Validation and Verification of the Transmission Control Protocol (TCP)

Lineker Tomazeli (260348949), Etienne Perot (260377858)

December 5, 2012

ECSE 429 Software Validation
Professor Katarzyna Radecka
McGill University

Table of Contents

Introduction

TCP (Transmission Control Protocol) is an important protocol used on an IP network. It serves as the transport protocol for most applications requiring reliable, ordered data transfer, thanks to its integrity and reliability guarantees. Since an IP network doesn't provide those guarantees, TCP ensures them through the use of timeouts, retransmissions, and acknowledgements.

Our model covers a subset of TCP, in which the data transmission is unidirectional (from initiator to recipient) and in which only the initiator may close the connection. We do provide TCP's data integrity guarantee during data transmission, but we do not support packet loss during the connection set-up and teardown phases.

Because of this simplified design and the differentiation between initiator (sender) role and recipient (receiver) role, we have two separate processes, each with their own finite state machine, rather than one single giant finite state machine attempting to cover both cases.

We have implemented an additional feature that wasn't on the assignment specification: message acknowledgement retransmission. Since we assume that packet loss may occur at any point during the central phase of the protocol (message transmission), then we need to consider both the case where a message sent by the sender gets lost, and the case where an acknowledgement message sent by the receiver gets lost.

Specification

The following two sections will explain in detail the specification for a Sender and Receiver. The original FSM used to derive our specification can be found on the Appendix - Figure I.
On our experiments the Sender is responsible for starting a connection with the receiver, sending messages, and initiate the tear down with the Receiver. Communication from Sender to Receiver and from Receiver to Sender is done through channels, and state variables are used to switch from/ to different states.

Sender

The sender follows the following finite state machine:

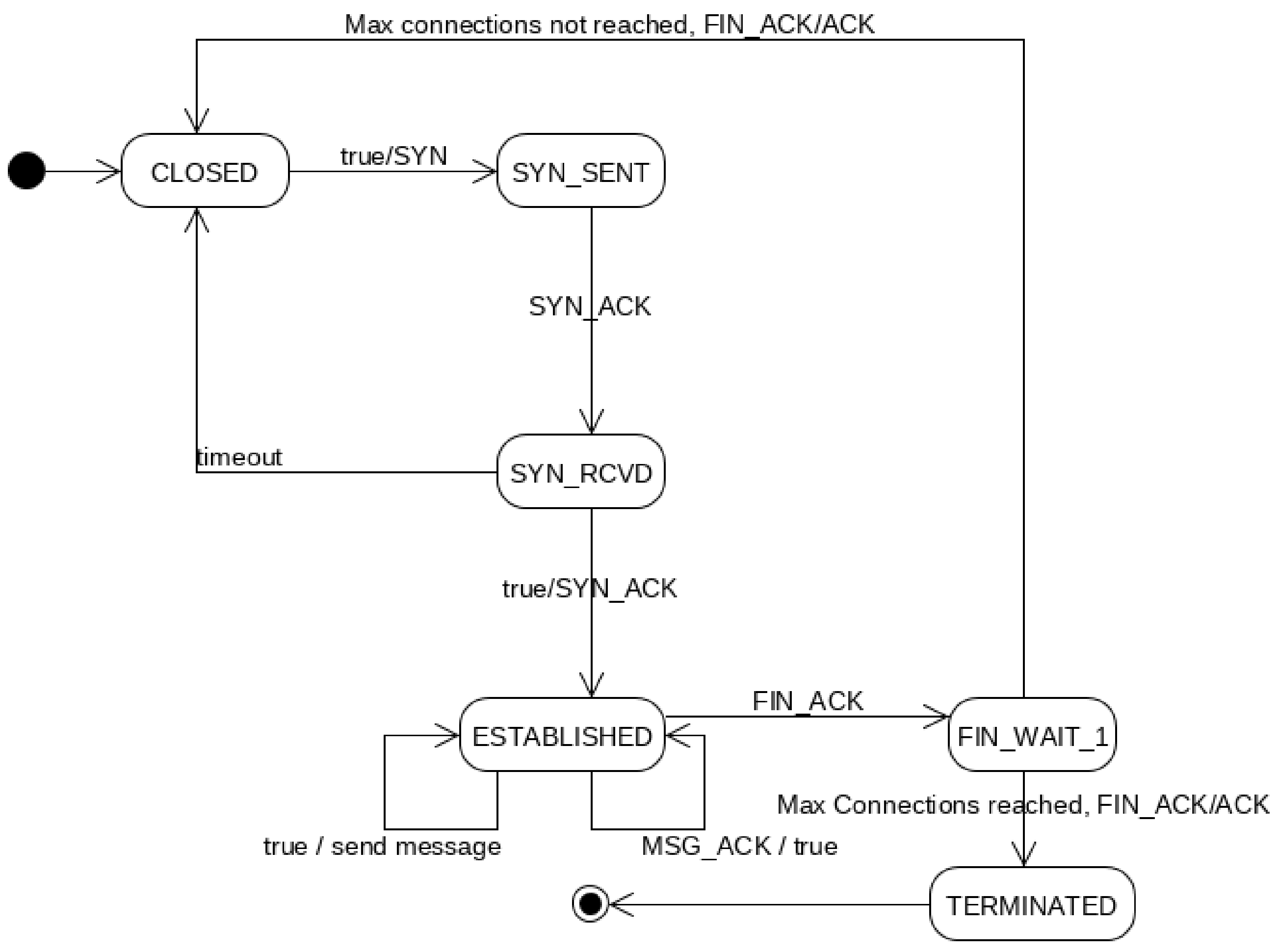


Figure 1 - Sender finite state machine

The sender starts in the CLOSED state. In this state, it initialize some necessary variables to hold unique identifier information, messages, number of messages and number of (re)connections. After initialization is done, it sends a connection request (SYN) to the receiver and as shown in Figure 1, it moves to state SYN_SENT where it waits for an acknowledgement message (SYN_ACK) from the receiver. Once SYN_ACK is received it moves to state SYN_RCVD. At this state, an integrity check is done to verify that the SYN_ACK and the unique identifier returned by receiver is consistent. If receiver has returned an invalid identifier, it will move to the CLOSED state and try to reconnect. Otherwise, sender increments the receiver unique identifier and sends it back with an SYN_ACK message. At this point, the sender has moved to the ESTABLISHED state and assumes that the connection was successfully established.

To send messages, the sender uses a channel dedicated for data transfer. It should send the message and its unique identifier so the receiver that differentiate between new messages and retransmissions. After sending the message, the sender has three options: It can either wait for message received acknowledgements (MSG_ACK) from the receiver; or it can Timeout because it never received a MSG_ACK. In the case where a MSG_ACK was successfully received, it has three possible outcomes: the acknowledgment message and its unique identifier corresponds to the last message sent by the sender, in which case we know that the receiver successfully received our message. Then we can either send another message or decide that we want to close the connection. If the unique identifier doesn't match, it means we received an old MSG_ACK, which we simply ignore; or we can decide to just ignore the message, to simulate that something went wrong with the network. This process continues until sender decides that it wants to close the connection.

Upon deciding to close the connection, the sender sends a FIN_ACK message to the receiver, and moves to state FIN_WAIT_1. At this point, it waits for a termination acknowledgement (FIN_ACK) from the receiver, but at the same time will make sure to handle correctly any leftover message acknowledgement on the sender channel. Once a FIN_ACK message is received from receiver, the sender will send an acknowledgement (ACK) and consider the connection closed.

At this point, the sender will flush its channel, and increase its counter of connection attempts. If the sender has reached the maximum number of connections allowed (an arbitrary cap on the number of connections used to limit the size of the state space of our model), it moves to the TERMINATED state. Otherwise, it goes back to the CLOSED state and will soon initiate another connection with the receiver.

