Automatic Testing of TCP/IP Implementations Using Quickcheck

Javier Paris
University of A Coruna. Spain
javierparis@udc.es

Thomas Arts
IT University of Gothenburg and Quviq AB. Sweden
thomas.arts@ituniv.se

Abstract

We describe how to use model based testing for testing a network stack. We present a framework that together with the property based testing tool QuickCheck can be used to test the TCP layer of the Internet protocol stack. TCP is a rather difficult protocol to test, since it hides a lot of operations for the user that communicates to the stack via a socket interface. Internally, a lot happens and by only controlling the interface, full testing is not possible. This is typical for more complex protocols and we therefore claim that the presented method can easily be extended to other cases.

We present an automatic test case generator for TCP using Quickcheck. This tester generates packet flows to test specific features of a TCP stack. It then controls the stack under test to run the test by using the interface provided by it (for example, the socket interface), and by sending replies to the packets created by the stack under test. We validated the test framework on the standard Linux TCP/IP implementation.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing Tools

General Terms Reliability

1. Introduction

One of the problems when developing a TCP/IP stack is testing it. Test cases are not easy to create because of the nature of the environment in which the stack operates: it requires the ability to create a flow of packets that check particular features of TCP/IP, which often involve timing, where there are two actors involved (the local and the remote stack, with complex interactions).

Compliance is usually tested using another stack included in a commercial OS (such as Linux) and a packet sniffer to see the actual packet exchange and manually check if there is any deviation from the standard. There are several problems in this approach:

- Checking a packet dump by hand takes time and it is easy to overlook errors.
- It is hard to check special features which only arise under specific conditions because it is difficult to get the connection into that state. It is even more difficult to test behaviour under failure conditions.
- Because of the dependency on timing and concurrency, stack operations may seem to behave non-deterministically, making it harder to find error cases, but also difficult to reproduce error cases and re-check the stack when the error is fixed.

We introduce a tool for automatically testing TCP implementations through the use of the Quickcheck automatic test case generator. Using Quickcheck we can generate a large number of different test cases with complex behaviour. Specifically, we will focus on two scenarios that we can automatically generate as test cases for the stack. One of them is a simple connection establishment example, and will provide a good introduction. The other example is a complex situation of a simultaneous close which would be difficult to test a manually executed test case.

The rest of the paper is organized as follows. In Sect. 2 we introduce TCP/IP and a couple of test cases we want to test. In Sect. 3 we describe the environment in which the tests are performed. In Sect. 4 we provide a short introduction to Quickcheck, and provide an small example of a test using it. In Sect. 5 we describe in detail how a scenario of the test cases described in Sect. 2 would be. In Sect. 6 we describe some of the problems observed during the design of the tool. Finally, in section 7 we provide our conclusions and explain the future work.

2. TCP in a nutshell

The Internet protocol stack is the defacto standard for Internet communication. It specifies several protocols in five different layers. Each of the lower layers provides services to the upper layer:

- The *physical* layer specifies the electric and physical components of the network: connectors, cables, frequencies (for radio transmission), etc.
- The *link* layer uses the physical network to send packets between directly connected computers. Two well known protocols in this layer are Ethernet or PPP.
- The *network* layer, which uses the *link* layer to send packets between indirectly connected computers, that is, between computers which may be in different but interconnected physical networks. The main protocol in this layer is IP, but there are several others, like ICMP and IGMP.
- The *transport* layer, which uses the *network* layer to control the packet interchange between the two ends of the communication, providing multiplexing, rate control, and packet loss recovery. Typical examples are TCP (Transmission Control Protocol) and UDP, where TCP is the most complex of the two, since it guarantees lossless ordered packet delivery.
- The *application* layer that uses TCP or UDP for transport of the application specific data. Examples are HTTP or IMAP.

Erlang'09, September 5, 2009, Edinburgh, Scotland, UK.

