A rigorous approach to networking: TCP, from implementation to protocol to service

Tom Ridge[1] Michael Norrish[2] Peter Sewell[1]

[1]University of Cambridge [2]NICTA, Canberra

FM'08, Turku, May 2008

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

NetSem Project, 2000–2008

Tempting applications for Formal Methods — two kinds:

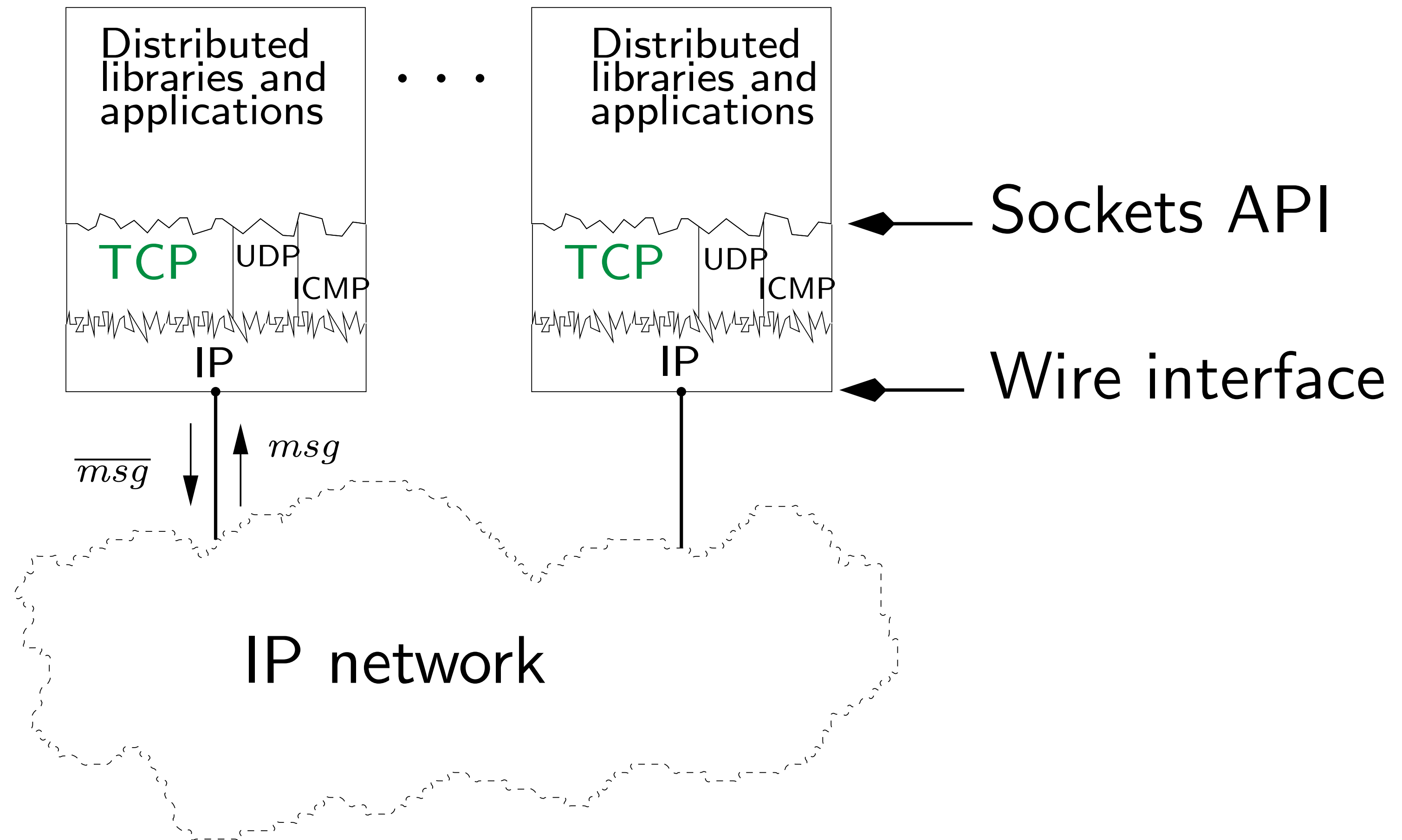
1. specific critical systems
2. key infrastructure abstractions
   - stable abstractions, about which other software changes
   - but complex — informal description is not enough

Here we focus on (2), specifically:

- TCP and Sockets (post-hoc protocol specification)
- SWIFT optically switched network (design-time specification)

TACS 2001, ESOP 2002, SIGOPS EW 2002, SIGCOMM 2005, ATS 2005, POPL 2006, ICNP 2006, FM 2008.

