TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification

Volume 1: Overview

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

*University of Cambridge Computer Laboratory
†NICTA, Canberra

March 18, 2005

Abstract

We have developed a mathematically rigorous and experimentally-validated post-hoc specification of the behaviour of TCP, UDP, and the Sockets API. It characterises the API and network-interface interactions of a host, using operational semantics in the higher-order logic of the HOL automated proof assistant. The specification is *detailed*, covering almost all the information of the real-world communications: it is in terms of individual TCP segments and UDP datagrams, though it abstracts from the internals of IP. It has broad *coverage*, dealing with arbitrary API call sequences and incoming messages, not just some well-behaved usage. It is also *accurate*, closely based on the *de facto* standard of (three of) the widely-deployed implementations. To ensure this we have adopted a novel experimental semantics approach, developing test generation tools and symbolic higher-order-logic model checking techniques that let us validate the specification directly against several thousand traces captured from the implementations.

The resulting specification, which is annotated for the non-HOL-specialist reader, may be useful as an informal 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 automated testing of future implementations, and a basis for design and formal proof of higher-level communication layers. More generally, the work demonstrates that it is feasible to carry out similar rigorous specification work at design-time for 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.

This document gives an overview of the project, discussing the goals and techniques and giving an introduction to the specification. The specification itself is given in the companion volume:

> *TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 2: The Specification.* Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. xxiv+359pp. [BFN+05]

which is automatically typeset from the (extensively annotated) HOL source. As far as possible we have tried to make the work accessible to four groups of intended readers: workers in *networking* (implementors of TCP/IP stacks, and designers of new protocols); in *distributed systems* (implementors of software above the Sockets API); in *distributed algorithms* (for whom this may make it possible to prove properties about executable implementations of those algorithms); and in *semantics and automated reasoning.*

Contents

List of Figures

1 Introduction

Networking rests on a number of well-known protocols, especially the transport protocols TCP and UDP, the underlying IP, and various routing protocols. The network as a whole works remarkably well, but these protocols are very complex. As Vern Paxson writes in SIGCOMM 97 [Pax97]:

> *"implementing TCP correctly is extremely difficult".*

The application programmer interface to the protocols —the Sockets API— is also complex, with subtle behavioural differences between implementations. In short,

> *using TCP and the Sockets API correctly is also difficult.*

In part these difficulties are intrinsic: the protocols must deal with loss and congestion in a very challenging environment. In part it is due to historical design choices that are now baked in — these protocols and the API are so widely deployed that change is very difficult. Part of the difficulty, however, comes from the fact that the protocols and API are not precisely defined. There is no clear sense in which an implementation may be 'correct' or not. To understand what their behaviour is (let alone why it is like that) one must be familiar with a collection of various RFC standards, with well-known texts such as those of Stevens [Ste94, WS95, Ste98] and Comer [Com00, CS99, CS00], with the BSD implementations that act as an approximate reference, and with whatever other implementations are in use in the target environment. The RFCs and texts are informal documents, which are inevitably imprecise and incomplete.

To address this, we have developed a mathematically rigorous and experimentally-validated post-hoc specification of the behaviour of TCP, UDP, and the Sockets API. It characterises the API and network-interface interactions of a host, using operational semantics in the higher-order logic of the HOL automated proof assistant [GM93, HOL]. The specification is *detailed*, covering almost all the information of the real-world communications: it is in terms of individual TCP segments and UDP datagrams, though it abstracts from the internals of IP. It has broad *coverage*, dealing with arbitrary API call sequences and incoming messages, not just some well-behaved fragment of possible usage. It is also *accurate*, closely based on the *de facto* standard of (three of) the widely-deployed implementations. To provide this accuracy we have adopted a novel experimental semantics approach, developing test generation tools and symbolic higher-order-logic model checking techniques that let us validate the specification directly against several thousand traces captured from the implementations.

The resulting specification is, to the best of our knowledge, the first of its kind. It has been annotated to make it accessible for the non-HOL-specialist reader, and hence may be useful as an informal 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 automated testing of future implementations, and a basis for design and formal proof of higher-level communication layers.

More signficantly, the work demonstrates that it is feasible to carry out similar rigorous specification work *at design-time* for new protocols, which are predominantly still defined with similar informal specification idioms. We believe the increased clarity and precision over informal specifications, and the possibility of automated specification-based testing, would make this very much worthwhile.

In contrast to much research on network protocols, our aim is not to suggest specific protocol improvements (such as new congestion control schemes) or performance analysis, but rather to develop better ways of expressing what a protocol design *is*. The focus is on dealing with the discrete behaviour of protocols in full detail rather than on (e.g.) quantitative, though approximate, analysis of how protocols behave under particular loads.

This document gives an overview of the project, discussing the goals and techniques and giving an introduction to the specification. The specification itself is given in the companion volume:

> *TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 2: The Specification.* Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. xxiv+359pp.

which is automatically typeset from the annotated HOL source. As far as possible we have tried to make the work accessible to four groups of intended readers: workers in *networking* (implementors of TCP/IP stacks, and designers of new protocols); in *distributed systems* (implementors of software above the Sockets API); in *distributed algorithms* (for whom this may make it possible to prove properties about executable implementations of those algorithms); and in *semantics and automated reasoning.*

1.1 Network Background

We first recall the basic network protocols that comprise today's network infrastructure: primarily IP, UDP, and TCP.

- IP (the *Internet Protocol*) allows a machine to send a message (an *IP datagram*) to another.

  Each machine has one or more *IP addresses*, 32-bit values such as `192.168.0.11` (for the current IPv4 version of the protocol). IP datagrams have their destination specified as an IP address. They carry a payload of a sequence of bytes and contain also a source address and various additional data. They have a maximum size of 65535 bytes, though many are smaller, constructed to fit in a 1500 byte Ethernet frame body. IP message delivery is asynchronous and unreliable; IP does not provide acknowledgements that datagrams are received, or retransmit lost datagrams. Message delivery is implemented by a combination of local networks, e.g. ethernets, and of packet forwarding between routers; these may silently drop packets if they become congested. A variety of *routing protocols* are used to establish state in these routers, determining, for any incoming packet, to which neighbouring router or endpoint machine it should be forwarded.

- UDP (the *User Datagram Protocol*) is a thin layer above IP that provides multiplexing.

  It introduces a set $\{1, .., 65535\}$ of *ports*; a UDP datagram is an IP datagram with a payload consisting of a source port, a destination port, and a sequence of bytes. Just as for IP, delivery is asynchronous and unreliable.

- TCP (the *Transmission Control Protocol*) is a thicker layer above IP that provides bidirectional byte-stream communication.

  It too uses a set $\{1, .., 65535\}$ of ports. A *TCP connection* is typically between an IP address and port of one machine and an IP address and port of another, allowing data (unstructured streams of bytes) to be sent in both directions. The two endpoints exchange *TCP segments* embedded in IP datagrams. The protocol deals with retransmission of lost data, reordering of data that arrives out of sequence, flow control to prevent the end systems being swamped with data faster than they can handle, and congestion control to limit the use of network bandwidth.

In addition, ICMP (the *Internet Control Message Protocol*) is another thin layer above IP, primarily used for signalling error conditions to be acted on by IP, UDP, or TCP.

Many other protocols are used for specific purposes, but TCP above IP is dominant. It underlies web (HTTP), file transfer (FTP) and mail protocols, and TCP and UDP together underlie the domain name service (DNS) that associates textual names with numerical IP addresses.

The first widely-available release of these protocols was in 4.2BSD, released in 1983. They are now ubiquitous, with implementations in all the standard operating systems and in many embedded devices.

Application code can interact with the protocol implementations via the *sockets interface*, a C language API originating in 4.2BSD with calls socket(), bind(), connect(), listen(), etc. The sockets interface is commonly used for interaction with UDP and TCP, not for direct interaction with IP.

1.2 Standard Practice: Protocol and API descriptions

The basic IP, UDP, TCP and ICMP protocols are described in *Request For Comment (RFC)* standards from 1980–81:

User Datagram Protocol	RFC 768	1980	3pp	STD 6
Internet Protocol	RFC 791	1981	iii+45pp	STD 5
Internet Control Message Protocol	RFC 792	1981	21pp	STD 5
Transmission Control Protocol	RFC 793	1981	iii+85pp	STD 7

The sockets interface appears as part of the POSIX standard [IEE00]. Additional information is contained in the documentation for various implementations, in particular the Unix `man` pages, and well-known texts such as those of Stevens [Ste94, WS95, Ste98].

From the titles of these documents the reader might gain the impression that TCP (say) is a single well-defined protocol. Unfortunately that is not the case, for several different reasons.

- As the protocol has been used ever more widely, in network environments that are radically different from that of the initial design, various clarifications and proposals for changes to the protocol have been made. A small sample of later RFCs include:

Requirements for Internet Hosts — Communication Layers	RFC 1122	1989
TCP Extensions for High Performance	RFC 1323	1992
The NewReno Modification to TCP's Fast Recovery Algorithm	RFC 3782	2004

Deployment of these changes is inevitably piecemeal, depending both on the TCP/IP stack implementers and on the deployment of new operating system versions, which —on the scale of the Internet— cannot be synchronised.

- Implementations also diverge from the standards due to misunderstandings, disagreements and bugs. For example, RFC 2525 collects a number of known TCP implementation problems. The BSD implementations have often served as another reference, distinct from the RFCs, for example with the text [WS95] based on the 4.4 BSD-Lite code.

- The existence of multiple implementations with differing behaviour gives rise to another 'standard': in addition (or as an alternative) to checking that an implementation conforms to the RFCs one can check that it interoperates satisfactorily with the other common implementations. The early RFC791 enshrined the doctrine that implementations should, as far as possible, interoperate even with non-RFC-conformant implementations:

  > *The implementation of a protocol must be robust. Each implementation must expect to interoperate with others created by different individuals. While the goal of this specification is to be explicit about the protocol there is the possibility of differing interpretations. In general, an implementation must be conservative in its sending behavior, and liberal in its receiving behavior. That is, it must be careful to send well-formed datagrams, but must accept any datagram that it can interpret (e.g., not object to technical errors where the meaning is still clear).*

- Neither the RFCs nor POSIX attempt to specify the behaviour of the sockets interface in any detail. The RFCs focus on the wire protocols (RFC793 also describes a model API for TCP, but one that bears little resemblence to the sockets interface as it developed); POSIX describes the C types and some superficial behaviour of the API but does not go into details as to how the interface behaviour relates to protocol events.

- Finally, both RFCs and POSIX are informal natural-language documents. Their authors were clearly at pains to be as clear and precise as they could, but almost inevitably the level of rigour is less than that of a mathematical specification, and there are many ambiguities and missing details.

1.3 The Problems of Complexity

The history of networking over the last 25 years has been one of astonishing success: those initial protocols, with relatively few modifications, have scaled to an Internet that has grown by as much as six orders of magnitude. The network does work, remarkably well in most circumstances, and large-scale distributed applications can be written above the sockets interface.

Despite this success the current state of the art is unsatisfactory. Developing high-quality software above TCP/IP and the sockets interface is a matter of craft. It requires much experience to deal with the many complex issues involved, handling the various error cases correctly, ensuring reasonable performance, etc. The total effort expended by developers getting to grips with networking is impossible to quantify, but is surely vast. Even the number of socket programming tutorials is very large.

There are two interlinked problems. Firstly, networking *is* complex. Some of the complexity is intrinsic, arising from the need to deal with partial failure, asynchrony, congestion, and the intricate state spaces of concurrent systems. Other complexity is contingent, arising from historical design choices and implementation differences, but is nonetheless real. Secondly the protocols and the sockets interface are not well-defined even in informal prose, let alone with mathematical rigour, and there are many subtle differences between the behaviour of different implementations.

The very success of networking makes these problems hard to address. For the first, it is tempting to think of redesigning the protocols and sockets interface, removing some of the unnecessary complexity and addressing other issues. However, the existing protocols are very widely deployed and must continue to interoperate, and a great deal of application code depends on the sockets interface. Replacing either with redesigned variants is therefore impractical, at least in the short term and, for some changes, perhaps indefinitely. (This is not to say that no change is possible — witness the gradual deployment of IPv6, ECN, and SACK, and proposals such as SCTP — but replacement is very difficult.)

Nonetheless, something can be done for the second problem, to improve the precision, clarity, and coverage of the specifications. The use of informal natural-language specifications probably was the best choice for early IP networking research and development, ensuring the specifications were accessible to a broad audience. The lack of precision was compensated for by an emphasis on working code and interoperability, the role of the BSD implementations as a primary reference, and a small and expert community. Now, however, the cost of this lack of precision, and of the consequent behavioural differences between implementations, has become large. Most common behaviour can be found somewhere in the combination of the RFCs, the POSIX standard, `man` pages, and texts such as [Ste94, WS95, Ste98], but these are not complete, nor completely correct. To resolve some questions one may have to resort to the source code of the particular implementation(s) one is dealing with. That code is not always available. Even where it is, it is —emphatically— not arranged for clarity. Instead, it is more-or-less optimised for performance, and embodies many historical artifacts arising from incremental change. Analysing the source to determine what its external behaviour is requires considerable expertise, not available to the typical user of the sockets interface.

Moreover, new protocols are continually under development, to address changes in use and in the underlying network; for example SCP, DCCP, and XCP. By and large they are described in similar informal prose, likely establishing the conditions for future misunderstandings and implementation differences. This falls far short of the ideal: protocol specifications should be both unambiguous and clear, and it should be possible to directly test conformance of implementations against them.

1.4 Our Contribution

In this paper we demonstrate, for the first time, a practical technique for rigorous protocol specification that make 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, and we have developed tools for automated conformance testing between the specification and real-world implementations.

To develop the technique, and to establish its feasiblity, we have produced a post-hoc specification of existing protocols: a mathematically rigorous and experimentally-validated characterisation of the behaviour of TCP, UDP, relevant parts of ICMP, and the Sockets API. It is mathematically rigorous, detailed, accurate, and covers a wide range of usage.

The Specification The main part of the specification (shown in Figure 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 return of the socket API, and between host and network via message sends and receives. 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 interface calls.

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 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_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
dur	for time dur passing

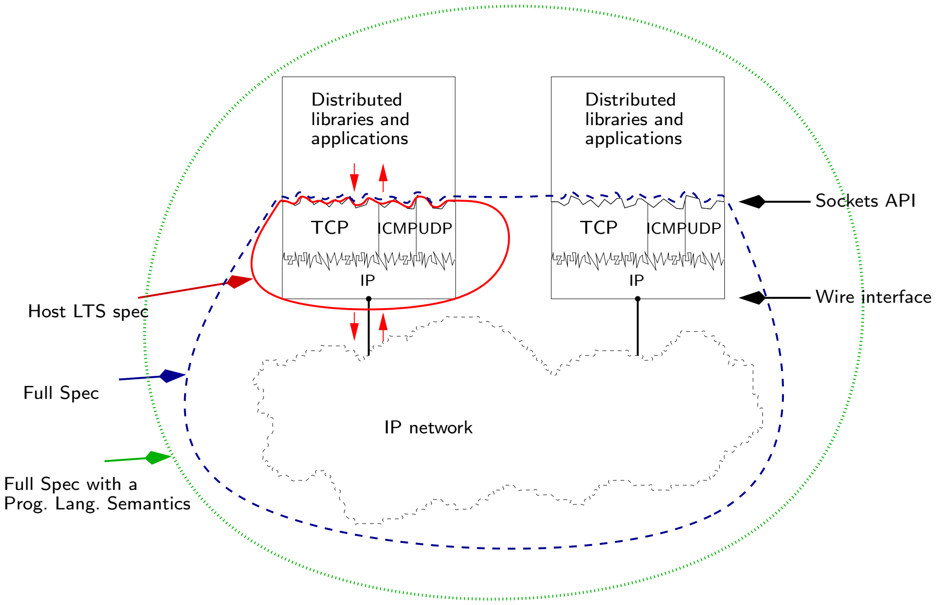

Distributed libraries and applications
Distributed libraries and applications
Sockets API
TCP
ICMP
UDP
IP
TCP
ICMP
UDP
IP
Wire interface
Host LTS spec
Full Spec
Full Spec with a Prog. Lang. Semantics
IP network

Figure 1: The Scope of the Specification

The transition relation is defined by some 140 rules for the socket calls (5–10 for each interesting call) and some 45 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. The rules form the main part of the specification in Volume 2, from §15 onwards.

Rigour The specification is expressed as an operational semantics definition in higher-order logic, mechanized using the HOL proof assistant [GM93, HOL]. Such machine-processed mathematics, in a well-defined logic, is the most rigorous form of definition currently possible.

Detail The specification is at a rather low level of abstraction, including many important details. The wire interface interactions are captured in terms of individual TCP segments (and UDP and ICMP datagrams), modelling almost all the information of the real-world communications. We do abstract from the internals of IP — we do not describe routing or IP fragmentation, and the modelled segments/datagrams include IP source and destination addresses but no other IP header fields. The sockets interface interactions are at the level of individual calls made by language threads, and returns to these calls. We use a slightly cleaned-up interface, with purely value-passing (not pointer-passing) variants of the sockets calls. This abstracts from C-language-specific intricacies of the interface, obviating the need to model the user and OS memory space. The internal behaviour of TCP is almost all included — flow control, congestion control, timeouts, etc.

Coverage The host LTS captures the behaviour of a host in response to arbitrary socket calls and incoming segments/datagrams, not just some restricted 'well-behaved' usage. Our cleaned-up sockets interface includes almost everything that can be done with `AF_INET` `SOCK_STREAM` and `SOCK_DGRAM` sockets, including (e.g.) TCP urgent data. At present we do not cover UDP multicast, for historical reasons rather than any fundamental limitation.

Accuracy Given the current state of networking standards and implementations, as outlined in §1.2, there are several quite different senses in which a rigorous specification might be 'accurate'. One could:

1. attempt to restate mathematically the content of (a chosen selection of) the natural-language RFC and POSIX standards;
2. try to capture the 'essence' of TCP with an idealised model, including enough detail to permit formal reasoning about some aspect of the protocol (e.g., to prove that data is not delivered out of order) but abstracting freely from other aspects in order to make such proof feasible;
3. try to state a sufficient condition on the behaviour of an arbitrary implementation to ensure that it interoperates satisfactorily (whatever that means precisely) with any other implementation that satisfies the condition, aiming to make the condition loose enough to admit the common extant implementations; or

4. try to describe the behaviour of some particular extant implementation(s).

Option 1 does not seem useful, as the natural-language standards do not cover many important details. One would end up either with a very loose specification or have to design the missing details. In either case the specification would not admit the common implementations, which diverge from some details that are expressed in the natural-language standards. The first case might be useful as a basis for finding certain bugs in implementations, but would not suffice either as documentation for users and implementors or as a basis for proofs about higher-level abstractions.

Option 2 would be of considerable intellectual interest, and might contribute to future protocol design by isolating precisely why some aspects of TCP behave well, but again would not be useful for users, implementors or theorists concerned with the real deployed systems.

Option 3 might be the ideal form of a specification, allowing a wide range of implementation differences but still ensuring interoperability. For example, it might impose tight constraints on flow control but allow a range of congestion control and retransmission behaviour. This is what one should aim at for future protocols. Given the complexity of TCP as it is, however, we regard this as a (challenging!) possibility for future work.

In the present work we follow 4, aiming to capture the *de facto* standard of some of the widely-deployed implementations. Our specification is based on the behaviour of three operating systems: BSD, Linux, and Windows XP. More specifically, it is based on particular versions: FreeBSD 4.6–RELEASE, Linux 2.4.20–8, and Windows XP Professional SP 1. Behavioural differences between the three are made explicit in the specification, in which hosts are parameterised on their implementation version.

This means there is a precise lower bound that the specification should satisfy: it should admit all the behaviour (respectively) of those implementations. There is not a corresponding precise upper bound, but only an informal one of utility — for example, the constant **T** satisfies the lower bound, but it is not a *useful* specification, either for documentation or as a basis for reasoning about programs that use the sockets interface.

While based on those particular versions, the specification is far from a simple restatement of their code. It is nondeterministic in many ways, allowing certain reasonable variations of behaviour, and is structured for clarity rather than executability and performance.

As we shall describe later in more detail, in §7, at the time of writing the UDP and sockets specification is reasonably accurate for all three implementations, while for TCP the specification is reasonably accurate for BSD. Some, but certainly not all, Linux and Windows XP differences have been captured.

1.5 Validation

A key part of our work is the development of automated techniques to ensure accuracy, validating the specification against the behaviour of the implementations. These techniques can also be used for conformance testing, comparing the behaviour of future implementations against the specification.

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),, the POSIX standard [IEE00], the Stevens texts [Ste94, WS95, Ste98], the 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. The host LTS defines a transition relation $h \xrightarrow{lbl} h'$ with labels as outlined in §1.4; we validate it by checking that the traces of this automaton include a wide range of real-world traces $lbl_1, \ldots, lbl_n$ captured from the implementations. In more detail, we:

1. set up a test network with machines running each of the implementation OSs;
2. instrumented the sockets API and the network to capture traces (represented as HOL lists of labels) consisting of timestamped socket calls and returns and wire message sends and receives;
3. generate a large number (some 6000) of such traces, covering a wide range of behaviour and for all three implementations, using a test harness that makes socket calls and injects packets as necessary;
4. wrote a symbolic evaluator for the specification within HOL, allowing it to check whether particular traces are included in the model; and
5. wrote tools to distribute this (computationally intensive) trace checking as background jobs over many machines, and to present the results in a usable form.

Developing the specification has been an iterative 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 *experimental semantics* approach enables us to produce an accurate description of the behaviour of complex pre-existing software artifacts, and seems to be the only practical way of doing so. In principle one could instead formally analyse the source code of the implementations to *derive* a semantic description (combining the results with a simple model of the network hardware). That implementation, however, consists of a great deal of intricate multi-threaded C code — even producing a language semantics for the fragment of C used would be a major task (cf. Norrish's partial C semantics [Nor98, Nor99]), let alone reasoning about the implementation above it.

We cannot, of course, claim total confidence in the accuracy of our specification — it certainly still does contain some errors. 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. Nonetheless, our automated validation against a formal specification is, by the standards of normal software engineering, an extremely demanding test. It is worth noting also that the fact that our symbolic evaluator is written within HOL means that *its* correctness depends only on the small HOL kernel. Each check that a trace is included in the model involves proving a theorem (in fully-expansive Edinburgh LCF style [GMW79]) to that effect.

The *post-hoc* nature of the specification contrasts with most other formal work in the literature. We are taking the behaviour of the implementations as primary, aiming to make precise the *de facto* standard that they embody, rather than taking an ideal specification as primary and checking that implementations conform to it. Our validation process is thus aimed at finding problems in the specification, not problems in the code. Nonetheless, in the process we have discovered a number of oddities (there is no precise sense in which they might be 'bugs') in the implementations; we discuss these in §9. We have also discovered many differences between the three implementations.

Our validation tools, however, could equally be used for conformance testing. Having established a high-quality specification, one could generate traces for a new implementation and check whether they are included in the specification. Cases where they are not would indicate either bugs in the new implementation or points at which the specification is tighter than desired (perhaps more BSD-specific).

Our symbolic evaluator is essentially a symbolic model checker, but in a rather different sense to that normally understood. We are working with an undecidable logic. Indeed, the specification has a very complex state space, requiring a substantial fragment of HOL to express — including sets, lists, finite maps, records, other user-defined datatypes, naturals, reals, and mod 2^{32} numbers. It makes heavy use of the HOL inductive definitions support, and involves nested and alternating quantifiers. On the other hand, the evaluator is only called on to determine the truth of essentially existential goals: *"can the model exhibit the following (experimentally observed) trace?"* (Contrast with the usual *"does every possible execution of the model lead to a safe state?"*) Moreover, the undecidability of the underlying logic is of marginal interest: we are confident that our models will not move beyond the decidable fragment, e.g. linear arithmetic. We use (undecidable) higher-order logic because of its superior expressiveness, even over this very concrete domain. The bulk of the specification is first-order, and the exceptions are mainly finite sets.

1.6 Overview

We begin in §2 with a discussion of what aspects of the real systems we model, and of how we do so. In §3 we describe our test instrumentation and trace generation infrastructure, showing how the real system behaviours are abstracted. §4 introduces the specification itself, with a quick introduction to the higher-order logic in which it is written, some of the key types, a few sample transition rules, and some sample checked traces annotated with the rule firings discovered by the checker.

Our validation process in described in §5 and §6, with the first of these describing our HOL symbolic evaluation engine and the second the checking infrastructure and visualisation tools. The current validation status of the specification is given in §7.

In §8 we present a very abstract view of the specification, a version of the 'TCP state diagram' that includes essentially all the possible transitions. Some of the most significant implementation oddities and differences we have discovered are presented in §9. We conclude with discussion of related work in §10, an overview of the project history in §11, and discussion in §12. This summarises our contribution, discusses how the specification could be used, outlines possible future work, and discusses how one might go about rigorous specification at design-time for future protocols and APIs.

2 Modelling

This section discusses modelling choices: what aspects of the real network system are covered, and how they are modelled. Our focus is on the discrete behaviour of the system. We therefore aim to have a clear —though necessarily informal— relationship between certain events in the real system and events in the model. That relationship is based on choices:

1. of where to cut — what system events are represented;
2. of what to abstract from those events; and
3. of how to restrict the system behaviour to a manageable domain.

The first two choices become embodied in the validation infrastructure, which must instrument the events which are represented as observable events in the model and must calculate their abstract views.

Focussing on the *behaviour* of the system, we do not attempt to establish any particular relationship (beyond what is required to get the behaviour correct) between the *states* of the system and of the model. This permits useful flexibility in modelling. If one had a more deeply instrumented system, in which all the state could be observed, expressing an explicit state abstraction function could be worthwhile, making validation more rigorous.

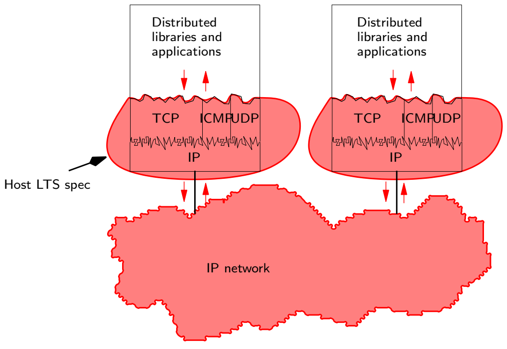
2.1 Where to cut

There are four main options for where to cut the system in order to pick out events.

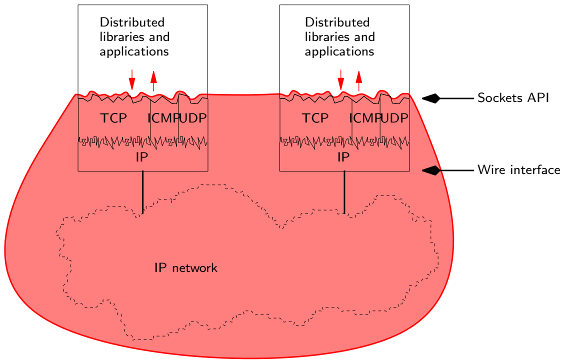
1. An *endpoint* specification deals with events at the Sockets API and at the network interface of a single machine. Our Host LTS is of this form, as shown on the left below. In these diagrams the shaded areas indicate the part of the system covered by the models, which may treat each shaded region as a black box, abstracting from its internal structure. The short red arrows indicate the interactions specified by the models, between the modelled part of the system and the remainder.

   Our specification is at a rather low level of abstraction (‘*segment-level*’), including many network interface event details —essentially of the TCP protocol as it exchanges TCP segments— which cannot be directly observed above the Sockets API.

2. An *end-to-end* specification, as shown on the right below, describes the end-to-end behaviour of the network as observed by users of the Sockets API on different machines, abstracting from the details of what occurs on the wire. For TCP such a specification could model TCP connections roughly as a pair of streams of data, together with additional data capturing the failure behaviour, connection shutdown, etc. It should be substantially simpler than a segment-level specification.


Distributed libraries and applications
Distributed libraries and applications
TCP
ICMP
UDP
IP
TCP
ICMP
UDP
IP
Host LTS spec
IP network

Endpoint specification (segment–level)

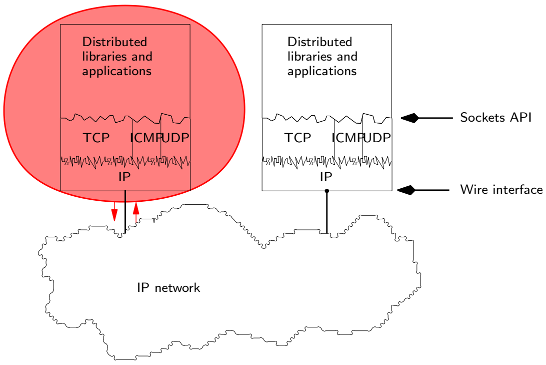

Distributed libraries and applications
Distributed libraries and applications
Sockets API
TCP
ICMP
UDP
IP
TCP
ICMP
UDP
IP
Wire interface
IP network

End–to–end specification (stream–level)

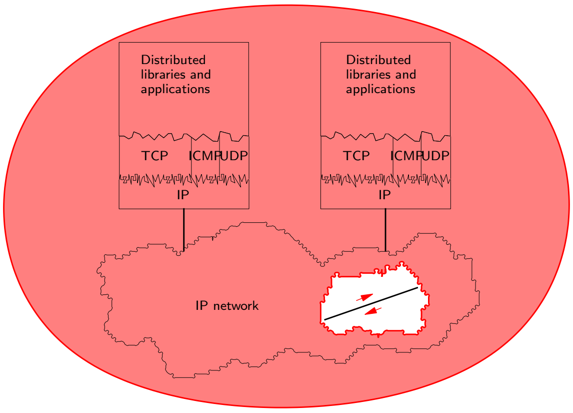
3. One could give an endpoint specification for the network interface only, as shown on the left below. This would aim to capture the wire protocol in a pure form, defining what the legal sequences of message input and outputs are for a single host.

4. Finally, a *network-interior* specification would define the legal sequences of messages for (say) a single TCP connection as observed at a point in the interior of the network, as shown on the right

below. Here there is the possibility of message loss, duplication, and reordering between the hosts and the monitoring point.


Distributed libraries and applications
TCP
ICMP
UDP
IP
Distributed libraries and applications
TCP
ICMP
UDP
IP
Sockets API
Wire interface
IP network

Endpoint specification (wire interface only)

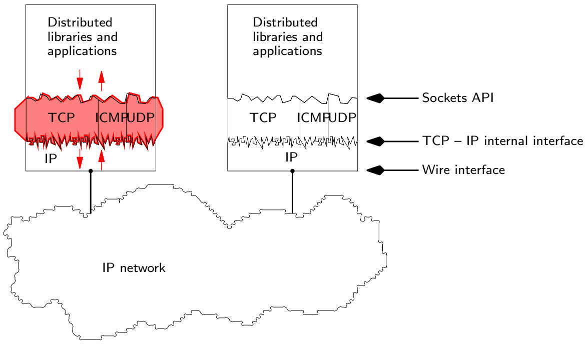

Distributed libraries and applications
TCP
ICMP
UDP
IP
Distributed libraries and applications
TCP
ICMP
UDP
IP
IP network

Network−interior specification

All four would be useful. Implementors of TCP/IP stacks might be most concerned with the endpoint specification; users of the Sockets API (implementors of distributed libraries and middleware) can largely think at the end-to-end stream level; protocol designers focussed exclusively on the wire protocol might want network-interface-only endpoint specifications; and workers in network monitoring might want a network-interior specification.

Given an endpoint specification one could in principle *derive* the other three. To build an end-to-end model one would form a parallel composition of two copies of the Host LTS together with a simple abstract model of the IP network (abstracting from routing and fragmentation but allowing segments to be delayed, reordered, duplicated, and lost), and hiding the network-interface interactions. (We defined such a model in our earlier UDP specification work; it could easily be ported to the current system, making it compatible with the Host LTS.) To build a network-interface-only endpoint specification one would take the Host LTS and existentially quantify over all possible sequences of Sockets API calls. To build a network-interior specification one would do both, forming a parallel composition of copies of the Host LTS, together with an abstract model of the IP network, and existentially quantify over all possible sequences of Sockets API calls at each host. For any of these, however, the result would probably be so complex as to be hard to work with, with an intricate internal structure. It might well be worthwhile constructing such specifications directly, which one would expect to be logically equivalent to the composite forms but much simpler.

There is also a fifth possibility we should mention: one could try to specify the behaviour of the TCP part of a host implementation, as depicted below.


Distributed libraries and applications
TCP
ICMP
UDP
IP
Distributed libraries and applications
TCP
ICMP
UDP
IP
Sockets API
TCP − IP internal interface
Wire interface
IP network

pure transport−protocol specification

Here the events in the model would be Sockets API interactions, as before, together with the interactions across the interface —internal to the TCP/IP stack— between the TCP code and the implementation of IP. This would have the advantage that one would be specifying the behaviour of a more tightly delimited body of code: the specification would not be taking into account the behaviour of the IP implementation, network hardware drivers, or network card itself. There could therefore be more scope for treating that implementation code rigorously, e.g. checking a specification corresponds to an automatically-generated abstract interpretation of the C code, or even proving that the implementation code does satisfy the specification. There are several major disadvantages. That internal interface is complex, differs radically between implementations, and is not instrumented. Moreover, the behaviour at that interface is not of *direct* interest to either users of the Sockets API or those who interact with the TCP/IP stack from across the network.

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 a *de facto* standard, with its behaviour of key interest to a large body of developers. Ambiguities, errors, and implementation differences here are just as important as for the wire protocol. Secondly, the failure semantics for a segment-level model is rather straightforward, essentially with individual segments either being delivered correctly or not. In the early part of the project we considered a stream-level end-to-end specification, but to develop an accurate failure semantics for that from scratch seemed too big an abstraction from the real systems, whereas given a correct segment-level specification it should be reasonably straightforward to produce an accurate stream-level specification based on it. 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 (without substantially perturbing the implementation).

2.2 Network interface issues

The network interface events of our model are the transmission and reception of UDP datagrams, ICMP datagrams, and TCP segments; as seen on the wire. In the actual systems these are embedded in IP packets, which in turn (typically) are encapsulated in Ethernet frames. We abstract completely from the Ethernet frame structure. We abstract from IP in two ways. Firstly, the model abstracts from most of the IP header data. It covers the source and destination addresses and (implicitly) the protocol and payload length, but abstracts from all other header fields. Secondly, the model abstracts from IP fragmentation. IP packets may be fragmented in order to traverse parts of the network with MTUs smaller than their size, to be reassembled by the final destination. Only if all fragments are delivered (within some bounds) is the entire packet successfully delivered; our specification essentially deals with the latter events.

The model also abstracts slightly from TCP options: a model TCP segment may have each of the options that we deal with either present or absent. This is a minor unfortunate historical artifact rather than a deliberate choice; it means that certain pathological TCP segments, e.g. with repeated options, cannot be faithfully represented in the model.

These abstractions are embodied in the `injector` and `slurp` tools of the validation infrastructure described in §3. The former takes model TCP segments, constructs IP packets, and puts them on the wire; the latter takes IP packets from the wire, reassembles them if necessary, and constructs model segments. The model types used to represent datagrams and segments are reproduced in §4.2.

Given the 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 we would like to describe host behaviour in response to malicious attack, as well as to unintended partial failure.

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 of the protocol stack and network card need be dealt with explicitly. In the model hosts have queues of input and output messages; each queue models the combination of buffering in the protocol stack and in the network interface.

2.3 Sockets interface issues and programming language bindings

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. This ensures the model is relevant to any application above the Sockets API. In reality, most applications use the API only in limited idioms — for example, urgent data is now rarely used. An interesting topic for future work, which one might combine with work on a stream-level specification, would be to characterise those idioms precisely and to make any simplifications of the model that become possible.

