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

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

[1] University of Cambridge
[2] NICTA

Abstract. Despite more then 30 years of research on protocol specification, the major protocols deployed in the Internet, such as TCP, are described only in informal prose RFCs and executable code. In part this is because the scale and complexity of these protocols makes them challenging targets for formalization.

In this paper we show how these difficulties can be addressed. We develop a high-level specification for TCP and the Sockets API, expressed in the HOL proof assistant, describing the byte-stream service that TCP provides to users. This complements our previous low-level specification of the protocol internals, and makes it possible for the first time to state what it means for TCP to be correct: that the protocol implements the service. We define a precise abstraction function between the models and validate it by testing, using verified testing infrastructure within HOL. This is a pragmatic alternative to full proof, providing reasonable confidence at a relatively low entry cost.

Together with our previous validation of the low-level model, this shows how one can rigorously tie together concrete implementations, low-level protocol models, and specifications of the services they claim to provide, dealing with the complexity of real-world protocols throughout.

1 Introduction

Real-world network protocols are usually described in informal prose RFCs, which inevitably have unintentional ambiguities and omissions, and which do not support conformance testing, verification of implementations, or verification of applications that use these protocols. Moreover, there are many subtly different realisations, including the TCP implementations in BSD, Linux, WinXP, and so on. The Internet protocols have been extremely successful, but the cost is high: there is considerable legacy complexity that implementors and users have to deal with, and there is no clear point of reference. To address this, we have developed techniques to put practical protocol design on a rigorous footing, to make it possible to specify protocols and services with mathematical precision, and to do verified conformance testing directly against those specifications. In this paper we demonstrate our approach by developing and validating a high-level specification of the service provided by TCP.

Our specification deals with the full complexity of the service provided by TCP (except for performance properties). It includes the Sockets API (`connect`,

`listen`, etc.), hosts, threads, network interfaces, the interaction with ICMP and UDP, abandoned connections, transient and persistent connection problems, unexpected socket closure, socket self-connection and so on. The specification comprises roughly 30 000 lines of (commented) higher-order logic, and mechanized tool support has been essential for work on this scale. It is written using the HOL system [11]. The bulk of the definition is an operational semantics, using idioms for timed transition relations, record-structured state, pattern matching and so on.

We relate this service-level specification to our previous protocol description by defining, again in HOL, an abstraction function from the (rather complex) low-level protocol states, with sets of TCP segments on the wire, flow and congestion control data, etc., to the (simpler) service-level states, comprising byte streams and some status information. This makes explicit how the protocol implements the service.

The main novelty of the approach we take here is the *validation* of this abstraction function. Ideally, one would *prove* that the abstraction relationship holds in all reachable states. Given the scale and complexity of the specifications, however, it is unclear whether that would be pragmatically feasible, especially with the limited resources of an academic team. Accordingly, we show how one can validate the relationship by verified testing. We take traces of the protocol-level specification (themselves validated against the behaviour of the BSD TCP implementation), and verify (automatically, and in HOL) that there are corresponding traces of the service-level specification, with the abstraction function holding at each point. Our previous protocol-level validation, using a special-purpose symbolic evaluator, produced symbolic traces of the protocol-level specification. We now *ground* these traces, using a purpose-built constraint solver to instantiate variables to satisfy any outstanding constraints, and use a new symbolic evaluator to apply the abstraction function and check that the resulting trace lies in the service-level specification. By doing this all within HOL, we have high confidence in the validation process itself.

Obviously, such testing cannot provide complete guarantees, but our experience with the kind of errors it detects suggests that it is still highly discriminating (partly due to the fact that it examines the internal states of the specifications at every step along a trace) and one can develop useful levels of confidence relatively quickly.

In the following sections, we first recall our previous protocol model (Sect. 2), before describing the new service-level specification (Sect. 3) and abstraction function (Sect. 4), giving small excerpts from each. We then discuss the validation infrastructure, and the results of validation (Sect. 5). Finally, we discuss related work and conclude.

2 Background: our previous low-level protocol model

Our previous low-level specification [5,6] characterises TCP, UDP and ICMP at the protocol level, including hosts, threads, the Sockets API, network inter-

faces and segments on the wire. As well as the core functionality of segment retransmission and flow control, TCP must handle details of connection setup and tear-down, window scaling, congestion control, timeouts, optional TCP features negotiated at connection setup, interaction with ICMP messages, and so on. The model covers all these. It is parameterized by the OS, allowing OS-dependent behaviour to be specified cleanly; it is also non-deterministic, so as not to constrain implementations unnecessarily.

This level of detail results in a model of roughly 30 000 lines of (commented) higher-order logic (similar in size to the implementations, but structured rather differently). As further evidence of its accuracy and completeness, it has been successfully used as the basis for a Haskell implementation of a network stack [13].

The main part of the protocol model (the pale shaded region below) 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 protocol model uses the host LTS, and a model of the TCP, UDP and ICMP segments on the wire, to describe a network of communicating hosts.
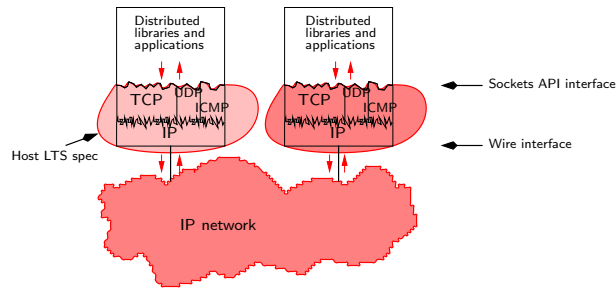
The host labelled transition relation, $h \xrightarrow{lbl} h'$, is defined by some 148 rules for the socket calls (5–10 for each interesting call) and some 46 rules for message send/receive and for internal behaviour. An example of one of the simplest rules is given in Fig. 1. The rule describes a host with a blocked thread attempting to send data to a socket. The thread becomes unblocked and transfers the data to the socket's send queue. The send call then returns to the user.

