Timing UDP: Mechanized Semantics for Sockets, Threads, and Failures

Keith Wansbrough, Michael Norrish, Peter Sewell, and Andrei Serjantov

Computer Laboratory, University of Cambridge

{First.Last}@cl.cam.ac.uk
www.cl.cam.ac.uk/users/pes20/Netsem

Abstract. This paper studies the semantics of failure in distributed programming. We present a semantic model for distributed programs that use the standard sockets interface; it covers message loss, host failure and temporary disconnection, and supports reasoning about distributed infrastructure. We consider interaction via the UDP and ICMP protocols. To do this, it has been necessary to: • construct an experimentally-validated post-hoc specification of the UDP/ICMP sockets interface; • develop a timed operational semantics with threads, as such programs are typically multithreaded and depend on timeouts; • model the behaviour of partial systems, making explicit the interactions that the infrastructure offers to applications; • integrate the above with semantics for an executable fragment of a programming language (OCaml) with OS library primitives; and • use tool support to manage complexity, mechanizing the model with the HOL theorem prover. We illustrate the whole with a module providing naïve heartbeat failure detection.

1 Introduction

Distributed systems are – almost by definition – concurrent and subject to partial failure; many are also subject to malicious attack. This complexity makes it hard to achieve a clear understanding of their behaviour based only on informal descriptions, in turn making it hard to build robust systems. This paper reports on work towards a rigorous treatment of distributed programming. We have constructed a operational semantics which makes it possible to reason about distributed programs, written in general-purpose programming languages, using standard communication primitives, and in the presence of failure. Developing a model that covers enough of the distributed phenomena (sufficiently accurately) to do this has required a number of problems to be addressed; we introduce them below, sketching our contribution to each.

As a preliminary, we must select the communication abstractions to consider. Interactions between machines can be viewed at many levels. We are primarily interested in the abstractions provided by the standard TCP, UDP and ICMP protocols above IP, for two reasons. Firstly, they are ubiquitous: almost all distributed interaction is ultimately mediated by them. More particularly, we want a model that accurately reflects the information about failure that is available to the application programmer – at the level of these protocols, the failure behaviour can be seen clearly. This should provide a solid basis for the design, verification and implementation of higher-level distributed abstractions. To investigate feasibility and techniques, we consider unicast UDP (providing unreliable asynchronous messages) and the associated ICMP messages (providing various error reporting); we do not touch on the more complex TCP (providing reliable streams). The protocols themselves are defined in RFCs [Pos80,Pos81,Bra89].

1.1 Sockets and Experimental Semantics An application programmer must understand not only the protocols, which for UDP and ICMP are relatively simple, but also the *sockets interface* [CSR83,IEE00,Ste98] to the operating system code that implements them. The behaviour of this interface is complex and not well documented (to the best of our knowledge there exist only informal natural-language documents, covering common behaviour but not precise, complete or correct). It is not feasible to analyse the sockets code and hence derive a semantics, nor is it feasible to alter the widely-deployed implementations. We must therefore produce a post-hoc specification with an *experimental semantics* approach: experimentally determining the behaviour of particular implementations.

1.2 Failure and Time Addressing failure requires two things. Firstly, we must model the actual failures – in this paper, we consider message loss and duplication, crash failure of hosts, and connection/disconnection of hosts from the network. More interestingly, we must be able to reason about the behaviour of programs that cope with failure. UDP communication is asynchronous, so these programs typically use timeouts, *e.g.* in calls to select. To model these accurately we use a *timed* operational semantics, involving time bounds for certain operations. Some operations have both a lower and upper bound (message propagation); some must happen immediately (recvfrom must return as soon as a message arrives); and some have an upper bound but may occur arbitrarily quickly (an OS return). For some of these requirements time is essential, and for others time conditions are simpler and more tractable than the corresponding fairness conditions [LV96, §2.2.2]. We draw on earlier work on timed automata [SGSAL98] and process calculi here, but have kept the semantics as lightweight as possible – in part, by building in a *local receptiveness* property.

1.3 Infrastructure Properties: Partial Systems and Threads We are particularly interested in implementations of distributed infrastructure (or middleware), rather than complete distributed systems, as a rigorous approach should be more fruitful in the former. This means that the semantics must be able to describe the behaviour of *partial* systems, consisting of a module that provides some abstraction to higher-level application code (*e.g.* a library for 'reliable' communication), instantiated on many machines. Interesting infrastructure usually also requires intra-machine concurrency in the form of *threads* (at minimum, one for the infrastructure and one for the application). Our model includes threads and simple modules, making explicit the possible interactions offered to a distributed application by per-machine instances of an infrastructure module.

1.4 Executable Code: Language Independence and MiniCaml We aim to reason about executable code, written in general-purpose programming languages. This contrasts with work on distributed algorithm verification, in which algorithms are usually described in pseudocode or in automata or calculi tuned for verification; it should reduce the 'semantic gap' between verified algorithm and actual system. Most of what we model, however, is language-independent. We therefore factor the semantics, regarding infrastructure modules as labelled transition systems (LTSs) of a certain form. The standard operational semantics of a variety of languages can be extended to give such LTSs. We do so for *MiniCaml*, a fragment of OCaml [L^+01]. MiniCaml's *types* include the standard built-in bool, int, string, tuples, lists, references, exceptions, and functions, together with types required for networking (e.g., fd, ip, port, *etc.*) The constructors, values, expressions and patterns are as one might expect, as are the typing rules. The dynamic semantics extends a standard operational semantics with labelled transitions for system calls, and by specifying the behaviour of modules. We have implemented an OCaml module that provides exactly the system calls of the model, so MiniCaml programs can be compiled with the standard `ocamlopt` compiler.

