The UDP Calculus:  
Rigorous Semantics for Real Networking

Andrei Serjantov  Peter Sewell  Keith Wansbrough

University of Cambridge

{Andrei.Serjantov,Peter.Sewell,Keith.Wansbrough}@cl.cam.ac.uk
http://www.cl.cam.ac.uk/users/pes20/Netsem

Abstract. Network programming is notoriously hard to understand: one has to deal with a variety of protocols (IP, ICMP, UDP, TCP etc), concurrency, packet loss, host failure, timeouts, the complex sockets interface to the protocols, and subtle portability issues. Moreover, the behavioural properties of operating systems and the network are not well documented.

A few of these issues have been addressed in the process calculus and distributed algorithm communities, but there remains a wide gulf between what has been captured in semantic models and what is required for a precise understanding of the behaviour of practical distributed programs that use these protocols.

In this paper we demonstrate (in a preliminary way) that the gulf can be bridged. We give an operational model for socket programming with a substantial fraction of UDP and ICMP, including loss and failure. The model has been validated by experiment against actual systems. It is not tied to a particular programming language, but can be used with any language equipped with an operational semantics for system calls – here we give such a language binding for an OCaml fragment. We illustrate the model with a few small network programs.

1 Introduction

1.1 Background and Problem Distributed applications consist of many concurrently-executing systems, interacting by network communication. They are now ubiquitous, but writing reliable code remains challenging. Most fundamentally, concurrency introduces the classic (but still problematic) difficulties of nondeterminism: large state spaces, deadlocks, races etc. Additional difficulties arise from intrinsic properties of networks: communication is asynchronous and lossy, and hosts are subject to failure. The communication abstractions provided by standard protocols (IP, ICMP, UDP, TCP etc.) are therefore necessarily more complex than simple message-passing or streams. Further, the programmer must understand not only the protocols – the inter-machine communication disciplines – but also the library interface to them. There is a ‘standard’ networking library,
the sockets interface [CSR83,IEE00], lying between applications and the protocol endpoint code on a machine; the programmer must deal with what is visible through this interface, which has a subtle relationship to the underlying protocols. This relationship, and the behaviour of the sockets interface, has not been precisely described, and varies between implementations.

To provide a rigorous understanding of these issues requires precise mathematical models of the behaviour 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.

Previous work on the theories of distributed algorithms and of process calculi has developed models and reasoning techniques for concurrency and failure, but these models are generally rather abstract and/or idealised: to our knowledge, none address the sockets interface and the behaviour it makes visible, most ignore interesting aspects of the core protocols, and most do not support reasoning about executable code. The protocols and sockets interface are worth detailed attention – they are implemented on almost all machines, and underlie higher-level services, including those providing resilience against failure and attack.

1.2 Contribution We give a model that provides a rigorous understanding of the sockets interface and UDP, in realistic networks. To this we add an operational semantics for a programming language (an ML fragment), allowing reasoning about executable distributed programs. We have:

– carefully chosen a useful fragment of the sockets interface and built a thin layer of abstraction above it, focussing on UDP as a starting-point;
– constructed an experimentally-validated operational semantics that covers concurrency, asynchrony, failure and loss;
– developed language-independent semantic idioms for interaction between an application thread, its host OS, and the network;
– instantiated the model with a semantics for an executable fragment of OCaml, MiniCaml; and
– exercised our semantics by proving properties of some small example distributed programs.

Taken together, the above also provide a theorists’ introduction to sockets/UDP programming.

