indicated by MAX+PLUS II).
- Segment_send = 5 X Clock = 5 X 7.7 = 38.5 ns (we considered that a TCP segment has 5 words only).
- Time_MSL = 100 ns (the maximum value for segment transmission).
Based on the FSM we prepared the TCP State Machine Implementation in Altera. The encapsulated TCP circuit is represented in Figure 3.

The self-testing implementation of TCP is made up of three main components:
1. TCP chip
2. Pseudo-Random Pattern Generator (PRPG)
3. Parallel Signature Analyzer (PSA)
For the implementation of the PRPG, we used a shift register with nine flip-flops, as shown in the figure below.

Figure 5: Pseudo-Random Pattern Generator (PRPG)

For the implementation of the PSA, we used a shift register with three flip-flops, as shown below.

Figure 6: Parallel Signature Analyzer (PSA)
The self-testing implementation of TCP is represented in the following way:

![TCP Self Test Symbol](image)

**Figure 7 : TCP Self Test Symbol**

The times for establishing a connection are:

**Server:**
CLOSED–LISTEN–SYN_RCVD–ESTABLISHED 19.7 ns + 18.1 ns + Segment_send + 17.3 ns = 93.6 ns

**Client:**
CLOSED–SYN_SENT–ESTABLISHED 19.7 ns + Segment_send + 18.1 ns + Segment_send = 114.8 ns

The times for closing a connection:

**Server:**
ESTABLISHED–CLOSED_WAIT–LAST_ACK–CLOSED 20.1 ns + Segment_send + 19.6 ns + Segment_send + 17.3 ns = 134 ns

**Client (we considered only the usual transitions):**
ESTABLISHED–FIN_WAIT_1–FIN_WAIT_2–TIME_WAIT–CLOSED 19.6 ns + Segment_send + 17.3 ns + 20.1 ns + Segment_send + 2 X Time_MSL = 334 ns

### 5. Conclusion

We made our comparison on the duration of transmitting and receiving TCP segments for different samples. SO, figure 4 shows the comparative timing results for the hardware and software implementation. This graphic demonstrates that the hardware implementation is much more faster than the software one.

The TCP state machine and the Self Test Logic used to test the TCP device can be built in the same chip. The PRPG provides the test patterns, so there is no need for an external generator.
Resource usage 1

- Logic A: Logic Cells: 68%
- Logic B: Logic Cells: 50%
- Logic A: I/O Pins: 62%
- Logic B: I/O Pins: 18%
- Logic A: Shareable Expanders: 0%
- Logic B: Shareable Expanders: 12%
- Logic A: External Interconnect: 100%
- Logic B: External Interconnect: 80%

Resource usage 2

- Total dedicated input pins used: 25%
- Total I/O pins used: 40%
- Total logic cells used: 43%
- Total shareable expanders used: 25%
- Total Turbo logic cells used: 56%
- Total shareable expanders not available (n/a): 6%
- Synthesized logic cells: 34%

TCP □ TCP with self test
Resource usage 3

<table>
<thead>
<tr>
<th>Total input pins required</th>
<th>Total output pins required</th>
<th>Total bidirectional pins required</th>
<th>Total logic cells required</th>
<th>Total flipflops required</th>
<th>Total product terms required</th>
<th>Total logic cells lending parallel expanders</th>
<th>Total shareable expanders in database</th>
<th>Average fan-in</th>
<th>Total fan-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>27</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>27</td>
</tr>
</tbody>
</table>

Device summary

<table>
<thead>
<tr>
<th>Input Pins</th>
<th>Output Pins</th>
<th>Bidir Pins</th>
<th>LCs</th>
<th>Shareable Expanders</th>
<th>Utilized</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>TCP with self test</td>
<td>TCP</td>
<td>TCP with self test</td>
<td>TCP</td>
<td>TCP with self test</td>
</tr>
</tbody>
</table>

- Input Pins: 11, 12
- Output Pins: 3, 3
- Bidir Pins: 0, 0
- LCs: 8
- Shareable Expanders: 13, 18
- Utilized: 84%
As shown in the charts above, the self-testing implementation of TCP uses the resources almost completely, whereas the classical implementation uses about one quarter of the total system resources.

References:

[3] Andrew Tannenbaum - Computer Networks
[4] Harald Welte - The journey of a packet through the linux network stack