The Internet



IP: unreliable asynchronous small messages, routed to IP addresses e.g. 128.34.1.14.

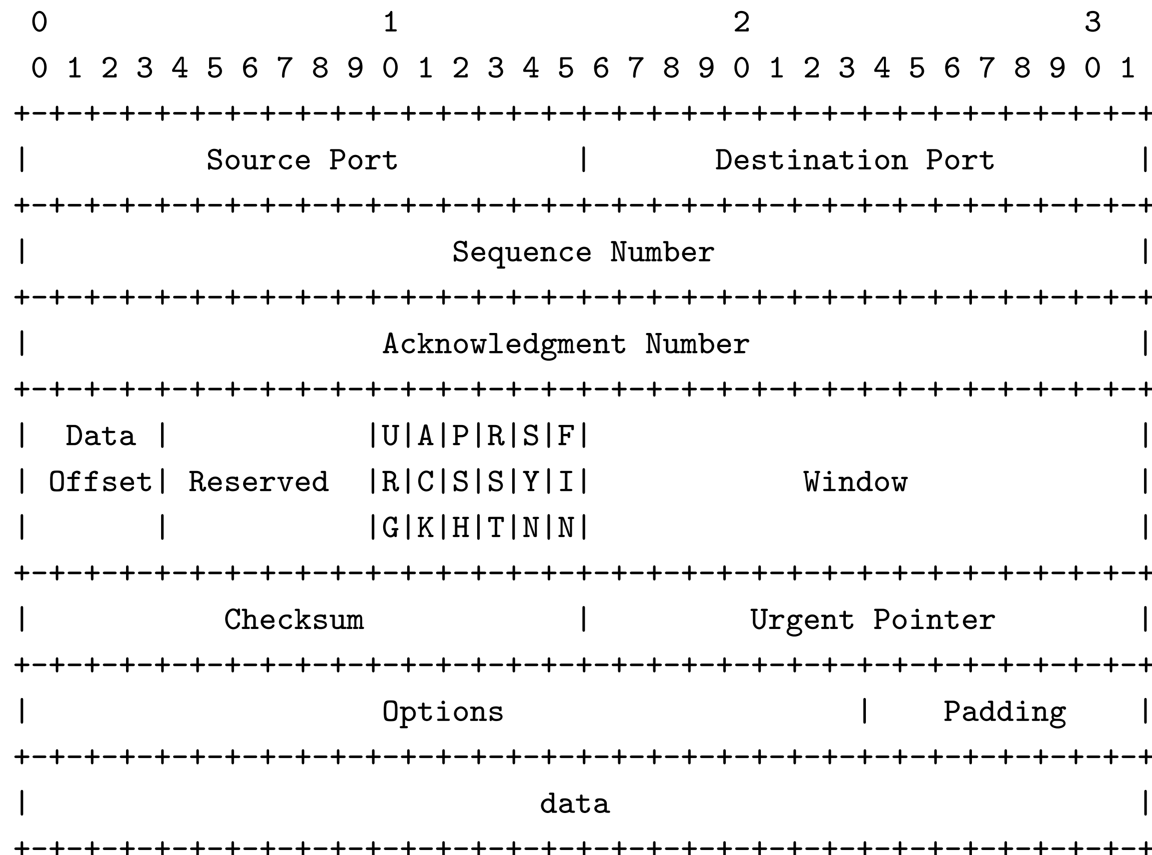
TCP: duplex byte streams, with retransmission, flow control, congestion control, DoS protection, MTU estimation, etc. Between IP/Port pairs, e.g. 128.34.1.14:80. Underlies web, email, etc.

What is TCP?

1: Defined by standards:

RFC793 TRANSMISSION CONTROL PROTOCOL, Sept. 1981

(and RFC 1122, 1323, 2414, 2581, 2582, 2988, 3522, 3782, ...)



and the behaviour of endpoints?

What is TCP, *really*?

2: Defined by the common implementations: BSD, MS, Linux, ...

They differ in many ways, but mostly interop ok.

This *de facto* standard is ultimately the code:

- C
- around 15 000 lines in BSD
- multi-threaded, time-dependent, entangled with OS
- optimised for performance, tweaked over time

Informal RFCs contain ambiguities and omissions . . .

. . . and have no clear relationship to the code. Crucially, they do not support conformance testing, or verification.

Our Goals

Specific:

1. Characterise precisely this *de facto* standard — what the behaviour of (some of) the deployed implementations really is, especially the failure semantics.