1.3 Experimental Semantics A key goal of our work is to provide a clear and close correspondence between our semantics and the behaviour of actual systems. To achieve this, we cannot alter the extant widely-deployed OS networking code; the most we can do is choose which fragment to model, and add a thin regularising layer above it. Even then, the systems are too complex to analyse and hence derive an accurate semantics; consider the body of machine code and hardware logic embedded in their operating systems, machines, network cards and routers. We are forced therefore both to invent an appropriate level of abstraction at which to express our semantics, and to experimentally determine and validate that semantics. We call this activity experimental semantics.
In our case, the semantics is expressed at the level of the system calls used to communicate between the application language and the operating system sockets code. It was initially based on the relevant natural-language documentation (man pages, RFCs [Pos80,Pos81,Bra89], the Posix standard [IEE00], and standard references [Ste98,Ste94]), and on inspection of the sources of the Linux implementation. We validated the semantics by a combination of ad hoc and automated testing: writing code that interacted with the C sockets interface in the described ways, and confirming that the resulting behaviour corresponded with our model.

To date, the semantics has only been validated against the Linux implementation (in fact, against the Red Hat 7.0 distribution, kernel version 2.2.16-22, glibc 2.1.92). We intend also to use our automated test scripts to identify differences with BSD and with Windows operating systems, if possible picking out a useful common core.

1.4 Overview In the remainder of this section, we give a very brief informal introduction to networks, the protocols IP, UDP, and ICMP, and the sockets interface to them. We then discuss our choice of what to include in the model, and its structure, and highlight some subtleties that must be understood for reliable programming.

In Section 2 we describe the model, making these subtleties precise. Unfortunately the complete definition is too large to include – inevitably so, as the behaviour of even our small (but useful) fragment of the sockets interface is large and irregular by the standards of process calculi and toy languages. Most details are therefore omitted; they appear in the technical report [SSW01]. Section 3 outlines the MiniCaml programming language we adopt for expressing distributed programs, a fragment of OCaml 3.00 [L+00]. Again most details are omitted – these are routine.

Section 4 discusses our experimental setup and validation. The semantics is illustrated with a few small examples in Section 5. Finally, we discuss related work and conclude in Sections 6 and 7.

1.5 Background: Networks and Protocols. Informally At the level of abstraction of our model, a network consists of a number of machines connected by a combination of LANs (eg. 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

\[ \text{IP}(i_s, i_d, \text{body}) \]

where \(i_s\) and \(i_d\) are the source and destination addresses. The implementation of IP (consisting of the routers within the network and the protocol endpoint code in machines) is responsible for delivering the datagram to the correct machine.

\footnote{We discuss in §1.7 and §4 how the model relates to actual systems.}
We can therefore abstract from routing and network topology, and depict a network as below (in fact this is our test network).

![Network Diagram]

Delivery is asynchronous and unreliable – IP does not provide acknowledgments that datagrams are received, or retransmit lost messages.

UDP (the User Datagram Protocol) is a thin layer above IP that provides multiplexing. It associates a set \{1, ..., 65535\} of ports to each machine; a UDP datagram

\[
\text{IP}(i, j, \text{UDP}(ps_1, ps_2, data))
\]

is an IP datagram with a body of the form UDP\((ps_1, ps_2, data)\), containing a source and destination port and a short sequence of bytes of data.

ICMP (the Internet Control Message Protocol) is another thin layer above IP dealing with some control and error messages. Here we are concerned only with two, relating to UDP:

\[
\text{IP}(i, j, \text{ICMP\_PORT\_UNREACH}(i_3, ps_3, i_4, ps_4)), \text{ and }
\text{IP}(i, j, \text{ICMP\_HOST\_UNREACH}(i_3, ps_3, i_4, ps_4)).
\]

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.

TCP (the Transmission Control Protocol) is a rather thicker layer above IP that provides bidirectional stream communication, with flow control and re-transmission of lost data. Most networked applications are built above TCP, with some use of UDP, but we do not yet consider it.
The protocol endpoint code on a machine, implementing the above, is depicted below (together with LIB, which we define in §2.1.3).