The transition $h \langle\!\langle ... \rangle\!\rangle \xrightarrow{\tau} h \langle\!\langle ... \rangle\!\rangle$ appears at the top, where the thread pointed to by *tid* and the socket pointed to by *sid* are unpacked from the original and final hosts, along with the send queue *sndq* for the socket. Host fields that are modified in the transition are highlighted. The initial host has thread *tid* in state SEND2, blocking attempting to send *str* to *sndq*. After the transition, *tid* is in state RET(OK...), about to return to the user with str'', the data that has not been sent, here constrained to be the empty string.

The bulk of the rule is the condition (a predicate) guarding the transition, specifying when the rule applies and what relationship holds between the input and output states. The condition is simply a conjunction of clauses, with no temporal ordering. The rule only applies if the state of the socket, *st*, is either ESTABLISHED or CLOSE_WAIT. Then, provided send_queue_space is large

send_3 **tcp: slow nonurgent succeed** **Successfully return from blocked state having sen t data**

$h \, (\!| ts := ts \oplus (tid \mapsto \boxed{(\text{SEND2}(sid, *, str, opts))_d} \,);$
    $socks := socks \oplus \quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$
      $\text{TCP\_Sock}(st, cb, *, \boxed{sndq}, \boxed{sndurp}, rcvq, rcvurp, iobc)))] |\!)$
$\xrightarrow{\tau}$
  $h \, (\!| ts := ts \oplus (tid \mapsto \boxed{(\text{RET}(\text{OK}(\mathbf{implode} \ str'')))_{\text{sched\_timer}}} \,);$
    $socks := socks \oplus \quad [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$
      $\text{TCP\_Sock}(st, cb, *, \boxed{sndq ++ str'}, \boxed{sndurp'}, rcvq, rcvurp, iobc)))] |\!)$

$st \ \in \{\text{ESTABLISHED}; \text{CLOSE\_WAIT}\} \wedge$
$space \ \in \ \text{send\_queue\_space}(sf.n(\text{SO\_SNDBUF}))$
    $(\mathbf{length} \ sndq)(\text{MSG\_OOB} \ \in \ opts)$
      $h.\text{arch} \ cb.t\_maxseg \ i_2 \wedge$
$space \geq \mathbf{length} \ str \wedge$
$str' = str \wedge str'' = [] \wedge$
$sndurp' = \mathbf{if} \ \text{MSG\_OOB} \ \in \ opts \ \mathbf{then} \ \uparrow(\mathbf{length}(sndq ++ str') - 1) \ \mathbf{else} \ sndurp$

HOL syntax   For optional data items, $*$ denotes absence (or a zero IP or port) and $\uparrow x$ denotes presence of value x. Concrete lists are written $[1, 2, 3]$ and appending two lists is written using an infix $++$. Records are written within angled brackets $(\!|...|\!)$. Record fields can be accessed by dot notation or by pattern-matching. Record fields may be overridden: $cb' = cb \, (\!| irs := seq |\!)$ states that the record cb' is the same as the record cb, except that field $cb'.irs$ has the value seq. The expression $f \oplus [(x, y)]$ or $f \oplus (x \mapsto y)$ denotes the finite map f updated to map x to y.

Fig. 1. Protocol-level model, example rule

enough, *str* is appended to the *sndq* in the final host. Lastly, the urgent pointer *sndurp'* is set appropriately.

Although the bulk of the model deals with the relatively simple Sockets API, with many rules like that of Fig. 1, the real complexity arises from internal actions that are largely invisible to the Sockets user, such as retransmission and congestion control. For example, the rule *deliver_in_3* (not shown) that handles normal message receipt comprises over 1 000 lines of higher-order logic.

The model has been validated against several thousand real-world network traces, designed to test corner cases and unexpected situations. Of these, 92% are valid according to the model, and we believe that for many purposes the model is sufficiently accurate — certainly enough to be used as a reference, in conjunction with the standard texts.

3   The new service-level specification

The service-level specification, illustrated below, describes the behaviour of a network of hosts communicating over TCP, as observed at the Socket APIs of the connections involved. It does not deal with TCP segments on the wire (though it necessarily does include ICMP and UDP messages).
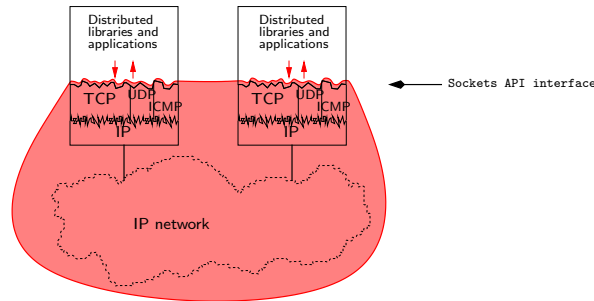
In principle one could derive a service-level specification directly from the protocol model, taking the set of traces it defines and erasing the TCP wire segment transitions. However, that would not give a *usable* specification: one in which key properties of TCP, that users depend on, are clearly visible. Hence, we built the service-level specification by hand, defining a more abstract notion of host state, an abstract notion of stream object, and a new network transition relation, but aiming to give the same Sockets-API-observable behaviour.