Receiver

The receiver follows the following finite state machine:

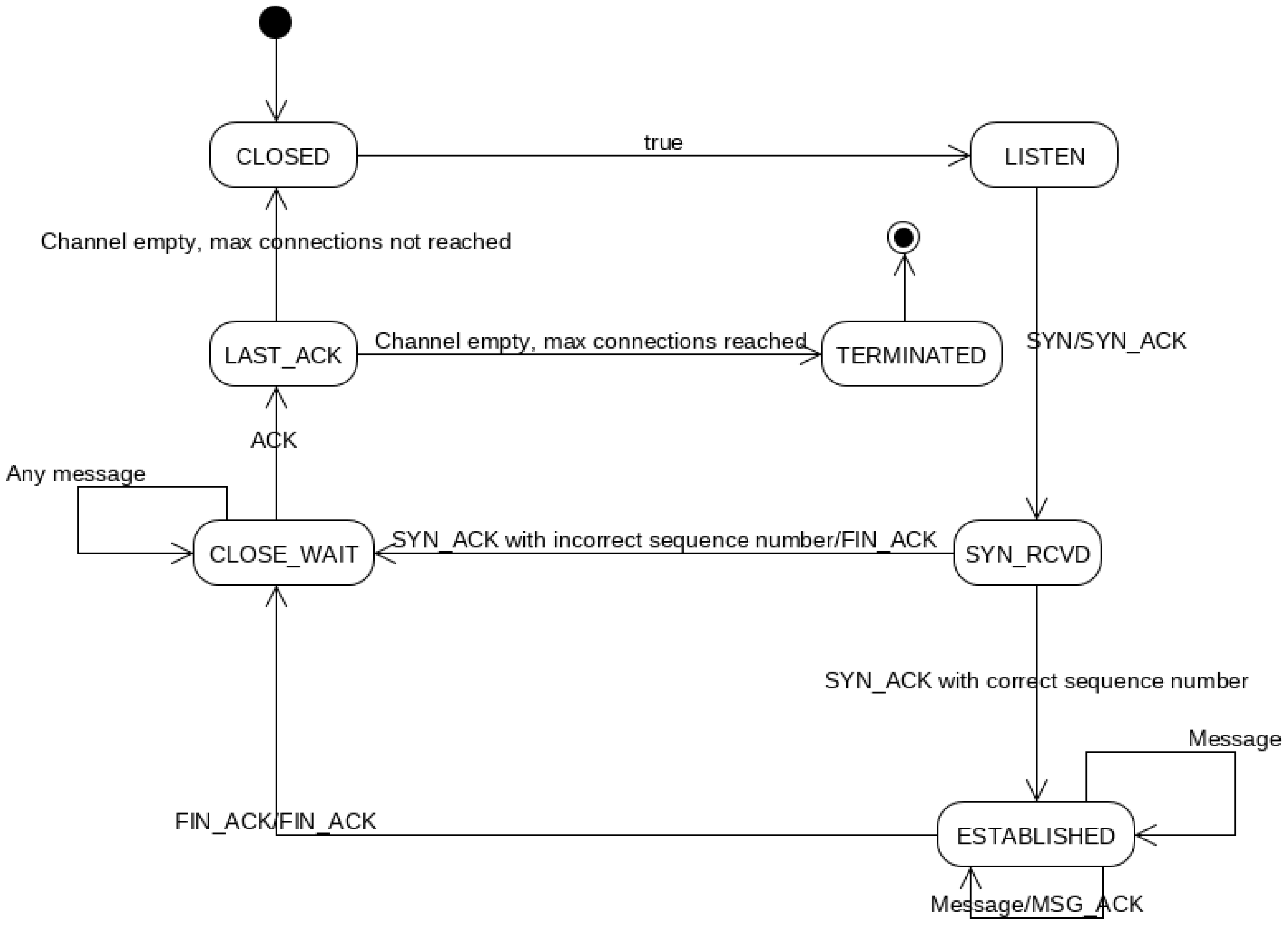

CLOSED
true
LISTEN
Channel empty, max connections not reached
LAST_ACK
Channel empty, max connections reached
TERMINATED
SYN/SYN_ACK
ACK
Any message
CLOSE_WAIT
SYN_ACK with incorrect sequence number/FIN_ACK
SYN_RCVD
SYN_ACK with correct sequence number
Message
FIN_ACK/FIN_ACK
ESTABLISHED
Message/MSG_ACK

Figure 2 - Receiver finite state machine

The receiver starts in the CLOSED state. In this state, it initializes some state variables, then moves directly to the LISTEN state, in which it is ready to receive connection requests.

Once the receiver receives a connection request (SYN), it sends back a SYN acknowledgement message (SYN_ACK) and moves on to the SYN_RCVD state. It expects a SYN_ACK message back, with the correct sequence number. If the sequence number is incorrect, it jumps into the CLOSE_WAIT state where it doesn't accept any data until the sender closes the connection. If the sequence number is correct, it goes into the ESTABLISHED state where it is ready to listen for messages.

While in the ESTABLISHED state, the receiver will accept any number of messages on the message channel. A message on the message channel has an ID and a payload. Upon receiving a message, if the receiver hasn't previously seen a message with this ID, it will either do nothing (simulating packet loss), or it will acknowledge the packet (send a MSG_ACK with the ID of the message). Otherwise, if the message ID has already been seen by the receiver, then this message is a retransmission of a previous message. In this situation, the receiver can either do nothing (simulating packet loss), or acknowledge the message again (to compensate for the fact that the MSG_ACK packet sent by the receiver may have been lost due to packet loss).

If the receiver receives a FIN_ACK message on the receiver channel, then it stops accepting messages and jumps into the CLOSE_WAIT state, sending back a FIN_ACK message to the sender in the process.

In the CLOSE_WAIT state, the receiver flushes out any remaining message on the message channel. Indeed, it is possible for the sender to both send some messages and a FIN_ACK before the receiver has a chance to act on either of them. As such, there may be some messages left in the message channel by that time. If that is the case, we simply ignore them. The sender will not get acknowledgements for them, but that is fine because at this point the sender has already given up and closed the connection. Therefore, the sender assumes that we have not received these messages, thereby guaranteeing the TCP's integrity guarantee: we may not have transmitted everything requested, but the sender and the receiver both exactly agree on what was successfully transmitted.

Once the receiver receives the sender's final ACK message, it jumps into the LAST_ACK state. In this state, it checks whether we have reached the maximum number of connections allowed by our model (an arbitrary cap on the number of connection attempts in order to keep the state space short enough). If so, then it goes to the TERMINATED state and terminates. Otherwise, it goes back to the CLOSED state, to jump to the LISTEN state shortly afterwards, ready for a new connection.

Implementation

We implemented the model using the **Promela** language we used the **spin** model-checking tool.

Our implementation is split into multiple Promela files:

tcp.pml: The main TCP model. Doesn't contain much by itself other than state variables and constant declarations. It includes sender.pml and receiver.pml.

sender.pml: Defines the sender proctype, along with some sender-related macros.

receiver.pml: Defines the receiver proctype, along with some receiver-related macros.

assertions/*.pml: All our assertions are spread out in various Promela files located in the assertions folder. Each assertion includes tcp.pml, therefore each assertion can be run directly with spin in order to check it.

mutants/*.pml: All our mutants are also spread out in various Promela files located in the mutants folder. Each mutant includes tcp.pml, therefore each mutant can be run directly with spin. However, to use assertions together with a mutant, the tcp.pml include directive needs to be replaced with whichever assertion we wish to test against the mutant. The checkmutant.sh script automates this process.

modelcheck.sh: Runs spin's model checker over a file given as argument. If no argument is given, runs the model checker over tcp.pml.

checkassertions.sh: Runs spin's model checker over a given assertion .pml file, or a directory containing assertions given as argument. If no arguments is given, runs the model checker over all assertions in the assertions folder.

checkmutant.sh: Iterates over all mutants in the mutants directory. For each mutant .pml file, check it against each assertion .pml file in the assertions directory. Whenever one assertion catches the mutant, move on to the next mutant. If a mutant .pml file is given as argument, restricts the search that single mutant file.

Sender

The sender implementation is all in sender.pml, with some state variables defined in tcp.pml.
Each state is preceded by a label (prefixed with `l_`) and enclosed in brackets.

This section documents each part of the implementation. It doesn't include any mutant-inserted code and omits some `printf()` statements, however.

