Rigorous Specification and Conformance Testing Techniques for Network Protocols, as applied to TCP, UDP, and Sockets

Steve Bishop[*] Matthew Fairbairn[*] Michael Norrish[†]
Peter Sewell[*] Michael Smith[*] Keith Wansbrough[*]

[*]University of Cambridge [†]NICTA, Canberra

http://www.cl.cam.ac.uk/users/pes20/Netsem

Network Protocols

All those protocols: BGP, OSPF, RIP,..., IP, UDP, TCP, ...

They work.

And you probably all understand them.

But...

Network Protocols. Mostly They Work, But...

They're complicated!

Both for intrinsic reasons:

- packet loss, host failure, flow- and congestion-control
- concurrency, time dependency
- defence against attack

and contingent reasons:

- many historical artifacts (in the Sockets API too)

So what *are* they, really?

How are the protocols described? Standard practice:

For UDP and TCP:

- Original RFCs from 1980s: 768, 793,...
- Later RFCs, options, modifications; POSIX (for Sockets API)
- Well-known texts, e.g. Stevens's TCP/IP Illustrated
- The Code (esp. BSD implementations). C, 15 000–20 000 lines, multi-threaded, time-dependent, entangled with OS, optimised for performance, tweaked over time

Detailed wire formats, but informal prose/pseudocode/C for the endpoint behaviour.

Those informal descriptions

good in the early days (arguably):

- accessible? easy to change? discouraged over-specification?
- emphasis on interop compensated for inevitable vagueness and ambiguity.

but now we all pay the price:

- protocols hard to implement ‘correctly’ (what does ‘correctly’ mean?! how can you test?!)
- API hard to use correctly
- many subtle differences between implementations. Some intended, some not.

Our Goals

Focus on TCP (and UDP, ICMP, and the Sockets API).

1. describe the *de facto* standard — what the behaviour of (some of) the deployed implementations really is

2. develop pragmatically-feasible ways to write better protocol descriptions

'Better' Protocol Descriptions

Protocol descriptions should be simultaneously:

1. *clear*, accessible to a broad community, and easy to modify
2. *unambiguous*, precise about all the behaviour that is specified
3. sufficiently *loose*, not over-specifying (permitting high-performance implementations without over-constraining their structure)
4. directly usable as a basis for *conformance testing*, not read-and-forget documents

What we've done

Developed a *post-hoc* specification of the behaviour of TCP, UDP, relevant parts of ICMP, and the Sockets API that is:

- mathematically rigorous
- detailed
- readable
- accurate
- covers a wide range of usage

(oh, and found sundry bugs and wierdnesses on the way...)

How have we done it? Experimental Semantics...

Take *de facto* standard seriously: pick 3 common impls (FreeBSD 4.6–RELEASE, Linux 2.4.20–8, WinXP SP1).

Gain confidence in accuracy by *validating* the specification against their real-world behaviour:

- Write draft spec
- Generate 3000+ implementation traces on a small network
- Test that those implementation traces are allowed by the spec, using a special-purpose symbolic model checker. (computationally heavy: 50 hours on 100 processors)
- Fix and iterate.

What we've not done

- Redesign TCP better
- Reimplement TCP better
- Prove that the implementations are 'correct' (wrt our spec)
- Prove that the protocol design is 'correct' (wrt some stream abstraction)
- Model-check the implementation code directly
- Generate tests from the spec

Part 1: Introduction

Part 2: Modelling Choices

Part 3: The Specification

Part 4: Validation

Part 5: What we have learned

Specification language

Spec must be loose enough to allow variations:

- TCP options, initial window sizes, other impl diffs
- OS scheduling, processing delays, timer variations, ...

This nondeterminism means we can't use a conventional programming language (*not* a reference impl).

But, need rich language:

- queues, lists, timing properties, mod-2^{32} sums

hence... use operational semantics idioms in higher-order logic – lets us write arbitrary mathematics.

Specification tool – HOL

Machine-process the definition in the HOL system.

HOL system does machine-checking of proofs, and provides scriptable proof tactics, for higher-order logic.

Separate concerns:

- optimize spec for clarity
- build testing algorithmics into checker
- script checker above HOL, so it's guaranteed sound

  (In testing that a real-world trace is allowed by the spec, the checker produces a machine-checked theorem to that effect.)

Modelling choices

Network interface:

- Model UDP datagrams, ICMP datagrams, TCP segments.
- Abstract from IP fragmentation
- Given that, consider arbitrary incoming wire traffic.

Sockets interface:

- Cover arbitrary API usage (and misusage) for `SOCK_STREAM` and `SOCK_DGRAM` sockets.
- Abstract from the pointer-passing C interface, e.g. from

  ```
  int accept(int s, struct sockaddr *addr,socklen_t *addrlen)
  ```

  to a value-passing $\mathrm{accept} : \mathrm{fd} \to \mathrm{fd} * (\mathrm{ip} * \mathrm{port})$.

