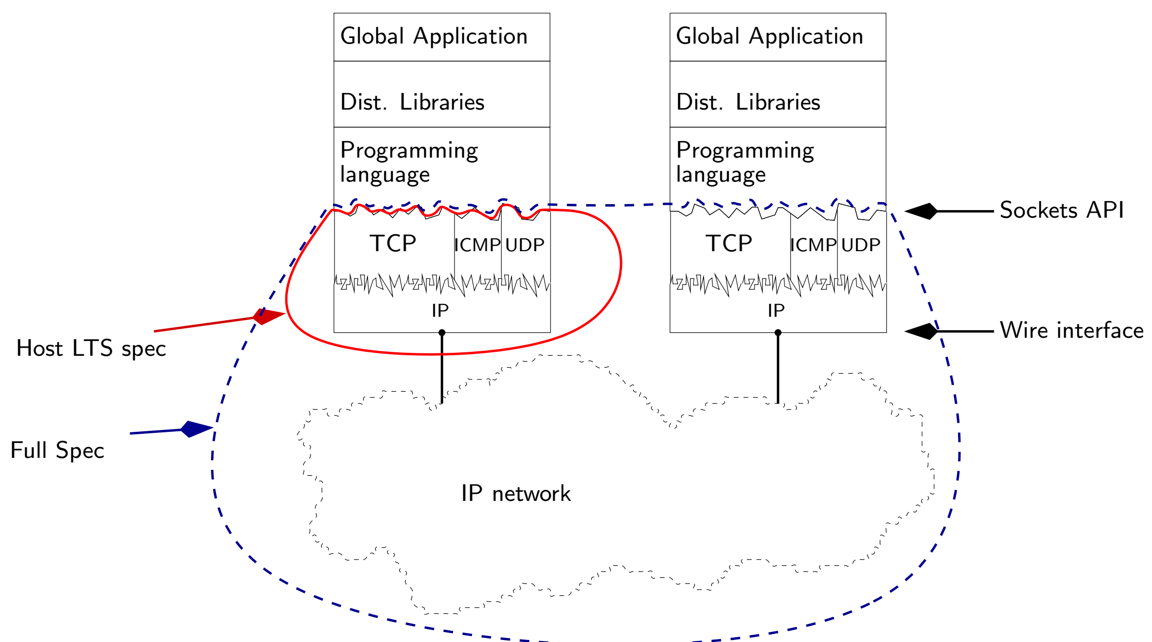
The TCP Specification: A Quick Introduction

Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, Keith Wansbrough

March 18, 2005

What is this specification? A mathematical model of TCP, UDP, relevant parts of ICMP, and the Sockets API, describing precisely what their behaviour is.

The main part of the model is the "host LTS", describing the possible interactions of a host OS: between program threads and host via calls and return of the socket API, and between host and network via message sends and receives. The network is modelled as a fuzzy cloud, abstracting from routing topology.



What behaviour is included? Just about everything — congestion control, timeouts, etc., and especially all the possible error cases. We simplify the API slightly, using a pure value-passing interface instead of the standard C pointer-passing interface.

What is the specification written in? Higher-order logic, using the HOL proof assistant. The main part of the model defines a labelled transition system over host states, with labels for each of the interactions mentioned above. It is defined by some 148 rules for the socket calls (5–10 for each interesting call) and some 46 rules for message send/receive and for internal behaviour. The whole thing is roughly 360 typeset pages — large, but still manageable.

How was the specification constructed, and why should one have any confidence in it? The first drafts were based on the RFCs, POSIX standard, Steven's texts, the BSD and Linux source code, and some ad-hoc testing. Using a test network and various instrumentation we generated several thousand real-world traces, of three widely-deployed implementations: FreeBSD 4.6–RELEASE, Linux 2.4.20–8, and Windows XP

Professional SP1, chosen to give good coverage of possible behaviours. A special-purpose symbolic model checker, written above HOL, lets us check that the specification includes these traces. This *experimental semantics* process is essentially complete for the TCP behaviour for BSD, and for the UDP behaviour for all three OS versions; some differences in the TCP behaviour for the other two OSs have been identified.

How does the specification differ from an implementation?

1. It is written to be as clear as possible, in contrast to the code, which was written to execute the common cases fast and includes many historical artifacts. It has been extensively annotated to make it readable by non-HOL-experts.
2. It is highly nondeterministic: at many points the specification includes all reasonable behaviours, whereas any particular implementation would make some particular choice.

How does the specification differ from the RFCs and texts? It is completely precise, covering all cases, and validated to ensure that it is a correct description of the implementations, modulo a few minor outstanding issues.

