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 Computer Laboratory †NICTA, Canberra

https://i o co do o b .os ar.com/users/pes20/Netsem

Abstract

Network protocols are hard to implement correctly. Despite the existence of RFCs and other standards, implementations often have subtle differences and bugs. One reason for this is that the specifications are typically informal, and hence inevitably contain ambiguities. Conformance testing against such specifications is challenging.

In this paper we present a practical technique for rigorous protocol specification that supports specification-based testing. We have applied it to TCP, UDP, and the Sockets API, developing a detailed 'post-hoc' specification that accurately reflects the behaviour of several existing implementations (FreeBSD 4.6, Linux 2.4.20-8, and Windows XP SP1). The development process uncovered a number of differences between and infelicities in these implementations.

Our experience shows for the first time that rigorous specification is feasible for protocols as complex as TCP. We argue that the technique is also applicable 'pre-hoc', in the design phase of new protocols. We discuss how such a design-for-test approach should influence protocol development, leading to protocol specifications that are both unambiguous and clear, and to high-quality implementations that can be tested directly against those specifications.

Categories and Subject Descriptors C.2.2 [**Computer-Communications Networks**]: Network Protocols; C.2.6 [**Computer-Communications Networks**]: Internetworking—*Standards (e.g., TCP/IP)*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms Documentation, Design, Standardization, Theory, Verification.

Keywords Network Protocols, TCP/IP, Sockets, API, Specification, Conformance Testing, Higher-order Logic, HOL, Operational Semantics

SIGCOMM'05, August 21–26, 2005, Philadelphia, Pennsylvania, USA.

1. INTRODUCTION

Background Network protocols such as TCP are typically described using informal prose and pseudocode to characterise the behaviour of the systems involved. This use of informal natural-language descriptions, coupled with an emphasis on working code and interoperability, has served the field well. It made the early specifications accessible, letting protocols evolve as required, and discouraged heavy over-specification disconnected from implementation.

There is a downside, however. Protocols such as TCP are hard to implement correctly, as one can see from the many subtle differences between implementations [30], [RFC2525]; they are also hard to use correctly. The protocols are complex, both intrinsically and due to the accumulation of historical artifacts. Informal descriptions, despite the care taken by their authors, are inevitably somewhat ambiguous and incomplete, leaving room for inconsistent interpretations. Moreover, conformance testing is challenging. There is in fact no clear way to define what it means for an implementation of TCP to conform to the RFC standards, let alone to test whether it does. (Of course, there are many ways in which an implementation might obviously *not* conform.)

Ideally we would have the best of both worlds: protocol descriptions that are simultaneously:

1. clear, accessible to a broad community and easy to modify;
2. unambiguous, characterising exactly what behaviour is specified;
3. sufficiently loose, characterising exactly what is *not* specified, and hence what is left to implementors (especially, permitting high-performance implementations without over-constraining their structure); and
4. directly usable as a basis for conformance testing, not read-and-forget documents.

Contribution We demonstrate, for the first time, a practical technique for rigorous protocol specification that makes this ideal attainable for protocols as complex as TCP. We describe specification idioms that are rich enough to express the subtleties of TCP endpoint behaviour and that scale to the full protocol, all while remaining readable. We also describe novel tools for automated conformance testing between specifications and real-world implementations.

To develop the technique, and to establish its feasibility, we have produced a post-hoc specification of existing protocols: a mathematically rigorous and experimentally-validated characterisation of the behaviour of TCP, UDP, and the Sockets API, as implemented in practice.

The resulting specification may be useful in its own right in several ways. It has been extensively annotated to make it usable as a reference for TCP/IP stack implementors and Sockets API users, supplementing the existing informal standards and texts. It can also provide a basis for high-fidelity conformance testing of future implementations, and a basis for design (and conceivably formal proof) of higher-level communication layers.

Perhaps more significantly, the work demonstrates that it would be feasible to carry out similar rigorous specification work for new protocols, in a tolerably light-weight style, both at design-time and during standardisation. We believe the increased clarity and precision over informal specifications, and the possibility of automated specification-based testing, would make this very much worthwhile, leading to clearer protocol designs and higher-quality implementations. We discuss some simple ways in which protocols could be designed to make testing computationally straightforward.

Approach: The Specification Language A reasonable specification of TCP must be nondeterministic: it must be a *loose* specification in various ways, e.g. to allow TCP options to be chosen or not, to allow variations in initial window sizes and initial sequence numbers, and so forth. Moreover, it must admit the variations in behaviour that can arise from OS scheduling, message processing delays, timer variations, etc. It can not therefore be written directly in a conventional programming language, and is quite different from a reference implementation. Nor, however, can it be expressed in any simple logic or calculus: the protocol endpoints have complex state-spaces, with various queues and lists, timing properties, and extensive mod-2^{32} arithmetic.

Accordingly, we write our specification as an operational semantics definition in higher-order logic, mechanized using the HOL system [14, 17]. Higher-order logic is similar to conventional first-order logic (or predicate calculus) with the normal logical operations, quantifiers, etc., but with the addition of a rich type structure (including numeric types, lists, and functions) and the ability to quantify over any type. It lets one write more-or-less arbitrary mathematics idiomatically.

HOL is a system for manipulating higher-order logic definitions, type-checking them and performing proof. The system provides the programmer with a variety of decision procedures and scriptable tactics. HOL is not a fully automatic theorem-prover or model checker, as higher-order logic is not decidable, but its programmability allows the development of standalone tools, tailored to particular domains. Machine-processed mathematics in a system such as HOL, in a well-defined logic, is the most rigorous form of definition currently possible.

The use of higher-order logic may be unfamiliar at first sight but should not present any real difficulty in understanding the specification. Our experience is that smart undergraduate students, previously unfamiliar with HOL, can quickly get up to speed, making useful contributions within a week or two; most of that time is taken in understanding the protocols rather than higher-order logic. We give a brief introduction to HOL in §3; previous exposure to it is not needed to read this paper.

Approach: Conformance Checking To relate such a specification to implementations we have had to develop new verification techniques. A typical implementation of TCP/IP is a complex artifact, with many thousands of lines of multi-threaded C code, interrupt-driven timers, function pointers, and various optimizations, e.g. for fast-path processing. Relating the two by conventional model-checking or proof-based verification techniques would be very challenging. Machine-checked proof of code on this scale seems beyond the current state of the art. Model-checking could be used to check some simple properties of implementations, but the complexity of the state space suggests it would be hard to get good coverage.

We therefore take a specification-based testing approach. We have written a special-purpose checker which performs symbolic evaluation of the specification with respect to captured real-world traces, maintaining sets of constraints as it works along a trace, simplifying them (and using various decision procedures) as new information becomes available, and backtracking if required. The checker is itself above HOL, and so its results are guaranteed correct (modulo only the possibility of errors in the small HOL kernel): checking each trace essentially involves an automatically-generated and machine-checked proof that that trace is accepted by the specification.

The flexibility of HOL lets us write the specification declaratively, in as clear an idiom as we can, decoupled from these algorithmic details of conformance checking. The structure of the specification is also unconstrained by the algorithmic issues of efficient protocol implementation: it can be optimized for clarity, not performance, but still be checked against production implementations.

Approach: Experimental Semantics We began writing a rigorous specification of TCP by trying to restate the relevant RFC and POSIX standards mathematically. Unfortunately, those standards omit many important aspects of behaviour, and the common implementations do not conform to everything that they do prescribe. One would have to design the missing aspects, and the result would be hard to relate to current practice.

We therefore took seriously the fact that for many purposes TCP is defined by the *de facto* standard of the common implementations — intended differences, bugs, and all. Our specification aims to capture the behaviour of three widely-deployed implementations: FreeBSD 4.6–RELEASE, Linux 2.4.20–8, and Windows XP Professional SP1. Behavioural differences between the three are made explicit by parameterisation on the OS version. (These clearly do not cover the behaviour of all important implementations, but they are an indicative sample.)

The first drafts of the specification were based on the RFCs (especially 768, 791, 792, 793, 1122, 1323, 2414, 2581, 2582, 2988, 3522, and 3782), POSIX standard [18], Stevens's texts [37, 40, 38], BSD and Linux source code, and *ad hoc* tests. Such work is error-prone; one should have little confidence in the result without some form of validation. We therefore used our symbolic evaluation techniques to validate the specification directly against the implementations (as opposed to the more common validation of an implementation against a specification). We wrote tools to generate several thousand traces from the implementations running on an instrumented test network, chosen to give as broad coverage as we could, and used the checker to ensure that the specification does admit those traces. Validation is computationally intensive, so for reasonable performance it was necessary to distribute the task over a bank of machines (currently around 100 processors).

