Testing an Open Source Erlang TCP/IP Stack

Thomas Arts

Ulf Norell

Quviq

Rick Payne

Otolo Networks

Javier Paris

University of A Coruna

TCP in Erlang

The **Transmission Control Protocol** (**TCP**) is one of the main **protocols** of the Internet **protocol** suite.

TCP provides reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts communicating by an IP network. Major Internet applications such as the World Wide Web, email, remote administration, and file transfer rely on TCP.

https://en.wikipedia.org/wiki/Transmission_Control_Protocol

An Erlang implementation

Javier Paris
original author

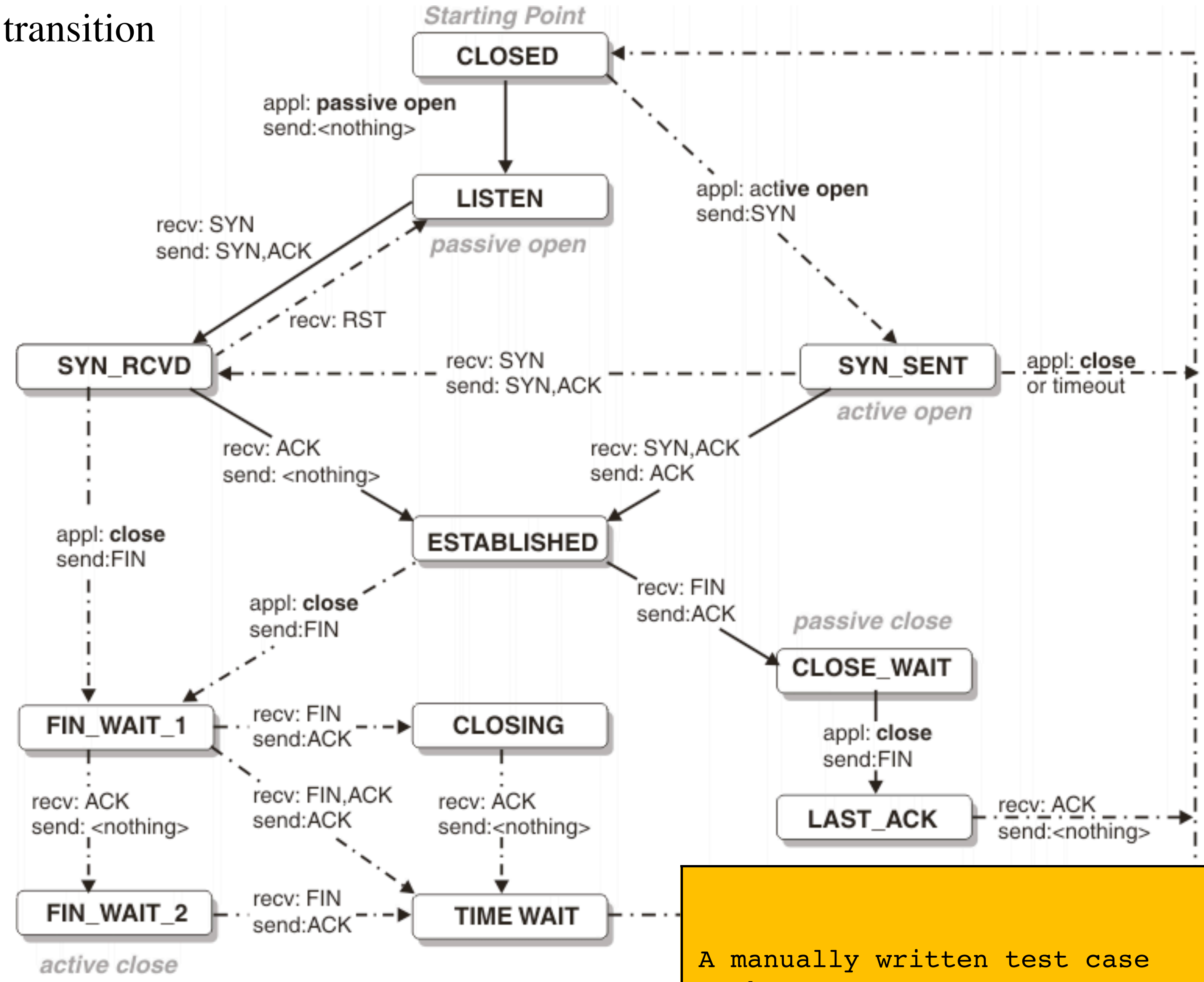
https://github.com/javier-paris/erlang-tcpip

Highly concurrent to serve many connections simultaneous

Performance important, but main goal:

User level stack!

TCP state transition diagram



```
A manually written test case
socket:start().
LS = socket:listen(12345).
socket:close(LS).
```

source: IBM Knowledge Center

Testing TCP

API very little: listen, connect, close, ...