Initialization:

```
active proctype Sender()
{
	int senderuid = 0, receiveruid, message = 1, temp, totalconnections = 0,
	nummessages;
	...
```

The sender is initialized, declaring its state variables:

`senderuid` will contain the sender's sequence number used in the 3-way handshake

`receiveruid` will contain the ID used for the receiver acknowledgement number during the 3-way handshake

`message` will hold the output message payload

`temp` will be used as variable to hold various things from incoming messages

`totalconnections` will contain the number of connection attempts

CLOSED:

```
	l_CLOSED: {
		senderState = CLOSED; /* Initial state */
		(receiverState == LISTEN);
		nummessages = 0; /* We have sent 0 messages so far */
		do /* Flush sender channel */
		:: senderchan ? _, _, _;
		:: empty(senderchan) -> break;
		od;
		senderuid = senderuid + 1;
		atomic {
			receiverchan ! SYN, senderuid, 0; /* Send SYN */
			senderState = SYN_SENT;
		}
	}
```

The sender sets its state variable to CLOSED and waits for `receiverState` to be at LISTEN. Once receiver is listening, it sets the number of messages that it has sent, flushes its own channel, and increments its `senderuid`. Once this initial setup is done, the sender is ready to send a SYN message and moves to state SYN_SENT.

SYN_SENT:

```
        l_SYN_SENT: {
                atomic {
                        senderchan ? SYN_ACK, receiveruid, temp;
                        senderState = SYN_RCVD; /* Change State */
                }
        }
```
The sender then waits for a SYN_ACK message to arrive. Once it does, it goes to state SYN_RCVD (SYN received).

SYN_RCVD:
```
        l_SYN_RCVD: { /* State once we've received a SYN+ACK message */
                if
                :: temp != senderuid + 1 ->
                        goto l_CLOSE; /* senderuid doesn’t match */
                :: else -> skip;
                fi;
                senderuid = temp;
                receiveruid = receiveruid + 1;
                atomic {
                        receiverchan ! SYN_ACK, senderuid, receiveruid;
                        senderState = ESTABLISHED;
                }
        }
```
Once at state SYN_RCV the sender will check if the sequence number received from the receiver was set correctly. If not, sender moves to state CLOSED and try to reconnect. Otherwise, it increments its `receiveruid` sequence number, and sends back a proper SYN_ACK with the receiver’s sequence number incremented by 1. Then, it goes into the ESTABLISHED section.

ESTABLISHED:
```
l_ESTABLISHED: {
        /* Start sending messages */
        messagechan ! senderuid, message;
        if
        :: senderchan ? MSG_ACK, temp, 0 ->
                if
                :: temp == senderuid ->
                        /* successfully received */
                        senderuid = senderuid + 1;
                        payloadHashSent = payloadHashSent * hashmod + message % hashmod;
                        nummessages = nummessages + 1;
                        if
                        :: nummessages < maxmessages ->
                                /* send another message... */
                                message = message + 1;
                        :: true -> /* decided to close connection */
                                goto l_CLOSE;
                        fi;
                :: temp < senderuid ->
                        /* received old msg, so we retransmit */
                :: true ->
                        /* message dropped by the network */
                fi;
        :: empty(senderchan) -> /* We didn't get response, so timeout */
                /* Try to resend the message */
        fi;
        goto l_ESTABLISHED;
}
```

The ESTABLISHED state is the most complex of the sender's states.
On this section, we use non-deterministic tools provided by Promela/spin to cover all branches of the decision making. Initially, the sender sends a message to receiver using the dedicated data channel `messagechan` and then waits for a MSG_ACK or timeout. If the sender times out, it means that it never received an MSG_ACK and it should try to resend the last message. But if it receives an MSG_ACK, it verifies the value `senderuid`, or simulates a message dropped by the network. If the `senderuid` received is less then the current `senderuid`, we have to retransmit the message because it is an acknowledgment for an old message. Otherwise, it increments `senderuid` and `message`. If we have reached the maximum number of messages, it moves to the l_CLOSE section. If not, it may decide to send another message, or it may decide to move to the l_CLOSE section.

CLOSE:

```
        l_CLOSE: {
                atomic {
                        receiverchan ! FIN_ACK, senderuid, receiveruid;
                        senderState = FIN_WAIT_1;
                }
        }
```

This section starts the teardown phase of the connection. The sender send a FIN_ACK (FIN acknowledgment) to the receiver and moves to state FIN_WAIT_1.

FIN_WAIT_1:

```
        l_FIN_WAIT_1: {
                do
                :: senderchan ? MSG_ACK, _, _ ->
                        printf("[S] Received a retransmitted MSG_ACK. Ignoring, because
we
                        are trying to close the connection here.\n");
                :: senderchan ? FIN_ACK, receiveruid, senderuid ->
                        printf("[S] Received FIN_ACK from receiver\n");
                        goto l_FIN_WAIT;
                od;
        }
```

At this state, the sender waits for a FIN_ACK response from the receiver acknowledging the closing of the connection. It might still receives some leftover MSG_ACK's, in which case it flushes them out. After FIN_ACK received it can move to the termination section.

Termination:

```
        l_FIN_WAIT: {
                atomic {
                        receiverchan ! ACK, senderuid, receiveruid;
                        printf("[S] --- Sender closed ---\n");
                        if
                        :: totalconnections < connections ->
                                totalconnections = totalconnections + 1;
                                goto l_CLOSED;
                        :: else ->
                                printf("[S] Sender reached max connections; process
                                 terminating.\n");
                        fi;
                }
        }
        senderState = TERMINATED;
}
```

On this section the sender sends a final ACK to receiver, just to acknowledge the arrival of the FIN_ACK message. Then it moves to CLOSED state. If `totalconnections` hasn't reached its threshold, it will start the protocol again, otherwise it will terminate.

Receiver

The receiver implementation is all in receiver.pml, with some state variables defined in tcp.pml. Each state is preceded by a label (prefixed with `l_`) and enclosed in brackets.

This section documents each part of the implementation. It doesn't include any mutant-inserted code, however.

Initialization:

```
int receiver_totalconnections = 0;

active proctype Receiver()
{
	int receiveruid = 0, senderuid, message, temp, last_received;
	...
```

The receiver is initialized, declaring its state variables:

`receiver_totalconnections` will contain the number of connection attempts

`senderuid` will contain the sequence number of the sender, received in the initial SYN message when the handshake starts

`receiveruid` will contain the ID used for the receiver acknowledgement number during the 3-way handshake

`message` will hold the payload of incoming messages

`temp` will be used as variable to hold various things from incoming messages

`last_received` contains the largest ID number of all messages we received, thus the last packet sent by the sender that we successfully received.

CLOSED:

```
	l_CLOSED: {
		receiverState = CLOSED;
		last_received = 0;
	}
```

The receiver sets its state variable to CLOSED and the `last_received` value to 0, because it hasn't seen any message yet. Then, it goes into the LISTEN section.

LISTEN:

```
	l_LISTEN: {
		receiverState = LISTEN;
		atomic {
			receiverchan ? SYN, senderuid, temp; /* Wait for SYN */
			printf("[R] Received SYN\n");
			receiveruid = receiveruid + 1;
			/* increment sequence number */
```

```
                senderchan ! SYN_ACK, receiveruid, senderuid + 1;
                /* Send back SYN+ACK */
                printf("[R] Sent SYN+ACK\n");
            }
        }
```
The receiver sets its state variable to LISTEN and waits for a SYN message to arrive. Once it does, it increments its own sequence number (this is done only so that it is different from previous handshakes; the protocol doesn't require it), and sends back a proper SYN_ACK with the sender's sequence number incremented by 1. Then, it goes into the SYN_RCVD section.

SYN_RCVD:
```
    l_SYN_RCVD: {
            atomic {
                receiverState = SYN_RCVD;
                receiverchan ? SYN_ACK, senderuid, temp;
            }
            printf("[R] Received SYN+ACK\n");
            if
            :: temp != receiveruid + 1 ->
                printf("[R] Sequence number (receiveruid) send by sender doesn't
                match the expected value! Closing.\n");
                goto l_CLOSE_WAIT;
            :: else -> skip;
            fi;
            receiveruid = temp;
    }
```
The receiver sets its state variable to SYN_RCVD and waits for the third message of the 3-way handshake (SYN+ACK from the sender). Once it is received, it checks whether the sequence number from the message matches the expected ID. If that it not the case, the receiver jumps to the CLOSE_WAIT state. If it is the case, the receiver updates its sequence number variable and moves on to the ESTABLISHED state.

