ECSE-429

TCP Protocol

Simulated using Promela

Julian Cooper - 260246648
David Donna - 260366467
12/5/2012

Table of Contents

Introduction

In this project we will be implementing the TCP protocol using a Promela simulation. Our main basis for this implementation is the description provided in the project outline, specifically the finite state machine on page 4 (Figure 4) and the timing diagrams in figures 1-3. Furthermore, we referred to RFC-793 [1] for further clarifications of the protocol, in the case of conflicting descriptions, RFC-793 took precedence.

TCP is a protocol is a wrapper for the Internet Protocol, its functionality centers primarily around transmission reliability. This is a brief overview of some of its features and characteristics:

- Point-to-point:
    - Every connection has one sender and one receiver
- Connection-oriented:
    - Each message is encapsulated by handshaking/control messages
- Reliable, in-order data stream.
- Send and receive buffers
    - These buffers prevent unnecessary data retransmission caused by packet delay (not implemented in our model)
- Full duplex data:
    - Data can flow to-and-from sender and receiver
- Congestion management:

- TCP will modify its timeout delay to compensate for network congestion, limiting the number of unnecessarily retransmitted packets.

Our program will focus on the handshaking and data transmission aspects of TCP. That is to say, connection setup; data transmission, timeout, and retransmission; and clean connection teardown will be accurately modeled and thoroughly verified.

Specification

The program must simulate the creation, data-transfer, and tear-down of a TCP connection between a Client and a Server. The model must properly conduct the opening handshaking, the transmission of data after handshaking, and the tear-down of the connection after all data has been transmitted. It must also handle unexpected time-outs during data transmission.

Opening Handshaking

TCP can open a connection in two ways. The most common way is a three way handshake, initiated by a client and sent to a host. The second way only occurs when two machines attempt to establish a connection with each-other simultaneously.

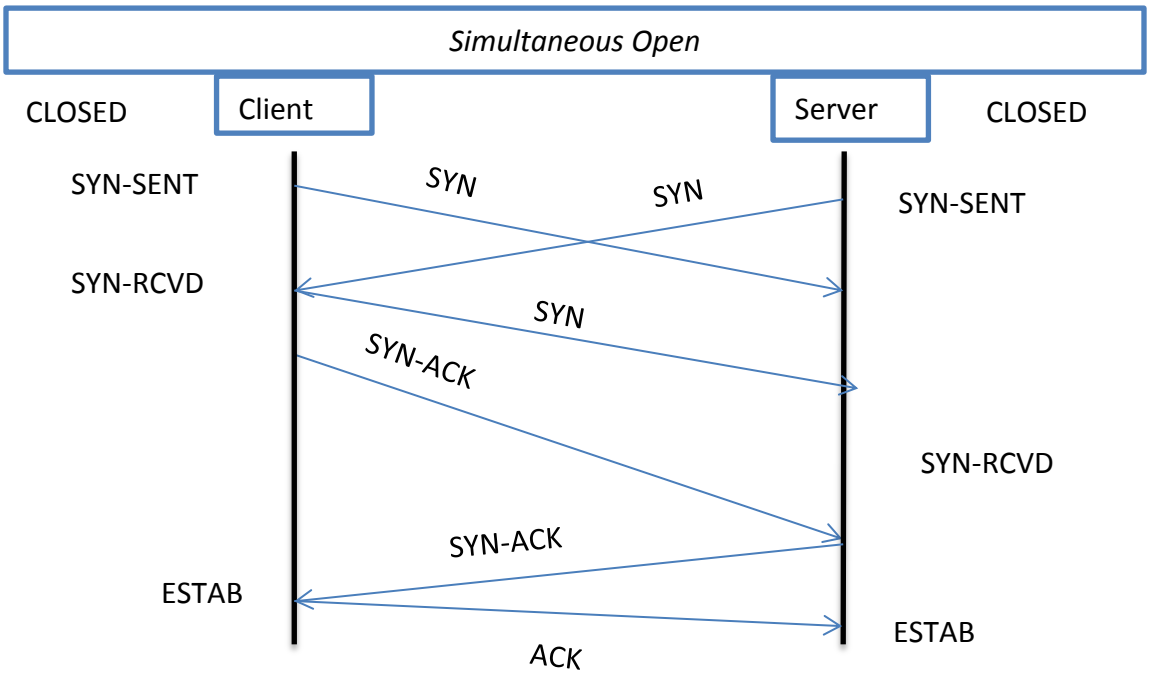
Thee Way Handshake Setup

Firstly, the client machine, A, will send a connection request to a host machine, B. The request is a message labelled SYN contains a sequence number, 'SEQ = *x*', which is the starting byte of the transmission sequence. Secondly, upon receipt SYN, B will send back a SYN-ACK,