What about differences between implementations? The specification explicitly models differences between the behaviour of our three OS versions.

Have you found any bugs? This is a *post-hoc* specification: we are primarily trying to describe what is out there, which (however strange) is a *de facto* standard which cannot readily be changed. Our validation therefore aims to correct the specification to include all real-world behaviour, not to correct those implementations. But that said — yes, some observed behaviour is clearly wrong.

How might this be useful? As:

1. documentation for users of the Sockets API;
2. documentation for implementors of TCP/UDP;
3. a basis for specifications and proofs about higher-level distributed infrastructure layers above TCP/UDP, including interesting distributed algorithms,
4. a basis for design of more abstract calculi and programming languages with well-defined failure semantics, with clear relationships to the real-world behaviour;
5. a proof-of-feasibility that designers of new network protocols (and perhaps of TCP variants, e.g. with different congestion control) could use this style of specification *in the design phase*, leading to cleaner designs.
6. a demonstration that this kind of automated validation can be used for conformance testing of behaviourally-complex software, leading to better implementations;
7. a source of user feedback for automated reasoning research; and, generally,
8. a demonstration of how it can be feasible to use rigorous semantic methods for real systems.

Note that we are not proposing to prove any significant correctness properties of TCP (as described by the specification). That would be an enormous proof effort, and perhaps out of proportion to what one would be likely to discover.

For many purposes it would be more useful to have a higher-level (end-to-end, stream-level) TCP specification, with a model in terms of bidirectional streams and additional data rather than the current (endpoint, segment-level) specification that describes all the TCP segments on the wire. It should be reasonably straightforward to construct such a stream-level specification from the complete segment-level specification.

How to Read the Specification: Quick Start

The best place to start is with Volume 1, especially Section 2 *Modelling* and Section 4 *The Specification – Introduction.* Then one should be able to browse the specification itself in Volume 2, starting by browsing the rules for a few socket calls, in Chapter 15, e.g. for socket(), bind(), connect() etc. One can then look at the rules for network input and output, Chapter 21, and at the protocol rules for either UDP (Chapter 19) or TCP (Chapter 16 for input, Chapter 17 for output, and Chapter 18 for the TCP timers). The auxiliary definitions and types in the earlier part of the specification, especially those in `params` and `auxFns`, may be best looked at as one comes across usages in the rules, rather than attempting to read the specification from beginning to end. The text annotating the rules is written to be used as reference material, so one should be able to go directly to (say) a socket call without having to read all the preceding definitions.

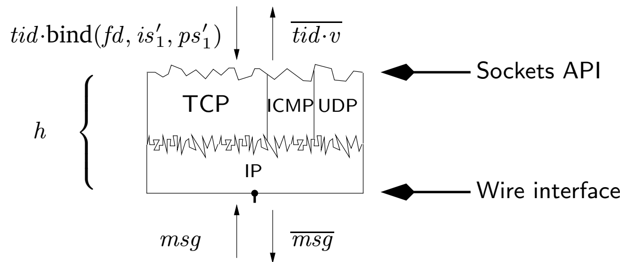
The main part of the specification is the *host labelled transition system*, or host LTS for short. It defines a transition relation

$$h \xrightarrow{lbl} h'$$

where h and h' are host states, modelling the relevant parts of the OS and network card of a single machine, and *lbl* is an interaction on either the socket API or wire interface. Typical labels are:

msg	for the host receiving a datagram msg from the network
$\overline{msg}$	for the host sending a datagram msg to the network
$tid \cdot \mathsf{bind}(fd, is_1', ps_1')$	for a bind() call being made to the sockets API by thread tid, with arguments (fd, is_1', ps') for the file descriptor, IP address, and port
$\overline{tid \cdot v}$	for value v being returned to thread tid by the sockets API
τ	for an internal transition by the host, e.g. for a datagram being taken from the host's input queue and processed, possibly enqueuing other datagrams for output

A host h and some transitions (but not the destination hosts h') are shown below.