1.5 Semantic Complexity and HOL Mechanization As one can imagine, the need to deal simultaneously with sockets, failure, time, modules and threads has led to large definitions. The most complex part, for sockets, has been validated experimentally. To keep the whole internally consistent, we resort to automated tools. The entire definition (except for the MiniCaml semantics) has been expressed in the HOL theorem proving system [GM93], which we are using to check various sanity properties. The HOL and MiniCaml code in this paper has been automatically typeset from the sources using special-purpose tools. Mechanization identified a number of errors in earlier drafts of the semantics. The process has also been a useful stress-test of the HOL implementation.

1.6 Overview The remainder of this paper contains a brief introduction to UDP sockets, outlines the static and dynamic structure of the model, discusses its experimental validation and HOL mechanization, and analyses a simple heartbeat example in MiniCaml. Most details are perforce omitted; they will be in a forthcoming technical report. The HOL definitions are available electronically [WNSS01]. This work is a continuation of that reported in [SSW01a,SSW01b], which did not address time, threads, modules or mechanization.

2 Background

2.1 The Protocols At the level of abstraction of our model, a network consists of a number of machines connected by a combination of LANs (*e.g.* ethernets) and routers. Each machine has one or more *IP addresses* i, which are 32-bit values such as 192.168.0.11. The *Internet Protocol* (IP) allows one machine to send messages (*IP datagrams*) to another, specifying the destination by one of its IP addresses. IP datagrams have the form $\mathrm{IP}(i_1, i_2, \mathit{body})$, where i_1 and i_2 are the source and destination addresses. The implementation of IP is responsible for delivering the datagram to the correct machine; it abstracts from routing and network topology. Delivery is asynchronous and unreliable – IP does not provide acknowledgments that datagrams are received, or retransmit lost messages. Messages may be duplicated.

The *User Datagram Protocol* (UDP) is a thin layer above IP that provides multiplexing. It associates a set $\{1, .., 65535\}$ of *ports* with each machine; a UDP datagram is an IP datagram with body $\mathrm{UDP}(ps_1, ps_2, data)$, containing a source and destination port and a short sequence of bytes of *data*.

The *Internet Control Message Protocol* (ICMP) is another thin layer above IP dealing with some control and error messages. Here we are concerned only with two, relating to UDP, with bodies: $\mathrm{ICMP\_PORT\_UNREACH}(is_3, ps_3, is_4, ps_4)$ and $\mathrm{ICMP\_HOST\_UNREACH}(is_3, ps_3, is_4, ps_4)$. The first may be generated by a machine receiving a UDP datagram for an unexpected port; the second is sometimes generated by routers on receiving unroutable datagrams. They contain the IP addresses and ports of the original datagram.

2.2 Sockets The OS protocol endpoint code in each host maintains a collection of *sockets*: data structures that we write

$$\textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)$$

which mediate between application threads and the asynchronous message delivery activities. The file descriptor fd uniquely identifies this socket within the host. The IP addresses and ports is_1, ps_1 and is_2, ps_2 are a pair of 'local' and 'remote' pairs, some elements of which may be wildcards; the 4-tuple is used for addressing outgoing datagrams and matching incoming datagrams. The flag es stores any pending error condition, while the flags f hold an assortment of socket options. Finally, the message queue mq holds incoming messages that have been delivered by the OS to this socket but not yet received by the application.

The standard *sockets interface* [CSR83,IEE00,Ste98] is the library interface made available to applications. It includes calls socket and close for creating and closing sockets; bind and connect, for manipulating the local and remote pairs of IP addresses and ports; sendto and recvfrom, for sending and receiving messages; and select, allowing an application to block until either a timeout occurs or a file descriptor is ready for reading or writing. To avoid dealing with the uninteresting complexities of the standard C sockets interface, we introduce a thin abstraction layer that provides a clean strongly-typed view of the C sockets interface without sacrificing useful functionality. The model is expressed in terms of this interface, which we call LIB and present in Appendix A. To allow MiniCaml programs to be executed, we also implement the interface as a thin layer above the OCaml socket and thread libraries.

There are many behavioural subtleties which the model covers but which we cannot describe here, including: wildcard and loopback IP addresses; wildcard, privileged and ephemeral ports; blocking and non-blocking sendto, recvfrom and select; local errors; and multiple interfaces. These are discussed in detail in

```
val start_heartbeat_k : () → ()
val start_heartbeat_a : () → int ref
val get_status : int ref → int
```

```
(* code for player Kurt *)
let start_heartbeat_k() =
  let sender_thread() =
    let p = port_of_int (7658) in
    let i_a = ip_of_string (" 192.168.0.14 ") in
    let fd = socket() in
    let _ = bind(fd, *, *) in
    let _ = connect(fd, i_a, ↑p) in
    while TRUE do
      try
        sendto(fd, *, " ping ", FALSE);
        delay1000000;
      with
        UDP(ECONNREFUSED) → ()
    done in
  let t = create sender_thread() in ()
```

```
(* code for player Alan *)
let start_heartbeat_a() =
  let status_ref = ref 0 in
  let receiver_thread() =
    let p = port_of_int (7658) in
    let i_k = ip_of_string (" 192.168.0.11 ") in
    let fd = socket() in
    let _ = bind(fd, *, ↑p) in
    let _ = connect(fd, i_k, *) in
    while TRUE do
      let (fds, _) = select([fd], [ ], ↑2500000) in
      if fds = [ ] then
        status_ref := 0
      else
        let (_, _, _) = recvfrom(fd, FALSE) in
        status_ref := 1
    done in
  let t = create receiver_thread() in
  status_ref

let get_status status_ref =
  !status_ref
```

Fig. 1. `rhbeat2.mli` and `rhbeat2.ml`. The $*$ and $\uparrow$ are constructors of option types $T\uparrow$; `unit` is typeset as ().