with the data 'SEQ = y' and ACK = x + 1. Thirdly, A will send an ACK< SEQ = x+1, ACK = y+1 >. A can now send data to B.

Simultaneous Open

It is possible, though unlikely, that two machines attempt to initiate a connection with each other, in this case Both machines A and B send a SYN request, and both machines respond with a SYN+ACK, finally both machines send the last ACK. Both clients can send data to each other in the ESTAB state.



Data Transfer (ESTAB)

After a connection is established, Host A will data send packets to B, < SEQ = ACK_B, ACK = SEQ_B + 1> where ACK_B and SEQ_B are the last ACK and sequence numbers received from B,

after sending a data packet, A will wait for an ACK< SEQ, ACK > from B. Unfortunately, it is possible for data to be lost during transmission, as such, both A and B have value of *t* seconds. If A does not receive an ACK from B after *t* seconds, A will re-transmit the oldest packet that was not ACK'd, i.e its sequence number corresponds to the last ACK value received from B. Similarly, if B does not receive a data packet within *t* seconds after sending an ACK, it will assume that its ACK packet was dropped, and B will retransmit the ACK.

Connection Tear-down

The connection tear-down is another three way (giggity) handshake. Firstly, during the ESTAB state, the client, A, will send a FIN+ACK message to the server, B. Secondly, B will repeat the message back to A. Finally, A will ACKnowledge the receipt of that message. Once these three steps are completed, the connection is terminated and both machines enter initial state.

Implementation

As previously mentioned, our project is based on the finite state machine shown in Figure 4 of the project description. In our implementation, we simplified the FSM by splitting it in to two separate machines that run concurrently; one machine for the sender, and one machine for the receiver. Each state is a defined by a label within the Promela code, and the state transitions are governed by signals sent over channel data types. The current states of both machines is also stored in a set of Boolean variables, these variables are used to verify the correctness of the model during various points of execution.

Finite State Machines

A finite state machine can be formally described by a set of states, Q, a starting state $q_0 \epsilon\ Q$, a set of signals S, and transition function t, where δ takes a $q\ \epsilon\ Q$ and a signal $s\ \epsilon\ S$ which returns a signal to broadcast and the state to transition to.

Client FSM description

The client machine will not start in the LISTEN state, as it will automatically send a connection request.

CLOSED		
Transition Input Signal	**Transition Output Signal**	**Next State**
---	SYN	SYN_SENT
---	---	(Shutdown FSM)

Initial state, will automatically send SYN.

LISTEN		
Transition Input Signal	**Transition Output Signal**	**Next State**
---	SYN	SYN_SENT

Initial state, will just send SYN signal.

SYN_SENT		
Transition Input Signal	**Transition Output Signal**	**Next State**
SYN_ACK	ACK	ESTABLISHED
SYN	SYN_ACK	SYN_RECEIVED

This state occurs after the Client has decided to initiate a connection.

SYN_RECEIVED		
Transition Input Signal	**Transition Output Signal**	**Next State**
RST	---	LISTEN

This state is a part of the formal description of TCP, and exists in our FSM, however due to the implementation of one defined sender (Client) and one receiver (Server) the sender should never enter this state.

ESTABLISHED		
Transition Input Signal	**Transition Output Signal**	**Next State**
---	DATA	ESTAB_WAIT
ACK	DATA	ESTAB
---	FIN	FIN_WAIT_1

In established state, the client sends data to the server, then it waits for ACK, or it sends a tear-down request to the server

ESTAB_WAIT		
Transition Input Signal	**Transition Output Signal**	**Next State**
ACK	---	ESTABLISHED
timeout	---	ESTABLISHED

The client is waiting for the server to acknowledge the receipt of the DATA packet.