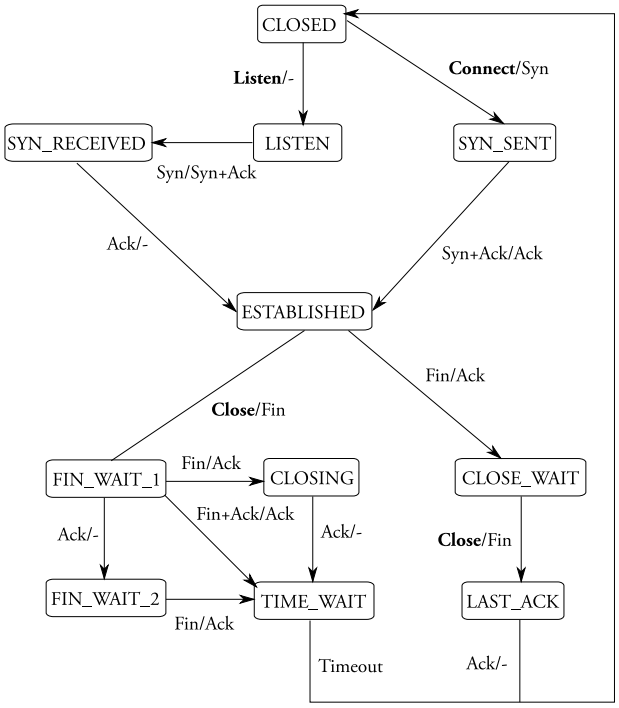

Figure 1. TCP Connection States

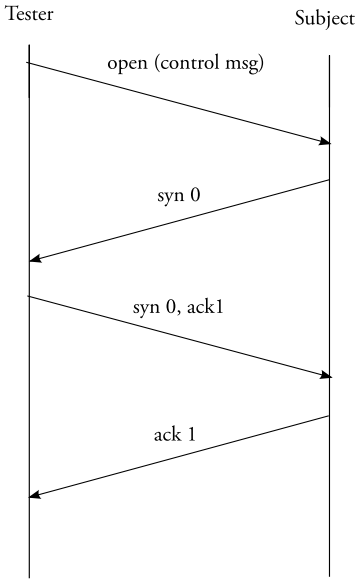

Figure 2. Typical TCP Connection Establishment

The TCP protocol ensures reliable connections on top of unreliable layers. When sending a webpage from one computer to another, TCP divides the page in packages of a certain size and ensures that they are transported to a TCP layer on the other side, which puts the packages together. TCP takes care of resending and assembling, such that the application layer need not to care about that. This requires TCP to have state in order to 'remember' which packets have been sent and together with the high dynamic nature of the protocol, it makes it the most complex one of the four lower layers. TCP connections require synchronization between the peers. All TCP packets have a *s*equence number that identifies the order of the data that is transmitted. This number is used by the peer to order the data as it is received, and to identify lost packets. It is important to note that because a TCP connection is symmetric there is a different sequence number for each of the peers.

TCP packets also include an *acknowledge* number, which reports the last sequence number that the sender of the packet has seen. This is used by the receiver to control the transmission rate and to identify lost packets.

On a high abstraction level, TCP can be modelled by a simple state machine that keeps track of a connection [RFC793 1981]. Of course, many of these connections are concurrently active at the same time. Transitions from one state to another are triggered by timers, the reception of packets or user commands.

At the start of a TCP connection both peers exchange their initial sequence numbers. This process is the connection establishment, and can be seen at the top of Figure 1. A more detailed description of the TCP state machine can be found at [RFC793 1981],[Zaghal and Khan].

When both peers have finished the connection establishment both should be in the ESTABLISHED state, where the actual data transmission takes place.

After the data exchange is over, the connection is closed. This allows both peers to free the resources used to keep the state of the connection. A connection which is closing will transit through the states at the lower part of Figure 1.

Note that the event description in Figure 1 is of the form *Receive/Sent*, that is, when the peer in a certain state receives the *Receive* event, it will send a *Sent* to the other peer and transit to the next state. Also note that the standard interface to TCP, i.e., the socket interface, only gives access to stimulate the events: *Listen*, *Connect*, *Send*, *Receive*, and *Close*. Our approach aims to also control all other events in order to perform proper testing.

Testing TCP

In this paper we describe how we can test TCP by using the above presented state machine as a model for our test cases. We assume all lower layers of the protocol to be correct and check at the end points of communication.

For example, we want to test the transitions caused by connection establishment and closing, including obscure cases such as simultaneous closing[1]. In other words, we want to generate sequences of events that make the TCP implementation move through all possible states in the state machine in all possible ways. We need to create different sequences to get to state ESTABLISHED and different sequences to get to CLOSED again. A typical connection establishment would follow Figure 2:

1. Both Peers start with the connection in the `CLOSED` state.
2. A program in Peer B listens for incoming connections and opens a socket. This puts the connection in the `LISTEN` state.
3. Peer A wants to start a connection and sends a packet with its initial sequence number (0 in our example), and moves to the `SYN_SENT` state.
4. Peer B receives the packet sent by A, and replies with its own initial sequence number (also 0), and acknowledges the *syn* sent by peer A saying that the next sequence number it expects is 1. Peer B moves to the `SYN_RECEIVED` state.

[1] This happens when both peers start a connection close at the same time, instead of doing it sequentially.

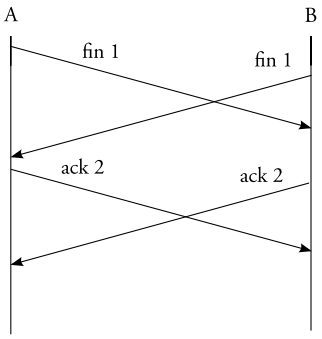

Figure 3. Simultaneous Connection Close

5. Peer A receives the packet sent by B in the previous state, and acknowledges it by sending an *Ack* packet. After this Peer A moves to the `ESTABLISHED` state.
6. Peer B receives the *Ack* and moves to the `ESTABLISHED` state.

This is a simple example, but there are several things that can be checked when this sequence is traversed. The tested stack must generate the correct sequence of packets (in both cases, as starter and as receiver); it must create correct packets (for example, the checksum check should always pass); it also has to keep the state correctly (the sequence and acknowledge numbers of the sequence of packets must be the same).

A rarer scenario is the simultaneous closing of a connection, which happens when both peers try to close the connection at the same time. Figure 3 shows the packet exchange that would happen.

As it can be seen, both peers send a *Fin* packet to start the closing procedure. In this case the connection is simply closed by acknowledging the *Fin* sent by the other peer. While the case is fairly simple to understand, it is difficult to reproduce in a real environment, because it is necessary to perfectly synchronize the connection close in both peers. This requires the tester to have full control over both Peers.

3. Test Setup

TCP connections are usually controlled by the application layer protocols, the most common being the socket interface. These interfaces abstract most of the complex behaviour of TCP, which is good for normal connections, but makes it difficult to analyze if the behaviour of the stack is correct. The tests should not only test the behaviour seen at the interface level, but go deeper.