Modelling choices

Protocols:

TCP: roughly what's in FreeBSD 4.6-RELEASE: MSS; RFC1323 timestamp and window scaling; PAWS; RFC2581/RFC2582 New Reno congestion control; observable behaviour of syncaches.

no RFC1644 T/TCP (is in that code), SACK, ECN,...

Time:

Ensure the specification includes the behaviour of real systems with (boundedly) inaccurate clocks, loosely constraining host 'ticker' rates, and putting lower and/or upper bounds on times for various operations.

Part 1: Introduction

Part 2: Modelling Choices

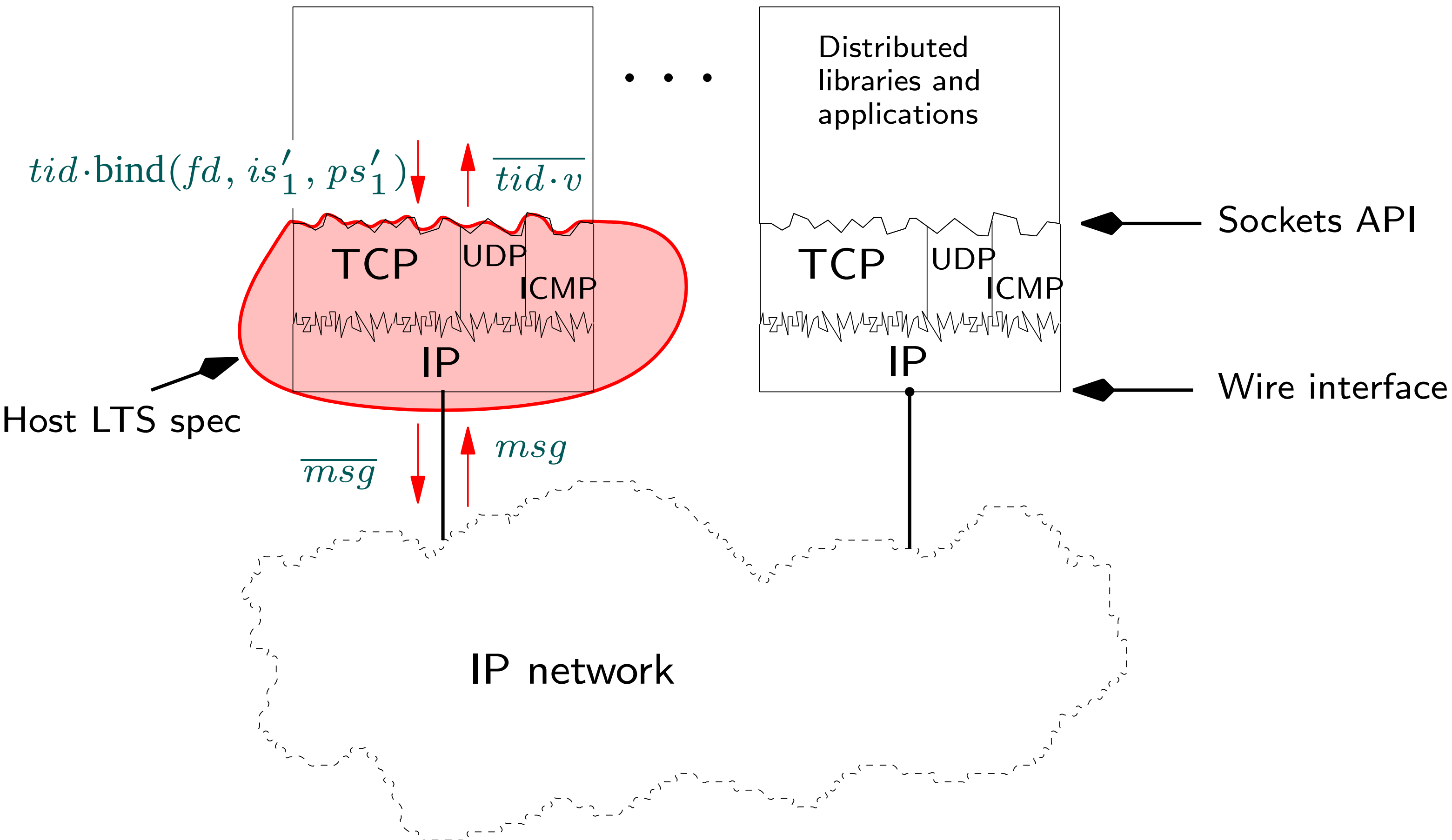
Part 3: The Specification

Part 4: Validation

Part 5: What we have learned

What part of the system to model?