[SSW01a]. Here, we shall highlight only the existence of asynchronous errors: a machine receiving a UDP datagram addressed to a port that does not have an associated socket may send back an ICMP_PORT_UNREACH message to the sender. This error message is received *asynchronously*—the sendto that nominally caused the error has (in general) long since returned to the application, and so some means of notification must be found. The sockets interface solves this problem by storing the last such error in the socket, returning it to the application whenever a subsequent communication operation (which may be quite unrelated) is attempted on that socket. The operation will fail but the error will be cleared, allowing subsequent operations to succeed.

2.3 Example Figure 1 gives a simple example of the kind of program which our model allows us to reason about. It is a MiniCaml module that provides a failure-detection service for two machines, using a naïve heartbeat algorithm. The *start_heartbeat_k* function should be called by an application running on machine KURT. It spawns a thread that creates a socket, sets its remote address to that of the other machine (192.168.0.14) and its remote port to an agreed value (7658), and then repeatedly sends "ping" messages, with a 1-second delay between each. The *start_heartbeat_a* function, to be called by an application on machine ALAN, creates a reference cell *status_ref* to hold its current guess of the status of KURT. It then spawns a *receiver_thread* and returns the reference cell. The thread creates a socket, sets its local port to the agreed 7658, and repeatedly waits for up to 2.5 seconds for a "ping" message. If it receives one, it sets the status to 1 to indicate that KURT is believed to be up (running and connected to the network), otherwise it sets it to 0. The application on machine ALAN can check the status of KURT by calling *get_status*, passing it *status_ref*. We are using OCaml's safe shared-memory communication between the ALAN threads.

3 The Model

This section outlines the main design choices and the static structure of the model. Discussion of the host semantics, which captures the behaviour of the library calls and UDP-related part of the operating system, is deferred to §4.

3.1 Overall Structure A network N is a parallel composition of UDP and ICMP messages in transit (on the wire, or buffered in routers) and of machines. Each machine comprises several *host components hc* – the OS state, a module, the states of threads, the store, *etc.* To simplify reasoning we bring all these components into the top-level parallel composition, maintaining the association between the components of a particular machine by tagging them with *host names n* (not to be confused with IP addresses or DNS names). Networks terms are therefore:

$N ::=$	0	empty network
	$N \mid N$	parallel composition
	msg_d	IP datagram in transit
	$n{\cdot}hc$	component of machine n

while host components, to be explained below, are of the forms:

$hc ::=$	$\text{HOST}(conn, h)$	OS state
	$\text{MODULE}(t)$	module code
	$\text{THREAD}(tid, org, t)_d$	running thread snippet
	$\text{THREADCREATE}(tid, org, t, t')_d$	pending create
	$\text{STORE}(st)$	shared store
	$\text{STORERET}(tid, tlty, v)_d$	pending return

Networks are subject to a well-formedness condition, network_ok N, which requires that no two machines share a host name n or non-loopback IP address, that each machine has exactly one store, and that each host component satisfies its own well-formedness condition, written host_ok h *etc.* We omit the details.

The semantics of a network is defined as a labelled transition system of a certain form. It uses three kinds of labels: labels that engage in binary CCS-style synchronisations, *e.g.* for a call of a host LIB routine by a thread; labels that do not synchronise, *e.g.* for τ actions resulting from binary synchronisations; and labels on which all terms must synchronise, used for time passing, hosts crashing and programs terminating. Parallel composition is defined using a single synchronisation algebra to deal with all of these, and we also use a non-standard restriction on the visible traces of the entire system, to force certain synchronisations to occur. The model is in some respects a nondeterministic loose specification, abstracting from some details, such as the relative precedence of competing errors, that should not be depended upon.

In contrast to standard process calculi we have a *local receptiveness* property: in any reachable state, if one component can do an output on a binary-sync label then there will be a unique possible counterpart, which is guaranteed to offer an input on that label. This means the model has no local deadlocks (though obviously threads can block waiting for a slow system call to return).

3.2 Threads and Modules In order to express and reason about the semantics of infrastructure code, such as the `rhbeat2` module, we must define the semantics of a partial system, exposing the interactions that an application program could have with such a module (and with the OS and store). Threads complicate the problem, as we must deal with *external thread snippets*, executing some module routine that has been called by an application thread, and with *internal* thread snippets, spawned by a module routine calling create directly. External snippets may return a value or exception to the application, whereas internal snippets do not return.

Thread snippets are written $\textsc{Thread}(tid, org, t)_d$, with a thread id tid, org either $\textsc{Extern}$ or $\textsc{Intern}$, state t, and timer d (see §3.4). During thread creation there is a transient state $\textsc{ThreadCreate}(tid, org, t, t')_d$.

To keep as much as possible of the model language-independent, the behaviours of modules $\textsc{Module}(t)$, and the resulting states t of thread snippets, are taken to be arbitrary labelled transition systems satisfying various sanity conditions, *e.g.* that a thread cannot simultaneously call two OS routines. This permits automata-theoretic descriptions of infrastructure algorithms, when convenient. To allow reasoning about executable code, though, we can use our MiniCaml semantics to derive such an LTS from any MiniCaml source program.