We also want the test to be as general as possible, that is, not geared towards testing a particular TCP/IP stack. It should be easy to adapt the test specification to check any TCP/IP stack. If no particular structure can be assumed in the tested stack the only way to test it is by doing black box tests.

We have decided to test the behaviour by looking at the actual packets sent over the network. This should provide a reasonable way of doing an in-depth test and providing a test specification that may be used to test very different implementations of TCP. This approach has its limitations. For example, there may be errors that do not show in the packet trace, or are difficult to spot. As an example, an incorrect computation of timers would not be seen in the trace unless it was very serious.

The test setup can be seen in Figure 4, and includes:

1. A computer with the stack under test (from now on **Subject**). This stack needs to be controlled and we use a controller program for doing so. This controller translates the high level commands to execute during the test, such as opening and closing connections, and sending data, to the specific API of Subject. The controller has no state, it just translates the commands coming from QuickCheck [2].
2. A computer running QuickCheck (from now on **Tester**) simulating TCP by means of a QuickCheck specification module and a special IP stack. The IP stack is used to send the TCP packets generated by QuickCheck and to receive incoming packets from the subject. We use a special IP stack, written in Erlang since we want our TCP model to communicate with an IP layer. This would require RAW sockets, but the implementation of them is different for each IP implementation. Thus, using our own IP stack on the Tester side, provides a portable solution for testing. It also allows us in future testing to send incorrect IP packets, which would be hard when using an off-the-shelf IP implementation.
3. A network connecting both computers. Initially both are supposed to be directly connected by an Ethernet network, but this test scenario will be expanded when we add test for cases that require routing. For simplicity we assume that there are no problems in the Ethernet network, that is, no packet loss or delay. These are very rare anyway and we want to control them, not having them spontaneously appear.

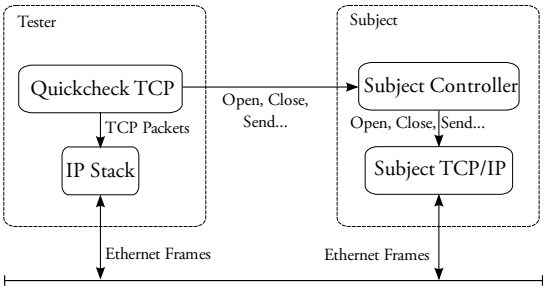

Figure 4. Test Setup

The IP stack is based on the TCP/IP stack described in [Paris et al. 2005]. This processes packets from the network, and creates IP packets to send the TCP packets created by QuickCheck. Because it is implemented in userspace, it uses a packet sniffer to capture packets from the network, and to inject packets into it. This sniffer has been developed using the `libpcap` library[Jacobson et al.] and it supports several operating systems including Linux, Windows, and several BSD variants (such as FreeBSD or Mac OS X).

4. QuickCheck

We base our approach on property-based testing using QuickCheck [Claessen and Hughes 2000], in a commercial version for Erlang developed by Quviq AB [Hughes 2007, Arts et al. 2006]. Compared to the original version of QuickCheck, the commercial version of it has many supporting libraries for protocols, among which a library for finite state machine models that we in particular use.

QuickCheck tests universally quantified *properties*, instead of single test cases. QuickCheck generates random test cases from each property, tests whether the property is true in that case, and reports cases for which the property fails. QuickCheck "shrink" failing test cases automatically, by searching for similar, but smaller

[2] This is a potential weakness of the testing model because Erlang uses TCP for its distribution protocol and we use Erlang to connect to the controller.

test cases that also fail. The result of shrinking is a "minimal"[3] failing case, which often makes the root cause of the problem very easy to find.

This approach has several advantages. Properties are easier to understand than normal test cases, and a property can be used to generate thousands of different test cases. Each test case can also be more complex than a manually written test case.

As an example of Quickcheck we will present a test for IP checksum implementations.

4.1 Testing IP Checksum Implementations with Quickcheck

The IP checksum is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. Ones complement is a way of representing negative numbers.

The IP checksum uses base 16, that is 2 bytes. In 16 bits you can represent the numbers 0 to 65535. The idea with ones' complement is to use half the numbers in this interval for representing negative numbers. Thus, 0 up to 32767 are the positive numbers and 65535 is -0, or an alternative representation of zero. The number 65534 is -1 etc. Until 32768 which is -32767. Hence the interval -32767 up to 32767 can be represented.

4.1.1 Ones Complement

Given an implementation of the IP checksum, a first function we want to test is the ones' complement of a word, say that this is implemented as Erlang function `ip_checksum:ones_complement/1` and that it takes a binary with one 16 bit word as input.

The QuickCheck property that we use to test this function should **not be a re-implementation** of the function under test. This holds for testing in general and also for QuickCheck properties. We do not want to implement a function in order to test it. A way of testing without writing a new version of the ones' complement is to follow the definition and say that a number and its ones' complement sum to -0. Since we want to be general, we specify the property for any base and check it with argument `Base` substituted by 16.

```
prop_ones_complement(Base) ->
 ?FORALL(I,choose(0,(1 bsl Base)-1),
      begin
        CI = ip_checksum:ones_complement(I),
        (1 bsl Base)-1 == I+CI
      end).
```

The QuickCheck macro `?FORALL` is the logic quantifier with 3 arguments, the first one, the variable `I` is bound to a random value generated by the second argument `choose(0,(1 bsl Base)-1)`. This generator randomly chooses a number between zero and the number obtained by shifting a bit `Base` positions to the left and subtracting one, thus between 0 and $2^{Base} - 1$.