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 cut out 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 (even for the same socket) 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. This complexity arises from its use for diverse protocols, from the limitations of C types, from a concern with unnecessarily copying large bodies of data between application and OS address spaces, and from historical artifacts. 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);
```

Here the `s` is the listening socket's file descriptor, the returned `int` is either non-negative, for the file descriptor of a newly-connected socket, or `-1` to indicate an error, in which case the error code is in `errno`. The `addr` is a pointer to a `sockaddr` structure and `addrlen` is a pointer to its length. If the former is non-null then the peer address is returned in the structure and the length updated. In the model, on the other hand, accept() has type

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

taking an argument of type fd and either returning a tuple of type $\mathsf{fd} * (\mathsf{ip} * \mathsf{port})$ or raising an error. For simplicity the model accept() always returns the peer address, and all calls that may return errors check the appropriate return code. The model API has several other simplifications, including for example:

- The model socket : sock_type → fd takes just a single sock_type argument, which can be either SOCK_STREAM or SOCK_DGRAM.
- The variant forms `send`, `sendto`, `sendmsg`, `recv`, `recvfrom`, and `recvmsg` are collapsed into single send() and recv() forms. The former returns the unsent substring rather than a byte count.
- The `setsockopt` and `getsockopt` calls are split into multiple forms for the different types of arguments (boolean, integer, and time).
- The use of `read` and `write` (for IO to an arbitrary file descriptor) are not covered.

The abstraction from the system API to the model API is embodied in the `nssock` C library of the validation tools, described in §3. This provides the calls of the standard API, prefixed by `ns_`. They have (almost exactly) the same behaviour as the standard calls[1], and are implemented in terms of them, but in addition also calculate the abstract views of the call and return and dump them to a log.

The model is language-neutral: other language bindings to the Sockets API can be related to the model by identifying exactly what model API events correspond to each language invocation. For languages implemented above the C Sockets library, of course, it suffices to identify what C calls correspond to each language invocation.

In addition, we have an OCaml [L$^+$04] version of the API that is very close to that of the model, with almost identical types and pure value-passing. This is implemented as a wrapper above `nssock`, and so can also log events. The OCaml binding, shown in §4.3, raises exceptions for all errors.

2.4 Protocol issues

We work only with IPv4, not IPv6, though at this level of abstraction there should be little observable difference.

For UDP the specification deals only with unicast communication, not multicast or broadcast. This restriction was made for simplicity during our early specification experiments, and has persisted since then; we would not anticipate any difficulty in relaxing it.

[1] The exceptions are, for example, that the `ns_accept()` call will always acquire the peer address, though it will not return it to the application unless it was requested, and that the `getifaddrs()` call is direct on FreeBSD, not implemented on WinXP, and implemented on Linux with two `ioctl()`'s. For the few calls implemented with multiple real API calls like this there may be atomicity issues.

For TCP there are a number of protocol developments, e.g. of improved congestion control algorithms, and protocol options. Broadly we cover those in the BSD version (FreeBSD 4.6–RELEASE) used for validation. We include:

- the maximum segment size option
- the RFC1323 timestamp and window scaling options, and PAWS
- the RFC2581 and RFC2582 'New Reno' congestion control algorithms
- the observable behaviour of syncaches

As for the API, we include urgent data in the protocol, even though little new code should use it. We do not include the RFC1644 T/TCP variant (though it is in that codebase), the RFC2018 SACK selective acknowledgements, or RFC3168 ECN.

For ICMP we include the relevant types, i.e. for ICMP_UNREACH, ICMP_SOURCE_QUENCH, ICMP_REDIRECT, ICMP_TIME_EXCEEDED, and ICMP_PARAMPROB. The specification describes their behaviour, but this part has not at present been experimentally validated .

2.5 Nondeterminacy

The behaviour of a TCP/IP implementation is difficult to predict exactly, even if the code is known and the sockets and network interfaces are fully instrumented. This is for three reasons. Firstly, TCP is, in one particular respect, a randomized algorithm: the initial sequence number of a connection is chosen randomly to inhibit spoofing. Secondly, and more seriously for our purposes, the behaviour can strongly depend on the OS scheduling. This determines the relative sequencing of user threads (and the sockets calls they make), of the internal protocol processing, and of network interface interrupt handlers. Incoming messages may be queued within the network hardware and within the protocol implementation to be processed later; any output messages generated by that processing may likewise be queued and not appear on the wire for some time. The point at which they are processed may therefore not be externally observable. Thirdly, TCP implementations involve a number of timers; the exact points at which they time out are affected by scheduling and by the machine hardware.

A model that aims to include all real-world observable traces must therefore be a *loose* specification, nondeterministically permitting a range of possible behaviours. This requirement for nondeterminism has fundamental effects: on the language needed to express the specification and on the techniques that can be used to validate it against real-world traces. In designing future protocols one may wish to consider how nondeterminism can be limited — we return to this in §12.3.

Our specification is nondeterministic in several ways:

- It allows arbitrary sequencing of the possible events. For example, in a state with a pending return to a socket call, a pending message output, and a queued input message awaiting processing, any of the three may occur first.
- Initial sequence numbers are unconstrained.
- The protocol options offered by an endpoint at connection-establishment time are unconstrained.
- In cases where several different errors might be returned from a sockets call, e.g. if a UDP send() is to an illegal address and also has an illegally-long string of data, the choice of error is unconstrained. Individual implementations will generally be deterministic here, depending on which condition they test first, but modelling that exactly would complicate the specification and it seems unlikely that users of the API depend on the priority.
- Initial window sizes are weakly constrained.
- The rate of time passage is (as discussed below) only loosely constrained, allowing for clock fuzz.
- As mentioned above, blocking calls may nondeterministically return EINTR, as they would if interrupted by a signal. The model does not cover signals, so nothing more precise is possible.
- A number of errors, e.g. ENFILE, ENOBUFS, and ENOMEM, indicate exhaustion of some OS resource. The model does not (and should not) cover enough of the OS state to determine exactly when these might occur, and so allows them nondeterministically.

  One would expect most reasoning above the model to depend on an assumption that the resource limits are not reached, and these errors are guarded by an INFINITE_RESOURCES predicate to make that simple.

- The points at which TCP output may occur are only weakly constrained. This is a historical artifact, and is not hard (in principle) to fix. It means that the specification does not fully capture the 'ack clock' property, which is important for implementations.

Differences between the three OS implementations we consider are not dealt with by nondeterminism but by making the host specification parametric on an OS version. This seems simpler, given that in some cases different implementations have radically different behaviour. The resulting specification should be more useful in the (fairly common) case in which one is writing for a particular OS, and also highlights the differences between them sharply.

We conjecture that the level of nondeterminism should make the specification resilient in the face of many minor implementation changes, but this would have to be confirmed by experience in applying it to new OS versions.

It would be interesting to refine the specification, resolving the nondeterministic choices to the point where it expresses an implementation. Our validation technology (the symbolic evaluator of §5 and the `slurp` and `injector` tools of §3) should make it possible to execute such a refined specification in the network, albeit with a substantial slowdown. One could then test that it interoperates with existing implementations (with artificially-slowed protocol timeouts to match the speed of the evaluator).

2.6 Specification language

The form of the host LTS specification is simply 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, an internal transition, (with label τ), or a record of time passage (with label *dur*). The nondeterminism of the specification means that a given host state h may have transitions with many different labels, perhaps including τ and *dur*, and that for each label there may be many possible resulting host states.

The choice of specification language is very important. Our initial work [SSW01a, SSW01b] used rigorous but informal (non-mechanised) mathematics, in the style commonly used for definitions of the operational semantics of process calculi and programming languages. This proved unsatisfactory — even that specification was sufficiently large that keeping it internally consistent without at least automatic typechecking was challenging, and the translation of the specification into a form that could be used for validation was manual and error-prone. Several alternatives are conceivable:

1. One could write the specification explicitly as a conformance-checking algorithm in a conventional programming language.

2. One could write a more declarative specification in a typed logic programming language such as Lambda Prolog [Lam] or Mercury [Mer].

3. One could use a restricted logic for which conventional symbolic model checking is feasible.

4. One could embed the specification in a general-purpose proof assistant for mechanised mathematics, e.g. HOL [HOL], Isabelle [Isa], Coq [Coq] or PVS [PVS].

Option 1 might be the best way to produce a highly efficient conformance checker, but it would probably be a poor way to write the specification — the result would have algorithmic checking details, e.g. of search strategies and heuristics needed to deal with the nondeterminism, tightly intertwined with protocol specification.

Option 2 is plausible but we did not investigate it in detail. Potential issues are the extensive use of arithmetic constraints in TCP and the fine control of search strategies needed for validation. It also would not support machine-checked proof, either of properties of the specification or of code written above it.

Option 3 may be feasible for checking selected properties of implementations, but we suspect the complexity of the TCP state space and behaviour render it impractical as a way of writing a complete specification.

Finally, we are left with option 4, of using a general-purpose proof assistant. These support expressive logics in which more-or-less arbitrary relations can be defined. They also support the data structures needed for describing host states — tuples, records, lists, finite maps, numeric types, etc.— so expressing the specification is reasonably straightforward. Checking whether such a specification includes particular traces, on the other hand, is non-trivial. We return to this in §5.

In contrast to large-scale programming language definitions our specification involves no variable binding (except of the built-in logical quantifiers), the formalisation of which is still problematic, so it is particularly well-suited to mechanisation.

We use the HOL proof assistant, a mature and modular system with an extensive API, allowing the development of standalone tools on top of a rich logical context. Compared to other developments using the system, we use a high proportion both of the system's logical library (its types and associated theories, such as those of lists, numbers, sets and finite maps), and of its reasoning facilities: the simplifier and various arithmetic decision procedures. As is typical with mechanised computer science, we make heaviest use of HOL's concrete types. In most places, the specification is effectively written in a many-sorted first order logic, with typical exceptions being the notational convenience of set comprehensions, and standard functional-programming functionals such as "map" and "filter".

2.7 Specification idioms and process

The main part of the specification, after some preliminary type definitions and auxiliary functions and relations, is the definition of the transition relation. It is expressed as the least relation satisfying a set of rules, all abstractly of the form

$$\vdash P(h_0, l, h) \Rightarrow h_0 \xrightarrow{l} h$$

where P is a condition (on the free variables of h_0, l, and h) under which host state h_0 can make a transition labelled l to host state h. In contrast to most structural operational semantics definitions this is essentially flat — very few rules have a predicate that itself involves transitions. Rules are equipped also with names (e.g. *bind_5*), the protocol for which they are relevant (TCP, UDP, or both), and categories (e.g. *FAST* or *BLOCK*).

The partition of the system behaviour into particular rules is an important aspect of the specification. We have tried to make it as clear as possible with each rule dealing with a conceptually different behaviour, separating (for example) the error cases from the non-error cases. This means there is some repetition of clauses between rules. For example, many rules have a predicate clause that checks that a file descriptor is legitimate. Individual rules correspond very roughly to execution *paths* through implementation code, in which all paths share the same code for such checks. For substantial aspects of behaviour, on the other hand, we try to ensure they are localised to one place in the specification. For example, calls such as accept() might have a successful return either immediately or from a blocked state. The final outcome is similar in both, and so we have a single rule (*accept_1*) that deals with both cases. Another rule (*accept_2*) deals with entering the blocked states, and several others with the various error cases:

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

Similarly, the 'normal case' processing of TCP input segments in connected states is dealt with by a single rule (*deliver_in_3*) with successful connection establishment pulled into separate rules (*deliver_in_1* and *deliver_in_2*) and error cases in still other rules. Often an error-case rule will have the explicit negation of a clause from the corresponding successful rule.

The predicate P of a rule constrains variables that occur in the initial state h, the label *lbl* and the final state h'. Often it is useful to think of a part of P as being a 'guard', which is a sufficient condition for the rule to be applicable, and the remainder as a constraint, which should always be satisfiable, on the final state h'. This distinction is not formally present, however.

To structure the specification more clearly a number of details are pulled into auxiliary definitions. The relational nature of the specification permeates these too: many auxiliaries define relations instead of functions, expressing loose constraints between their 'inputs' and 'outputs'.

The reference implementations are expressed in imperative C code, and the early parts of segment processing can have side-effects on the host data structures, especially on the TCP control block, before the outcome of processing is determined. Disentangling this imperative behaviour into a clear declarative specification is non-trivial, and our *deliver_in_3* rule makes use of a relational monad structure to expose certain intermediate states (as few as possible).

The detailed design of the host state type is also a matter of craft. It is largely composed of typed records, together with option types, lists, and finite maps. The overall structure roughly follows that of the system state: hosts have collections of socket data structures, message input and output queues, etc.; sockets have local and remote IP addresses and ports, etc.; TCP sockets have a TCP control block, and so on. The details vary significantly, however, with our structures arranged for clarity rather than performance — recall that as we are specifying only the externally-observable behaviour, we are free to choose the internal state structure.

In designing the host state and the rules it is important to ensure that only the relevant part of the state need be mentioned in each rule, to avoid overwhelming visual noise. We use a combination of patterns (in the h and h') and of (functional) record projection and update operations.

There are many invariants that we believe the model satisfies, e.g. that certain timers are not active simultaneously, or that in certain TCP states the local and remote addresses are fully specified (not the wildcard value). Ideally these would be expressed in HOL, and we would carry out machine-checked proofs that they are preserved by all transitions. We did this for our earlier UDP HOL model [WNSS02], but have not attempted it for the TCP model described here, for lack of time rather than any fundamental difficulty. We expect the proofs would be lengthy but straightforward, as each rule can be considered in isolation.

By working in a general-purpose proof assistant we have been able to choose specification idioms almost entirely on clarity, not on their algorithmic properties. As we describe in §5, our automated validation has involved proving theorems relating the specification as written to more tractable forms. This is a very flexible approach, which has been successful —and is perhaps necessary— for the rather complex system we are working with, but it requires considerable HOL expertise.

2.8 Relationship between code structure and specification structure

We aim for the specification to be clear, coherent, modular, and extensible. The implementation code is intended to be fast and extensible, but coherence and modularity have suffered from the multi-decade, multi-author development process. In particular, changes have been incremental and localised, resulting in growth by accretion. Developing the specification has involved untangling this complexity and (as far as we can) revealing an underlying simplicity.

The code has a basic layered structure. For example, consider the FreeBSD implementation of socket close (the Linux implementation is similar). A *user program* invocation of the API call `close` is first handled by the *C library*, which invokes the underlying system call. The kernel's *system call handler* dispatches the call to the *kernel* `close` function, which updates the relevant kernel structures and, if this is the last descriptor for the file, invokes the file-type-specific `fo_close` operation. For a socket, this is the *sockets layer* `soclose` function. This updates the relevant socket structures and, if the socket is connected, invokes the protocol-specific `pru_disconnect` operation. For a TCP socket, this is the *TCP layer* `tcp_usr_disconnect` function. This performs the protocol close operation, and updates the relevant protocol structures. Some of the updates are performed by the *Internet layer*'s `in_pcbdetach`. Emission of a RST or FIN segment is performed by the *IP layer*'s `ip_output`, which constructs one or more fragments and invokes the interface-specific `if_output` function. For Ethernet, this is the *interface layer* `ether_output` function, which constructs an Ethernet frame and invokes the NIC-specific driver function. Wire events (incoming segments) follow roughly the reverse path.

However, this layered structure is not strictly adhered to. The interfaces between layers are broad, and upper layers frequently make assumptions about lower layers, perform work for them, or even store information strictly belonging to a one layer in another layer's structures (in either direction); calls also occur in both directions. In particular, there is a tight coupling between the Internet, TCP, and IP layers, and significant coupling between the TCP and sockets layers.

Even within a layer, a single logical operation is usually implemented by multiple functions scattered across several source files.

The specification models everything from the C library to the TCP layer, including a little of the Internet and IP layers. We preserve the kernel/sockets/protocol division to some extent, as this is a useful one, but collapse the remaining distinctions. The C library and kernel behaviour are specified first, supported by various of the host structures, in particular the file descriptor map *fds*. The sockets layer is next, supported by the *socks* map. The specification of these layers is based on POSIX. Beneath the sockets layer is the protocol behaviour itself, modelling the TCP (and UDP), Internet, and (partial) IP layers and supported by the *cb* structure. The specification is here based on the RFCs and on Stevens. A fuzzy line can be drawn between rules of these three layers: system-call rules that interact at most with *fds*; system-call rules that interact with *socks* but go no deeper; and protocol rules which manipulate protocol-specific structures. Only a few API rules fall into the third category.

In structuring the specification, we have attempted to give one rule for each possible behaviour, with rules clustered by system call or protocol operation. Each rule is as far as possible declarative: it defines a relation between inputs and outputs, with intermediate states and internal state variables introduced only where necessary. The historical and pragmatic idiosyncrasies of the code mean that the sometimes these *are* necessary in order to capture the observable behaviour.

We also remove irrelevancies and artifacts: kernel memory management, booleans that are always false, options and extensions that are never enabled or out-of-scope (T/TCP, IPsec, firewalling), redundant flags which are always set or cleared in a particular state, the TCP input fast-path code, and so on. Sometimes complete removal is impossible: for example, the FreeBSD fast-path code neglects to update certain state variables, forcing us to model the condition that determines whether a segment should be treated on the fast path or not. However, wherever possible we preserve the standard names (from RFCs, BSD, POSIX) for variables, for ease of understanding and to aid in white-box testing.

An important question is the level of temporal granularity, or what the atomic state changes are. For a fast system call we choose to regard all effects as occurring instantaneously at the moment of invocation, although the return may be delayed. For a slow system call there is typically one atomic step in which it enters a blocked state but there are few other changes, a second step in which it is unblocked (e.g. when data becomes available), and a third in which the result is returned. Network activity is again essentially atomic: rule *deliver_in_99* receives a message from the network and adds it to a host's input queue; in rule *deliver_in_1* an incoming SYN segment instantaneously initiates a new connection and enqueues an outgoing SYN,ACK segment; and *deliver_out_99* deals with sending messages from the host's output queue to the network.

The imperative C code, on the other hand, modifies state in a series of incremental changes through the course of a function. Usually the intermediate states are not observable, but they are sometimes revealed by failure (execution stops part-way through a function, updating some variables but not others) or concurrency (a shared variable is altered by another thread during the course of execution of a function). Thankfully the latter is rare, since most of the network protocol code is guarded by a coarse-grained lock and so does not in fact run concurrently; applying a degree of fuzziness to all times and deadlines absorbs almost all the remainder. The former is modelled by an atomic change in the success case, and a composed reverse update in the failure case (e.g. for IP datagram send, where failure causes post-transmission updates are skipped, the reverse update is performed by rollback_tcp_output).

Certain transitions that might in principle be modelled atomically are more naturally separated into multiple atomic transitions. For example, return from a slow call normally involves two transitions: a silent τ transition changing the blocking state into a returning state, and a return transition returning the result to the caller. This is done to reduce duplication, and to improve modularity of the specification. In situations where non-duplication and modularity require such a separation, but the semantics requires atomicity, our model of time allows the second transition to be labelled urgent, forcing the two to happen at the same instant of time; note however that other transitions may still be interleaved as long as they are enabled at that instant. In general, though, we have few τ transitions.

Gathering up the incremental changes through the course of a C function into a single atomic change becomes difficult when several successive computations and conditional branches occur, each dependent on some subset of those preceding. For the complicated *deliver_in_3* rule, and to a lesser degree for tcp_output_really, we were not able successfully to merge all the state changes — disentangling the places where a conditional or computation depends on state variables whose values have been changed from their initial values proved either too difficult or to lead to an unintelligible specification. Instead the transition is divided into a small number of mutations composed together sequentially by a relational monadic combinator (see Section 4.9). The transition is still atomic, but its definition is composed of smaller parts. (It may be that a two-layer transition system, with a *deliver_in_3* transition composed of multiple mini-transitions, would be more perspicuous; we leave this for later work.)

It is important to note that this complexity is only necessary because we cover what the implementations *actually do* — a redesign of TCP in our specification style would not require such contortions, and would we believe be significantly simpler. It may be worthwhile to structure the specification differently, giving first that simpler specification, in conjunction with a complex but separate term constraining the actual behaviour as appropriate. We leave this to future work.

Ideally we would be able to give a specification with a clear modular structure, e.g. with the congestion control algorithms factored out, and allowing the determinisation mentioned above (Section 2.5) to be applied cleanly. The use of relational auxiliaries and monads has taken us some way towards this, but the protocol RFCs and the implementation code are not modular in this way; it is unclear how much further one can go.

2.9 Time

We precisely model the timing of sockets API and wire interactions. Much TCP behaviour is driven by timers and timeouts, and distributed applications generally depend on timeouts in order to cope with asynchronous communication and failure.

Our model bounds the time behaviour of certain operations: for example, the failing bind call of Section 4.6 will return after a scheduling delay of at most dschedmax; a call to pselect with no file descriptors specified and a timeout of 30sec 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. For some of these requirements time is essential, and for others time conditions are simpler and more tractable than the corresponding fairness conditions [LV96, §2.2.2].

Timed process calculi have been much studied over the last decade or so; Yi's *Timed CCS* [Yi91] was one of the earliest, and a brief survey can be found in [Sch00, §10.4]. We draw primarily on Lynch et al.'s general automaton model for timing-based systems [LV96, SGSAL98]. Following the majority of recent work, we use a two-phase model: a labelled transition system in which the usual discrete action labels are augmented with time passage labels d, where d is a nonzero element of some time domain. Discrete actions are performed instantaneously, and a trace is a sequence of discrete and/or time-passage labels. We take our time domain to be the nonnegative reals.

Time passage is modelled by transitions labelled $d \in \mathbb{R}_{>0}$ interleaved with other transitions. In a system composed from multiple components, these labels use multiway synchronisation, modelling global time which passes uniformly for all participants (although it cannot be accurately observed by them).

Certain states are defined as *urgent*, if there is a discrete action which we want to occur immediately. This is modelled by *prohibiting* time passage steps d from (or through) an urgent state. We have carefully arranged the model to avoid pathological timestops by ensuring a local receptiveness property holds.

The model is constructed to satisfy the two time axioms of [LV96, §2.1]. Time is *additive*: if $s_1 \xrightarrow{d} s_2$ and $s_2 \xrightarrow{d'} s_3$ then $s_1 \xrightarrow{d + d'} s_3$; and time passage has a *trajectory*: roughly, if $s_1 \xrightarrow{d} s_2$ then there exists a function w on $[0, d]$ such that $w(0) = s_1$, $w(d) = s_2$, and for all intermediate points t, $s_1 \xrightarrow{t} w(t)$ and $w(t) \xrightarrow{d - t} s_2$. These axioms ensure that time passage behaves as one might expect.

The timing properties of the host are specified using a small range of *timers*, each with a particular behaviour. A single transition rule *epsilon_1* models time passage, duration d say, by evolving each timer in the model state forward by d. If any timer cannot progress this far, or the initial model state is marked as urgent for another reason, then the rule fails and the time passage transition is disallowed. Note that, by construction, the model state may only become urgent at the expiry of a timer or after a non-time-passage transition. This guarantees correctness of the above rule. The timers are explained in detail in Chapter 18; briefly, they are the *deadline timer* which expires nondeterministically inside a specified interval (preventing time from progressing further), the *time-window timer* which is similar but simply clears itself on expiry without becoming urgent, the *ticker* which ticks at a fuzzily-specified rate, incrementing a 32-bit counter at each tick, and the *stopwatch* which records the time elapsed since it was reset. A second transition rule *epsilon_2* models unobservable (τ) transitions occurring within an observed time interval. This machinery ensures the specification includes the behaviour of real systems with (boundedly) inaccurate clocks.

The actual time intervals used are parameters of the model: these include the maximum scheduling delay dschedmax, the maximum input-queue scheduling delay diqmax, the tick rate parameters tickintvlmin and tickintvlmax, the granularities of the slow, fast, and kernel TCP timers, the RFC1323 timestamp validity period dtsinval, the maximum segment lifetime TCPTV_MSL, the initial retransmit time TCPTV_RTOBASE, and so on.

Many timed process algebras enforce a *maximal progress* property [Yi91], requiring that any action (such as a CCS synchronisation) must be performed immediately it becomes enabled. We found this too inflexible for our purposes; we wish to specify the behaviour of, e.g., the OS scheduler only very loosely, and so it must be possible to nondeterministically delay an enabled action, but we did not want to introduce many nondeterministic choices of delays. Our calculus therefore does not have maximal progress; instead we ensure timeliness properties by means of timers and urgency. Our reasoning using the model so far involves only finite trace properties, so we do not need to impose Zeno conditions.

2.10 Network model

The host LTS may be combined with other host and application LTSs and embedded within a network model, in order to form a complete system model. We have done this for our previous UDP/IP model [WNSS02]; the present extension to TCP should be handled in exactly the same manner. In the present section we briefly summarise that model. It covers loss, reordering and duplication of IP while abstracting from IP routing.

A network is a parallel composition of IP messages in transit (on the wire, or buffered in routers) and of machines. Each machine comprises several *host components*: the OS state, executing threads, store, etc. To simplify reasoning, we bring all these components into the top-level parallel composition, maintaining the association between components of a particular machine by tagging them with a host identifier.

The semantics of a network is defined as a labelled transition system of a certain form. It uses three kinds of labels: labels that engage in binary CCS-style synchronisations, e.g. for invocation of a host system call by a thread; labels that do not synchronise, e.g. for τ actions resulting from binary synchronisations; and labels on which all terms must synchronise, used for time passing, hosts crashing and programs terminating. Parallel composition is defined using a single synchronisation algebra to deal with all of these. In contrast to standard process calculi we have a *local receptiveness* property: in any reachable state, if one component can do an output on a binary-sync label then there will be a unique possible counterpart, which is guaranteed to offer an input on that label. This means the model has no local deadlocks (though obviously threads can block waiting for a slow system call to return).

Message propagation through the network is defined by the rules below. A message sent by a host is accepted by the network with one of three rules. The normal case is *net_accept_single*:

$$\mathbf{0} \xrightarrow{n \cdot \overline{msg}} (msg)_{\mathrm{dprop}}$$

The timer dprop models propagation delay by expiring nondeterministically at some time within the range [dpropmin, dpropmax]. Time aside, this treatment of asynchrony is similar to Honda and Tokoro's asynchronous π-calculus [HT91]. Message propagation is modelled simply by time passage. Once the message arrives, it may be emitted by the network to a listening host, by rule *net_emit*:

$$(msg)_0 \xrightarrow{\overline{n \cdot msg}} \mathbf{0}$$

This rule is only enabled at the instant the timer expires, modelling the fact that the host has no choice over when it receives the message. Note that the network rules do not examine the message in any way – it is the host LTS that checks whether the IP address is one of its own.

Messages in the network may be reordered, and this is modelled simply by the nondeterministic propagation times. They may also be finitely duplicated, or lost. Rule *net_accept_dup* is similar to *net_accept_single* above except that it yields $k \geq 2$ copies of the message, each with its own independent propagation delay timer; rule *net_accept_drop* simply absorbs the message:

$$\mathbf{0} \xrightarrow{n \cdot \overline{msg}} \prod_{1}^{k} (msg)_{\mathrm{dprop}} \qquad k \geq 2$$

$$\mathbf{0} \xrightarrow{n \cdot \overline{msg}} \mathbf{0}$$

3 Validation — Test Generation

This section describes the tools we developed for capturing real-world traces: a test network; instrumentation and test generation infrastructure; and the actual test generation itself. We describe the test infrastructure before describing the specification itself in more detail to clarify exactly which aspects of the real-world behaviour are the subject of the specification.

3.1 Trace generation infrastructure

To generate traces of the real-world implementations in a controlled environment we set up an isolated test network, with machines running the three OSs we consider, FreeBSD, Linux, and WinXP. At present it is configured as in Figure 2. Traces are HOL-parsable text files containing an initial host state (describing the interfaces, routing table, etc.), an initial time, and a list of labels (as introduced in §1.4 and given in detail in §4.4). An example trace is shown in Figure 3. Skipping over the preamble comments and

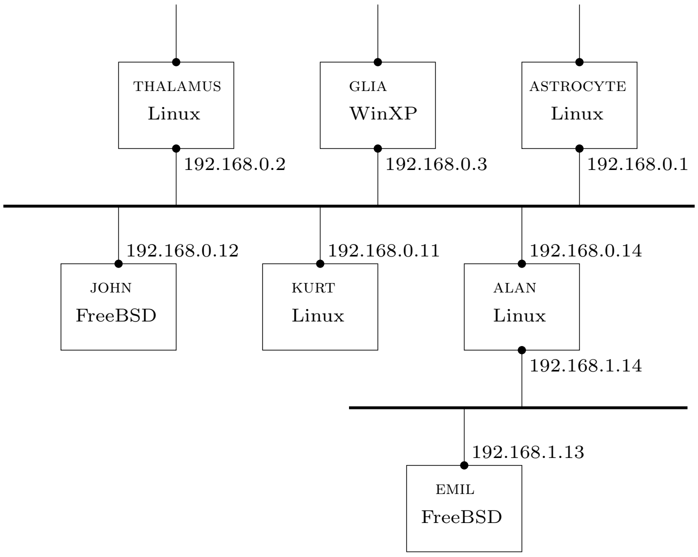


Figure 2: Test Network

initial host, it has just three labels: a call to `socket`, an `Lh_epsilon` label indicating time passage, and a return to the thread that made the socket call.

Generating traces requires a number of tools to instrument and drive the test network, with the main components shown in Figure 4. They are largely written in OCaml, with additional C code and scripts as necessary. The tools instrument the Sockets API, the network, and (on FreeBSD) certain TCP debug records, merging events from all three into a single trace. To instrument the Sockets API we have a C library `nssock` which provides the system calls of the standard API, prefixed by `ns_`. The calls have the same behaviour as the standard calls, and are implemented in terms of them, but in addition dump call and return labels (timestamped) in HOL format to a listening socket. Above this is an OCaml library `ocamllib` which provides the OCaml Sockets API as given in §4.3. This library is a thin layer that converts between C types and OCaml types, abstracting from the complexities of the (pointer-passing) C interface. The OCaml types are very close to the HOL types used in the model, as defined in `TCP1_LIBinterfaceScript.sml`. Instrumenting the network is done with an OCaml program `slurp`. This uses (an OCaml binding for) `libpcap` to capture traffic and outputs timestamped HOL-format labels to a listening socket. It performs reassembly of IP-fragmented packets.

To drive the network we have a LIB call daemon `libd` and a datagram injector daemon `injector`. The first allows `ocamllib` socket calls to be performed remotely. It listens (on a TCP or Unix domain socket) for HOL-format call labels, performs the calls, and also outputs any return labels. The second allows datagrams to be injected into the network. Again it is controlled by sending HOL-format labels, here of the form `Lh_senddatagram ...`, along a TCP or Unix domain socket. It uses an OCaml binding for raw sockets to perform the actual injection.

In addition, the BSD kernel, when compiled with `TCP_DEBUG` on, provides debugging hooks into its TCP stack. A socket with the option `SO_DEBUG` set inserts debug records containing IP and TCP headers, a timestamp and a copy of the current TCP control block into a ring buffer at certain checkpoints in the code. We have a tool `holtcpcb-v8` that inspects the ring buffer and (again) outputs HOL-format labels, consisting mostly of data from the recorded TCP control block. Early versions were based on the BSD `trpt` tool.

To control these various components we have an executive `tthee`, which provides a clean API for starting the tools on remote machines, sending commands and receiving back results. Results are often sent back twice: once over the tools logging channel so that they can be merged into a resulting trace, and separately over the tools command channel so that results can be parsed and passed back via `tthee` to the caller.

During a typical test run, several tools are invoked, and the results are merged in corrected chronological order to produce a trace. `tthee` abstracts from the sockets-layer control of the tools, uses `ssh` (or, on windows, a `custom_rsh`) for remote invocation, sets up logging channels, allows callbacks to be registered for individual log channels (so tests can make `tthee` requests based on previous observed behaviour of the system), and merges the results. The tools must let one deal correctly with blocking system calls

```
(* Test Host: LINUX(alan)  Aux Host: BSD(john) *)
(* Test Description: [TCP normal] socket() -- call socket() and return normally with a fresh file descriptor *)

(* -------------------------------------------------------------------------- *)
(* Netsem logging & merging tool (mlogger) - Date: Fri Jul 30 14:27:45 2004   *)
(* -------------------------------------------------------------------------- *)
(* ns_socket library initialised: connected to 192.168.0.2:55105 *)
(* Date: Fri Jul 30 14:27:50 2004 Host: alan *)
(* NTP STATUS:
status=06f4 leap_none, sync_ntp, 15 events, event_peer/strat_chg,
offset=-0.281
 *)
(* -------------------------------------------------------------------------- *)
(* Applying NTP offset: -281us *)
(* -------------------------------------------------------------------------- *)

(* -------------------------------------------------------------------------- *)
(* HOST *)
initial_host (IP 192 168 0 14) (TID 20595) (Linux_2_4_20_8) F []
[(ETH 0, <| ipset := IP 192 168 0 14; primary := IP 192 168 0 14; netmask := NETMASK 24; up := T |>);
 (ETH 1, <| ipset := IP 192 168 1 14; primary := IP 192 168 1 14; netmask := NETMASK 24; up := T |>);
 (LO, <| ipset := IP 127 0 0 1; primary := IP 127 0 0 1; netmask:= NETMASK 8; up := T |>)]
[<| destination_ip := IP 127 0 0 1; destination_netmask := NETMASK 8; ifid := LO |>;
 <| destination_ip := IP 192 168 0 0; destination_netmask := NETMASK 24; ifid := ETH 0 |>;
 <| destination_ip := IP 192 168 1 0; destination_netmask := NETMASK 24; ifid := ETH 1 |>]
initial_ticker_count initial_ticker_remdr
(* TSOH *)
(* Injector: not running *)
(* -------------------------------------------------------------------------- *)




(* BEGIN  *)

(* BASETIME *)
abstime 1091194070 545607
(* EMITESAB *)




(** 1091194070.545607 "ns0" **)
(* Merge Index: 0 *)
 Lh_call(TID 20595, socket(SOCK_STREAM));

(* Merge Index: 1 *)
Lh_epsilon(duration 0 32362);