Go for an endpoint (segment-level) specification. The main part of the spec is the *host labelled transition system (LTS)* $h \xrightarrow{lbl} h'$



with internal (τ) and time passage (dur) transitions

The Specification: Host State Type

host $=\langle\!|$ $arch$: arch; (* OS version *)
$privs$: bool; (* whether process has privilege *)
$ifds$: ifid $\mapsto$ ifd; (* network interfaces *)
$rttab$: routing_table; (* routing table *)
ts : tid $\mapsto$ hostThreadState timed; (* host view of each thread state *)
$files$: fid $\mapsto$ file; (* open file descriptions *)
$socks$: sid $\mapsto$ socket;(* sockets *)
$listen$: sid list; (* list of listening sockets *)
$bound$: sid list; (* bound sockets in order *)
iq : msg list timed;(* input queue *)
oq : msg list timed;(* output queue *)
$bndlm$: bandlim_state; (* bandlimiting *)
$ticks$: ticker; (* kernel timer *)
fds : fd $\mapsto$ fid (* process file descriptors *)$|\!\rangle$

The Specification: Sample rules defining $h \xrightarrow{lbl} h'$

(roughly 148 for Sockets, 46 for message processing)

accept_1	Return new connection; either immediately or from a blocked state.
accept_2	Block waiting for connection
accept_3	Fail with EAGAIN: no pending connections and non-blocking semantics set
accept_4	Fail with ECONNABORTED: the listening socket has *cantsndmore* set or has become CLOSED. Returns either immediately or from a blocked state.
accept_5	Fail with EINVAL: socket not in LISTEN state
accept_6	Fail with EMFILE: out of file descriptors
accept_7	Fail with EOPNOTSUPP or EINVAL: accept() called on a UDP socket

The Specification: A Simple Sample Rule

bind_5 **rp_all: fast fail** **Fail with EINVAL: the socket is already bound to an address and does not support rebinding; or socket has been shutdown for writing on FreeBSD**

$$h\langle\!\langle ts := ts \oplus (tid \mapsto (\mathsf{Run})_d)\rangle\!\rangle$$

$$\xrightarrow{tid \cdot \mathrm{bind}(fd, is1, ps1)} \quad h\langle\!\langle ts := ts \oplus (tid \mapsto (\mathsf{Ret}(\mathsf{FAIL\ EINVAL}))_{\mathrm{sched\_timer}})\rangle\!\rangle$$

$$fd \in \mathbf{dom}(h.fds) \wedge fid = h.fds[fd] \wedge$$
$$h.files[fid] = \mathsf{File}(\mathsf{FT\_Socket}(sid), ff) \wedge$$
$$h.socks[sid] = sock \wedge$$
$$(sock.ps1 \neq * \vee$$
$$(\mathrm{bsd\_arch}\ h.arch \wedge sock.pr = \mathsf{TCP\_PROTO}(tcp\_sock) \wedge \ldots))$$

The Specification: A Less Simple Sample Rule

deliver_in_1 **tcp: network nonurgent**

Passive open: receive SYN, send SYN,ACK

$h \, \langle\!| socks := socks \oplus [(sid, sock)];$ (* listening socket *)

$iq := iq;$ (* input queue *)

$oq := oq |\!\rangle$ (* output queue *)

$\xrightarrow{\tau}$

$h \, \langle\!| socks := socks \oplus$

(* listening socket *)

$[(sid, \mathsf{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, csm, crm,$

$\mathsf{TCP\_Sock}(\mathsf{LISTEN}, cb, \uparrow lis', [\,], *, [\,], *, \mathsf{NO\_OOB})));$

(* new connecting socket *)

$(sid', \mathsf{Sock}(*, sf', \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, csm, crm,$

$\mathsf{TCP\_Sock}(\mathsf{SYN\_RCVD}, cb'', *, [\,], *, [\,], *, \mathsf{NO\_OOB})))];$

$iq := iq';$

$oq := oq' |\!\rangle$

(* check first segment matches desired pattern; unpack fields *)

$\mathrm{dequeue\_iq}(iq, iq', \uparrow(\mathsf{TCP}\ seg)) \wedge$

$(\exists win\_\ ws\_\ mss\_\ PSH\ URG\ FIN\ urp\ data\ ack.$

$seg =$

$\langle\!| is_1 := \uparrow i_2; is_2 := \uparrow i_1; ps_1 := \uparrow p_2; ps_2 := \uparrow p_1;$

$seq := tcp\_seq\_flip\_sense(seq : \mathrm{tcp\_seq\_foreign});$

$ack := tcp\_seq\_flip\_sense(ack : \mathrm{tcp\_seq\_local});$

$URG := URG; ACK := \mathbf{F}; PSH := PSH;$

$RST := \mathbf{F}; SYN := \mathbf{T}; FIN := FIN;$

$win := win\_; ws := ws\_; urp := urp; mss := mss\_; ts := ts;$

$data := data$

$|\!\rangle \wedge$

$\mathbf{w2n}\ win\_ = win \wedge$ (* type-cast from word to integer *)

$\mathbf{option\_map}\ \mathbf{ord}\ ws\_ = ws \wedge$

$\mathbf{option\_map}\ \mathbf{w2n}\ mss\_ = mss) \wedge$

(* IP addresses are valid for one of our interfaces *)

$i_1 \in \mathrm{local\_ips}\ h.ifds \wedge$

$\neg(\mathrm{is\_broadormulticast}\ h.ifds\ i_1) \wedge \neg(\mathrm{is\_broadormulticast}\ h.ifds\ i_2) \wedge$

(* sockets distinct; segment matches this socket; unpack fields of socket *)