Developing the specification has been an iterative *experimental semantics* process, in which we use the trace checker to identify errors in the specification, inadequacies in the symbolic evaluator, and problems in the trace generation process, repeating until validation succeeds.

This work builds on our earlier semantics for UDP, first in informal mathematics [33, 34] and then in HOL [39, 27].

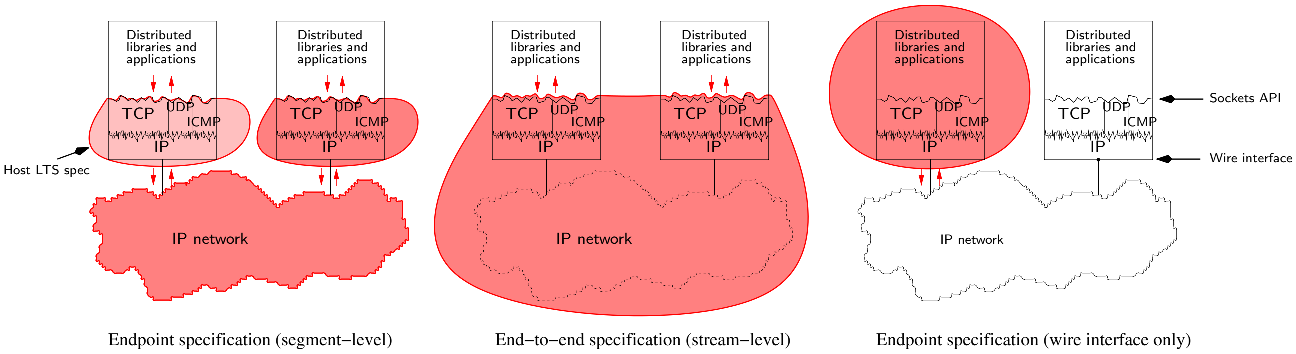


Figure 1: Modelling: where to cut.

The Specification The specification is available online, together with a full description of the project [7, 8]. As discussed above, it is fully *rigorous*. It is *detailed*, with almost all important aspects of the real-world communications at the level of individual TCP segments and UDP datagrams, with timing, and with congestion control. It abstracts from the internals of IP. It has broad *coverage*, dealing with the behaviour of a host for arbitrary incoming messages and Sockets API call sequences, not just some well-behaved usage — one of our main goals was to characterise the failure semantics under network loss, duplication and reordering, API errors, and malicious attack. It is also remarkably *accurate*, satisfying almost all the test traces.

We cannot, of course, claim total accuracy. Hosts are (ignoring memory limits) infinite-state systems; our generated traces surely do not explore all the interesting behaviour; and a few traces are still not successfully checked. Moreover, while for UDP and the Sockets API we dealt with all three OS versions in depth, for TCP we focussed primarily on the BSD version, identifying only some differences with the other two. Nonetheless, our automated validation process is, by the standards of normal software development, an extremely demanding test.

Overview We begin in §2 by discussing some of our key choices: exactly what our specification should model and how it should be expressed. The specification itself is outlined in §3. In §4 we describe our experimental validation process, with details of test generation and checking. §5 and §6 give the current validation status of the specification and highlight some of the anomalies we have discovered in the implementations. In §7 we discuss how what we have learnt can be applied during future protocol design. Finally, §8 discusses related work, and we conclude in §9.

2. MODELLING

For the specification to be useful, and for experimental validation to be possible, there must be a clear (though necessarily informal) relationship between certain events in the real system and events in the specification. We have to choose what system events are taken as observable, what is abstracted from them, and how to restrict the system to a manageable domain. These choices become embodied in the validation infrastructure, as instrumentation of the observable events and calculation of their abstract views.

Where to cut There are five main options for where to cut the system to pick out events, three of which are shown in Fig. 1. In each part of the figure the shaded areas indicate the part of the system covered by the model (which can abstract freely from the interior structure of these parts) and the short arrows indicate the specified interactions, between the modelled part and the remainder.

An *endpoint* specification, shown on the left, deals with events at the network interface and Sockets API of a single machine, but abstracts from the implementation details within the network stack. For TCP the obvious wire interface events are at the level of individual TCP segments sent or received.

An *end-to-end* specification, shown in the middle, describes the end-to-end behaviour of the network as observed by users of the Sockets API on different machines, abstracting from what occurs on the wire. For TCP such a specification could model connections roughly as a pair of streams of data, together with additional data capturing the failure behaviours, connection shutdown, etc.

A *wire-interface-only endpoint* specification, shown on the right, would specify the legal TCP segments sent by a single host irrespective of whatever occurs at the API.

One could also think of a *network interior* specification, characterising the possible traffic at a point inside the IP network, of interest for network monitoring, or a *pure transport-protocol* specification, defining the behaviour of just the TCP part of a TCP/IP stack with events at the Sockets API and (OS-internal) IP interfaces.

All would be useful. We chose to develop a segment-level endpoint specification, for three main reasons. Firstly, we considered it essential to capture the behaviour at the Sockets API. Focussing exclusively on the wire protocol would be reasonable if there truly were many APIs in common use, but in practice the Sockets API is also a *de facto* standard, with its behaviour of key interest to a large body of developers. Ambiguities, errors, and implementation differences here are often just as important as for the wire protocol. Secondly, the segment-level specification has a straightforward model of network failure, essentially with individual segments either being delivered correctly or not; the observable effects of network failure in an end-to-end model would be far more clearly characterised as a corollary of this than directly. Thirdly, it seemed likely that automated validation would be most straightforward for an endpoint model: by observing interactions as close to a host as possible (on the Sockets and wire interfaces) we minimise the amount of nondeterminism in the system and maximise the amount of information our instrumentation can capture.

The form of the specification The main part of the specification (the pale shaded region of Fig. 1) is the *host labelled transition system*, or *host LTS*, describing the possible interactions of a host OS: between program threads and host via calls and returns of the

Sockets API, and between host and network via message sends and receives.

The host LTS defines a transition relation

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

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

- msg for the host receiving a datagram msg from the network;
- $\overline{msg}$ for the host sending a datagram msg to the network;
- $tid{\cdot}\mathrm{bind}(fd, is_1, ps_1)$ for a bind() call being made to the Sockets API by thread tid, with arguments (fd, is_1, ps_1) 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; and
- dur for time dur passing.

The transition relation 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. Each rule has a name, e.g. *bind_5*, *deliver_in_1* etc., and various attributes. These rules form the main part of the specification.

The host LTS can be combined with a model of the IP network, e.g. abstracting from routing topology but allowing message delay, reordering, and loss, to give a full specification. In turn, that specification can be used together with a semantic description of a programming language to give a model of complete systems: an IP network; many hosts, each with their TCP/IP protocol stack; and executable code on each host making Sockets API calls. Compton [11] has demonstrated fully formal reasoning about executable OCaml code above our earlier UDP specification, using the Isabelle proof assistant [19].

Network interface issues The network interface events $\overline{msg}$ and msg are the transmission and reception of UDP datagrams, ICMP datagrams, and TCP segments. We abstract from IP, omitting the IP header data except for source and destination addresses, protocol, and payload length. We also abstract from IP fragmentation, leaving our test instrumentation to perform IP reassembly.

Given these abstractions, the model covers unrestricted wire interface behaviour. It describes the effect on a host of arbitrary incoming UDP and ICMP datagrams and TCP segments, not just of the incoming data that could be sent by a 'well-behaved' protocol stack. This is important, both because 'well-behaved' is not well-defined, and because a good specification should describe host behaviour in response to malicious attack as well as to loss.

Cutting at the wire interface means that our specification models the behaviour of the entire protocol stack and also the network interface hardware. Our abstraction from IP, however, means that only very limited aspects of the lower levels need be dealt with explicitly. For example, a model host has queues of input and output messages; each queue models the combination of buffering in the protocol stack and in the network interface.

Sockets interface issues The Sockets API is used for a variety of protocols. Our model covers only the TCP and UDP usage, for `SOCK_STREAM` and `SOCK_DGRAM` sockets respectively. It covers almost anything an application might do with such sockets, including the relevant `ioctl()` and `fcntl()` calls and support for TCP urgent data. Just as for the wire interface, we do not impose any restrictions on sequences of socket calls, though in reality most applications use the API only in limited idioms.

The Sockets API is not independent of the rest of the operating system: it is intertwined with the use of file descriptors, IO, threads, processes, and signals. Modelling the full behaviour of all of these would have been prohibitive, so we have had to select a manageable part that nonetheless has broad enough coverage for the model to be useful. The model deals only with a single process, but with multiple threads, so concurrent Sockets API calls are included. It deals with file descriptors, file flags, etc., with both blocking and non-blocking calls, and with `pselect()`. The `poll()` call is omitted. Signals are not modelled, except that blocking calls may nondeterministically return `EINTR`.

