Automatic Testing of TCP/IP Implementations using Quickcheck

Erlang Workshop 2009

Javier Paris[1] Thomas Arts[2]

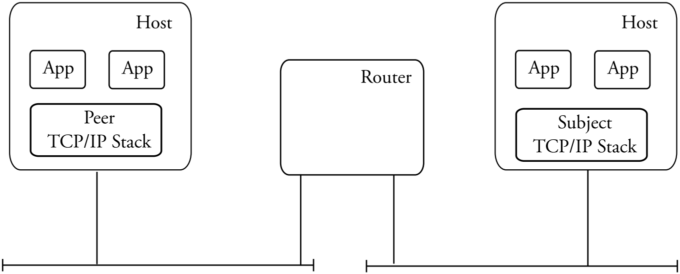
javierparis@udc.es thomas.arts@quviq.com

[1]University of A Coruña

[2]IT University of Gothenburg and Quviq AB

What's the Problem?

- When developing a protocol stack (e.g. a TCP/IP Stack) => How to test?
- Test scenario for checking a protocol stack:
  - The stack we want to test (the subject).
  - The network.
  - A peer.



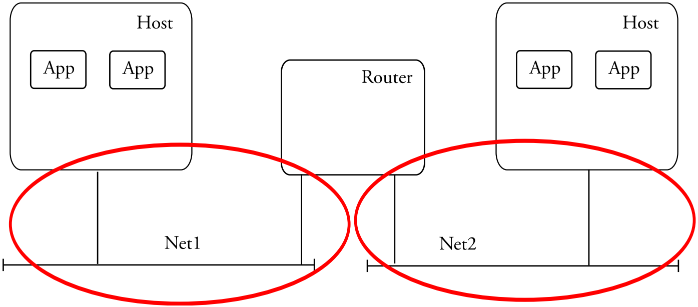
TCP/IP Overview

Layered protocol stack:

- Several different protocols
- Each protocol uses the inmediate lower level to provide services to the upper level.
- The network is abstracted as we go up.

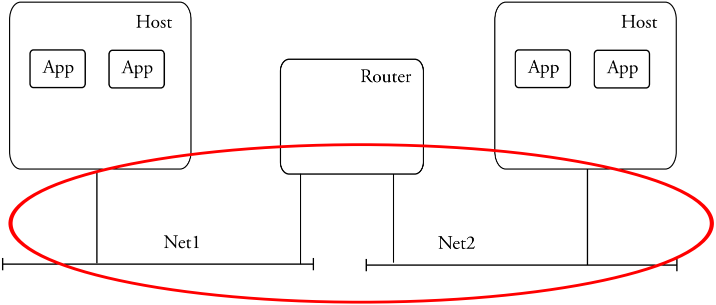
TCP/IP Overview: Link Layer

Link layer provides communication between directly connected devices. (e.g. Ethernet, ppp).



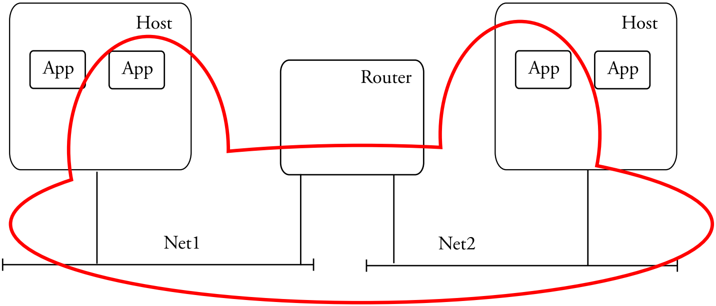
TCP/IP Overview: Network Layer

Network layer provides packet communication over different networks (e.g. IP) => no guarantees of delivery, order, data may be duplicated.
Best effort, but no guarantees.