The abstract host states are substantially simpler than those of the protocol-level model. For example, the protocol-level TCP control block contains 44 fields, including retransmit and keep-alive timers; window sizes, sequence position and scaling information; timestamping and round trip times. Almost none of these are relevant to the service-level observable behaviour, and so are not needed in the service-level TCP control block.

Along with this, the transition rules that define the protocol dynamics, such as *deliver_in_3*, become much simpler. The rules that deal with the Sockets API must be adapted to the new host state, but they remain largely as before. The overall size of the specification is therefore not much changed, at around 30 000 lines (including comments).

A naive approach to writing the individual rules would be to existentially quantify those parts of the host state that are missing at the service level (and then to logically simplify as much as possible). However, this would lead to a highly non-deterministic and ultimately less useful specification. Instead, we relied on a number of invariants of the low-level model, arguing informally that, given those, the two behaviours match. We rely on the later validation to detect any errors in these informal arguments.

In the rest of this section we aim to give a flavour of the service-level specification, referring the interested reader to the complete specification online [23].

The heart of the specification is a model of a bidirectional TCP connection as a pair of unidirectional byte streams between Sockets endpoints:

— **unidirectional stream :**
tcpStream =⟨| i : ip; (* source IP *)
$\quad\quad\quad\quad$ p : port; (* source port *)
$\quad\quad\quad\quad$ *flgs* : streamFlags;
$\quad\quad\quad\quad$ *data* : byte list;
$\quad\quad\quad\quad$ *destroyed* : bool
$\quad\quad\quad$ |⟩

The data in the stream is a byte list. Further fields record the source IP address and port of the stream, control information in the form of flags, and a boolean indicating whether the stream has been destroyed at the source (say, by deleting the associated socket). Some of these fields are shared with the low-level specification, but others are purely abstract entities. Note that although a stream may be destroyed at the source, previously sent messages may still be on the wire, and might later be accepted by the receiver, so we cannot simply remove the stream when it is destroyed. Similarly, if the source receives a message for a deleted socket, a RST will typically be generated, which must be recorded in the stream flags of the destroyed stream. These flags record whether the stream is opening (SYN,$SYNACK$), closing normally (FIN) or abnormally (RST).

$-$ **stream control information :**
streamFlags $=(\!|$ SYN : bool; (* SYN, no ACK *)
$\qquad\qquad$ $SYNACK$: bool; (* SYN with ACK *)
$\qquad\qquad$ FIN : bool;
$\qquad\qquad$ RST : bool
$\qquad\qquad |\!)$

This control information is carefully abstracted from the protocol level, to capture just enough structure to express the user-visible behaviour. Note that the SYN and $SYNACK$ flags may be set simultaneously, indicating the presence of both kinds of message on the wire. The receiver typically lowers the stream SYN flag on receipt of a SYN: even though messages with a SYN may still be on the wire, subsequent SYNs will be detected by the receiver as invalid duplicates of the original. A bidirectional stream is then just an unordered pair (represented as a set) of unidirectional streams.

The basic operations on a byte stream are to read and write data. The following defines a write from Sockets endpoint (i_1, p_1) to endpoint (i_2, p_2).

$-$ **write flags and data to a stream :**
write$(i_1, p_1, i_2, p_2)(flgs, data)s$ $s' = ($
$\quad \exists in_-$ out in' out'.
$\quad$ sync_streams$(i_1, p_1, i_2, p_2)s(in_-, out) \wedge$
$\quad$ sync_streams$(i_1, p_1, i_2, p_2)s'(in', out') \wedge$
$\quad in' = in_- \wedge$
$\quad out'.flgs =$
$\quad (\!|$ $SYN :=(out.flgs.SYN \vee flgs.SYN)$;
$\quad\quad SYNACK :=(out.flgs.SYNACK \vee flgs.SYNACK)$;
$\quad\quad FIN :=(out.flgs.FIN \vee flgs.FIN)$;
$\quad\quad RST :=(out.flgs.RST \vee flgs.RST)$
$\quad |\!) \wedge$
$\quad out'.data = (out.data + + data))$

Stream s' is the result of writing $flgs$ and $data$ to stream s. Stream s consists of a unidirectional input stream in_- and output stream out, extracted from the bidirectional stream using the auxiliary sync_streams function. Similarly s', the

$send\_3$ **tcp: slow nonurgent succeed** **Successfully return from blocked state having sent data**

$$(h \ (\!|ts := ts \oplus (tid \mapsto \boxed{(\textsc{Send2}(sid, *, str, opts))_d} \ );$$
$$socks := socks \oplus [(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$$
$$\textrm{TCP\_Sock}(st, cb, *)))]\!|),$$
$$S_0 \oplus [(\textrm{streamid\_of\_quad}(i_1, p_1, i_2, p_2), \boxed{s} \ )], M)$$
$$\xrightarrow{\tau} \quad (h \ (\!|ts := ts \oplus (tid \mapsto \boxed{(\textsc{Ret}(\textrm{OK}(\mathbf{implode} \ str'')))_{sched\_timer}} \ );$$
$$socks := socks \oplus [(sid, \textsc{Sock}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$$
$$\textrm{TCP\_Sock}(st, cb, *)))]\!|),$$
$$S_0 \oplus [(\textrm{streamid\_of\_quad}(i_1, p_1, i_2, p_2), \boxed{s'} \ )], M)$$