1.6 Background: The Sockets Interface, Informally To show how application programs can interact with the UDP endpoint code on their machines, we give the simplest possible example of two programs communicating a single UDP datagram. We describe a small part of the sockets interface informally, presenting only a crude intuition of the behavior. The sender and receiver programs, $e_s$ and $e_r$ respectively, are below. They are written in MiniCaml (with some typographic conventions automatically applied to the executable code).

$$e_s = \begin{array}{l}
\text{let } p = \text{port of int 7654 in} \\
\text{let } i = \text{ip of string "192.168.0.11" in} \\
\text{let } fd = \text{socket() in} \\
\text{let } s = \text{connect(fd, i, } \uparrow p) \text{ in} \\
\text{let } v = \text{sendto(fd, } *\text{, "hello", } \text{false)} \text{ in} 
\end{array}
$$

$$e_r = \begin{array}{l}
\text{let } p' = \text{port of int 7654 in} \\
\text{let } i' = \text{ip of string "192.168.0.11" in} \\
\text{let } fd' = \text{socket() in} \\
\text{let } s = \text{bind(fd', } \uparrow i', \uparrow p') \text{ in} \\
\text{let } v = \text{print_endline_flush "ready" in} \\
\text{let } (\cdot, v) = \text{recvfrom(fd', false) in} \\
\text{print_endline_flush v} 
\end{array}
$$

Here the $*$ and $\uparrow$ are the constructors of option types $T\uparrow$. The types of the library calls are as in Figure 3, but without the 'err', as in MiniCaml an error return raises an exception. The example involves types $\text{fd}$ of file descriptors, $\text{ip}$ of IP addresses, and $\text{port}$ of ports 1-65535.

The sender program $e_s$, which should be run on ALAN, defines a port $p$ and an IP address $i$ (in fact one of machine KURT) and creates a new socket. A socket consists of assorted data maintained by the OS, including an identifier (a file descriptor, which here will be bound to $fd$) and a pair of 'local' and 'remote' pairs of an IP address and a port. These are used for matching incoming datagrams and addressing outgoing datagrams. Program $e_s$ then sets the remote pair of the socket to $i$ and $p$ using connect, and sends a UDP datagram via $fd$ with body "hello".

The receiver $e_r$, which should be run on KURT, defines $i'$ and $p'$ to be the same IP address and port, creates a new socket $fd'$, sets the local pair of $fd'$ to permit reception of datagrams sent to $(i', p')$, and prints "ready". It then blocks, waiting for a datagram to be received by the socket, after which it prints the datagram body.

If $e_s$ and $e_r$ are run on ALAN and KURT respectively (but $e_r$ is started first), and there is no failure in either machine or the network, a single UDP datagram will be sent from one machine to the other.
1.7 Choices: What to Model? To address the issues of §1.1, and support the desired rigorous understanding, the model must satisfy several criteria.

1. It must have a clear relationship (albeit necessarily informal) to what goes on in actual systems; it must be sufficiently accurate for reasoning in the model to provide assurances about the behaviour of those systems. For this, it is essential to include the various failures that can occur.

2. It must cover a large enough fragment of the network protocols and sockets interface to allow interesting distributed algorithms to be expressed. In particular, we want to provide as much information about failure as possible to the programmer, to support failure-aware algorithms.

3. In tension with both of these, the model must be as simple as possible, for reasoning to be tractable.

The full range of network protocols and OS interactions is very large by the standards of semantic definitions. As a starting point, in this paper we choose to address (unicast) UDP and the associated part of ICMP, with a single thread of control per machine, in a flat network. We choose the fragment of the sockets interface that is most useful for programming in these circumstances, and deal with the sockets interface view of message loss, host failure and various local errors. For simplicity, we do not as yet deal with any of the following, despite their importance.

- TCP, and associated ICMP messages
- broadcast and multicast UDP communication
- multithreaded machines and inter-thread communication
- other IO primitives (in this paper we choose, minimally, ‘print’ and ‘exit’)
- persistent storage
- network partition (especially for machines with intermittent connections)
- DNS
- IPv6 protocols
- machine reconfiguration and other privileged operations

We are not modelling the implementation of IP (routing, fragmentation etc.) or lower levels (Ethernet, ARP, etc.), as we aim to support reasoning about distributed applications and algorithms above IP, rather than implementations of low-level network protocols.