FIN_WAIT_1		
Transition Input Signal	**Transition Output Signal**	**Next State**
ACK	---	FIN_WAIT_2
FIN	ACK	CLOSING
FIN	FIN_ACK	TIME_WAIT

Step two of the tear-down handshake

FIN_WAIT_2		
Transition Input Signal	**Transition Output Signal**	**Next State**
FIN	ACK	TIME_WAIT

Tear-down complete, client shuts down connection.

CLOSING		
Transition Input Signal	**Transition Output Signal**	**Next State**
ACK	---	TIME_WAIT

Client A waits to ensure that there is no more data from B.

TIME_WAIT		
Transition Input Signal	**Transition Output Signal**	**Next State**
Timeout	ACK	CLOSED

A does not hear back from B, assumes that B received the FINACK.

Server FSM description

The Server acts as the receiver in the FSM simulation.

LISTEN		
Transition Input Signal	**Transition Output Signal**	**Next State**
SYN	SYN_ACK	SYN_RECEIVED

After receiving connection request, Server will acknowledge that message.

SYN_RECEIVED		
Transition Input Signal	**Transition Output Signal**	**Next State**
ACK	---	ESTABLISHED

Waits for third step of handshake and goes to data exchange state

ESTABLISHED		
Transition Input Signal	**Transition Output Signal**	**Next State**
DATA	ACK	ESTABLISHED
FIN	ACK	CLOSE_WAIT
timeout	FIN	ESTABLISHED

Server receives data from client, and acknowledges. On timeout, server will re-acknowledge the last data packet sent. Upon the receipt of FIN, server will begin its side of the tear-down procedure.

CLOSE_WAIT		
Transition Input Signal	**Transition Output Signal**	**Next State**
---	FIN	LAST_ACK

Second part of ACK-FIN

LAST_ACK		
Transition Input Signal	**Transition Output Signal**	**Next State**
ACK	---	CLOSED

Server received third part of handshake from Client. Can now safely shut down

TIME_WAIT		
Transition Input Signal	**Transition Output Signal**	**Next State**
Timeout	ACK	CLOSED

Implementation differences

Both machines should start in the LISTEN state, however in order to clearly define one sender and one receiver, the client will place the initial SYN message in the channel to begin the simulation.