$st \ \in \{\textrm{ESTABLISHED}; \textrm{CLOSE\_WAIT}\} \wedge$
$space \ \in \ \textrm{UNIV} \wedge$
$space \geq \mathbf{length} \ str \wedge$
$str' = str \wedge str'' = [] \wedge$
$flgs = flgs \ (\!| \ SYN := \mathbf{F}; SYNACK := \mathbf{F}; FIN := \mathbf{F}; RST := \mathbf{F}|\!) \wedge$
$\textrm{write}(i_1, p_1, i_2, p_2)(flgs, str')s \ s'$

Fig. 2. Service-level specification, example rule

state of the stream after the write, consists of in' and out'. Since we are writing to the output stream, the input stream remains unchanged, $in' = in_-$. The flags on the output stream are modified to reflect $flgs$. For example, SYN is set in $out'.flgs$ iff $flgs$ contains a SYN or $out.flgs$ already has SYN set. Finally, $out'.data$ is updated by appending $data$ to $out.data$.

Fig. 2 gives the service-level analogue of the previous protocol-level rule. The transition occurs between triples $(h \ (\!|...|\!), S_0 \oplus [...], M)$, each consisting of a host, a finite map from stream identifiers to streams, and a set of UDP and ICMP messages. The latter do not play an active part in this rule, and can be safely ignored. Host state is unpacked from the host as before. Note that protocol-level constructs such as $rcvurp$ and $iobc$ are absent from the service-level host state. As well as the host transition, there is a transition of the related stream s to s'. The stream is unpacked from the finite map via its unique identifier streamid_of_quad(i_1, p_1, i_2, p_2), derived from its quad.

As before, the conditions for this rule require that the state of the socket st must be ESTABLISHED or CLOSE_WAIT. Stream s' is the result of writing string str' and flags $flgs$ to s. Since $flgs$ are all false, the write does not cause any control flags to be set in s', although they may already be set in s of course.

This rule, and the preceding definitions, demonstrate the conceptual simplicity and stream-like nature of the service level. Other interesting properties of TCP are clearly captured by the service-level specification. For example, individual writes do not insert record boundaries in the byte stream, and in general, a read returns only part of the data, uncorrelated with any particular write. The model also makes clear that the unidirectional streams are to a large extent independent. For example, closing one direction does not automatically cause the other to close.
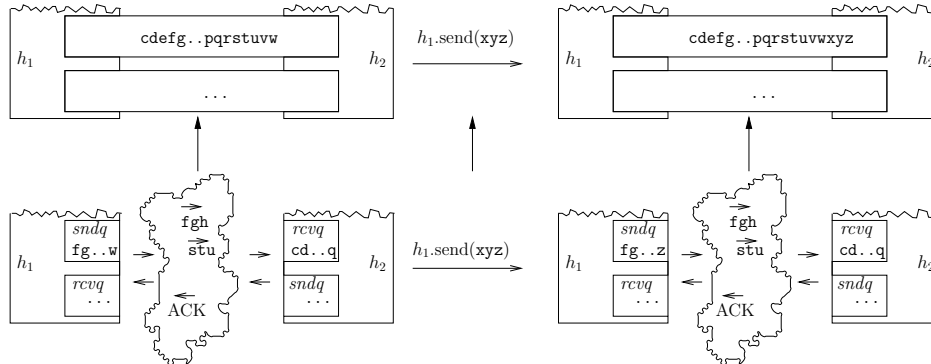
Fig. 3. Abstraction function, illustrated (data part only)

4 The abstraction function

While the service specification details *what* service an implementation of TCP provides to the Sockets interface, the abstraction function details *how*. The abstraction function maps protocol-level states and transitions to service-level states and transitions. A protocol-level network consists of a set of hosts, each with their own TCP stacks, and segments on the wire. The abstraction function takes this data and calculates abstract byte streams between Sockets API endpoints, together with the abstract connection status information.

The latter is the more intricate part, but we can give only a simple example here: the *destroyed* flag is set iff either there is no socket on the protocol-level host matching the quad for the TCP connection or the state of the TCP socket is CLOSED.

The former is illustrated in Fig. 3. For example, consider the simple case where communication has already been established, and the source is sending a message to the destination that includes the string "`abc...xyz`", of which bytes up to "`w`" have been moved to the source *sndq*. Moreover, the destination has acknowledged all bytes up to "`f`", so that the *sndq* contains "`fgh...uvw`", and *snd_una* points to "`f`". The destination *rcvq* contains "`cde...opq`", waiting for the user to read from the socket, and *rcv_nxt* points just after "`q`".