The Sockets API is a C language interface, with much use of pointer passing, of moderately complex C structures, of byte-order conversions, and of casts. While it is important to understand these details for programming above the C interface, they are orthogonal to the network behaviour. Moreover, a model that is low-level enough to express them would have to explicitly model at least pointers and the application address space, adding much complexity. Accordingly, we abstract from these details altogether, defining a pure value-passing interface. For example, in FreeBSD the `accept()` call has type:

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

In the model, on the other hand, accept() has type

$$\mathsf{fd} \rightarrow \mathsf{fd} * (\mathsf{ip} * \mathsf{port})$$

taking an argument of type fd and either returning a triple of type $\mathsf{fd} * (\mathsf{ip} * \mathsf{port})$ or raising one of several possible errors. The abstraction from the system API to the model API is embodied in an `nssock` C library, which has almost exactly the same behaviour as the standard calls but also calculates the abstract HOL views of each call and return, dumping them to a log.

The model is language-neutral, but we also have an OCaml [23] library (implemented above `nssock`) with types almost identical to those of the model.

Protocol issues We work only with IPv4, though there should be little difference for IPv6. For TCP we cover roughly the protocol developments in FreeBSD 4.6-RELEASE. We include MSS options; the RFC1323 timestamp and window scaling options; PAWS; the RFC2581 and RFC2582 New Reno congestion control algorithms; and the observable behaviour of syncaches. We do not include the RFC1644 T/TCP (though it is in this codebase), SACK, or ECN. For UDP, for historical reasons we deal only with unicast communication.

Time It is essential to model time passage explicitly: much TCP behaviour is driven by timers and timeouts. Time passage is modelled by transitions labelled $dur \in \mathbb{R}_{>0}$ interleaved with other transitions, modelling global time which passes uniformly for all participants.

Global time cannot be directly observed, however, and the specification imposes loose bounds on the time behaviour of certain operations: for example, a call to pselect() with no file descriptors specified and a timeout of 30s will return at some point in the interval $[30, 30 + \mathrm{dschedmax}]$ seconds. Some operations have both a lower and upper bound; some must happen immediately; and some have an upper bound but may occur arbitrarily quickly. Especially, the rate of a host's 'ticker' is constrained only to be within certain bounds of unity. This ensures the specification includes the behaviour of real systems with (boundedly) inaccurate clocks.

Relationship between code and specification structure In writing the specification we have examined the implementation source code closely, but the two have very different structure. The code is in C with a rough layer structure (but tight coupling between some layers). It has accreted changes over the years, giving a tangled control flow in some parts, and is optimised for fast-path performance. For the specification, however, clarity is the prime concern. In structuring it we have tried to isolate each conceptually-distinct behaviour into a single rule. Each rule is as far as possible declarative, defining a relation between outputs and inputs. In some cases we have been forced to introduce extra structure to mirror oddities in the implementations, e.g. intermediate state variables to record side effects that subsidiary functions have before a segment is dropped, and clauses to model the fact that the BSD fast-path optimisation is not precisely equivalent to the slow path. The specification is loose at many points, allowing certain variations in behaviour.

3. THE SPECIFICATION

The specification is written as a mechanized higher-order logic definition in HOL [14, 17], a language that is rich and expressive yet supports both internal consistency checking (type checking in particular is essential with a definition of this scale) and our automated testing techniques. We have tried hard to establish idioms with as little syntactic noise as possible, e.g. with few explicit 'frame conditions' concerning irrelevant quantities.

It is a moderately large document, around 360 pages typeset automatically from the HOL source. Of this around 125 pages is preamble defining the main types used in the model, e.g. of the representations of host states, TCP segments, etc., and various auxiliary functions. The remainder consists primarily of the host transition rules, each defining the behaviour of the host in a particular situation, divided roughly into the Sockets API rules (160 pages) and the protocol rules (75 pages). This includes extensive comments, e.g. with summaries for each Sockets call and differences between the model API and the three implementation APIs.

These rules are supported by type definitions for each interaction and state component, constant definitions for the various protocol parameters, and auxiliary function definitions for common operations and glue. For instance, the state of a *host* is defined as a HOL type as follows:

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

This introduces a record type with labelled fields. Several of the fields hold finite maps (lookup tables), e.g. $socks$ maps from the type sid of socket identifiers to the type socket of socket records.

The iq field has type msg list timed, denoting a list of IP datagrams with an attached timer used to model the delay between receipt and processing. The ts field holds a lookup table from thread IDs tid to thread states (running, blocked in a system call, or ready to be returned a value), again with an attached timer (used to model scheduling delays). Other fields are internal kernel tables (e.g. $socks$), protocol data (e.g. $listen$), and configuration (e.g. $arch$).

Each socket has internal structure, some of which is protocol-dependent: e.g., flags, local and remote IP addresses and ports, pending error, TCP state, send and receive queues, and TCP control block (cb) variables.

Another key type is that of TCP segments:

$$
\begin{array}{l}
\mathsf{tcpSegment}\\
= \langle\!|\ is_1 : \mathsf{ip\ \ option};\ \text{(* source IP address *)}\\
\quad is_2 : \mathsf{ip\ \ option};\ \text{(* destination IP address *)}\\
\quad ps_1 : \mathsf{port\ \ option};\ \text{(* source port *)}\\
\quad ps_2 : \mathsf{port\ \ option};\ \text{(* destination port *)}\\
\quad seq : \mathsf{tcp\_seq\_local};\ \text{(* sequence number *)}\\
\quad ack : \mathsf{tcp\_seq\_foreign};\ \text{(* acknowledgment number *)}\\
\quad URG : \mathsf{bool};\ ACK : \mathsf{bool};\ PSH : \mathsf{bool};\\
\quad RST : \mathsf{bool};\ SYN : \mathsf{bool};\ FIN : \mathsf{bool};\\
\quad win : \mathsf{word16};\ \text{(* window size *)}\\
\quad ws : \mathsf{byte\ \ option};\ \text{(* window scaling option, typically 0..14 *)}\\
\quad urp : \mathsf{word16};\ \text{(* urgent pointer *)}\\
\quad mss : \mathsf{word16\ \ option};\ \text{(* max segment size option *)}\\
\quad ts : (\mathsf{ts\_seq}\#\mathsf{ts\_seq})\ \mathsf{option};\ \text{(* RFC1323 option *)}\\
\quad data : \mathsf{byte\ list}\\
\quad |\!\rangle
\end{array}
$$

This is a fairly precise model of a TCP segment: we include all TCP flags and the commonly-used options, but abstract from option order, IP flags, and fragmentation. Address and port values are modelled with HOL option types to allow the zero values to be distinguished.

Fig. 2 shows an example rule from our specification, $deliver\_in\_1$, eliding some details. This rule models the behaviour of the system on processing a SYN addressed to a listening socket. It is of intermediate complexity: many rules are rather simpler than this, a few more substantial.

The transition $h\ \langle\!|...|\!\rangle \xrightarrow{\tau} h\ \langle\!|...|\!\rangle$ appears at the top: the input and output queues are unpacked from the original and final hosts, along with the listening socket pointed to by sid and the newly-created socket pointed to by sid'.

The bulk of the rule, below the line, is the condition (a predicate) guarding the transition, specifying when the rule applies and what relationship holds between the input and output states. The condition is simply a conjunction of clauses, with no temporal ordering.

Notice first that the rule applies only when dequeueing of the topmost message on the input queue iq (as defined by predicate dequeue_iq) results in a TCP segment TCP seg, leaving remaining input queue iq'. The rule then unpacks and constrains the fields of seg by pattern matching: $seg.is_1$ must be nonzero (hence $\uparrow$) and is bound to variable i_2; similarly for i_1, p_2, p_1; fields $seg.seq$ and $seg.ack$ are bound to seq and ack (cast to the type of foreign and local sequence number respectively); field $seg.URG$ is ignored (along with FIN and PSH), and so we existentially bind it; of the other TCP flags, ACK is false, RST is false, SYN is true; and so on.