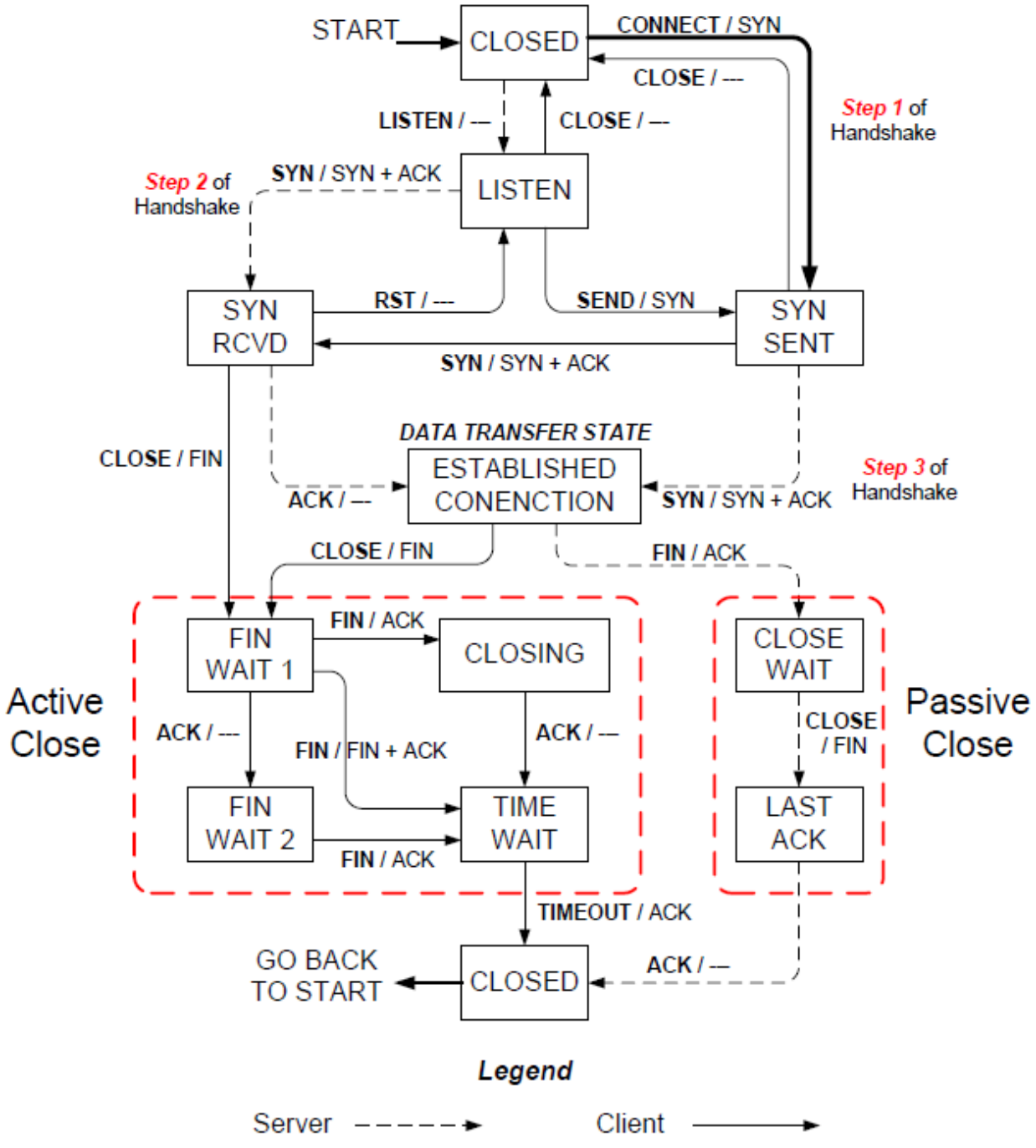


Figure 4: Finite State Machine of TCP Protocol

1

[1] 429-project-spec.pdf

Verification

Immediate State Assertions

See Appendix A for table of assertions and their descriptions. These assertions were used to monitor state consistency within the machine. Each state in each machine had its own set of assertions, to thoroughly make sure that it was in a valid state. The states are monitored using global variables, this way the Client can check if its current state coincided with a valid Server state, and vice versa. Assertions were also written to ensure that no unreachable states were ever reached. More specific details behind each assertion can be found in Appendix A.

LTL-Assertions

See Appendix B for all the LTL assertions created to verify the simulation. These assertions were designed to ensure correctness of state transitions, and further reinforce correct states.

Mutations

We grouped the assertions according to the overall pattern of behaviour that they enforced. For example, one group of assertions would notice if a process made a state transition that was not allowed by the FSM, another would trigger if a process received an unexpected message, and yet a third would make sure that sequence numbers and acks lined up correctly. In selecting our mutation tests, we made sure to exercise each type of assertion. So, we had at least one mutation that violated transition rules, at least one for a bad message, and so on.

Appendices

A: Immediate Assertions

#	*Line*	*Text*	*Checks for...*
01	113	false	Client exits closed state.
02	131	false	Client exits listen state.
03	138	inack == seq	Message received has correct ack value.
04	139	msg == SYN_ACK || msg == SYN	Message is of a valid type.
05	160	false	Client exits syn_sent state.
06	169	inack == seq	Message received has correct ack value.
07	170	msg == RST	Message is of a valid type.
08	181	false	Client exits syn_received state.
09	205	Inack == seq	Message received has correct ack value.
10	206	msg == ACK	Message is of a valid type.
11	218	false	Client exits established state.
12	225	inack == seq	Message received has correct ack value.
13	226	msg == ACK || msg == FIN	Message is of a valid type.
14	248	false	Client exits fin_wait_1 state.
15	257	inack == seq	Message received has correct ack value.
16	258	msg == FIN	Message is of a valid type.