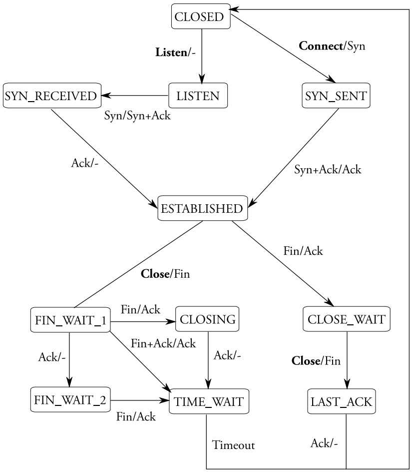
TCP/IP Overview: Transport Layer

Transport layer provides communication between applications in different parts of the network (e.g. TCP) => Guarantees delivery, order, rate control, reordering. Abstracts the network a lot (stream-like interface)



TCP/IP Overview: TCP

TCP uses stateful connnections and its behaviour is given by a fsm:



Writing Tests Manually

- Usual interaction with a stack is through an API (socket API).
- Some behaviours we want to observe cannot be generated by using the API => Must inject traffic manually into the network.
- There are complex conditions that cannot be observer through the API => Checking for correctness requires looking at the actual network traffic.
- It is difficult to create static test cases because there are many independent actors.

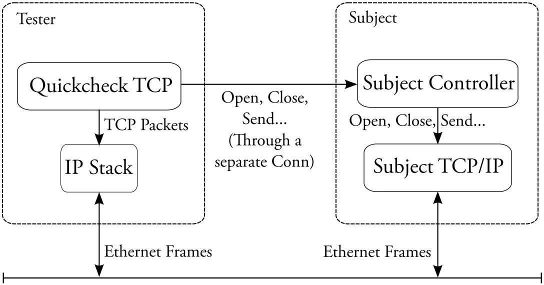
Goals

- To generate and check test cases automatically.
- To be able to check strange conditions which are difficult to create in the network (e.g. malformed packets)

Getting Quickcheck in the Picture

- Quickcheck provides a module for testing finite state machines, and TCP connections behave like a pair of interlocked FSMs (one for each peer).
- We can test by writing a FSM module that follows the state of the TCP stack under test.
- Quickcheck and this module act as the peer of the connection, by checking the received packets and generating suitable replies.

Test Setup

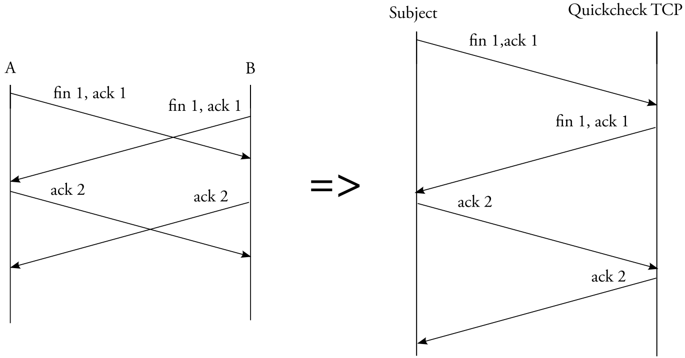

Tester
Quickcheck TCP
TCP Packets
IP Stack
Open, Close, Send... (Through a separate Conn)
Subject
Subject Controller
Open, Close, Send...
Subject TCP/IP
Ethernet Frames
Ethernet Frames

Quickcheck FSM (II)

- The Quickcheck TCP module will not provide a full TCP implementation. Replies are generated based on previously received and sent packets.
- By controlling the subject stack through its API, and generating the packets using the TCP module we can move the TCP state of the subject wherever we want. Example: we can generate a simultaneous close easily:

Quickcheck FSM (III)

Simulation of simultaneous close:



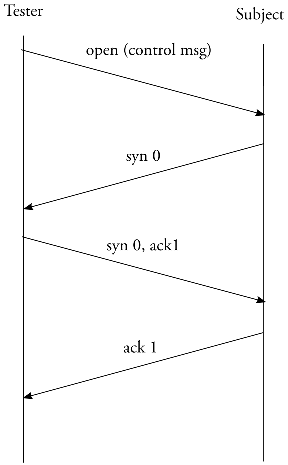
Quickcheck FSM (III)

For each possible state we must provide Quickcheck with:

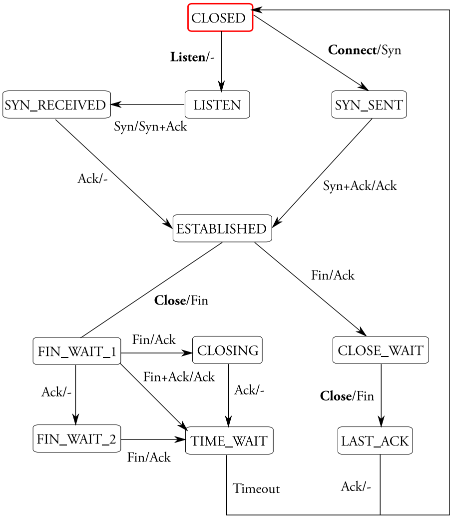
- The possible transitions to other states.
- A set of preconditions for each transition.
- How to actually perform the transition (that is, a function that performs whatever tasks are necessary).
- Postconditions to check after the state transition.
- A description of the changes on the state as a result of the transtition.