General:

- Understand how to deal rigorously with the behaviour of complex critical infrastructure:
  - working with the real protocols, not idealised models
  - both post-hoc and at design-time

What we did first

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

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

- mathematically rigorous – in the HOL proof assistant
- detailed and with broad coverage – many modelling choices
- readable – careful structuring, much annotation
- accurate – will come back to this

The Protocol-level Specification

The main part of the spec is the *host labelled transition system (LTS)* $h \xrightarrow{lbl} h'$ modelling a single endpoint, at a segment level of abstraction.

Distributed libraries and applications

$tid \cdot \mathrm{bind}(fd, is'_1, ps'_1)$

$\overline{tid \cdot v}$

Sockets API

TCP UDP ICMP

IP

Wire interface

Host LTS spec

$\overline{msg}$

msg

IP network

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

The Protocol-level Specification: Overall Structure

- Preamble, defining types and auxiliaries (host states, TCP segments,...). 125pp.
- Host transition rules:
  - Sockets API rules (148). 160pp.
  - Message processing rules (46). 75pp.

Not much shorter than the C code – but organised for clarity...

Experimental Semantics

Q: How could we ever have confidence such a thing does describe the behaviour of the impls?

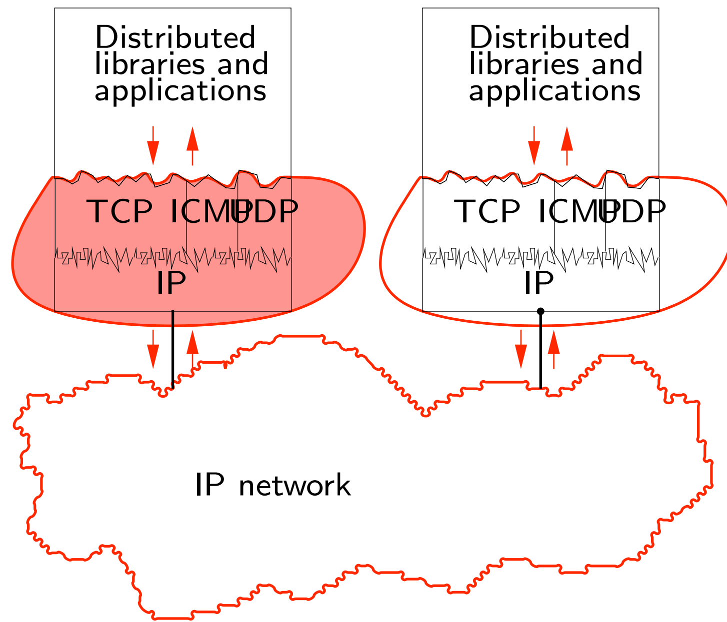
A: *experimental* (forensic?) semantics:

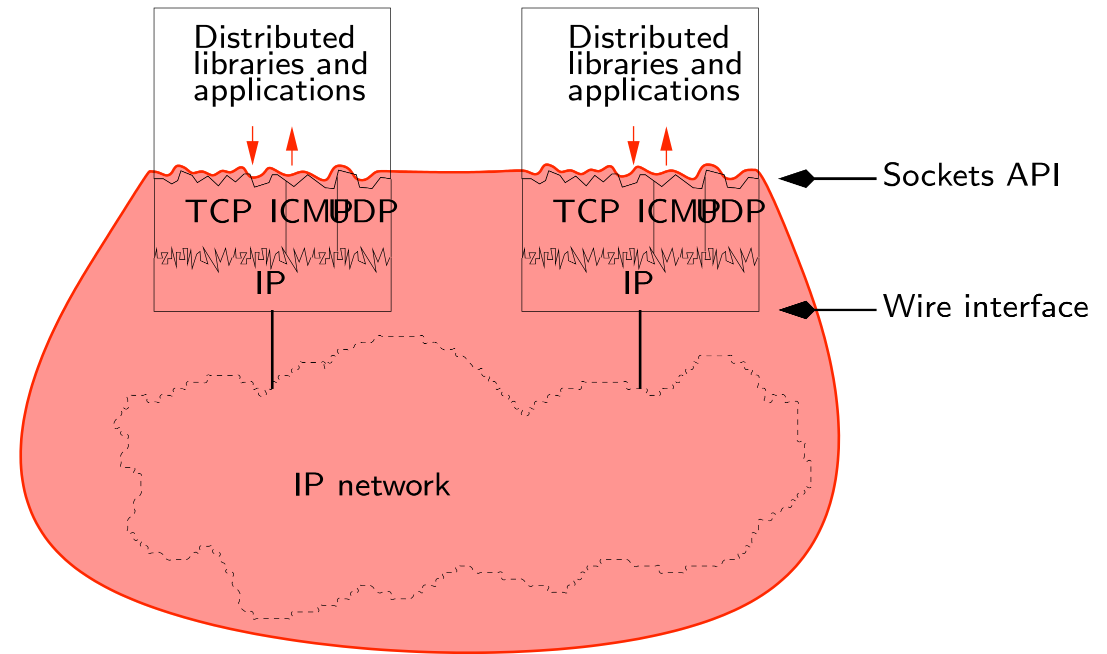
Establish tools for conformance testing between spec and impls