(** 1091194070.577969 "ns1" **)
(* Merge Index: 2 *)
 Lh_return(TID 20595, OK(FD 7));
```

Figure 3: Sample trace

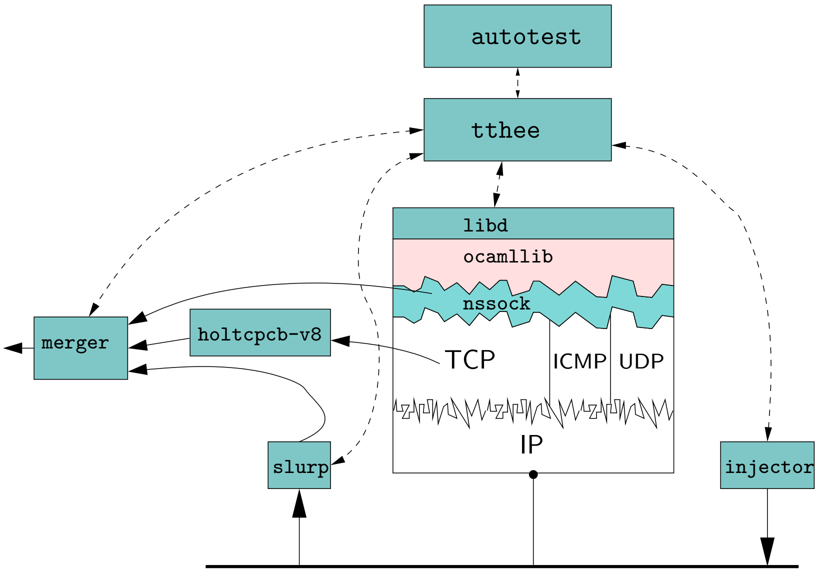


Figure 4: Test Instrumentation

that do not return during a test; `tthee` uses asynchronous calls to the other tools and carefully pairs up calls and returns or drops unwanted return values.

Merging events from the different sources into a correct order is problematic, requiring high-accuracy timestamps. To synchronise clocks `ntp` is used on each host. Each test tool that produces instrumented output reports the current `ntp` offset at the time it was started. These are used by `tthee` to correct the results from individual tools before merging into chronological order. Merging must also account for systematic delays, due to propagation delays or system scheduling issues. The merger can correct for these delays through a caller-specified correction factor. Once messages are finally corrected, merging proceeds in strict chronological order. This is done by an on-line merging algorithm (rather than a batch algorithm) so that long traces could be handled, leading to further complications that we do not discuss here. The final merged output is written to an output channel provided by the caller.

Timestamping BSD debug records required further work, as the standard BSD 4.6 kernel does not use high-precision timestamps for the events recorded in the TCP debug ring buffer. We constructed a set of patches to fix this problem; the test network BSD hosts are built with these patches applied and `TCP_DEBUG` enabled.

On WinXP timing has presented a problem. The `libd` and `slurp` tools need to have a consistent view of time: if they do not then the traces may have a datagram observed on the wire by `slurp` before a send() call was made by `libd` or similarly a successful return from a recv() call with data, before the datagram containing that data had been observed on the wire. Due to the inaccuracy of the WinXP clock used to record when datagrams were seen on the wire by the network card, the `slurp` tool was not providing accurate timestamps for datagrams. To fix this would require modifying the device driver to use accurate timestamps. This problem is still unresolved but has not stopped WinXP validation: of over 800 WinXP UDP traces just 36 exhibit this problem.

The BSD TCP debug trace records permit earlier resolution of nondeterminism in the trace checking process, reducing the amount of backtracking required, but three issues complicate their use. Firstly, not all the relevant state appears in the trace record; secondly, the model deviates in its internal structures from the BSD implementation in several ways; and thirdly, BSD generates trace records at various points in the middle of processing messages. These mean that in different circumstances only some of the debug record fields correspond to fields in the model state. We threfore compare trace records with the model's TCP control block with a special equality that only inspects certain fields, leaving the others unconstrained. Moreover, the is_1, ps_1, is_2, ps_2 quad is not always available, since although the TCP control block is structure-copied into the trace record, the embedded Internet control block is not. However, in cases where these are not available, the *iss* should be sufficiently unique to identify the socket of interest.

3.2 Tests

The tests themselves are scripted above the Ocaml library autotest and are run by the program of the same name. The autotest library allows the properties of each host in a test network to be described (e.g., the names of interfaces, the assigned IP addresses and the port values that lie within different ranges), and provides a common library above which tests can be written and managed.

The library provides functionality to start tools on specific hosts in a uniform way, each of which may merge their output to one or more different traces. Once the tools are running, raw tthee calls can be made to perform socket calls, inject segments or register a callback for slurped segments. The results from these actions are returned to the test (as well as having the equivalent result written to the trace), for the test to analyse and base the decision of what to do next or when to do it. The callback support provided by tthee is most useful for receiving slurped segments; the test can analyse these and perform an appropriate action, which in the common case is injecting a raw "spoofed" segment in reply. This behaviour enables tests that involve a real TCP implementation talking to a virtual host to be written.

The autotest library also defines functions for creating a fresh socket on one of the test hosts and then manipulating that socket into one of a pre-defined set of useful states. Some socket states are achieved simply by performing a sequence of socket calls, e.g. bind() then listen(), the most common and interesting of which are pre-defined. Other socket states require more control than that provided by the socket's interface. In these cases the host being instrumented communicates to a virtual host whose behaviour is faked by the test script. This involves registering a slurper callback function that upon receipt of segments may inject an appropriate reply segment into the network. This technique is used to manipulate a TCP socket into a plethora of useful states, from being simply connected to another host to having experienced congestion because segments were 'lost', or more precisely were not actually injected by the virtual host. The virtual host also has the ability to inject illegal segments. The virtual host used in our tests always assumed the same configuration (with an IP address 192.168.0.99 and name psyche).

Each test is described by a simple script which is a quadruple of a test thunk, a description of the test in English and two tuples which describe the test environment, i.e., which tools to start on each machine.

The autotest program contains a list of tests to run and performs each test on every test host it has been told about, writing the results from each with its description and initial host descriptions into a file.

The test thunks themselves typically use one of the functions in the autotest library to create a socket in a given state on the host under test, e.g., it may request a socket in the CLOSING state that has only ever received data, not sent any. The library function proceeds by making an appropriate sequence of socket calls coupled with any virtual host behaviour, before returning control to the test script. The script then continues to perform the test on the socket in that state by making socket calls through libd and may use slurper callbacks and injection to mimic its own virtual host. A simple test script:

```
let test1 = (
  (function id -> function ts ->
    let fd = get_local_bound_socket (the ts.th_libd_hnd)
        (Some (mk_ip id.test_host.test_ip_address))
        (Some (mk_port id.test_host.test_ephm_port)) in
    let listen_call = (tid_of_int_private 0, LISTEN(fd, 3)) in
    ignore(libd_call (the ts.th_libd_hnd) listen_call)),
  "listen() --  get a fresh bound socket, call listen() with a backlog of 3",
  ((true, false), true, true, false),
  ((false, false), false, false, false)
 )
```

obtains a socket locally bound to an ephemeral port on the test host, constructs a listen() call, and performs it, before autotest shutdowns everything down cleanly again.

More complicated tests that use the virtual host, psyche, may include a slurper callback function which could be as simple as:

```
 let slurper_callback_fun holmsg =
      match holmsg with
         TCPMSG(datagram) ->
           if (List.length datagram.data > 0) &&
             (datagram.is1 = Some(hol_ip id.test_host.test_ip_address)) &&
             (datagram.is2 = Some(hol_ip id.virtual_host_ip)) &&
          (datagram.ps2 = Some(uint id.virtual_host_port))
        then
          let ack_datagram = {datagram with
                              is1 = datagram.is2;
                              is2 = datagram.is1;
                              ps1 = datagram.ps2;
                              ps2 = datagram.ps1;
                              sYN = false;
                              aCK = true;
                              fIN = false;
                              seq = datagram.ack;
                              ack = datagram.seq +.
                                (uint (List.length datagram.data)) +.
                                if datagram.fIN = true then (uint 1) else (uint 0);
                              ts = timestamp_update_swap datagram.ts (uint 1);
                              data = []} in
          injector_inject (the ts.th_injector_hnd) (TCPMSG(ack_datagram));
          last_dgram := RECV_DATAGRAM(ack_datagram)
        else ()
    | _ -> ()
  in
```

which matches a segment from a given IP address, to a given IP address and port, with at least one byte of data, and injects an ACK segment in reply.

The interface provided by `autotest` allows tests to be written quickly and clearly with the minimum clutter.

Writing tests is a difficult task as hosts have an infinite and complex state space, so there can be no question of exhaustive testing. There are two different approaches to writing tests, both of which are necessary but test different behaviours. The first involves a test setup with two machines, one of which is being instrumented, and a socket is connected (or not) between them. The test script drives a sequence of socket calls at both ends, recording the socket calls and network behaviour observed by the host under observation. This approach when performed between varying machine architectures provides traces of the common-case interactions between the architectures, and the behaviour of their TCP stacks and sockets layer.

The second style of test is between a test host and a virtual host. Here socket calls are performed on the test host and from time to time, perhaps driven by a segment emitted from the test host, the virtual host injects segments destined for the test host (or other hosts). This permits tests that are not easily, if at all, producible deterministically through the use of the sockets layer, e.g., producing segment re-ordering or loss, adding in transmission delays or emitting or replying with illegal or nonsense segments. By use of a virtual host a more interesting set of tests beyond the common-case behaviour can be performed that test the inner workings of the TCP stacks and their failure modes.

We have written tests to, as far as possible, exercise all the interesting behaviour of the Sockets API and protocols. Tests are almost all run on all three architectures (FreeBSD, Linux, WinXP). Many tests are iterated over a selection of initial TCP socket states, e.g. for a local port that is unspecified, privileged, ephemeral, or other, or for a socket within each different TCP state. The `autotest` program has a few hundred lines of test code for each socket call, and a few thousand to exercise the *deliver* rules; it generates around 6000 traces. A small sample of trace titles is below. As we discuss later, trace checking is computationally expensive. This number of traces is around the upper limit of what our current tools and resources can handle.

TCP trace0000	BSD(john) Aux Host: LINUX(alan): [TCP normal] socket() – call socket() and return normally with a fresh file descriptor
TCP trace1000	BSD(john) Aux Host: LINUX(alan): [TCP normal] accept() – for a non-blocking socket in state CLOSE_WAIT(data sent rcvd), call accept()
TCP trace2000	LINUX(alan) Aux Host: BSD(john): [TCP normal] *deliver_in_3* – in state FIN_WAIT_2(data sent rcvd), virtual host sends a segment to the test host and waits for its acknowledgement. It then sends a segment that lies completely off the right hand edge of the window just advertised by the test host
UDP trace0500	WINXP(glia) Aux Host: BSD(john): [UDP normal] bind() – call bind(*fd*, REMOTE_IP, UNAVAILABLE_PORT) on un-privileged UDP socket with bound quad WILDCARD_IP, KNOWN_PORT, WILDCARD_IP, WILDCARD_PORT
UDP trace1500	WINXP(glia) Aux Host: BSD(john): [UDP normal] send() – for a blocking UDP socket with binding quad WILDCARD_IP, KNOWN_PORT_ERR, WILDCARD_IP, WILDCARD_PORT, attempt to send() data

The test tool infrastructure takes longer than a typical host would to execute commands, analyse results and inject packets. A few tests require a fast response, and in these the test script is written to inject segments early (the author having statically predicted what to inject) in order that events occur in time. Some test scripts have statically defined delays in them (`sleep()`'s). These are needed where the test script performs some action on the socket which is not directly observable by the test script (i.e. produces no observable tool behaviour, or emits only BSD debug record(s), which are not passed to the test script) but must have happened before the next event occurs.

Tests are segregated into normal and exhaustive tests. Most tests are classified as normal and are run and checked routinely. A few tests are marked as exhaustive (e.g. those that exhaust all of a processes file descriptors) — these are run separately and at present may not be checked due to their demanding nature.

3.3 Coverage

As for coverage, it is straightforward to check how many times each of the host LTS rules has been used in a check run (over the entire trace set). This has prompted the addition of a few more tests. It shows that almost all the rules are covered, with the common-case rules exercised several hundred times. One cannot write tests without knowing what is being tested, but if it is too familiar it is easy to build in wrong assumptions and miss a set of interesting cases. In general it seemed better to write tests for a given feature from a high-level understanding of the feature rather from a reading of the rule or the source code for it. Of course, afterwards it is still worthwhile to read the code and rule to add anything that may have been overlooked. Note that if a test script is wrong and is not testing what you think it is testing, it still may be useful — whatever the observed behaviour is should be accurately modelled. Some interesting behaviour has been found from tests that are not quite right.

There are other ways in which one might ensure good coverage which we have not explored:

- One could check not just that each host LTS rule is covered, but that each of their disjunctive clauses is.
- For our earlier UDP specification [SSW01a, SSW01b] we proved receptiveness and semideterminacy theorems for the model, ensuring roughly that in any host state any socket call and datagram could be accepted by the model, and that in many non-error cases a unique host LTS rule is applicable. We expect that analogous results should hold here, giving a sense in which the model is a complete specification, and hence showing that checking coverage of all host LTS rules does imply coverage of a good range of real-world behaviour.
- One could take some application(s) of interest, link them to our instrumented `nssock` library instead of to the system sockets library, and run them on an instrumented network. We have done this for `wget`, `lynx`, and `mozilla`, but have not as yet attempted to work with the resulting traces.
- Given a suitably-instrumented TCP/IP implementation, one might check how well the tests exercise all code paths.

Moreover, given a specification which is fully validated with respect to a set of traces, it would be interesting to see how many new issues are picked up by validating against new tests (perhaps partially randomly-generated).

We have not attempted to automatically generate tests from the specification. For example, one might attempt to identify boundary cases in which particular rules 'just' apply. This seems to be a very hard problem, and even if it could be done for single rules, one would then have to work backwards to construct a trace that would build the required host state.

3.4 Trace visualisation

We have already seen a generated trace, in Figure 3. This example is misleadingly concise — most traces are around 100 steps long, and transition labels for TCP datagrams and TCP control block information are typically 20–30 lines long each. To aid the human reader in absorbing this information, we have two alternate presentations.

Firstly, we provide (via HTML) a colour-highlighted version of the trace, looking exactly as in Figure 3 except that call labels are red, return labels are green, control block information labels are lavender, send datagrams are yellow, receive datagrams are orange, and time passage labels are unhighlighted. This is a dramatic improvement in readability, while preserving the full information content of the trace.

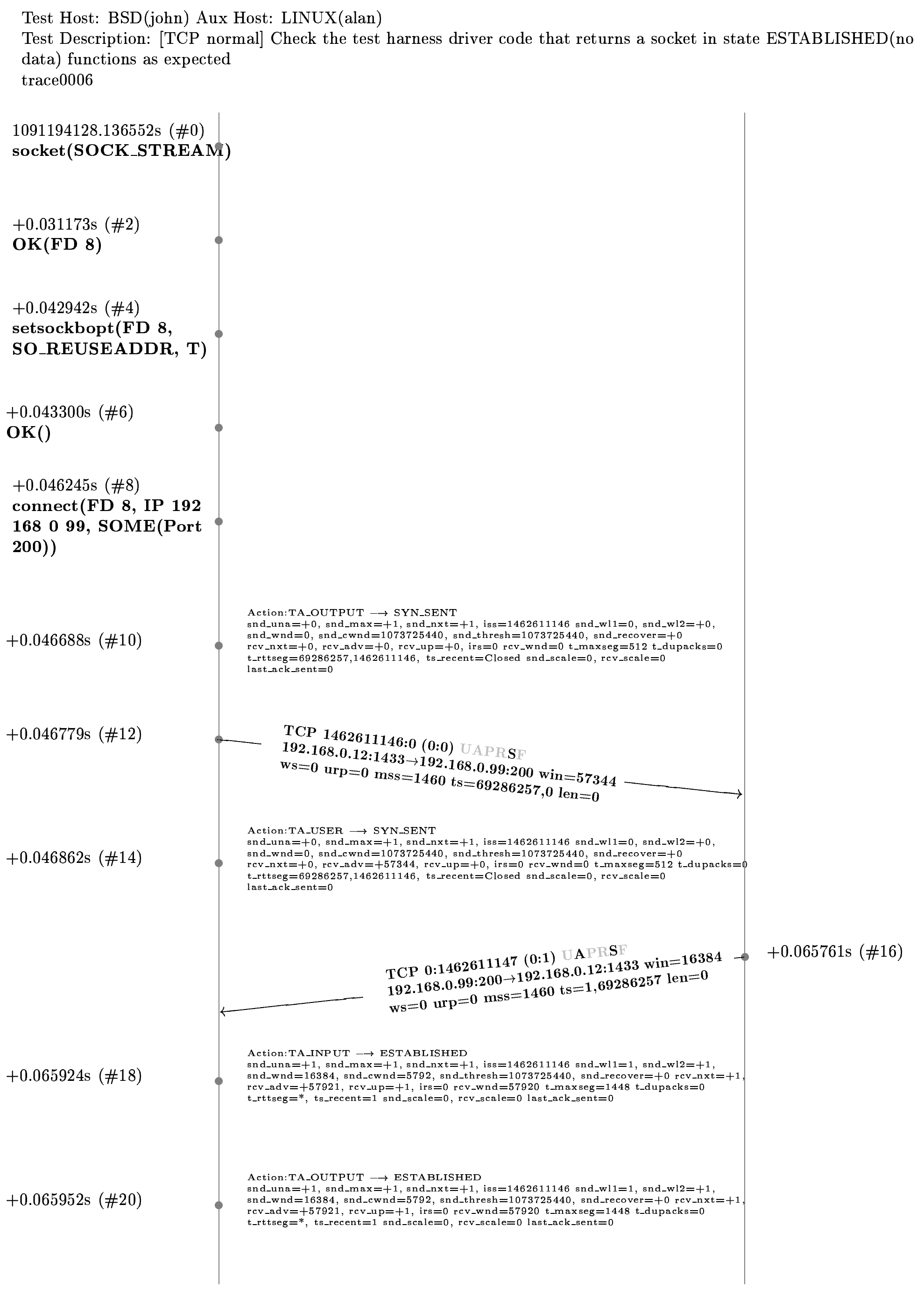
Secondly, we provide a PostScript rendering of the trace, showing most of the useful information in graphical form (Figure 5). This is by far the most-used representation, in most cases allowing the precise situation at each step to be recognised instantly. After an initial summary, time progresses downwards, with the test host on the left and the remote host or network on the right. Each event is labelled with its time (relative to the start time of the the trace) and step number; time passage is implicit. Host-local behaviour (calls and returns) appear as labelled points on the left; control block information appears as a point on the left with the key information in the middle; and sent and received datagrams appear as rightward and leftward arrows respectively, labelled with their key content. The `mkdiag` tool that produces this is an OCaml program that parses a trace (with the HOL label parser of the test tools) and generates LaTeX picture environment commands.

4 The Specification — Introduction

In this section we describe the high-level structure of the specification, as presented in Volume 2 [BFN$^+$05]. We introduce the language in which it is written —the higher-order logic supported by HOL— and give some of the key types used to model network messages, sockets interface events, and the internal states of hosts. This is far from a complete introduction, either to HOL, or to the specification, and assumes some familiarity with the design of the protocols. The aim is rather to show the level of detail and style used, and give the reader enough background that they can then turn to the specification itself.

The specification is a moderately large document, around 350 pages. 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 (150 pages) and the protocol rules (70 pages). This includes extensive comments, e.g. with summaries for each Socket call and differences between the model API and the three implementation APIs. It is defined in HOL script files and automatically typeset from them. The files are listed below together with their sizes in lines and bytes, including all commentary (as of 2005/1/30). Despite the `TCP1` prefix, these include all the specification, including the UDP and ICMP parts. The host LTS rules are by far the largest component.

```
 364   11543 TCP1_utilsScript.sml
 119    4104 TCP1_errorsScript.sml
  66    1619 TCP1_signalsScript.sml
 807   27482 TCP1_baseTypesScript.sml
 329   10008 TCP1_netTypesScript.sml
 270    8920 TCP1_LIBinterfaceScript.sml
 181    6847 TCP1_host0Script.sml
 108    4553 TCP1_ruleidsScript.sml
 388   13728 TCP1_timersScript.sml
 857   36009 TCP1_hostTypesScript.sml
 697   24634 TCP1_paramsScript.sml
2900  122541 TCP1_auxFnsScript.sml
```



Test Host: BSD(john) Aux Host: LINUX(alan)
Test Description: [TCP normal] Check the test harness driver code that returns a socket in state ESTABLISHED(no data) functions as expected
trace0006

1091194128.136552s (#0)
socket(SOCK_STREAM)

+0.031173s (#2)
OK(FD 8)

+0.042942s (#4)
setsockbopt(FD 8, SO_REUSEADDR, T)

+0.043300s (#6)
OK()

+0.046245s (#8)
connect(FD 8, IP 192 168 0 99, SOME(Port 200))

+0.046688s (#10)

```
Action:TA_OUTPUT --> SYN_SENT
snd_una=+0, snd_max=+1, snd_nxt=+1, iss=1462611146 snd_wl1=0, snd_wl2=+0,
snd_wnd=0, snd_cwnd=1073725440, snd_thresh=1073725440, snd_recover=+0
rcv_nxt=+0, rcv_adv=+0, rcv_up=+0, irs=0 rcv_wnd=0 t_maxseg=512 t_dupacks=0
t_rttseg=69286257,1462611146, ts_recent=Closed snd_scale=0, rcv_scale=0
last_ack_sent=0
```

+0.046779s (#12)

```
TCP 1462611146:0 (0:0) UAPRSF
192.168.0.12:1433→192.168.0.99:200 win=57344
ws=0 urp=0 mss=1460 ts=69286257,0 len=0
```

+0.046862s (#14)

```
Action:TA_USER --> SYN_SENT
snd_una=+0, snd_max=+1, snd_nxt=+1, iss=1462611146 snd_wl1=0, snd_wl2=+0,
snd_wnd=0, snd_cwnd=1073725440, snd_thresh=1073725440, snd_recover=+0
rcv_nxt=+0, rcv_adv=+57344, rcv_up=+0, irs=0 rcv_wnd=0 t_maxseg=512 t_dupacks=0
t_rttseg=69286257,1462611146, ts_recent=Closed snd_scale=0, rcv_scale=0
last_ack_sent=0
```

+0.065761s (#16)

```
TCP 0:1462611147 (0:1) UAPRSF win=16384
192.168.0.99:200→192.168.0.12:1433 len=0
ws=0 urp=0 mss=1460 ts=1,69286257
```

+0.065924s (#18)

```
Action:TA_INPUT --> ESTABLISHED
snd_una=+1, snd_max=+1, snd_nxt=+1, iss=1462611146 snd_wl1=1, snd_wl2=+1,
snd_wnd=16384, snd_cwnd=5792, snd_thresh=1073725440, snd_recover=+0 rcv_nxt=+1,
rcv_adv=+57921, rcv_up=+1, irs=0 rcv_wnd=57920 t_maxseg=1448 t_dupacks=0
t_rttseg=*, ts_recent=1 snd_scale=0, rcv_scale=0 last_ack_sent=0
```

+0.065952s (#20)

```
Action:TA_OUTPUT --> ESTABLISHED
snd_una=+1, snd_max=+1, snd_nxt=+1, iss=1462611146 snd_wl1=1, snd_wl2=+1,
snd_wnd=16384, snd_cwnd=5792, snd_thresh=1073725440, snd_recover=+0 rcv_nxt=+1,
rcv_adv=+57921, rcv_up=+1, irs=0 rcv_wnd=57920 t_maxseg=1448 t_dupacks=0
t_rttseg=*, ts_recent=1 snd_scale=0, rcv_scale=0 last_ack_sent=0
```

Figure 5: Trace visualisation — sample trace (second page not shown)

```
 18553  778153 TCP1_hostLTSScript.sml
   175    6030 TCP1_evalSupportScript.sml
 25814 1056171 total
```

The specification is largely presented in the logical order, as it is input to HOL. (There are a few exceptions: auxiliary definitions that are typeset close to where they are used rather than where they are defined). It begins with some basic utility definitions in `utils`, then various files define the types used in the model: `errors`, `signals`, and `baseTypes` are used throughout; `netTypes` specifies the structure of network events (TCP segments and UDP and ICMP datagrams); `LIBinterface` gives the sockets interface types; `host0` and `ruleids` define the type of host transition labels and the set of LTS rule names; and `timers` and `hostTypes` define the type of host states. The `params` file specifies host parameters (scheduling delay bounds, flag defaults, queue sizes, etc.). The `auxFns` file makes a number of substantial auxiliary definitions. The main definition of the host labelled transition system is in `hostLTS`, which is essentially a single inductive definition of the least transition relation satisfying the various axioms. Finally, `evalSupport` is used for the initial host states used in testing.

To read the specification one needs first some familiarity with the HOL language and with the key types used, as described below. Given that, one can browse Volume 2. It is perhaps best to begin 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. Each socket call has a substantial preamble giving an overview of its behaviour, a list of the possible errors, a description of some common cases of rule sequences, a discussion of the API used in the model and how it relates to the Posix, FreeBSD, Linux, and WinXP APIs, and a summary of the rules for that call. Having looked at the preamble one can then go directly to a particular rule covering some behaviour of interest; the rule descriptions do not depend strongly on one another (and hence there is some repetition).

The annotation text in the specification does not subsume the existing texts on networking and TCP. It discusses *what* happens in detail, but presupposes some background knowledge as to *why* the protcol behaviour is designed in this way. One should consult the RFCs and (e.g.) the Stevens texts [Ste94, WS95, Ste98] in conjunction with the specification, e.g. for introductions to flow control and the various congestion control algorithms used.

4.1 The HOL language

The HOL logic is a typed higher-order predicate calculus [GM93], derived from Church's Simple Theory of Types [Chu40]. There are two main differences from standard first order logic. First, each term has a type: types are used in order to build well-formed terms, and quantifiers can range over any type (including function types, hence the 'higher order'). Second, there is no distinction in HOL between terms and formulas: HOL has only terms, with terms of type bool playing the role of first-order logic formulas. It is a classical logic and has a set theoretic semantics, in which types denote non-empty sets and the function space denotes total functions. The HOL logic is built on the syntax of a lambda calculus with an ML-style polymorphic type system. The syntax is based on signatures for types (Ω) and terms (Σ_Ω). The type signature assigns arities to type constants, while the term signature delivers the types of constants.

4.1.1 Types

A HOL type can be a type variable α, a type constant c of arity 0, or a compound type, which is a type constant c of arity n applied to a list of n types.

$$\sigma \quad ::= \quad \alpha \quad | \quad c \quad | \quad (\sigma_1, \ldots, \sigma_n)c$$

The type signature Ω always contains type constants bool and ind (an infinite set of individuals) and a type constant of arity 2 for function space, written $\sigma_1 \to \sigma_2$. A large collection of types can be definitionally constructed in HOL, including algebraic data types and record types, using these initial types. An algebraic data type is introduced by a definition

$$\mathsf{t} = \textsc{Con}_1 \ \textbf{of} \ \sigma_1 \mid \ldots \mid \textsc{Con}_n \ \textbf{of} \ \sigma_n$$

providing a type t with constructors $\text{C}\textsc{on}_1 \quad \ldots \quad \text{C}\textsc{on}_n$. The values of t are of the form $\text{C}\textsc{on}_i(v_i)$ $(i = 1..n)$ where v_i has type σ_i. A record type is introduced by a definition

$$\mathsf{t} = \{\!|\, label_1 : \sigma_1;\ \ldots;\ label_n : \sigma_n \,|\!\}$$

for field labels $label_1 \quad \ldots \quad label_n$. The values of this type are records with the form

$$\{\!|\, label_1 = v_1;\ \ldots;\ label_n = v_n \,|\!\}$$

where each v_i has type σ_i.

The specification makes frequent use of option types σ option, with values NONE and SOME(v) for v of type σ. These values often represent a wildcard and a lifted value, respectively, so are written $*$ and $\uparrow v$. It also uses types $\sigma_1\#\sigma_2$, of pairs of values of types σ_1 and σ_2; types σ list, of finite lists of values of type σ; and types $\sigma_1 \mapsto \sigma_2$, of finite maps from type σ_1 to type σ_2.

4.1.2 Terms

A HOL term t is either a variable, a constant from the term signature Σ_Ω, a function, or a function application. The term grammar is roughly:

$$t ::= x \mid c \mid \lambda x.t \mid t_1\ t_2$$

Here $\lambda x.t$ is a function with formal argument x and body t, and $t_1\ t_2$ is t_1 applied to t_2. In fact all terms are associated with a unique type, so a more accurate grammar of terms is

$$t_\sigma \quad ::= \quad x_\sigma \quad \mid \quad c_\sigma \quad \mid \quad (\lambda x_{\sigma_1}.\ t_{\sigma_2})_{\sigma_1 \to \sigma_2} \quad \mid \quad (t_{\sigma' \to \sigma}\ t'_{\sigma'})_\sigma$$

but as the system uses ML-style type inference the user does not need to write type annotations. Logical operators, e.g. conjunction $t_1 \wedge t_2$, are simply constants of the appropriate type (here bool $\to$ bool $\to$ bool). The system has support for defining sugared mixfix syntax, both for parsing and pretty-printing, allowing applications of this constant to its two arguments to be written in the normal infix syntax.

Other standard logical constants are below:

Kind of term	HOL notation (typeset)	(ASCII)	Description
Truth	**T**	T	true
Falsity	**F**	F	false
Negation	$\neg t$	~t	not t
Disjunction	$t_1 \vee t_2$	t1\/t2	t_1 or t_2
Conjunction	$t_1 \wedge t_2$	t1/\t2	t_1 and t_2
Implication	$t_1 \implies t_2$	t1==>t2	t_1 implies t_2
Equality	$t_1 = t_2$	t1=t2	t_1 equals t_2
$\forall$-quantification	$\forall x.t$	!x.t	for all x . t
$\exists$-quantification	$\exists x.t$	?x.t	for some x. t
ε-term	$@x.t$	@x.t	an x such that t
Conditional	**if** t **then** t_1 **else** t_2	if t then t1 else t2	if t then t_1 else t_2

If t is a record then $t\ \{\!|\, label := e \,|\!\}$ is the record t with field *label* overridden to e. Record projection is written e.g. $t.label2$ for the *label2* field of t. The syntax $t \oplus (v_1 \mapsto v_2)$ denotes the finite map t overridden with v_1 mapped to v_2, and $t[v]$ is the image of v in the finite map t.

The record and finite map update operations are purely functional, simply returning an updated value: there is no notion of store or imperative side-effect here.

4.1.3 Proofs

The HOL system provides rich support for machine-assisted proof in this logic, in the LCF style [GMW79]. This ensures that the logical soundness of the system depends only on the correctness of a rather small kernel of HOL implementation code (and, of course, of the fragment of the underlying programming language implementation and hardware that is used).

Our automated validation tools make heavy use of this support, proving machine-checked theorems for each checked transition. To read the specification itself, however, one does not need to be familiar with HOL proof.

4.2 Network interface types

A host interacts with the network by exchanging *messages*, which can be either TCP segments, UDP datagrams, or ICMP datagrams. This embodies our abstraction from the details of IP, and in particular from IP fragmentation: we consider a message is exchanged in an implementation only when all the IP fragments that compose it have crossed the network interface. From `netTypes`:

– **IP message type :**

msg = TCP **of** tcpSegment | ICMP **of** icmpDatagram | UDP **of** udpDatagram

UDP datagrams contain just the source and destination addresses and some data, which is just a list of bytes:

– **UDP datagram type :**

udpDatagram
=⦃ is_1 : ip option; (* source IP *)
   is_2 : ip option; (* destination IP *)
   ps_1 : port option; (* source port *)
   ps_2 : port option; (* destination port *)
   data : *byte* list
⦄

Here IP addresses and ports are of the types below (from `baseTypes`).

– **:**

port = Port **of** num (* really 16 bits, non-zero *)

Description TCP or UDP port number, non-zero.

– **:**

ip = ip **of** num (* really 32 bits, non-zero *)

Description IPv4 address, non-zero.

The specification distinguishes between the types port and ip, for which we do not use the zero values, and option types port option and ip option, with values $*$ (modelling the zero values, often used as wildcards in the API) and $\uparrow p$ and $\uparrow i$, modelling the non-zero values. Zero values are used as wildcards in some places and are forbidden in others; this typing lets that be captured explicitly.

The HOL type num is of arbitrary-size integers, whereas implementations use 16-bit values for ports and 32-bit values for IPv4 addresses. It is an invariant of the model that no larger values occur, but this is not captured by the HOL type system. Other invariants include, for example, the facts that zero values are not used and that the *data* in a UDP datagram is not too long.

TCP segments contain source and destination addresses as before, sequence- and acknowledgement numbers, boolean flags *ACK*, *SYN* etc., a window size, and urgent pointer, various options, and again some data:

– **TCP datagram type :**

tcpSegment
$=\llparenthesis$ is_1 : ip option; (* source IP *)
is_2 : ip option; (* destination IP *)
ps_1 : port option; (* source port *)
ps_2 : port option; (* destination port *)
seq : tcp_seq_local; (* sequence number *)
ack : tcp_seq_foreign; (* acknowledgment number *)
URG : bool;
ACK : bool;
PSH : bool;
RST : bool;
SYN : bool;
FIN : bool;
win : $word16$; (* window size (unsigned) *)
ws : $byte$ option; (* TCP option: window scaling; typically 0..14 *)
urp : $word16$; (* urgent pointer (unsigned) *)
mss : $word16$ option; (* TCP option: maximum segment size (unsigned) *)
ts : (ts_seq # ts_seq) option; (* TCP option: RFC1323 timestamp value and echo-reply *)
$data$: $byte$ list
$\rrparenthesis$

Description The use of ”local” and ”foreign” here is with respect to the *sending* TCP.

ICMP datagrams include source and destination IP addresses and information from the message that caused their generation:

– **ICMP datagram type :**
icmpDatagram
$=\llparenthesis$ is_1 : ip option; (* this is the sender of this ICMP *)
is_2 : ip option; (* this is the intended receiver of this ICMP *)

(* we assume the enclosed IP always has at least 8 bytes of data, i.e., enough for all the fields below *)
is_3 : ip option; (* source of enclosed IP datagram *)
is_4 : ip option; (* destination of enclosed IP datagram *)
ps_3 : port option; (* source port *)
ps_4 : port option; (* destination port *)
$proto$: protocol; (* protocol *)
seq : tcp_seq_local option; (* seq *)
t : icmpType
$\rrparenthesis$

4.3 Sockets interface types

The model sockets interface exists in two closely-related forms. The primary definition is that in the HOL, in `LIBinterface`, which defines a type LIB_interface with a constructor for each call, accept, bind, etc. They take the arguments of the corresponding call, so for example accept(fd) (for a file descriptor fd) is a value of type LIB_interface. The file also defines the return type of each call. Return values are of a HOL type TLang which is a labelled union of all the language types.

The specification is not tied to any particular programming language, and for any reasonable language one could give a language binding with a clear relationship to the specified API. For languages with a rich enough type structure the two can be made almost identical. We have done so for a version of the interface as a library for the OCaml programming language, from which it can be invoked directly by programs. It differs from the HOL interface only in trivial ways, e.g. in using OCaml 31-bit `int`s rather than HOL nums as the listen-queue-length argument to listen(). This OCaml library is similar but not identical to that included as part of the Unix module with the OCaml distribution.

Here we give the OCaml version of the interface, as it is more concise and easier to read than the HOL (in which argument and return types of the calls are separated). It uses the subgrammar of OCaml types:

t	::=	*tc*	defined type constructor name
		unit	type of the unit value ()
		bool	booleans
		int	integers
		string	strings
		t1 * t2	pairs (v1,v2) of values of types t1 and t2
		t1 -> t2	functions from t1 to t2
		t option	either None or Some v for a value v of type t
		t list	lists of values of type t

The type of error codes consists roughly all the possible Unix errors. Not all error codes are used in the body of the specification; those that are are described in the 'Errors' section of each socket call.

```
type error =     (* The type of error codes *)
    E2BIG
  | EACCES
  | EADDRINUSE
  | ...
```

The type of signals includes all the signals known to POSIX, Linux, and BSD. The specification does not model signal behaviour in detail (it treats them very nondeterministically), but they occur as an argument to pselect() so must be defined here.

```
type signal =    (* The type of signals *)
    SIGABRT
  | SIGALRM
  | ...
```

File descriptors, IP addresses, ports, etc. are abstract types in the OCaml interface, preventing accidental misuse. There are various coercions (which we do not give here) to construct values of these types. For interface identifiers (ifid) the specification supposes the existence of a loopback identifier and numbered ethernet identifiers. Any particular host may or may not have an interface with each identifier, of course.

```
type fd       (* The abstract type of file descriptors. In HOL:  FD of num       *)
type ip       (* The abstract type of IP addresses.     In HOL:  IP of num       *)
type port     (* The abstract type of inet ports.       In HOL:  Port of num     *)
type netmask  (* The abstract type of netmasks.         In HOL:  NETMASK of num  *)
type ifid     (* The abstract type of ifids.            IN HOL:  LO | ETH of num *)
```

The sockets interface involves various flags, for files, sockets, and messages. Both the HOL and OCaml interfaces define them as new types, preventing misuse.

```
type filebflag =    (* The type of boolean-valued file flags *)
    O_NONBLOCK
  | O_ASYNC

type sockbflag =    (* The type of boolean-valued socket flags *)
    SO_BSDCOMPAT
  | SO_REUSEADDR
  | SO_KEEPALIVE
  | SO_OOBINLINE
  | SO_DONTROUTE

type socknflag =    (* The type of numeric-valued socket flags *)
    SO_SNDBUF
  | SO_RCVBUF
  | SO_SNDLOWAT
  | SO_RCVLOWAT

type socktflag =    (* The type of time-valued socket flags *)
    SO_LINGER
  | SO_SNDTIMEO
  | SO_RCVTIMEO

type msgbflag =     (* The type of boolean-valued message flags *)
    MSG_PEEK
  | MSG_OOB
  | MSG_WAITALL
  | MSG_DONTWAIT

type sock_type =    (* The type of socket types *)
    SOCK_DGRAM
  | SOCK_STREAM
```

The OCaml interface indicates error returns to socket calls by raising the exception below.

```
exception Unix_error of error * string * string
```

Finally, the types of the socket calls themselves are as follows. For the behaviour of these, including the meanings of the arguments and results, we refer the reader to the socket call preambles of Volume 2.

