PEPITO

IST-2001-33234

PEer-to-Peer Implementation and TheOry

Deliverable no: D1.4

Study of possible semantics of failures

REPORT VERSION: first

REPORT PREPARATION DATE: 2005.2.28

CLASSIFICATION: Public

DELIVERABLE NO: D1.4 DUE DATE: Month 38 DELIVERY DATE: Month 38

PROJECT START DATE: 2002.01.01 PROJECT DURATION: 36+4 months

RESPONSIBLE PARTNER: UCAM

PARTICIPATING PARTNERS: UCAM

PROJECT COORDINATOR: Swedish Institute of Computer Science AB

PROJECT PARTNERS: EPFL Lausanne, INRIA Paris, KTH Stockholm, UCL Louvain, University of Cambridge UK

Project funded by the European Community under the 'Information Society Technologies' Programme (1998–2002)

Project Number: IST-2001-33234
Project Acronym: PEPITO
Title: PEer-to-Peer Implementation and TheOry
Deliverable No: D1.4
Study of possible semantics of failures
Due date: project month 38
Delivery date: 2005.2.28

Responsible Partner: UCAM
Participating Partners: UCAM

9th March 2005

Author: Peter Sewell

Contents

1 Introduction **4**

2 Motivation **4**

3 Contribution **5**

4 Approach **5**

4.1 The Specification Language 5

4.2 Conformance Checking 6

4.3 Experimental Semantics 6

5 The Specification **7**

6 Modelling **8**

6.1 Where to cut 8

6.2 The form of the specification 9

6.3 Network interface issues 9

6.4 Sockets interface issues 10

6.5 Protocol issues 10

6.6 Time 11

6.7 Relationship between code and specification structure 11

7 The Specification **11**

8 Experimental Validation **12**

8.1 Trace generation 12

8.2 Tests 12

8.3 Symbolic evaluation 12

8.4 Distributed checking infrastructure 13

9 Results **14**

10 Implementation anomalies **14**

11 Related work **15**

12 Required background and resource **16**

13 Conclusion **17**

13.1 Summary 17

13.2 Future Work 17

1 Introduction

In this Deliverable we report on the low-level failure semantics subtask of Task

▷ Transactions and the Semantics of Failure

in Workpackage 1.

Deliverable D1.2 *Survey of formal challenges and solutions for peer-to-peer computation* puts this work in the wider context of WP1 and PEPITO, and Deliverable D1.9 *Final report on formal models* gives the associated papers.

A main goal here was to establish an accurate failure semantics for low-level interaction. As we stated in the project proposal:

> "We will develop rigorous semantic models of the low-level interactions and failures that may occur in a peer-to-peer system — not in the rather abstract styles of traditional distributed algorithm theory or process calculi, but for the actual networking APIs used."

The final period of PEPITO has seen this achieved. We have produced a behavioural specification of the networking *Sockets API* and the underlying TCP and UDP protocols that exactly characterises behaviour in the presence of message loss, malicious attack with arbitrary messages, and arbitrary use and misuse of the API. It is mathematically rigorous, detailed, and has been shown to be accurate by an experimental validation process. The specification and an extended discussion document have been made available; one paper has been submitted for publication and more will follow in the future. The work establishes practical techniques for rigorous description and automated testing that can also be used for future protocol designs, at either the transport or peer-to-peer overlay levels.

[1] Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. Submitted for publication. Available at `http://www.cl.cam.ac.uk/users/pes20/Netsem/paper.pdf`, February 2005. 15pp.

[2] Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 1: Overview. Available at `http://www.cl.cam.ac.uk/users/pes20/Netsem/tr.pdf`, February 2005. vi+96pp.

[3] 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. Available at `http://www.cl.cam.ac.uk/users/pes20/Netsem/alldoc.pdf`, February 2005. xxv+380pp.

2 Motivation

Most peer-to-peer programs, in common with almost all distributed libraries and applications, must be programmed above the well-known *Sockets API* interface to the ubiquitous TCP, UDP, and IP network protocols. These protocols are described by a variety of *Request for comment* (RFC) standards, and the API is described by the POSIX standard. Unfortunately, however, these descriptions are lacking: they rely on informal prose to

characterise the behaviour of systems, and hence are inevitably ambigous and incomplete. Partly as a consequence of this, the behaviour of the API and protocols *as they are actually implemented* is extremely complex; implementations often have subtle differences and bugs, and conformance testing is very challenging. This is especially true in the (unavoidable) presence of host failure, message loss, and malicious attack injecting fraudulent messages into the system.

We set out to characterise precisely what behaviour (especially, what information about failure) is visible to the application programmer above the Sockets API. To do so, it is obviously also necessary to capture the full normal-case behaviour, and for the work to be useful it should also make clear what the differences are between various deployed implementations.