The main HOL syntax used is as follows. Record fields can be accessed by dot notation $h.ifds$ or by pattern-matching. Since all variables are logical, there is no assignment or record update *per se*,

$deliver\_in\_1$ **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 *)
$[(sid, \textsc{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, csm, crm,$
$\text{TCP\_Sock}(\text{LISTEN}, cb, \uparrow lis', [\,], *, [\,], *, \text{NO\_OOB})));$
(* new connecting socket *)
$(sid', \textsc{Sock}(*, sf', \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, csm, crm,$
$\text{TCP\_Sock}(\text{SYN\_RCVD}, cb'', *, [\,], *, [\,], *, \text{NO\_OOB})))];$
$iq := iq'$;
$oq := oq'|\!\rangle$

(* check first segment matches desired pattern; unpack fields *)
$\text{dequeue\_iq}(iq, iq', \uparrow(\text{TCP } seg)) \wedge$
$(\exists win\_\ ws\_\ mss\_\ PSH\ URG\ FIN\ urp\ data\ ack.$
$seg =$
$\langle\!| is_1 := \uparrow i_2; is_2 := \uparrow i_1; ps_1 := \uparrow p_2; ps_2 := \uparrow p_1;$
$seq := tcp\_seq\_flip\_sense(seq : \text{tcp\_seq\_foreign});$
$ack := tcp\_seq\_flip\_sense(ack : \text{tcp\_seq\_local});$
$URG := URG; ACK := \mathbf{F}; PSH := PSH;$
$RST := \mathbf{F}; SYN := \mathbf{T}; FIN := FIN;$
$win := win\_; ws := ws\_; urp := urp; mss := mss\_; ts := ts;$
$data := data$
$|\!\rangle \wedge$
$\mathbf{w2n}\ win\_ = win \wedge$ (* type-cast from word to integer *)
$\mathbf{option\_map\ ord}\ ws\_ = ws \wedge$
$\mathbf{option\_map\ w2n}\ mss\_ = mss) \wedge$

(* IP addresses are valid for one of our interfaces *)
$i_1 \in \text{local\_ips } h.ifds \wedge$
$\neg(\text{is\_broadormulticast } h.ifds\ i_1) \wedge \neg(\text{is\_broadormulticast } h.ifds\ i_2) \wedge$

(* sockets distinct; segment matches this socket; unpack fields of socket *)
$sid \notin (\mathbf{dom}(socks)) \wedge sid' \notin (\mathbf{dom}(socks)) \wedge sid \neq sid' \wedge$
$\text{tcp\_socket\_best\_match } socks(sid, sock) seg\ h.arch \wedge$
$sock = \textsc{Sock}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, csm, crm,$
$\text{TCP\_Sock}(\text{LISTEN}, cb, \uparrow lis, [\,], *, [\,], *, \text{NO\_OOB})) \wedge$

(* socket is correctly specified (note BSD listen bug) *)
$((is_2 = * \wedge ps_2 = *) \vee$
$(\text{bsd\_arch } h.arch \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2)) \wedge$
$(\mathbf{case}\ is_1\ \mathbf{of}\ \uparrow i1' \rightarrow i1' = i_1 \parallel * \rightarrow \mathbf{T}) \wedge$
$\neg(i_1 = i_2 \wedge p_1 = p_2) \wedge$

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

(* place new socket on listen queue *)
$\text{accept\_incoming\_q0 } lis\ \mathbf{T} \wedge$
(* (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 *)
$(rcvbufsize', sndbufsize', t\_maxseg', snd\_cwnd') =$
$\text{calculate\_buf\_sizes } advmss\ mss * (\text{is\_localnet } h.ifds\ i_2)$
$(sf.n(\text{SO\_RCVBUF}))(sf.n(\text{SO\_SNDBUF}))$
$tf\_doing\_tstmp'\ h.arch \wedge$
$sf' = sf\ \langle\!|\ n := \text{funupd\_list } sf.n[(\text{SO\_RCVBUF}, rcvbufsize');$
$(\text{SO\_SNDBUF}, sndbufsize')]|\!\rangle \wedge$

(* choose whether this host wants window scaling; negotiate with other side *)
$req\_ws \in \{\mathbf{F}; \mathbf{T}\} \wedge$
$tf\_doing\_ws' = (req\_ws \wedge \mathbf{is\_some}\ ws) \wedge$
$(\mathbf{if}\ tf\_doing\_ws'\ \mathbf{then}$
$rcv\_scale' \in \{n \mid n \geq 0 \wedge n \leq \text{TCP\_MAXWINSCALE}\} \wedge$
$snd\_scale' = \mathbf{option\_case}\ 0\ \mathbf{I}\ ws$
$\mathbf{else}$
$rcv\_scale' = 0 \wedge snd\_scale' = 0) \wedge$

(* choose initial window *)
$rcv\_window \in \{n \mid n \geq 0 \wedge$
$n \leq \text{TCP\_MAXWIN} \wedge$
$n \leq sf.n(\text{SO\_RCVBUF})\} \wedge$

(* record that this segment is being timed *)
$(\mathbf{let}\ t\_rttseg' = \uparrow(\text{ticks\_of } h.ticks, cb.snd\_nxt)\ \mathbf{in}$

(* 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 *)
$cb' =$
$cb\ \langle\!|\ tt\_keep := \uparrow((())_{\text{slow\_timer TCPTV\_KEEP\_IDLE}});$
$tt\_rexmt := \text{start\_tt\_rexmt } h.arch\ 0\ \mathbf{F}\ cb.t\_rttinf;$
$iss := iss; irs := seq;$
$rcv\_wnd := rcv\_window; tf\_rxwin0sent := (rcv\_window = 0);$
$rcv\_adv := ack' + rcv\_window; rcv\_nxt := ack';$
$snd\_una := iss; snd\_max := iss + 1; snd\_nxt := iss + 1;$
$snd\_cwnd := snd\_cwnd'; rcv\_up := seq + 1;$
$t\_maxseg := t\_maxseg'; t_{advmss} := advmss';$
$rcv\_scale := rcv\_scale'; snd\_scale := snd\_scale';$
$tf\_doing\_ws := tf\_doing\_ws';$
$ts\_recent := \mathbf{case}\ ts\ \mathbf{of}$
$* \rightarrow cb.ts\_recent \parallel$
$\uparrow(ts\_val, ts\_ecr) \rightarrow (ts\_val)^{\text{TimeWindow}}_{\text{kern\_timer dtsinval}};$
$last\_ack\_sent := ack';$
$t\_rttseg := t\_rttseg';$
$tf\_req\_tstmp := tf\_doing\_tstmp';$
$tf\_doing\_tstmp := tf\_doing\_tstmp'$
$|\!\rangle) \wedge$

(* generate outgoing segment *)
$\mathbf{choose}\ seg' :: \text{make\_syn\_ack\_segment } cb'$
$(i_1, i_2, p_1, p_2)(\text{ticks\_of } h.ticks).$

(* attempt to enqueue segment; roll back specified fields on failure *)
$\text{enqueue\_or\_fail } \mathbf{T}\ h.arch\ h.rttab\ h.ifds[\text{TCP } seg']oq$
$(cb$
$\langle\!|\ snd\_nxt := iss;$
$snd\_max := iss;$
$t\_maxseg := t\_maxseg';$
$last\_ack\_sent := \text{tcp\_seq\_foreign } 0w;$
$rcv\_adv := \text{tcp\_seq\_foreign } 0w$
$|\!\rangle)cb'(cb'', oq')$

Figure 2: A sample TCP transition rule.

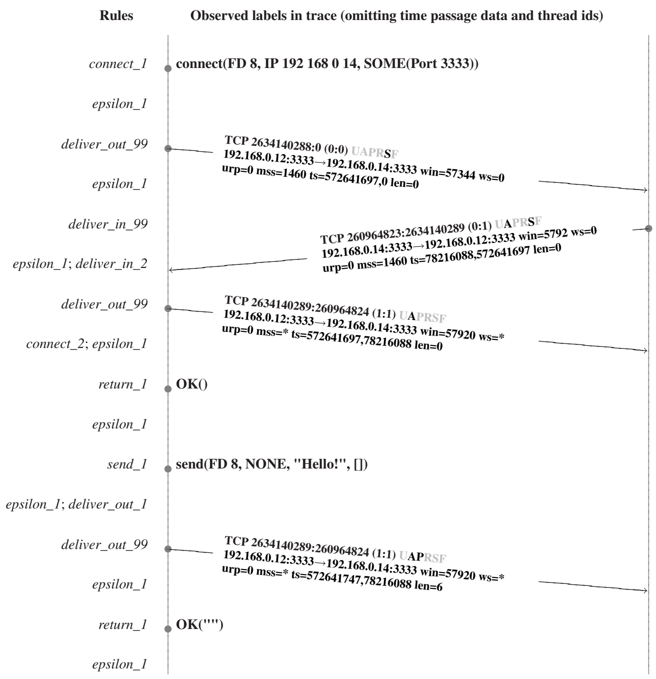


Figure 3: Extract from sample checked TCP trace, with rule firings.

but we may construct a new record by copying an existing one and providing new values for specific fields: $cb' = cb\ \llparenthesis irs := seq \rrparenthesis$ states that the record cb' is the same as the record cb, except that field $cb'.irs$ has the value seq. For optional data items, $*$ denotes absence (or a zero IP or port) and $\uparrow x$ denotes presence of value x.