```
val accept: fd -> fd * (ip * port)
val bind: fd -> ip option -> port option -> unit
val close: fd -> unit
val connect: fd -> ip -> port option -> unit
val disconnect: fd -> unit
val dup: fd -> fd
val dupfd: fd -> int -> fd
val getfileflags: fd -> filebflag list
val setfileflags: fd -> filebflag list -> unit
val getifaddrs: unit -> (ifid * ip * ip list * netmask) list
val getsockname: fd -> ip option * port option
val getpeername: fd -> ip * port
val getsockbopt: fd -> sockbflag -> bool
val getsocknopt: fd -> socknflag -> int
val getsocktopt: fd -> socktflag -> (int * int) option
val getsockerr: fd -> unit
val getsocklistening: fd -> bool
val listen: fd -> int -> unit
val pselect: fd list -> fd list -> fd list -> (int * int) option -> signal list option
             -> fd list * (fd list * fd list)
val recv: fd -> int -> msgbflag list -> (string*((ip option*port option)*bool) option)
val send: fd -> (ip * port) option -> string -> msgbflag list -> string
val setsockbopt: fd -> sockbflag -> bool -> unit
val setsocknopt: fd -> socknflag -> int -> unit
val setsocktopt: fd -> socktflag -> (int * int) option -> unit
val shutdown: fd -> bool -> bool -> unit
val sockatmark: fd -> bool
val socket: sock_type -> fd
```

4.4 Host transition types

Given the types of network interface and socket interface events, we can define the type Lhost0 of host transition labels. These transitions are all that is externally visible of a host, to either the network or to application programs above the sockets API.

– **Host transition labels :**

Lhost0 =

(* library interface *)

LH_CALL **of** tid#LIB_interface (* invocation of LIB call, written e.g. tid·(socket(socktype)) *)

| LH_RETURN **of** tid#TLang (* return result of LIB call, written $\overline{\mathsf{tid}\cdot v}$ *)

(* message transmission and receipt *)

| LH_SENDDATAGRAM **of** msg (* output of message to the network, written $\overline{\mathsf{msg}}$ *)

| LH_RECVDATAGRAM **of** msg (* input of message from the network, written msg *)

| LH_LOOPDATAGRAM **of** msg (* loopback output/input, written $\overleftrightarrow{\mathsf{msg}}$ *)

(* connectivity changes *)

| LH_INTERFACE **of** ifid#bool (* set interface status to boolean *up*, written LH_INTERFACE(*ifid*, *up*) *)

(* miscellaneous *)

| τ (* internal transition, written τ *)

| LH_EPSILON **of** duration (* time passage, written *dur* *)

| LH_TRACE **of** *tracerecord* (* TCP trace record, written LH_TRACE *tr* *)

4.5 Host internal state types

Host states are values of the record type below, containing the sockets *socks* (a finite map from socket ids to socket structures), the host's input and output message queues *iq* and *oq*, and so forth.

– **host details :**

host =⟨[

arch : *arch*; (* architecture *)

privs : bool; (* whether process has root/CAP_NET_ADMIN privilege *)

ifds : *ifid* ↦ ifd; (* interfaces *)

rttab : *routing_table*; (* routing table *)

ts : tid ↦ hostThreadState timed; (* host view of each thread state *)

files : *fid* ↦ file; (* files *)

socks : sid ↦ socket; (* sockets *)

listen : sid list; (* list of listening sockets *)

bound : sid list; (* list of sockets bound: head of list was first to be bound *)

iq : msg list timed; (* input queue *)

oq : msg list timed; (* output queue *)

bndlm : *bandlim_state*; (* bandlimiting *)

ticks : ticker; (* ticker *)

fds : fd ↦ *fid*(* file descriptors (per-process) *)

]⟩

Description The input and output queue timers model the interrupt scheduling delay; the first element (if any) must be processed by the timer expiry.

Sockets are records with local and remote IP addresses and ports, various flags, and some protocol-specific data for either TCP or UDP:

– **details of a socket :**

socket

=⟨[*fid* : *fid* option; (* associated open file description if any *)

sf : sockflags; (* socket flags *)

is_1 : ip option; (* local IP address if any *)

ps_1 : port option; (* local port if any *)

```
    is2 : ip option; (* remote IP address if any *)
    ps2 : port option; (* remote port if any *)
    es : error option; (* pending error if any *)
    cantsndmore : bool; (* output stream ends at end of send queue *)
    cantrcvmore : bool; (* input stream ends at end of receive queue *)
    pr : protocol_info (* protocol-specific information *)
  ⟧
```

– protocol-specific socket data :

```
protocol_info = TCP_PROTO of tcp_socket
              | UDP_PROTO of udp_socket
```

For UDP the protocol-specific data is just a receive queue:

– details of a UDP socket :

```
udp_socket
 =⟦ rcvq : dgram list⟧
```

Description UDP sockets are very simple – the protocol-specific content is merely a receive queue. The receive queue of a UDP socket, however, is not just a queue of bytes as it is for a TCP socket. Instead, it is a queue of *messages* and (in some implementations) *errors*. Each message contains a block of types and some ancilliary data.

Variations

WinXP	On WinXP, errors are returned in order w.r.t. messages; this is modelled by placing them in the receive queue.
FreeBSD,Linux	On FreeBSD and Linux, only messages are placed in the receive queue, and errors are treated asynchronously.

For TCP, however, the protocol-specific data is extensive. It is partitioned into the tcp_socket record structure below, containing the 'TCP state' *st*, send and receive buffers, etc., which is all that most socket rules need to refer to, and the embedded tcpcb TCP protocol control block given after, which contains many fields used by the protocol. The 'TCP state' *st* is obviously only a tiny part of the actual protocol endpoint state.

– details of a TCP socket :

```
tcp_socket
 =⟦ st : tcpstate; (* here rather than in tcpcb for convenience as heavily used. Called t_state in BSD *)
    cb : tcpcb;
    lis : socket_listen option; (* invariant: ∗ iff not LISTEN *)
    sndq : byte list;
    sndurp : num option;
    rcvq : byte list;
    rcvurp : num option; (* was ”oobmark” *)
    iobc : iobc
```

```
⟧
```

The TCP control block structure broadly follows that of BSD, which in turn broadly follows the original RFC, but there are many differences.

– the TCP control block :

```
tcpcb =⟦

    (* timers *)
    tt_rexmt : (rexmtmode#num)timed option; (* retransmit timer, with mode and shift; ∗ is idle *)
    (* see tcp_output.c:356ff for more info. *)
    (* as in BSD, the shift starts at zero, and is incremented each time the timer fires. So it is zero during
    the first interval, 1 after the first retransmit, etc. *)
    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, overlays keep in BSD *)
    tt_fin_wait_2 : () timed option; (* FIN_WAIT_2 timer, overlays 2msl in BSD *)
    t_idletime : stopwatch; (* time since last segment received *)

    (* flags, some corresponding to BSD TF_ flags *)
    tf_needfin : bool; (* send FIN (implicit state, used for app close while in SYN_RECEIVED) *)
    tf_shouldacknow : bool; (* output a segment urgently – similar to TF_ACKNOW, but used less often*)
    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 *)
    snd_wl1 : tcp_seq_foreign; (* seq number of most recent window update segment *)
    snd_wl2 : tcp_seq_local; (* ack number of most recent window update segment *)
    iss : tcp_seq_local; (* initial send sequence number *)
    snd_wnd : num; (* send window size: always between 0 and 65535*2**14 *)
    snd_cwnd : num; (* congestion window *)
    snd_ssthresh : num; (* threshold between exponential and linear snd_cwnd expansion (for slow
                           start)*)

    (* receive variables *)
    rcv_wnd : num; (* receive window size *)
    tf_rxwin0sent : bool; (* have advertised a zero window to receiver *)
    rcv_nxt : tcp_seq_foreign; (* lowest sequence number not yet received *)
    rcv_up : tcp_seq_foreign; (* received urgent pointer if any, else = rcv_nxt *)
    irs : tcp_seq_foreign; (* initial receive sequence number *)
    rcv_adv : tcp_seq_foreign; (* most recently advertised window *)
    last_ack_sent : tcp_seq_foreign; (* last acknowledged sequence number *)

    (* connection parameters *)
    t_maxseg : num; (* maximum segment size on this connection *)
    t_advmss : num option; (* the mss advertisment sent in our initial SYN *)
    tf_doing_ws : bool; (* doing window scaling on this connection? (result of negotiation) *)
    request_r_scale : num option; (* pending window scaling, if any (used during negotiation) *)
    snd_scale : num; (* window scaling for send window (0..14), applied to received advertisements
                        (RFC1323) *)
    rcv_scale : num; (* window scaling for receive window (0..14), applied when we send advertisements
                        (RFC1323) *)

    (* timestamping *)
```

tf_doing_tstmp : bool; (* are we doing timestamps on this connection? (result of negotiation) *)
tf_req_tstmp : bool; (* have/will request(ed) timestamps (used during negotiation) *)
ts_recent : ts_seq timewindow; (* most recent timestamp received; TimeWindowClosed if invalid. Timer models the RFC1323 end-§4.2.3 24-day validity period. *)

(* round-trip time estimation *)
t_rttseg : (ts_seq # tcp_seq_local) option; (* start time and sequence number of segment being timed *)
t_rttinf : rttinf; (* round-trip time estimator values *)

(* retransmission *)
t_dupacks : num; (* number of consecutive duplicate acks received (typically 0..3ish; should this wrap at 64K/4G ack burst?) *)
t_badrxtwin : () timewindow; (* deadline for bad-retransmit recovery *)
snd_cwnd_prev : num; (* *snd_cwnd* prior to retransmit (used in bad-retransmit recovery) *)
snd_ssthresh_prev : num; (* *snd_ssthresh* prior to retransmit (used in bad-retransmit recovery) *)
snd_recover : tcp_seq_local; (* highest sequence number sent at time of receipt of partial ack (used in RFC2581/RFC2582 fast recovery) *)

(* other *)
t_segq : tcpReassSegment list; (* segment reassembly queue *)
t_softerror : error option (* current transient error; reported only if failure becomes permanent *)
(* could cut this down to the actually-possible errors? *)
$|\!\rangle$

4.6 Sample transition rule – *bind_5*

The transition system is defined by a set of rules of the form

$$\vdash P(h_0, l, h) \Rightarrow h_0 \xrightarrow{l} h$$

where P is a condition under which host state h_0 can make a transition labelled l to host state h. Each rule has a name, e.g. *bind_5*, *deliver_in_1* etc., a protocol, either RP_TCP, RP_UDP, or RP_ALL, and a category, e.g. FAST SUCCEED for a sockets API rule that cannot block and returns a value rather than an error. Rules are typeset in the form below, with the 'sidecondition' P below the transition.

rule_schema_1 **protocol: category** **short description**

$h_0 \quad \xrightarrow{l} \quad h$

$P(h_0, l, h)$

Description Informal text describing the main points of the rule.

Variations Informal text highlighting how the different architectures behave differently in the rule.

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

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

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

$$\xrightarrow{\overline{tid \cdot \mathsf{bind}(fd, is_1, ps_1)}} \quad h \;\langle\!|\, ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL EINVAL}))_{\text{sched\_timer}}) \,|\!\rangle$$

$fd \in \mathbf{dom}(h.fds) \wedge$
$fid = h.fds[fd] \wedge$
$h.files[fid] = \textsc{File}(\text{FT\_}\textsc{Socket}(sid), ff) \wedge$
$h.socks[sid] = sock \wedge$
$(sock.ps_1 \neq * \vee$
$(\text{bsd\_arch}\;\; h.arch \wedge sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge$
$\quad (sock.cantsndmore \vee$
$\quad tcp\_sock.cb.bsd\_cantconnect)))$

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 $\overline{tid \cdot \mathsf{bind}(fd, is_1, ps_1)}$ 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 6 rules for $\mathsf{bind}()$. 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 $(\textsc{Run})_d$, 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 $(\textsc{Ret}(\text{FAIL EINVAL}))_{\text{sched\_timer}}$, 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 $\textsc{File}(\text{FT\_}\textsc{Socket}(\mathsf{sid}), ff)$, i.e. to a socket identifier sid and file flags *ff*. The fourth says that that socket identifier sid is mapped to a socket structure *sock*. These first four conditions appear in many socket-call rules. The fifth condition states that either 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 that this is a BSD host with a TCP socket that either has its *cantsndmore* flag set or the *bsd_cantconnect* flag in the TCP control block of the protocol-specific information in the socket.

Note that field names (e.g. *tid* 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).

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

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

4.7 Sample transition rule – network

The rules for sending and receiving messages, with transitions labelled $\overline{msg}$, msg, $\overleftrightarrow{msg}$, and τ, are rather simple. They transfer a message between a host's output or input queues and the network, or (in the loopback case) from queue to queue. The interesting protocol behaviour is rather in the rules that process the message at the head of the host's input queue and add messages to the end of it's output queue.

An example network rule is below.

deliver_in_99 **all: network nonurgent** **Really receive things**

$h \;\langle\!\langle iq := iq \rangle\!\rangle \;\xrightarrow{msg}\; h \;\langle\!\langle iq := iq' \rangle\!\rangle$

$\mathrm{sane\_msg}\ msg\ \wedge$
$\uparrow i_1 = msg.is_2\ \wedge$
$i_1\ \in\ \mathrm{local\_ips}(h.ifds)\ \wedge$
$\mathrm{enqueue\_iq}(iq, msg, iq', queued)$

Description Actually receive a message from the wire into the input queue. Note that if it cannot be queued (because the queue is full), it is silently dropped.

We only accept messages that are for this host. We also assert that any message we receive is well-formed (this excludes elements of type *msg* that have no physical realisation).

Note the delay in in-queuing the datagram is not modelled here.

4.8 Sample transition rule – *deliver_in_1*

Fig. 6 shows an example protocol rule, *deliver_in_1*, eliding some details. This rule models the behaviour of the system on receiving a SYN addressed to a listening socket. It is of intermediate complexity: many rules are rather simpler than this, a few are more substantial.

The transition $h \;\langle\!\langle ... \rangle\!\rangle \xrightarrow{\tau} h \;\langle\!\langle ... \rangle\!\rangle$ appears at the top; the input and output queue 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'*.

Recall that 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, but we may construct a new record by copying an existing one and providing new values for specific fields: $cb' = cb\ \langle\!\langle irs := seq \rangle\!\rangle$ 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.

The bulk of the rule is the condition (simply a HOL predicate) guarding the transition, specifying when the rule applies and what relationship holds between the input and output states. Notice first that the rule applies only when dequeueing the topmost message on the input queue results in a TCP segment TCP*seg* of a particular form: a SYN segment (ACK is false, RST is false, SYN is true, and the others are arbitrary).

After some validity checks, the host computes values required to generate the response segment and to update the host state. For instance, the host nondeterminstically may or may not wish to do timestamping (here the nondeterminism models the unknown setting of a configuration parameter). This choice is combined with whether the incoming segment contained a timestamping request in order to decide whether timestamping will be performed. Several other local values are specified nondeterministically in this manner: 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 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 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: due to the listen bug() discussed below (§9), in BSD implementations 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 URG or FIN may be set on an initial SYN, but this is ignored by all implementations we consider.

```
$\mathit{deliver\_in\_1}$   tcp: network nonurgent
                 Passive open: receive SYN, send SYN,ACK

    $h\ \langle\!| \mathit{socks} := \mathit{socks} \oplus [(\mathit{sid}, \mathit{sock})];$ (* listening socket *)
      $\mathit{iq} := \mathit{iq};$ (* input queue *)
      $\mathit{oq} := \mathit{oq} |\!\rangle$ (* output queue *)
$\xrightarrow{\tau}$
    $h\ \langle\!| \mathit{socks} := \mathit{socks} \oplus$
        (* listening socket *)
       $[(\mathit{sid}, \textsc{Sock}(\uparrow \mathit{fid}, \mathit{sf}, \mathit{is}_1, \uparrow p_1, \mathit{is}_2, \mathit{ps}_2, \mathit{es}, \mathit{csm}, \mathit{crm},$
         $\text{TCP\_Sock}(\text{LISTEN}, \mathit{cb}, \uparrow \mathit{lis}', [\,], *, [\,], *, \text{NO\_OOB})));$
        (* new connecting socket *)
       $(\mathit{sid}', \textsc{Sock}(*, \mathit{sf}', \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathit{csm}, \mathit{crm},$
         $\text{TCP\_Sock}(\text{SYN\_RCVD}, \mathit{cb}'', *, [\,], *, [\,], *, \text{NO\_OOB})))];$
      $\mathit{iq} := \mathit{iq}';$
      $\mathit{oq} := \mathit{oq}' |\!\rangle$
    ____________________________________________

(* check first segment matches desired pattern; unpack fields *)
$\text{dequeue\_iq}(\mathit{iq}, \mathit{iq}', \uparrow(\text{TCP } \mathit{seg})) \wedge$
$(\exists \mathit{win\_}\ \mathit{ws\_}\ \mathit{mss\_}\ \mathit{PSH}\ \mathit{URG}\ \mathit{FIN}\ \mathit{urp}\ \mathit{data}\ \mathit{ack}.$
  $\mathit{seg} =$
    $\langle\!| \mathit{is}_1 := \uparrow i_2; \mathit{is}_2 := \uparrow i_1; \mathit{ps}_1 := \uparrow p_2; \mathit{ps}_2 := \uparrow p_1;$
      $\mathit{seq} := \mathit{tcp\_seq\_flip\_sense}(\mathit{seq} : \text{tcp\_seq\_foreign});$
      $\mathit{ack} := \mathit{tcp\_seq\_flip\_sense}(\mathit{ack} : \text{tcp\_seq\_local});$
      $\mathit{URG} := \mathit{URG}; \mathit{ACK} := \mathbf{F}; \mathit{PSH} := \mathit{PSH};$
      $\mathit{RST} := \mathbf{F}; \mathit{SYN} := \mathbf{T}; \mathit{FIN} := \mathit{FIN};$
      $\mathit{win} := \mathit{win\_}; \mathit{ws} := \mathit{ws\_}; \mathit{urp} := \mathit{urp}; \mathit{mss} := \mathit{mss\_}; \mathit{ts} := \mathit{ts};$
      $\mathit{data} := \mathit{data}$
    $|\!\rangle \wedge$
$\mathbf{w2n}\ \mathit{win\_} = \mathit{win} \wedge$ (* type-cast from word to integer *)
$\mathbf{option\_map}\ \mathbf{ord}\ \mathit{ws\_} = \mathit{ws} \wedge$
$\mathbf{option\_map}\ \mathbf{w2n}\ \mathit{mss\_} = \mathit{mss}) \wedge$

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

(* sockets distinct; segment matches this socket; unpack fields of socket *)
$\mathit{sid} \notin (\mathbf{dom}(\mathit{socks})) \wedge \mathit{sid}' \notin (\mathbf{dom}(\mathit{socks})) \wedge \mathit{sid} \neq \mathit{sid}' \wedge$
$\text{tcp\_socket\_best\_match}\ \mathit{socks}(\mathit{sid}, \mathit{sock})\mathit{seg}\ h.\mathit{arch} \wedge$
$\mathit{sock} = \textsc{Sock}(\uparrow \mathit{fid}, \mathit{sf}, \mathit{is}_1, \uparrow p_1, \mathit{is}_2, \mathit{ps}_2, \mathit{es}, \mathit{csm}, \mathit{crm},$
            $\text{TCP\_Sock}(\text{LISTEN}, \mathit{cb}, \uparrow \mathit{lis}, [\,], *, [\,], *, \text{NO\_OOB})) \wedge$

(* socket is correctly specified (note BSD listen bug) *)
$((\mathit{is}_2 = * \wedge \mathit{ps}_2 = *) \vee$
$(\text{bsd\_arch}\ h.\mathit{arch} \wedge \mathit{is}_2 = \uparrow i_2 \wedge \mathit{ps}_2 = \uparrow p_2)) \wedge$
$(\mathbf{case}\ \mathit{is}_1\ \mathbf{of}\ \uparrow \mathit{i1}' \to \mathit{i1}' = i_1 \parallel * \to \mathbf{T}) \wedge$
$\neg(i_1 = i_2 \wedge p_1 = p_2) \wedge$

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

(* place new socket on listen queue *)
$\text{accept\_incoming\_q0}\ \mathit{lis}\ \mathbf{T} \wedge$
(* (elided: if drop_from_q0, drop a random socket yielding q0') *)
$\mathit{lis}' = \mathit{lis}\ \langle\!| q_0 := \mathit{sid}' :: q_0' |\!\rangle \wedge$

(* choose MSS and whether to advertise it or not *)
$\mathit{advmss} \in \{n \mid n \geq 1 \wedge n \leq (65535 - 40)\} \wedge$
$\mathit{advmss}' \in \{*; \uparrow \mathit{advmss}\} \wedge$

(* choose whether this host wants timestamping; negotiate with other side *)
$\mathit{tf\_rcvd\_tstmp}' = \mathbf{is\_some}\ \mathit{ts} \wedge$
$(\mathbf{choose}\ \mathit{want\_tstmp} :: \{\mathbf{F}; \mathbf{T}\}.$
$\mathit{tf\_doing\_tstmp}' = (\mathit{tf\_rcvd\_tstmp}' \wedge \mathit{want\_tstmp})) \wedge$

(* calculate buffer size and related parameters *)
$(\mathit{rcvbufsize}', \mathit{sndbufsize}', \mathit{t\_maxseg}', \mathit{snd\_cwnd}') =$
$\text{calculate\_buf\_sizes}\ \mathit{advmss}\ \mathit{mss} * (\text{is\_localnet}\ h.\mathit{ifds}\ i_2)$
  $(\mathit{sf}.n(\text{SO\_RCVBUF}))(\mathit{sf}.n(\text{SO\_SNDBUF}))$
  $\mathit{tf\_doing\_tstmp}'\ h.\mathit{arch} \wedge$
$\mathit{sf}' = \mathit{sf}\ \langle\!| n := \text{funupd\_list}\ \mathit{sf}.n[(\text{SO\_RCVBUF}, \mathit{rcvbufsize}');$
                                        $(\text{SO\_SNDBUF}, \mathit{sndbufsize}')] |\!\rangle \wedge$

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

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

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

(* choose initial sequence number *)
$\mathit{iss} \in \{n \mid \mathbf{T}\} \wedge$

(* acknowledge the incoming SYN *)
$\mathbf{let}\ \mathit{ack}' = \mathit{seq} + 1\ \mathbf{in}$

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

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

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

Figure 6: A sample TCP transition rule.

4.9 The protocol rules and *deliver_in_3*

The most complex rule is *deliver_in_3*, which models the processing of an incoming 'normal' TCP segment for an established TCP connection. The rule applies when a non-RST TCP segment can be dequeued from the host's input queue, with address quad matching a SYN_RECEIVED, ESTABLISHED, or later socket, with a valid acknowledgement number if the state is SYN_RECEIVED, with *SYN* only if the state is SYN_RECEIVED, and (in certain states) if a thread is still associated with the socket. (Other rules apply in the remaining cases.) The rule computes the appropriate changes to the host state, including enqueuing any segment or segments that should be constructed in response and (in SYN_RECEIVED) moving the connection from the incomplete to the completed connections queue.

The bulk of *deliver_in_3*'s behaviour is contained in four auxiliary definitions *topstuff*, *ackstuff*, *datastuff*, and *ststuff*, dealing respectively with initial checks, incoming-ACK processing (retransmission, recovery, dropping acked data from send queue, and so on), incoming data (reassembly, urgent data, immediate ACK of out-of-order segments, and so on), and TCP state transitions. The reader is referred to the specification for the definitions, which are heavily commented there and too long to include here. These definitions are built and threaded together using the relational monad (c.f. §2.7), which makes it easy to express incremental state modification, early exit, and optional/incremental segment generation.

The use of the relational monad can be seen in the rule below. A monadic action (such as *topstuff*) takes an initial socket and band-limiter state and relates it to a final socket and band-limiter state, a list of segments, and a boolean denoting whether execution should continue or stop immediately. The combinator andThen combines two actions into one, sequencing the second after the first unless the first indicates that execution should stop. Other combinators used in the definitions permit nondeterministic binding of variables, access to and modification of socket and band-limiter state, generation of segments, and so on (see Chapter 13 of the specification). Actions are relations, not functions, thus permitting nondeterminism to be modelled.

deliver_in_3 **tcp: network nonurgent** **Receive data, FINs, and ACKs in a connected state**

$$\begin{array}{lcl} h\ \langle\!|\mathit{socks} := \mathit{socks} \oplus [(\mathit{sid}, \mathit{sock})]; & \xrightarrow{\tau} & h\ \langle\!|\mathit{socks} := \mathit{socks}'; \\ \mathit{iq} := \mathit{iq}; & & \mathit{iq} := \mathit{iq}'; \\ \mathit{oq} := \mathit{oq}; & & \mathit{oq} := \mathit{oq}'; \\ \mathit{bndlm} := \mathit{bndlm}|\!\rangle & & \mathit{bndlm} := \mathit{bndlm}'|\!\rangle \end{array}$$

$\mathit{sid} \notin (\mathbf{dom}(\mathit{socks})) \wedge$
$\mathit{sock.pr} = \mathrm{TCP\_PROTO}(\mathit{tcp\_sock}) \wedge$

(* Assert that the socket meets some sanity properties. This is logically superfluous but aids semi-automatic model checking. See sane_socket (p**??**) for further details. *)
sane_socket *sock* $\wedge$

(* Take TCP segment *seg* from the head of the host's input queue *)
dequeue_iq($\mathit{iq}, \mathit{iq}', \uparrow(\mathrm{TCP}\ \mathit{seg})$) $\wedge$

(* The segment must be of an acceptable form *)
(* Note: some segment fields (namely TCP options *ws* and *mss*), are only used during connection establishment and any values assigned to them in segments during a connection are simply ignored. They are equal to the identifiers *ws_discard* and *mss_discard* respectively, which are otherwise unconstrained. *)
($\exists \mathit{win\_}\ \mathit{urp\_}\ \mathit{ws\_discard}\ \mathit{mss\_discard}.$
$\mathit{seg} = \langle\!|$
  $\mathit{is}_1 := \uparrow i_2;$
  $\mathit{is}_2 := \uparrow i_1;$
  $\mathit{ps}_1 := \uparrow p_2;$
  $\mathit{ps}_2 := \uparrow p_1;$
  $\mathit{seq} := \mathit{tcp\_seq\_flip\_sense}(\mathit{seq} : \mathrm{tcp\_seq\_foreign});$
  $\mathit{ack} := \mathit{tcp\_seq\_flip\_sense}(\mathit{ack} : \mathrm{tcp\_seq\_local});$
  $\mathit{URG} := \mathit{URG};$ (* Urgent/OOB data is processed by this rule *)
  $\mathit{ACK} := \mathit{ACK};$ (* Acknowledgements are processed *)
  $\mathit{PSH} := \mathit{PSH};$ (* Push flag maybe set on an incoming data segment *)
  $\mathit{RST} := \mathbf{F};$ (* *RST* segments are not handled by this rule *)
  $\mathit{SYN} := \mathit{SYN};$ (* *SYN* flag set may be set in the final segment of a simultaneous open *)

$\quad\quad FIN := FIN$; (* Processing of FIN flag handled *)
$\quad\quad win := win\_$;
$\quad\quad ws := ws\_discard$;
$\quad\quad urp := urp\_$;
$\quad\quad mss := mss\_discard$;
$\quad\quad ts := ts$;
$\quad\quad data := data$ (* Segment may have data *)
$\quad \rrbracket \wedge$

(* Equality of some type casts, and application of the socket's send window scaling to the received window advertisment *)
$win = \mathbf{w2n}\ win\_ \ll tcp\_sock.cb.snd\_scale \wedge$
$urp = \mathbf{w2n}\ urp\_$
$) \wedge$

(* The socket is fully connected so its complete address quad must match the address quad of the segment *seg*. By definition, *sock* is the socket with the best address match thus the auxiliary function tcp_socket_best_match is not required here. *)
$sock.is_1 = \uparrow i_1 \wedge sock.ps_1 = \uparrow p_1 \wedge$
$sock.is_2 = \uparrow i_2 \wedge sock.ps_2 = \uparrow p_2 \wedge$

(* The socket must be in a connected state, or is in the SYN_RECEIVED state and *seg* is the final segment completing a passive or simultaneous open. *)
$tcp\_sock.st \notin \{\mathrm{CLOSED}; \mathrm{LISTEN}; \mathrm{SYN\_SENT}\} \wedge$
$tcp\_sock.st \in \{\mathrm{SYN\_RECEIVED}; \mathrm{ESTABLISHED}; \mathrm{CLOSE\_WAIT}; \mathrm{FIN\_WAIT\_1}; \mathrm{FIN\_WAIT\_2};$
$\quad\quad\quad\quad \mathrm{CLOSING}; \mathrm{LAST\_ACK}; \mathrm{TIME\_WAIT}\} \wedge$

(* For a socket in the SYN_RECEIVED state check that the ACK is valid (the acknowledge value *ack* is not outside the range of sequence numbers that have been transmitted to the remote socket) and that the segment is not a LAND DoS attack (the segment's sequence number is not smaller than the remote socket's (the receiver from this socket's perspective) initial sequence number) *)
$\neg(tcp\_sock.st = \mathrm{SYN\_RECEIVED} \wedge$
$((ACK \wedge (ack \leq tcp\_sock.cb.snd\_una \vee ack > tcp\_sock.cb.snd\_max)) \vee$
$\quad seq < tcp\_sock.cb.irs)) \wedge$

(* If socket *sock* has previously emitted a *FIN* segment check that a thread is still associated with the socket, i.e. check that the socket still has a valid file identifier $fid \neq *$. If not, and the segment contains new data, the segment should not be processed by this rule as there is no thread to read the data from the socket after processing. Query: how does this *st* condition relate to *wesentafin* below? *)
$\neg(tcp\_sock.st \in \{\mathrm{FIN\_WAIT\_1}; \mathrm{CLOSING}; \mathrm{LAST\_ACK}; \mathrm{FIN\_WAIT\_2}; \mathrm{TIME\_WAIT}\} \wedge$
$sock.fid = * \wedge$
$seq + \mathbf{length}\ data > tcp\_sock.cb.rcv\_nxt) \wedge$

(* A *SYN* should be received only in the SYN_RECEIVED state. *)
$(SYN \implies tcp\_sock.st = \mathrm{SYN\_RECEIVED}) \wedge$

(* Socket *sock* has previously sent a *FIN* segment iff *snd_max* is strictly greater than the sequence number of the byte after the last byte in the send queue *sndq*. *)
$\mathbf{let}\ wesentafin = tcp\_sock.cb.snd\_max > tcp\_sock.cb.snd\_una + \mathbf{length}\ tcp\_sock.sndq\ \mathbf{in}$

(* If the socket *sock* has previously sent a *FIN* segment it has been acknowledged by segment *seg* if the segment has the *ACK* flag set and an acknowledgment number $ack \geq cb.snd\_max$. *)
$\mathbf{let}\ ourfinisacked = (wesentafin \wedge ACK \wedge ack \geq tcp\_sock.cb.snd\_max)\ \mathbf{in}$

(* Process the segment and return an updated socket state *)
(* The segment processing is performed by the four relations below, i.e., di3_topstuff, di3_ackstuff, di3_datastuff and di3_ststuff. Each of these relates a socket and bandwidth limiter state before the segment is processed to a tuple containing an updated socket, new bandwidth limiter state, a list of zero or more segments to output and a continue flag. The aim is to model the progression of the segment through `tcp_input()`. When the continue flag is **T** segment processing should continue. The infix function andThen applies the function on its left hand side and only continues with the function on its right hand side if the left hand function's continue flag is **T**. For a further explanation of this relational monad behaviour see *aux_relmonad* (p**??**). *)

```
let $topstuff$ =
    (* Initial processing of the segment: PAWS (protection against wrap sequence numbers); ensure segment
    is not entirely off the right hand edge of the window; timer updates, etc. For further information see
    di3_topstuff (p??).*)
    di3_topstuff $seg\ h.arch\ h.rttab\ h.ifds$(ticks_of $h.ticks$)
 and $ackstuff$ =
    (* Process the segment's acknowledgement number and do congestion control. See di3_ackstuff (p??).*)
    di3_ackstuff $tcp\_sock\ seg\ ourfinisacked\ h.arch\ h.rttab\ h.ifds$(ticks_of $h.ticks$)
 and $datastuff\ theststuff$ =
    (* Extract and reassemble data (including urgent data). See di3_datastuff (p??). *)
    di3_datastuff $theststuff\ tcp\_sock\ seg\ ourfinisacked\ h.arch$
 and $ststuff\ FIN\_reass$ =
    (* Possibly change the socket's state (especially on receipt of a valid $FIN$). See di3_ststuff (p??). *)
    di3_ststuff $FIN\_reass\ ourfinisacked\ ack$
 in
($topstuff$ andThen
   $ackstuff$ andThen
   $datastuff\ ststuff$)
($sock, bndlm$) (* state before *)
(($sock', bndlm', outsegs$), $continue'$)$\wedge$ (* state after *)

$sock'.pr$ = TCP_PROTO($tcp\_sock'$) $\wedge$

 (* If socket $sock$ was initially in the SYN_RECEIVED state and after processing $seg$ is in the ESTABLISHED
 state (or if the segment contained a $FIN$ and the socket is in one of the FIN_WAIT_1, FIN_WAIT_2 or
 CLOSE_WAIT states), the socket is probably on some other socket's incomplete connections queue and $seg$ is
 the final segment in a passive open. If it is on some other socket's incomplete connections queue the other socket
 is updated to move the newly connected socket's reference from the incomplete to the complete connections queue
 (unless the complete connection queue is full, in which case the new connection is dropped and all references
 to it are removed). If not, $seg$ is the final segment in a simultaneous open in which case no other sockets are
 updated. The auxiliary function di3_socks_update (p??) does all the hard work, updating the relevant sockets
 in the finite map $socks$ to yield $socks'$. *)
(if $tcp\_sock.st$ = SYN_RECEIVED $\wedge$
   $tcp\_sock'.st$ $\in$ {ESTABLISHED; FIN_WAIT_1; FIN_WAIT_2; CLOSE_WAIT} then
      di3_socks_update $sid(socks \oplus (sid, sock'))socks'$
 else
    (* If the socket was not initially in the SYN_RECEIVED state, i.e.$seg$ was processed by an already connected
    socket, ensure the updated socket is in the final finite maps of sockets. *)
   $socks' = socks \oplus (sid, sock'))$ $\wedge$

 (* Queue any segments for output on the host's output queue. In the common case there are no segments to be
 output as output is handled by $deliver\_out\_1$ etc. The exception is that di3_ackstuff (and its auxiliaries) require
 an immediate $ACK$ segment to be emitted under certain congestion control conditions. See di3_ackstuff (p??)
 and di3_newackstuff (p??) for further details. *)
enqueue_oq_list_qinfo($oq, outsegs, oq'$)
```

4.10 Example TCP traces

In Figures 7 and 8 we show the sequence of host LTS transition rules discovered by the automatic checker for two simple usages of TCP.

Both tests involves a BSD local host and a Linux auxiliary host. The first test creates a listening socket on the auxiliary host; creates a socket on the local host and connects it to the listening socket; accepts the connection; sends a string and then receives the string on the auxiliary host; and closes the connected socket. The second is a dual, with a listening socket on the BSD local host. It is not the other half of the first connection but a separate run, so the communicated segments are not identical.