Such a characterisation should be useful both as a reference for developers working above the Sockets API (supplementing the existing standards and texts), as a basis for formal proof about higher-level communication libraries, as a basis for the design of failure semantics for high-level programming languages, and as a reference for implementors of protocols. More generally, the techniques we develop should be useful in the design phase of future protocols, both at the transport level and for peer-to-peer overlays.

3 Contribution

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

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

4 Approach

4.1 The Specification Language

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

Accordingly, we write our specification as an operational semantics definition in higher-order logic, mechanized using the HOL system [GM93, HOL]. Higher-order logic is similar to conventional first-order logic (or predicate calculus) with the normal logical operations, quantifiers, etc., but with the addition of a rich type

structure (including numeric types, lists, and functions) and the ability to quantify over any type. It lets one write more-or-less arbitrary mathematics idiomatically.

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

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

4.2 Conformance Checking

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

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

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

4.3 Experimental Semantics

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

We therefore took seriously the fact that for many purposes TCP is defined by the *de facto* standard of the common implementations — intended differences, bugs, and all. Our specification aims to capture the behaviour of

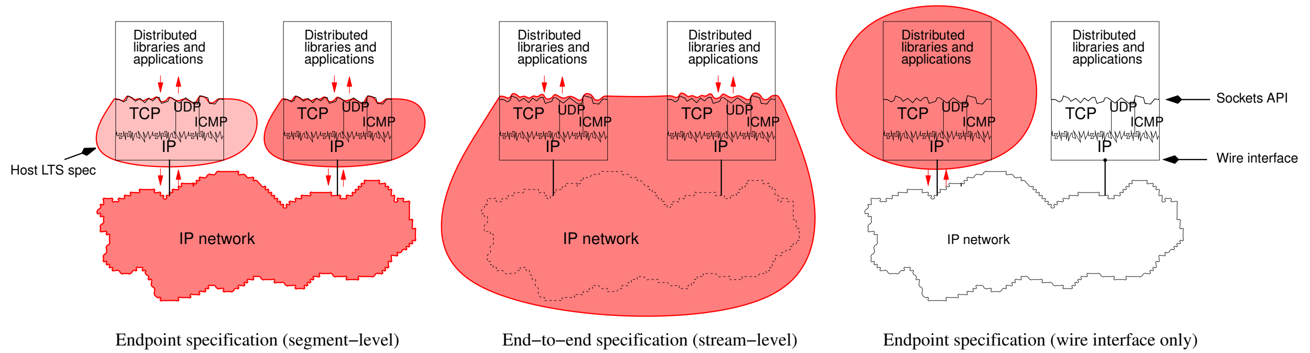

Figure 1: Modelling: where to cut.

three widely-deployed implementations: FreeBSD 4.6–RELEASE, Linux 2.4.20–8, and Windows XP Professional SP1. Behavioural differences between the three are made explicit by parameterisation on the OS version. (These clearly do not cover the behaviour of all important implementations, but they are an indicative sample.)

The first drafts of the specification were based on the RFCs (especially 768, 791, 792, 793, 1122, 1323, 2414, 2581, 2582, 2988, 3522, and 3782), POSIX standard [IT01], Stevens's texts [Ste94, WS95, Ste98], BSD and Linux source code, and *ad hoc* tests. Such work is error-prone; one should have little confidence in the result without some form of validation. We therefore used our symbolic evaluation techniques to validate the specification directly against the implementations (as opposed to the more common validation of an implementation against a specification). We wrote tools to generate several thousand traces from the implementations running on an instrumented test network, chosen to give as broad coverage as we could, and used the checker to ensure that the specification does admit those traces. Validation is computationally intensive, so for reasonable performance it was necessary to distribute the task over a bank of machines (currently around 100 processors).

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

5 The Specification

The specification is available online, together with a full description of the project [BFN+05c, BFN+05a].

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

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

our automated validation process is, by the standards of normal software development, an extremely demanding test.

6 Modelling

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

6.1 Where to cut

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

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

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

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

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

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

6.2 The form of the specification

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

The host LTS defines a transition relation

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

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

- msg for the host receiving a datagram msg from the network
- $\overline{msg}$ for the host sending a datagram msg to the network
- $tid{\cdot}\mathrm{bind}(fd, is_1, ps_1)$ for a $bind()$ call being made to the Sockets API by thread tid, with arguments $(fd, is1, ps1)$ 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

The transition relation is defined by some 148 rules for the socket calls (5–10 for each interesting call) and some 46 rules for message send/receive and for internal behaviour. Each rule has a name, e.g. $bind - 5$, $deliver - in - 1$ etc., and various attributes. These rules form the main part of the specification.