3.3 Interactions In more detail, the interactions between network terms are as follows. The external application can call a routine provided by the module. A thread snippet will then be spawned off; ultimately this may return a value or an exception. Thread snippets (and the external application) can call host LIB routines, which may later return. A special case is a call to create, which will both create a new thread ID and spawn off a thread snippet with that ID. Another special case is a call to exit, which will terminate all the threads of the host and close any sockets they have opened. Thread snippets (and the external application) can call the store operations new, set and get, which will quickly return. Hosts can send and receive IP datagrams. Hosts and thread snippets can perform internal computation. Hosts can crash, whereupon all their components are removed, can be disconnected from the network, and can be reconnected. Hosts can output strings on their console. Time can pass.

3.4 Time Time passage is modelled by transitions labelled $d \in \mathbb{R}_{>0}$ interleaved with other transitions. These labels uses multiway synchronisation, modelling global time which passes uniformly for all participants (although it cannot be accurately observed by them).

Our semantics is built using *timers*, variables containing elements of $\mathbb{R}_{\geq 0} \cup \{\infty\}$ that decrement uniformly as time passes (until zero); the state becomes *urgent* as soon as any timer in it reaches zero. Speaking loosely, only binary-synchronising output actions are constrained by timers. Urgent states are those in which there is a discrete action which should occur immediately. This is modelled by prohibiting time passage steps d from (or through) an urgent state. We have carefully arranged the model to avoid pathological timestops by ensuring the local receptiveness property holds.

Our model has a number of timing parameters: the minimum message propagation delay *dpropmin* and the maximum scheduling delay *dsch*, outqueue scheduling delay doq, store access delay *dstore*, thread evaluation step duration *dthread*, and message propagation delay *dpropmax*.

Many timed process algebras enforce a *maximal progress* property [Yi91], requiring that any action must be performed immediately it becomes enabled. We choose instead to ensure timeliness properties by means of timers and urgency. Our reasoning using the model so far involves only finite trace properties, so we do not need to impose Zeno conditions.

3.5 Messages/Networks Message propagation through the network is defined by the rules below.

$$\begin{array}{rcl@{\qquad}l@{\qquad}l}
0 & \xrightarrow{n \cdot msg} & msg_d & d \in [dpropmin,\ dpropmax] & net\_accept\_single \\
msg_{d+d'} & \xrightarrow{d'} & msg_d & d' > 0 & net\_msg\_time \\
msg_0 & \xrightarrow{\overline{n \cdot msg}} & 0 & & net\_emit \\
0 & \xrightarrow{n \cdot msg} & 0 & & net\_accept\_drop
\end{array}$$

A message sent by a host is accepted by the network with one of three rules. The normal case is *net_accept_single*, which places the message on the network with a timer d attached. The timer is initialised with the propagation delay, chosen nondeterministically. Message propagation is modelled simply by time passage: the rule *net_msg_time* decrements the timer until it reaches zero, making the state urgent. The delivery rule *net_emit* is thus forced to fire at exactly the instant the message arrives. Once the message arrives, it may be emitted by the network to a listening host by *net_emit*. This rule is only enabled at the instant the timer reaches zero, modelling the fact that the host has no choice over when it receives the message. Note that the network rules do not examine the message in any way – it is the host LTS that checks whether the IP address is one of its own. Time aside, this treatment of asynchrony is similar to Honda and Tokoro's asynchronous π-calculus [HT91]. Messages in the network may be reordered, and this is modelled simply by the nondeterministic propagation times. They may also be finitely duplicated, or lost. Rule *net_accept_dup* (not shown) is similar to *net_accept_single* above except that it yields $k \geq 2$ copies of the message, each with an independently-chosen propagation delay; rule *net_accept_drop* simply absorbs the message.

3.6 Stores A store $\textsc{Store}(st)$ has a state st which is simply a finite map from (typed) locations to values. It can receive new, set and get labels, spawning off a $\textsc{StoreRet}(tid, tlty, v)_{dstore}$ which will return the value v (of type $tlty$) to thread tid within time $dstore$. For simplicity, as the MiniCaml types are not all embedded into the HOL model, the store is restricted to first-order values.

3.7 Hosts A host $\textsc{Host}(conn, h)$ has a boolean $conn$, indicating whether it is currently connected to the network, and a host state h, a record modelling the relevant aspects of the OS state. The $h.ifds$ field is a set of interfaces, each with a set of IP addresses and other data. We assume all hosts have at least a loopback interface and one other. We sometimes write $i \in h.ifds$ for $\exists ifd.i \in ifd.ipset \wedge ifd \in h.ifds$. The operating system's view of the state of each thread is stored in a finite map $h.ts$ from thread identifiers tid to host thread states. Each thread may be running (Run), exiting (Exit), or waiting for the OS to return from a call. In the last case, the OS may be about to return a value ($\textsc{Ret}v$) or the thread may be blocked waiting for a slow system call to complete ($\textsc{Sendto2}v$, $\textsc{Recvfrom2}v$, $\textsc{Select2}v$, $\textsc{Delay2}$, $\textsc{Print2}v$, Zombie). The host's current list of sockets is stored in $h.s$. The *outqueue*, a queue of outbound IP messages, is given by $h.oq$ and $h.oqf$, where $h.oq$ is the list of messages (with a timer) and $h.oqf$ is set when the queue is full. In HOL syntax, the record (h with $\langle\!|ts := ts|\!\rangle$) is the record h with the ts field replaced by the value ts.

4 The Model Continued: Host Dynamics

The host semantics is defined by 78 host transition axioms, encoding the precise behaviour of each of the library calls in Figure 2 and of the UDP subsystem of the operating system. To give a flavour, we examine some of the behaviour of the heartbeat example program of §2.3, explaining the main points of a few rules.