The figures show the behaviour of the BSD host. On the right each shows the externally-visible transitions (socket calls and returns, and segment sends or receives) of the captured trace. On the left is the sequence of rule names of the trace found by the checker: *socket_1*; *epsilon_1*; *return_1*; etc. For each

Test Host: BSD(john) Aux Host: LINUX(alan)
Test Description: [TCP normal] Demonstration: create a listening socket on the auxiliary host; create a socket on the local host and connect to the listening socket; accept the connection; send a string and then receive the string on the auxiliary host; close both sockets (no tcpcb)
/usr/groups/tthee/batch/demo-traces/trace5000

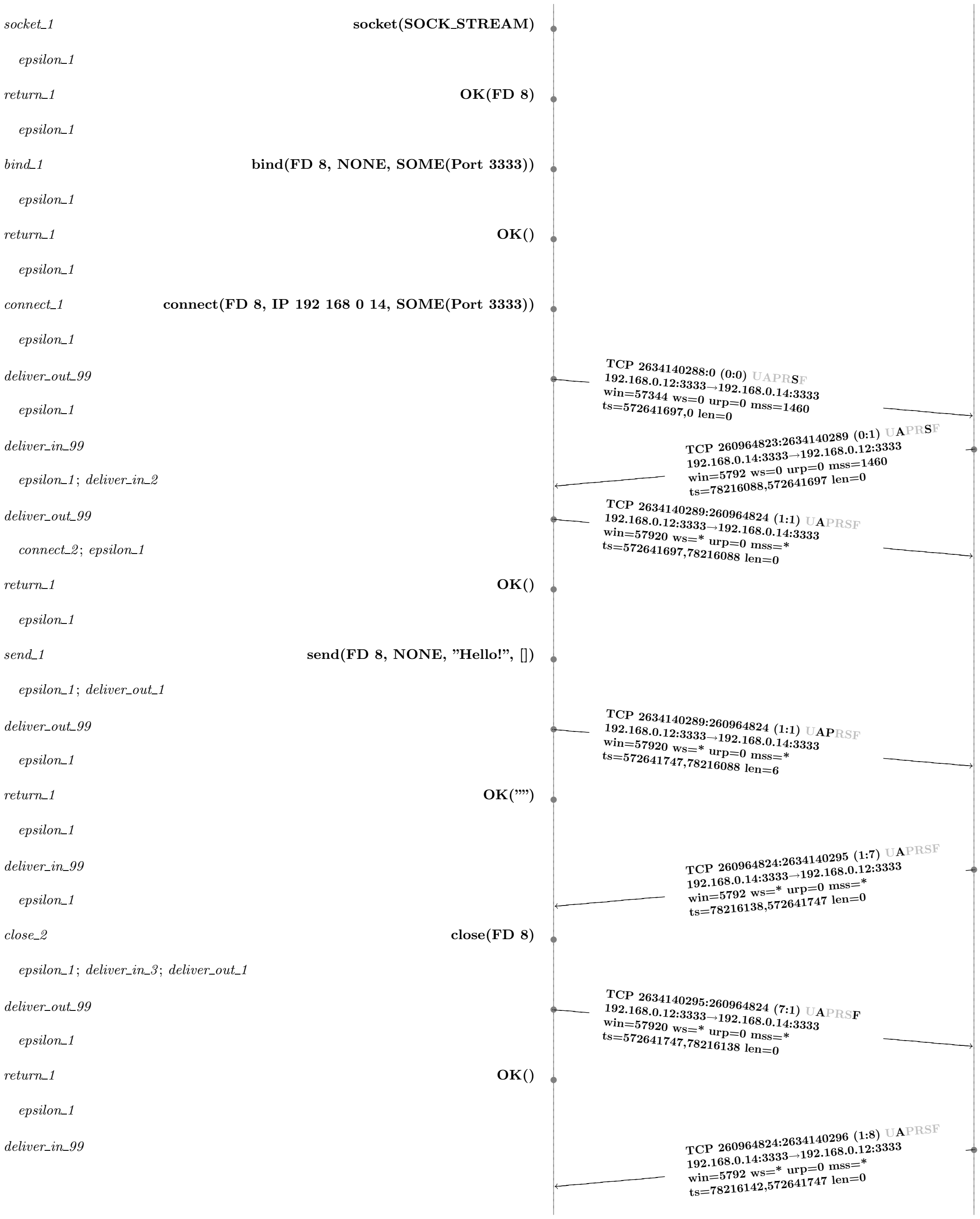

socket_1
socket(SOCK_STREAM)
epsilon_1
return_1
OK(FD 8)
epsilon_1
bind_1
bind(FD 8, NONE, SOME(Port 3333))
epsilon_1
return_1
OK()
epsilon_1
connect_1
connect(FD 8, IP 192 168 0 14, SOME(Port 3333))
epsilon_1
deliver_out_99
TCP 2634140288:0 (0:0) UAPRSF
192.168.0.12:3333→192.168.0.14:3333
win=57344 ws=0 urp=0 mss=1460
ts=572641697,0 len=0
epsilon_1
deliver_in_99
TCP 260964823:2634140289 (0:1) UAPRSF
192.168.0.14:3333→192.168.0.12:3333
win=5792 ws=0 urp=0 mss=1460
ts=78216088,572641697 len=0
epsilon_1; deliver_in_2
deliver_out_99
TCP 2634140289:260964824 (1:1) UAPRSF
192.168.0.12:3333→192.168.0.14:3333
win=57920 ws=* urp=0 mss=*
ts=572641697,78216088 len=0
connect_2; epsilon_1
return_1
OK()
epsilon_1
send_1
send(FD 8, NONE, ”Hello!”, [])
epsilon_1; deliver_out_1
deliver_out_99
TCP 2634140289:260964824 (1:1) UAPRSF
192.168.0.12:3333→192.168.0.14:3333
win=57920 ws=* urp=0 mss=*
ts=572641747,78216088 len=6
epsilon_1
return_1
OK(””)
epsilon_1
deliver_in_99
TCP 260964824:2634140295 (1:7) UAPRSF
192.168.0.14:3333→192.168.0.12:3333
win=5792 ws=* urp=0 mss=*
ts=78216138,572641747 len=0
epsilon_1
close_2
close(FD 8)
epsilon_1; deliver_in_3; deliver_out_1
deliver_out_99
TCP 2634140295:260964824 (7:1) UAPRSF
192.168.0.12:3333→192.168.0.14:3333
win=57920 ws=* urp=0 mss=*
ts=572641747,78216138 len=0
epsilon_1
return_1
OK()
epsilon_1
deliver_in_99
TCP 260964824:2634140296 (1:8) UAPRSF
192.168.0.14:3333→192.168.0.12:3333
win=5792 ws=* urp=0 mss=*
ts=78216142,572641747 len=0

Figure 7: Sample checked TCP trace, with rule firings – connect() end

Test Host: BSD(john) Aux Host: LINUX(alan)
Test Description: [TCP normal] Demonstration: create a listening socket on the auxiliary host; create a socket on the local host and connect to the listening socket; accept the connection; send a string and then receive the string on the auxiliary host; close both sockets (no tcpcb)
/usr/groups/tthee/batch/demo-traces/trace5002

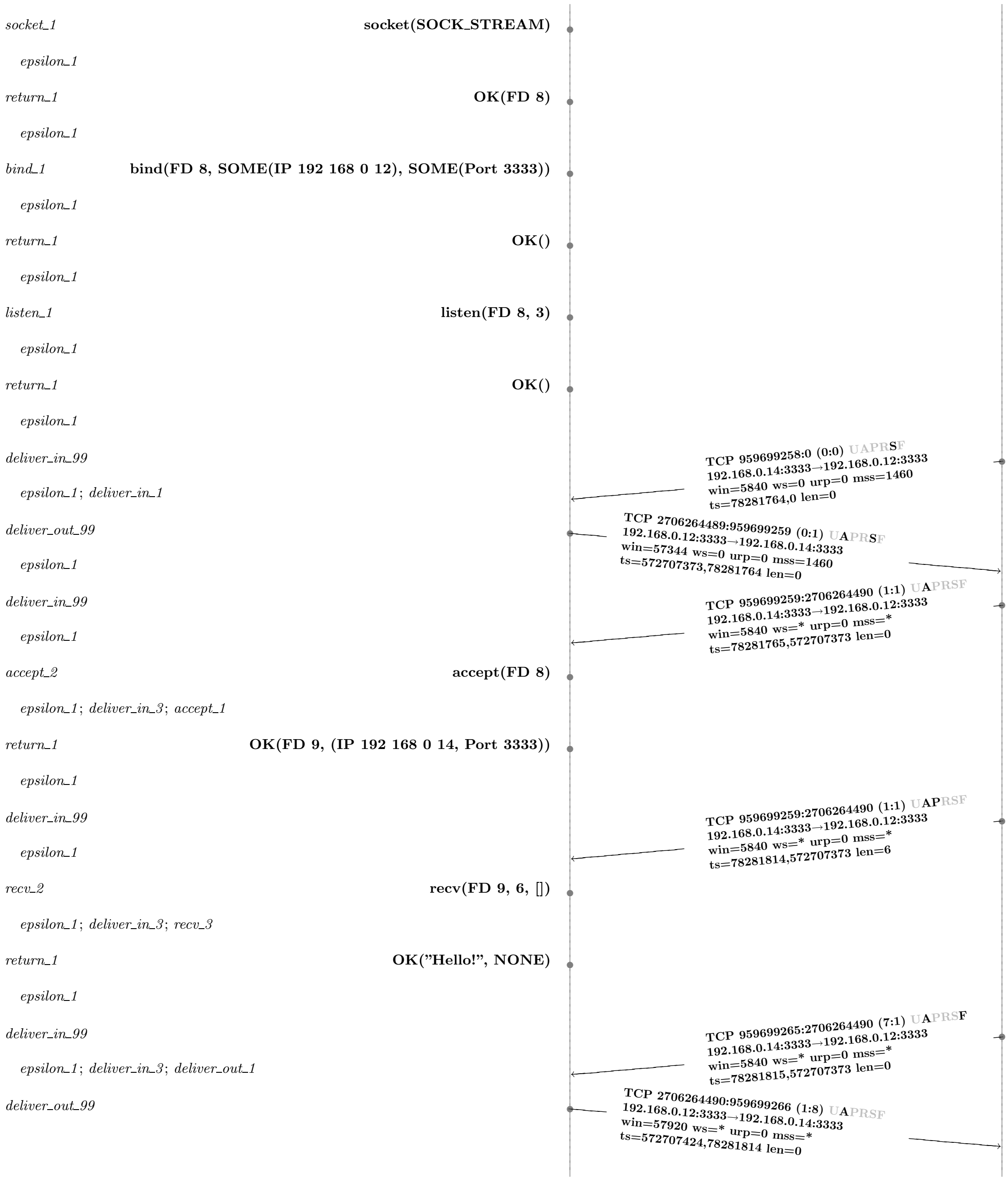

socket_1 **socket(SOCK_STREAM)**
epsilon_1
return_1 **OK(FD 8)**
epsilon_1
bind_1 **bind(FD 8, SOME(IP 192 168 0 12), SOME(Port 3333))**
epsilon_1
return_1 **OK()**
epsilon_1
listen_1 **listen(FD 8, 3)**
epsilon_1
return_1 **OK()**
epsilon_1
deliver_in_99
TCP 959699258:0 (0:0) UAPRSF 192.168.0.14:3333→192.168.0.12:3333 win=5840 ws=0 urp=0 mss=1460 ts=78281764,0 len=0
epsilon_1; *deliver_in_1*
deliver_out_99
TCP 2706264489:959699259 (0:1) UAPRSF 192.168.0.12:3333→192.168.0.14:3333 win=57344 ws=0 urp=0 mss=1460 ts=572707373,78281764 len=0
epsilon_1
deliver_in_99
TCP 959699259:2706264490 (1:1) UAPRSF 192.168.0.14:3333→192.168.0.12:3333 win=5840 ws=* urp=0 mss=* ts=78281765,572707373 len=0
epsilon_1
accept_2 **accept(FD 8)**
epsilon_1; *deliver_in_3*; *accept_1*
return_1 **OK(FD 9, (IP 192 168 0 14, Port 3333))**
epsilon_1
deliver_in_99
TCP 959699259:2706264490 (1:1) UAPRSF 192.168.0.14:3333→192.168.0.12:3333 win=5840 ws=* urp=0 mss=* ts=78281814,572707373 len=6
epsilon_1
recv_2 **recv(FD 9, 6, [])**
epsilon_1; *deliver_in_3*; *recv_3*
return_1 **OK("Hello!", NONE)**
epsilon_1
deliver_in_99
TCP 959699265:2706264490 (7:1) UAPRSF 192.168.0.14:3333→192.168.0.12:3333 win=5840 ws=* urp=0 mss=* ts=78281815,572707373 len=0
epsilon_1; *deliver_in_3*; *deliver_out_1*
deliver_out_99
TCP 2706264490:959699266 (1:8) UAPRSF 192.168.0.12:3333→192.168.0.14:3333 win=57920 ws=* urp=0 mss=* ts=572707424,78281814 len=0

Figure 8: Sample checked TCP trace, with rule firings – listen() end

externally-visible transition there is the single rule name of the rule that fired with that label. In between these there are the sequences of rule names for internal (τ) and time-passage (dur) transitions. These traces were generated without BSD trace debug transitions, to reduce visual clutter. Being reasonably simple, they can be checked without them. Timestamp values, thread ids, and the data carried in TCP segments are elided from the figure, as are the symbolic host states computed at each point.

5 Validation — the Evaluator

5.1 Essence of the problem

Given a (non-deterministic) labelled transition system $\xrightarrow{l}$, an initial host h_0, and a sequence of (experimentally observed) labels $l_1 \dots l_n$, 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. The presence of these additional steps will need to be inferred. Recall that the transition system is given as a set of rules of the form

$$\vdash P(h_0, l, h) \Rightarrow h_0 \xrightarrow{l} h$$

Ignoring the τ transitions that might be interspersed with the other labels, the aim is to demonstrate a sequence

$$h_0 \xrightarrow{l_1} h_1 \xrightarrow{l_2} \cdots \xrightarrow{l_{n-1}} h_{n-1} \xrightarrow{l_n} h_n$$

Alternatively, the problem can be seen as a simple form of model-checking: determining the validity of the following existential statement:

$$M, h_0 \vdash \langle l_1 \rangle \dots \langle l_n \rangle. \top$$

with M the model, h_0 the initial host, and $\langle l \rangle.\phi$ a modal formula expressing the notion of being able to make an l-step, and then having the resulting state satisfy ϕ.

If the system were deterministic, the problem would be easily solved. The initial conditions are completely specified and the problem would be one of mechanical calculation with values that were always ground.

Because the system is non-deterministic, the problem becomes one of exploring the tree of all possible traces that are consistent with the given label sequence. This exploration is not entirely label-driven, as it must consider the possibility that a τ transition is required.

Non-determinism arises in two different ways:

- two or more rules may apply to the same host-label pair (or the host may be able to undergo a τ transition);

- a single rule's side conditions may not constrain the resulting host to take on just one possible value.

These two sorts of non-determinism do not correspond to any deep semantic difference, but do affect the way in which the problem is solved.

Because labels come in a small number of different categories, the number of rules that might apply to any given host-label pair is relatively small. It is clearly reasonable to explicitly model this non-determinism by explicit branching within a tree-structured search-space. The search through this space is done depth-first.

Possible τ transitions are checked last: if considering host h and a sequence of future labels, and no normal rule allows for a successful trace, posit a τ transition at this point, followed by the same sequence of labels. As long as hosts can not make infinite sequences of τ transitions, the search-space remains finite.

An example of the second sort of non-determinism comes when a resulting host is to include some numeric quantity, but where the model only constrains this number to fall within certain bounds. It is clearly foolish to explicitly model all these possibilities as branching (indeed, if the value is real-valued, there are an infinite number of possibilities). Instead, the system maintains sets of constraints, attached to each transition. Instead of finding a sequence of theorems of the form

$$\begin{aligned}
&\vdash h_0 \xrightarrow{l_1} h_1\\
&\vdash h_1 \xrightarrow{l_2} h_2\\
&\cdots\\
&\vdash h_{n-1} \xrightarrow{l_n} h_n
\end{aligned}$$

HOL must find a sequence of theorems of the form

$$\begin{aligned}
&\Gamma_0 \vdash h_0 \xrightarrow{l_1} h_1\\
&\Gamma_0 \cup \Gamma_1 \vdash h_1 \xrightarrow{l_2} h_2\\
&\cdots\\
&\textstyle\bigcup_{i=1}^{n} \Gamma_i \vdash h_{n-1} \xrightarrow{l_n} h_n
\end{aligned}$$

where each Γ_i is the set of constraints generated by the i-th transition. If the fresh constraints were only generated because new components of output hosts were under-constrained, there would be no difficulty with this.

Unfortunately, the side-conditions associated with each rule will inevitably refer to input host component values that are no longer ground, but which are instead constrained by a constraint generated by the action of an earlier rule.

For example, imagine that the first transition of a trace has made the v component of the host have a value between 1 and 100. Now faced with an l-transition, the system must eliminate those rules which allow for that transition if v is greater than 150.

HOL accumulates constraint sets as a trace proceeds, and checks them for satisfiability. The satisfiability check takes the form of simplifying each assumption in turn, while assuming all of the other assumptions as context. HOL simplification includes the action of arithmetic decision procedures, so unsatisfiable arithmetic constraints are discovered as well as more straightforward problems (for example, the simplifier "knows" that $(s = [\,]) \wedge (s = h :: t)$ is impossible because the nil and cons constructors for lists are disjoint).

5.1.1 Constraint instantiation

As a checking run proceeds, later labels may determine variables that had initially been under-determined. For example, Windows XP picks file descriptors for sockets non-deterministically, so on this architecture the specification for the socket call only requires that the new descriptor be fresh. As a trace proceeds however, the actual descriptor value chosen will be revealed (a label or two later, the value will appear in the return-label that is passed back to the caller). In this situation, and others like it, the set of constraints attached to the relevant theorem will get smaller when the equality is everywhere eliminated. Though the checker does not explicitly do this step, the effect is as if the earlier theorems in the run had also been instantiated with the value chosen. Of course, if the value is inconsistent with the initial constraints, then this will be detected because those constraints will have been inherited from the stage when they were generated.

5.1.2 Case splitting

Sometimes a new constraint will be of a form where it is clear that it is equivalent to a disjunction of two possibilities. Then it often makes sense to case-split and consider each arm of the disjunction separately

$$\begin{tikzcd}
 & \Gamma, p \vee q \vdash h_{i-1} \xrightarrow{l_i} h_i \arrow[dl] \arrow[dr] & \\
\Gamma, p \vdash h_{i-1} \xrightarrow{l_i} h_i & & \Gamma, q \vdash h_{i-1} \xrightarrow{l_i} h_i
\end{tikzcd}$$

At the moment, such splitting is done on large disjunctions (as above), and large conditional expressions that appear in the output host. For example, if the current theorem is

$$\Gamma \vdash h_0 \xrightarrow{l} (\ldots \mathsf{if}\ p\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \ldots)$$

then two new theorems are created:

$$\Gamma, p \vdash h_0 \xrightarrow{l} (\dots e_1 \dots)$$

and

$$\Gamma, \neg p \vdash h_0 \xrightarrow{l} (\dots e_2 \dots)$$

and both branches are explored (again, in a depth-first order).

5.1.3 Adding constraints and completeness

It is always *sound* to add fresh assumptions to a theorem. The following is a rule of inference in HOL:

$$\frac{\Gamma \vdash t}{\Gamma, p \vdash t}$$

Adding arbitrary constraints in this way may allow heuristic knowledge to be added, and thus used to guide the search for a satisfying path. To date, we have not attempted to do this. The risk of such an activity is not one of unsoundness, but rather incompleteness: if we add an assertion p, and then find that this produces an unsatisfiable set of constraints, we may incorrectly conclude that there is no satisfying path.

On the other hand, we *do* add constraints that are consequences of existing assumptions. This preserves satisfiability. For example, traces often produce rather complicated expressions about which arithmetic decision procedures can not reason directly. We help the procedures draw conclusions by separately inferring upper and lower bounds information about such expressions, and adding these new (but redundant) assumptions to the theorem.

5.2 Model translation

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 analsyses. 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 equivalent forms. At their best, these translations are ML code written to handle an infinite family of possibilities. In other cases, we prove rather specific theorems that state a particular rule or auxiliary function is equivalent to an alternative form. This theorem then justifies the use of the more efficient expression of the same semantics.

5.3 Time and urgency

Our specification explicitly models the passage of time. The relevant rule is

epsilon_1 **all: misc nonurgent** **Time passes**

$$h \xRightarrow{dur} h'$$

let $hs' = \text{Time\_Pass\_host}\ dur\ h$ **in**
is_some $hs' \land$
$h' \in (\textbf{the}\ hs') \land$
$\neg(\exists rn\ rp\ rc\ lbl\ h'.rn/*rp, rc*/h \xrightarrow{lbl} h' \land \text{is\_urgent}\ rc)$

Description Allow time to pass for *dur* seconds. This is only enabled if the host state is not urgent, i.e. if no urgent rule can fire. Notice that, apart from when a timer becomes zero, a host state never becomes urgent due merely to time passage. This means we need only test for urgency at the beginning of the time interval, not throughout it.

The rule says that host h can have its internal timers updated by the duration dur to become host state h', where h is not an *urgent* state. A host is urgent if it is able to undergo an urgent τ transition. Such transitions represent actions that are held to happen instantaneously, and which must "fire" before any time elapses.

The trace-checker does not check for non-urgency by actually trying all of the urgent rules in turn. Instead, it uses a theorem (proved once and for all as the system builds), that provides an approximate characterisation of non-urgency. If this is satisfied, the above rule's side-condition's can be discharged, and progress made. If the approximation can not be proved true, then a τ step is attempted.

5.4 Laziness in symbolic evaluation

Because hosts quickly lose their groundedness as a checking run proceeds, many of the values being computed with are actually constrained variables. Such variables may even come to be equated with other expressions, where those expressions in turn include unground components. It is important in this setting to retain variable bindings rather than simply substituting them out. Substituting unground expressions through large terms may result in many instances of the same, expensive computation when those expressions do eventually become ground.

This is analogous to the way in which a lazy language keeps pending computations hidden in a "thunk" and does not evaluate them prematurely. The difference is that lazy languages "force" thunks when evaluation determines that their values are required. In the trace-checking setting, expressions yield values as the logical context becomes richer, not on the basis of whether or not those values are required elsewhere.

Moreover, as soon as an expression yields up a little information about its structure it is important to let this information flow into the rest of the formula. For example, if the current theorem is

$$x = E \vdash \ldots (\mathsf{if}\ x = [\,]\ \mathsf{then}\ f(x)\ \mathsf{else}\ g(x)) \ldots$$

then it is important not to substitute E for x and end up working with two copies of (presumably complicated) expression E. On the other hand, future work may reveal that E is actually of the form $h :: t$ for some (themselves complicated) expressions h and t.

In this case, the theorem must become

$$v_1 = h, v_2 = t \vdash \ldots (g(v_1 :: v_2)) \ldots$$

In this situation, the application of g to a list known to a be a "cons-cell" may lead to future useful simplification.

To implement this, the checker knows how to isolate equalities to prevent them from being instantiated, and how to detect certain expressions as value-forms, or partial value-forms.

5.5 Checker outcomes

There are several possible results of running the checker on a trace. It may *succeed*, indicating that a sequence of symbolic transitions has been found that the entire trace matches, or *fail*, indicating that no such sequence exists. Additionally, we use several heuristics to quickly terminate runs that are likely to fail: a run is *too complicated* if the constraints have become large, is terminated with *excessive backtracking* if the amount of backtracking is large compared with the length of the trace, may be terminated with *output queue too long*, or is terminated with *send datagram mismatch* if the structure of a sent datagram does not match that seen. Runs may also end with a HOL *internal error* or a *crash* (most often if the machine is rebooted). There is no guarantee that a run will terminate, and during development we have encountered some apparently-nonterminating runs, e.g. due to simplification cycles, but it has not been hard to avoid them.

A *fail* indicates there is definitely a problem in the model (or, in the early stages of the project, in the trace generation tools). The other non-*success* results may indicate problems in the model or limitations in the symbolic evaluator.

At the end of a successful run the checker outputs the list of the rule names used for the resulting transition sequence, together with (for TCP) the socket state changes involved.

5.6 Example checker output

The checker outputs an HTML log file during its search, showing each step, the label for that step, the rule(s) considered, and other data. An extract from the log file for a UDP trace in shown in Figure 9.

```
HOL Trace: trace1148

[Show/hide variables and constraints.]

==Working on trace file /usr/groups/tthee/batch/autotest-udp-2004-05-25T00:00:00+0100/trace1148 [plain] [ps]
==Date: 2004-12-15 T 18:11:37 Z (Wed)

(* Test Host: WINXP(glia)  Aux Host: BSD(john) *)
(* Test Description: [UDP normal] shutdown() -- for a fresh UDP socket with bound quad LOCAL_IP,KNOWN_PORT_ERR,REMOTE_IP,
KNOWN_PORT, call shutdown() on the receive side only and attempt to receive more data when there is data on the rcvq *)

[...]

==Step 20 at <2004-12-15 T 18:13:27 Z (Wed)> 1103134407:
 Lh_call (TID 66040,send (FD 18272,NONE,"Generate ICMP",[]))
== Backtrack limit counts down to 61
initial: 0.250s (#poss: 22)
==Attempting send_1 -- pre_host -- post_host -- REJECTED
==Attempting send_2 -- pre_host -- post_host -- REJECTED
==Attempting send_9 -- pre_host -- post_host -- phase2 -- ctxtclean
CPU time elapsed : 13.040 seconds (unwind: 0.000)

==Successful transition of send_9
==Backtrack limit counts down to 60



==Step 21 at <2004-12-15 T 18:13:40 Z (Wed)> 1103134420:
 attempting time passage with duration 87 / 500000
CPU time elapsed : 8.200 seconds(unwind: 0.000)

==Successful transition of epsilon_1

==New variables: (ticks'10 :ticker)
==New constraints:  Time_Pass_ticker (87 / 500000) ticks'9 ticks'10



==Step 22 at <2004-12-15 T 18:13:49 Z (Wed)> 1103134429:
 Lh_return (TID 66040,TL_err (OK (TL_string "")))
== Backtrack limit counts down to 59
initial: 0.280s (#poss: 2)
==Attempting return_1 -- pre_host -- post_host -- phase2 -- ctxtclean
CPU time elapsed : 2.550 seconds (unwind: 0.000)

==Successful transition of return_1
==Backtrack limit counts down to 58



==Step 23 at <2004-12-15 T 18:13:52 Z (Wed)> 1103134432:
 attempting time passage with duration 193 / 1000000
CPU time elapsed : 8.600 seconds(unwind: 0.000)

==Successful transition of epsilon_1

==New variables: (ticks'11 :ticker)
==New constraints: Time_Pass_ticker (193 / 1000000) ticks'10 ticks'11



==Step 24 at <2004-12-15 T 18:14:01 Z (Wed)> 1103134442:
 Lh_senddatagram
  (UDP
     <|is1 := SOME (IP 192 168 0 3); is2 := SOME (IP 192 168 0 12);
       ps1 := SOME (Port 3333); ps2 := SOME (Port 3333);
       data :=
         [CHR 71; CHR 101; CHR 110; CHR 101; CHR 114; CHR 97; CHR 116;
          CHR 101; CHR 32; CHR 73; CHR 67; CHR 77; CHR 80]|>)
== Backtrack limit counts down to 57
initial: 0.890s (#poss: 2)
==Attempting deliver_out_99 -- pre_host -- post_host -- phase2 -- ctxtclean
CPU time elapsed : 4.090 seconds (unwind: 0.000)

==Successful transition of deliver_out_99
==Backtrack limit counts down to 56
```

Figure 9: Checker output

```
|- deliver_out_99 /* rp_all, network nonurgent */
   <|arch := WinXP_Prof_SP1; privs := T;
     ifds :=
       FEMPTY |+
       (ETH 0,
        <|ipset := {IP 192 168 0 3}; primary := IP 192 168 0 3;
          netmask := NETMASK 24; up := T|>) |+
       (LO,
        <|ipset := {IP 127 0 0 1}; primary := IP 127 0 0 1;
          netmask := NETMASK 8; up := T|>);
     rttab :=
       [<|destination_ip := IP 127 0 0 1;
          destination_netmask := NETMASK 8; ifid := LO|>;
        <|destination_ip := IP 192 168 0 0;
          destination_netmask := NETMASK 24; ifid := ETH 0|>;
        <|destination_ip := IP 192 168 1 0;
          destination_netmask := NETMASK 24; ifid := ETH 0|>;
        <|destination_ip := IP 128 232 13 142;
          destination_netmask := NETMASK 20; ifid := ETH 1|>];
     ts :=
       FEMPTY |+
       (TID 66040,
        Timed (Run,Timer (193 / 1000000,time_infty,time_infty)));
     files :=
       FEMPTY |+ (FID 0,File (FT_Console,<|b := (\x. F)|>)) |+
       (fid,File (FT_Socket sid,<|b := ff_default_b|>));
     socks :=
       FEMPTY |+
       (sid,
        <|fid := SOME fid;
          sf :=
            <|b := sf_default_b;
              n := sf_default_n WinXP_Prof_SP1 SOCK_DGRAM;
              t := sf_default_t|>; is1 := SOME (IP 192 168 0 3);
          ps1 := SOME (Port 3333); is2 := SOME (IP 192 168 0 12);
          ps2 := SOME (Port 3333); es := NONE; cantsndmore := F;
          cantrcvmore := F; pr := UDP_PROTO <|rcvq := []|>|>);
     listen := []; bound := [sid];
     iq := Timed ([],Timer (203089 / 500000,time_infty,time_infty));
     oq :=
       Timed
         ([UDP
             <|is1 := SOME (IP 192 168 0 3);
               is2 := SOME (IP 192 168 0 12); ps1 := SOME (Port 3333);
               ps2 := SOME (Port 3333);
               data :=
                 [CHR 71; CHR 101; CHR 110; CHR 101; CHR 114; CHR 97;
                  CHR 116; CHR 101; CHR 32; CHR 73; CHR 67; CHR 77;
                  CHR 80]|>],Timer (367 / 1000000,time 0,time 1));
     bndlm := []; ticks := ticks'11; fds := FEMPTY |+ (FD 18272,fid)|>



     -- Lh_senddatagram
          (UDP
             <|is1 := SOME (IP 192 168 0 3);
               is2 := SOME (IP 192 168 0 12); ps1 := SOME (Port 3333);
               ps2 := SOME (Port 3333);
               data :=
                 [CHR 71; CHR 101; CHR 110; CHR 101; CHR 114; CHR 97;
                  CHR 116; CHR 101; CHR 32; CHR 73; CHR 67; CHR 77;
                  CHR 80]|>) -->



   <|arch := WinXP_Prof_SP1; privs := T;
     ifds := ...
     rttab := ...
     ts := ...
     files := ...
     socks := ...     listen := []; bound := [sid];
     iq := Timed ([],Timer (203089 / 500000,time_infty,time_infty));
     oq := Timed ([],Timer (0,time_infty,time_infty));
     bndlm := [];  ticks := ticks'11; fds := FEMPTY |+ (FD 18272,fid)|>
```

Figure 10: Checker output: the symbolic transition derived for Step 24

For Step 20 the figure shows that two rules (*send_1* and *send_2*) were attempted before the matching *send_9* transition was found. No backtracking is visisble in this example. The figure shows new variables and constraints introduced at each step, which in this example are just the `ticks'10` and `ticks'11` variables and time passage constraints on them.

Javascript is used to allow the user to toggle display of this data and also the details of the actual symbolic transitions found: clicking on the Step 24 region of this display expands it to show the information in Figure 10. This shows a transition of rule *deliver_out_99*, taking a UDP datagram from the host's outqueue and putting it on the wire. The figure shows the initial symbolic host state `<| arch := ...;... |>`, the transition label `-- Lh_senddatagram (UDP ...) -->`, and the resulting symbolic host state, which has an empty outqueue `oq`. In this example the host states are almost ground, with just the variables `fid`, `sid`, and `ticks'11` kept symbolic. In TCP traces, especially during connection establishment, there may be many more complex constraints. For example, a *connect_1* transition 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.

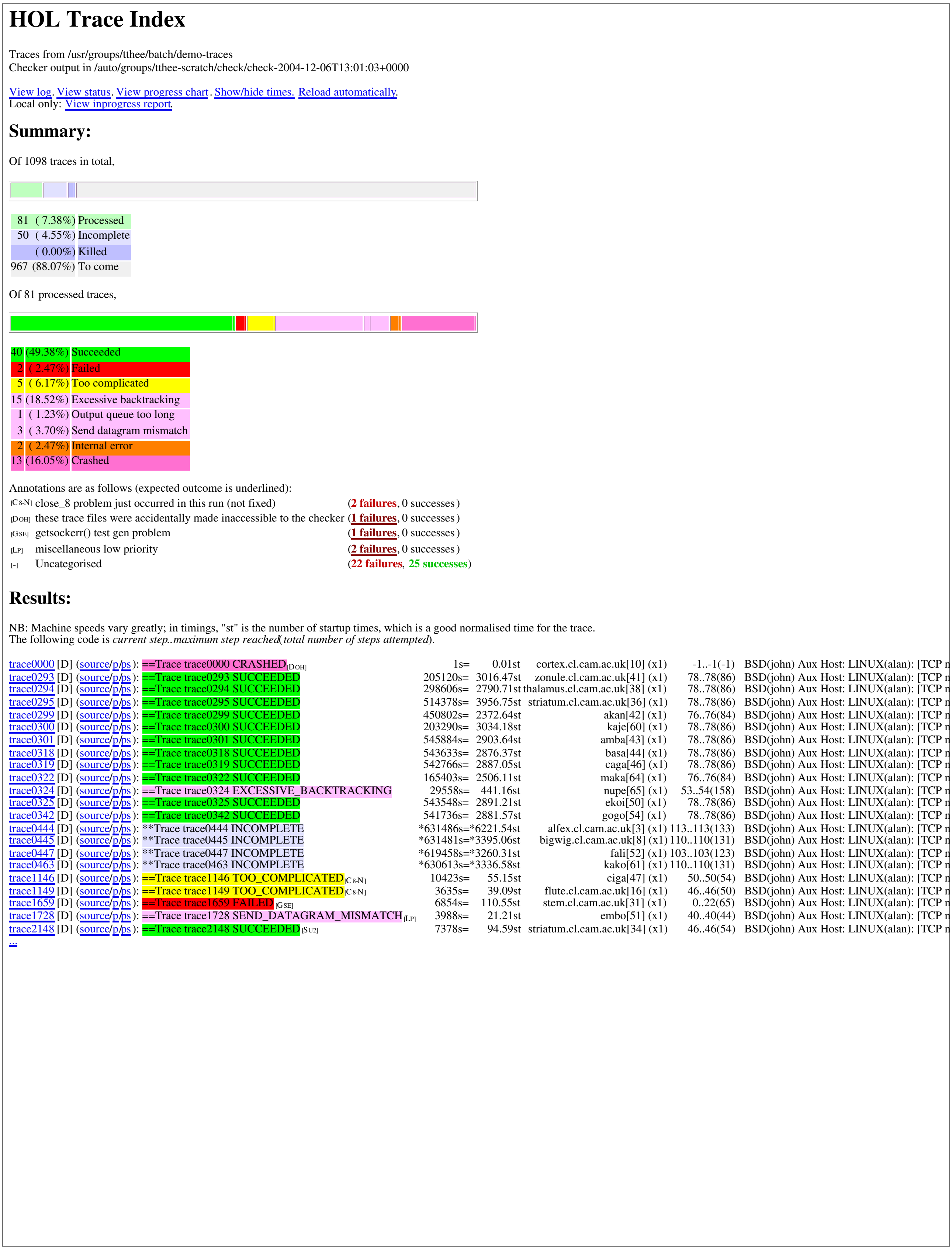
6 Validation – Checking infrastructure

Each trace is checked: validated against the model using the symbolic evaluator of §5. The results of validation are used to correct the model, working towards a model that accepts every trace. Trace checking is computationally expensive, but for good coverage we want to check many traces. We therefore distribute checking over as many processors as possible. Each trace (apart from initialisation of the evaluator) is independent, so this is conceptually straightforward, but coordinating the whole effort still requires a non-trivial infrastructure.

Checking is compute-bound, not memory- 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. 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 machines vary in speed by a factor of 3 (excluding a few slow outliers), with the fastest currently having dual 3.06GHz Xeon or dual Opteron 250 processors. On the former it is most efficient overall to run 4 trace check jobs simultaneously, though for an individual trace it would be significantly faster to run at most 2 in parallel. We currently rely on a common NFS-mounted file system shared between all the worker machines. This is a limitation, and we would like to generalise our tools to permit remote clients communicating programs and data on an as-needed basis with the coordinator.



HOL Trace Index

Traces from /usr/groups/tthee/batch/demo-traces
Checker output in /auto/groups/tthee-scratch/check/check-2004-12-06T13:01:03+0000

View log. View status. View progress chart. Show/hide times. Reload automatically.
Local only: View inprogress report.

Summary:

Of 1098 traces in total,

81	(7.38%)	Processed
50	(4.55%)	Incomplete
	(0.00%)	Killed
967	(88.07%)	To come

Of 81 processed traces,

40	(49.38%)	Succeeded
2	(2.47%)	Failed
5	(6.17%)	Too complicated
15	(18.52%)	Excessive backtracking
1	(1.23%)	Output queue too long
3	(3.70%)	Send datagram mismatch
2	(2.47%)	Internal error
13	(16.05%)	Crashed

Annotations are as follows (expected outcome is underlined):

[C8-N]	close_8 problem just occurred in this run (not fixed)	(**2 failures**, 0 successes)
[DOH]	these trace files were accidentally made inaccessible to the checker	(**1 failures**, 0 successes)
[GSE]	getsockerr() test gen problem	(**1 failures**, 0 successes)
[LP]	miscellaneous low priority	(**2 failures**, 0 successes)
[–]	Uncategorised	(**22 failures**, **25 successes**)

Results:

NB: Machine speeds vary greatly; in timings, "st" is the number of startup times, which is a good normalised time for the trace.
The following code is *current step..maximum step reached*(*total number of steps attempted*).