The host LTS can be combined with a model of the IP network, e.g. abstracting from routing topology but allowing message delay, reordering, and loss, to give a full specification. In turn, that specification can be used together with a semantic description of a programming language to give a model of complete systems: an IP network; many hosts, each with their TCP/IP protocol stack; and executable code on each host making Sockets API calls.

Compton has demonstrated fully formal reasoning about executable OCaml code above our earlier UDP specification [Com05], using the Isabelle proof assistant [Isa].

6.3 Network interface issues

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

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

Cutting at the wire interface means that our specification models the behaviour of the entire protocol stack and also the network interface hardware. Our abstraction from IP, however, means that only very limited aspects

of the lower levels need be dealt with explicitly. For example, a model host has queues of input and output messages; each queue models the combination of buffering in the protocol stack and in the network interface.

6.4 Sockets interface issues

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

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

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

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

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

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

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

The model is language-neutral, but we also have an OCaml [L+04] library (implemented above `nssock`) with types almost identical to those of the model.

6.5 Protocol issues

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

6.6 Time

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

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

6.7 Relationship between code and specification structure

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

7 The Specification

The specification is written as a mechanized higher-order logic definition in HOL [GM93, HOL], a language that is rich and expressive yet supports both internal consistency checking (type checking in particular is essential with a definition of this scale) and our automated testing techniques. We have tried hard to establish idioms with as little syntactic noise as possible, e.g. with few explicit 'frame conditions' concerning irrelevant quantities.

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

These rules are supported by type definitions for each interaction and state component, constant definitions for the various protocol parameters, and auxiliary function definitions for common operations and glue.

8 Experimental Validation

8.1 Trace generation

To generate traces of the real-world implementations in a controlled environment we set up an isolated test network, with machines running each of our three OS versions, and wrote instrumentation and test generation tools. We instrument the wire interface with a `slurp` tool above `libpcap`, instrument the Sockets API with an `nssock` wrapper, and on BSD additionally capture TCP control block records generated by the `TCP_DEBUG` kernel option. All three produce HOL format records which are merged into a single trace; this requires accurate timestamping, with careful management of NTP offsets between machines and propagation delays between them. A test executive `tthee` drives the system by making Sockets API calls (via a `libd` daemon) and directly injecting messages with an `injector` tool. These tools are written in OCaml [L+04] with additional C libraries. The resulting traces are HOL-parsable text files containing an initial host state (its interfaces, routing table, etc.), an initial time, and a list of timestamped labels.

8.2 Tests

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

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

For example, trace 1484, of intermediate complexity, is informally described as follows: "send() – for a non-blocking socket in state ESTABLISHED(NO_DATA), with a reduced send buffer that is almost full, attempt to send more data than there is space available."

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

8.3 Symbolic evaluation

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

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

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

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

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

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

8.4 Distributed checking infrastructure

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

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

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

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

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

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

9 Results

The experimental validation process shows that the specification admits almost all the test traces generated. For UDP, over all three implementations (BSD, Linux, and WinXP), 2526 (97.04%) of 2603 traces succeed. For TCP we have focussed recently on the BSD traces, and here 1002 (91.5%) of 1095 traces succeed.

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

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

Of the TCP non-successes: 47 are due to checker problems (mainly memory limits); 5 are due to problems in test generation; and the remaining 41 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). For TCP, however, it would be good to check more medium-length traces, to be sure that the various congestion-control regimes are fully explored. Our trace set is perhaps weighted more towards connection setup/teardown and Sockets API issues.

10 Implementation anomalies

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

Nonetheless, in the course of the work we have found many behavioural anomalies, some of which are certainly bugs in the conventional sense. There are explicit OS version dependencies on around 260 lines of the

specification, and the report [BFN+05b] details around 30 anomalies. All are relatively local issues — the implementations are extremely widely used, so it would be very surprising to find serious problems in the common-case paths. We list a few briefly below, mostly for BSD TCP. By describing these oddities we hope primarily to give some sense of what kind of fine-grain detail can be captured by our automated testing process, in which window values, time values, etc. are checked against their allowable ranges as soon as possible. Some may be already known.

- The receive window is updated on receipt of a bad segment.
- Simultaneous open can respond with an ACK rather than a SYN,ACK.
- The code has an erroneous definition of the `TCPS_HAVERCVDFIN` macro, making it possible, for example, to generate a `SIGURG` signal from a socket after its connection has been closed.

[listen()] can be (erroneously) called from any state, which can lead to pathological segments being transmitted (with no flags or only a FIN).