The standard sockets interface is a C language library. To avoid dealing with irrelevant complexities of a C interface (weak typing and explicit memory management) we introduce a thin abstraction layer, providing a clean strongly-typed view (we also clean up the interface by omitting redundancy). This LIB interface is defined in Figure 3; it was shown in the diagram at the end of §1.5.

In this paper we describe only an interleaving semantics. We anticipate that it will be straightforward to add fairness constraints, which are required for reasoning about non-trivial examples, and intend to investigate lightweight timing annotations, for more precise properties about examples involving time-outs. The model is not intended for quantitative probabilistic reasoning, eg. for quality of service issues. It may, however, provide a useful model for reasoning about
some forms of malicious attack – *e.g.* for networks with some malicious hosts, though with our flat network topology we do not deal with firewalls.

Blocking system calls are a key aspect of sockets programming, so it is natural to deal with sequential threads, rather than a concurrent programming language with language-level parallelism (for which blocking system calls would block the entire runtime).

1.8 Structuring the Model (and Language Independence) We want to reason about executable implementations of distributed algorithms, expressed in some programming language(s), not in a modelling language. We do not wish to fix on a single language, however, as the behaviour of the sockets interface and network is orthogonal to the programming language used to express the computation on each machine. We therefore factor the model, allowing threads to be arbitrary labelled transition systems (LTSs) of a certain form. One can extend the operational semantics of a variety of languages with labelled transitions, for library calls and returns, so that programs denote these LTSs (values used by the sockets interface are all of rather simple types, not involving callbacks, so this is straightforward). In this paper we do so for a fragment of OCaml with functions, references and exceptions. This allows our example programs to be executed without change, by linking them with a module providing our thin layer of abstraction, LIB, above the OCaml sockets library (in turn implemented above the C library).

It will be convenient to be able to describe partial systems, for example to consider the interactions between the collection of all threads and the rest of the system, so we allow hosts and their threads to be syntactically separated. Networks therefore consist of a parallel composition of IP datagrams, hosts (each with a state \(v\), giving the host's IP addresses, states of sockets *etc.*), and threads (each with a state \(e\) of an LTS). The precise definition is in §2.1.4, which uses the grammar below.

\[
\begin{align*}
N &::= 0 & \text{empty} \\
N &| N & \text{parallel composition} \\
\{v\} & & \text{IP datagram in transit} \\
\text{Host } v & & \text{host } n \text{ with state } v \\
\text{Thread } n^e & & \text{thread of host } n \text{, with state } e
\end{align*}
\]

The host semantics— the heart of the model— is outlined in §2.3. The behaviour of networks is defined in §2.2.2 by a structural operational semantics (SOS), combining the LTSs of hosts and threads, using process calculus techniques (we give a direct operational semantics, rather than a complex encoding into an existing calculus).

1.9 It’s Not Really So Easy The informal introductions to the protocols and sockets interface in §§1.5.1.6 above give a deceptively simple view. Real
network programming must take into account the following, all of which are captured in our model:

1. IP addresses and ports with zero values have special meanings, being treated roughly as wildcards, both in the arguments to bind, connect, etc. and in the socket states. Our ip and port are types of non-zero IP addresses and ports; we use option types ip* and port where the zero values (*) may occur.

2. The system-call interactions between a thread and its host are weakly coupled to the interactions between a host and the network. Messages may arrive at a machine, and be processed (and buffered) by the network hardware and OS, at almost any time. The sendto and recvfrom calls can block, until there is queue space to send a message or until a message arrives, respectively. Further, select allows blocking until one of a number of file descriptors is ready for reading or writing, or a specified time has elapsed. Communication between hosts is asynchronous, due both to buffering and the physical media.

3. Machines can fail; messages can be lost, reordered, or duplicated. There is buffering (and potential loss) at many points: in the operating system, in the network cards, and in the network routers. UDP provides very little error detection and no recovery. UDP datagrams typically contain a checksum (here we idealise, assuming that the checksum is perfect and hence that all corrupted datagrams are discarded). More interestingly, remote failure can sometimes be detected: a machine receiving a UDP datagram addressed to a port that does not have an associated socket may send back an ICMP message. These can asynchronously set an error flag in the originating socket, giving rise to an error from a blocked or future library call.