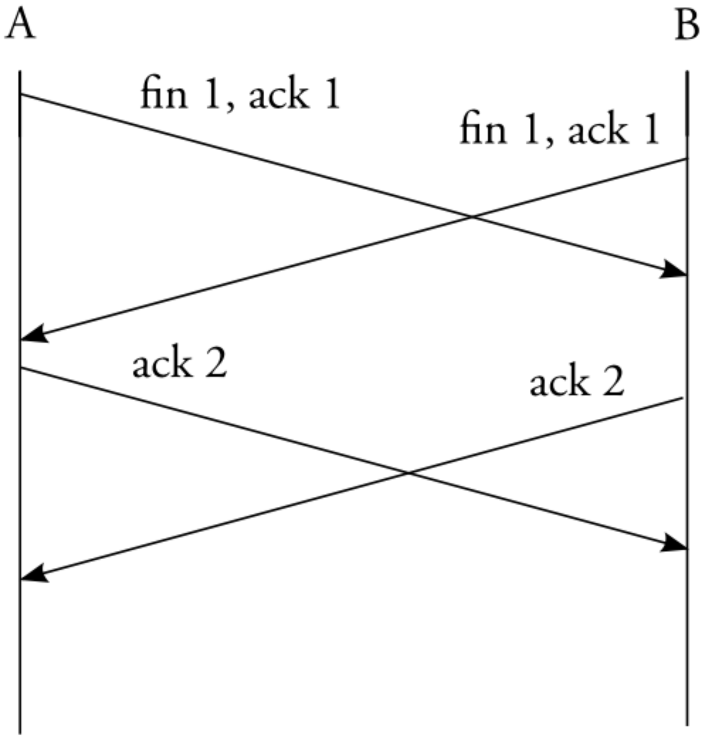
passive open   active open

Need to test many different scen

...but the API does not ste

Race conditions may occur: nee

Faults may appear (message los



Don’t write tests!

Generate them

From the specification

DEMO

Process Registration

Three operations to test:

- register(atom(), pid()) -> true
- unregister(atom()) -> true
- whereis(atom()) -> pid() | undefined

Functions depend on the current state

- register/unregister change it

Testing TCP

Need to test many different scenario's...

...but the API does not steer the scenario!

Race conditions may occur: needs testing

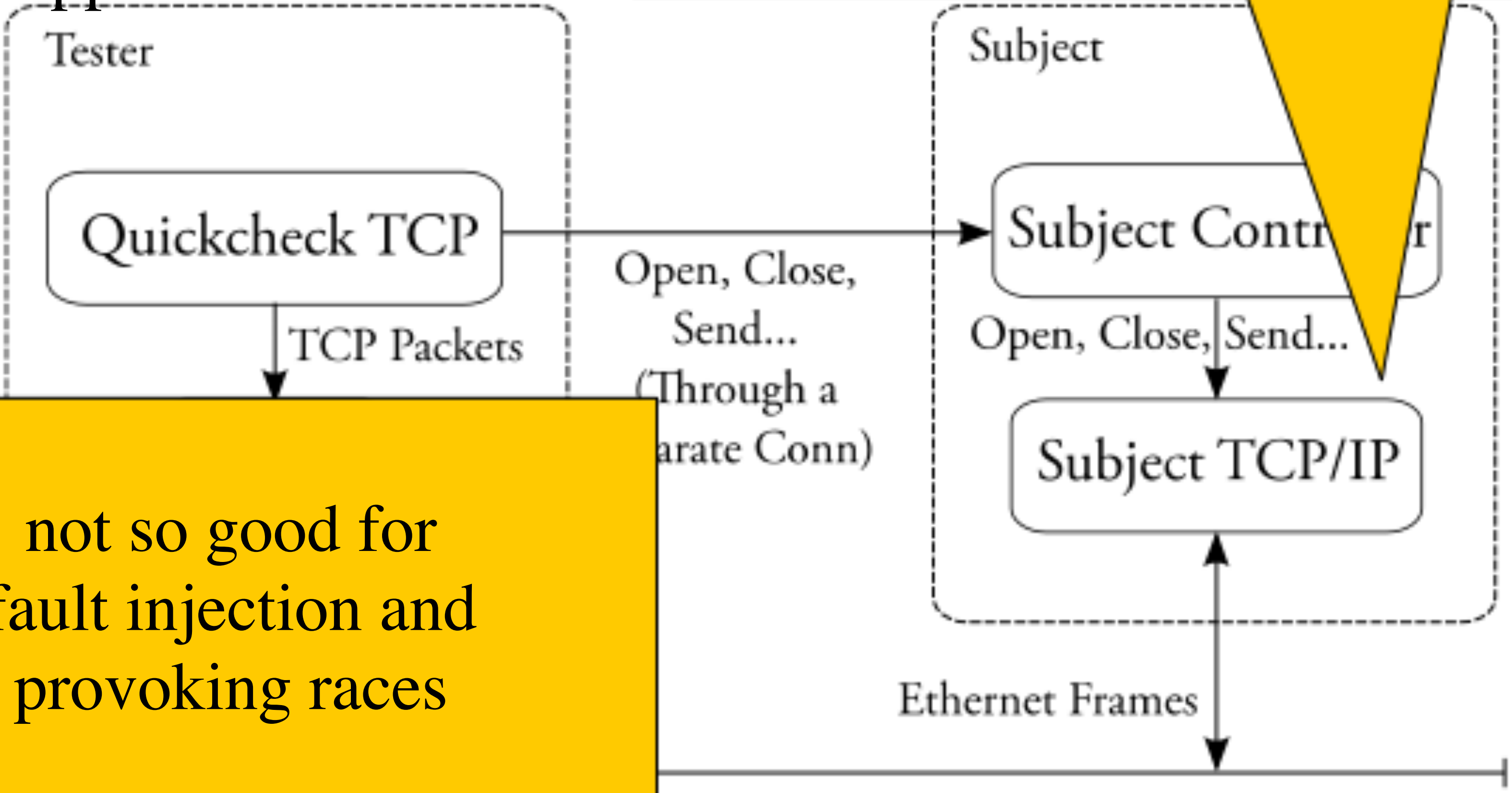
Approach:

QuickCheck: generate tests to cover all scenario's

PULSE: generate random schedules to test for concurrency errors

Testing TCP

Traditional QuickChec
approach

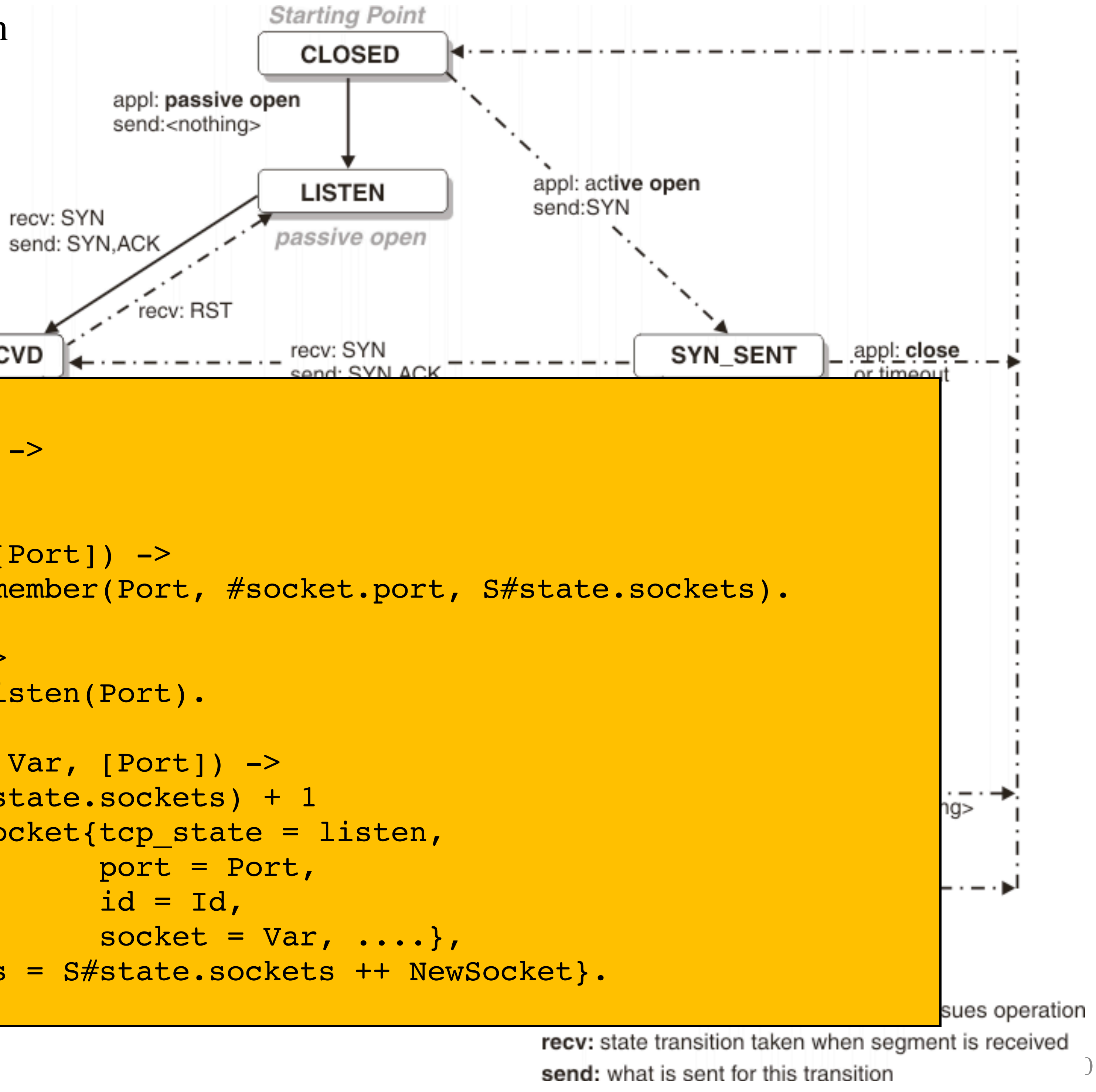


almost system level testing

good for finding incompatibilities

not so good for
fault injection and
provoking races

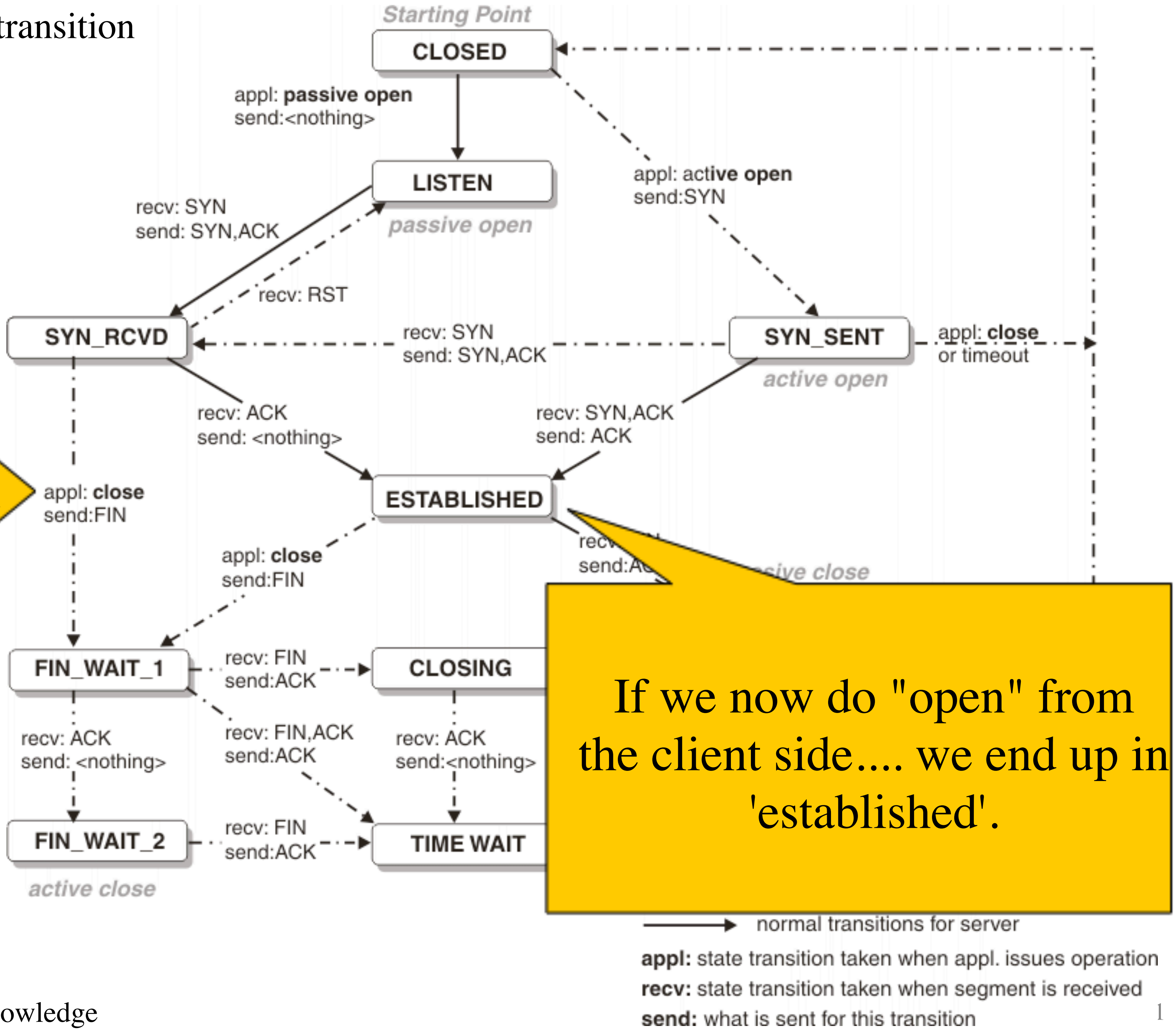
TCP state transition diagram



```
listen_args(_) ->
 [port()].

listen_pre(S, [Port]) ->
 not lists:keymember(Port, #socket.port, S#state.sockets).

listen(Port) ->
 tcp_con:usr_listen(Port).

listen_next(S, Var, [Port]) ->
Id = length(S#state.sockets) + 1
NewSocket = #socket{tcp_state = listen,
                    port = Port,
                    id = Id,
                    socket = Var, ....},
S#state{sockets = S#state.sockets ++ NewSocket}.
```