Example: Connection Establishment

Example: Test a connection establishment started by the subject.



Example: Connection Establishment (II)



Example: Connection Establishment (III)

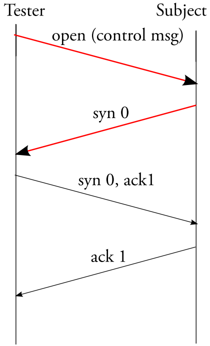
We describe the possible transtitions from `CLOSED`:

```
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}]}}
].
```

Example: Connection Establishment (IV)

For starting a connection the open function would be called to:

- Call the open function through the subject API to start a connection (e.g. `gen_tcp:connect`)
- Listen for the incoming *syn* packet that the previous call caused.



Example: Connection Establishment (V)

Now we check that the subject did it right by looking at the *syn* packet:

```
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 == <<>>);
```

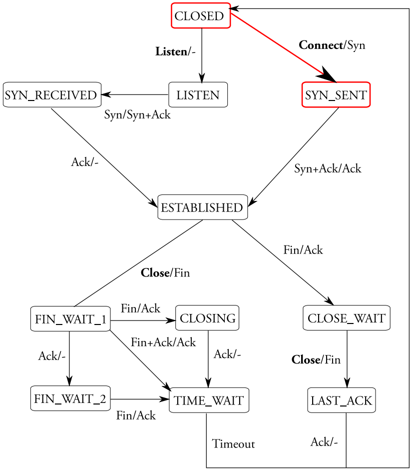
Example: Connection Establishment (VI)

Before transitioning to the next state (SYN_SENT) we update the state:

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

Example:Connection Establishment (VII)

The Quickcheck FSM now transitions to the `SYN_SENT` state:



Example: Connection Establishment (VIII)

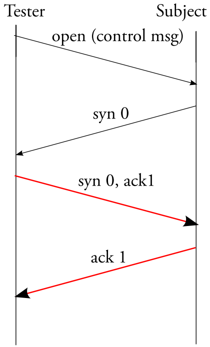
We are now in the SYN_SENT state, from where there is only one possible transition:

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

Example: Connection Establishment (IX)

In this state we must

- Create a packet to reply to the *syn* we received in the previous state. Note that we do not implement a TCP stack, we build the packet using values from the received packet.
- Listen for the *ack* that the subject will send us.



Example: Connection Establishment (X)

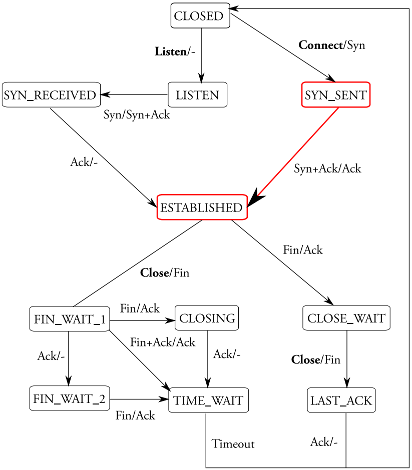
We now check the ack we received:

```
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==<<>>);
```

The state will be updated in a similar way to the previous state.

Example: Connection Establishment (XI)

The Quickcheck FSM now transitions to the `ESTABLISHED` state:



Real World Stuff

- Run tests on the Linux kernel stack: everything was ok (to be expected).
- Run on our Erlang TCP/IP stack: One bug found!
  - Arised on several passive closing of connections using the same port number, which were treated as one single connection.
- What we learnt:
  - It found a bug that we did not know about (good!)
  - Shrinking did not work very well => It made finding the actual problem hard (bad!). Shrinking has to be improved.

Conclusions

- Automatic testing for protocols provides:
  - More comprehensive testing.
  - Testing of features difficult to test by hand (simulation of strange conditions like simultaneous closing because we control the injection of packets).
- Negative conditions are easier to test because we build and decide when reply packets are sent (packet loss, corruption, malformed packets).

Thanks!

Get it at `http://www.madsgroup.org/~paris/tcpcheck.zip`