4. Many local errors are possible, for example (just considering bind): a port may be already in use or in a privileged range; an IP address may not belong to the machine; the OS may run out of resources; the file descriptor may not identify a socket. In MiniCaml, these are reported via exceptions, which may be caught and handled.

5. Machines can have more than one IP address – in fact, a machine may have several interfaces each of which has a primary IP address and possibly also other alias IP addresses. Typically each interface will correspond to a hardware device, but a machine will also have a loopback interface which echoes messages back.

6. The sockets interface includes assorted other functionality – further library calls, socket options etc.

2 UDP – The Model

We now present the UDP Calculus, our model of the network and of the sockets interface to UDP. As the definition is far too large to include here, we give only the basic structure and selected highlights, leaving the full details to the technical report [SSW01]. Section 2.1 presents the static structure of the model, Section 2.2 explains the interactions between parts of the model, Section 2.3 illustrates the
\[ T ::= \text{int} \]
\[ \text{bool} \]
\[ \text{string} \]
\[ () \quad \text{unit type} \]
\[ T_1 \times \cdots \times T_n \quad \text{tuple} \quad (n \geq 2) \]
\[ T \text{ list} \quad \text{list} \]
\[ T^\dagger \quad \text{optional type} \]
\[ T \text{ err} \quad T \text{ or error} \]
\[ \text{void} \quad \text{empty type} \]
\[ \text{fd} \quad \text{file descriptor} \]
\[ \text{ip} \quad \text{IP address} \]
\[ \text{port} \quad \text{port} \]
\[ \text{error} \quad \text{OS error} \]
\[ \text{netmask} \quad \text{netmask} \]
\[ \text{ifid} \quad \text{interface descriptor} \]
\[ \text{sockopt} \quad \text{socket options} \]
\[ T \text{ set} \quad \text{finite set} \]
\[ \text{ipBody} \quad \text{body of IP datagram} \]
\[ \text{msg} \quad \text{IP datagram} \]
\[ \text{ifid} \quad \text{interface descriptor table entry} \]
\[ \text{flags} \quad \text{flags from socket descriptor table entry} \]
\[ \text{socket} \quad \text{socket descriptor table entry} \]
\[ \text{hostid} \quad \text{unique identifier of a host} \]
\[ \text{hostThreadState} \quad \text{the OS view of a thread} \]
\[ \text{host} \quad \text{a single host} \]

The clauses annotated by TL form a subgrammar of T, the language types. All values passed between a thread and its host OS are of a language type.

\[ \text{Fig. 1. Types} \]

host semantics by means of some key rules, and Section 2.4 discusses some sanity results.

2.1 Statics: Types, Values, and Judgements

The model is largely built from the types T shown in Figure 1, which have values v composed of the constructors c ∈ Con given in Figure 2; constructors can be polymorphic. Each constructor has a natural number arity and a non-empty set of sequences (of length one plus that arity) of types; the sequences are written with arrows \( \rightarrow \). The obvious typing judgement for values is written \( \vdash v : T \).

A number of invariants are captured by additional judgements, omitted here. Notation: We typically let i, p, e range over values of types ip, port, error, and is, ps, es over values of types ip^\dagger, port^\dagger, error^\dagger.

2.1.1 Hosts and Threads We separate a running machine into two parts: the host, representing the machine itself and its operating system; and the thread
Partition \( \text{Con} \) into the language constructors:

\[
\ldots, -1, 0, 1, 2, \ldots \quad : \text{int} \\
\text{true, false} \quad : \text{bool} \\
\text{octet-sequence} \quad : \text{string} \\
() \quad : () \\
(\ldots) \quad : \text{mixfix}
\]