After some validity checks, and determining the matching socket, the predicate computes values required to generate the response segment and to update the host state. For instance, the host nondeterministically may or may not wish to do timestamping (here the nondeterminism models the unknown setting of a configuration parameter). Timestamping will be performed if the incoming segment also contains a timestamping request. Several other local values are specified nondeterministically: the advertised MSS may be anywhere between 1 and 65495, the initial window is anywhere between 0 and the maximum allowed bounded by the size of the receive buffer, and so on. Buffer sizes are computed based on the (nondeterministic) local and (received) remote MSS, the existing buffer sizes, whether the connection is within the local subnet, and the TCP options in use. The algorithm used differs between implementations, and is specified in the auxiliary function $\mathrm{calculate\_buf\_sizes}$ (definition not shown).

Finally, the internal TCP control block cb' for the new socket is created, based on the listening socket's cb. Timers are restarted, sequence numbers are stored, TCP's sliding window and congestion window are initialised, negotiated connection parameters are saved, and timestamping information is logged. An auxiliary function $\mathrm{make\_syn\_ack\_segment}$ constructs an appropriate response segment using parameters stored in cb'; if the resulting segment cannot be queued (due to an interface buffer being full or for some other reason) then certain of the updates to cb' are rolled back.

Some non-common-case behaviour is visible in this rule: (1) in the BSD implementation it is possible for a listening socket to have a peer address specified, and we permit this when checking the socket is correctly formed; and (2) URG or FIN may be set on an initial SYN, though this is ignored by all implementations we consider.

Fig. 3 shows an extract from a captured trace with the observed labels for Sockets API calls and returns and TCP segment sends and receives. It is annotated on the left with the sequence of rules $\mathit{connect\_1}$, $\mathit{epsilon\_1}$,... used to match this trace when it was checked. Note the time passage transitions (rule $\mathit{epsilon\_1}$) and the various internal (τ) steps: $\mathit{deliver\_in\_2}$ dequeuing a SYN,ACK segment and generating an ACK (to be later output by $\mathit{deliver\_out\_99}$), $\mathit{connect\_2}$ setting up the return from the blocked $\mathrm{connect}()$, and $\mathit{deliver\_out\_1}$ enqueuing the "Hello!" segment for output. The diagram shows only the rule names, omitting the symbolic internal state of the host which is calculated at each point.

This trace shows a common case, and should be unsurprising. However, the specification covers TCP in *full detail*: fast retransmit and recovery, RTT measurement, PAWS, and so on, and for *all possible inputs*, not just common cases: error behaviour, pathological corners, concurrent socket calls, and so on. Such completeness of specification is an important part of our rigorous approach.

4. EXPERIMENTAL VALIDATION

Trace generation To generate traces of the real-world implementations in a controlled environment we set up an isolated test network, with machines running each of our three OS versions, and wrote instrumentation and test generation tools. A sample test configuration is illustrated in Fig. 4. We instrument the wire interface with a `slurp` tool above `libpcap`, instrument the Sockets API with an `nssock` wrapper, and on BSD additionally capture TCP control block records generated by the `TCP_DEBUG` kernel option. All three produce HOL format records which are merged into a single trace; this requires accurate timestamping, with careful management of NTP offsets between machines and propagation delays between them. A test executive `tthee` drives the system by making Sockets API calls (via a `libd` daemon) and directly injecting messages with an `injector` tool. These tools are written in OCaml [23] with additional C libraries. The resulting traces are HOL-parsable text files containing an initial host state (its interfaces, routing table, etc.), an initial time, and a list of timestamped labels (as in §2).

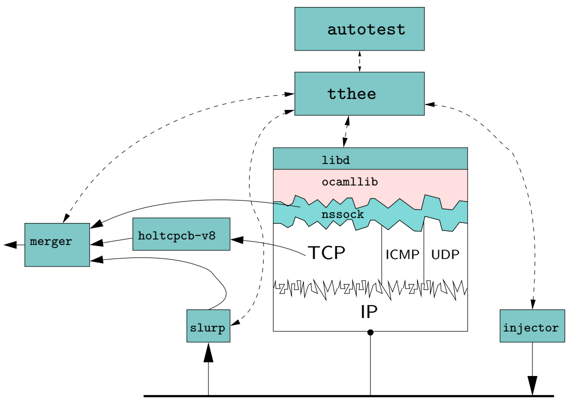
Tests Tests are scripted above `tthee`. They are of two kinds. The most straightforward use two machines, one instrumented and an auxiliary used as a communication partner, with socket calls invoked remotely. The others use a virtual auxiliary host, directly injecting messages into the network; this permits tests that are not easily produced via the Sockets layer, e.g. with re-ordering, loss, or illegal or nonsense segments.

We have written tests to, as far as we could, exercise all the interesting behaviour of the protocols and API. Almost all tests are run on all three OSs; many are automatically iterated over a selection of TCP socket states, port values, etc. In total around 6000 traces are generated.

For example, trace 1484, of intermediate complexity, is informally described as follows: "send() – for a non-blocking socket in state $\mathrm{ESTABLISHED}(\mathit{NO\_DATA})$, with a reduced send buffer that is almost full, attempt to send more data than there is space available."

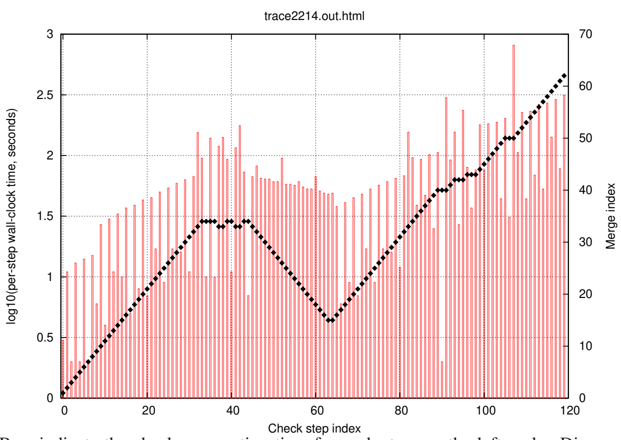
Assessing coverage of the traces is non-trivial, as the system is essentially infinite-state, but we can check that almost all the host LTS rules are covered.

Symbolic evaluation Given the (nondeterministic) transition system $\xrightarrow{l}$ defined by the host LTS, an initial host h_0, and an



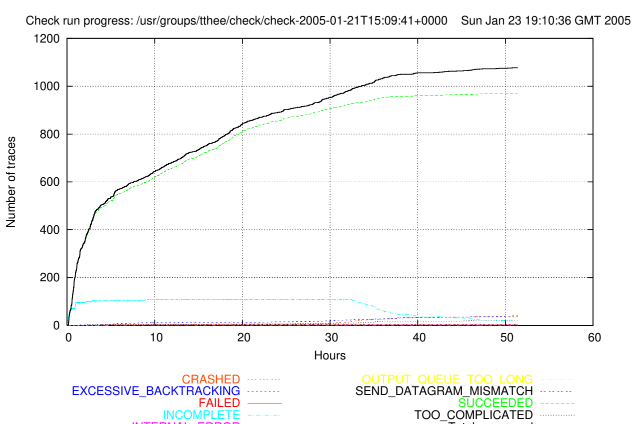
This shows a sample configuration of our instrumentation and test generation tools.

Figure 4: Testing infrastructure.



Bars indicate the checker execution time for each step, on the left scale. Diamonds indicate how far through the trace each step is, on the right scale. This trace, atypically, required significant backtracking; most need no backtracking of depth greater than one.

Figure 5: Checker monitoring: timed step graph.



This indicates how an entire check run progressed, showing the number of traces processed, succeeded, and non-succeeded for various reasons. The INCOMPLETE line (dropping around 33 hours) indicates roughly how many worker machines were active. The shape of the graph is largely determined by the order of traces in the run, shortest first.

Figure 6: Checker monitoring: progress of a TCP run.

experimentally-observed trace of labels $l_1 \ldots l_n$, we want to determine whether h_0 could have exhibited this behaviour. Because the transition system includes unobservable τ labels, the sequence of events undergone by h_0 may have also included τ steps whose presence will need to be inferred. Nondeterminism arises in two ways: two or more rules may apply to the same host-label pair (often one for a τ step), and a single rule may weakly constrain some part of the resulting host, e.g. constraining a number to fall within certain bounds, or an error code to be one of several possibilities. The first is dealt with by a depth-first search (checking τ possibilities last). For the second, explicit search is clearly impractical. Instead, the system maintains sets of constraints, which are just arbitrary HOL formulae, attached to each transition. These constraints are simplified (including the action of arithmetic decision procedures) and checked for satisfiability as checking proceeds. Later labels often fully determine variables that were introduced earlier, e.g. for file descriptors, TCP options, etc.