17	263	false	Client exits fin_wait_2 state.
18	268	false	Client exits close_wait state.
19	277	inack == seq	Message received has correct ack value.
20	278	msg == ACK	Message is of a valid type.
21	281	false	Client exits closing state.
22	286	false	Client exits last_ack state.
23	298	false	Client exits time_wait state.
24	340	false	Server exits closed state.
25	349	msg == SYN	Message is of a valid type.
26	356	false	Server exits listen state.
27	365	msg == SYN	Message is of a valid type.
28	372	false	Server exits syn_sent state.
29	381	msg == ACK	Message is of a valid type.
30	382	inack == seq	Message received has correct ack value.
31	386	false	Server exits syn_received state.
32	393	msg == FIN || msg == DATA	Message is of a valid type.
33	394	inack == SEQ	Message received has correct ack value.
34	416	false	Server exits established state.
35	421	false	Server exits fin_wait_1 state.
36	426	false	Server exits fin_wait_2 state.

37	439	false	Server exits close_wait state.
38	444	false	Server exits closing state.
39	453	msg == ACK	Message is of a valid type.
40	454	inack == seq	Message received has correct ack value.
41	458	false	Server exits last_ack state.
42	463	false	Server exits time_wait state.

B: LTL Assertions

#	***Never...***	***Tests For...***
01	c_close_wait	Client never enters close_wait state.
02	c_last_ack	Client never enters last_ack state.
03	s_fin_wait_1	Server never enters fin_wait_1 state.
04	s_fin_wait_2	Server never enters fin_wait_2 state.
05	s_closing	Server never enters closing state.
06	s_time_wait	Server never enters time_wait state.
07	c_exit && <>!c_exit	Client never leaves exit state.
08	s_exit && <>!s_exit	Server never leaves exit state.
09	c_closed && !(c_closed U (c_syn_sent || c_exit))	Client transitions from closed to syn_sent or exit state.
10	c_listen && !(c_listen U (c_closed || c_syn_sent))	Client transitions from listen to closed or syn_sent state.
11	c_syn_sent && !(c_syn_sent U (c_established || c_syn_received))	Client transitions from syn_sent to established or syn_received state.
12	c_established && !(c_established U (c_fin_wait_1))	Client transitions from established to fin_wait_1 state.
13	c_fin_wait_1 && !(c_fin_wait_1 U (c_fin_wait_2 || c_closing || c_time_wait))	Client transitions from fin_wait_1 to fin_wait_2 or closing or time_wait state.
14	c_fin_wait_2 && !(c_fin_wait_2 U (c_time_wait))	Client transitions from fin_wait_2 to time_wait state.
15	c_closing && !(c_closing U c_time_wait)	Client transitions from closing to time_wait state.

16	c_time_wait && !(c_time_wait U c_closed)	Client transitions from time_wait to closed state.
17	s_closed && !(s_closed U (s_listen || s_syn_sent || s_exit))	Server transitions from closed to listen or syn_sent or exit state.
18	s_listen && !(s_listen U s_syn_received)	Server transitions from listen to syn_received state.
19	s_syn_sent && !(s_syn_sent U s_established)	Server transitions from syn_sent to established state.
20	s_syn_received && !(s_syn_received U s_established)	Server transitions from syn_recieved to established state.
21	s_established && !(s_established U s_close_wait)	Server transitions from established to close_wait state.
22	s_close_wait && !(s_close_wait U s_last_ack)	Server transitions from close_wait to last_ack state.
23	s_last_ack && !(s_last_ack U s_closed)	Server transitions from last_ack to closed state.

All of the above were compiled in to ltl.pml

C: Mutations

#	Line	Code should...	Assertion	Caught
01	467	Let process end	LTL_08	yes
02	384	Use a goto to switch states.	IMM_31	yes
03	350	Increment ack.	IMM_03	yes
04	193	Go to fin_wait_1 state.	LTL_12	yes
05	198	Send DATA instead of SYN.	IMM_32	yes
06	193	Client should never enter close_wait state.	LTL_01	yes
07	437	Server should never enter closing state.	LTL_05	yes
08	106	Client should send SYN instead of ACK.	IMM_25	yes
09	134	Client state should be SYN_SENT, never LAST_ACK	LTL_02	yes