	$\downarrow snd\_una$ $\quad$ $\downarrow rcv\_nxt$
message	`...abcdefghijklmnopqrstuvwxyz...`
source *sndq*	`fghijklmnopqrstuvw`
destination *rcvq*	`cdefghijklmnopq`
DROP$(rcv\_nxt - snd\_una)sndq$	`rstuvw`
stream	`cdefghijklmnopqrstuvw`

The data that remains in the stream waiting for the destination endpoint to read, is the byte stream "`cdefghijklmnopqrstuvw`". This is simply the destination *rcvq* with part of the source *sndq* appended: to avoid duplicating the

– **unidirectional abstraction function :**

abs_hosts_one_sided$(i_1, p_1, i_2, p_2)(h, msgs, i) = ($

  (* messages that we are interested in, including *oq* and *iq* *)

  let $(hoq, iiq) =$

    case $(h.oq, i.iq)$ **of** $((msgs)_{\_1}, (msgs')_{\_2}) \rightarrow (msgs, msgs')$ **in**

  let $msgs = $ **list_to_set** $hoq \cup msgs \cup ($**list_to_set** $iiq)$ **in**

  (* only consider TCP messages ... *)

  let $msgs = \{msg \mid TCP \ msg \in msgs\}$ **in**

  (* ... that match the quad *)

  let $msgs = msgs \cap$

    $\{msg \mid msg = msg \ (\!| is_1 := \uparrow i_1; ps_1 := \uparrow p_1; is_2 := \uparrow i_2; ps_2 := \uparrow p_2 |\!)\}$ **in**

  (* pick out the send and receive sockets *)

  let $smatch \ i_1 \ p_1 \ i_2 \ p_2 \ s =$

    $((s.is_1, s.ps_1, s.is_2, s.ps_2) = (\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2))$ **in**

  let $snd\_sock = Punique\_range(smatch \ i_1 \ p_1 \ i_2 \ p_2)h.socks$ **in**

  let $rcv\_sock = Punique\_range(smatch \ i_2 \ p_2 \ i_1 \ p_1)i.socks$ **in**

  let $tcpsock_of \ sock = $ **case** $sock.pr$ **of**

    $TCP1\_hostTypes \ \$TCP\_PROTO \ tcpsock \rightarrow tcpsock$

    $\| \ \_3 \rightarrow ERROR$"abs_hosts_one_sided:tcpsock_of"

  in

  (* the core of the abstraction function is to compute *data* *)

  let $(data : $ byte list$) = $ **case** (snd_sock, rcv_sock) **of**

    $(\uparrow(\_8, hsock), \uparrow(\_9, isock)) \rightarrow ($

      let $htcpsock = tcpsock\_of \ hsock$ **in**

      let $itcpsock = tcpsock\_of \ isock$ **in**

      let $(snd\_una, sndq) = (htcpsock.cb.snd\_una, htcpsock.sndq)$ **in**

      let $(rcv\_nxt, rcvq) = (itcpsock.cb.rcv\_nxt, itcpsock.rcvq)$ **in**

      let $rcv\_nxt = tcp\_seq\_flip\_sense \ rcv\_nxt$ **in**

      let $sndq' = DROP(($**num**$(rcv\_nxt - snd\_una)))sndq$ **in**

      $rcvq + +sndq')$

    $\| \ (\uparrow(\_8, hsock), *) \rightarrow ($

      let $htcpsock = tcpsock\_of \ hsock$ **in**

      $htcpsock.sndq)$

    $\| \ (*, \uparrow(\_9, isock)) \rightarrow ($

      let $itcpsock = tcpsock\_of \ isock$ **in**

      let $(rcv_nxt : tcpLocal \ seq32, rcvq : $ byte list$) =$

        $(tcp\_seq\_flip\_sense(itcpsock.cb.rcv\_nxt), itcpsock.rcvq)$ **in**

      $rcvq + +($stream_reass $rcv\_nxt \ msgs))$

    $\| \ (*, *) \rightarrow ERROR$"abs_hosts_one_sided:data"

  in

$(\!| \ i := i_1;$

 $p := p_1;$

 $flgs :=$

 $(\!| \ SYN := (\exists msg.msg \in msgs \wedge msg = msg \ (\!| SYN := \mathbf{T}; ACK := \mathbf{F} |\!));$

   $SYNACK := (\exists msg.msg \in msgs \wedge msg = msg \ (\!| SYN := \mathbf{T}; ACK := \mathbf{T} |\!));$

   $FIN := (\exists msg.msg \in msgs \wedge msg = msg \ (\!| FIN := \mathbf{T} |\!));$

   $RST := (\exists msg.msg \in msgs \wedge msg = msg \ (\!| RST := \mathbf{T} |\!))$

 $|\!);$

 $data := data;$

 $destroyed := ($**case** snd_sock **of**

 $\uparrow(sid, hsock) \rightarrow ((tcpsock_of \ hsock).st = $ CLOSED$)$

 $\| \ * \rightarrow \mathbf{T})$

$|\!))$

Fig. 4. Abstraction function, excerpt

shared part of the byte sequence, $(rcv\_nxt - snd\_una)$ bytes are dropped from $sndq$ before appending it to $rcvq$.

An excerpt from the HOL definition appears in Fig. 4. It takes a quad (i_1, p_1, i_2, p_2) identifying the TCP connection, a source host h, a set of messages $msgs$ on the wire, and a destination host i, and produces a unidirectional stream. It follows exactly the previous analysis: $(rcv\_nxt - snd\_una)$ bytes are dropped from $sndq$ to give $sndq'$, which is then appended to $rcvq$ to give the data in the stream.

Note that, in keeping with the fact that TCP is designed so that hosts can retransmit any data that is lost on the wire, this abstraction does not depend on the data in transit — at least for normal connections in which neither endpoint has crashed.