- After repeated retransmission timeouts the RTT estimates are incorrectly updated.
- After 2^{32} segments there is a 16 segment window during which, if the TCP connection is closed, the RTT values will not be cached in the routing table.
- The received urgent pointer is not updated in the fast-path code, so if 2GB of data is received in the fast path, subsequent urgent data will not be correctly signalled.
- On Linux, options can be sent in a SYN,ACK that were not in the received SYN.

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

11 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]. The properties checked were of two kinds: resource leaks and invalid memory accesses, and protocol-specific properties specified by a hand translation of the RFC793 state diagram into C code. While this is a useful model of the protocol, it is an extremely abstract view, omitting flow control, congestion control etc. Four bugs in the Linux implementation were found.

Bhargavan *et al.* develop an automata-theoretic approach for monitoring of network protocol implementations, with classes of properties that can be efficiently checked on-line in the presence of network effects [BCMG01]. They show that certain properties of TCP implementations can be expressed.

In a rare application of rigorous techniques to actual standards, Bhargavan, Obradovic, and Gunter use a combination of the HOL proof assistant and the SPIN model checker to study properties of distance-vector routing protocols [BOG02], proving correctness theorems. In contrast to our experience for TCP, they found that for RIP the existing RFC standards were precise enough to support "without significant supplementation, a detailed proof of correctness in terms of invariants referenced in the specification". The protocols are significantly simpler: their model of RIP is (by a naïve line count) around 50 times smaller than the specification we present here.

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

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

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. Hofmann and Lemmen report on testing of a protocol stack based on an SDL specification of TCP/IP [HL00]. Billington and Han have produced a coloured Petri net model of the service provided by TCP for a highly idealised ISO-style interface [BH03]. 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 dealt with early Petri net protocol models [Pos74].

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, and 96 for Slow Start, Congestion Control, High Performance and SACK extensions.

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, allowable nondeterminism means they could not be used as oracles for conformance testing.

12 Required background and resource

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

The total effort required for the project, from our earliest experiments with UDP, has been perhaps 9 man-years over 3.5 calendar years. Of this, much has been devoted to idiom and tool development, and much to unpicking the intricacies of the existing TCP implementations. Similar work in future should be much less

arduous, especially if performed at design-time, and in any case this is a small amount of effort compared with that devoted to designing, implementing, and using such protocols.

13 Conclusion

13.1 Summary

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

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

The specification has been extensively annotated, aiming to make it usable as a reference by TCP/IP stack implementors, users of the Sockets API, and designers of protocol modifications. It is available (in a version automatically typeset from the HOL source) on the web [BFN+05c]. Also available from that link is a version of the 'TCP state diagram' based on the specification, which is rather more complete than the diagrams from RFC793 or the later Stevens texts.

13.2 Future Work

There are many possible directions for future work which hope to explore but have been unable to address within the PEPITO time frame — developing and validating the specification has spanned the entire project, and given the scale of the work as it has developed this is unsurprising. During the original proposal we hoped that more would be possible, but anticipated that realistically it might well not be; the project time frame was designed with this in mind.

There are three main future directions:

1. One can use and develop this specification, using it as a reference. Initial experiments on this line are reported in D1.2, with Compton's work on verifying properties of executable OCaml/Acute code above the earlier UDP specification.
2. One can use this specification as a basis for a higher-level end-to-end specification in terms of pseudo-reliable streams. This would be more directly of use to developers working above the Sockets API (and hence to most workers in P2P). Such an end-to-end specification was in fact our initial goal, but it quickly became clear that to understand the failure semantics in detail one would have to work at the segment level. Having established the current validated specification, however, it should be reasonably straightforward to abstract from it.
3. One can use these techniques —our HOL idioms, our symbolic testing tools, and perhaps our test instrumentation— to address new protocols. P2P protocols are a very interesting test case for this.

References

[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+05a] Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. Submitted for publication. Available at `http://www.cl.cam.ac.uk/users/pes20/Netsem/paper.pdf`, February 2005. 15pp.

[BFN+05b] Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 1: Overview. Available at `http://www.cl.cam.ac.uk/users/pes20/Netsem/tr.pdf`, February 2005. vi+96pp.

[BFN+05c] 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. Available at `http://www.cl.cam.ac.uk/users/pes20/Netsem/alldoc.pdf`, February 2005. xxv+380pp.

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

[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.

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

[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.

[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/`.

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

[IT01] IEEE and The Open Group. *IEEE Std 1003.1™-2001 Standard for Information Technology — Portable Operating System Interface (POSIX®)*. December 2001. Issue 6. Available `http://www.opengroup.org/onlinepubs/007904975/toc.htm`.

[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/`.

[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.

[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.

[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.

[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.

[Smi96] M. A. S. Smith. Formal verification of communication protocols. In *Proc. FORTE IX/PSTV 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.

[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.

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