The third argument of the `?FORALL` macro is the actual logic expression implemented as an Erlang expression. First the function is applied to the randomly chosen value `I`, which results in `CI`. The sum of `I` and `CI` should be -0 or $2^{Base} - 1$.

This property is checked by QuickCheck by generating random values specified by the generator and evaluating the Erlang expression. If the result is `false`, then an error is detected and smaller values are tested until a minimum example is found for which the property can be falsified.

4.1.2 Padding

It is not clear from the specification presented above, but if you need to compute the checksum of a list of bytes in base 16, then there should be an even number of bytes. Likewise, if we would like to do ones' complement in 32 bits base, we would need to extend a sequence of bytes such that it is divisible by 4.

Extending a bit string such that it is divisible by the base is called padding. We assume there is a function that performs the padding called `ip_checksum:padd/1` taking a bit string as argument and returning a new bit string which is an extended version with as many zero bits as needed.

We like to present a general example that works for padding bitstrings with any number of bits. We assume a generator for bitstrings given in QuickCheck, but for older versions of QuickCheck, one can define a simplistic version oneself for bitstrings up to 200 bits:

```
bitstring() ->
   ?LET(NrBits,choose(0,200),bitstring(NrBits)).

bitstring(NrBits) ->
   ?LET(Bits,vector(NrBits,choose(0,1)),
        to_bitstring(Bits)).

to_bitstring([]) ->
   <<>>;
to_bitstrong([Bit|Bits]) ->
   Rest = to_bitstring(Bits),
   << Bit:1, Rest/bitstring>>.
```

After being able to generate arbitrary bitstrings, we can specify the property for padding arbitrary strings for an arbitrary number of bytes. In the specific case of IP checksum, we would use 16 as base.

```
prop_padding(PadSize) ->
  ?FORALL(BitString,bitstring(),
     begin
       Bits = bit_size(BitString),
       <<B:Bits/bits, Padded/bits>> =
         ip_checksum:padd(BitString),
            Zeros = bit_size(Padded),
       ((Bits+Zeros) rem PadSize) == 0 andalso
              B == BitString andalso
         <<0:Zeros>> == Padded
      end).
```

In this property, the function under test is called with an arbitrary bitstring; it should pad this bitstring for `PadSize` bits. We match the result with a bitlist in which the prefix contains exactly the number of bits that the generated bitstring has and the rest is the padded suffix. We check that the number of bits in the result is divisible by the `PadSize`, that the original bits are unchanged and that the padded bits are all zeros.

4.1.3 Ones Complement Sum

The ones' complement sum is computed by adding a number of words in ones complement. We assume it is implemented by the function `ip_checksum:ones_sum/2` which takes a bitstring as argument. We assume that padding is done outside the `ones_sum` function and only test that the function works for bitstrings of which the length is divisible by the given base.

We do not assume that the `ones_sum/2` function returns the ones' complement of the sum, it just returns the sum. We can compute the sum, add its ones' complement to the exiting list, compute the sum once more and expect to get zero (or actually -0) out of the result. We present a general property and can call it with base 16 in the specific case.

```
prop_ones_sum(Base) ->
   ?FORALL(I,choose(0,1024),
      ?FORALL(Bin,bitstring(I*Base),
         begin
```

[3] In the sense that it cannot shrink to a failing test with the shrinking algorithm used.

```
        Sum = ip_checksum:sum(Bin),
        CSum = ip_checksum:ones_complement(Sum),
    ip_checksum:sum(<<CSum/bits, Bin/bits>>) ==
       <<((1 bsl Base)-1):Base>>
    end)).
```

4.1.4 Checksum

After computing ones' complement sum, one has to take the ones' complement of the result to compute the checksum, as we already did in the above property. For checking the checksum function, we use the same trick as we used for checking the sum function, knowing that if we extend the binary with the checksum and we compute the ones' complement sum we get -0. Alternatively, we could also compute the checksum and do a logical and with the ones' complement sum or so.

```
prop_checksum(Base) ->
   ?FORALL(Bin,bitstring(),
      begin
        CheckSum = ip_checksum:checksum(Bin),
        ip_checksum:sum(<<CheckSum/bits, Bin/bits>>) ==
            <<((1 bsl Base)-1):Base>>
      end).
```

In this way, simple side-effect free functions can be checked by QuickCheck. One uses a, possibly user defined, data generator to create random values in a certain domain. These values are used in a test that reflects a property of the function more than a concrete case.

However, in most Erlang Software, side-effects play an important role and we want to be able to test functions that have side-effects as well. Quviq QuickCheck supports this by libraries that help to specify properties for such software. One of them is the library for finite state machines.

4.2 Quickcheck Finite State Machine Specifications

The TCP protocol is a so called stateful protocol, i.e., specific events may have different effects depending in which state the TCP connection is. If we would perform traditional testing, we would create sequences of events to get TCP into a certain state and consecutive sequences to test certain specifics. With QuickCheck we follow a similar approach, but then quantifying over all possible sequences of events.

Similar to the above described side-effect free case we define a property which in this case states that for all sequences of events the TCP protocol respects the postconditions of these events. The property is rather simple, defining the postconditions and the generator that generates all possible valid sequences is where the work is.