source: IBM Knowledge Center

TCP state transition diagram

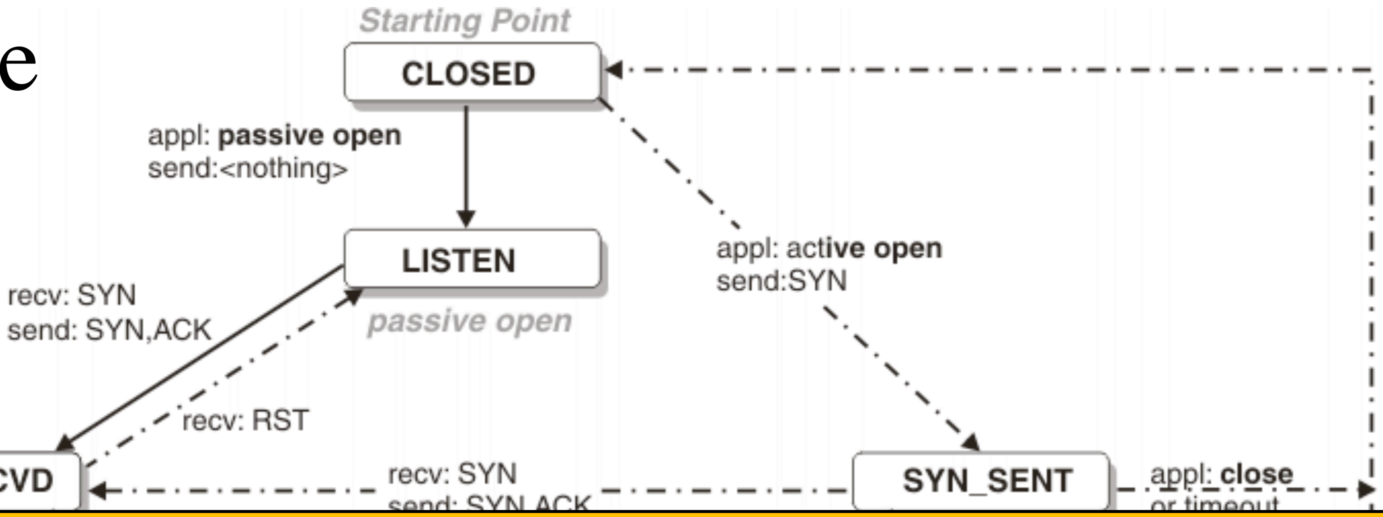


hard to test

If we now do "open" from the client side.... we end up in 'established'.

source: IBM Knowledge Center

Mocking the client



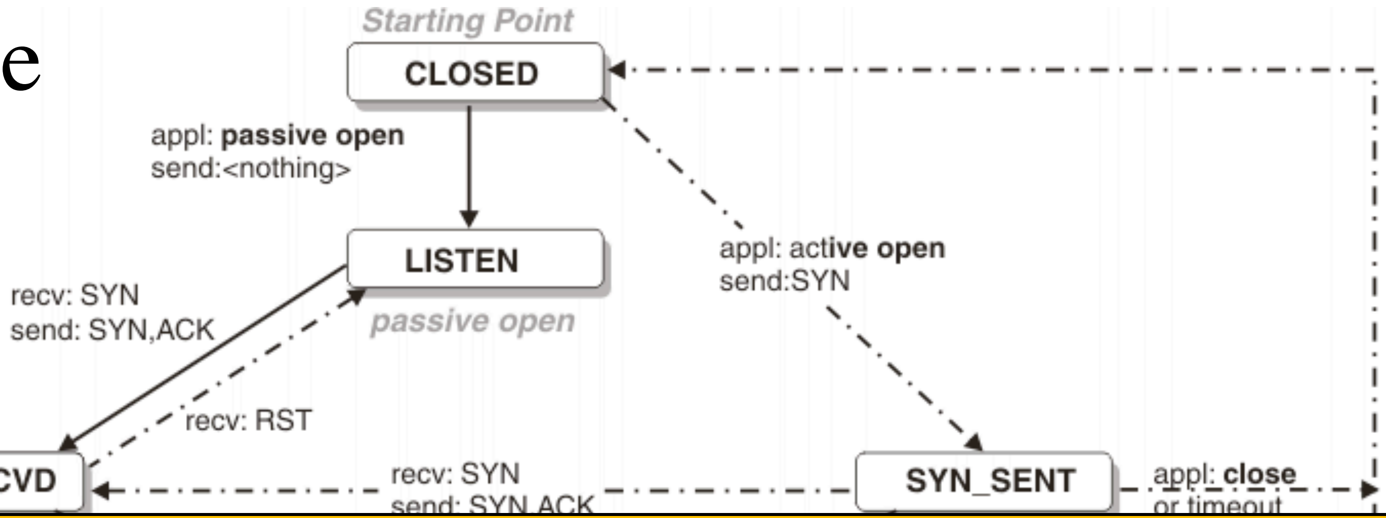
```
syn_pre(S) ->
   [] /= sockets_in_state(S, listen).

syn_args(S) ->
 ?LET(Sock, elements(sockets_in_state(S, listen)),
      [Sock#socket.ip, Sock#socket.port, ip(), port(), uint32(),
      Sock#socket.id]).

syn(Ip, Port, RemoteIp, RemotePort, RemoteSeq, _Id) ->
 Packet =
   #pkt{sport = RemotePort,
        dport = Port,
        seq   = RemoteSeq,
        flags = [syn] },
 Data = encode(RemoteIp, Ip, Packet),
 tcp:recv(ip2int(RemoteIp), ip2int(Ip), Data).
```