We first consider KURT's execution of *sender_thread*(). Once the thread has converted the port and IP address, it calls socket() to allocate a new socket. The thread performs the call by making an output transition $\cdot \xrightarrow{\overline{tid \cdot \mathrm{socket}()}} \cdot$, where tid is the thread ID of the sender thread, and the host synchronises by making the corresponding input transition according to rule *socket_1*:

socket_1 **<u>succeed</u>**

$$h \text{ with } ts := ts \oplus (tid \mapsto \textsc{Run}_d)$$

$$\xrightarrow{tid \cdot (\mathrm{socket}())}$$

$$h \text{ with } \langle\!| ts := ts \oplus (tid \mapsto \textsc{Ret}(\mathrm{OK}fd)_{dsch}); \quad s := (\textsc{Sock}(fd, *, *, *, *, *, \textsc{Flags}(\mathbf{F}, \mathbf{F}), [\,]) :: h.s) |\!\rangle$$

$$fd \notin \mathrm{sockfds}\, h.s$$

Each rule is of the form "$h \xrightarrow{l} h'$ where *cond*", where h, h' are host states and l is a host transition label. The rules have been automatically typeset from the HOL source (see §6). In *socket_1* the initial host state (above the arrow) requires only that the host thread state for the thread is Run_d for some d; Run means the host is waiting for a call from the thread (in reachable states, the timer on a Run will always be ∞). The side condition (given below the transition itself) states that *fd* is some file descriptor not in the set of file descriptors already used in $h.s$. The final host state (below the arrow) updates $h.s$ by adding a freshly initialised socket with the chosen *fd* to the list, and sets the host thread state for the thread to $\text{Ret}(\text{OK}\mathit{fd})_{dsch}$. This will cause the host to return the value $\text{OK}\mathit{fd}$ to the thread within delay *dsch*, by rule *ret_1* (not shown). Unlike *net_emit* above, this may occur at any time from 0 up until *dsch*; this models a nondeterministic scheduler. The rules are partitioned into several classes; the **succeed** indicates which class *socket_1* belongs to.

We omit the bind and connect calls, and proceed to the top of the while loop, where the application invokes sendto($\mathit{fd}, *,$ "ping", $\mathbf{F}$). Assuming there is room on the outqueue for the message, rule *sendto_1* fires:

sendto_1 **succeed** **autobinding**

$$\begin{array}{l}
h \text{ with } \langle\!|\mathit{ts} := \mathit{ts} \oplus (\mathit{tid} \mapsto \text{Run}_d);\\
\qquad\quad s := SC(s \text{ with } \mathit{es} := *)|\!\rangle\\
\xrightarrow{\mathit{tid}\cdot \text{sendto}(s.\mathit{fd},\mathit{ips},\mathit{data},\mathit{nb})}\\
h \text{ with } \langle\!|\mathit{ts} := \mathit{ts} \oplus (\mathit{tid} \mapsto \text{Ret}(\text{OK}())_{dsch});\\
\qquad\quad s := SC(s \text{ with } \langle\!|\mathit{es} := *; \mathit{ps}_1 := \uparrow p_1'|\!\rangle);\\
\qquad\quad \mathit{oq} := \mathit{oq}'; \mathit{oqf} := \mathit{oqf}'|\!\rangle
\end{array}$$

$$\begin{array}{l}
\text{socklist\_context } SC \wedge\\
p_1' \in \text{autobind}(s.\mathit{ps}_1, SC) \wedge\\
(\mathit{oq}', \mathit{oqf}', \mathbf{T}) \in \text{dosend}(h.\mathit{ifds}, (\mathit{ips}, \mathit{data}),\\
\qquad\qquad\qquad\qquad (s.\mathit{is}_1, \uparrow p_1', s.\mathit{is}_2, s.\mathit{ps}_2),\\
\qquad\qquad\qquad\qquad h.\mathit{oq}, h.\mathit{oqf}) \wedge\\
\text{string\_size } \mathit{data} \le \text{UDPpayloadMax} \wedge\\
((\mathit{ips} \ne *) \vee (s.\mathit{is}_2 \ne *))
\end{array}$$

The auxiliary function dosend builds the message IP $(i\_k, i\_a, \text{UDP}(\uparrow p_1, \uparrow p_2,$ "ping")) and places it on the output queue, ready to be delivered to the network; the flag **T** indicates that it succeeded. The remaining side conditions check that the payload is not too large for a UDP message, check that a destination IP address is specified either explicitly or implicitly, and automatically provide a local port if none is specified (autobind).

Once the message is on the outqueue, it will eventually be emitted onto the network by rule *delivery_out_1*:

delivery_out_1 **<u>misc</u>** **put UDP or ICMP to the network from** *oq*

$$h \xrightarrow{\overline{\mathrm{IP}(i_1, i_2, body)}} h \text{ with } \langle\!| oq := oq'; oqf := oqf' |\!\rangle$$

$(\mathrm{IP}(i_1, i_2, body), oq', oqf') \in \mathrm{dequeue}(h.oq, h.oqf) \wedge$
$i_2 \notin \mathrm{LOOPBACK} \cup \mathrm{MARTIAN} \wedge$
$i_1 \notin \mathrm{MARTIAN}$

The auxiliary function dequeue takes the top message (if present) from the outqueue and resets the outqueue timer to doq if oq' is nonempty, or ∞ otherwise. We also check that the source and destination addresses are valid for the network; martian [Bak95, §5.3.7] and loopback addresses are handled by other rules.

Once the message is placed on the network (and if it is not lost) it will eventually be delivered to the remote host, where it will either be delivered to a waiting socket, rejected with an ICMP_PORT_UNREACH, or dropped.