1. write draft spec
2. generate 3000+ implementation traces on a small network
3. test that those implementation traces are allowed by the spec, using a special-purpose symbolic model checker, within HOL
4. goto 1

Our new service-level TCP specification

The Service-level Specification





Endpoint specification
Segments
Low-level
How the protocol works

End-to-end specification
Streams
High-level
What service the protocol provides

The Service-level Specification

The service provided by TCP is essentially reliable end-to-end byte streams between Sockets API endpoints.

But also:

- the full socket API (`connect`, `listen`, etc.)
- hosts, threads, and network interfaces
- interaction with ICMP and UDP
- abandoned connections, unexpected socket closure, socket self-connection
- transient and persistent connection problems
- ...

The Service-level Specification

- wrote spec, by deleting previous protocol details and adding streams. Retain Sockets API behaviour.

  (host states & protocol rules much simpler, but overall size comparable)

  (used informal intuition w.r.t. invariants of low-level spec)

- define abstraction function (in HOL) from low- to high-level states and transitions

  (Key property: mapping the abstraction function over a protocol-level trace should give a service-level trace, so Sockets API behaviour is preserved.)

- validate abstraction function on (a sample of 30) real-world traces, using and extending previous symbolic checking infrastructure

Protocol-level host types

– the TCP control block :

```
tcpcb =⟨[ (* 44 fields*)
     (* timers *)
     tt_rexmt : (rexmtmode#num)timed option; (* retransmit timer *)
     (* see tcp_output.c:356ff for more info. *)
     tt_keep : () timed option; (* keepalive timer *)
     tt_2msl : () timed option; (* 2 * MSL TIME_WAIT timer *)
     tt_delack : () timed option; (* delayed ACK timer *)
     tt_conn_est : () timed option; (* connection-establishment timer *)
     tt_fin_wait_2 : () timed option; (* FIN_WAIT_2 timer *)
     t_idletime : stopwatch; (* time since last segment received *)

     (* flags, some corresponding to BSD TF_ flags *)
     tf_needfin : bool; (* used for app close in SYN_RECEIVED *)
     tf_shouldacknow : bool; (* output a segment urgently*)
     bsd_cantconnect : bool; (* connection establishment attempt has failed having sent a SYN – on BSD
                               this causes further connect() calls to fail *)

     (* send variables *)
     snd_una : tcp_seq_local; (* lowest unacknowledged sequence number *)
     snd_max : tcp_seq_local; (* highest sequence number sent; used to recognise retransmits *)
     snd_nxt : tcp_seq_local; (* next sequence number to send *)
```

Service-level host types

– the TCP control block :

tcpcb $=\langle\!\![$

(* timers *)