recv: state transition taken when segment is received

send: what is sent for this transition

source: IBM Knowledge Center

Mocking the client



```
syn_callouts(S, [_Ip,  Port, RemoteIp, RemotePort, RemoteSeq, Id]) ->
 ?MATCH(NewId, ?APPLY(spawn_socket, [])),
 ?SET(NewId, port,    Port),
 ?SET(NewId, rip,     RemoteIp),
 ?SET(NewId, rport,   RemotePort),
 ?SET(NewId, rseq,    {RemoteSeq, 1}),
 ?SET(NewId, socket_type, accept),
 ?SET(NewId, parent, Id),
 Sock = get_socket(S, NewId),
 ?MATCH(Packet, ?APPLY(sent, [NewId])),
 ?ASSERT(?MODULE, check_packet, [Packet, '_', Sock#socket.rseq, [ack, syn]]),
 ?SET(NewId, seq, {{call, erlang, element, [#pkt.seq, Packet]}, 1}),
 ?SET(NewId, tcp_state, syn_rcvd).
```

recv: state transition taken when segment is received

send: what is sent for this transition

source: IBM Knowledge Center

Specification → Testing

QuickCheck model: a specification of the diagram

We automatically generate tests to check whether erlang-tcpip follows the specification

The QuickCheck model is general !

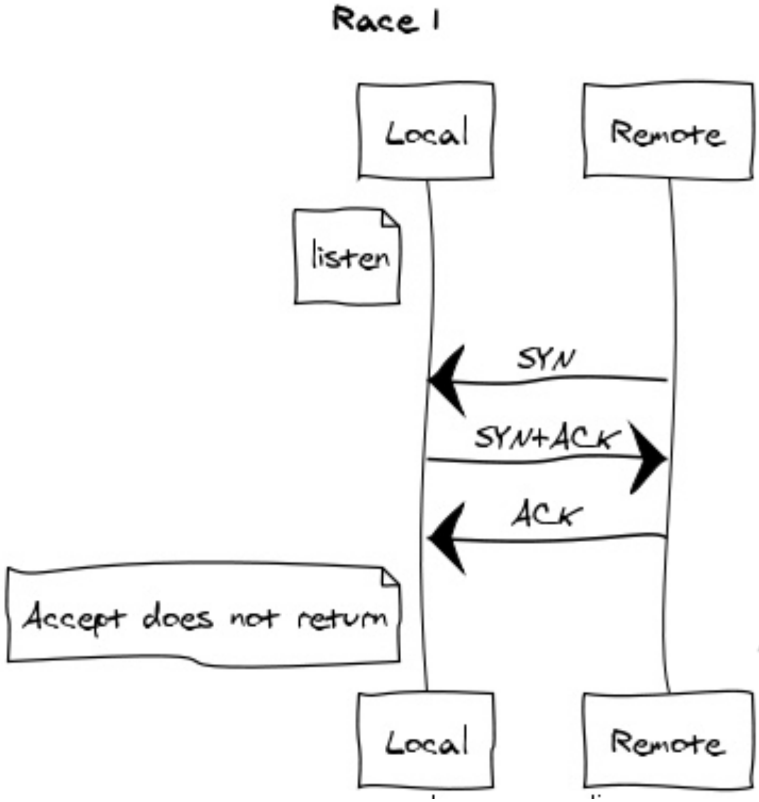
- we can use it for any TCP implementation
- we can fault inject at any possible place