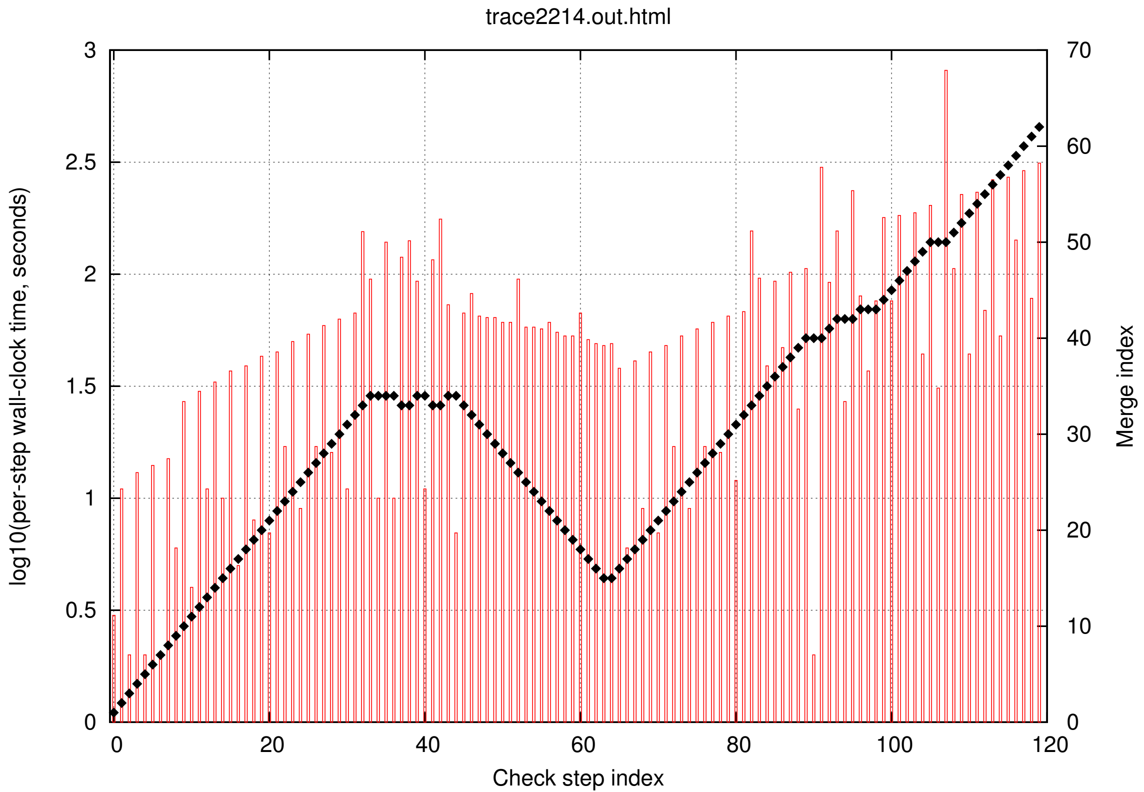
trace0000 [D] (source/p/ps):	==Trace trace0000 CRASHED [DOH]	1s=	0.01st	cortex.cl.cam.ac.uk[10] (x1)	-1..-1(-1)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0293 [D] (source/p/ps):	==Trace trace0293 SUCCEEDED	205120s=	3016.47st	zonule.cl.cam.ac.uk[41] (x1)	78..78(86)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0294 [D] (source/p/ps):	==Trace trace0294 SUCCEEDED	298606s=	2790.71st	thalamus.cl.cam.ac.uk[38] (x1)	78..78(86)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0295 [D] (source/p/ps):	==Trace trace0295 SUCCEEDED	514378s=	3956.75st	striatum.cl.cam.ac.uk[36] (x1)	78..78(86)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0299 [D] (source/p/ps):	==Trace trace0299 SUCCEEDED	450802s=	2372.64st	akan[42] (x1)	76..76(84)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0300 [D] (source/p/ps):	==Trace trace0300 SUCCEEDED	203290s=	3034.18st	kaje[60] (x1)	78..78(86)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0301 [D] (source/p/ps):	==Trace trace0301 SUCCEEDED	545884s=	2903.64st	amba[43] (x1)	78..78(86)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0318 [D] (source/p/ps):	==Trace trace0318 SUCCEEDED	543633s=	2876.37st	basa[44] (x1)	78..78(86)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0319 [D] (source/p/ps):	==Trace trace0319 SUCCEEDED	542766s=	2887.05st	caga[46] (x1)	78..78(86)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0322 [D] (source/p/ps):	==Trace trace0322 SUCCEEDED	165403s=	2506.11st	maka[64] (x1)	76..76(84)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0324 [D] (source/p/ps):	==Trace trace0324 EXCESSIVE_BACKTRACKING	29558s=	441.16st	nupe[65] (x1)	53..54(158)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0325 [D] (source/p/ps):	==Trace trace0325 SUCCEEDED	543548s=	2891.21st	ekoi[50] (x1)	78..78(86)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0342 [D] (source/p/ps):	==Trace trace0342 SUCCEEDED	541736s=	2881.57st	gogo[54] (x1)	78..78(86)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0444 [D] (source/p/ps):	**Trace trace0444 INCOMPLETE	*631486s=	*6221.54st	alfex.cl.cam.ac.uk[3] (x1)	113..113(133)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0445 [D] (source/p/ps):	**Trace trace0445 INCOMPLETE	*631481s=	*3395.06st	bigwig.cl.cam.ac.uk[8] (x1)	110..110(131)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0447 [D] (source/p/ps):	**Trace trace0447 INCOMPLETE	*619458s=	*3260.31st	fali[52] (x1)	103..103(123)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace0463 [D] (source/p/ps):	**Trace trace0463 INCOMPLETE	*630613s=	*3336.58st	kako[61] (x1)	110..110(131)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace1146 [D] (source/p/ps):	==Trace trace1146 TOO_COMPLICATED [C8-N]	10423s=	55.15st	ciga[47] (x1)	50..50(54)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace1149 [D] (source/p/ps):	==Trace trace1149 TOO_COMPLICATED [C8-N]	3635s=	39.09st	flute.cl.cam.ac.uk[16] (x1)	46..46(50)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace1659 [D] (source/p/ps):	==Trace trace1659 FAILED [GSE]	6854s=	110.55st	stem.cl.cam.ac.uk[31] (x1)	0..22(65)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace1728 [D] (source/p/ps):	==Trace trace1728 SEND_DATAGRAM_MISMATCH [LP]	3988s=	21.21st	embo[51] (x1)	40..40(44)	BSD(john) Aux Host: LINUX(alan): [TCP n
trace2148 [D] (source/p/ps):	==Trace trace2148 SUCCEEDED [SU2]	7378s=	94.59st	striatum.cl.cam.ac.uk[34] (x1)	46..46(54)	BSD(john) Aux Host: LINUX(alan): [TCP n

...

Figure 11: Checker monitoring — HOL trace index

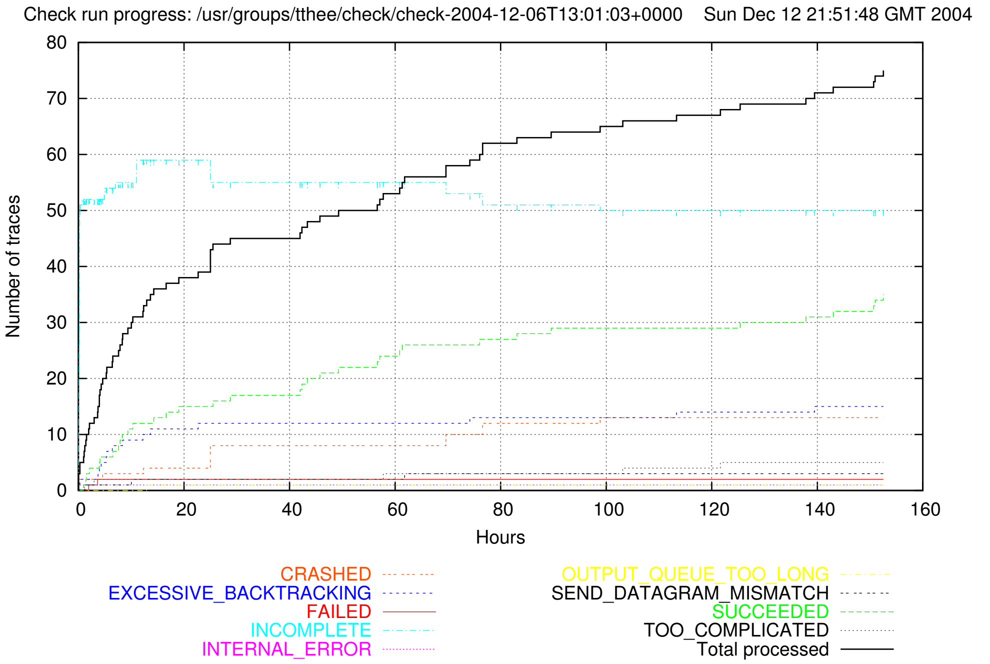
```
Worker status at <2004-09-14 T 19:54:15 Z (Tue)> 1095191655:
[  1] on        alfex.cl.cam.ac.uk:    4 jobs,    3 cmpl    0 crsh    0 kill,    1 starts.  Processing trace1140.
[  2] on        alfex.cl.cam.ac.uk:    5 jobs,    4 cmpl    0 crsh    0 kill,    1 starts.  Processing trace1163.
[  3] on    astrocyte.cl.cam.ac.uk:    3 jobs,    0 cmpl    0 crsh    2 kill,    4 starts.  Processing trace1274.
[  4] on         bann.cl.cam.ac.uk:    0 jobs,    0 cmpl    0 crsh    0 kill,    1 starts.  CantStart.
[  5] on         bass.cl.cam.ac.uk:    3 jobs,    2 cmpl    0 crsh    0 kill,    1 starts.  Processing trace0434.
[  6] on       bigwig.cl.cam.ac.uk:    2 jobs,    1 cmpl    0 crsh    0 kill,    1 starts.  Processing trace0464.
[  7] on      cluseau.cl.cam.ac.uk:    2 jobs,    1 cmpl    0 crsh    0 kill,    1 starts.  Processing trace0482.
[  8] on       cortex.cl.cam.ac.uk:    8 jobs,    3 cmpl    0 crsh    5 kill,    5 starts.  WaitingForIdle.
[  9] on       cortex.cl.cam.ac.uk:    6 jobs,    1 cmpl    0 crsh    5 kill,    5 starts.  WaitingForIdle.
[ 10] on       cortex.cl.cam.ac.uk:    9 jobs,    4 cmpl    0 crsh    5 kill,    5 starts.  WaitingForIdle.
[ 11] on       cortex.cl.cam.ac.uk:    6 jobs,    1 cmpl    0 crsh    5 kill,    5 starts.  WaitingForIdle.
[ 12] on         cosi.cl.cam.ac.uk:    0 jobs,    0 cmpl    0 crsh    0 kill,    1 starts.  WaitingForIdle.
[ 13] on         erme.cl.cam.ac.uk:    2 jobs,    1 cmpl    0 crsh    0 kill,    1 starts.  Processing trace0415.

...

[ 63] on                     samo:    3 jobs,    2 cmpl    0 crsh    0 kill,    1 starts.  Processing trace1132.
[ 64] on                     toro:    2 jobs,    1 cmpl    0 crsh    0 kill,    1 starts.  Processing trace0439.
[ 65] on                     vere:    3 jobs,    2 cmpl    0 crsh    0 kill,    1 starts.  Processing trace1268.
[ 66] on                     yela:    0 jobs,    0 cmpl    0 crsh    0 kill,    1 starts.  WaitingForIdle.
[ 67] on                     ziba:    2 jobs,    1 cmpl    0 crsh    0 kill,    1 starts.  Processing trace0447.
Worklist status: 160 complete, 49 pending, 364 todo of 573
```

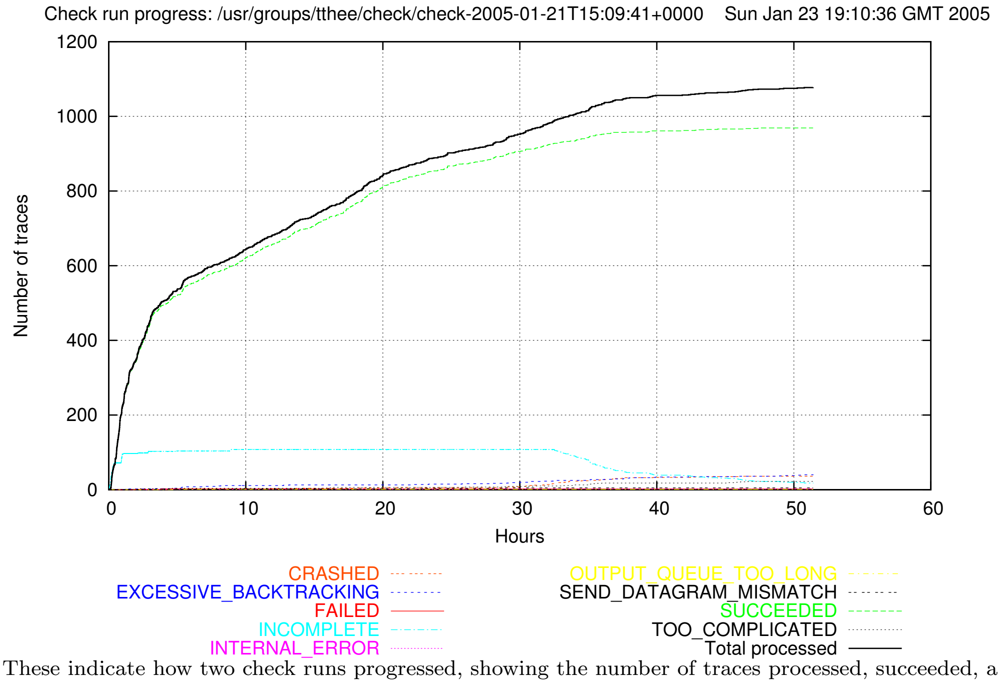
Figure 12: Checker monitoring — worker status



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.

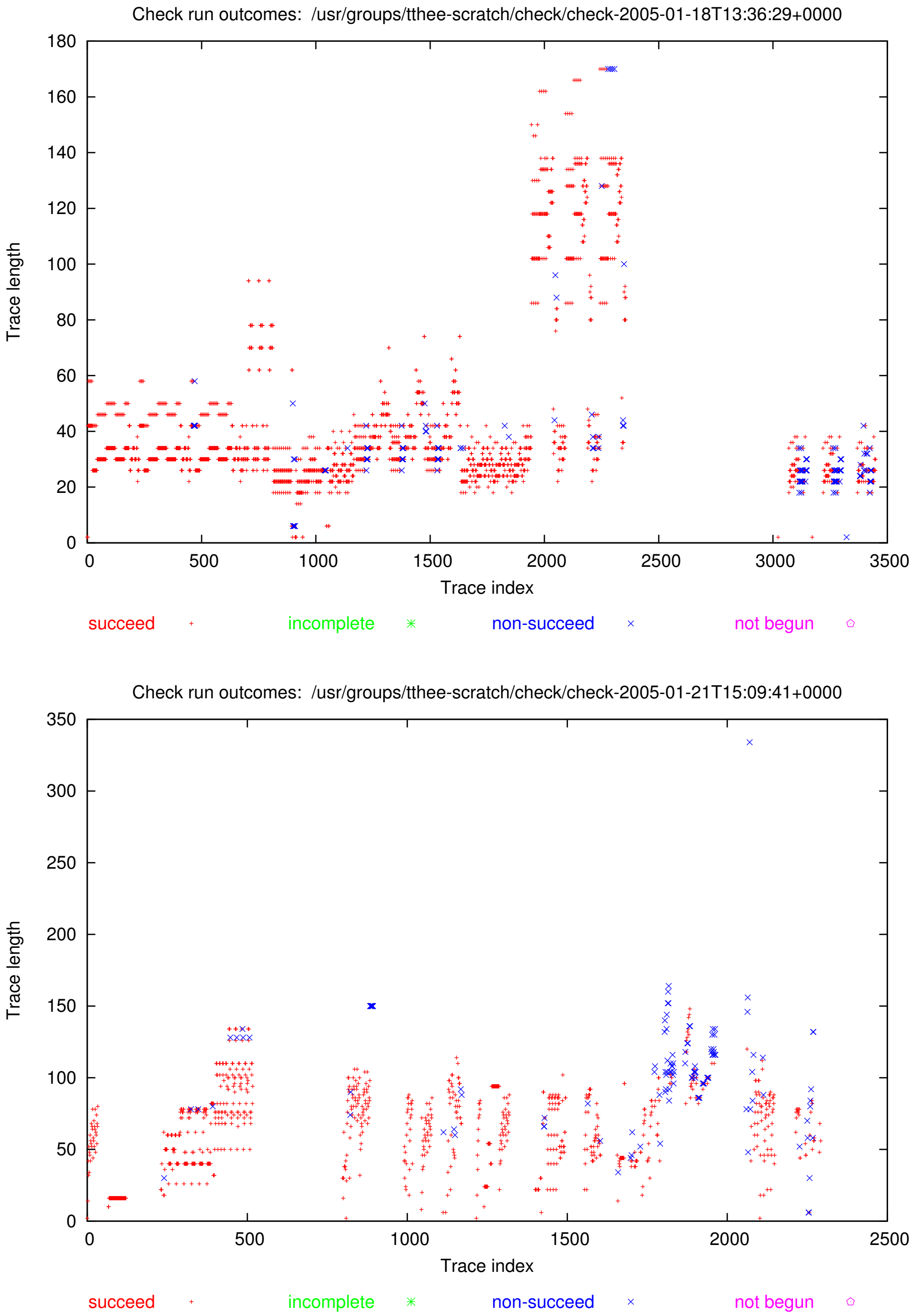
Figure 13: Checker monitoring: timed step graph.





These indicate how two check runs progressed, showing the number of traces processed, succeeded, and non-succeeded for various reasons. The first is a run from some time ago taking certain problematic traces, which had not succeeded before, first. The second is a more recent run on the entire BSD trace set. The INCOMPLETE lines indicate roughly how many worker machines were active.

Figure 14: Checker monitoring: progress of two TCP runs.



These indicate how two check runs progressed, one for UDP and one for TCP, showing the traces processed, succeeded, and non-succeeded for various reasons, indexed by the trace number and trace length.

Figure 15: Checker monitoring: progress of a UDP and a TCP run.

Initialising early versions of the evaluator took significant time (15 minutes on our fastest machines and up to an hour on the slowest). We amortise this time by processing multiple traces in a single evaluator instance, beginning a new one as each is complete. Only when the process dies or is killed do we need to reinitialise. This is managed by a simple front-end program, CheckTraces (written in MoSML/HOL), which performs initialisation, accepts the name of a trace to process on standard input, sets up the per-trace output file, parses the trace, passes the result to the evaluator, logs its success or failure, and repeats until end-of-file is detected on input.

The coordinator, otracechecker (written in OCaml), takes a specification of the worker machines and the traces to be checked, and assigns the traces to the workers, monitoring and logging their progress, restarting workers and retrying traces as necessary. Detailed log and status files are written; these are human-readable to aid debugging, but also machine-readable to support the various monitoring and visualisation tools described in Section 6.1 below. The algorithm is a standard worklist algorithm. Workers are started sequentially with a short delay to reduce load on the file server. If a worker crashes during startup, it is assumed to have a fault, and is not used again. If a worker crashes while processing a trace, it is restarted. If a worker is killed, it is restarted and the trace (if any) is retried later.

Provision is made for "in-use detection", to avoid using a machine when it is being used by its owner; this is important because even with our job set to the lowest possible scheduling priority, response times are sometimes affected. Two styles of control are offered: for machines with a well-defined owner the owner can write a simple configuration file specifying the time intervals (in cron format) that checking can operate in, whereas for lab machines we check whether any other users are active. For both, the in-use status is polled frequently, for responsiveness. If a machine becomes in-use while checking a trace, our job is suspended for later restart rather than killed. This became important as check times grew longer.

6.1 Visualisation and monitoring tools

The dataset of the results of the checker for many thousand traces is large and complex, and good visualisation tools are necessary for working with it. Our main tool is an HTML display of the results of a check run, of which a cut-down example is shown in Figure 11. This *HOL trace index* has a summary showing how many traces of the run have been processed and how many of those succeeded, failed the check, or terminated for some other reason. It then shows the detailed results for each trace processed. Each line has the trace number, the outcome, the time taken (both in seconds and normalised by machine speed), the step count reached and amount of backtracking, the machine used, and the trace description (mostly not visible in this excerpt). Hyperlinks take one to the output of the checker, as in Figure 9; colourised and raw version of the trace source, as in Figure 3; and the zig-zag diagram of the trace, as in Figure 5. Additional links let one generate a step graph for the trace, as in Figure 13, which shows how the (potentially backtracking) search of the checker proceeds. For a successful TCP trace one can view the zig-zag diagram annotated with the sequence of rule names in the discovered trace, as in Figures 7 and 8.

To help us manage the outstanding issues, for each check run we maintain a file of trace annotations, identifying subsets of the traces that have not succeeded for some particular reason and indicating whether that problem should have been resolved. The display shows the expected and actual number of successes for these.

Other important diagnostics include displays of the status of the worker machines, as in Figure 12, and graphs of the progress of the check run as a whole, as in Figure 14 The latter is especially useful to determine when best to abort an existing run in order to restart with an improved specification. Figure 15 shows the progress of two check runs indexed by the trace number and trace length, useful for seeing patterns of non-successes.

We also built an explicit regression tester, comparing the results of multiple check runs (which might be on overlapping but non-identical trace sets), but have not used it heavily — the annotation display is more useful, especially as we reach closer to 100% success.

6.2 Automated typesetting tool

HOL source is fairly readable in the small, but typesetting and clear large-scale structure are essential to make a large specification intelligible and printable, and manual approaches would be tedious and error-prone. We have therefore built an automated typesetting system that takes the HOL source and outputs LaTeX. The parts of the specification quoted in this document are taken automatically from this. The HOL source is used to determine the various different kinds of identifiers (types, constructors, auxiliary definitions, and quantified or lambda-bound variables), which are set in appropriate fonts. The tool does not do a full HOL parse, however, so identifiers used at more than one kind are occasionally set wrongly. Most of the tool is general, not tied to this specification: it has been used for other HOL work and for typesetting unrelated and non-HOL papers.

7 Validation — Current status

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 strings, exceeding UDPpayloadMax, 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 (getsockerr() not outputting the error correctly; WinXP setfileflags() not being correctly implemented; and dupfd() being implemented using `dup2()` rather than `fcntl()`); 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. Moreover, test generation was largely independent of the validation process (some additional tests were constructed during validation, and some particularly long traces were excluded). At present, of the 194 host LTS rules 142 are covered in at least one of the above successful trace check run; 32 should not be covered by the tests (most of these are rules dealing with resource limits, e.g. if there are no remaining file descriptors, or non-BSD TCP behaviour); and 20 either have not had tests written or not yet succeeded in validation.

For TCP 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.

The ICMP aspects of the specification have not been well tested. This is due to unfixed (but fixable) problems in the test-generation infrastructure.

7.1 Checker performance

Achieving satisfactory performance of the symbolic evaluator has been critical for this work to be feasible. To do so, we have made algorithmic improvements to HOL itself (e.g. in the treatment of multi-field records), to the evaluator (e.g. in better heuristics for search, and the lazy computation and urgency approximations mentioned in §5), and to the checking infrastructure, distributing over more machines and using them more efficiently.

For UDP the resulting performance is completely satisfactory: the UDP check run described above took approximately 5 hours.

For TCP the checker has a much more complex task. TCP host states are typically more symbolic, with more fields that are only loosely constrained and with larger sets of constraints. Also, longer traces are required to reach the various possible states of the system. Currently a complete run on the BSD traces takes around 50 hours. Before our recent improvements, TCP runs had some individual traces taking 500 000 – 1 000 000s (wall-clock) to validate, with a whole run around 500 hours. Multi-week check runs are awkward, making it hard to iterate and do regression testing on the whole set as often as one would like.

For future work the performance should be further improved. Performance analysis of HOL code is difficult (with high-level logic simplification tools and decision procedures being used), but it appears that much of the cost of current runs arises from simplifying timing constraints — many steps introduce new time parameters which are loosely constrained, reflecting the fact that we cannot specify the rates of the host's timers exactly, and in some traces simplifying these constraints appears to have a cost exponential in the trace length. It seems plausible that one will be able to agglomerate multiple such constraints more efficiently than we do at present.

To date, we have not used any tools outside of the core HOL system. Though the timing constraints mentioned above may no longer be an efficiency bottleneck, the work done by HOL's arithmetic decision procedures might still be farmed out to external tools. HOL can be made to accept the verdicts of such external "oracles", and the prospect of substituting an optimised C library for interpreted Moscow ML code is an appealing one. Solving arithmetic constraints is an obvious, and easily isolated sub-problem. Finding other well-defined problems that can be independently solved by external tools would clearly be valuable.

Another avenue to be explored is in shifting HOL's implementation from Moscow ML to the state-of-the-art MLton compiler. MLton is a whole-program compiler, and this does not sit well with HOL's representation of logical theories as ML structures. Though exasperating, this is not an intellectually significant obstacle, and a MLton implementation may be another route to significant improvements in efficiency.

8 The TCP state diagram

TCP is often presented using a 'TCP state diagram', giving an approximate view of the state of a TCP socket and how that changes with API calls and segments sent and received. The original RFC793 state diagram is reproduced in Figure 16 for reference; an improved diagram is given in the Stevens texts. The states in these diagrams are simply the *st* component of a TCP socket: CLOSED, LISTEN, ESTABLISHED, etc. This can be a useful abstraction, broadly explaining how the SYN, ACK, and FIN flags in TCP segments are used. It is, however, only a tiny part of the complete socket state that the protocol behaviour depends on (c.f. the model types socket, tcp_socket, and tcpcb in §4.5, which define the entire socket state). Moreover, the RFC793 and Stevens diagrams give only some of the more common transitions.

For comparison, and to show which rules in the specification deal with what behaviour, we have drawn a state diagram based on the specification. It is given in two versions in Figures 16 and 17. A larger view of the former is available on the Netsem project web page [Net].

The states are the classic 'TCP states' with the addition of a NONEXIST state for a nonexistent socket. The traditional diagrams have transitions involving two different sockets, e.g. from LISTEN to SYN_RECEIVED where a SYN_RECEIVED socket is created in response to a SYN received by a LISTEN socket. The model diagrams refer strictly to the state of a single socket.

The transitions are a conservative over-approximation to the set of all the transitions in the model which either (1) affect the 'TCP state' of a socket, or (2) involve processing TCP segments from the host's input queue or adding them to its output queue, except that transitions involving ICMPs are omitted, as are transitions modelling the pathological BSD behaviour in which arbitrary sockets can be moved to LISTEN states.

Transitions are labelled by their Host LTS rule name (e.g. *socket_1*, *deliver_in_3*, etc.), any socket call involved (e.g. close()), and constraints on the ACK/RST/SYN/FIN flags of any TCP segment received and sent, with a/r/s/f indicating the flag is clear and A/R/S/F indicating it is set. For segments sent by *deliver_in_3*, the flag constraints depends on the rest of the state in a complex way, so these are simply marked di3out. Transitions involving segments (either inbound or outbound) with RST set are coloured orange; others that have SYN set are coloured green; others that have FIN set are coloured blue; others are coloured black. The FIN indication includes the case of FINs that are constructed by reassembly rather than appearing in a literal segment.

It would be desirable to have a precise statement of the relationship between transitions in the specification and in the diagram — at least stated, if not proved, in HOL. Such a statement would be nontrivial, both for this FIN subtlety and because the segments in the diagram strictly refer to segments enqueued and dequeued from the host's queues rather than those that appear on the network interface.

The diagrams are based on data extracted by a manual abstract interpretation of the HOL specification. The data does not capture all the invariants of the model, so some depicted transitions may not be reachable in the model (or in practice). Similarly, the constraints on flags shown may be overly weak.

```
                                          Transmission Control Protocol
                                               Functional Specification

                             +---------+ ---------\      active OPEN
                             |  CLOSED |            \    -----------
                             +---------+<---------\   \   create TCB
                               |     ^              \   \  snd SYN
                  passive OPEN |     |   CLOSE        \   \
                  ------------ |     | ----------       \   \
                   create TCB  |     | delete TCB         \   \
                               V     |                      \   \
                             +---------+            CLOSE    |    \
                             |  LISTEN |          ---------- |     |
                             +---------+          delete TCB |     |
                  rcv SYN      |     |     SEND              |     |
                 -----------   |     |    -------            |     V
+---------+      snd SYN,ACK  /       \   snd SYN          +---------+
|         |<-----------------           ------------------>|         |
|   SYN   |                    rcv SYN                     |   SYN   |
|   RCVD  |<-----------------------------------------------|   SENT  |
|         |                    snd ACK                     |         |
|         |------------------           -------------------|         |
+---------+   rcv ACK of SYN  \       /  rcv SYN,ACK       +---------+
  |           --------------   |     |   -----------
  |                  x         |     |     snd ACK
  |                            V     V
  |  CLOSE                   +---------+
  | -------                  |  ESTAB  |
  | snd FIN                  +---------+
  |                   CLOSE    |     |    rcv FIN
  V                  -------   |     |    -------
+---------+          snd FIN  /       \   snd ACK          +---------+
|  FIN    |<-----------------           ------------------>|  CLOSE  |
| WAIT-1  |------------------                              |   WAIT  |
+---------+          rcv FIN  \                            +---------+
  | rcv ACK of FIN   -------   |                            CLOSE  |
  | --------------   snd ACK   |                           ------- |
  V        x                   V                           snd FIN V
+---------+                  +---------+                   +---------+
|FINWAIT-2|                  | CLOSING |                   | LAST-ACK|
+---------+                  +---------+                   +---------+
  |                rcv ACK of FIN |                 rcv ACK of FIN |
  |  rcv FIN       -------------- |    Timeout=2MSL -------------- |
  |  -------              x       V    ------------        x       V
   \ snd ACK                 +---------+delete TCB         +---------+
    ------------------------>|TIME WAIT|------------------>| CLOSED  |
                             +---------+                   +---------+

                     TCP Connection State Diagram
                              Figure 6.               September 1981
```

Figure 16: The RFC793 TCP state diagram

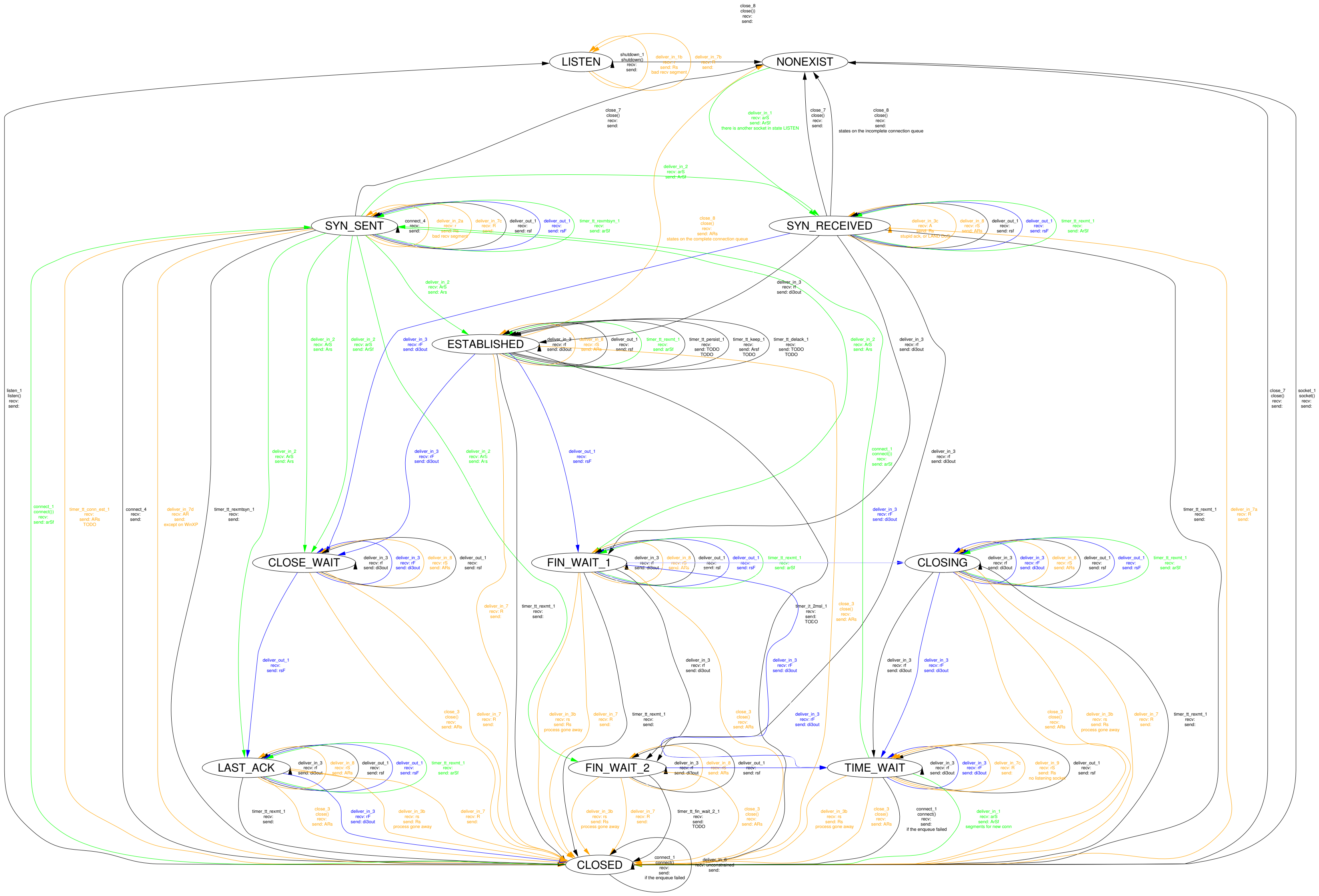

close_8 close()) recv: send:
LISTEN
shutdown_1 shutdown() recv: send:
deliver_in_1b recv: r send: Rs bad recv segment
deliver_in_7b recv: R send:
NONEXIST
close_7 close() recv: send:
deliver_in_1 recv: arS send: ArSf there is another socket in state LISTEN
close_7 close() recv: send:
close_8 close() recv: send: states on the incomplete connection queue
deliver_in_2 recv: arS send: ArSf
SYN_SENT
connect_4 recv: send:
deliver_in_2a recv: r send: Rs bad recv segment
deliver_in_7c recv: R send:
deliver_out_1 recv: send: rsf
deliver_out_1 recv: send: rsF
timer_tt_rexmtsyn_1 recv: send: arSf
close_8 close() recv: send: ARs states on the complete connection queue
SYN_RECEIVED
deliver_in_3c recv: A send: Rs stupid ack, or LAND DoS
deliver_in_8 recv: rS send: ARs
deliver_out_1 recv: send: rsf
deliver_out_1 recv: send: rsF
timer_tt_rexmt_1 recv: send: ArSf
deliver_in_2 recv: ArS send: Ars
deliver_in_3 recv: rf send: di3out
ESTABLISHED
deliver_in_3 recv: rf send: di3out
deliver_in_8 recv: rS send: ARs
deliver_out_1 recv: send: rsf
timer_tt_rexmt_1 recv: send: arSf
timer_tt_persist_1 recv: send: TODO TODO
timer_tt_keep_1 recv: send: Arsf TODO
timer_tt_delack_1 recv: send: TODO TODO
deliver_in_2 recv: ArS send: Ars
deliver_in_3 recv: rf send: di3out
deliver_in_2 recv: ArS send: Ars
deliver_in_2 recv: arS send: ArSf
deliver_in_3 recv: rF send: di3out
listen_1 listen() recv: send:
close_7 close() recv: send:
socket_1 socket() recv: send:
deliver_in_2 recv: ArS send: Ars
deliver_in_3 recv: rF send: di3out
deliver_in_2 recv: ArS send: Ars
deliver_out_1 recv: send: rsF
connect_1 connect()) recv: send: arSf
deliver_in_3 recv: rf send: di3out
connect_1 connect()) recv: send: arSf
timer_tt_conn_est_1 recv: send: ARs TODO
connect_4 recv: send:
deliver_in_7d recv: AR send: except on WinXP
timer_tt_rexmtsyn_1 recv: send:
deliver_in_3 recv: rF send: di3out
timer_tt_rexmt_1 recv: send:
deliver_in_7a recv: R send:
CLOSE_WAIT
deliver_in_3 recv: rf send: di3out
deliver_in_3 recv: rF send: di3out
deliver_in_8 recv: rS send: ARs
deliver_out_1 recv: send: rsf
FIN_WAIT_1
deliver_in_3 recv: rf send: di3out
deliver_in_8 recv: rS send: ARs
deliver_out_1 recv: send: rsf
deliver_out_1 recv: send: rsF
timer_tt_rexmt_1 recv: send: arSf
CLOSING
deliver_in_3 recv: rf send: di3out
deliver_in_3 recv: rF send: di3out
deliver_in_8 recv: rS send: ARs
deliver_out_1 recv: send: rsf
deliver_out_1 recv: send: rsF
timer_tt_rexmt_1 recv: send: arSf
deliver_in_7 recv: R send:
timer_tt_rexmt_1 recv: send:
timer_tt_2msl_1 recv: send: TODO
close_3 close() recv: send: ARs
deliver_out_1 recv: send: rsF
deliver_in_3 recv: rf send: di3out
deliver_in_3 recv: rF send: di3out
deliver_in_3 recv: rf send: di3out
deliver_in_3 recv: rF send: di3out
close_3 close() recv: send: ARs
deliver_in_7 recv: R send:
deliver_in_3b recv: rs send: Rs process gone away
deliver_in_7 recv: R send:
timer_tt_rexmt_1 recv: send:
close_3 close() recv: send: ARs
deliver_in_3 recv: rF send: di3out
close_3 close() recv: send: ARs
deliver_in_3b recv: rs send: Rs process gone away
deliver_in_7 recv: R send:
timer_tt_rexmt_1 recv: send:
LAST_ACK
deliver_in_3 recv: rf send: di3out
deliver_in_8 recv: rS send: ARs
deliver_out_1 recv: send: rsf
deliver_out_1 recv: send: rsF
timer_tt_rexmt_1 recv: send: arSf
FIN_WAIT_2
deliver_in_3 recv: rf send: di3out
deliver_in_8 recv: rS send: ARs
deliver_out_1 recv: send: rsf
TIME_WAIT
deliver_in_3 recv: rf send: di3out
deliver_in_3 recv: rF send: di3out
deliver_in_7c recv: R send:
deliver_in_9 recv: rS send: Rs no listening socket
deliver_out_1 recv: send: rsf
timer_tt_rexmt_1 recv: send:
close_3 close() recv: send: ARs
deliver_in_3 recv: rF send: di3out
deliver_in_3b recv: rs send: Rs process gone away
deliver_in_7 recv: R send:
deliver_in_3b recv: rs send: Rs process gone away
deliver_in_7 recv: R send:
timer_tt_fin_wait_2_1 recv: send: TODO
close_3 close() recv: send: ARs
deliver_in_3b recv: rs send: Rs process gone away
close_3 close() recv: send: ARs
connect_1 connect() recv: send: if the enqueue failed
deliver_in_1 recv: arS send: ArSf segments for new conn
CLOSED
connect_1 connect() recv: send: if the enqueue failed
deliver_in_6 recv: unconstrained send:

Figure 17: The TCP state diagram for the specification.