In the meantime, ALAN is running *receiver_thread*(). ALAN begins listening for a heartbeat by invoking $\mathsf{select}([fd], [\,], \uparrow 2500000)$, giving a timeout of 2.5 seconds. Rule *select_1* fires:

select_1 **<u>enter2</u>** **entering Select2 state**

$$h \text{ with } ts := ts \oplus (tid \mapsto \mathrm{RUN}_d) \xrightarrow{tid \cdot \mathsf{select}(readseq, writeseq, tms)} h \text{ with } ts := ts \oplus (tid \mapsto \mathrm{SELECT2}(readseq, writeseq)_{d'})$$

$\mathbf{list\_to\_set}(readseq \mathbin{@} writeseq) \subseteq \mathrm{sockfds}\, h.s \wedge$
$(\forall i.(tms = \uparrow i) \implies 0 \leq i) \wedge$
$(d' = \mathbf{case}\ tms\ \mathbf{of}$
$\quad * \to \infty \parallel$
$\quad \uparrow i \to \mathrm{time}(\mathrm{real\_of\_int}\, i/1000000))$

select is a *slow* call [Ste98, p124], meaning that it may block rather than returning immediately to the caller. Here the host transitions into a special blocked $\mathrm{SELECT2}([fd], [\,])_{2.5}$ state, recording the lists of file descriptors on which it is waiting. The timer on this state is set to the timeout specified; this forces us to leave the state at or before the end of the timeout. The other side conditions state that all the file descriptors must be valid, and that the timeout must be nonnegative.

If the heartbeat fails to arrive within 2.5 seconds, the blocked state becomes urgent and rule *select_4* will fire, returning $\mathrm{OK}([\,], [\,])$ and leading ALAN to suspect that KURT is down. If the message *does* arrive, however, it is accepted asynchronously into the listening socket's message queue by *delivery_in_udp_1*, which matches the addressing fields of an incoming message to the address quadruples of the host sockets. (Of course, if there were no matching socket, rule *delivery_in_udp_2* might send an ICMP_PORT_UNREACH message back to the sender.) With a message in the queue, the blocked SELECT2 state becomes urgent, forcing rule *select_3* to fire, informing *receiver_thread* of the waiting message by returning OK([*fd*], [ ]). The thread then invokes recvfrom to read the message.

5 Experimental Validation

Our model is based on the existing natural-language documentation [Ste98,Ste94] and [IEE00], inspection of the sources of the Linux implementation (kernel version 2.2.16-22), and a combination of *ad hoc* and automated testing. Our test network comprised a non-routed subnet with three Linux (RedHat 7.0) and two Windows 2000 machines (in a few cases we ran tests further afield). Tests were written in C, using the `glibc` 2.1.92 sockets library on Linux. Our *ad hoc* tests used C programs to display the results of short sequences of socket calls, using `tcpdump` to observe the network traffic. Later, we wrote an automatic tool, `udpautotest`, that simulates the model (hand-translated into C) in parallel with the real socket calls. This tests representatives of most cases of the host transition semantics, giving us a high level of confidence in our model. It helped us greatly in correctly stating the more subtle corners of the semantics. We also tested some aspects of OCaml thread handling. The limitations of our closed-box testing are discussed in [SSW01b].

Having based the semantics on the Linux implementation, we are now using a combination of `udpautotest` and ad-hoc testing to compare it against the Win2K implementation (v. 5.0, build 2195, no service packs, Winsock2, WS2_32.DLL). The most substantial difference observed so far is that sendto calls are unaffected by earlier ICMP_PORT_UNREACH messages – they successfully send, and do not return the pending error. In contrast, the behaviour of recvfrom and select appears to be as in Linux.

6 HOL Mechanization

We were driven to use mechanized tool support by experience with the model of our earlier work, expressed in conventional non-mechanized mathematics. It was substantially simpler than the model presented here, but its size and complexity already made it hard to keep internally consistent. By expressing the current semantics in HOL we know that its auxiliary functions and semantic rules are all well-typed. We are also using HOL to prove some “sanity” theorems about the model, showing that various invariants on host and network states are maintained, and that the semantic rules cover all possible cases (and overlap only where intended). These results are not especially deep, but proving them has brought up further important points. At the time of writing our most significant result is the following:

Theorem 1 (host_ok **preservation).** *If* host_ok h_0 *and* $h_0 \xrightarrow{l} h$, *then* host_ok h.
The proof (some 2800 lines of script) proceeds by rule category (e.g., **fail**, **succeed**, **slowsucceed**). For each category we prove additional statements that embody type correctness. The host_ok predicate is quite complicated, embodying constraints such as: if a thread is blocking on a recvfrom call on a file-descriptor *fd*, then there must be a socket with that descriptor, and it must have a non-null ps_1 field. (This requirement is maintained in the face of the possibility that some other thread may call close on the *fd*-socket.) Higher order logic seems well-suited to our task. It is quite an expressive logic, and Hindley-Milner type-inference ensures that terms can be written concisely, without excessive type annotation. The mechanization has not required any treatment of binders, simplifying matters.

The HOL system has been used to define operational semantics for various programming languages in the past, including SML and C [Van96,Nor98], so we were confident that the various tools needed for our own definitions and proofs would be present. The implementation of HOL continues to develop (see [NS02]), and our experience has been a substantial prompt to further development.

To make the semantics readable (for ourselves as well as others) we depend on automatic typesetting tools – special-purpose tools we have written to take HOL source and render it into LaTeX, applying the various notational conventions seen in the remainder of the paper.

7 Example: Repeated Heartbeat

At this point, we have (finally) set up enough semantic technology to analyse the example of §2.3. Casual examination of the code may convince the reader that it 'works'; we are now in a position to state this more precisely, and to prove it. We have been able to state a key property of the heartbeat failure detector in HOL, and have carried out a hand proof. The interest is not so much in the specific property, but in the fact that we can express it formally, and the various preconditions which it requires.