In slightly more detail, a host h is a record of various data — a value of the HOL type host, defined as follows:

```
host =⟨[ arch : arch; (* OS version *)
        privs : bool; (* whether process has privilege *)
        ifds : ifid ↦ ifd; (* network interfaces *)
        rttab : routing_table; (* routing table *)
        ts : tid ↦ hostThreadState timed; (* host view of each thread state *)
        files : fid ↦ file; (* open file descriptions *)
        socks : sid ↦ 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 *)
$\rrbracket$

In turn, a socket is a record containing local and remote IP and ports, some other data, and the protcol-specific information (which for TCP includes an analogue of the TCP Control Block used by implementations):

is_1 : ip option;
ps_1 : port option;
is_2 : ip option;
ps_2 : port option;
pr : protocol_info
...

The transition relation $h \xrightarrow{lbl} h'$ is defined by some 195 rules, in `TCP1_hostLTSScript.sml`. Each rule has a name, e.g. *bind_5*, *deliver_in_1* etc., a protocol, e.g. RP_TCP for TCP-specific rules, and a category, e.g. FAST SUCCEED for a sockets API rule that cannot block and returns a value rather than an error.

A sample rule (one of the simplest) is shown below.

bind_5 <u>all: fast fail</u> **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**

$$h \ \llparenthesis ts := ts \oplus (tid \mapsto (\textsc{Run})_d) \rrparenthesis \quad \xrightarrow{tid \cdot \mathsf{bind}(fd, is_1, ps_1)} \quad h \ \llparenthesis ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL EINVAL}))_{\text{sched_timer}}) \rrparenthesis$$

$fd \ \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\text{FT_}\textsc{Socket}(sid), ff) \wedge$
$h.socks[sid] = sock \wedge$
$(sock.ps_1 \neq * \vee$
$(\text{bsd_arch} \ \ h.arch \wedge sock.pr = \text{TCP_PROTO}(tcp_sock) \wedge$
$\quad (sock.cantsndmore \vee$
$\quad tcp_sock.cb.bsd_cantconnect)))$

Description From thread tid, which is in the RUN state, a $\mathsf{bind}(fd, is_1, ps_1)$ call is made where fd refers to a socket $sock$. The socket already has a local port binding: $sock.ps_1 \neq *$, and rebinding is not supported.

A $tid \cdot \mathsf{bind}(fd, is_1, ps_1)$ transition is made, leaving the thread state RET(FAIL EINVAL).

Variations

FreeBSD	This rule also applies if fd refers to a TCP socket which is either shut down for writing or has its *bsd_cantconnect* flag set.

This is one of 7 rules for bind(). It deals with the case where a thread tid calls $\mathsf{bind}(fd, is_1', ps_1')$ for a socket

referenced by the file descriptor fd that already has its local port bound; the error EINVAL will be returned to the thread.

The variables in the rule (h, fd, tid etc.) are all implicitly universally quantified, so this rule permits transitions $h \xrightarrow{lbl} h'$ to happen for any h, lbl and h' that can match the structure in the displayed transition and also satisfy the sidecondition below.

In the host on the left of the transition, the thread state map ts maps thread id tid to $(\text{RUN})_d$, indicating that the thread is running (in particular, it is not currently engaged in a socket call). In the host on the right of the transition, that thread is mapped to $(\text{RET}(\text{FAIL EINVAL}))_{\text{sched_timer}}$, indicating that within time sched_timer the failure EINVAL should be returned to the thread (all returns are handled by a single rule *return_1*, which generates labels $\overline{tid \cdot v}$).

The sidecondition is a conjuction of 5 clauses. The first ensures that the file descriptor fd is in the host's file descriptor map $h.fds$. The second says that fid is the file identifier for this file descriptor. The third says that this fid is mapped by the host's files map $h.files$ to $\text{FILE}(\text{FT_SOCKET}(\mathsf{sid}), ff)$, i.e.to a socket identifier sid and file flags ff. The fourth simply picks out the socket structure *sock* associated with the socket id sid. The fifth says that the local port of the socket with that sid is not equal to the wildcard $*$, i.e.that this socket has already got its local port bound, or some BSD-specific condition holds.

The syntax is the HOL higher-order logic, which is essentially standard logic, though the specification makes extensive use of records and finite maps. Records in HOL are written e.g. $\langle\!| label1 := expression1, label2 := expression2 |\!\rangle$, and if h is a record then $h \langle\!| ts := e |\!\rangle$ is the record h with field ts overridden to e. Note that field names (e.g. *label1* and the ts on the left of a := in the rule) are distinct from variables (e.g. the other occurrences of ts in the rule). Record projection is written e.g. $h.fds$ for the fds field of h. The syntax $ts \oplus (tid \mapsto e)$ denotes the finite map ts overridden with tid mapped to e, and $h.fds[fd]$ is the image of fd in the finite map $h.fds$.

The rule as shown is automatically typeset from the HOL source (as is the rest of the specification). For example $h \langle\!| ts := ts \oplus (tid \mapsto (\text{RUN})_d) |\!\rangle$ is the typeset version of the HOL source below.

```
h with <| ts := FUPDATE ts (tid,Timed(Run,d)) |>
```