$sid \notin (\mathbf{dom}(socks)) \wedge sid' \notin (\mathbf{dom}(socks)) \wedge sid \neq sid' \wedge$

$\mathrm{tcp\_socket\_best\_match}\ socks(sid, sock) seg\ h.arch \wedge$

$sock = \mathsf{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, csm, crm,$

$\mathsf{TCP\_Sock}(\mathsf{LISTEN}, cb, \uparrow lis, [\,], *, [\,], *, \mathsf{NO\_OOB})) \wedge$

(* socket is correctly specified (note BSD listen bug) *)

$((is_2 = * \wedge ps_2 = *) \vee$

$(\mathrm{bsd\_arch}\ h.arch \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2)) \wedge$

$(\mathbf{case}\ is_1\ \mathbf{of}\ \uparrow i1' \rightarrow i1' = i_1 \parallel * \rightarrow \mathbf{T}) \wedge$

$\neg(i_1 = i_2 \wedge p_1 = p_2) \wedge$

(* (elided: special handling for TIME_WAIT state, 10 lines) *)

(* place new socket on listen queue *)

$\mathrm{accept\_incoming\_q0}\ lis\ \mathbf{T} \wedge$

(* (elided: if drop_from_q0, drop a random socket yielding q0') *)

$lis' = lis\ \langle\!| q_0 := sid' :: q_0' |\!\rangle \wedge$

(* choose MSS and whether to advertise it or not *)

$advmss \in \{n \mid n \geq 1 \wedge n \leq (65535 - 40)\} \wedge$

$advmss' \in \{*; \uparrow advmss\} \wedge$

(* choose whether this host wants timestamping; negotiate with other side *)

$tf\_rcvd\_tstmp' = \mathbf{is\_some}\ ts \wedge$

$(\mathbf{choose}\ want\_tstmp :: \{\mathbf{F}; \mathbf{T}\}.$

$tf\_doing\_tstmp' = (tf\_rcvd\_tstmp' \wedge want\_tstmp)) \wedge$

(* calculate buffer size and related parameters *)

$(rcvbufsize', sndbufsize', t\_maxseg', snd\_cwnd') =$

$\mathrm{calculate\_buf\_sizes}\ advmss\ mss * (\mathrm{is\_localnet}\ h.ifds\ i_2)$

$(sf.n(\mathsf{SO\_RCVBUF}))(sf.n(\mathsf{SO\_SNDBUF}))$

$tf\_doing\_tstmp'\ h.arch \wedge$

$sf' = sf\ \langle\!| n := \mathrm{funupd\_list}\ sf.n[(\mathsf{SO\_RCVBUF}, rcvbufsize');$

$(\mathsf{SO\_SNDBUF}, sndbufsize')] |\!\rangle \wedge$

(* choose whether this host wants window scaling; negotiate with other side *)

$req\_ws \in \{\mathbf{F}; \mathbf{T}\} \wedge$

$tf\_doing\_ws' = (req\_ws \wedge \mathbf{is\_some}\ ws) \wedge$

$(\mathbf{if}\ tf\_doing\_ws'\ \mathbf{then}$

$rcv\_scale' \in \{n \mid n \geq 0 \wedge n \leq \mathrm{TCP\_MAXWINSCALE}\} \wedge$

$snd\_scale' = \mathbf{option\_case}\ 0\ \mathbf{I}\ ws$

$\mathbf{else}$

$rcv\_scale' = 0 \wedge snd\_scale' = 0) \wedge$

(* choose initial window *)

$rcv\_window \in \{n \mid n \geq 0 \wedge$

$n \leq \mathrm{TCP\_MAXWIN} \wedge$

$n \leq sf.n(\mathsf{SO\_RCVBUF})\} \wedge$

(* record that this segment is being timed *)

$(\mathbf{let}\ t\_rttseg' = \uparrow(\mathrm{ticks\_of}\ h.ticks, cb.snd\_nxt)\ \mathbf{in}$

(* choose initial sequence number *)

$iss \in \{n \mid \mathbf{T}\} \wedge$

(* acknowledge the incoming SYN *)

$\mathbf{let}\ ack' = seq + 1\ \mathbf{in}$

(* update TCP control block parameters *)

$cb' =$

$cb\ \langle\!| tt\_keep := \uparrow((())_{\mathrm{slow\_timer\ TCPTV\_KEEP\_IDLE}});$

$tt\_rexmt := \mathrm{start\_tt\_rexmt}\ h.arch\ 0\ \mathbf{F}\ cb.t\_rttinf;$

$iss := iss; irs := seq;$

$rcv\_wnd := rcv\_window; tf\_rxwin0sent := (rcv\_window = 0);$

$rcv\_adv := ack' + rcv\_window; rcv\_nxt := ack';$

$snd\_una := iss; snd\_max := iss + 1; snd\_nxt := iss + 1;$

$snd\_cwnd := snd\_cwnd'; rcv\_up := seq + 1;$

$t\_maxseg := t\_maxseg'; t_{advmss} := advmss';$

$rcv\_scale := rcv\_scale'; snd\_scale := snd\_scale';$

$tf\_doing\_ws := tf\_doing\_ws';$

$ts\_recent := \mathbf{case}\ ts\ \mathbf{of}$

$* \rightarrow cb.ts\_recent \parallel$

$\uparrow(ts\_val, ts\_ecr) \rightarrow (ts\_val)^{\mathrm{TimeWindow}}_{\mathrm{kern\_timer\ dtsinval}};$