For example, a *connect_1* transition modelling the connect() invocation in TCP trace 0999 introduces:

```
==New variables: (advmss :num), (advmss' :num option),
  (cb'_2_rcv_wnd :num), (n :num), (rcv_wnd0 :num),
  (request_r_scale :num option), (ws :char option)

==New constraints:
∀n2. advmss' = SOME n2 ==> n2 <= 65535
∀n2. request_r_scale = SOME n2 ==> ORD (THE ws) = n2
pending (cb'_2_rcv_wnd=rcv_wnd0* 2**case 0 I request_r_scale)
pending (ws = OPTION_MAP CHR request_r_scale)
advmss <= 65495
cb'_2_rcv_wnd <= 57344
n <= 5000
rcv_wnd0 <= 65535
1 <= advmss
1 <= rcv_wnd0
1024 <= n
advmss' = NONE ∨ advmss' = SOME advmss
request_r_scale=NONE ∨ ∃n1.request_r_scale=SOME n1 ∧ n1<=14
nrange n 1024 3976
nrange rcv_wnd0 1 65534
case ws of NONE -> T || SOME v1 -> ORD v1 <= TCP_MAXWINSCALE
```

Some of these will be further constrained by the first segments that appear; as they become ground it becomes possible to substitute them out altogether.

An important aim of the formalisation has been to support the use of a natural, mathematical idiom in the writing of the specification. This does not always produce logical formulas well-suited to automatic analysis. Even making sure that the conjuncts of a side-condition are “evaluated” (simplified) in a suitable order can make a big difference to the efficiency of the tool. Rather than force the specification authors to behave like Prolog programmers, we have developed a variety of tools to automatically translate a variety of idioms into provably-equivalent forms that are easier to check.

Distributed checking infrastructure Trace checking is computationally expensive, but for good coverage we want to check many traces. We therefore distributed checking over as many processors as possible. Each trace (apart from initialisation of the evaluator) is independent, so this is conceptually straightforward.

Checking is compute-bound, not space- or IO-limited. A typical trace check run might require 100MB of memory (a few need more); most trace input files are only of the order of 10KB, and the raw checker output for a trace is 100KB – 3MB.

At present we use approximately 100 processors, running background jobs on personal workstations and lab machines (the fastest being dual 3.06GHz Xeons) and using a processor bank of 25 dual Opteron 250s. We currently rely on a common NFS-mounted file system. Checking around 2600 UDP traces takes approximately 5 hours; checking around 1100 TCP traces (for BSD only) takes approximately 50 hours.

Considerable work has gone in to achieving this performance, e.g. with recent improvements to the simplifier reducing the total TCP check time from 500 hours, which was at the upper limit of what was practical.

The resulting dataset is large and good visualisation tools are necessary for working with it. Our main tool is an HTML display of the results of each check run, with for each trace a link to the checker output, the trace in HTML and graphical form (as in Fig. 3), the short description, and a graph showing the backtracking and progress of the checker (as in Fig. 5). The progress of a whole run can be visualised as in Fig. 6.

Fig. 5 suggests the checker run-time per step rises piecewise exponentially with the trace length, though with a small exponent. This is due to the gradual accumulation of constraints, especially time passage rate constraints. In principle there is no reason why in long traces they could not be agglomerated.

5. RESULTS

The experimental validation process shows that the specification admits almost all the test traces generated. For UDP, over all three implementations (BSD, Linux, and WinXP), 2526 (97.04%) of 2603 traces succeed. For TCP we have focussed recently on the BSD traces, and here 1004 (91.7%) of 1095 traces succeed.

While we have not reached 100% validation, we believe these figures indicate that the model is for most purposes very accurate — certainly good enough for it to be a useful reference. Further, we believe that closing the gap would only be a matter of additional labour, fixing sundry very local issues rather than needing any fundamental change to the specification or the tools.

Of the UDP non-successes: 36 are due to a problem in test generation (difficulties with accurate timestamping on WinXP); 27 are tests which involve long data strings for which we hit a space limitation of the HOL string library (which uses a particularly non-space-efficient representation at present); 11 are because of known problems with test generation; and 3 are due to an ICMP delivery problem on FreeBSD.

Of the TCP non-successes: 42 are due to checker problems (mainly memory limits); 6 are due to problems in test generation; and the remaining 43 traces due to a collection of 20 issues in the specification which we have roughly diagnosed but not yet fixed.

Much of the TCP development was also carried out for all three implementations, and the specification does identify various differences between them. In the later stages we focussed on BSD for two reasons. Firstly, the BSD debug trace records make automated validation easier in principle. Secondly, as a small research team we have had only rather limited staff resources available. We believe that extending the TCP work to fully cover the other implementations would require little in the way of new techniques.

The success rates above are only meaningful if the generated traces do give reasonable coverage. Care was taken in the design of the test suite to cover interesting and corner cases, and we can show that almost all rules of the model are exercised in successful trace checking. Of the 194 host LTS rules 142 are covered in at least one successful trace check run; 32 should not be covered by the tests (e.g. rules dealing with file-descriptor resource limits, or non-BSD TCP behaviour); and 20 either have not had tests written or not yet succeeded in validation. Moreover, test generation was largely independent of the validation process (some additional tests were constructed during validation, and some particularly long traces were excluded). For TCP, however, it would be good to check more medium-length traces, to be sure that the various congestion-control regimes are fully explored. Our trace set is perhaps weighted more towards connection setup/teardown and Sockets API issues.

6. IMPLEMENTATION ANOMALIES

The goal of this project was not to find bugs in the implementations. Indeed, from a *post-hoc* specification point of view, the implementation behaviour, however strange, is a *de facto* standard which users of the protocols and API should be aware of. Moreover, to make validation of the specification against the implementation behaviour possible, it must include whatever that behaviour is.

Nonetheless, in the course of the work we have found many behavioural anomalies, some of which are certainly bugs in the conventional sense. There are explicit OS version dependencies on around 260 lines of the specification, and the report [7] details around 30 anomalies. All are relatively local issues — the implementations are extremely widely used, so it would be very surprising to find serious problems in the common-case paths. We list a few briefly below, mostly for BSD TCP. By describing these oddities we hope primarily to give some sense of what kind of fine-grain detail can be captured by our automated testing process, in which window values, time values, etc. are checked against their allowable ranges as soon as possible. Some may be already known.

- The receive window is updated on receipt of a bad segment.
- Simultaneous open can respond with an ACK rather than a SYN,ACK.
- The code has an erroneous definition of the `TCPS_HAVERCVDFIN` macro, making it possible, for example, to generate a `SIGURG` signal from a socket after its connection has been closed.
- listen() can be (erroneously) called from any state, which can lead to pathological segments being transmitted (with no flags or only a FIN).
- After repeated retransmission timeouts the RTT estimates are incorrectly updated.
- After 2^{32} segments there is a 16 segment window during which, if the TCP connection is closed, the RTT values will not be cached in the routing table.
- The received urgent pointer is not updated in the fast-path code, so if 2GB of data is received in the fast path, subsequent urgent data will not be correctly signalled.
- On Linux, options can be sent in a SYN,ACK that were not in the received SYN.

The main point we observe in the implementations is that their behaviour is extremely complex and irregular, but that is not subject to any easy fix.

7. DESIGN FOR TEST

Our experience with post-hoc rigorous specification and specification-based testing for existing protocols suggests that similar work would be feasible and desirable at design-time for new protocols.

Rigorous specification alone, if it is possible in a sufficiently lightweight form, promises several benefits:

- As Anderson *et al.* write in their *Design guidelines for robust Internet protocols*, the value of *conceptual simplicity* is widely accepted but hard to realise [2, Guideline #1]. Writing a behavioural specification makes complexity apparent early in the design process, drawing attention to unnecessary irregularities and asymmetries in a way that informal prose and working code do not.
- Specification is a form of communication, both within the protocol design group and later to implementors and users. The

added *clarity* of rigorous specification aids precise communication and reduces ambiguity.

- Incompleteness is a fertile source of bugs. Specifications that can be machine-checked for *completeness*, or that make completeness self-evident, can address this.