Test results

Quviq tests

Otolo Networks fixes bugs

test experts
TCP/IP
experts



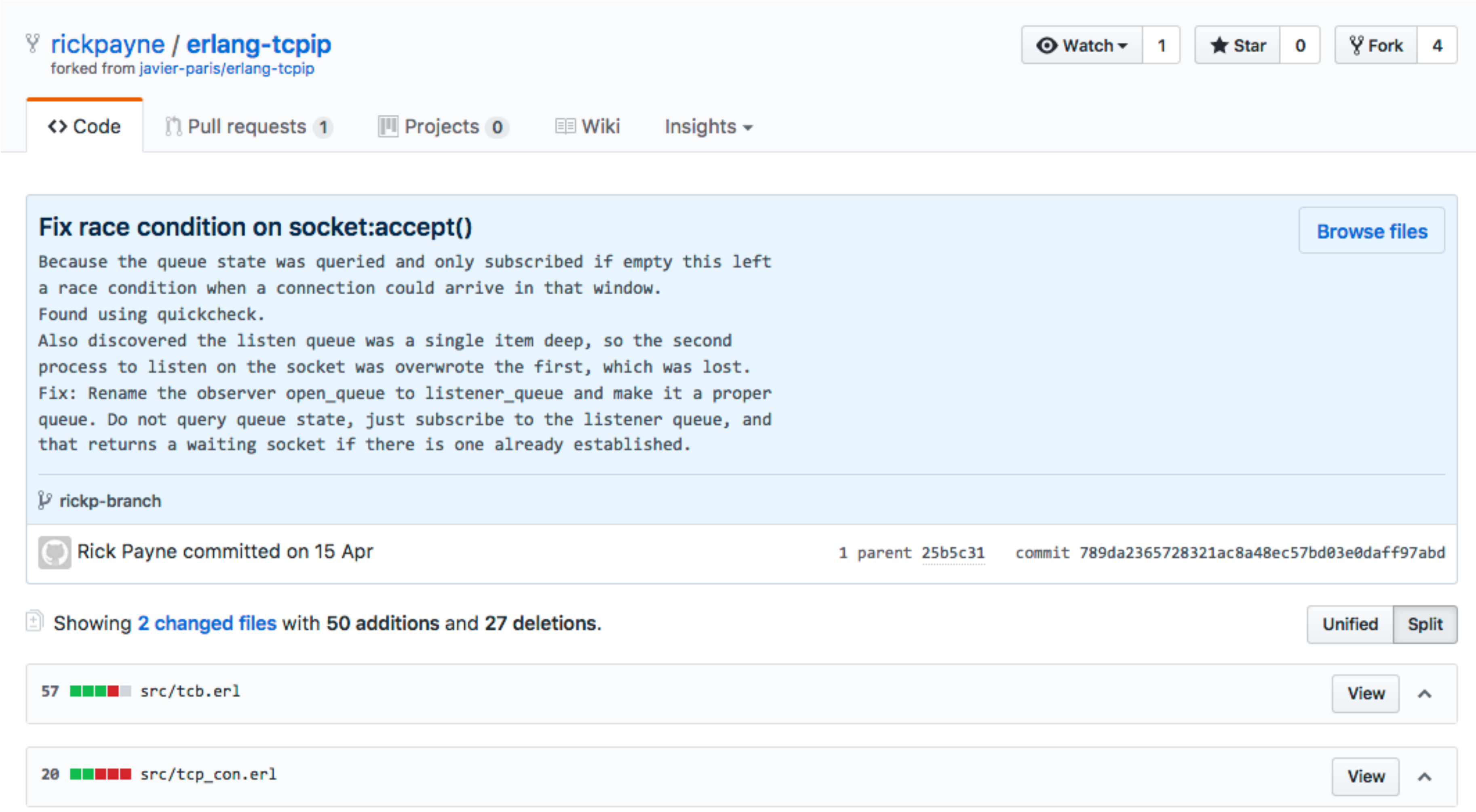
Race condition in implementation

Need to run same test often to find this

Otolo Networks

https://github.com/rickpayne/erlang-tcpip

Fix: 789da2365728321ac8a48ec57bd03e0daff97abd



Fix race condition on socket:accept()

```
Because the queue state was queried and only subscribed if empty this left
a race condition when a connection could arrive in that window.
Found using quickcheck.
Also discovered the listen queue was a single item deep, so the second
process to listen on the socket was overwrote the first, which was lost.
Fix: Rename the observer open_queue to listener_queue and make it a proper
queue. Do not query queue state, just subscribe to the listener queue, and
that returns a waiting socket if there is one already established.
```

rickp-branch

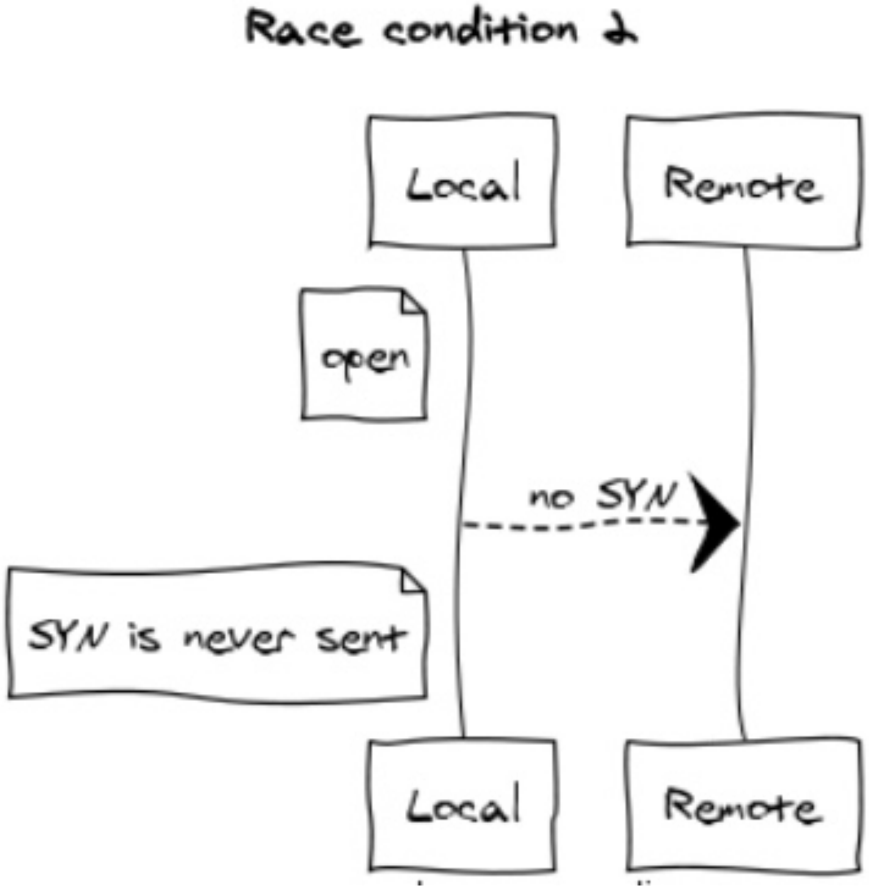
Rick Payne committed on 15 Apr   1 parent 25b5c31   commit 789da2365728321ac8a48ec57bd03e0daff97abd