Obviously we must assume reasonably fast message delivery, and not too many messages dropped by the network; less obviously, we assume that the threads on ALAN and KURT run fast enough to clear backlogs. For an example case of the proof, suppose KURT is started first, and the receiver on ALAN is not yet listening. The first "ping" may be duplicated by the network, with each arriving message potentially generating an ICMP. In turn, the ICMPs may be duplicated and each duplicate arrive immediately before each call to sendto, causing it to return immediately with an error (*sendto_5*) and forcing *sender_thread* to retry. (This is why the try .. with must enclose the delay as well as the sendto.) A similar situation applies to the receiving end, and there are many other possibilities to be considered. We also prove that no uncaught exceptions arise.

The precise statement of the theorem is in HOL, and is quite elaborate; we here translate it into English.

Consider traces of a network N consisting of ALAN and KURT, quiescent, each with a store and an identical copy of the module in Figure 1. For brevity, we first make some simplifying assumptions, restricting the traces of N which we consider: no incoming messages from outside; rule *net_accept_dup* never creates more than 3 duplicates; rule *net_accept_drop* never drops more than one successive message from each host; the application calls the `rhbeat2` module, but does not call the host or store directly; neither ALAN nor KURT crash; ALAN does not become disconnected, while KURT may become disconnected and reconnected at any point; and the kernel does not run out of memory, nor does any slow system call get interrupted. These assumptions are severe, but are appropriate for the algorithm we consider. A less naïve algorithm would allow them to be relaxed.

Further (trace) assumptions state that the application uses the module correctly: ALAN and KURT each make a single call to the associated start function; and calls to *get_status* occur only on ALAN, after *start_heartbeat_a*(), with the reference returned by it.

Finally, some model timing parameter assumptions. We impose some crude bounds, supposing that *dthread*, *dsch*, doq, and *dstore* are all less than say 1ms, and *dpropmax* is at most 200ms, to obtain a theorem with a simple statement. These ensure: $2\mathit{dpropmax} + 3\,\mathrm{doq} < 1.0$, and hence receipt of an ICMP generated from one “ping” cannot be delayed beyond the sending of the next; and $(2.0 + 120\mathit{dthread} + 22\mathit{dsch} + 2\,\mathrm{doq} + \mathit{dpropmax} - \mathit{dpropmin}) < 2.5$, hence a single message being lost cannot cause a timeout leading to a false failure report.

Given all this, we can identify certain intervals during which a reply to *get_status* is guaranteed to be correct:

Theorem 2 (Correct within reasonable time). *For any trace of N under the above assumptions, if get_status is called at time t and returns a value v, then v is the correct result if t is at least 2.6 seconds after the latest of* KURT*'s last status change and* ALAN*'s call to start_heartbeat_a().*

Of course, this is only one desirable property amongst many [ACT99]. It also does not state that *get_status* returns quickly (or at all). Further, we would like to be able to relax some of the conditions (possibly with a more general algorithm), *e.g.*, to allow the applications to perform other communication operations, and inhabit a larger network. This would require a more elaborate proof, but no changes to the semantics.

8 Conclusion

8.1 Contribution We have given a mathematically precise and experimentally validated model of an interesting class of distributed systems, covering UDP sockets programming, threads, message loss and duplication, host failure and disconnection, timeouts, and rudimentary modules. It is expressed in the HOL theorem prover, and illustrated with a simple heartbeat example. This demonstrates that it is feasible to address the combination of features above, though the experimental approach and tool support (for mechanization, testing and typesetting) have been essential. Our work is a step towards a rigorous understanding of distributed systems – such models can: (1) improve our informal understanding and system-building, (2) underpin proofs of robustness and security properties of particular programs, and (3) support the design, proof and implementation of higher-level distributed abstractions.

8.2 Related work The literature contains a great deal of work on the verification of protocols and distributed algorithms. This includes models of TCP using IO automata and LOTOS [Smi96,Sch96], and work on monitoring TCP implementations from outside the hosts [BCMG01]. To the best of our knowledge, however, there is no other work that accurately models the detailed behaviour of the sockets interface, an understanding of which is critical for actually programming with these protocols. At a higher level of abstraction, Arts and Dam [AD99] have a similar goal to ours – they prove properties of executable concurrent programs, written in Erlang – and the IOA language [GLV00] allows certain forms of IO automata to be executed.

Turning to failure detection, the literature contains sophisticated algorithms and their applications, *e.g.* to consensus problems [ACT99]. Such algorithms satisfy more useful (and more subtle) properties than our naive Theorem 2, but are expressed in informal psuedocode. We have begun to consider how they might be expressed in an extended MiniCaml.

8.3 Future Directions To date, our work on the semantics has been mostly descriptive, focussing on developing an accurate model. This addresses (1) above, but for (2) and (3) we must consider more substantial examples, which will require proof techniques to be adapted from the theories of process calculi and distributed algorithm verification. Extending the coverage of the model would also be valuable, in many directions: UDP multicast, PPP connections, TCP, network partition, other OS socket implementations (especially Win2K and BSD), a larger fragment of OCaml, or other language bindings. Finally, we would like to automatically generate tests from the HOL model.

The work also raises some more general problems. Perhaps surprisingly, even the non-distributed part of the semantics is not routine – to reason about properties of infrastructure implementations we need a semantics for modules with multiple threads that is truly compositional, not dependent on substituting out the module expressions. Our model embodies an *ad hoc* solution to the special case of a single infrastructure module that provides only first-order functions; a more general solution is required, perhaps using game-theoretic techniques. Dealing with module initialisation and separate compilation is also important. From the process-calculus point of view, our parallel composition and restriction are non-standard, both in combining binary and multi-way synchronisation, and in having the local receptiveness property. While timed finite trace equivalence is relatively straightforward in this setting, one might expect interesting differences in the theory of finer observational congruences.