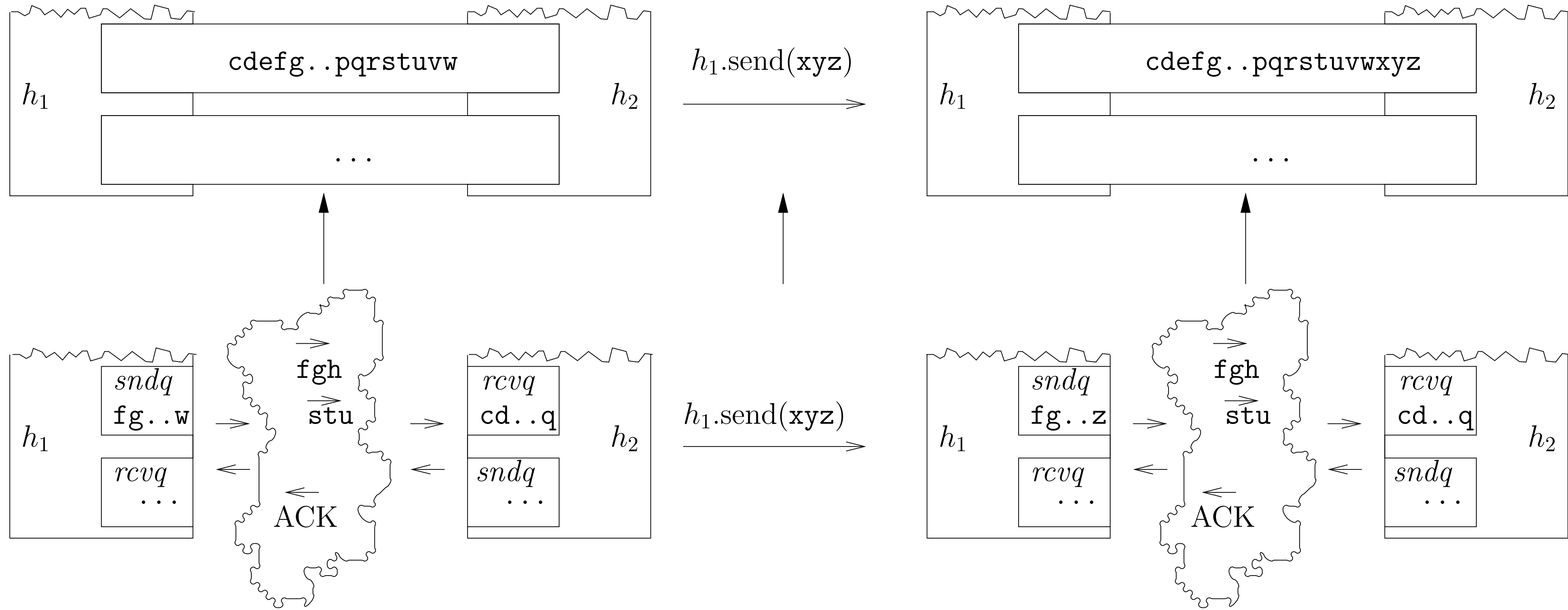
tt_keep : $()$ $timed$ option; (* keepalive timer *)

(* other *)

$t_softerror$: $error$ option (* current transient error; reported only if failure becomes permanent *)

$]\!\!\rangle$

The abstraction function



	↓ snd_una ↓ rcv_nxt
message	...abcdefghijklmnopqrstuvwxyz...
source $sndq$	fghijklmnopqrstuvw
destination $rcvq$	cdefghijklmnopq
$\mathrm{DROP}(rcv_nxt - snd_una)\,sndq$	rstuvw
stream	cdefghijklmnopqrstuvw

Service-level validation

- We wrote new tests to record behaviour at each end of a TCP connection.
- We checked each of the two endpoint traces independently against the protocol-level specification, producing *ground* protocol-level traces for each endpoint.
- We combined these to form a protocol-level trace of the network, and then applied the abstraction function.
- We checked the resulting *ground* service-level network trace, using a new special-purpose symbolic model checker in HOL.
- Extremely high assurance: the checker executes the specification within HOL, producing a machine checked proof of admissibility for each trace.

Results

- Several iterations, involving changes to the specifications and the abstraction function.
- In all, validated 30 traces against both specifications and the abstraction function (compared to 2000 traces for the protocol level).
- Currently no traces with packet loss, reordering, duplication and severe delay (although there is no technical barrier to doing so).
- A man year or so of further testing would give greater confidence.

  (Tiny amount of effort for industry, but unlikely to lead to new research insights)

Typical service-level specification error

1. TCP packets do not appear in the service level.
2. The service-level host output queue still exists, to carry ICMP and UDP messages.
3. Protocol-level action: a TCP packet moves from output queue to wire
4. Service-level: a "tau" action, i.e. nothing happens.
5. However... at the protocol-level there is an output queue timer to ensure messages are sent.
6. Sending a TCP packet resets the timer, which must be reflected at the service level (otherwise the service-level requires messages to be sent earlier than they should be).
7. So, the protocol-level transition must correspond to a service-level transition which resets the queue timer.

Conclusion (1 of 2)

- A formal, mechanized service-level specification of TCP, tackling the full details of the real-world protocol.
- The service-level is a precise statement of end-to-end correctness for the protocol level.
- The abstraction function explains the connection between the two specifications, and allows validation infrastructure to be lifted from one to the other.
- We believe the specification is appropriate for formal and informal reasoning about applications built above TCP, and about the service TCP provides to the Sockets layer.

Conclusion (2 of 2): No Verification

We eschewed traditional ‘full’ verification in favour of rigorous specifications from which we could build (verified) oracles to test against.

- not so much confidence
- but much less effort, and still very useful (esp. where multiple implementations of a spec. are the norm, and easy conformance testing is very valuable)

Challenge: understand how to do that in less bespoke fashion.

Challenge: get take-up by protocol designers

Challenge: verify (after all) the code/protocol/service relationships

The End