ESTABLISHED:
```
    l_ESTABLISHED: {
            receiverState = ESTABLISHED;
            do
            :: messagechan ? temp, message -> /* Receive a message */
                printf("[R] Message #%d received with payload \"%d\"\n",
                temp, message);
                if
                :: temp < last_received ->
                    printf("[R] Message #%d was already received
                    (last = %d), so ignore. It was probably a
                    retransmission.\n", message, last_received);
                    :: temp >= last_received ->
                            /* Send ACK that message was received */
                            if
                            :: temp == last_received ->
                                    printf("[R] Message #%d is the same as the one
                                    last received. Our acknowledgement probably got
                                    ignored. Acknowledging again.\n", temp);
                            :: temp > last_received ->
                                    printf("[R] Message #%d is a new message.
                                    Acknowledging.\n", temp);
                                    printf("[R] [Payload] Receiver commits to
                                    payload \"%d\". Payload hash: %d -> %d\n",
                                    message, payloadHashReceived,
                                    payloadHashReceived * hashmod + message %
                                    hashmod);
                                    payloadHashReceived = payloadHashReceived *
                                    hashmod + message % hashmod;
                                    last_received = temp;
                            fi;
                            senderchan ! MSG_ACK, last_received, 0;
                            printf("[R] Sent MSG_ACK for message #%d\n",
                            last_received);
                    :: true -> /* Possible timeout */
                            printf("[R] Receiver pretending we didn't see the
                            message, waiting for retransmission.\n");
                    fi;
                    goto l_ESTABLISHED;
            :: receiverchan ? FIN_ACK, senderuid, receiveruid ->
                    printf("[R] Received FIN_ACK from sender\n");
                    goto l_CLOSE_WAIT; /* change state because we received
                    a FIN */
            od;
    }
```

The ESTABLISHED state is the most complex of the receiver states.
First, the receiver sets its state to ESTABLISHED. Then, it waits on two possible events:

Either we get a FIN_ACK on the receiver channel, in which case we jump to the CLOSE_WAIT state because the sender wishes to close the connection

Either we get a message on the message channel.

Note that it is entirely possible for both of those actions to be executable at the same time, in which case the model checker nondeterministically processes them (ie it tests both cases).

In case we get a message on the message channel, we grab its ID in the variable `temp`, and its payload in the variable `message`.

Then, one of three things may happen:

The receiver may choose to ignore the message completely, to simulate packet loss. In this case, the receiver goes back to the beginning of the ESTABLISHED section.

If the receiver doesn't ignore the message, it checks if its ID (held in the `temp` variable) is smaller, equal to, or larger than the largest message ID it has ever seen in this connection attempt.

- If the message ID is larger than the largest message ID it has ever seen in this connection attempt, it treats the message as a new message in the transmission. It accepts its payload as valid (commit), updates the hash of the data transmitted so far, updates the largest message ID received (`last_received`), and sends an acknowledgement of the new message.
- If the message ID is the same as the largest message ID it has ever seen in this connection attempt, then it is possible that the sender didn't receive the acknowledgement for that message ID that we sent before, either because it didn't get to it yet, either because it ignored it (simulating packet loss). Either way, we resend an acknowledgement for that message. The sender may or may not need it, so we send it to be safe.
- If the message ID is lower than the largest message ID it has ever seen in this connection attempt, then it means that we got a message from the past somehow. We simply ignore the message in this case.

Unlike the sender, the receiver doesn't have a limit as to how many messages it can receive. Indeed, since it relies on the sender sending messages in the first place, it would be unnecessary to put a cap on the number of messages of the receiver as well. Additionally, since the sender doesn't have to send exactly the maximum number of messages it is allowed to send (it can decide to stop sending messages at any time), the receiver has to rely on the sender to tell it when the message stream ends. This is done through the FIN_ACK message. Once received, the receiver goes to the CLOSE_WAIT section.

CLOSE_WAIT:

```
l_CLOSE_WAIT: {
        receiverState = CLOSE_WAIT;
        printf("[R] Clearing out leftover messages in message channel
        (if any)...\n");
        do
        :: messagechan ? temp, message ->
                printf("[R] Received leftover message #%d from sender, with
                payload \"%d\"\n", temp, message);
        :: empty(messagechan) ->
                break;
        od;
        senderchan ! FIN_ACK, receiveruid, senderuid;
        printf("[R] Sent FIN_ACK to sender\n");
        receiverchan ? ACK, senderuid, receiveruid;
        printf("[R] Received the last ACK\n");
}
```

The receiver sets its state to CLOSE_WAIT. It checks for any leftover message in the message channel. If there is any, it prints information about it but doesn't do anything with the message (doesn't send an acknowledgement). Once the channel is empty, it sends a FIN_ACK message to the sender, and waits for the sender's final ACK message. Once this ACK message is received, it moves on to the LAST_ACK section.

LAST_ACK:

```
	l_LAST_ACK: {
		receiverState = LAST_ACK;
		printf("[R] --- Receiver closed ---\n");
		if
		:: receiver_totalconnections < connections ->
			receiver_totalconnections = receiver_totalconnections + 1;
			goto l_CLOSED;
		:: else ->
			printf("[R] Receiver reached max connections; process
			terminating.\n");
		fi;
	}
```

The receiver sets its state to LAST_ACK. Then, it checks if the receiver has reached the maximum number of connections attempts. If that is not the case, then it increments the number of connection attempts by one, then goes back to the CLOSED section in order to be ready to LISTEN for connections once more. Otherwise, it continues onwards to the termination section.

Termination:

```
	receiverState = TERMINATED;
}
```

The termination section is just one line: the receiver sets its state to TERMINATED. The } character is the closing brace of the Receiver `proctype`.

The TERMINATED state doesn't exist in the real TCP finite state machine. We have added it in our model so that we can use it in our assertions. Indeed, it is often useful to check when processes terminate. It is possible to do so using Promela's `_nr_pr` variable, but that doesn't give information as to which process is alive or not. Moreover, a low `_nr_pr` might either mean that a process has terminated, or hasn't even started yet. Adding a TERMINATED state allows us to clear things up.

Verification

Note: In this section, assertions are shown in their expanded form (i.e. with macros replaced with the real expression behind them). In the .pml files, they use macros for most equality or channel message availability statements.

Miscellaneous assertions

We have one important immediate assertion that can't be categorized into either the sender or the receiver part of the model, because it affects both parts:

- ***IMM_01* - assertions/transmission_integrity.pml**:
  Checks if the data received by the receiver matches what the sender sent. It runs once both the sender and the receiver have terminated, and compares the hashes computed by both sides. If the hashes differ, then there has been a transmission error.
  Unlike the other immediate assertions, which are inserted within the sender or the receiver, this one lives in its own process so that it can run when both the sender and the receiver processes have both terminated.
  Assertion:

```
active proctype integrityCheck() {
   /* Wait for both processes to terminate */
   (senderState == TERMINATED && receiverState == TERMINATED);
   /* Now check that the hashes are equal and nonzero */
   assert payloadHashSent != 0
          && payloadHashSent == payloadHashReceived;
}
```