For a given TCP connection, the full abstraction function uses the unidirectional function twice to form a bidirectional stream constituting the service-level state. As well as mapping the states, the abstraction function maps the transition labels. Labels corresponding to visible actions at the Sockets interface, such as a `connect` call, map to themselves. Labels corresponding to internal protocol actions, such as the host network interface sending and receiving datagrams from the wire, are invisible at the service level, and so are mapped to τ, indicating no observable transition. Thus, for each protocol-level transition, the abstraction function gives a service-level transition with the same behaviour at the Sockets interface. Mapping the abstraction function over a protocol-level trace gives a service-level trace with identical Sockets behaviour. Every valid protocol-level trace should map to a valid service-level trace.

5 Experimental validation

How can we ensure that TCP implementations (written in C), our previous protocol-level model (in HOL), and our new service-level specification (also in HOL) are consistent? Arguing that a small specification corresponds to a simple real-world system can already be extremely challenging. Here, we are faced with very large specifications and a very complex real-world system. Ideally one would verify the relationship between the protocol and service specifications by proving that their behaviours correspond, making use of the abstraction function. One would also prove that the Sockets behaviour of the endpoint implementations (formalized using a C semantics) conformed to the protocol model.

Proving the relationships between the levels in this way would be a very challenging task indeed. One of the main barriers is the scale of TCP implementations, including legacy behavioural intricacies of TCP and Sockets, which were not designed with verification in mind.

Hence, we adopt the pragmatic approach of validating the specifications to provide reasonable confidence in their accuracy. Note that for TCP the implementations are the de facto standard. In producing specifications after the fact, we aim to validate the specifications against the implementation behaviour. Our techniques could equally well be used in the other direction for new protocol de-

signs. Our service-level validation builds on our earlier protocol-level work [5,6], so we begin by recalling that.

Protocol-level validation   We instrumented a test network and wrote tests to drive hosts on the network, generating real-world traces. We then ensured that the protocol specification admitted those traces by running a special-purpose symbolic model checker in HOL, correcting the specification, and iterating, when we discovered errors. Because it is based directly on the formal specification, and deals with all the internal state of hosts, the checker is extremely rigorous, producing a machine checked proof of admissibility for each successfully validated trace. Obviously no testing-based method can be complete, but this found many issues in early drafts of the specification, and also identified a number of anomalies in TCP implementations.

Service-level validation   For the service-level validation of this paper, we began with a similar instrumented test network, but collected double-ended traces, capturing the behaviour of two interacting hosts, rather than just one endpoint. We then used our previous symbolic evaluation tool to discover symbolic traces of the protocol-level model that corresponded to the real-world traces. That is a complex and computationally intensive process, involving backtracking depth-first search and constraint simplification, essentially to discover internal host state and internal transitions that are not explicit in the trace.

We then *ground* these symbolic traces, finding instantiations of their variables that satisfy any remaining constraints, to produce a ground protocol-level trace in which all information is explicit. Given such a ground trace, we can map the abstraction function over it to produce a candidate ground service-level trace.

It is then necessary to check validity of this trace, which is done with a service-level test oracle. As at the protocol level, we wrote a new special-purpose service-level checker in HOL which performs symbolic evaluation of the specification with respect to ground service-level traces. Crucially, this checking process is much simpler than that at the protocol level because all host values, and all transitions, are already known. All that remains is to check each ground service-level transition against the specification.

The most significant difference between the old and new checkers is that the former had to perform a depth-first search to even determine which rule of the protocol model was appropriate. Because that work has already been done, and because the two specifications have been constructed so that their individual rules correspond, the service-level checker does not need to do this search. Instead, it can simply check the service-level version of the rule that was checked at the protocol level, dealing with each transition in isolation. In particular, this means that the service-level checker need not attempt to infer the existence of unobservable τ-transitions.

Another significant difference between the two checkers is that the service-level checker can aggressively search for instantiations of existentially quantified variables that arise when a rule's hypothesis has to be discharged. At the protocol level, such variables may appear quite unconstrained at first appearance, but

then become progressively more constrained as further steps of the trace are processed.

For example, a simplified rule for the socket call might appear as

$$\frac{fd \notin \mathsf{usedfds}(h_0)}{h_0 (\!| socks := socks |\!) \xrightarrow{tid \cdot \mathsf{socket}()} h_0 (\!| socks := socks \oplus (sid, fd) |\!)}$$

stating that when a socket call is made, the host h_0's socks map is updated to associate the new socket (identified by sid) with file-descriptor fd, subject only to the constraint that the new descriptor not already be in use. (This under-specification is correct on Windows; on Unix, the file-descriptor is typically the next available natural number.)

In the protocol-level checker, the fd variable must be left uninstantiated until its value can be deduced from subsequent steps in the trace. In the service-level checker, both the initial host and the final host are available because they are the product of the abstraction function applied to the previously generated, and ground, protocol trace. In a situation such as this, the variable from the hypothesis is present in the conclusion, and can be immediately instantiated.

In other rules of the service-level specification, there can be a great many variables that occur only in the hypothesis. These are existentially quantified, and the checker must determine if there is an instantiation for them that makes the hypothesis true. The most effective way of performing this check is to simplify, apply decision procedures for arithmetic, and to then repeatedly case-split on boolean variables, and the guards of if-then-else expressions to search for possible instantiations.

The above process is clearly somewhat involved, and itself would ordinarily be prone to error. To protect against this we built all the checking infrastructure within HOL. So, when checking a trace, we are actually building machine-checked proofs that its transitions are admitted by the inductive definition of the transition relation in the specification.