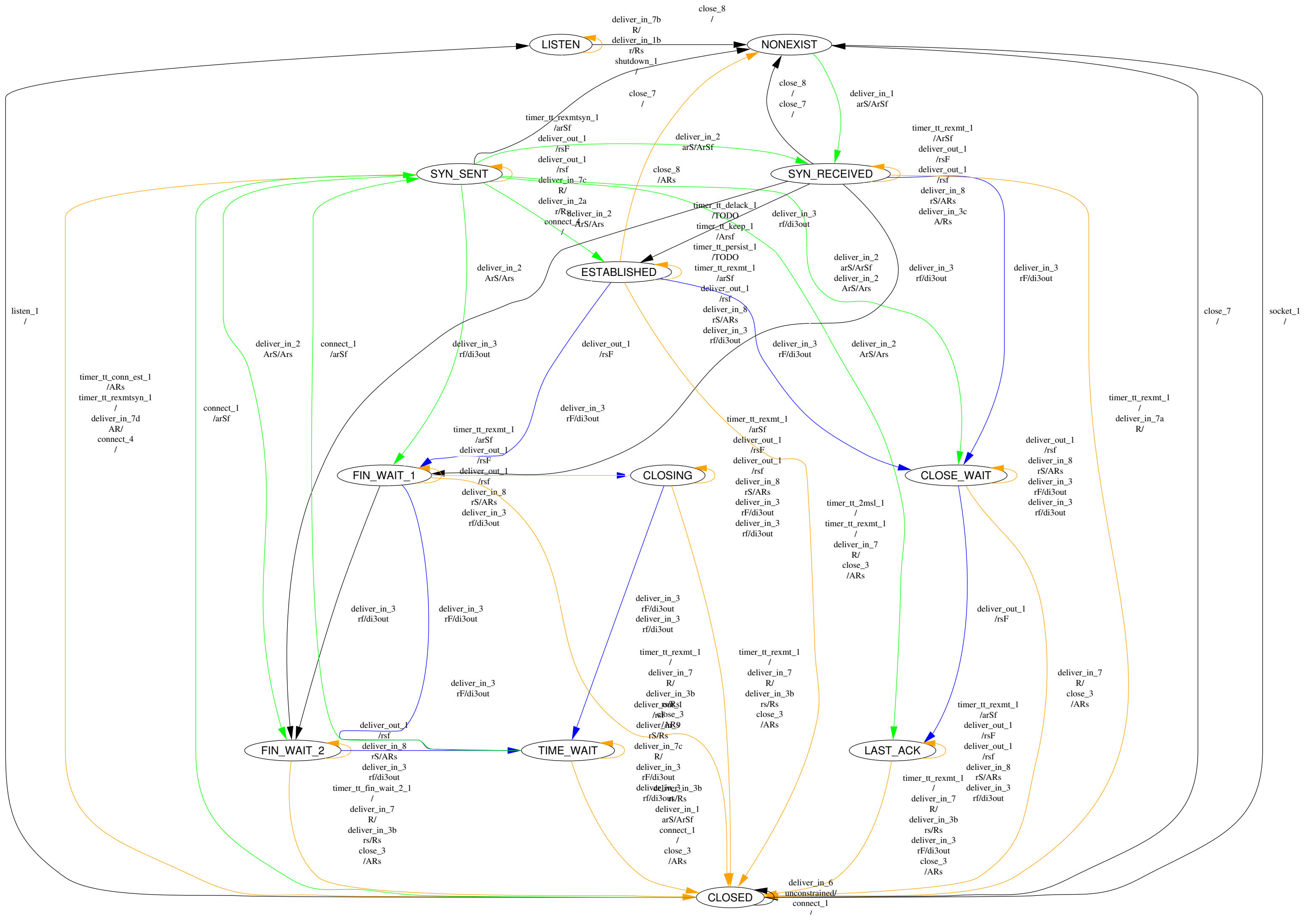

LISTEN
NONEXIST
SYN_SENT
SYN_RECEIVED
ESTABLISHED
FIN_WAIT_1
CLOSING
CLOSE_WAIT
FIN_WAIT_2
TIME_WAIT
LAST_ACK
CLOSED
close_8 /
deliver_in_7b R/ deliver_in_1b r/Rs shutdown_1 /
close_7 /
close_8 / close_7 /
deliver_in_1 arS/ArSf
timer_tt_rexmtsyn_1 /arSf deliver_out_1 /rsF deliver_out_1 /rsf deliver_in_7c R/ deliver_in_2a r/Rs connect_4 /
deliver_in_2 arS/ArSf
close_8 /ARs
deliver_in_2 ArS/Ars
timer_tt_rexmt_1 /ArSf deliver_out_1 /rsF deliver_out_1 /rsf deliver_in_8 rS/ARs deliver_in_3c A/Rs
timer_tt_delack_1 /TODO timer_tt_keep_1 /Arsf timer_tt_persist_1 /TODO timer_tt_rexmt_1 /arSf deliver_out_1 /rsf deliver_in_8 rS/ARs deliver_in_3 rf/di3out
deliver_in_3 rf/di3out
deliver_in_2 ArS/Ars
deliver_in_2 arS/ArSf deliver_in_2 ArS/Ars
deliver_in_3 rf/di3out
deliver_in_3 rF/di3out
listen_1 /
timer_tt_conn_est_1 /ARs timer_tt_rexmtsyn_1 / deliver_in_7d AR/ connect_4 /
connect_1 /arSf
deliver_in_2 ArS/Ars
connect_1 /arSf
deliver_in_3 rf/di3out
deliver_out_1 /rsF
deliver_in_3 rF/di3out
deliver_in_3 rF/di3out
deliver_in_2 ArS/Ars
timer_tt_rexmt_1 / deliver_in_7a R/
close_7 /
socket_1 /
timer_tt_rexmt_1 /arSf deliver_out_1 /rsF deliver_out_1 /rsf deliver_in_8 rS/ARs deliver_in_3 rf/di3out
timer_tt_rexmt_1 /arSf deliver_out_1 /rsF deliver_out_1 /rsf deliver_in_8 rS/ARs deliver_in_3 rF/di3out deliver_in_3 rf/di3out
timer_tt_2msl_1 / timer_tt_rexmt_1 / deliver_in_7 R/ close_3 /ARs
deliver_out_1 /rsf deliver_in_8 rS/ARs deliver_in_3 rF/di3out deliver_in_3 rf/di3out
deliver_in_3 rf/di3out
deliver_in_3 rF/di3out
deliver_in_3 rF/di3out
deliver_in_3 rF/di3out deliver_in_3 rf/di3out
timer_tt_rexmt_1 / deliver_in_7 R/ deliver_in_3b rs/Rs close_3 /ARs
timer_tt_rexmt_1 / deliver_in_7 R/ deliver_in_3b rs/Rs close_3 /ARs
deliver_out_1 /rsF
deliver_in_7 R/ close_3 /ARs
deliver_out_1 /rsf deliver_in_8 rS/ARs deliver_in_3 rf/di3out timer_tt_fin_wait_2_1 / deliver_in_7 R/ deliver_in_3b rs/Rs close_3 /ARs
deliver_out_1 /rsf deliver_in_9 rS/Rs deliver_in_7c R/ deliver_in_3 rF/di3out deliver_in_3 rf/di3out deliver_in_3b rs/Rs deliver_in_1 arS/ArSf connect_1 / close_3 /ARs
timer_tt_rexmt_1 /arSf deliver_out_1 /rsF deliver_out_1 /rsf deliver_in_8 rS/ARs deliver_in_3 rf/di3out
timer_tt_rexmt_1 / deliver_in_7 R/ deliver_in_3b rs/Rs deliver_in_3 rF/di3out close_3 /ARs
deliver_in_6 unconstrained/ connect_1 /

Figure 18: The TCP state diagram for the specification, with parallel transitions collapsed.

Several other components of a TCP socket state are as important as the 'TCP state' in determining its behaviour: especially the retransmit mode (None, RexmtSyn, Rexmt, or Persist), but also *cantrcvmore*, *cantsndmore*, *tf_needfin*, $sndq \neq []$, etc. Obviously simply taking the product of these would yield an undrawable diagram, but reclustering (slicing) in a different way might be useful. For example, most of the TCP code is independent of which state in {ESTABLISHED, CLOSE_WAIT, LAST_ACK, FIN_WAIT_1, CLOSING, FIN_WAIT_2, TIME_WAIT} is current; instead, the retransmit mode is of much more interest. It is possible that coalescing this class, and then taking the product with the retransmit state, would yield a manageable set of nodes.

The Figure 17 diagram is based on the same data but has parallel transitions (with the same source and target 'TCP state') collapsed into one. The diagrams are automatically drawn using the `graphviz` package [Gra].

9 Implementation anomalies

The goal of this project was not to find bugs in the implementations. Indeed, from our *post-hoc* specification point of view, there is no such thing as a bug — however strange the implementation behaviour, it 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 reflect whatever that behaviour is. The implementations are also extremely widely used. It would be very surprising to find serious problems in the common-case paths. Nonetheless, in the course of the work we have found many oddities in their behaviour. In this section we describe some of the most significant. These are broadly classified as to whether they primarily concern the API or the wire protocol, and whether they are clearly bugs in the conventional sense (points which implementation groups might consider fixing) or issues subject to debate. These are all relatively local issues, not including anything that would require substantial redesign of either protocols or API. We do not claim they are previously unknown, but they were certainly not obvious to us from the existing documentation.

There are also many other differences between the implementations included in the specification. A simple line count shows around 260 lines of the specification with an explicit test of the host OS version.

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.

By describing these oddities we hope also 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.

There is, of course, a danger that our specification may be *too* accurate: in the process of ensuring that implementation oddities are covered we may over-specify, potentially leading programmers to depend on pathological behaviour which may well change in future implementations. It may even be arguable that the extreme looseness of the early RFC textual specifications was beneficial, allowing implementations to evolve, though now the disadvantages (the differing implementations and the lack of clarity) have become evident. This only highlights the need for specifications —especially those for future protocols— to be appropriately loose, prescribing neither too much nor too little about the intended behaviour.

UDP

1. (API issue) send() pending errors for UDP on WinXP Recall that UDP sockets may have their pending error flag set by incoming ICMP datagrams, e.g. ICMP_UNREACH PORT, asynchronously with any Sockets API calls. This is reported differently on the various platforms.

- On BSD and Linux a subsequent send() or recv() call will immediately return the pending error, whereas on WinXP a subsequent send() call will succeed, emitting a datagram. See rules *send_9* and *send_23*.
- On BSD and Linux the error flag is set as soon as an incoming ICMP datagram is processed, whereas on WinXP an error marker is queued on the sockets receive queue and so will be observable only when preceeding datagrams have been returned by recv(). See rule *deliver_in_icmp_1*.

2. (API/Protocol bug) Calling connect() with a wildcard port on Linux On Linux, a call to connect() with no specified port will return successfully. Subsequently calling getpeername() returns ENOTCONN. However, if a call to send() is made, this returns successfully, sending out a datagram with no destination port set. Nothing can be done with this packet, and it is inconsistent for getpeername() to return ENOTCONN but for the send() call not to, when both calls see the socket in this same state. See rules *connect_7*, *connect_8*, *getpeername_2*, *send_9*. IMPACT: This only affects applications which use the Sockets API in a nonstandard or erroneous way. Such a call is most likely an application error, which here will not be detected as early as it might be. Moreover —perhaps contrary to expectation— there can be non-forged datagrams on the wire with destination port 0.

3. (API issue) Pending error return for a connect() call on BSD If a UDP socket has a pending error, then calling connect() on it will cause the call to fail with that error. However, under BSD, the socket's peer address will still be set to the values specified by the call. See *connect_10*.

4. (getpeername() returns incorrectly on Linux) API bug The peer address of a UDP socket may be set by calling connect() and specifying the remote IP address but not the port. It will then only accept an incoming datagram with the given IP address as its source. However, calling getpeername() on such a socket fails with ENOTCONN because there is no port set. See rules *getpeername_1*, *getpeername_2*.

TCP

Unless otherwise stated these refer to the BSD behaviour.

5. (API Bug) listen() may be called from any state Under WinXP and Linux, listen() may only be called from either the CLOSED or LISTEN states. BSD, however, fails to enforce this constraint. If a connected socket enters the LISTEN state, then it retains its full quad (as the BSD listen() call essentially does nothing but change the state of the socket to LISTEN), thus only enabling it to accept connections from the same remote IP/port. An accept() call may occur in the usual way if this is the case. Note that despite having a full quad and the *SS_ISCONNECTED* flag set, the socket cannot send any data, since a call to send() causes BSD to check its actual state (and tries unsuccessfully to call connect()).

Due to this, it is possible for a BSD socket in the LISTEN state to have the retransmit timer set. When the timer fires, a 'phantom segment' will be emitted, with either no flags set at all, or only the FIN flag (if we were retransmitting a FIN). If FIN is set, these will continue as usual for a retransmission. If FIN is clear, only one will be emitted.

Further, calling getpeername() on a post-established socket for which listen() has been called may return incorrectly, giving the peer IP address and port of the previous connection. See rules *listen_∗*, *timer_tt_rexmt_1*, *getpeername_∗*. IMPACT: This only affects applications which use the Sockets API in a nonstandard or erroneous way. Such a call is most likely an application error, which here will not be detected as early as it might be. Moreover, there can be non-forged segments on the wire with no flags or only the FIN flag set.

6. (Protocol issue) Calling close() from SYN_RECEIVED drops the socket According to the standard TCP state transition diagram, calling close() on a socket in the SYN_RECEIVED state should result in a *FIN* being sent, and a direct transition to the FIN_WAIT_1 state. BSD, however, simply drops the socket in this situation, sending no indication of its doing so to the remote host. Both behaviours are arguably desirable. See *close_7*.

7. (Protocol issue) State changes when calling shutdown() from a pre-established state

If shutdown() is called on a socket in SYN_RECEIVED, BSD emits a *FIN* and remains in the same state (setting `TF_NEEDFIN`). If it then receives an *ACK* that acknowledges its *SYN* but not its *FIN*, BSD incorrectly changes the state to FIN_WAIT_2 rather than FIN_WAIT_1 as would be expected. See *shutdown_∗*, *deliver_in_∗*.

8. (Protocol bug) Response to *SYN*,*FIN* segments. In the SYN_SENT state, it is possible to receive a *FIN* along with the required *SYN*. In the case of a *SYN*,*FIN*,*ACK* being received, BSD will *ACK* both the *SYN* and the *FIN*, moving into CLOSE_WAIT, which is perfectly reasonable behaviour. If, however, a *SYN*,*FIN* segment is received (a simultaneous open), BSD incorrectly bypasses the SYN_RECEIVED state and moves directly into CLOSE_WAIT without waiting for our *SYN* to be acknowledged. See *deliver_in_2*, *deliver_in_3*.

9. (API bug) Calling send() with MSG_WAITALL or MSG_PEEK set In the context of a send() call, MSG_WAITALL and MSG_PEEK are invalid options. BSD ignores these options, rather than failing with an EOPNOTSUPP error. See *send_8*. IMPACT: This only affects applications which use the Sockets API in a nonstandard or erroneous way. Such a call is most likely an application error, which here will not be detected as early as it might be.

10. (Protocol bug) Window of no RTT cache updates After 2^{32} packets, there is a 16 packet window during which time, if the TCP connection is closed, the RTT values will not be cached in the routing table entry. This is because of an overflow/wraparound problem in `t_rttupdated`. Impact: Very rarely, after the closure of 1 in 2^{28} connections, the round-trip time estimator will be less accurate that it might be, adversely affecting the performance of a subsequent connection.

11. (Protocol bug) Incorrect updates of RTT estimates after repeated retransmission timeouts In accordance with RFC2988, the RTO value used is computed from the estimators as $\mathit{rto} = \max(1, \mathit{srtt} + 4\mathit{rttvar})$ (where *rttvar* is never allowed to drop to zero). Here *srtt* is stored in `t_srtt` in fixed point format with 5 fractional bits, and *rttvar* is stored in `t_rttvar` with 4 fractional bits.

When the retransmit timer expires the 4th time ($\frac{\mathit{max\_retransmissions}}{4} + 1$), the socket's RTT estimate is invalidated; this is signalled by zeroing `t_srtt`. However, in the absence of any other information, the used value of *rto* should remain identical. One way of achieving this would be the following:

$$\begin{aligned} \mathit{srtt}' &= 0 \\ \mathit{rttvar}' &= \mathit{rttvar} + \mathit{srtt}/4 \end{aligned}$$

(note that $\mathit{srtt}' + 4\mathit{rttvar}' = 4(\mathit{rttvar} + \mathit{srtt}/4) = \mathit{srtt} + 4\mathit{rttvar}$ as required).

Instead, BSD does the following:

$$\begin{aligned} \mathit{srtt}' &= 0 \\ \mathit{rttvar}' &= \mathit{rttvar} + \mathit{srtt}/2 \end{aligned}$$

which yields a new value of $\mathit{rto}' = 2\text{srtt} + 4\mathit{rttvar} > \mathit{rto}$.

This should be visible since the retransmit times for BSD are supposed to be in the ratio 1 2 4 8 16 32 64 64 etc, and this will indeed be seen if the actual RTT is less than 1s. However, what we see in practice is (assuming negligible variance in comparison to the RTT):

RTT / s	**Retransmission Times / s**								
1	1	2	4	8	16	32	64	64	...
2	2	4	8	16	16	32	64	64	...
4	4	8	16	32	16	32	64	64	...
8	8	16	32	64	32	64	128	128	...

Note that this is not visible for *SYN* retransmits (only for ordinary retransmits), since the value of `t_srtt` is zero already. The above figures depend on what the variance is; if the variance is large, then the effect may not be very noticeable.

In our model, this behaviour is incorporated into the timeout of the retransmission timer by setting BSD_RTTVAR_BUG to **T**. See *timer_tt_rexmtsyn_1*, *timer_tt_rexmt_1*.

12. (Protocol bug) States in which we have received a *FIN* In the BSD code, the macro `TCPS_HAVERCVDFIN(s)` is defined as:

```
#define TCPS_HAVERCVDFIN(s) ((s) >= TCPS_TIME_WAIT)
```

Clearly, this set of states should also include CLOSE_WAIT, LAST_ACK and CLOSING, since we must have received a *FIN* segment to enter such a state.

This macro is used three times in the code (in `tcp_input.c`), preventing the following from happening if we believe we have received a *FIN*:

1. Processing of urgent data (i.e. from segments with the *URG* flag set).

2. Processing normal data data, and arranging to *ACK* it.

3. Processing a *FIN* segment and performing the appropriate state changes.

See *deliver_in_**. Impact: A consequence of the first of the above is that it is possible (with suitably crafted segments) to generate a `SIGURG` signal from a socket after its connection has been closed. Data may also be received by a closing socket. Similarly, extra *FIN*s will be processed, causing an *ACK* to be emitted and an increment of the sequence number (of course this will only happen if the peer's TCP stack is broken, or malicious).

13. (Protocol bug) Timestamp updates not performed when delaying *ACK*s The following code appears in `tcp_input.c`:

```
if ((to.to_flags & TOF_TS) != 0 &&
    SEQ_LEQ(th->th_seq, tp->last_ack_sent)) {
       tp->ts_recent_age = ticks;
       tp->ts_recent = to.to_tsval;
}
```

The intention is to record the timestamp of the segment if the last *ACK* sent lies within its sequence numbers. However, `th->th_seq` has already been advanced by the left-end trimming code (i.e. trimming any data that we have already received from the start of the segment), thus it is always the case that:

$$\texttt{th->th\_seq} \geq \texttt{tp->rcv\_nxt} \geq \texttt{tp->last\_ack\_sent}.$$

This means that the condition can only be true when `th->th_seq = tp->last_ack_sent`; i.e., when we receive an inorder segment, and the previous segment was *ACK*ed without delay. The consequence is that the timestamp does not get updated in the socket's state if we delayed the last *ACK*. *deliver_in_3*

14. (Protocol bug) Update of the `TF_RXWIN0SENT` socket flag The `TF_RXWIN0SENT` flag in the socket's control block indicates whether a window of zero has been sent to the peer, and therefore that we have closed their send window. This is used to disable the sending of delayed *ACK*s to the receiver, so that it will receive a new window update as soon as possible. This flag is set, however, if the *calculated* receive window, rather than the window in the sent segment, is zero. These may differ if the window scaling, `rcv_scale`, is non-zero, in that a small calculated window may be truncated to zero; transmitted as a literal zero. See *deliver_out_1*. Impact: The re-opening of closed windows will be delayed.

15. (Protocol bug) Initialisation of the retransmit timer When the initial retransmit timer (`t_rxtcur`) is set from the RTT statistics in the route metric cache (rmx), it is calculated incorrectly. If we compare the calculation done by the normal-path code for a received segment with that done by the rmx code (both in `tcp_input.c`), we see the following. The normal-path code sets the retransmit timer, since the values are scaled, as:

$$\texttt{t\_rxtcur} = \tfrac{1}{32}\texttt{t\_srtt} + \tfrac{1}{4}\texttt{t\_rttvar} = \texttt{SRTT} + 4\times\texttt{RTTVAR}$$

However, the rmx code calculates it as:

$$\texttt{t\_rxtcur} = \tfrac{1}{8}\texttt{t\_srtt} + \tfrac{1}{2}\texttt{t\_rttvar} = 4\times\texttt{SRTT} + 8\times\texttt{RTTVAR}$$

The cause of the discrepancy is revealed by the comments in the code, which disagree with what actually happens. Clearly, the scale factors were previously 8 and 4, rather than the current values of 32 and 16. Hence the rmx code would originally have computed $\texttt{SRTT} + 2\times\texttt{RTTVAR}$, which is still a little out, but not badly so. The rmx code should be changed to:

```
((tp->t_srtt >> (TCP_RTT_SHIFT - TCP_DELTA_SHIFT)) + tp->t_rttvar) >> TCP_DELTA_SHIFT
```

This would agree with the normal-path code, rather than using hard-coded constant shifts.

16. (Protocol bug) Simultaneous open responds with an *ACK* rather than *SYN*, *ACK* BSD incorrectly implements the diagram bug seen in RFC 793:

```
      TCP A                                                TCP B
1.    CLOSED                                               CLOSED
2.    SYN-SENT     -->  <SEQ=100><CTL=SYN>                 ...
3.    SYN-RECEIVED <--  <SEQ=300><CTL=SYN>                 <-- SYN-SENT
4.                 ...  <SEQ=100><CTL=SYN>                 --> SYN-RECEIVED
5.    SYN-RECEIVED -->  <SEQ=100><ACK=301><CTL=SYN,ACK>    ...
6.    ESTABLISHED  <--  <SEQ=300><ACK=101><CTL=SYN,ACK>    <-- SYN-RECEIVED
7.                 ...  <SEQ=101><ACK=301><CTL=ACK>        --> ESTABLISHED
```

Here, it should be the case that line 7 has the *SYN* flag set, as this is the same segment that was sent out in line 6. The BSD implementation sends an *ACK*-only segment in response to receiving a *SYN* in the SYN_SENT state. If the retransmit timer fires, however, then the correct *SYN*, *ACK* segment is sent. Note that under normal operation of a simultaneous open, the sent *ACK* will correctly cause the peer to become ESTABLISHED. However, it may have been the case that the initial *SYN* was lost, in which case the peer is in the SYN_SENT state and expecting a *SYN*, *ACK*. See *deliver_in_2*, *timer_tt_rexmtsyn_1*. Impact: The connection handshake will be delayed by one retransmit interval.

17. (Protocol bug) Sending options in a SYN, ACK that are not in the received SYN on Linux RFC 1323 describes the timestamp and window scale option in TCP. Importantly, it describes a change from the specification given in RFCs 1072 and 1185:

> The spec was modified so that the extended options will be sent on SYN,ACK segments only when they are received in the corresponding SYN segments. This provides the most conservative possible conditions for interoperation with implementations without the extensions.

Linux, however, does not comply with this, in that it sends option values that were not specified in the received SYN segment, in the case of a simultaneous open. More specifically, it retransmits the options in its initial sent SYN without taking into account the options specified in the SYN it just received. See *deliver_in_2*.

18. (API issue) Restriction of the remote address for incoming TCP connections The current Sockets API implementations do not permit remote address for incoming TCP connections to be restricted. We must call bind() before calling listen(), but we can't do the equivalent of a UDP-style connect(). This means that the full connection handshake must take place, and accept() must be called, before the server can find out the peer's address and make a decision to close the socket or not. Note that this behaviour is not a requirement of TCP, but a design decision of the sockets API. Under UDP, we can make a call to connect() specifying the IP and port of the peer, in order to restrict the future quad of the socket. An interesting point to note is that the BSD bug of allowing listen() to be called from any state, can essentially achieve the same effect; we perform a non-blocking connect() which returns with EINPROGRESS, then call listen() on the socket, to get it into a state whereby it can only accept incoming connections from the given remote address. Relying on an implementation bug in this way, however, is not advisable. See *listen_**, *connect_**. IMPACT: This is a well-known Sockets API limitation.

19. (Protocol issue) No window scaling for SYN, ACK segments RFC 1323 states that SYN segments do not get their windows scaled, even in the SYN, ACK segment emitted when a listening socket accepts a SYN. In this case however, we know both the correct scaling to use and that the remote end supports such scaling, but we are not allowed to actually scale the window. BSD agrees with the RFC in this respect, and we suspect that it actually just sends the low-order 16 bits of the true window. This seems strange; especially if the window happens to be a multiple of 2^{16}. See *deliver_in_1*.

20. (Protocol bug) Reduced retransmit SYN time on receipt of invalid RST on WinXP In the usual case, WinXP performs a SYN retransmit after 3s (as is the case with the other architectures), but on receipt of an invalid RST it waits only 350ms. See *timer_tt_rexmtsyn_1*.

21. (API issue) Inverted writeability semantics in pselect() On a socket that has been shutdown in the write direction, Linux reports non-writeable, whereas BSD and WinXP report writeable. POSIX supports the historical (BSD and WinXP) behaviour. It could be argued that the Linux semantics makes good sense, but the BSD/WinXP/POSIX semantics has the advantage of an unambiguous definition, namely that a socket is readable/writeable iff a blocking call to recv/send would return immediately (irrespective of whether it succeeds or fails). See *pselect_**, sowriteable.

22. (API issue) pselect() does not return readable/writeable for CLOSED socket Attempting to call recv() or send() on a socket in the CLOSED state returns immediately with an error on all OSes, but BSD and WinXP fail to report such sockets as readable and writeable in pselect(). This deviates from the POSIX specification, that a socket is readable/writeable iff a blocking call to recv/send would return immediately (irrespective of whether it succeeds or fails). Linux has the correct behaviour. See *pselect_**, soreadable, sowriteable.

23. (API issue) Treatment of bad file descriptors by pselect() If a call to pselect() is made, giving one or more invalid file descriptors, the POSIX specification required that the call fail with EBADF. This behaviour is seen correctly under Linux. BSD, however, successfully returns the call, selecting true for each of the ready to read, ready to write, and error conditions on the bad fd. It could be argued that this behaviour is valid, in that (for example) a call to recv() with O_NONBLOCK clear would not block, but fail immediately with the error EBADF. This seems to be the interpretation of the semantics that BSD has taken. However, this behaviour is not POSIX compliant, and the programmer does not gain anything due to this (other than possible confusion), since the EBADF failure is simply being postponed. See *pselect_**.

24. (Protocol bug) The receive window is updated on receipt of a bad segment When tcp_input() is called under BSD, it updates the receive window of the socket (rcv_wnd) before it processes the incoming segment. This means that, although the segment may end up being dropped (possibly with an RST) and therefore ignored by tcp_input() in other respects, the window update still occurs.

Initially, when the TCP control block is attached to the socket by tcp_attach(), the receive window rcv_wnd is initialised to the sysctl tcp_recvspace (which has a value of 57344 by default). Subsequent sent segments have this same window, until we receive a segment from the other end (i.e. tcp_input() is called). Note that the initial SYN of a passive open does not count, as this is handled by the BSD syncache.

From the first received segment onwards, rcv_wnd is set by tcp_input() to the maximum of the space in the receive buffer and the current window being advertised. Since the receive buffer is initially empty, for a socket in state SYN_SENT, rcv_wnd is set to the full receive buffer size, rounded to a multiple of the MSS. So, for Ethernet with timestamp options and a default value of tcp_recvspace, this is 57920.

Under most circumstances, this setting of the receive window will not cause it to change if the incoming segment is dropped, as it is calculated on the basis of the data currently in the receive buffer, and not in the segment that is currently being processed. However, for a socket in the SYN_SENT state, an effect *is* seen, as we are updating from the default value. Thus, for subsequent retransmitted SYN segments, a different window is advertised. Although the impact of this is minor, it is quite clear that it is incorrect for a dropped incoming segment to alter the state of the socket. See *deliver_in_2*.

25. (Protocol bug) Conditions under which tcp_mss() is called The function tcp_input() carries out some updates to the socket state before processing the incoming segment. If the segment is not bound for a listening socket (which is dealt with by the syncache), then the first action taken is to process the options on a SYN segment! This means that regardless of the rest of the data on the segment, if it contains the SYN flag then its options are processed. This includes the MSS advertisment, for which tcp_mss() is called to set the value of t_maxseg. This is of course clamped to the peer's offer at the time of connection establishment (with a minimum value of 64 minus the TCP options), however a rogue SYN segment could be seen, dropped, and sent an RST, and the socket's internal MSS value would still be updated on the basis of the advertisment seen. See *deliver_in_2*,*deliver_in_3*. IMPACT: This conceivably opens up the potential for an attack, whereby the IP and port of the remote end of the socket are spoofed, and a SYN segment with extremely low MSS offer is sent to the socket. This would cause t_maxseg to be set to the minimum value allowed, thus the size of subsequently sent segments would be restricted, and data would be highly fragmented. In the case of bulk data transfer, this would cause a proliferation of packets on the network, which could result in denial of service effects.

26. (Protocol bug) Conditions under which tcp_mss() is called, part 2 Another bug in the way in which BSD processes options on a SYN segment is that tcp_mss() is only called if an MSS option was seen. This is incorrect, as the function is designed to deal with this scenario, and assumes a default value. The effect of this is that the value of t_maxseg remains at the default of 512, without the size of the options being subtracted from it. Furthermore, since we rely on tcp_mss() to initialise snd_cwnd, in the case where no MSS option is seen, the congestion window remains at some very large initial value. See *deliver_in_2*,*deliver_in_3*.

27. (Protocol bug) *rcv_wnd* and *rcv_adv* updated differently by a SYN_SENT and SYN_RECEIVED socket The movement of passive open processing from tcp_input() to the syncache has caused some differences between the behaviour of SYN_SENT and SYN_RECEIVED sockets that was not previously there. One of these discrepancies relates to the update of the receive window. This is done in tcp_input(), but only after the case of a listening socket has been dealt with. On completion of a passive open, the syncache creates a new SYN_RECEIVED socket, which is passed back to tcp_input() for further processing. However, the value of rcv_adv is incorrectly updated by the syncache before rcv_wnd gets updated. The effect is that although rcv_wnd still expands to its full size (the size of the receive buffer, rounded to a multiple of t_maxseg), rcv_adv remains limited by its initial default value. Contrast this to the SYN_SENT behaviour, which updates rcv_adv after updating rcv_wnd. See *deliver_in_3*.

28. (Protocol issue) MTU used by tcp_mss() compared with tcp_mssopt() In the calculation of both the MSS to advertise, and the MSS to use internally (t_maxseg), TCP consults the MTU of the underlying interface. There is, however, a discrepancy between the calculation in tcp_mss() (which stores the internal, negotiated value), and tcp_mssopt() (which calculates the advertised value). In the former, we use the MTU stored in the routing table metric cache in preference to the actual interface

MTU, however the latter always uses the interface's MTU.

It clearly makes sense for us to advertise our actual MTU for a new connection, rather than a cached value, since the aim is to find the maximum possible MSS that satisfies both ends and the link. The problem arises in that the two MTU values may differ, so it is possible for us to internally enforce a small MSS, even when both ends have advertised a much larger value. Note that TCP does not cache the route MTU in `tcp_close()`, as it does with other metrics such as the round trip time. See calculate_buf_sizes.

29. (API issue) Return mode of connect() for non-blocking sockets Under the usual circumstances, a call to connect() on a non-blocking socket will fail with EINPROGRESS, rather than blocking until the socket becomes ESTABLISHED. An interesting situation, however, arises when the connection is made over the loopback interface. Under BSD, the call to connect() proceeds to emit the initial *SYN* segment. However, since this is being sent over loopback, it is received again almost immediately, and an interrupt is thrown, allowing the underlying layers and then TCP to process the segment.

In this way, the segment exchange occurs so fast that the socket has connected before the thread that called connect() regains control. When it does, it sees that the socket has been connected, and therefore returns with success rather than failing with EINPROGRESS. Note that since this behaviour is due to timing, it may also be possible for the connect() call to return before all the segments have been sent; for example if there was an artificially imposed delay on the loopback interface.

Linux does not exhibit this behaviour, and the connect() call fails with EINPROGRESS under this circumstance (though the socket does become ESTABLISHED before the call returns). The argument may be made in favour of either case, though it seems that the approach taken by Linux is more consistent with the semantics of a non-blocking socket. The BSD behaviour is not incorrect though, and is hinted at in the man page, which states that EINPROGRESS will be returned if "the socket is non-blocking and the connection cannot be completed immediately." See *connect_1*.

30. (Protocol bug) Path MTU plateau table Path MTU discovery makes use of a table of likely Internet path MTU "plateaux". BSD uses the table that appears in RFC1191 (November 1990); Linux uses that table with the addition of three X.25-related MTUs (576, 216, 128) and the deletion of SLIP's and ARPANET's 1006.