\[
T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T_1 * \ldots * T_n \quad n \geq 2 \\
\text{NIL} \quad : \text{T list} \\
:: \quad : T \rightarrow \text{T list} \\
\* \quad : T \rightarrow T \\
\uparrow \quad : T \rightarrow T \uparrow \\
\text{OK} \quad : T \rightarrow T \text{err} \\
\text{Fail} \quad : \text{error} \rightarrow T \text{err}
\]

\[
1.2^{32} - 1 \quad : \text{ip} \\
1.65535 \quad : \text{port} \\
\text{FD_0, FD_1, \ldots} \quad : \text{fd} \\
\sum_{i \in \ldots , 31} 2^i \quad : \text{netmask} \quad \text{for } 0 \leq j \leq 31
\]

\[
\text{SO_BSDCOMPAT, SO_REUSEADDR} \quad : \text{sockopt} \\
\text{EACCES, EADDRINUSE, EADDRNOTAVAIL, EAGAIN, EBADF,} \\
\text{ECONNREFUSED, EHOSTUNREACH, EINTR, EINVAL, EMFILE,} \\
\text{EMSGSIZE, ENFILE, ENOBUF, ENOMEM, ENOTCONN,} \\
\text{ENOTSOCK} \quad : \text{error}
\]

and the non-language constructors:

\[
\text{IP} \quad : \text{ip * ip * ipBody} \rightarrow \text{msg} \\
\text{UDP} \quad : \text{port * port * string} \rightarrow \text{ipBody} \\
\text{ICMP, HOST-UNREACH} \quad : \text{ip * port * ip * port} \rightarrow \text{ipBody} \\
\text{ICMP, PORT-UNREACH} \quad : \text{ip * port * ip * port} \rightarrow \text{ipBody} \\
\text{HOST} \quad : \text{ifd set * hostThreadState * socket list * msg list * bool} \rightarrow \text{host} \\
\text{SOCK} \quad : \text{fd * ip * port * ip * port * ip * port * error * flags * [msg * ifd] list} \rightarrow \text{socket} \\
\text{IF} \quad : \text{ifid * ipset * ip * netmask} \rightarrow \text{ifid} \\
\text{RUN} \quad : \text{hostThreadState} \\
\text{TERM} \quad : \text{hostThreadState} \\
\text{RET TL} \quad : \text{TL} \rightarrow \text{hostThreadState} \\
\text{SENDTO2} \quad : \text{fd * (ip * port * string} \rightarrow \text{hostThreadState} \\
\text{RECVFROM2} \quad : \text{fd} \rightarrow \text{hostThreadState} \\
\text{SELECT2} \quad : \text{fd list * fd list * int} \rightarrow \text{hostThreadState} \\
\text{PRINT2} \quad : \text{String} \rightarrow \text{hostThreadState} \\
\text{FLAGS} \quad : \text{bool * bool} \rightarrow \text{flags} \\
\text{ALAN, KURT, ASTROCYTE, \ldots} \quad : \text{hostid}
\]

Elements of \( T \) set are written \( \{v_1, \ldots, v_n\} \). The \( TL \) subscript of \( \text{RET TL} \) will usually be elided.

**Fig 2:** Constructors
representing the application program controlling it. Threads are explained in §2.2.1. A host is of the form:

\[
\text{Host}(ifds, t, s, oq, oaf)
\]

A host has a set \(ifds : \text{idset}\) 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 \(i \in ifds\) to mean ‘\(i\) is an IP address of one of the interfaces in \(ifds\)’. The operating system’s view of the thread state is stored in \(t : \text{hostThreadState}\); the thread may be running (\(\text{RUN}\)), terminated (\(\text{TERM}\)), or waiting for the OS to return from a call. In the last case, the OS may be about to return a value from a system call (\(\text{RET } v\)) or the thread may be blocked waiting for a system call to complete (\(\text{SENDTO2 } v\), \(\text{RECVFROM2 } v\), \(\text{SELECT2 } v\), \(\text{PRINT2 } v\)). The host’s current list of sockets is given by \(s : \text{socket list}\). The \(\text{outqueue}\) a queue of outbound IP messages, is given by \(oq : \text{msg list}\) and \(oaf : \text{bool}\), where \(oq\) is the list of messages and \(oaf\) is set when the queue is full.