$last\_ack\_sent := ack';$

$t\_rttseg := t\_rttseg';$

$tf\_req\_tstmp := tf\_doing\_tstmp';$

$tf\_doing\_tstmp := tf\_doing\_tstmp'$

$|\!\rangle) \wedge$

(* generate outgoing segment *)

$\mathbf{choose}\ seg' :: \mathrm{make\_syn\_ack\_segment}\ cb'$

$(i_1, i_2, p_1, p_2)(\mathrm{ticks\_of}\ h.ticks).$

(* attempt to enqueue segment; roll back specified fields on failure *)

$\mathrm{enqueue\_or\_fail}\ \mathbf{T}\ h.arch\ h.rttab\ h.ifds[\mathsf{TCP}\ seg']oq$

$(cb$

$\langle\!| snd\_nxt := iss;$

$snd\_max := iss;$

$t\_maxseg := t\_maxseg';$

$last\_ack\_sent := \mathrm{tcp\_seq\_foreign}\ 0w;$

$rcv\_adv := \mathrm{tcp\_seq\_foreign}\ 0w$

$|\!\rangle) cb'(cb'', oq')$

Part 1: Introduction

Part 2: Modelling Choices

Part 3: The Specification

Part 4: Validation

Part 5: What we have learned

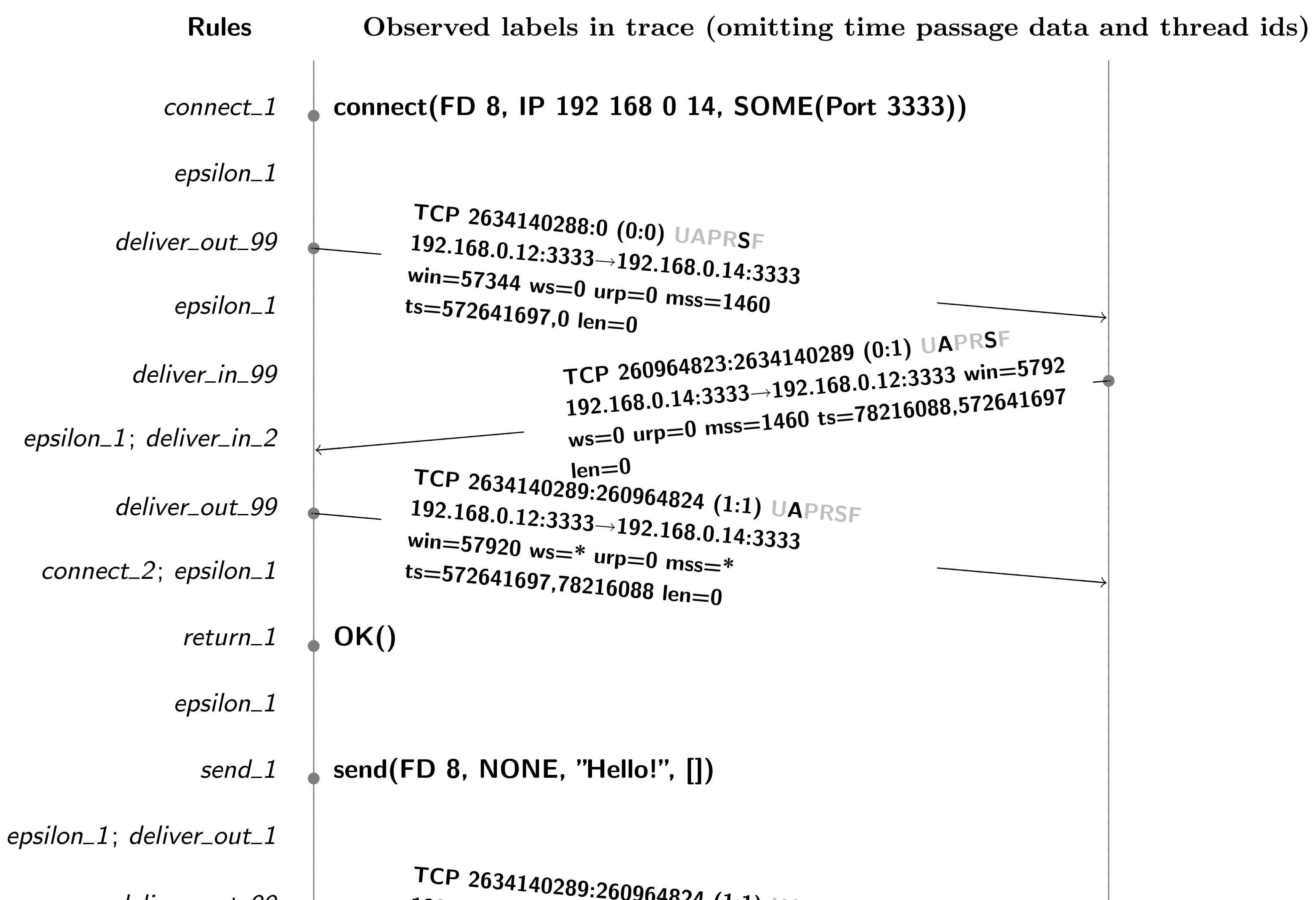
Tests

OCaml code that drives an instrumented network. Coverage:

- all three OSs
- exhaustive where we can get away with it
- aim to cover most of interesting things in the spec (rule coverage - ok) (code coverage - ?)

eg trace 1484: "$\mathrm{send}()$ – *for a non-blocking socket in state* ESTABLISHED(NO_DATA), *with a reduced send buffer that is almost full, attempt to send more data than there is space available.*"

Rules used for sample checked trace



Does it work?

UDP: 2526 (97.04%) of 2603 traces succeed (BSD, Linux, and WinXP).

TCP: 1004 (91.7%) of 1095 traces succeed (BSD).

(other OSs modelled and partially checked, but deferred for now)

Non-successes: test generation, HOL limits, a few outstanding spec problems.

Numbers only meaningful if coverage good. Of 194 rules:
142 covered, 32 resource limit, 20 not tested or not succeeded.

Did we find bugs?

Not really the point. But: Spec OS-dependent on 260 lines; 30 anomalies:

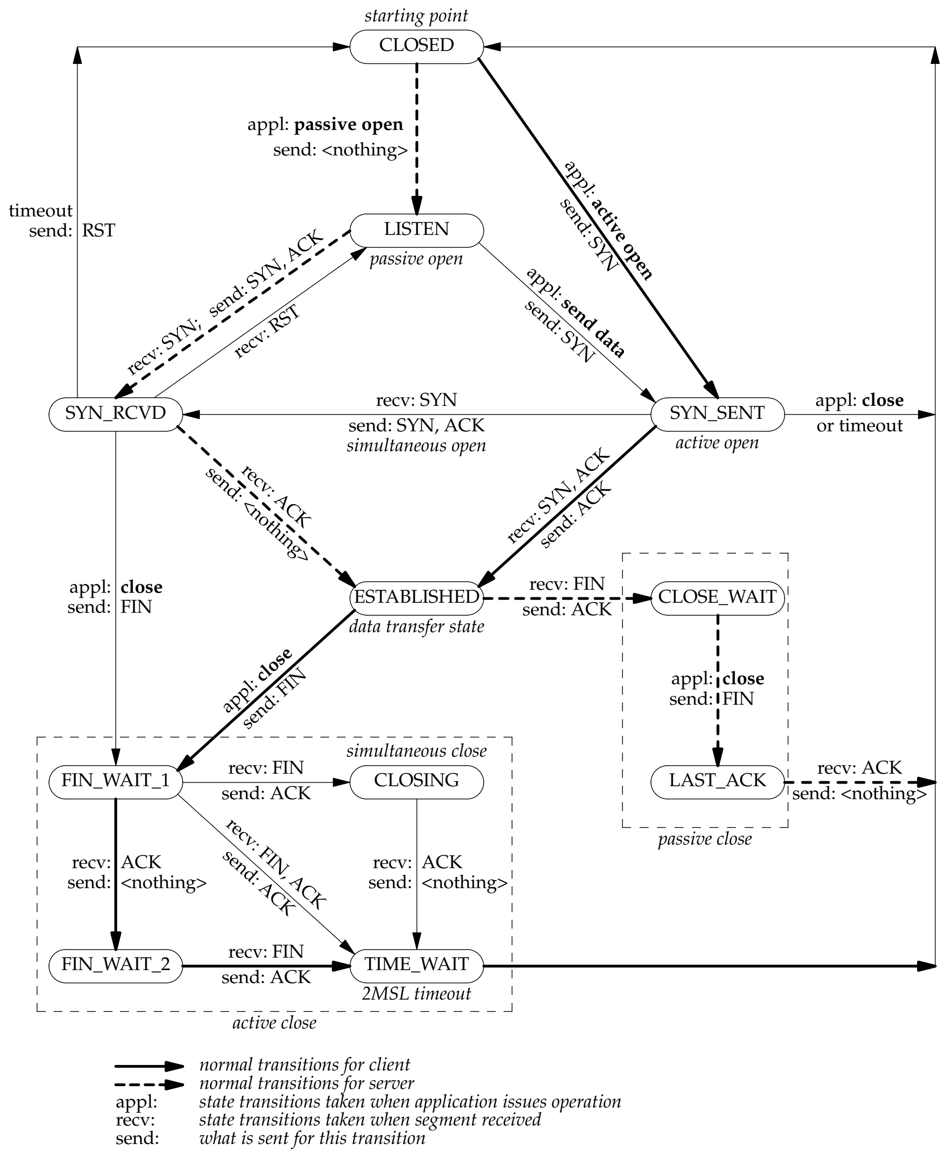
1. urgent pointer not updated in fastpath (so after 2GB, won't work for 2GB)
2. incorrect RTT estimate after repeated retransmission timeouts
3. `TCPSHAVERCVDFIN` wrong — so can `SIGURG` a closed connection
4. initial retransmit timer miscalculation
5. simultaneous open responds with ACK instead of SYN,ACK
6. receive window updated even for bad segment
7. shutdown state changes in pre-established states
8. (Linux) UDP connect with wildcard port
9. (Linux) sending options in a SYN,ACK that were not received in SYN

How the spec can be used

In different ways by different communities:

1. as reference documentation (right now)
2. for high-quality automated conformance testing of other protocol stacks (with more work);
3. for describing proposed changes to the protocols; and
4. as a basis for proof about executable descriptions of higher layers.