We use the finite state machine library named `eqc_fsm` to specify the state machine given in Sect. 2. That state machine is then used to generate test cases from. Specifying such a state machine is a bit like writing a state machine in `gen_fsm`, but the transitions are non-deterministic and chosen by QuickCheck when generating possible sequences.

In order to specify a state machine, one has to specify for each state which next states it has and which actions or events should be generated to get from one state to the next. Like in `gen_fsm` the state transitions are given as callback functions. One starts by specifying the initial state and initial state data and from there, functions are provided for each state. In addition to this more operational part from the state machine, one also specifies preconditions for events; a precondition is a logic expression specifying whether or not a certain event can take place in a certain state with certain state data. Likewise, one specifies postconditions. A postcondition is a logic expression that specifies whether the return value of a given action or event in a certain state could occur.

In the next section we describe our state machine model in detail.

5. State Machine Model

Our approach is to test the behaviour of a TCP connection with a state machine model. This state machine is modelled using the `eqc_fsm` module of Quickcheck described above.

A generated test simulates the behaviour of a real TCP connection. The current state of the finite state machine models the state in which the connection in the subject should be.

Given the state machine model that we describe in this section, QuickCheck generates a sequence of commands. Either the controller forwards these commands to Subject, which in its turn can result in a TCP packet being sent from the Subject to Tester; or the command provokes the Tester to send a TCP packet to Subject. The result are packages that arrive on the QuickCheck side and the postconditions in the model check that the expected packets have arrived. Replies are generated in advance, based on the expected behaviour.

Note that we test connections, we randomly choose a port, but do all operations on that port during one test. Concurrent connections should be tested differently with a more elaborated state machine model in which we have free and used ports and in which preconditions are used to select free ports when starting a connection.

5.1 Checking Connection Establishment

As an example, we will see how the **tester** checks that the subject correctly establishes a connection (see Figure 2). There are three relevant TCP states:

5.1.1 CLOSED

`CLOSED` is the state in which the connection should be before it is started. This state corresponds to an Erlang function `closed` in the state machine model and from this state, two other states can be reached, the `syn_sent` state and the `syn_rcvd` state.

```
closed(S) ->
[{syn_sent,
   {call, ?MODULE, open,
     [S#state.ip, S#state.port, {var, listener},
      {var, sut}]}},
 {syn_rcvd,
   {call, ?MODULE, listen,
     [S#state.sut_ip, S#state.sut_port, S#state.ip,
      S#state.port, {var, listener}, {var, sut}]}}
].
```

This describes the possible transitions from the `CLOSED` state. The state machine may either proceed to the syn_sent state (by calling the function `open`), or to the syn_receive state (by calling `listen`). Both calls take several parameters from the state of the test (`S`), such as the IP addresses and ports. It also uses two parameter of the test (`listener` and `sut`), which are the symbolic representation of the Erlang process identifiers of the IP stack and the subject controller respectively.

Note that there is no specific state for the `LISTEN` state. One can view the transition between `CLOSED` to `LISTEN` as a silent transition, or a τ step in process algebraic terms. The transition does not transmit any packet on the network.

The `LISTEN` state provides a way of receiving multiple incoming connections to a local port, it is only a way of telling apart ports on which a process is waiting for incoming connections, and ports where no one is listening. A connection is created by either accepting an incoming message on a listen call to a `CLOSED` port or by starting a connection from a `CLOSED` port. Our QuickCheck model

controls the Subject to either do the one or the other in a random way.

In our example, where the subject is starting the connection, the state would proceed to syn_sent. Quickcheck would call the open function, which would:

1. Tell the controller to open a connection.
2. Ask the IP stack to receive a TCP packet from the subject.

This should move the subject state to SYN_SENT. Quickcheck now calls the postcondition function to check if the received TCP packet is correct (a timeout in the listener will raise an exception if no packet is received at all).

```
postcondition(closed, syn_sent, S,
{call, ?MODULE, open, [Ip, Port, _, _]}, Syn) ->
   check_flags(Syn, [syn]) and
     (Syn#tcp.dst_ip==Ip) and
     (Syn#tcp.dst_port==Port) and
     (Syn#tcp.data == <<>>);
```

The postcondition is valid for a transition from CLOSED to SYN_SENT, caused by a call to open(Ip, Port, Listener, Sut), if the message was a Syn message, i.e., the corresponding flag in the header is set (see Fig. 1). It is also checked that the IP address and Port match and that the data part of the message is empty.

When the postcondition is valid, Quickcheck will now update the state. In addition, the state machine model supports state data, i.e., a data parameter that carries from one state to the other. We use the state data to store the last message that we have sent, in order to be able to read sequence numbers and other data in consecutive states.

```
next_state_data(closed, syn_sent, S, Syn,
               {call,_,_,_}) ->
   S#state{last_msg = Syn};
```

Now the peer Subject is in the SYN_SENT state and the peer Tester is in the SYN_RECEIVED state where a *Syn* message has been received.