The ability to directly test conformance of an implementation would result in higher-quality implementations, with fewer quirks and obscure bugs. A specification should clearly *define* how an acceptable implementation may behave. We have shown here that it is feasible to do so in a way that makes it possible to *test* whether an execution of an implementation is admissible, without any unchecked manual (error-prone) translation.

Bearing conformance testing in mind during protocol and API design could make testing much more straightforward. The treatment of loose specification and nondeterminism is crucial here. For TCP and the Sockets API there are many internal events (processing messages from input queues, timer firings, etc.) which are not directly observable either on the wire or via the API, and which are not determined in a simple way by the observable events. This meant that our automated validation had to maintain highly symbolic descriptions of states, with unresolved constraints on state variables, and had to perform a backtracking search. If one ensured the API could (on demand) reveal internal state changes, and enough of the internal state to compute the abstract specification state, then testing could be carried out in lock-step between implementation and specification, with fully-known states.

Ideally one would develop both an endpoint specification, prescribing the behaviour of a single protocol stack, and an end-to-end specification, characterising just the behaviour that API users can depend upon. Including the API behaviour in both is feasible, would reduce future portability problems, and would ease testing.

Good specification idioms are crucial for readability. Our use of HOL enables modular structuring to be used for maximum clarity, without constraining the structure of implementations. It would be desirable to isolate aspects of the specification that are loose, making it possible to modularly replace them with particular definitions to yield an executable prototype. Similarly, aspects that are subject to more frequent change (such as congestion control) should be replaceable modules.

The up-front effort and technical background required for a rigorous specification might be thought problematic, but we believe not. Gaining enough familiarity with HOL to understand and write this style of specification takes a matter of a few days. Working directly on improving the symbolic evaluator does need special expertise; a question for future work is to what extent our techniques there can be packaged and generalised.

The total effort required for the project, from our earliest experiments with UDP, has been perhaps 9 man-years over 3.5 calendar years. Of this, much has been devoted to idiom and tool development, and much to unpicking the intricacies of the existing TCP implementations. Similar work in future should be much less arduous, especially if performed at design-time, and in any case this is a small amount of effort compared with that devoted to designing, implementing, and using such protocols.

8. RELATED WORK

There is a vast literature devoted to verification techniques for protocols, with both proof-based and model-checking approaches, e.g. in conferences such as CAV, CONCUR, FORTE, ICNP, SPIN, and TACAS.

To the best of our knowledge, however, no previous work approaches a specification dealing with the full scale and complexity of a real-world TCP. In retrospect this is unsurprising: we have depended on automated reasoning tools and on raw compute resources that were simply unavailable in the 1980s or early 1990s. Our goals have also been different, and in some sense more modest, than the correctness theorems of traditional formal verification: we have not attempted to prove that an implementation of TCP meets an endpoint specification, or that two TCP endpoints do provide a psuedo-reliable stream service.

The most detailed rigorous specification of a TCP-like protocol we are aware of is that of Smith [36], an I/O automata specification and implementation, with a proof that one satisfies the other, used as a basis for work on T/TCP. The protocol is still substantially idealised, however: congestion control is not covered, nor are options, and the work supposes a fixed client/server directionality. Later work by Smith and Ramakrishnan uses a similar model to verify properties of a model of SACK [35].

Musuvathi and Engler have applied their CMC model-checker to a Linux TCP/IP stack [26]. The properties checked were of two kinds: resource leaks and invalid memory accesses, and protocol-specific properties specified by a hand translation of the RFC793 state diagram into C code. While this is a useful model of the protocol, it is an extremely abstract view, omitting flow control, congestion control etc. Four bugs in the Linux implementation were found.

Bhargavan *et al.* develop an automata-theoretic approach for monitoring of network protocol implementations, with classes of properties that can be efficiently checked on-line in the presence of network effects [3]. They show that certain properties of TCP implementations can be expressed.

In a rare application of rigorous techniques to actual standards, Bhargavan, Obradovic, and Gunter use a combination of the HOL proof assistant and the SPIN model checker to study properties of distance-vector routing protocols [4], proving correctness theorems. In contrast to our experience for TCP, they found that for RIP the existing RFC standards were precise enough to support "without significant supplementation, a detailed proof of correctness in terms of invariants referenced in the specification". The protocols are significantly simpler: their model of RIP is (by a naïve line count) around 50 times smaller than the specification we present here.

Alur and Wang address the PPP and DHCP protocols [1]. For each they check refinements between models that are manually extracted from the RFC specification and from an implementation.

There are I/O automata specifications and proof-based verification for aspects of the Ensemble group communication system by Hickey, Lynch, and van Renesse [15], and NuPRL proofs of fast-path optimizations for local Ensemble code by Kreitz [22].

For radically idealised variants of TCP, one has for example the PVS verification of an improved Sliding Window protocol by Chkliaev *et al.* [10], and Fersman and Jonsson's application of the SPIN model checker to a simplified version of the TCP establishment/teardown handshakes [12]. Schieferdecker verifies a property (expressed in the modal μ calculus) of a LOTOS specification of TCP, showing that data is not received before it is sent [32]. The specification is again roughly at the level of the TCP state diagram. Hofmann and Lemmen report on testing of a protocol stack based on an SDL specification of TCP/IP [16]. Billington and Han have produced a coloured Petri net model of the service provided by TCP for a highly idealised ISO-style interface [6]. Murphy and Shankar verify some safety properties of a 3-way handshake protocol analogous to that in TCP [24] and of a transport protocol based on this [25]. Finally, Postel's PhD thesis used early Petri net protocol models descriptively [31].

A number of tools exist for testing or fingerprinting of TCP implementations with hand-crafted ad-hoc tests, not based on a rigorous specification. They include the `tcpanaly` of Paxson [30], the TBIT of Padhye and Floyd [28], and Fyodor's `nmap` [13]. RFC2398 [29] lists several other tools. There are also commercial products such as Ixia's Automated Network Validation Library [20].

Implementations of TCP in high-level languages have been written by Biagioni in Standard ML [5], by Castelluccia *et al.* in Esterel [9], and by Kohler *et al.* in Prolac [21]. Each of these develops compilation techniques for performance. They are presumably more readable than low-level C code, but each is a particular implementation rather than a specification of a range of allowable behaviours: as for any implementation, allowable nondeterminism means they could not be used as oracles for conformance testing.

9. CONCLUSION

Future work There are many directions for future work based on the current specification.

While it has been subject to extensive validation and annotation, the specification surely still contains some errors, and is not as clearly presented as it might be. We would be interested to hear feedback on either point.

The specification has been developed based on three particular implementations, and with reference to the Linux and (especially, for TCP) the BSD source code, but we have aimed to make it sufficiently loose to admit other implementation differences. It would be interesting to run the validation tools on a fresh implementation that did not influence the specification development, to see how much implementation-specific change is required, and also on new (and longer) trace sets. Our automated validation for TCP makes use of the BSD `TCP_DEBUG` trace records to resolve nondeterminism early — how important this is is unclear.

It would be useful to maintain the specification, completing the Linux and WinXP validation for TCP, tracking version changes in all three implementations, and providing feedback to implementation groups. We believe this would now be an essentially routine task.

We have informal descriptions of many invariants which we believe the model satisfies; it would be useful to prove these (as we did for our earlier UDP specification).

Given the segment-level endpoint specification we expect one could directly produce a more abstract stream-level end-to-end specification of TCP and Sockets. In doing so it would be useful to state formally the intended abstraction relation between the segment-level and stream-level models. Proving this would advance the state of the art, but the effort required may be disproportionate to what would be learned about the protocol — which is, after all, known to work reasonably well in practice.

In the opposite direction, towards the concrete, we would like to specify particular choices at points of nondeterminism, thus deriving an actual implementation. It should then be possible to integrate the packet `slurp` tool, the specification, our symbolic evaluation engine, and the `injector`, forming a working prototype TCP. One could gain additional confidence in the validity of the specification by checking this interoperates with existing TCP/IP stacks (artificially slowed down to match the speed of the evaluator).

TCP does not have a clear modular structure, but rather has accreted functionality through a succession of RFCs and code changes. We have tried to clarify the behaviour as much as we could, but the imperative nature of the code is hard to escape — witness especially our $deliver\_in\_3$ rule, which processes normal segments in a connected state, and which must deal with many computation paths that have some important side-effects but then abort. Any improvement to this structure would be worthwhile. For example, good modular structure would let one cleanly replace the existing congestion control mechanisms by other proposals.

Finally, our symbolic evaluator is a special-purpose tool, part of which is particular to the details of our specification. Ideally one would have a more general-purpose system, applying to any specification in some identified language. However, the specification makes use of a substantial fragment of the HOL logic, with the evaluator performing non-trivial proof that certain parts of it are equivalent to algorithmically more tractable definitions. It is therefore hard to imagine that such a language can exist. Nonetheless, it would be interesting to characterise more sharply exactly what fragment of HOL is needed here.