Results Our earlier protocol-level validation involved several thousand traces designed to exercise the behaviour of single endpoints, covering both the Sockets API and the wire behaviour. To produce a reasonably accurate specification, we iterated the checking and specification-fixing process many times.

For the service-level specification, we have not attempted the same level of validation, simply due to resource constraints. Instead, we have focused on developing the method, doing enough validation to demonstrate its feasibility. Producing a specification in which one should have high confidence might require another man-year or so of testing — perfectly feasible, and a tiny amount of effort in terms of industrial protocol stack development, but unlikely to lead to new research insights. That said, most of the Sockets API behaviour does not relate to the protocol dynamics and is common between the two specifications, so is already moderately well tested. In all, 30 end-to-end tests were generated, covering a variety of connection setup and tear-down cases and end-to-end com-

munication, but not including packet loss, reordering, duplication, and severe delay. After correcting errors, all these traces were found to validate successfully.

To illustrate how discriminating our testing process is, we mention two errors we discovered during validation. At the protocol-level, a TCP message moving from a host output queue to the wire corresponds to an unobservable τ event at the service level. Naively we assumed the host state would be unchanged, since the output queue at the service-level carries only ICMP and UDP messages. However, this is not correct, since the transmission of a TCP message alters the timer associated with the output queue, increasing its value. The update to the timer permits the host to delay sending the ICMP and UDP messages. Without this side-effect, the service-level specification effectively required ICMP and UDP messages to be sent earlier than they would otherwise have been. To correct this error, the service specification had to allow the timer to be updated if at the protocol-level there was potentially a TCP message on the queue that might be transferred to the wire. Another error arose in the definition of the abstraction function. The analysis of the merging of the send and receive queues on source and destination hosts, described in Sect. 4, was initially incorrect, leading to streams with duplicated, or missing, runs of data. Fortunately this error was easy to detect by examining the ground service-level trace, where the duplicated data was immediately apparent.

Our validation processes check that certain traces are included in the protocol-level or service-level specification. As we have seen, this can be a very discriminating test, but it does not touch on the possibility that the specifications admit too many traces. That cannot be determined by reference to the de facto standard implementations, as a reasonable specification here must be looser than any one implementation. Instead, one must consider whether the specifications are strong enough to be useful, for proving properties of applications that use the Sockets API, or (as in [13]) as a basis for new implementations.

6   Related work

This work builds on our previous TCP protocol model [5,6], and we refer the reader there for detailed discussion of related work. We noted that "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". This also applies to the service-level specification. As before, this is unsurprising: we have depended on automated reasoning tools and on raw compute resources that were simply unavailable in the 1980s or early 1990s. Our goals have also been different, and in some sense more modest, than the correctness theorems of traditional formal verification: we have not attempted to *prove* that an implementation of TCP satisfies the protocol model, or that the protocol satisfies the service-level specification.

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, FM, FORTE, ICNP, SPIN, and TACAS. The most detailed

rigorous specification of a TCP-like protocol we are aware of is that of Smith [22], 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. Later work by Smith and Ramakrishnan uses a similar model to verify properties of a model of SACK [21]. A variety of work addresses radically idealised variants of TCP [8,9,19,10,3,15,16]. Finally, Postel's PhD thesis used early Petri net protocol models descriptively [18].

Implementations of TCP in high-level languages have been written by Biagioni in Standard ML [2], by Castelluccia *et al.* in Esterel [7], and by Kohler *et al.* in Prolac [12]. As for any implementation, allowable non-determinism means they cannot be used as oracles for conformance testing.

For concurrent and distributed systems, there are many abstraction-refinement techniques, such as abstraction relations (which include our abstraction function) and simulation relations, see [14] for an overview. As an example of these techniques, Alur and Wang address the PPP and DHCP protocols [1]. For each they check refinements between models that are manually extracted from the RFC specification and from an implementation. Although these techniques are widely used in verification, to the best of our knowledge, they have never been applied previously to real-world protocols on the scale of TCP.

7   Conclusion

Summary   We presented a formal, mechanized, service-level specification of TCP, tackling the full detail of the real-world protocol. The specification is appropriate for formal and informal reasoning about applications built above the Sockets layer, and about the service that TCP and TCP-like protocols provide to the Sockets layer. The service-level specification stands as a precise statement of end-to-end correctness for TCP. We also presented a formal abstraction function from our previous protocol-level model of TCP to the service-level specification, thereby explaining how stream-like behaviour arises from the protocol level. We used novel validation tools, coupled with the results of previous work, to validate both the service specification and the abstraction function. The specification, abstraction function, and testing infrastructure were developed entirely in HOL.

On the practice of protocol design   This paper is the latest in a line of work developing rigorous techniques for real-world protocol modelling and specification [20,24,17,5,6,4]. In most of this work to date we have focused on post-hoc specification of existing infrastructure (TCP, UDP, ICMP, and the Sockets API) rather than new protocol design, though the latter is our main goal. This is for two reasons. Firstly, the existing infrastructure is ubiquitous, and likely to remain so for the foreseeable future: these wire protocols and the Sockets API are stable articulation points around which other software shifts. It is therefore well worth characterising exactly what they are, for the benefit of both users and implementers. Secondly, and more importantly, they are excellent test cases. There has been a great deal of theoretical work on idealised protocols, but, to develop

rigorous techniques that can usefully be applied, they must be tested with realistic protocols. If we can deal with TCP and Sockets, with all their accumulated legacy of corner cases and behavioural quirks, then our techniques should certainly be applicable to new protocols. We believe that that is now demonstrated, and it is confirmed by our experience with design-time formalisation and conformance testing for an experimental MAC protocol for an optically switched network [4].