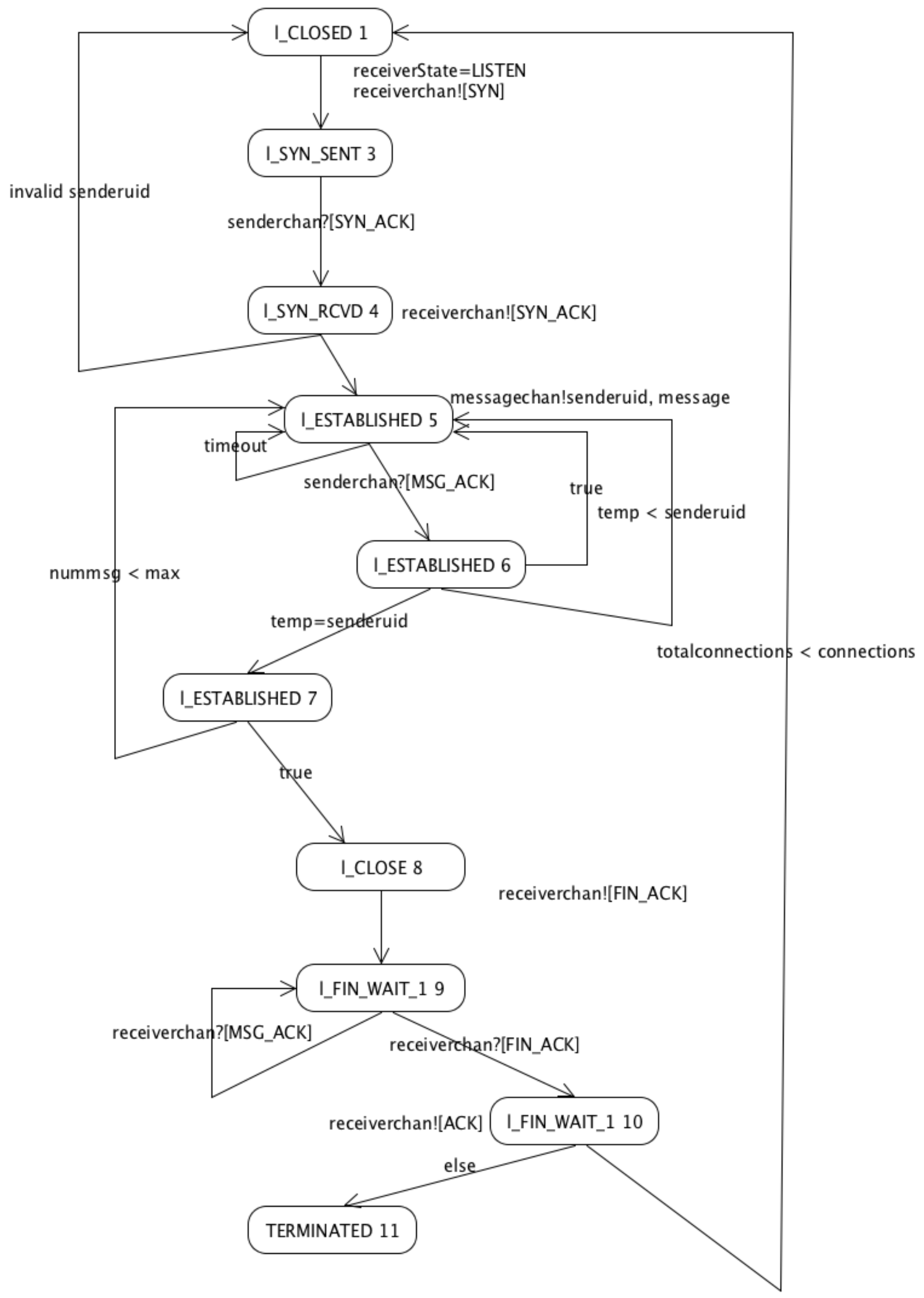
Equivalence check

In this section, we will demonstrate that our implementation matches the specification. To do this, we have built the control flow graph for each process (sender and receiver). For each process, we take the FSM from the specification, and its control flow graph. Then, we run a series of tests. Each test runs on both the FSM and the control flow graph in parallel. Our goal is to show that the output of the system, represented by the output messages sent by the process, are the same as the output messages specified in the FSM. We aim to achieve edge coverage of the control flow graph for both processes.

In this section, in order to keep test vectors small, we will assume that the maximum number of connections attempts allowed in our model is 1. In the implementation, this is defined in a constant and can be changed at will. As such, the sender and receiver will not go back to the **CLOSED** state after the first connection attempt ends. Instead, they will go to the **TERMINATED** state.

Sender

Control flow graph:



Test vectors:

Test 1:

- *Test vector*:
  `receiverState == LISTEN, senderchan?[SYN_ACK], senderuid = 0`

- *State transitions on FSM*:
  `CLOSED, SYN_SENT, SYN_RCVD, CLOSED`
- *Output messages on FSM*:
  `SYN`
- *Node traversed on control flow graph*:
  `l_CLOSED 1,  l_SYN_SENT 3,  l_SYN_RCVD, l_CLOSED 1`
- *Output messages on control flow graph*:
  `SYN`
- As we can see, the output messages of the FSM match those of the control flow graph.

Test 2:

- *Test vector*:
  `receiverState == LISTEN, senderchan?[`**`SYN_ACK`**`],`
  `senderuid = senderuid + 1, senderchan?[`**`MSG_ACK`**`], temp < senderuid`
- *State transitions on FSM*:
  `CLOSED, SYN_SENT,SYN_RCVD, ESTABLISHED`
- *Output messages on FSM*:
  `SYN, SYN_ACK`
- *Node traversed on control flow graph*:
  `l_CLOSED 1,  l_SYN_SENT 3,  l_SYN_RCVD 4,`
  `l_ESTABLISHED 5, l_ESTABLISHED 6`
- *Output messages on control flow graph*:
  `SYN, SYN_ACK`
- As we can see, the output messages of the FSM match those of the control flow graph.

Test 3:

- *Test vector*:
  `receiverState == LISTEN, senderchan?[`**`SYN_ACK`**`],`
  `senderuid = senderuid + 1, senderchan?[`**`MSG_ACK`**`],`
  `temp = senderuid, nummsg < max`
- *State transitions on FSM*:
  `CLOSED, SYN_SENT,SYN_RCVD, ESTABLISHED`
- *Output messages on FSM*:
  `SYN, SYN_ACK`
- *Node traversed on control flow graph*:
  `l_CLOSED 1,  l_SYN_SENT 3,  l_SYN_RCVD 4,`
  `l_ESTABLISHED 5, l_ESTABLISHED 6, l_ESTABLISHED 7`
- *Output messages on control flow graph*:
  `SYN, SYN_ACK`
- As we can see, the output messages of the FSM match those of the control flow graph.

Test 4:

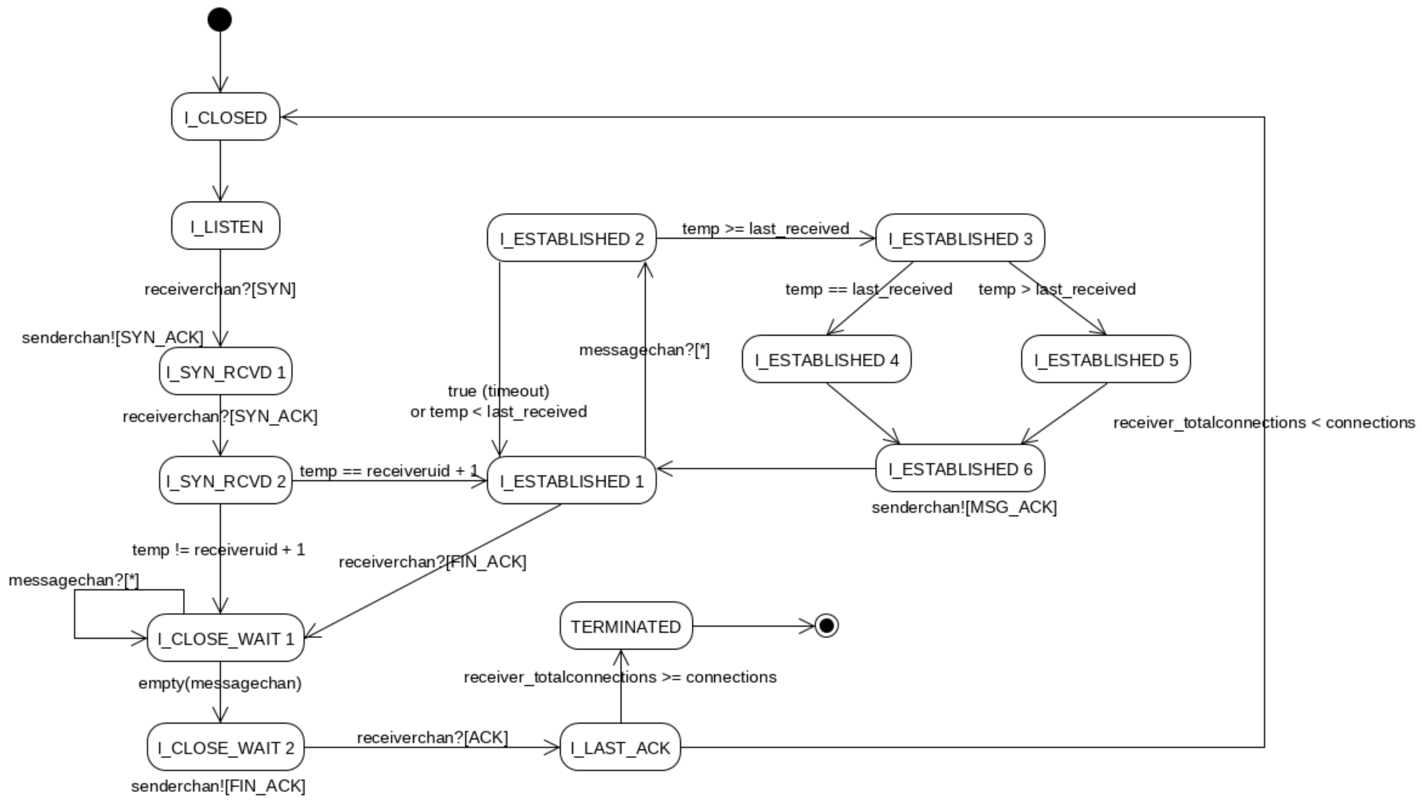
- *Test vector*:
  `receiverState == LISTEN, senderchan?[`**`SYN_ACK`**`],`
  `senderuid = senderuid + 1, senderchan?[`**`MSG_ACK`**`], temp = senderuid,`
  `nummsg >= max, receiverchan?[`**`FIN_ACK`**`],`
  `totalconnections < connections`
- *State transitions on FSM*:
  `CLOSED, SYN_SENT,SYN_RCVD, ESTABLISHED, FIN_WAIT_1, CLOSED`
- *Output messages on FSM*:
  `SYN, SYN_ACK, FIN_ACK`
- *Node traversed on control flow graph*:
  `l_CLOSED 1,  l_SYN_SENT 3,  l_SYN_RCVD 4, l_ESTABLISHED 5,`
  `l_ESTABLISHED 6, l_ESTABLISHED 7, l_CLOSE 8,`
  `l_FIN_WAIT_1 10, l_CLOSED 1`
- *Output messages on control flow graph*:
  `SYN, SYN_ACK, FIN_ACK`
- As we can see, the output messages of the FSM match those of the control flow graph.

Test 5:

- *Test vector*:
  `receiverState == LISTEN, senderchan?[`**`SYN_ACK`**`],`
  `senderuid = senderuid + 1, senderchan?[`**`MSG_ACK`**`],`
  `temp = senderuid, nummsg >= max, receiverchan?[`**`FIN_ACK`**`],`
  `totalconnections >= connections`
- *State transitions on FSM*:
  `CLOSED, SYN_SENT, SYN_RCVD, ESTABLISHED, FIN_WAIT_1, TERMINATED`
- *Output messages on FSM*:
  `SYN, SYN_ACK, FIN_ACK`
- *Node traversed on control flow graph*:
  `l_CLOSED 1,  l_SYN_SENT 3,  l_SYN_RCVD 4, l_ESTABLISHED 5,`
  `l_ESTABLISHED 6, l_ESTABLISHED 7, l_CLOSE 8,`
  `l_FIN_WAIT_1 10, TERMINATED 11`
- *Output messages on control flow graph*:
  `SYN, SYN_ACK, FIN_ACK`
- As we can see, the output messages of the FSM match those of the control flow graph.

Receiver

Control flow graph:



Test vectors:

Test 1 - Invalid receiveruid:

- *Test vector*:
  `receiverchan?[`**`SYN`**`],`
  `receiverchan?[`**`SYN_ACK`**`] with temp != receiveruid + 1,`
  `receiverchan?[`**`ACK`**`]`
- *State transitions on FSM*:
  `CLOSED, LISTEN, SYN_RCVD, CLOSE_WAIT, TERMINATED`
- *Output messages on FSM*:
  `SYN_ACK, FIN_ACK`
- *Node traversed on control flow graph*:
  `l_CLOSED, l_LISTEN, l_SYN_RCVD 1, l_SYN_RCVD 2, l_CLOSE_WAIT 1,`
  `l_CLOSE_WAIT 2, l_LAST_ACK, TERMINATED`
- *Output messages on control flow graph*:
  `senderchan![`**`SYN_ACK`**`], senderchan![`**`FIN_ACK`**`]`
- As we can see, the output messages of the FSM match those of the control flow graph for this test vector.

Test 2 - One message sent:

- *Test vector*:
  `receiverchan?[`**`SYN`**`],`
  `receiverchan?[`**`SYN_ACK`**`] with temp == receiveruid + 1,`
  `messagechan?[1,3], receiverchan?[`**`FIN_ACK`**`], receiverchan?[`**`ACK`**`]`
- ***Case 1: Acknowledgement sent***:

  State transitions on FSM:
  `CLOSED, LISTEN, SYN_RCVD, ESTABLISHED, ESTABLISHED,`
  `CLOSE_WAIT, TERMINATED`
  Output messages on FSM:
  `SYN_ACK, MSG_ACK, FIN_ACK`
  Node traversed on control flow graph:
  `l_CLOSED, l_LISTEN, l_SYN_RCVD 1, l_SYN_RCVD 2,`
  `l_ESTABLISHED 1, l_ESTABLISHED 2, l_ESTABLISHED 3,`
  `l_ESTABLISHED 5, l_ESTABLISHED 6, l_ESTABLISHED 1,`
  `l_CLOSE_WAIT 1, l_CLOSE_WAIT 2, l_LAST_ACK, TERMINATED`
  Output messages on control flow graph:
  `senderchan![`**`SYN_ACK`**`], senderchan![`**`MSG_ACK`**`], senderchan!`
  `[`**`FIN_ACK`**`]`
- ***Case 2: Message dropped***:

  State transitions on FSM:
  `CLOSED, LISTEN, SYN_RCVD, ESTABLISHED, ESTABLISHED,`
  `CLOSE_WAIT, TERMINATED`
  Output messages on FSM:
  `SYN_ACK, FIN_ACK`
  Node traversed on control flow graph:
  `l_CLOSED, l_LISTEN, l_SYN_RCVD 1, l_SYN_RCVD 2,`
  `l_ESTABLISHED 1, l_ESTABLISHED 2, l_ESTABLISHED 1,`
  `l_CLOSE_WAIT 1, l_CLOSE_WAIT 2, l_LAST_ACK, TERMINATED`
  Output messages on control flow graph:
  `senderchan![`**`SYN_ACK`**`], senderchan![`**`FIN_ACK`**`]`
- ***Case 3: Chose to process FIN_ACK before processing message***:

  Note: For this choice to happen, the messages `messagechan?[1,3]` and `receiverchan?[`**`FIN_ACK`**`]` need to be sent both before the receiver has the chance to act on either of them.

  State transitions on FSM:
  `CLOSED, LISTEN, SYN_RCVD, ESTABLISHED, CLOSE_WAIT,`
  `CLOSE_WAIT, TERMINATED`
  Output messages on FSM:
  `SYN_ACK, FIN_ACK`
  Node traversed on control flow graph:
  `l_CLOSED, l_LISTEN, l_SYN_RCVD 1, l_SYN_RCVD 2,`

`l_ESTABLISHED 1, l_CLOSE_WAIT 1, l_CLOSE_WAIT 1,`
`l_CLOSE_WAIT 2, l_LAST_ACK, TERMINATED`
Output messages on control flow graph:
`senderchan![`**`SYN_ACK`**`], senderchan![`**`FIN_ACK`**`]`

- As we can see, the output messages of the FSM match those of the control flow graph for this test vector, no matter the case chosen.