Wansbrough and Serjantov are funded by EPSRC research grant GRN24872 Wide-area programming. *Norrish is funded by a St Catharine's College Heller Research Fellowship. Sewell is funded by a Royal Society University Research Fellowship.*

References

[ACT99] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3-30, June 1999.

[AD99] T. Arts and M. Dam. Verifying a distributed database lookup manager written in Erlang. In *World Congress on Formal Methods (1)*, 1999.

[Bak95] F. Baker. Requirements for IP version 4 routers, RFC 1812. Internet Engineering Task Force, June 1995. `http://www.ietf.org/rfc.html`.

[BCMG01] K. Bhargavan, S. Chandra, P. J. McCann, and C. A. Gunter. What packets may come: Automata for network monitoring. In *Proc. POPL 2001*.

[Bra89] R. Braden. Requirements for internet hosts – communication layers, STD 3, RFC 1122. Internet Engineering Task Force, October 1989.

[CSR83] University of California at Berkeley CSRG. 4.2BSD, 1983.

[GLV00] S. J. Garland, N. Lynch, and M. Vaziri. IOA reference guide, December 2000. `http://nms.lcs.mit.edu/~garland/IOA/`.

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

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

[IEE00] IEEE. *Portable Operating System Interface (POSIX)—Part xx: Protocol Independent Interfaces (PII), P1003.1g*. March 2000.

[L^+01] X. Leroy et al. *The Objective-Caml System, Release 3.02*. INRIA, July 30 2001. Available `http://caml.inria.fr/ocaml/`.

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

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

[NS02] M. Norrish and K. Slind. A thread of HOL development. *Computer Journal*, 2002. To appear.

[Pos80] J. Postel. User Datagram Protocol, STD 6, RFC 768. Internet Engineering Task Force, August 1980. `http://www.ietf.org/rfc.html`.

[Pos81] J. Postel. Internet Protocol, STD 5, RFC 791. Internet Engineering Task Force, September 1981. `http://www.ietf.org/rfc.html`.

[Sch96] I. Schieferdecker. Abruptly terminated connections in TCP – a verification example. In *Proc. COST 247 International Workshop on Applied Formal Methods in System Design*, pages 136–145, 1996.

[SGSAL98] R. Segala, R. Gawlick, J. Søgaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. *Inf. and Comp.*, 141:119–171, 1998.

[Smi96] M. Smith. Formal verification of communication protocols. In *FORTE/PSTV'96*, pages 129–144, 1996.

[SSW01a] A. Serjantov, P. Sewell, and K. Wansbrough. The UDP calculus: Rigorous semantics for real networking. In *Proc TACS2001, Sendai*, October 2001.

[SSW01b] A. Serjantov, P. Sewell, and K. Wansbrough. The UDP calculus: Rigorous semantics for real networking. TR 515, Computer Laboratory, University of Cambridge, July 2001. `http://www.cl.cam.ac.uk/users/pes20/Netsem/`.

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

[Ste98] W. R. Stevens. *UNIX Network Programming Vol. 1: Networking APIs: Sockets and XTI*. Prentice Hall, second edition, 1998.

[Van96] M. VanInwegen. *The machine-assisted proof of programming language properties*. PhD thesis, University of Pennsylvania, December 1996.

[WNSS01] K. Wansbrough, M. Norrish, P. Sewell, and A. Serjantov. Timing UDP: the HOL model, 2001. `http://www.cl.cam.ac.uk/users/pes20/Netsem/`.

[Yi91] W. Yi. CCS + time = an interleaving model for real time systems. In *Proc. ICALP 1991, LNCS 510*, pages 217–228, 1991.

A The LIB interface

The sockets interface

socket	: ()	$\rightarrow$ fd
bind	: fd $*$ ip$\uparrow$ $*$ port$\uparrow$	$\rightarrow$ ()
connect	: fd $*$ ip $*$ port$\uparrow$	$\rightarrow$ ()
disconnect	: fd	$\rightarrow$ ()
getsockname	: fd	$\rightarrow$ ip$\uparrow$ $*$ port$\uparrow$
getpeername	: fd	$\rightarrow$ ip$\uparrow$ $*$ port$\uparrow$
sendto	: fd $*$ (ip $*$ port)$\uparrow$ $*$ string $*$ bool	$\rightarrow$ ()
recvfrom	: fd $*$ bool	$\rightarrow$ ip $*$ port$\uparrow$ $*$ string
geterr	: fd	$\rightarrow$ error$\uparrow$
getsockopt	: fd $*$ sockopt	$\rightarrow$ bool
setsockopt	: fd $*$ sockopt $*$ bool	$\rightarrow$ ()
close	: fd	$\rightarrow$ ()
select	: fd list $*$ fd list $*$ int$\uparrow$	$\rightarrow$ fd list $*$ fd list
port_of_int	: int	$\rightarrow$ port
ip_of_string	: string	$\rightarrow$ ip
getifaddrs	: () $\rightarrow$ (ifid $*$ ip $*$ ip list $*$ netmask) list	

Thread operations

create	: $(T \rightarrow T') \rightarrow T$	$\rightarrow$ tid
delay	: int	$\rightarrow$ ()

Basic operating system operations

print_endline_flush	: string	$\rightarrow$ ()
exit	: ()	$\rightarrow$ void

Exceptions

UDP	: error	$\rightarrow$ exn

Here error is a type of UDP-related Unix errors.

Fig. 2. The LIB interface, with MiniCaml types