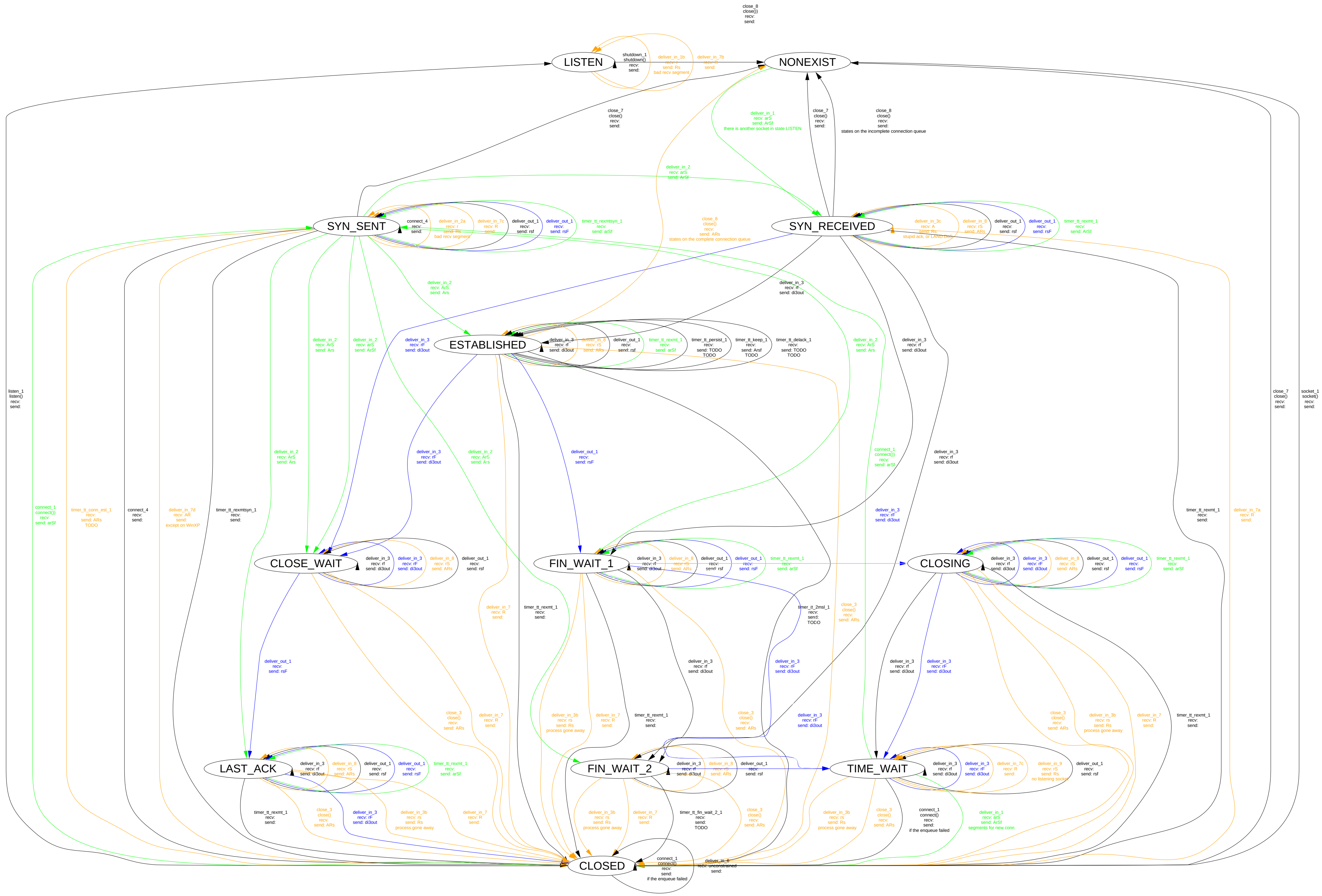
The TCP state diagram – as per Stevens



TCP state transition diagram.

Reprinted from *TCP/IP Illustrated, Volume 2: The Implementation*
by Gary R. Wright and W. Richard Stevens,

The TCP state diagram – a slightly better approximation



Part 1: Introduction

Part 2: Modelling Choices

Part 3: The Specification

Part 4: Validation

Part 5: What we have learned

Automated Testing

Automated testing from a specification — very powerful.

Not as much assurance as verification, but it scales.