Test 3 - One message sent, with retransmission:

- *Test vector*:
  `receiverchan?[`**`SYN`**`],`
  `receiverchan?[`**`SYN_ACK`**`] with temp == receiveruid + 1,`
  `messagechan?[1,3], messagechan?[1,3], receiverchan?[`**`FIN_ACK`**`],`
  `receiverchan?[`**`ACK`**`]`
- ***Case 1: Acknowledgement resent***:
  State transitions on FSM:
  `CLOSED, LISTEN, SYN_RCVD, ESTABLISHED, ESTABLISHED,`
  `ESTABLISHED, CLOSE_WAIT, TERMINATED`
  Output messages on FSM:
  `SYN_ACK, MSG_ACK, MSG_ACK, FIN_ACK`
  Node traversed on control flow graph:
  `l_CLOSED, l_LISTEN, l_SYN_RCVD 1, l_SYN_RCVD 2,`
  `l_ESTABLISHED 1, l_ESTABLISHED 2, l_ESTABLISHED 3,`
  `l_ESTABLISHED 5, l_ESTABLISHED 6, l_ESTABLISHED 1,`
  `l_ESTABLISHED 2, l_ESTABLISHED 3, l_ESTABLISHED 4,`
  `l_ESTABLISHED 6, l_ESTABLISHED 1, l_CLOSE_WAIT 1,`
  `l_CLOSE_WAIT 2, l_LAST_ACK, TERMINATED`
  Output messages on control flow graph:
  `senderchan![`**`SYN_ACK`**`], senderchan![`**`MSG_ACK`**`], senderchan!`
  `[`**`MSG_ACK`**`], senderchan![`**`FIN_ACK`**`]`
- There exists other cases, but they are not necessary in order to achieve full edge coverage.
- As we can see, the output messages of the FSM match those of the control flow graph for this test vector in the chosen case. We believe other cases would achieve similar results, although edge coverage doesn't require these to be tested as well.

Verification - Sender

Immediate assertions

We have one immediate assertion for the sender:

***IMM_02* - assertions/sender/imm_wrong_receiveruid.pml**
Check if receiveruid was correctly set.
Assertion:
```
assert (temp == receiveruid)
```

LTL Assertions

We have the following LTL assertions for the sender:

***LTL_01* - assertions/sender/established_until_finack.pml**
Check if the sender stays on state **ESTABLISHED** until it sends a **FIN_ACK** message to receiver.
Assertion:
```
[](senderState == ESTABLISHED ->
      (senderState == ESTABLISHED
     U receiverchan ? [FIN_ACK])
)
```

- ***LTL_02* - assertions/sender/received_synack_to_synrcvd.pml**
  Check if the sender moves to state **SYN_RCVD** when received **SYN_ACK**.
  Assertion:
  ```
  [](senderchan ? [SYN_ACK] -> senderState == SYN_RCVD)
  ```

  ***LTL_03* - assertions/sender/sent_ack_established.pml**
  Check if the sender moves to state **ESTABLISHED** after sending **ACK** message.
  Assertion:
  ```
  [](receiverchan ? [SYN_ACK] -> X senderState == ESTABLISHED)
  ```

- ***LTL_04* - assertions/sender/sent_finack_to_finwait.pml**
  Check if the sender moves to state **FIN_WAIT_1** after sending **FIN_ACK** message
  Assertion:
  ```
  [](receiverchan ? [FIN_ACK] -> X senderState == FIN_WAIT_1)
  ```

- ***LTL_05* - assertions/sender/sent_finalack_to_closed.pml**
  Check if the sender moves from state **FIN_ACK** to state **CLOSED** properly
  Assertion:
  ```
  [](receiverchan ? [ACK] -> X senderState == CLOSED)
  ```

  ***LTL_06* - assertions/sender/sent_syn_to_synsent.pml**

Check if the sender moves to state **SYN_SENT** after sending **SYN**
Assertion:
`[](receiverchan ? [SYN] -> X senderState == SYN_SENT)`

- ***LTL_07*** - **assertions/sender/synsent_until_synack.pml**
  Check if the receiver stay on state **SYN_SENT** until receives an **SYN_ACK**
  Assertion:
  `[](senderState == SYN_SENT -> (senderState == SYN_SENT U senderchan ? [SYN_ACK]))`

- ***LTL_08*** - **assertions/sender/established_eventually_finwait.pml**
  Check if the sender eventually moves from state **ESTABLISHED** to state **FIN_WAIT_1** properly.
  Assertion:
  `[](senderState == ESTABLISHED -> <> senderState == FIN_WAIT_1)`

Mutants

The following mutants were created for the sender:

MUTANT_01 - **mutants/sender/established_before_synack.pml** (inserted in sender.pml at line 33):
Sets state to **ESTABLISHED** before receiving **SYN_ACK**. The correct behaviour should set state to **ESTABLISHED** after receiving **SYN_ACK**
MUTANT_01* was caught by assertion *LTL_07

MUTANT_02 - **mutants/sender/set_wrong_state.pml** (inserted in sender.pml at line 72):
Sets state to **CLOSED** in the **ESTABLISHED** block. The correct behaviour should set state to **CLOSED** only after sending **ACK**.
MUTANT_02* was caught by assertion *LTL_01

MUTANT_03 - **mutants/sender/wrong_ack_reiceiveruid.pml** (inserted in sender.pml at line 58):
Sets returning receiveruid incorrectly. The correct behaviour should increment the receiveruid.
MUTANT_03* was caught by assertion *IMM_02

MUTANT_04 - **mutants/sender/wrong_finack_state.pml** (inserted in sender.pml at line 111):
Sets wrong state after sending **FIN_ACK** to receiver. The correct behaviour should set state to **FIN_WAIT_1**.
MUTANT_04* was caught by assertion *LTL_04

***MUTANT_05* - mutants/sender/wrong_synack_senderuid.pml** (inserted in sender.pml at line 46):
Sets returning senderuid incorrectly. The correct behaviour should increment the senderuid.
MUTANT_04* was caught by assertion *IMM_01

Verification - Receiver

Immediate assertions

We have one immediate assertion for the receiver:

***IMM_03* - assertions/receiver/imm_fin_ack_channel_empty.pml**:
Checks if the receiver channel is empty once we receive a **FIN_ACK**.
Assertion:

```
assert empty(receiverchan);
```

Inserted in receiver.pml in the **ESTABLISHED** section (token `IMM_RECEIVER_FIN_ACK_CHANNEL_EMPTY`).

LTL Assertions

We have the following LTL assertions for the receiver:

***LTL_09* - assertions/receiver/closed_listen.pml**:
Checks if the receiver moves from state **CLOSED** to state **LISTEN** properly.
Assertion:

```
[](receiverState == CLOSED ->
    (receiverState == CLOSED U receiverState == LISTEN)
)
```

***LTL_10* - assertions/receiver/closewait_lastack.pml**:
Checks if the receiver moves from state **CLOSE_WAIT** to state **LAST_ACK** properly.
Assertion:

```
[](receiverState == CLOSE_WAIT ->
    (receiverState == CLOSE_WAIT U receiverState == LAST_ACK)
)
```

***LTL_11* - assertions/receiver/closewait_sender_ended.pml**:
The receiver can only be in state **CLOSE_WAIT** if the sender is in **FIN_WAIT_1** or later.
Assertion:

```
[](receiverState == CLOSE_WAIT ->
    (senderState == FIN_WAIT_1 || senderState == CLOSED
                               || senderState == TERMINATED)
)
```

LTL_12 **- assertions/receiver/connection_start.pml**:
The receiver can only be **CLOSED** if the sender is not actively maintaining a connection.
Assertion:

```
[](receiverState == CLOSED ->
    !(senderState == ESTABLISHED || senderState == SYN_RCVD)
)
```

LTL_13 **- assertions/receiver/established_closewait.pml**:
Check if the receiver moves from state **ESTABLISHED** to **CLOSE_WAIT** properly.
Assertion:

```
[](receiverchan ? [FIN_ACK] ->
    (
        (receiverchan ? [FIN_ACK] || receiverState == ESTABLISHED)
       U receiverState == CLOSE_WAIT
    )
)
```