5.1.2 SYN_SENT

SYN_SENT is the state in which the subject will be after sending its *Syn* packet (Figure 1). In this state the subject will be awaiting a Syn+Ack packet from the tester. The possible transitions from this state are represented as follows in our state machine model:

```
syn_sent(S) ->
 [{established,
    {call, ?MODULE, syn_ack,
       [{var, listener}, S#state.last_msg]}}
 ].
```

That is, from this state the Subject stack only has one transition to ESTABLISHED by calling syn_ack:

```
syn_ack(Listener, Syn) ->
   Syn_ack = #tcp{dst_port = Syn#tcp.src_port,
                  src_port = Syn#tcp.dst_port,
                  dst_ip   = Syn#tcp.src_ip,
                  src_ip   = Syn#tcp.dst_ip,
                  seq      = 0,
                  ack      = seq:add(Syn#tcp.seq, 1),
                  is_ack   = 1,
                  is_syn   = 1,
                  window   = ?DEFAULT_WINDOW
                  },
   Bin_Packet = packet:tcp_to_binary_chksum(Syn_ack),
   tcp_listener:send_packet(Listener,
                            Bin_Packet,
                            Syn#tcp.src_ip),
   get_parsed_tcp_packet(Listener).
```

This function shows how the tester generates replies by using the values from the previous packet received (Syn). The newly created packet has the ack and syn flags set, and will acknowledge the syn by adding 1 to its sequence number.

The Erlang record representation of a packet is then converted into a binary and this packet is sent over the network. After the peer receives the packet we expect it to acknowledge it, so the last step is to get a new inbound TCP packet. Here we benefit from our refined model. The TCP interface would just allow us to listen on one site and open on the other side and we would have got a return after that both sides are in established. Now we create and inspect internal states and can check responses. We could inject faults, cause a timeout or whatever at this point, but we cannot communicate to the open connection of Subject via the API before getting it in the state ESTABLISHED.

The postcondition for this transition validates that the received message is indeed an *Ack*.

```
postcondition(syn_sent, established, S,
             {call, ?MODULE, syn_ack,[_, Syn]},Ack) ->
   check_flags(Ack, [ack]) and
   (Ack#tcp.seq==nxt_seq(Syn)) and
   (Ack#tcp.ack==1) and
   (Ack#tcp.data==<<>>);
```

In this case we validate that the only flag set is the ack flag. This is just an ack for the *Syn* sent by the tester, that the acknowledge sequence number is is increased by one (we sent zero), and that the sequence number should be one more than the syn packet it previously sent, to account the *Syn* flag of that packet. Additionally the data part should be empty.

The state data is once more updated by just storing the last message received, i.e., the *Ack*.

```
next_state_data(syn_sent, established, S, Ack,
               {call, _, _, _}) ->
   S#state{last_msg = Ack};
```

Now the Subject stack is in the state ESTABLISHED and the tester is in the ESTABLISHED state, since it has received an *Ack*. Therewith we have modeled the connection establishment of Figure 2 from one side. The opening where we instruct the Subject to go to the LISTEN state and where the tester sends the first *Syn* message is added to the same model in a similar way.

5.2 Testing Simultaneous Connection Closing

Now we add to the model the possibilities for connection closing and we show how one of the generated test cases corresponds to the simultaneous closing of two connections as shown in Figure 3. This is an interesting test case because it is difficult to simulate in a real network, but it is easy to generate from our model.

Usually, testing would be done by using an operating system stack as peer. Doing a simultaneous closing for testing would require calling the close system call at almost the same time in both peers of the connections. As there are some time issues that fall out of control of the socket interface (for example, the exact time at which the fin packet is generated by the kernel cannot be exactly controlled), it is very difficult to reliably generate a situation like this.

However, we will see how the state machine generates a test case that checks this behaviour in a similar way to the connection establishment.

In a simultaneous closing, both stacks would pass through the FIN_WAIT_1 state, they proceed to the CLOSING state, and finally to the TIME_WAIT state (see Figure 1). The CLOSING state is specific to the simultaneous closing, no other close situation passes through it.

We start with a connection for which both peers are in the ESTABLISHED state.

5.2.1 ESTABLISHED

There are two possible transitions from ESTABLISHED, one initialized via a close in the TCP interface and one initialized by the other peer:

```
established(S) ->
   [{fin_wait_1,
       {call, ?MODULE, active_close,
          [{var, listener}, {var, sut}]}},
    {close_wait,
       {call, ?MODULE, passive_close,
          [{var, listener},S#state.last_msg]}}].
```

A peer may go to the state FIN_WAIT_1 by an active close (the TCP stack got a close via the API), or to the CLOSE_WAIT state by doing a passive close (the stack receives a *Fin* from the other peer). It is important to recall that the current state reflects the state in which Subject should be, so a simultaneous close arises when the subject stack starts the close before receiving a *Fin* packet from the tester.

In a test that checks the simultaneous closing, Quickcheck would generate an active close, i.e. have the controller send a close to Subject, which corresponds to the first alternative in the state transitions above.

```
active_close(Listener, Sut) ->
   simple_sender:close(Sut),
   get_parsed_tcp_packet(Listener).
```

After the Subject is told to close the connection, a *Fin* packet is expected to be received in the IP stack of the tester. First, the postcondition for this transition is validated:

```
postcondition(established, fin_wait_1, S,
             {call,?MODULE, active_close, [_, _]},
             Fin) ->
   Ack = S#state.last_msg,
   check_flags(Fin, [ack, fin]) and
   (Fin#tcp.ack==1) and  %% Acks our Syn
   (Fin#tcp.seq==nxt_seq(Ack)) and
   (Fin#tcp.data== <<>>);
```

The check is fairly similar to the previous ones. The fin flag should be set as well as the ack flag[4]. Sequence numbers should increase and the data part is empty. The state data is updated as in the previous cases, storing the received message:

```
next_state_data(established, fin_wait_1, S, Fin,
               {call, _, _, _}) ->
   S#state{last_msg = Fin};
```

Now the Subject is in the state FIN_WAIT_1, but the Tester peer is still in the state ESTABLISHED. We can choose which action we want the tester to perform. Either it acknowledges the received *Fin*, or it also sends a *Fin*, because we close the connection (the simultaneous close case).