On the design of new protocols

- *design for test:* protocol specifications should be written so that implementations can be tested directly against them.
- exposing internal nondeterminism would simplify testing
- specifying may reveal conceptual (un)clarity
- nail down the abstraction relation between the real system and the spec
- specify the API behaviour in addition to the wire behaviour
- modularise the spec (to ease future changes). NB: spec modularity does not have to force the same decomposition on the implementations
- design for refinement of the spec to an executable prototype

Conclusion

It is feasible to do this — to work with rigorous models of real systems, and to test the two match up.

Spec, techreport, and papers available online:

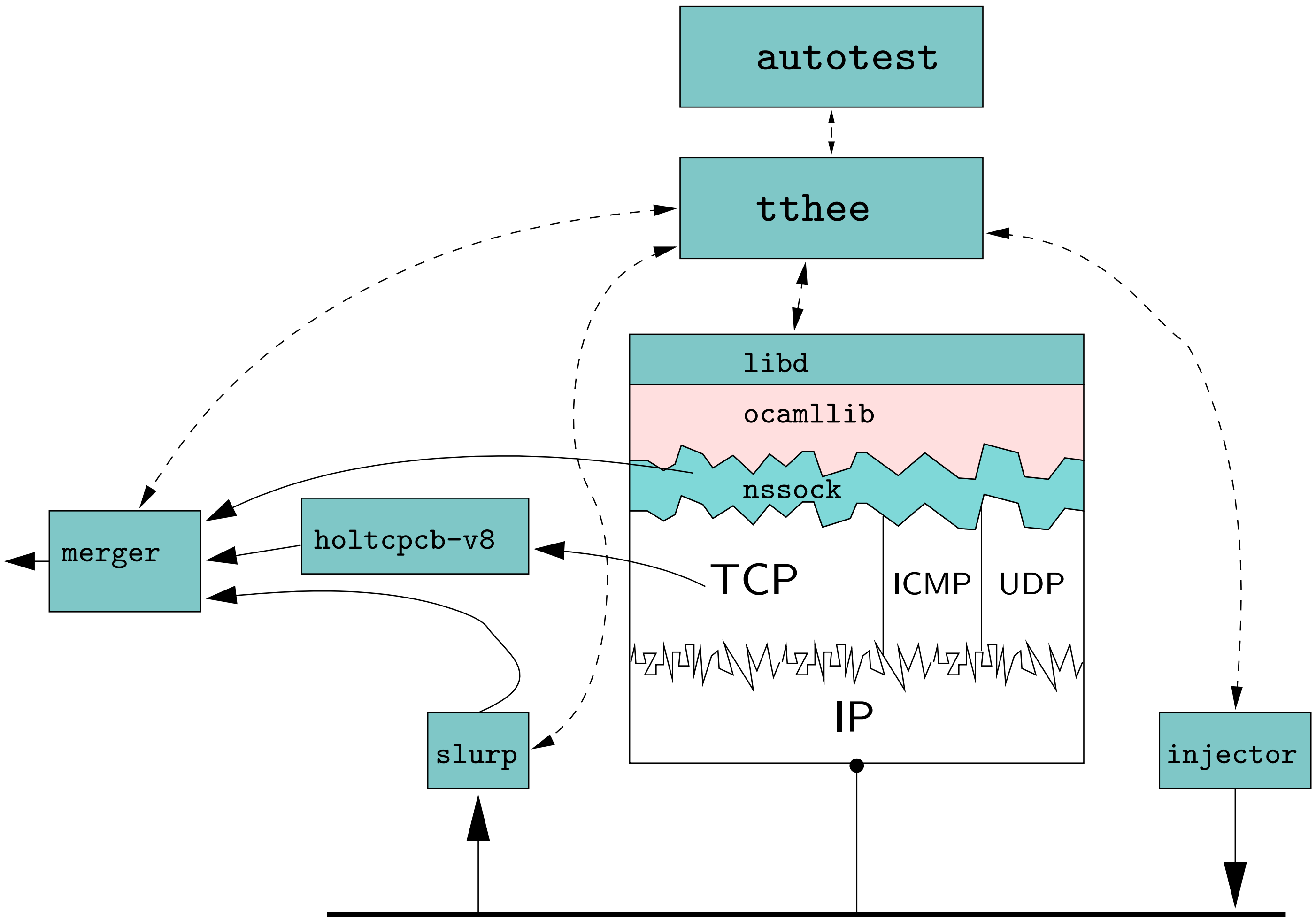
`google:"Netsem"`

`http://www.cl.cam.ac.uk/users/pes20/Netsem`

Feedback on content and accessibility very welcome.

The End

Trace generation infrastructure



Scale and Expertise

UDP (2000–2001): 2 man-years over 10 months (4 people)

TCP (2002–2005): 7 man-years over 30 months (6 people)

Result is 350 pages typeset (cf code size).

Not that much (and much was tool & idiom development, and forensic semantics). Contrast with the accumulated network protocol and sockets user investment...

Expertise with HOL not a problem for specifiers (days only). Taste and good idioms more important. Expertise is required for developing symbolic evaluator.