LTL_14 **- assertions/receiver/lastack_end.pml**:
Checks if the receiver moves from **LAST_ACK** to either **CLOSED**, or exits if it has reached the maximum number of connections.
Assertion:

```
[](receiverState == LAST_ACK ->
    (
        receiverState == LAST_ACK
     U (receiverState == CLOSED || (
            receiverState == TERMINATED &&
            receiver_totalconnections >= connections
       )
    )
)
```

LTL_15 **- assertions/receiver/listen_synrcvd.pml**:
Checks if the receiver moves from state **LISTEN** to state **SYN_RCVD** properly.
Assertion:

```
[](receiverchan ? [SYN] ->
    (receiverState == LISTEN U receiverState == SYN_RCVD)
)
```

LTL_16 **- assertions/receiver/output_finack.pml**:
The receiver may only send **FIN_ACK** messages while in the **CLOSE_WAIT** state.
Assertion:

```
[](senderchan ? [FIN_ACK] -> receiverState == CLOSE_WAIT)
```

LTL_17 **- assertions/receiver/output_msgack.pml**:
The receiver may only send **MSG_ACK** messages in the **ESTABLISHED** state, though the sender may not have received them by the time the receiver switches to the **CLOSE_WAIT** state.
Assertion:
```
[](senderchan ? [MSG_ACK] ->
    (receiverState == ESTABLISHED || receiverState == CLOSE_WAIT)
)
```

LTL_17 **- assertions/receiver/output_synack.pml**:
The receiver may only send **SYN_ACK** messages in the **LISTEN** state, though the sender may not have received them by the time the receiver switches to the **SYN_RCVD** state.
Assertion:
```
[](senderchan ? [SYN_ACK] ->
    (receiverState == LISTEN || receiverState == SYN_RCVD)
)
```

LTL_18 **- assertions/receiver/synrcvd_established_or_closewait.pml**:
Check if the receiver moves from state **SYN_RCVD** to either **ESTABLISHED** or **CLOSE_WAIT** upon receiving a **SYN_ACK** message.
Assertion:
```
[](receiverchan ? [SYN_ACK] ->
    (
        (receiverchan ? [SYN_ACK] || receiverState == SYN_RCVD)
      U (receiverState == ESTABLISHED || receiverState == CLOSE_WAIT)
    )
)
```

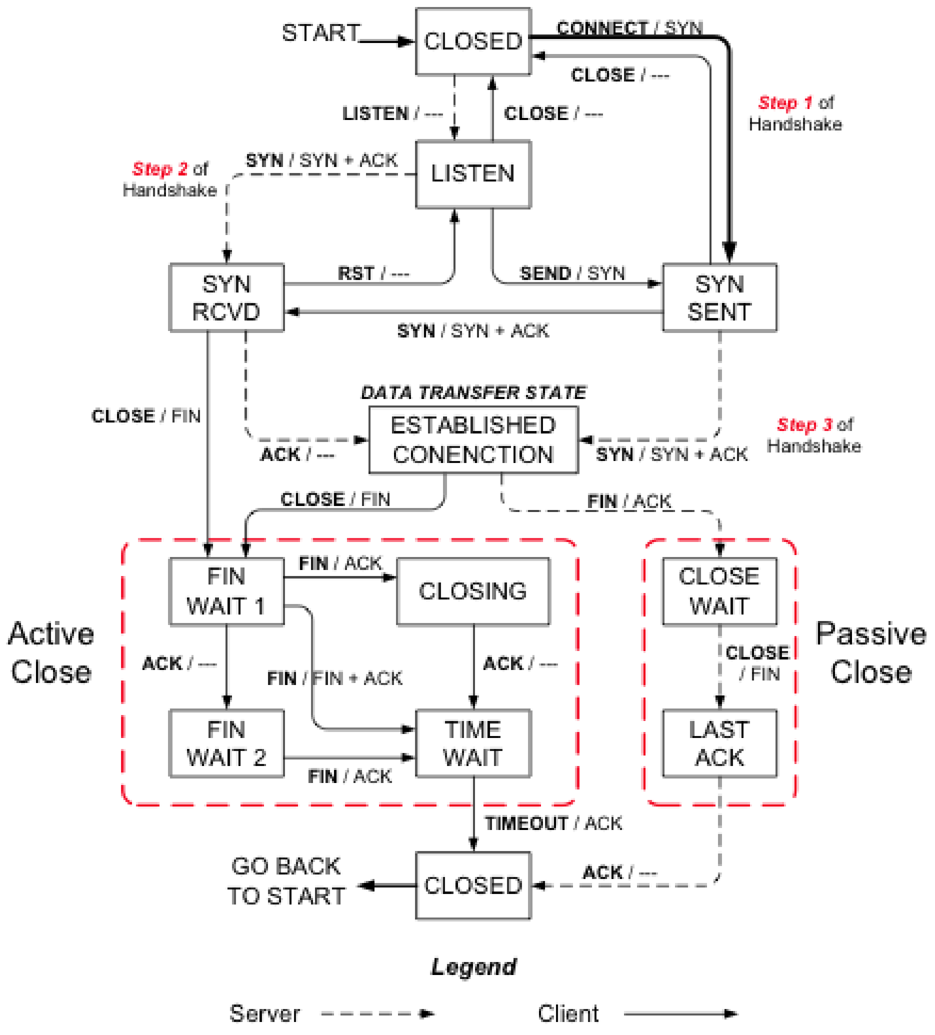
Mutants

The following mutants were created for the receiver:

MUTANT_06 **- mutants/receiver/corrupt_payload.pml** (inserted in receiver.pml at line 68):
Makes the receiver modify the payload received on all messages on the message channel to some other value, thereby corrupting the payload on the receiver side. The correct behavior would be to not modify the value of the received payload.
MUTANT_06 was caught by assertion IMM_01
MUTANT_07 **- mutants/receiver/dont_close.pml** (inserted in receiver.pml at line 101):
Makes the receiver jump to the **SYN_RCVD** state once it gets a **FIN_ACK** message in the **ESTABLISHED** state. The correct behavior in those circumstances would be to jump to the **CLOSE_WAIT** state.
MUTANT_07 was caught by assertion LTL_13
MUTANT_08 **- mutants/receiver/fin_ack_wrong_guard.pml** (inserted in receiver.pml at line 92):
Allows the receiver to jump from the **ESTABLISHED** to the **CLOSE_WAIT** state without

receiving a **FIN_ACK** message. The correct behavior should be to only jump to **CLOSE_WAIT** when the receiver receives a **FIN_ACK** message.
MUTANT_08 was caught by assertion IMM_03
MUTANT_09 - **mutants/receiver/invalid_msg_ack.pml** (inserted in receiver.pml at line 40):
Makes the receiver send a **MSG_ACK** message in the **LISTEN** state. The correct behavior is not to send any message while in the **LISTEN** state. **MSG_ACK** messages should only be sent during the **ESTABLISHED** state.
MUTANT_09 was caught by assertion LTL_17
MUTANT_10 - **mutants/receiver/set_wrong_state.pml** (inserted in receiver.pml at line 124):
Makes the receiver set its state variable to **CLOSED** in the **LAST_ACK** section. The correct behavior would be to set its state variable to **LAST_ACK**.
MUTANT_10 was caught by assertion LTL_10

Appendix



Original FSM for TCP 3 way-handshaking connection Setup and Tear Down

STATE NAME	DESCRIPTION
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED CONNECTION	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIMED WAIT	Wait for all packets to terminate
CLOSING	Both sides have tried to close the connection simulatenously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to terminate

Bibliography

Holzmann, Gerard J.. *Design and validation of computer protocols*. Englewood Cliffs, N.J.: Prentice Hall, 1991. Print.

Ammann, Paul, and Jeff Offutt.*Introduction to software testing*. New York: Cambridge University Press, 2008. Print.

"Spin Online References." *Spin Online References*. N.p., 29 May 2012. Web. 4 Dec. 2012. <http://spinroot.com/spin/Man/>.