5.2.2 FIN_WAIT_1

Thus, we expect two possible state transitions from the state FIN_WAIT_1. However, as can be seen in Figure 1, there are three possible transitions from this state. The third one is the tester sending a *Fin+Ack* to Subject. Here the specification of Figure 1 is hard to understand and additional knowledge is necessary. From additional text in the specification we learn that it is possible that the peer Tester in state ESTABLISHED receives *Fin* and then responds by sending a combined acknowledge on that *Fin* with its own *Fin*. This is then resulting in a state transition from Tester to LAST_ACK, thereby taking away the need to send an explicit *Close* from the interface. In Figure 1 it is shown that normally one would wait from a close from the interface before sending the last *Fin* in order to completely close the connection.

```
fin_wait_1(S) ->
 [{time_wait,
     {call, ?MODULE, fin_ack,
        [{var, listener}, S#state.last_msg]}},
  {fin_wait_2, {call, ?MODULE, ack_received_fin,
        [{var, listener}, S#state.last_msg]}},
  {closing, {call, ?MODULE, simultaneous_close,
        [{var, listener}, S#state.last_msg]}}
  ].
```

The test generated to check simultaneous closing would cause Subject to perform a state transition to CLOSING by calling the simultaneous_close function:

```
simultaneous_close(Listener, Fin) ->
   Local_Fin = #tcp{dst_port = Fin#tcp.src_port,
                    src_port = Fin#tcp.dst_port,
                    dst_ip = Fin#tcp.src_ip,
                    src_ip = Fin#tcp.dst_ip,
                    seq = 1,
                    ack = Fin#tcp.seq,
                    is_ack = 1,
                    is_fin = 1,
                    window= ?DEFAULT_WINDOW
                   },
   Bin_Packet = packet:tcp_to_binary_chksum(Local_Fin),
   tcp_listener:send_packet(Listener,
                            Bin_Packet,
                            Fin#tcp.src_ip),
   get_parsed_tcp_packet(Listener).
```

In the *Fin* packet we are creating the acknowledge number is taken directly from the sequence number of the subject's *Fin*, instead of increasing it by 1 to acknowledge the packet. This is correct, since we actually simulate to have not seen that message yet. After receiving the constructed packet the subject stack will 'see' a simultaneous close.

The postcondition that needs to be validated is that Tester received an *Ack*:

```
postcondition(fin_wait_1, closing,S,{call, ?MODULE,
                                     simultaneous_close,
                                     [_, _]}, Ack) ->
   Fin = S#state.last_msg,
   check_flags(Ack, [ack]) and
     (Ack#tcp.ack==2) and
     (Ack#tcp.seq==nxt_seq(Fin)) and
     (Ack#tcp.data== <<>>);
```

The next state data is similar to previous cases, we only save the last received message:

```
next_state_data(fin_wait_1, closing, S, Ack,
                            {call, _, _, _}) ->
   S#state{last_msg = Ack};
```

Now Subject has moved to state CLOSING whereas Tester is in the state FIN_WAIT_2, since it has already received the *Ack* from Subject.

5.2.3 CLOSING

In the state CLOSING Subject waits to receive an *Ack* packet that acknowledges the *Fin* it sent before.

[4] Apart from the first message, where the sequence number is unknown, all messages have the ack flag set and contain the sequence number of the previous message.

```
closing(S) ->
 [{time_wait,
    {call, ?MODULE, send_ack,
        [{var, listener}, S#state.last_msg]}}].
```

Thus in this state, QuickCheck generates an acknowledge to the *Fin* we ignored in the previous state:

```
send_ack(Listener, Fin) ->
   Ack = #tcp{dst_port = Fin#tcp.src_port,
              src_port = Fin#tcp.dst_port,
              dst_ip   = Fin#tcp.src_ip,
              src_ip   = Fin#tcp.dst_ip,
              seq      = 2,
              ack      = nxt_seq(Fin),
              is_ack   = 1,
              window   = ?DEFAULT_WINDOW
             },
   Bin_Packet = packet:tcp_to_binary_chksum(Ack),
   tcp_listener:send_packet(Listener,
                            Bin_Packet,
                            Fin#tcp.src_ip).
```

Notice that now the ack is computed using `nxt_fin`, which increases the sequence number by 1. After this packet is received by the subject, the connection would move to `TIME_WAIT`, which is similar to a closed connection. There is nothing to check in this state:

```
postcondition(closing, time_wait, _S,
              {call, ?MODULE, send_ack, [_, _Ack]},
              _) ->
   true;
```

.

There are a few states left, that need to be modeled in the same way. After that, a large number of random tests can be generated that open a connection in one of the two ways and closes them in any of the possible ways. We can create tests that open and close the same port in different ways quickly after each other by making silent transitions from `TIME_WAIT` and `LAST_ACK` to `CLOSED`.

We have in the examples always started the sequence numbers by zero, but one could actually pick a random number and increase from there as yet one more thing to test.

However, we only test one connection at a time and no concurrent connections, which may not trigger certain errors in the subject stack. We also have concentrated on positive testing only, that is, it is not checked that the subject reacts correctly to malformed or unexpected packets.