Summary We have established rigorous techniques for specification and specification-based conformance testing that are practical for protocols as complex as TCP. To the best of our knowledge this is the first time this has been done.

We have demonstrated our techniques by producing a *post hoc* specification of TCP, UDP, and the Sockets API. The specification is *rigorous* (in the mechanised higher-order logic of HOL), *detailed* (with almost all aspects of the real-world communications at the level of TCP segments and UDP datagrams), has *broad coverage* (with arbitrary incoming messages and Sockets API invocations), and is *accurate* (experimentally validated against the behaviour of some widely-deployed implementations).

The specification has been extensively annotated, aiming to make it usable as a reference by TCP/IP stack implementors, users of the Sockets API, and designers of protocol modifications. It is available (in a version automatically typeset from the HOL source) on the web [8]. Also available from that link is a version of the 'TCP state diagram' based on the specification, which is rather more complete than the diagrams from RFC793 or the later Stevens texts.

Perhaps most importantly, the practicality of our technique for TCP suggests that similar rigorous specification should be both feasible and desirable for future protocol design, leading to clearer specifications and higher-quality implementations, and we have discussed some key points for such a design-for-test approach.

Acknowledgements We thank Jon Crowcroft and Tim Griffin for helpful comments, and many members of the Laboratory, especially Piete Brooks, for making compute resource available.

We acknowledge support from a Royal Society University Research Fellowship (Sewell), a St Catharine's College Heller Research Fellowship (Wansbrough), EPSRC grant GR/N24872 *Wide-area programming: Language, Semantics and Infrastructure Design*, EPSRC grant EP/C510712 *NETSEM: Rigorous Semantics for Real Systems*, EC FET-GC project IST-2001-33234 PEPITO *Peer-to-Peer Computing: Implementation and Theory*, CMI UROP internship support (Smith), and EC Thematic Network IST-2001-38957 APPSEM 2. National ICT Australia is funded by the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

10. REFERENCES

[1] R. Alur and B.-Y. Wang. Verifying network protocol implementations by symbolic refinement checking. In *Proc. CAV '01, LNCS 2102*, pages 169–181, 2001.

[2] T. E. Anderson, S. Shenker, I. Stoica, and D. Wetherall. Design guidelines for robust internet protocols. *Computer Communication Review*, 33(1):125–130, 2003.

[3] K. Bhargavan, S. Chandra, P. J. McCann, and C. A. Gunter. What packets may come: automata for network monitoring. In *Proc. POPL*, pages 206–219, 2001.

[4] K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, 2002.

[5] E. Biagioni. A structured TCP in standard ML. In *Proc. SIGCOMM '94*, pages 36–45, 1994.

[6] J. Billington and B. Han. On defining the service provided by TCP. In *Proc. ACSC: 26th Australasian Computer Science Conference*, Adelaide, 2003.

[7] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 1: Overview. Technical Report UCAM-CL-TR-624, Computer Laboratory, University of Cambridge, Mar. 2005. 88pp. Available at https://i o co do o b .os ar.com/users/pes20/Netsem/

[8] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 2: The specification. Technical Report UCAM-CL-TR-625, Computer Laboratory, University of Cambridge, Mar. 2005. 386pp. Available at https://i o co do o b .os ar.com/users/pes20/Netsem/

[9] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. *IEEE/ACM Trans. Netw.*, 5(4):514–524, 1997. Full version of a paper in SIGCOMM '96.

[10] D. Chkliaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol. In *Proc. TACAS'03, LNCS 2619*, pages 113–127, 2003.

[11] M. Compton. Stenning's protocol implemented in UDP and verified in Isabelle. In *Proc. 11th CATS, Computing: The Australasian Theory Symposium*, pages 21–30, 2005.

[12] E. Fersman and B. Jonsson. Abstraction of communication channels in Promela: A case study. In *Proc. 7th SPIN Workshop, LNCS 1885*, pages 187–204, 2000.

[13] Fyodor. nmap. http://i o e 2. o f 2 .os ar.com/nmap/

[14] M. J. C. Gordon and T. Melham, editors. *Introduction to HOL: a theorem proving environment*. Cambridge University Press, 1993.

[15] J. Hickey, N. A. Lynch, and R. van Renesse. Specifications and proofs for Ensemble layers. In *Proc. TACAS, LNCS 1579*, pages 119–133, 1999.

[16] R. Hofmann and F. Lemmen. Specification-driven monitoring of TCP/IP. In *Proc. 8th Euromicro Workshop on Parallel and Distributed Processing*, Jan. 2000.

[17] The HOL 4 system, Kananaskis-2 release. http://i f co f 2 . f 2 o e .os ar.com/

[18] IEEE and The Open Group. *IEEE Std 1003.1™-2001 Standard for Information Technology — Portable Operating System Interface (POSIX®)*. Dec. 2001. Issue 6. Available http://i777777o6f70656e67726f7570o6f7267 .os ar.com/onlinepubs/007904975/toc.htm

[19] The Isabelle proof assistant. http://i 2 c c o . eo do .os ar.com/

[20] IXIA. IxANVL(tm) — automated network validation library, 2005. http://i o . f do f d .os ar.com/

[21] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable TCP in the Prolac protocol language. In *Proc. SIGGCOMM '99*, pages 3–13, August 1999.

[22] C. Kreitz. Building reliable, high-performance networks with the Nuprl proof development system. *J. Funct. Program.*, 14(1):21–68, 2004.

[23] X. Leroy et al. *The Objective-Caml System, Release 3.08.2*. INRIA, Nov. 2004. Available http://i d co e.2 o 2 .os ar.com/

[24] S. L. Murphy and A. U. Shankar. A verified connection management protocol for the transport layer. In *Proc. SIGCOMM*, pages 110–125, 1987.

[25] S. L. Murphy and A. U. Shankar. Service specification and protocol construction for the transport layer. In *Proc. SIGCOMM*, pages 88–97, 1988.

[26] M. Musuvathi and D. Engler. Model checking large network protocol implementations. In *Proc.NSDI: 1st Symposium on Networked Systems Design and Implementation*, pages 155–168, 2004.

[27] M. Norrish, P. Sewell, and K. Wansbrough. Rigour is good for you, and feasible: reflections on formal treatments of C and UDP sockets. In *Proceedings of the 10th ACM SIGOPS European Workshop (Saint-Emilion)*, pages 49–53, Sept. 2002.

[28] J. Padhye and S. Floyd. On inferring TCP behaviour. In *Proc. SIGCOMM '01*, Aug. 2001.

[29] S. Parker and C. Schmechel. RFC2398: Some testing tools for TCP implementors, Aug. 1998.

[30] V. Paxson. Automated packet trace analysis of TCP implementations. In *Proc. SIGCOMM '97*, pages 167–179, 1997.

[31] J. Postel. *A Graph Model Analysis of Computer Communications Protocols.* University of California, Computer Science Department, PhD Thesis, 1974.

[32] I. Schieferdecker. Abruptly-terminated connections in TCP – a verification example. In *Proc. COST 247 International Workshop on Applied Formal Methods In System Design*, June 1996.

[33] A. Serjantov, P. Sewell, and K. Wansbrough. The UDP calculus: Rigorous semantics for real networking. Technical Report 515, Computer Laboratory, University of Cambridge, July 2001. https://i777777o636co63616do6163o756b .os ar.com/users/pes20/Netsem/

[34] A. Serjantov, P. Sewell, and K. Wansbrough. The UDP calculus: Rigorous semantics for real networking. In *Proc. TACS 2001: Fourth International Symposium on Theoretical Aspects of Computer Software, Tohoku University, Sendai*, Oct. 2001.

[35] M. A. Smith and K. K. Ramakrishnan. Formal specification and verification of safety and performance of TCP selective acknowledgment. *IEEE/ACM Trans. Netw.*, 10(2):193–207, 2002.

[36] M. A. S. Smith. Formal verification of communication protocols. In *Proc. FORTE IX/PSTV XVI*, pages 129–144, 1996.

[37] W. R. Stevens. *TCP/IP Illustrated Vol. 1: The Protocols*. 1994.

[38] W. R. Stevens. *UNIX Network Programming Vol. 1: Networking APIs: Sockets and XTI*. Second edition, 1998.

[39] K. Wansbrough, M. Norrish, P. Sewell, and A. Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. In *Proceedings of ESOP 2002: the 11th European Symposium on Programming (Grenoble), LNCS 2305*, pages 278–294, Apr. 2002.

[40] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated Vol. 2: The Implementation*. 1995.