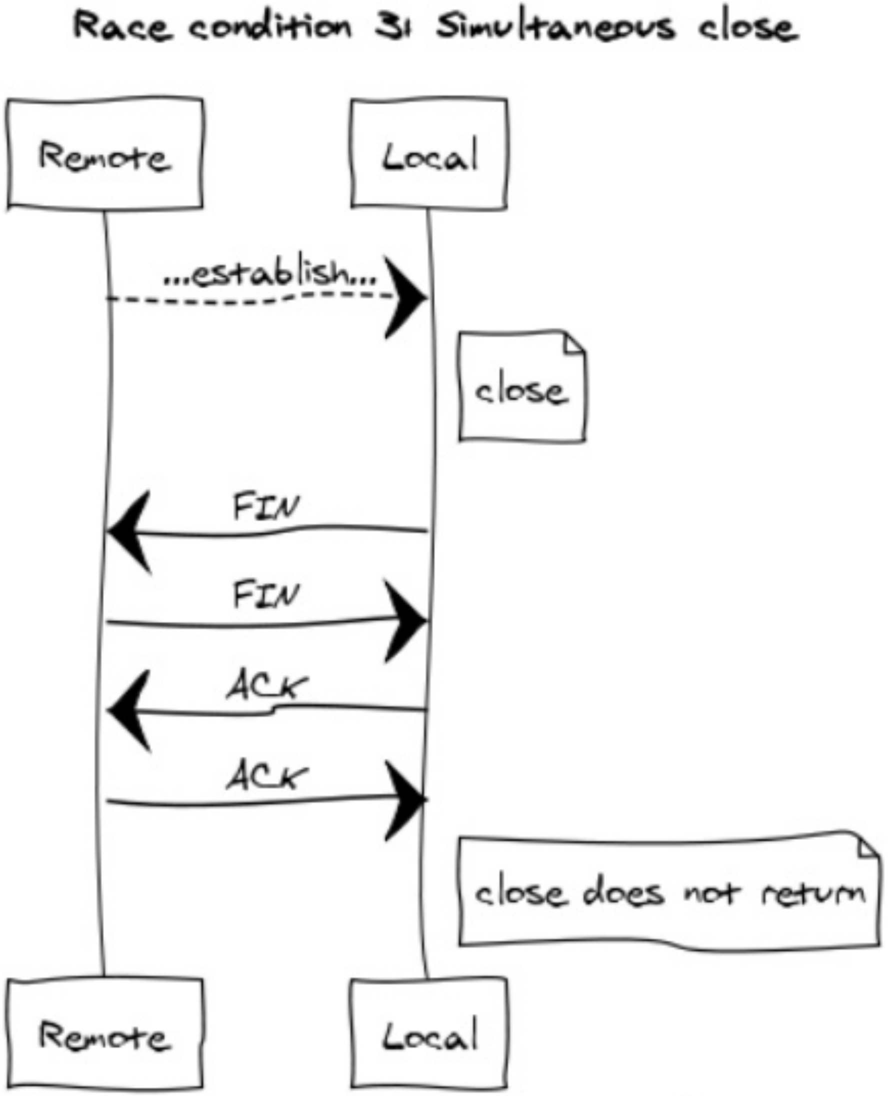
Showing 2 changed files with 50 additions and 27 deletions.

57 🟩🟩🟩🟥⬜ src/tcb.erl

20 🟩🟩🟥🟥🟥 src/tcp_con.erl

Some other issues found





PULSE

PULSE: user level scheduler for Erlang

PULSE is non-deterministic (random scheduling)

PULSE can re-run a schedule (repeatable tests)

When a test fails, PULSE shrinks the schedule to the minimal number of context switches needed to provoke the error.

Easy to understand problem!

How to use **PULSE**

pulse_instrument:

Instrumentation of the code at compile time

Implemented as parse_transform compiler option

Example:

c(example,[{parse_transform,pulse_instrument}]).

Calls to ***spawn***, ***link*** as well as statements ***!*** and ***receive***, etc are replaced by calls handled by **PULSE**

How **PULSE** works

- Controls the concurrency
  - Only one process is executing at a time
- Records all concurrency events
  - Message sending
  - Process spawning
  - Etc…
- **PULSE** can switch to executing another process (simulating context switch) at any time
- We make sure that unlikely scenarios get tested

Conclusions

Using QuickCheck and PULSE have shown to be effective in finding tricky errors.

What's next:

Adding:

RFC 2385 MD5 checksum signing of TCP packets

Contribute with your extensions!