6. Testing a reference stack

We developed our specification by testing against the Linux kernel TCP stack, assuming that that one is fairly well tested already. Thus, if problems arise, they are likely a fault in our specification. This assumption proved to be correct as all the problems found were in the specification. Two tricky issues one needs to be aware of.

Occupied Ports

We tested one connection at a time and read all TCP communication on the network. Thus, when the listener reads a *Fin* message, it assumed it to be part of the test. However, for a specific *Fin* message the sequence number and Port were wrong, as the postcondition revealed. This could have been an error in the Subject TCP stack, but was not. Out generators did not require the tests to end with the connection closed, so at times a connection was left in some intermediate state. After some time, the subject TCP/IP stack tried to close the connection and sent packets to the tester which were confused as packets related to the test run at that time.

An easy solution to this problem would be to make the listener specification ignore any packet with a wrong destination port. However, we did not want to do that because the tester would not be able to detect if Subject is sending incorrect port numbers. We needed to make the packet listener more intelligent and context aware. Of course, we would have serious problems when we extend the specification to test concurrent connections because this filtering will be unavoidable. However, if we have thoroughly tested the case of one connection and port numbers are always correct, we may probably assume that we can use the port as connection identifier for the concurrent case. This is not that tricky, since if the concurrent case would contain an error causing a changed port number in the return message, then something unexpected would happen and a test will fail.

Reusing Ports

Checking the behaviour in the `TIME_WAIT` state is also difficult. When the stack arrives at this state, it must wait for a time in case the last ack packet has been lost. This time is fairly long (several minutes). While the stack is in this state, trying to open a new connection using the same port number will result in a reset. This is the correct behaviour according to the standard, but it also makes tests much longer. We have added a configurable time for this state, with the idea that the developer of a TCP/IP stack has control over the time waited, and it can be changed to run the tests. However, this has to be considered when testing an out of the box stack, like we did with Linux.

Testing other Stacks

After we corrected several errors in the specification and the Linux stack passed the tests, we tried with a less well tested stack in order to try to actually find errors in the stack. We selected the stack described in [Paris et al. 2005] because we are very familiar with it, it is likely to have some bugs not yet discovered, and being written in Erlang it was very easy to adapt for the tests.

After running several tests we actually found an error in the `LAST_ACK` state, which caused sequential connections using the same port number to fail at times. The bug was due to a lag in cleaning up the list of currently opened connections, which caused the stack to react to the incoming `syn` for the new connections as if it belonged to the old one.

This did not always happen, so the ability of Quickcheck to generate hundreds of test cases was very useful to detect the problem.

7. Conclusions

We have developed a QuickCheck specification for automatic checking of TCP implementations. By generating sequences of TCP packets using quickcheck we are able to generate test cases which are not easy to reproduce in a real environment, like simultaneously closing the connnection from both sides.

This QuickCheck specification differs from earlier QuickCheck specifications for protocols in the use of two interfaces to communicate with the subject under test. Previously we have seen QuickCheck specifications that triggered the subject under test by communicating via the protocol API (e.g. [Arts et al. 2006]). For TCP this is insufficient, we also need to interact via the other end of the protocol, the IP level of the stack.

The use of a very simple controller for the subject makes it easy to adapt the tester to any stack. Ideally, if the stack uses the socket interface it should be able to use the provided controller without changes. Our goal is that the tester can easily be used by anyone developing a TCP/IP stack.

As future work we consider the extension of the model to cover operation of a connection in the `ESTABLISHED` state, such

as sending and receiving data, reordering of data, flow control, retransmissions, *etc*.

Another interesting extension is to add negative testing, which is fairly difficult to do in a real environment because it requires an incorrect implementation of TCP. Negative testing is useful to check the robustness of the subject stack. The idea is that the tester can generate illegal packets as part of a legal sequence and see whether the subject correctly handles these illegal packets.

Negative testing is especially interesting for a TCP/IP stack whose normal environment is the Internet. Any error detected is a potential security problem in a TCP/IP stack. It is also interesting because negative testing cannot be done by using a normal stack as a peer.

We also want to test the behaviour of TCP with many concurrent connections. This is interesting because network stacks are by design highly concurrent, and concurrency is usually a source of errors, so testing their behaviour under those conditions is very desirable.

Acknowledgments

Partially supported by Xunta de Galicia PGIDIT07TIC005105PR.

References

Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing Telecoms Software with Quviq QuickCheck. In *ERLANG'06:Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, New York, NY, USA, 2006. ACM Press.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

John Hughes. Quickcheck Testing for Fun and Profit. In Michael Hanus, editor, *9th International Symposium on Practical Aspects of Declarative Languages*. Springer, 2007.

Information Sciences Institute. RFC 793: Transmission control protocol, 1981. URL `http://rfc.sunsite.dk/rfc/rfc793.html`. Edited by Jon Postel. Available at http://rfc.sunsite.dk/rfc/rfc793.html.

Van Jacobson, Craig Leres, and Steven McCanne. libpcap: Packet capture library. URL `http://www.tcpdump.org`.

Javier Paris, Victor Gulias, and Alberto Valderruten. A high performance erlang TCP/IP stack. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 52–61, New York, NY, USA, 2005. ACM. ISBN 1-59593-066-3. doi: http://doi.acm.org/10.1145/1088361.1088372.

Raid Zaghal and Javed Khan. EFSM/SDL modeling of the original TCP standard (RFC793) and the Congestion Control Mechanism of TCP Reno. URL `http://www.medianet.kent.edu/technicalreports.html`.