In recent years there has been considerable interest in 'clean slate' networking design, and in initiatives such as FIND and GENI. Protocols developed in such work should, we argue, be developed as trios of running implementation, rigorous specification, and verified conformance tester between the two. Modest attention paid to this at design time would greatly ease the task — for example, specifying appropriate debug trace information, and carefully identifying the deterministic parts of a protocol specification, would remove the need for backtracking search during validation. Declarative specification of the intended protocol behaviour, free from the imperative control-flow imposed by typical implementation languages, enables one to see unnecessary behavioural complexities clearly. Verified conformance testing makes it possible to keep implementations and specifications in sync as they are developed. Together, they should lead to cleaner, better-understood and more robust protocols, and hence to less costly and more robust infrastructure.

More specifically to TCP, we see two main directions for future work. One is simply to scale up our validation process, covering a wide variety of common protocol stacks, increasing confidence still further by testing against more traces, identifying and testing additional invariants of connection states, and so forth, and producing a packaged conformance tester for TCP implementations. This would be useful, and on an industrial scale would be a relatively small project (compared, perhaps, to the QA effort involved in developing a new protocol stack), but doing this for an existing protocol may be inappropriate for a small research group. The weight of legacy complexity here is very large, so non-trivial resources (perhaps several man-years) would be needed to cope with the detail, but the basic scientific questions, of *how* to do this, have now been solved. Doing this for *new* protocols, on the other hand, seems clearly worthwhile, even with very limited resources.

The second, more research-oriented, question, is to consider not just validation of end-to-end functional correctness (as we have done here), but properties such as end-to-end performance. Ultimately one could envisage proving network-wide properties, such as network stability, thereby connecting highly abstract properties of these protocols to the low-level details of their implementations.

References

1. R. Alur and B.-Y. Wang. Verifying network protocol implementations by symbolic refinement checking. In *Proc. CAV '01, LNCS 2102*, pages 169–181, 2001.
2. E. Biagioni. A structured TCP in Standard ML. In *Proc. SIGCOMM '94*, 1994.
3. J. Billington and B. Han. On defining the service provided by TCP. In *Proc. ACSC: 26th Australasian Computer Science Conference*, Adelaide, 2003.
4. A. Biltcliffe, M. Dales, S. Jansen, T. Ridge, and P. Sewell. Rigorous protocol design in practice: An optical packet-switch MAC in HOL. In *Proc. ICNP*, Nov. 2006.
5. S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proc. SIGCOMM 2005*, Aug. 2005.
6. S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proc. POPL*, 2006.
7. C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. *IEEE/ACM Trans. Netw.*, 5(4):514–524, 1997.
8. 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.
9. E. Fersman and B. Jonsson. Abstraction of communication channels in Promela: A case study. In *Proc. 7th SPIN Workshop, LNCS 1885*, pages 187–204, 2000.
10. R. Hofmann and F. Lemmen. Specification-driven monitoring of TCP/IP. In *Proc. 8th Euromicro Workshop on Parallel and Distributed Processing*, Jan. 2000.
11. The HOL 4 system, Kananaskis-3 release. `hol.sourceforge.net`.
12. E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable TCP in the Prolac protocol language. In *Proc. SIGGCOMM '99*, pages 3–13, August 1999.
13. P. Li and S. Zdancewic. Combining events and threads for scalable network services. In *Proc. PLDI*, pages 189–199, 2007.
14. N. Lynch and F. Vaangdrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, Sept. 1995.
15. S. L. Murphy and A. U. Shankar. A verified connection management protocol for the transport layer. In *Proc. SIGCOMM*, pages 110–125, 1987.
16. S. L. Murphy and A. U. Shankar. Service specification and protocol construction for the transport layer. In *Proc. SIGCOMM*, pages 88–97, 1988.
17. M. Norrish, P. Sewell, and K. Wansbrough. Rigour is good for you, and feasible: reflections on formal treatments of C and UDP sockets. In *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 49–53, Sept. 2002.
18. J. Postel. *A Graph Model Analysis of Computer Communications Protocols*. University of California, Computer Science Department, PhD Thesis, 1974.
19. I. Schieferdecker. Abruptly-terminated connections in TCP. In *Proc. Int. Workshop on Applied Formal Methods In System Design*, 1996.
20. A. Serjantov, P. Sewell, and K. Wansbrough. The UDP calculus: Rigorous semantics for real networking. In *Proc. TACS 2001*, Oct. 2001.
21. M. A. Smith and K. K. Ramakrishnan. Formal specification and verification of safety and performance of TCP selective acknowledgment. *IEEE/ACM Trans. Netw.*, 10(2):193–207, 2002.
22. M. A. S. Smith. Formal verification of communication protocols. In *Proc. FORTE IX/PSTV XVI*, 1996.
23. The Netsem Project. Web page. `http://www.cl.cam.ac.uk/users/pes20/Netsem/`.
24. K. Wansbrough, M. Norrish, P. Sewell, and A. Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. In *Proc. ESOP, LNCS 2305*, 2002.