Discussion on `comp.protocols.tcp-ip`, Sun, 15 Feb 2004, `<102tjcifv6vgm02@corp.supernews.com>`, `kml@bayarea.net` (Kevin Lahey) suggests that this is out-of-date, and 2312 (WiFi 802.11), 9180 (common ATM), and 9000 (jumbo Ethernet) should be added. For some polemic discussion, see `http://www.psc.edu/~mathis/MTU/`. Indeed, RFC1191 itself says explicitly "We do not expect that the values in the table [...] are going to be valid forever. The values given here are an implementation suggestion, NOT a specification or requirement. Implementors should use up-to-date references to pick a set of plateaus [...]". BSD and Linux are therefore not compliant here. This table should be extended so as to be representative of the modern Internet. See mtu_tab, *deliver_in_icmp_2*. IMPACT: On certain routes the MTU will be incorrect, affecting performance.

31. (Protocol issue) Duplicate ACK detection ignores FIN In BSD at least, duplicate ACK detection ignores whether FIN is set or not. Further, the third (or later) duplicate ACK is dropped on the floor without further processing - in particular, without reaching FIN processing. This means that in a sequence of segments ACK, ACK, ACK, ACK+FIN, the FIN would be ignored, with the last segment treated as a duplicate like all the others, and triggering a retransmit. While the retransmit is (arguably) correct, not noticing the FIN is bad. See di3_ackstuff.

32. (Protocol bug) Received urgent pointer not updated in fast path The urgent pointer stored in the receiver is not updated in the fast path (header prediction succeeded) deliver-in code.

Normally, with each segment that is received, if the urgent flag is not set then the stored *rcv_up* is still pulled along with the left edge of the window. This ensures that later urgent-pointer comparison is not confused by the 2GB wraparound.

Omitting this in the fast path means that if 2GB of data is received in the fast path (i.e., always in order, always enough buffer space, no urgent flag set, etc.), *rcv_up* appears to be in the future. If now the urgent flag is set in an incoming segment, this will be ignored (since a later urgent pointer apparently exists). Eventually (after another 2GB of data, not necessarily on the fast path) the spurious urgent sequence number will be reached; however the byte will not be erroneously treated as urgent, since URG and urp of the segment need to be set for this to occur. Once this point is passed, behaviour returns to normal. See *deliver_in_3*. IMPACT: This situation is surely rare, but conceivable, in practice. Since the default is for OOB data to be received out-of-line, this means that a well-behaved, in-order connection (e.g., one with a fairly low data rate) with no urgent data for 2GB will forfeit the ability to signal urgent data for the subsequent 2GB.

33. (API issue) Network status estimators inaccessible TCP relies on estimates of the network status, with each endpoint maintaining round-trip-time estimates. This information could be of use to any applications that need to respond quickly to changes in connectivity or performance, but the Sockets API does not provide any means to access it.

10 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.

The most detailed rigorous specification of a TCP-like protocol we are aware of is that of Smith [Smi96], 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 [SR02].

Musuvathi and Engler have applied their CMC model-checker to a Linux TCP/IP stack [ME04]. Interestingly, they began by trying to work with just the TCP-specific part of the codebase (c.f. the pure transport-protocol specification figure of §2.1), but moved to working with the entire codebase on finding the TCP – IP interface too complex. The properties checked were of two kinds: resource leaks and invalid memory accesses, and protocol-specific properties. The latter were 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, with flow control, congestion control etc. not included. Four bugs in the Linux implementation were found.

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 [BOG02], proving correctness theorems. In constrast 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 naive line count) around 50 times smaller than the specification we present here.

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 [BCMG01]. They show that certain properties of TCP implementations can be expressed. Lee et al conduct passive testing of an OSPF implementation against an extended finite state machine model [LCH$^+$02].

There are I/O automata specifications and proof-based verification for aspects of the Ensemble group communication system by Hickey, Lynch, and van Renesse [HLvR99], and NuPRL proofs of fast-path optimizations for local Ensemble code by Kreitz [Kre04].

Alur and Wang address the PPP and DHCP protocols, for each checking refinements between models that are manually extracted from the RFC specification and from an implementation [AW01].

For radically idealised variants of TCP, one has for example the PVS verification of an improved Sliding Window protocol by Chkliaev et al [CHdV03], and Fersman and Jonsson's application of the SPIN model checker to a simplified version of the TCP establishment/teardown handshakes [FJ00]. 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 [Sch96]. The specification is again roughly at the level of the TCP state diagram. Billington and Han have produced a coloured Petri net model of the service provided by TCP (in our terminology, roughly an end-to-end specification), but for a highly idealised ISO-style interface, and a highly idealised model of transmission for a bounded-size medium [BH03, BH04]. Murphy and Shankar verify some safety properties of a 3-way handshake protocol analogous to that in TCP [MS87] and of a transport protocol based on this [MS88]. Finally, Postel's PhD thesis gave protocol models for TCP precursors in a modified Petri net style [Pos74].

Implementations of TCP in high-level languages have been written by Biagioni in Standard ML [Bia94], by Castelluccia et al in Esterel [CDO97], and by Kohler et al in Prolac [KKM99]. 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 nondeterminism means they could not be used as oracles for system testing. Hofmann and Lemmen report on testing of a protocol stack generated from an SDL specification of TCP/IP [HL00]. Few details of the specification are given, though it is said to be based on RFCs 793 and 1122. The focus is on performance improvement of the resulting code.

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 [Pax97], the TBIT of Padhye and Floyd [PF01], and Fyodor's `nmap` [Fyo]. RFC2398 [PS98] lists several other tools. There are also commercial products such as Ixia's Automated Network Validation Library (ANVL) [IXI05], with 160 test cases for core TCP, 48 for Slow Start, Congestion Control, etc., and 48 for High Performance and SACK extensions.

11 Project History

The initial spur for this project arose in mid-1998 during work on Nomadic Pict [SWP99, US01], a distributed programming language based on the π-calculus [MPW92] that was designed to allow distributed algorithms for mobile computation to be expressed as sharply as possible. This highlighted the need for a high-level but accurate semantic model for partial failure, to support reasoning about the behaviour of these algorithms. As time went on it became clear that the best way to produce such a model would be to base it on an accurate model at the level of the sockets interface, and that this itself (given the limitations of the existing standards and documentation) could be of wide interest.

Work began in earnest in October 2000, taking as a starting point (unicast) UDP and the associated parts of the Sockets API and of ICMP, as implemented in Linux 2.2.16–22. This produced a specification in non-mechanised mathematics, validated with the aid of a `udpautotest` program that simulated the model (hand-translated into C) in parallel with executing the real socket calls. We proved several sanity properties of the model. The specification was integrated with an operational semantics for a single-threaded 'MiniCaml' fragment of the OCaml language, enabling informal proof about a very simple (but executable) 'single heartbeat' program. This was published as a Technical Report and in TACS 2001:

- The UDP calculus: Rigorous semantics for real networking. Andrei Serjantov, Peter Sewell, and Keith Wansbrough. Technical Report 515, Computer Laboratory, University of Cambridge, July 2001. iv+70pp. [SSW01a]
- The UDP calculus: Rigorous semantics for real networking. Andrei Serjantov, Peter Sewell, and Keith Wansbrough. In *Proceedings of TACS 2001: Theoretical Aspects of Computer Software (Sendai), LNCS 2215*, pages 535–559, October 2001. [SSW01b]

The limitations of informal non-mechanised mathematics for such work quickly became apparent. The initial UDP host semantics, while much simpler than the current specification, still had some 82 rules; by the standards of typical process calculi it was rather large and keeping it internally self-consistent was non-trivial. In April 2001 we therefore began translating it into HOL, making it fully rigorous and with substantial extensions to model time, multi-threaded application programs, and the behaviour of partial systems, together with machine-checked proofs of the main sanity properties. This was reported in ESOP 2002, with a SIGOPS EW 2002 position paper reflecting on the experience of this and of Norrish's C formalisation work.

- Timing UDP: mechanized semantics for sockets, threads and failures. Keith Wansbrough, Michael Norrish, Peter Sewell, and Andrei Serjantov. In *Proceedings of ESOP 2002: the 11th European Symposium on Programming (Grenoble), LNCS 2305*, pages 278–294, April 2002. [WNSS02]
- Rigour is good for you, and feasible: reflections on formal treatments of C and UDP sockets. Michael Norrish, Peter Sewell, and Keith Wansbrough. In *Proceedings of the 10th ACM SIGOPS European Workshop (Saint-Emilion)*, pages 49–53, September 2002. [NSW02]

A new specification, covering TCP and sockets and initially based on the combination of the RFCs, POSIX, and BSD code, was begun in June 2002. The earlier UDP rules were later adapted and folded in. Work proceeded simultaneously on the specification, the automated testing machinery, and the symbolic evaluator; the first error in the specification found by automated testing was in September 2003.

Overall, the initial UDP work took perhaps two man-years over 10 months; the subsequent TCP (and UDP) work has taken approximately 7 man-years over 30 months, to February 2005. Of this, much has been devoted to idiom and tool development, and much to unpicking the intricacies of the existing TCP implementations. Given the complexity of the protocols and API, and the amount of effort devoted to them over the last 25 years, these figures seem rather modest. Moreover, similar work carried out at protocol design-time rather than post-hoc, and building on this experience, should be considerably less time-consuming.

12 Discussion

12.1 Summary

We have produced a rigorous post-hoc specification of the behaviour of real-world network protocols TCP and UDP, and their Sockets API, validated against deployed implementations.

The specification may be of wide use to different communities:

- as documentation for users and implementors of the sockets API (designers of distributed systems and implementors of protocol stacks respectively);
- as a basis for formal reasoning about executable descriptions of distributed algorithms (contrasting with the more usual proofs about non-executable pseudocode or automata-theoretic descriptions);
- as a clear starting point for description of changes to the protocols; and
- as a basis for the design of high-level abstractions and programming languages with accurate failure semantics.

Further, while our validation tools have been developed to validate the specification against existing implementations, the same tools could be used to test future implementations against the specification, giving high-quality automated conformance testing.

Our main conclusion is that such work is feasible. We have been able to deal rigorously with a behaviourally-complex real-world system without making unreasonable idealisations. Doing so has required careful choices of exactly what to model and how to model it, and extensive work on automated testing and validation. Automated testing, using a formal specification as an oracle, is an extremely powerful technique. It does not give as much assurance as formal verification of code, of course, but it can scale to systems for which formal verification would be prohibitive — in our case the C TCP/IP protocol stack implementations in existing operating systems. This is certainly not the first application of specification-based testing, but it may be one of the most substantial for systems with such nondeterministic and time-dependent behaviour.

We did not set out to find bugs in the existing implementations. Indeed, from the point of view of this post-hoc specification work there are no bugs — those implementations are the de facto standard. Moreover, the implementations have been extremely widely used (albeit usually in rather stylised idioms), and the network does by and large work, so it would be very surprising to find major problems. Nonetheless we did observe a number of behavioural oddities and implementation differences, as described in §9, some of which should be addressed by the implementation teams.

TCP, UDP and the Sockets API are among the most widely deployed, long lived, and most critical software infrastructure (together with, for example, IP, certain routing protocols, the x86 architecture, the JVM and .NET abstract machines, and common programming languages). Post-hoc specification work, dealing with historical details and complexities, is certainly not appropriate for arbitrary software, but can be worthwhile for these.

12.2 Future work

There are many directions for future work. One of the most interesting is to carry out specification and automated validation work *at design time* for new protocols, rather than (25 years!) after the fact. We discuss that in the following subsection, and list other future directions below.

- We have tried to make the specification usable as an informal reference, for developers working above the Sockets API and for TCP/IP stack implementors, by annotating and structuring the HOL logic definitions. It is now very interesting to discover how far this has succeeded, and what else might be done to assist. The problem is one common to all large protocols, especially those without a clear modular design: to fully understand the behaviour one needs to understand all the detail, but to develop an understanding one must take it piece by piece.

- While the specification has been developed based on three particular implementations, and with reference to the Linux and (especially, for TCP) the BSD source code, 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 OS-specific change is required. Our automated validation for TCP makes use of the BSD debug trace records to resolve nondeterminism early — how essential this is is unclear, and whether one could really produce a useful post-hoc specification for (especially) the congestion control algorithms of a black-box implementation is also open.

- Similarly, as the specification has been developed in a process with automated validation against a set of tests, and as that set has been rather static, it would be interesting to confirm that the specification does include other behaviour — checking whether our tests do in fact give good coverage. One could design new tests by hand, aiming to pick up tricky corner cases, or capture them from live applications (though avoiding traces too long to feasibly check), or generate them randomly. Coverage testing with respect to the implementation code would also be interesting.

- The BSD debug trace records make checking easier by resolving nondeterminism, but simultaneously make checking more demanding by allowing internal state of the model to be checked directly rather than only as it becomes visible in behaviour at the API or network interfaces. Without them, one might wish to use longer traces, exploring the host behaviour after events of interest.

- The checker proves an HOL theorem for each step of a successfully-checked trace, but those theorems are typically predicated on a few outstanding constraints. Examining these suggests that they are easily satisfiable (and otherwise, the simplifier would most likely have discovered a contradiction), but it would be useful to add an explicit satisfiability checking phase at the end of each trace.

- It would be useful to maintain and develop the specification, completing the Linux and WinXP validation for TCP, tracking version changes in all three operating systems, and providing feedback to implementation groups. We have used only the versions current during the initial setup of our test network: FreeBSD 4.6-RELEASE (June 2002); Linux 2.4.20-8 (November 2002 + patch 8); and Windows XP Professional SP1 (September 2002). At the time of writing the latest versions of the above are: FreeBSD 4.11 / 5.3 (`www.freebsd.org`); Linux 2.4.29 / 2.6.10 (`www.kernel.org`); and Windows XP Professional SP2. We believe that tracking changes would now be an essentially routine task, using our tools, though one requiring significant long-term effort.

- Given this segment-level endpoint specification we intend to produce a more abstract stream-level end-to-end specification of TCP and Sockets. This will involve informal analysis of the failure semantics of the current specification, characterising how it behaves for segment loss etc. Much of the Sockets part of the Host LTS will be reusable, while the protocol part should be radically simplified. Minor changes to the testing setup, new tests, and some new symbolic model checking work should enable it to be validated directly. A stream-level specification should have fewer differences between architectures.

- In the opposite direction, towards the concrete, we would like to be able to modularly refine the specification, resolving the points at which it is nondeterministic, so that it does describe an implementation. This would entail specifying aspects of the host scheduling (resolving nondeterminism between multiple rules that can fire simultaneously), giving algorithms for choosing initial sequence numbers, options, etc., and constraining TCP output so that it does have the ACK-clock behaviour. It should then be possible to integrate the specification, our symbolic evaluation engine, and the packet injector and slurp tools, forming a working TCP implementation — for example, on receiving a segment from the slurp tool, it would run the symbolic evaluator to calculate a new host state, which might produce new segments to output via the injector. One could gain additional confidence in the validity of the specification by checking this interoperates with existing TCP/IP stacks, though they would have to be artificially slowed down to match the speed of the evaluator. This would demonstrate that executable prototype implementations of future protocols could be directly based on similar specifications.

- As a formal artifact, above which one might do machine-checked proof, the specification is intimidatingly large. Nonetheless, there are several possibilities. Firstly, 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). Secondly, if developing a stream-level end-to-end specification (as above) it would be useful to formally state the intended abstraction relation between the segment-level and stream-level models. In principle one might then prove that relationship is maintained — amounting to a proof that the TCP protocol does indeed provide the service described by the stream-level model. That would be a major intellectual achievement, advancing the state of the art in machine-assisted proof, but for the time being we suspect it would not be feasible. Even if it is, the proof effort required may be disproportionate to what would be learned about the protocol — which is, after all, known to work reasonably well in practice.

- The specification was informed by the BSD and Linux C source code, but the abstraction gap between specification and code is both large and informal. For any future reverse-engineering of post-hoc specifications from code it would be very useful to have semantics-based tools for manipulating the C code, e.g. to partially evaluate functions for constrained inputs, and to perform meaning-preserving code transformations, to let the various conceptually-distinct execution paths be syntactically separated. Dealing with the full C semantics involved (with jumps through pointers, concurrency, etc.) may be too challenging at present, but even partial tools would be useful. One might also extract properties from the specification that could be used for software model checking of such pieces of a full TCP/IP stack. This would be a step towards the pure transport-layer specification mentioned in §2.1.

- 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 must deal with many computation paths that have some important side-effect but then abort. Any improvement to this structure would be worthwhile.

- There are various more radical proposals to replace the existing congestion control mechanisms, e.g. by congestion pricing algorithms.It would be very interesting to take our current specification, remove the existing congestion control (and perhaps other more-or-less obsolete aspects, e.g. urgent data), and specify the discrete behaviour of such proposals.

- The specification characterises the behaviour of a host in response to arbitrary Sockets API calls, yet some call sequences are clearly pathological (e.g. the BSD possibility of moving to LISTEN from any state) and others may be very uncommon (perhaps some simultaneous open scenarios). It would be interesting to identify what call sequences do occur in practice, in a large corpus of applications. One could then define a notion of legal use of the API, ensuring of course that it could be efficiently enforced by an implementation and that it dealt with concurrent calls. Finally, one could investigate how far the specification could be simplified under the assumption that all API usages are legal. (In principle the same could be done for the network interface, but there would be no point — we should be concerned with, and specify, the behaviour of implementations in the face of arbitrary malicious incoming traffic.)

- Closely related to the above, one might implement thin-layer libraries above Sockets that provide cleaner communication abstractions, encapsulating the implementation differences (as has been done many times). The semantics of these higher-level APIs could be given directly, with specifications roughly in the style of the TCP stream-level end-to-end specification mentioned above but hopefully much simpler. One could then consider independent experimental validation, and/or formal proof about those implementations, in terms of the segment-level specification.

- A specification of real-world communication primitives —either this for segment-level TCP (and UDP), or a stream-level TCP model, or a specification of a thin-layer library— can provide a basis for formal machine-checked proof of executable descriptions of distributed algorithms. Such proofs are usually carried out for pseudocode or automata-theoretic descriptions, and often without machine checking, leaving a wide abstraction gap between the ultimate implementations of the algorithms and the subject of the proof. By combining a network specification and a programming language semantics that gap can be much reduced. Michael Compton has carried out preliminary work along these lines, proving properties of an executable OCaml implementation of Stenning's protocol above our earlier UDP/Sockets specification, in the Isabelle [Isa] proof assistant [Com05].

  Proofs about signficant bodies of code above TCP will be very challenging (and would likely need a stream-level specification) but in the long term are well worth-while.

- Our symbolic evaluator is a special-purpose tool, much 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.

- The performance of the symbolic evaluator has been a significant constraint, pushing the HOL implementation to its limits. Porting HOL from its current implementation language, interpreted Moscow ML, to a native compiler such as SML/NJ or MLton could improve performance by a good constant factor. This would require some extensive but relatively straightforward engineering.

12.3 Specification at design-time

If it is feasible to specify the behaviour of complex existing protocols, it should certainly be feasible to use rigorous behavioural specification at design time for new protocols. We believe this would be very much worthwhile, contributing in two ways to the development of higher-quality protocol designs and implementations, and of software that uses them. Firstly, simply removing the ambiguity that is inescapable in natural language specification would aid precise communication, both amongst protocol design groups and to later implementors and users. Secondly, a rigorous specification should make it easier to see a design as a whole, and to ensure that all cases are covered appropriately. The process of producing an internally-consistent specification encourages clean design — unnecessary irregularities and complexities in the behaviour are brought out very sharply in a rigorous specification, whereas in a textual specification they may be concealed.

Network protocols and APIs are among the (perhaps relatively few) areas of computing where this argument can be made so forcefully. Here there are specifications which are the widely-accepted reference for multiple implementations. Interoperability between the implementations is vital, and hence so are clear specifications.

While the up-front effort required for a formal specification may be greater than that for a textual document, contrast the few man-years required to develop the specification presented here (including development of the idioms and tools) with the many man-years devoted over the last decades worldwide to struggling with network arcana, ambiguities and implementation differences. If the former saved only a small fraction of the latter it would be a big long-term win.

If specifying a future protocol at design time, our experience suggests several points to consider.

1. Ideally one would develop both an endpoint specification, prescribing the behaviour of a single protocol stack, and an end-to-end specification, characterising the behaviour that API users can depend upon.

2. An endpoint specification should include the interface and behaviour of the API, as well as the wire behaviour — neglecting the API will tend to lead to subtle implementation differences, and hence portability problems for software that uses it.

3. The specification should clearly define how an acceptable implementation may behave. Critically, it should do so in a way that makes it possible to *test* whether an execution of an implementation is admissible. The specification should be directly usable in conformance checking, without any unchecked manual (error-prone) translation.

4. The protocol and API should be designed with this testing in mind. For TCP and Sockets 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 are not determined in a simple way by the observable events. This has 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. Some fairly modest features of a protocol design would make automated testing much more straightforward (in lock-step between implementation and specification, with completely ground states):

   - ensuring the API can reveal internal state changes and time events;
   - ensuring the API can reveal nondeterministic choices of values, e.g. of randomly-chosen values such as initial sequence numbers;
   - ensuring the API can reveal the entire internal implementation state; and

- ensuring that abstraction functions from implementation state to the abstract states used in the specification can be programmed.

The instrumentation should be switchable without affecting the rest of the protocol and API behaviour, so that normal-case performance is not affected but when needed instrumentation can be performed transparently.

5. The treatment of loose specification and nondeterminism is crucial. A fully deterministic specification, e.g. of a protocol endpoint that for any incoming message immediately output a message that is a pure function of that input message, could be expressed in a conventional language. A purely functional language might be clearest, e.g. Haskell or the pure fragment of ML. Testing would simply check equality between the result of the specification and of an implementation. More typically, a protocol specification will have to be loose, allowing endpoints to operate at different speeds, with different internal scheduling, with different window sizes etc., and perhaps (as for different versions of TCP) with significantly different algorithms. A loose specification should clearly not be too loose — it should be the case that any implementation that formally satisfies it does interoperate well with any other. Given that, it is broadly desirable for the specification to be as loose as possible, allowing implementation freedom. As discussed above, more nondeterminism may make testing difficult, but this is not always true — for example, for the extreme specification that admits an arbitrary implementation, testing is trivial. For our TCP specification the (nondeterministic) labelled transition system is expressed by axioms over host states which are close to the implementation states. Another idiom for loose specification would be to explicitly constrain the allowable histories of events. For example, one could express band-limiting properties by predicates over traces stating directly an upper bound on the number of ICMPs that may be generated per second. For a design-time specification this might be a good idiom for many properties, capturing more precisely what the protocol designer has in mind.

6. The choice of specifcation language is also crucial, with the associated trade-off between decidability and expressiveness. For a nondeterministic specification one could either:

   (a) write the specification explicitly as a conformance-checking program in a conventional programming language; or

   (b) choose a logic for which the conformance-checking questions (e.g. 'is the following trace admitted by the specification') are decidable; or

   (c) choose a general-purpose logic, such as HOL, and define the specification within it using operational semantics. Here arbitrary properties will be undecidable, but for a particular specification it may be possible to produce a decision procedure for arbitrary traces, or even (as we have done) just sufficiently-good proof heuristics.

   Option (6a) is technically straightforward but may lead to rather opaque specifications, with algorithmic concerns for testing intertwined with the conceptual protocol design. Option (6b) is perhaps the ideal, but it places severe limits on the expressiveness of the logic. For TCP we have found a rich language of types, arithmetic, and nested quantification essential, and so are forced to take option (6c). HOL is flexible enough to let us specify in the clearest idioms we can, while developing provably-equivalent but algorithmically more tractable forms for testing. It allows explicit nondeterminism to be used where the specification should be loose.

7. The structure of a good endpoint specification may be quite different from that of a good implementation, with the former arranged for clarity and the latter for common-case performance. For TCP we have factored the behaviour into more-or-less conceptually distinct Host LTS rules, which correspond to overlapping parts of the implementation code. We have also introduced as much modular structure as we could, with auxiliary definitions and the relational monad used in *deliver_in_3*. As a post-hoc activity, however, this has been constrained by the imperative and intertwined implementation code. For a design-time specification one should be able to use a much clearer modular structure to factor out different aspects of the protocol. For example, one might arrange the specification in a microprotocol structure, while intending a monolithic optimised implementation. Algorithmic components that are likely to change, e.g. options, congestion control algorithms, etc. should be factored out as much as possible.

8. It is desirable to make an endpoint specification executable as a prototype implementation with as little change as possible. For this one might which to arrange for the points at which it only loosely

constrains implementations to be factored out, so they could be instantiated with particular algorithms. For our TCP specification such a prototype would be extremely slow, but still useful. How such a determinised specification could be automatically compiled to give a reasonable-performance reference implementation is an interesting problem.

9. Given both an endpoint and an end-to-end specification, one can gain confidence in the design by checking the two are consistent, in several ways:

   (a) The abstraction relationship between the two should be precisely defined. For example, for TCP this would take two (endpoint) host states and a network state of messages in transit, and give the set of (end-to-end) states —streams of data in transit— they may correspond to.

   (b) With minimal investment one could independently test implementations with respect to *both* specifications, and test that the abstraction relationship is maintained.

   (c) Ideally (but with a large investment) one could *prove* that the abstraction relationship is maintained. This would be a proof that the low-level endpoint specification, when composed with a network model, does correctly implement the end-to-end specification.

10. One can also gain confidence by precisely stating any other intended invariants of the specification, and either testing or (preferably) proving they are preserved. For the TCP specification we have a number of invariants noted, e.g. that certain timers are not active simultaneously, or that if a socket ID is in a host's list of bound sockets then there is an associated socket structure. They have not been formalised simply for lack of time. There may also be other useful sanity properties worth proving, e.g. that the specification does accept any incoming message in any state, and semideterminacy properties stating that a non-erroneous socket call in a particular host state has exactly one possible outcome.

11. To make the API independent of any particular programming language one might formally specify a pure value-passing interface (but take care to cover the concurrency aspects), as we did here. A value-passing interface can be related to a pointer-passing C language binding, or to an object-oriented API, in two ways: either by describing (informally and in code) what value-passing events to generate from the language invocations, as we did here, or by actually integrating the protocol model with an operational semantics for the programming language. While good in principle, the latter may be impractical at present.

12. An API specification should define what the legal usages of the API are (including concurrent use by different threads). One should ensure that this can be efficiently enforced by an implementation, with illegal usages immediately returning errors, to prevent pathological code being developed.

13. The protocol end of a specification should define the behaviour in response to arbitrary (perhaps malicious) incoming messages.

14. For our TCP specification, we have aimed to capture all the important details of the protocol — everything that an implementor would need to be concerned with. The current specification does not quite reach that goal (for example, the timing of TCP output is not sufficiently constrained), but we believe it is close. Design-time specifications might similarly describe entire protocol/API combinations. Sometimes, though, simplified specifications of certain aspects of a protocol are worth investigating, for example as a basis for correctness proofs about some particularly tricky point. For TCP one might specify flow control but be entirely nondeterministic about the details of congestion control, aiming to prove that the flow control algorithm never loses data. Simplified specifications could be written in the same idiom, as labelled transition systems in HOL, and prhaps even just as microprotocol specification layers.

15. The use of a logic for specification (rather than natural language or a programming language) means that proof assistant tools are needed, e.g. the HOL system [HOL], Isabelle [Isa], Coq [Coq], etc.. One must ask the question of how accessible such logics and tools are for protocol designers. In our experience getting to grips with the HOL system and TCP specification is reasonably straightforward for a good undergraduate student, taking only a few weeks before useful work can be done in writing and correcting specification definitions. Developing the symbolic evaluator code which we use for validation is a more specialist activity, needing an expert in the proof assistant.

16. Finally, we would like to emphasise the need to keep protocols and APIs simple. Complex functionality that is not often used will tend not to be implemented 'correctly', or at least not identically

in different architectures, and then will not be usable. As Anderson et al say in their *Design guidelines for robust Internet protocols* [ASSW03] ("Guideline #1: Value conceptual simplicity"), this is widely accepted but hard to follow. Writing a behavioural specification may make complexity apparent early in the design process, in a way that informal prose specifications and mostly-working code does not.

The original TCP/IP specifications were written in informal prose, around 1980. That informal approach, and the emphasis on working interoperable code that went along with it, served the community well in many ways — making the specifications accessible to a wide community, encouraging change where necessary, and discouraging early over-specification. Moreover, the specification tools of the day were probably not adequate to the task of describing real-world protocols. The original Edinburgh LCF system, of which HOL is a direct descendant, was also published around 1980 [GMW79], and a great deal of progress in automated reasoning has been made since. Moore's law has also contributed to making large-scale formal work tractable. Looking forward, however, we hope the long-term advantages of rigorously specified protocols, along with automated specification-based testing and (where possible) proof, are now clear.

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 Michael and Morven Heller Research Fellowship (Wansbrough), EPSRC grant GRN24872 *Wide-area programming: Language, Semantics and Infrastructure Design*, 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.

References

[ASSW03] Thomas E. Anderson, Scott Shenker, Ion Stoica, and David Wetherall. Design guidelines for robust internet protocols. *Computer Communication Review*, 33(1):125–130, 2003.

[AW01] Rajeev Alur and Bow-Yaw Wang. Verifying network protocol implementations by symbolic refinement checking. In *Proc. CAV '01, LNCS 2102*, pages 169–181, 2001.

[BCMG01] Karthikeyan Bhargavan, Satish Chandra, Peter J. McCann, and Carl A. Gunter. What packets may come: automata for network monitoring. In *Proc. POPL*, pages 206–219, 2001.

[BFN$^+$05] Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 2: The specification. Draft available at `http://www.cl.cam.ac.uk/users/pes20/Netsem/`, 2005. xxiv+359pp.

[BH03] Jonathan Billington and Bing Han. On defining the service provided by TCP. In *Proc. ACSC: 26th Australasian Computer Science Conference*, Adelaide, 2003.

[BH04] Jonathan Billington and Bing Han. Closed form expressions for the state space of TCP's data transfer service operating over unbounded channels. In *Proc. ACSC: 27th Australasian Computer Science Conference*, pages 31–39, Dunedin, 2004.

[Bia94] Edoardo Biagioni. A structured TCP in standard ML. In *Proc. SIGCOMM '94*, pages 36–45, 1994.

[BOG02] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, 2002.

[CDO97] Claude Castelluccia, Walid Dabbous, and Sean 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.

[CHdV03] 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.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Com00] Douglas E. Comer. *Internetworking With TCP/IP Volume 1: Principles Protocols, and Architecture.* 4th edition, 2000.

[Com05] Michael Compton. Stenning's protocol implemented in UDP and verified in Isabelle. In *Proceedings of The Australasian Theory Symposium*, 2005. To appear.

[Coq] The Coq proof assistant. `http://coq.inria.fr/`.

[CS99] Douglas E. Comer and David L. Stevens. *Internetworking With TCP/IP Volume II: Design, Implementation, and Internals.* 3rd edition, 1999.

[CS00] Douglas E. Comer and David L. Stevens. *Internetworking With TCP/IP Volume III: Client-Server Programming and Applications, Linux/POSIX Socket Version.* 2000.

[FJ00] Elena Fersman and Bengt Jonsson. Abstraction of communication channels in Promela: A case study. In *Proc. 7th SPIN Workshop, LNCS 1885*, pages 187–204, 2000.

[Fyo] Fyodor. `nmap`. `http://www.insecure.org/nmap/`.

[GM93] M. J. C. Gordon and T. Melham, editors. *Introduction to HOL: a theorem proving environment.* Cambridge University Press, 1993.

[GMW79] Michael Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF, LNCS 78.* 1979.

[Gra] Graphviz — graph visualization software. `http://www.graphviz.org/`.

[HL00] Richard Hofmann and Frank Lemmen. Specification-driven monitoring of TCP/IP. In *Proc. 8th Euromicro Workshop on Parallel and Distributed Processing*, January 2000.

[HLvR99] Jason Hickey, Nancy A. Lynch, and Robbert van Renesse. Specifications and proofs for Ensemble layers. In *Proc. TACAS, LNCS 1579*, pages 119–133, 1999.

[HOL] The HOL 4 system, Kananaskis-2 release. `http://hol.sourceforge.net/`.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP '91, LNCS 512*, pages 133–147, July 1991.

[IEE00] IEEE. *Information Technology—Portable Operating System Interface (POSIX)—Part xx: Protocol Independent Interfaces (PII), P1003.1g*. Institute of Electrical and Electronics Engineers, March 2000.

[Isa] The Isabelle proof assistant. `http://isabelle.in.tum.de/`.

[IXI05] IXIA. IxANVL(tm) — automated network validation library, 2005. `http://www.ixiacom.com/products/conformance_applications/`.

[KKM99] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable TCP in the Prolac protocol language. In *Proc. SIGGCOMM '99*, pages 3–13, August 1999.

[Kre04] Christoph Kreitz. Building reliable, high-performance networks with the Nuprl proof development system. *J. Funct. Program.*, 14(1):21–68, 2004.

[L$^+$04] Xavier Leroy et al. *The Objective-Caml System, Release 3.08.2*. INRIA, November 2004. Available `http://caml.inria.fr/`.

[Lam] Lambda Prolog. `http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/index.html`.

[LCH$^+$02] David Lee, Dongluo Chen, Ruibing Hao, Raymond E. Miller, Jianping Wu, and Xia Yin. A formal approach for passive testing of protocol data portions. In *Proc. ICNP*, 2002.

[LV96] Nancy Lynch and Frits Vaandrager. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[ME04] 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.

[Mer] Mercury. `http://www.cs.mu.oz.au/research/mercury/`.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I + II. *Information and Computation*, 100(1):1–77, 1992.

[MS87] S. L. Murphy and A. U. Shankar. A verified connection management protocol for the transport layer. In *Proc. SIGCOMM*, pages 110–125, 1987.

[MS88] S. L. Murphy and A. U. Shankar. Service specification and protocol construction for the transport layer. In *Proc. SIGCOMM*, pages 88–97, 1988.

[Net] Netsem page. `http://www.cl.cam.ac.uk/users/pes20/Netsem/`.

[Nor98] Michael Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.

[Nor99] Michael Norrish. Deterministic expressions in c. In *Proc. ESOP: 8th European Symposium on Programming*, pages 147–161, 1999.

[NSW02] Michael Norrish, Peter Sewell, and Keith 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, September 2002.

[Pax97] Vern Paxson. Automated packet trace analysis of TCP implementations. In *Proc. SIGCOMM '97*, pages 167–179, 1997.

[PF01] Jitendra Padhye and Sally Floyd. On inferring TCP behaviour. In *Proc. SIGCOMM '01*, August 2001.

[Pos74] J. Postel. *A Graph Model Analysis of Computer Communications Protocols.* University of California, Computer Science Department, PhD Thesis, 1974.

[PS98] S. Parker and C. Schmechel. RFC2398: Some testing tools for TCP implementors, August 1998.

[PVS] The PVS specification and verification system. `http://pvs.csl.sri.com/`.

[Sch96] 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.

[Sch00] Steve Schneider. *Concurrent and Real-time Systems: The CSP Approach.* Worldwide Series in Computer Science. John Wiley & Sons, 2000.

[SGSAL98] Roberto Segala, Rainer Gawlick, Jørgen Søgaard-Andersen, and Nancy Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141:119–171, 1998.

[Smi96] M. A. S. Smith. Formal verification of communication protocols. In *Proc. FORTE 96. Formal Description Techniques IX: Theory, application and tools, IFIP TC6 WG6.1 International Conference on Formal Description Techniques IX / Protocol Specification, Testing and Verification XVI*, pages 129–144, 1996.

[SR02] Mark 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.

[SSW01a] Andrei Serjantov, Peter Sewell, and Keith Wansbrough. The UDP calculus: Rigorous semantics for real networking. Technical Report 515, Computer Laboratory, University of Cambridge, July 2001. `http://www.cl.cam.ac.uk/users/pes20/Netsem/`.

[SSW01b] Andrei Serjantov, Peter Sewell, and Keith 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*, October 2001.

[Ste94] W. R. Stevens. *TCP/IP Illustrated Vol. 1: The Protocols.* 1994.

[Ste98] W. Richard Stevens. *UNIX Network Programming Vol. 1: Networking APIs: Sockets and XTI.* Second edition, 1998.

[SWP99] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*, pages 1–31, October 1999.

[US01] Asis Unyapoth and Peter Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proceedings of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (London)*, pages 116–127, January 2001.

[WNSS02] Keith Wansbrough, Michael Norrish, Peter Sewell, and Andrei 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, April 2002.

[WS95] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated Vol. 2: The Implementation.* 1995.

[Yi91] Wang Yi. CCS + time = an interleaving model for real time systems. In J. Leach Albert, B. Monien, and M. Rodríguez Artalejo, editors, *Automata, Languages and Programming, 18th International Colloquium, Madrid, Spain*, number 510 in Lecture Notes in Computer Science, pages 217–228. Springer-Verlag, July 1991.