2.1.2 Sockets The central abstraction of the sockets interface is the \(\text{socket}\). It represents a communication endpoint, specifying a local and a remote pair of an IP address and UDP port, along with other parts of the protocol implementation state. It is of the form

\[
\text{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)
\]

A socket is uniquely identified within the host by its file descriptor \(fd : \text{fd}\). The local and remote address/port pairs are \(is_1 : \text{ip}\), \(ps_1 : \text{port}\) and \(is_2 : \text{ip}\), \(ps_2 : \text{port}\) respectively; wildcards may occur. Asynchronous error conditions store the pending error in the error flag \(es : \text{error}\). An assortment of socket parameters are stored in \(f : \text{flags}\). Finally, \(mq : (\text{msg s ifid})\text{ list}\) is a queue of incoming messages that have been delivered to this socket but not yet received by the application.

2.1.3 The Sockets Interface A library interface defines the form of the interactions between a thread and a host, specifying the system calls that the thread can make. A library interface consists of a set of calls, each with a pair of language types. We take a library interface LIB, shown in Figure 3, consisting of the sockets interface together with some basic OS operations.

All of the sockets interface calls return a value of some type \(T\text{err}\) to the thread, which can be either \(\text{OK } v\) for \(v : T\) or \(\text{FAIL } e\) for a Unix error \(e : \text{error}\). A language binding may map these error returns into exceptions, as the MiniCaml binding of §3 does.

2.1.4 Networks A network \(N\) (a term of the grammar in §1.8) is a parallel composition of IP datagrams IP \(v\), hosts \(n : \text{HOST } v\), and their threads \(n : e\). To describe partial systems, we allow hosts and their threads to be split apart. The association between them is expressed by shared names \(n : \text{hostid}\), which are purely semantic devices, not to be confused with IP addresses or DNS names. A well-formed network must contain at most one host and at most one thread
The sockets interface:

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket</td>
<td>()</td>
<td>fd err</td>
</tr>
<tr>
<td>bind</td>
<td>fd * ip * port *</td>
<td>() err</td>
</tr>
<tr>
<td>connect</td>
<td>fd * ip * port *</td>
<td>() err</td>
</tr>
<tr>
<td>disconnect</td>
<td>fd</td>
<td>() err</td>
</tr>
<tr>
<td>getpeername</td>
<td>fd</td>
<td>(ip * port) err</td>
</tr>
<tr>
<td>getsockopt</td>
<td>fd</td>
<td>(ip * port) err</td>
</tr>
<tr>
<td>sendto</td>
<td>fd * (ip * port) * string * bool</td>
<td>() err</td>
</tr>
<tr>
<td>recvfrom</td>
<td>fd * bool</td>
<td>(ip * port * string) err</td>
</tr>
<tr>
<td>getsockopt</td>
<td>fd</td>
<td>error err</td>
</tr>
<tr>
<td>setsockopt</td>
<td>fd *sockopt</td>
<td>() err</td>
</tr>
<tr>
<td>close</td>
<td>fd</td>
<td>() err</td>
</tr>
<tr>
<td>select</td>
<td>fd list * fd list * int</td>
<td>(fd list * fd list) err</td>
</tr>
<tr>
<td>port of int</td>
<td>int</td>
<td>port err</td>
</tr>
<tr>
<td>ip of string</td>
<td>string</td>
<td>ip err</td>
</tr>
<tr>
<td>getifaddrs</td>
<td>()</td>
<td>(ifid * ip * ip list * netmask) list err</td>
</tr>
</tbody>
</table>

Basic operating system operations:

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_endline_flush</td>
<td>string</td>
<td>() err</td>
</tr>
<tr>
<td>exit</td>
<td>()</td>
<td>void</td>
</tr>
</tbody>
</table>

Fig. 3. The library interface LIB

---

**Fig. 4.** Thread. Host and Network
LTS for each name. Hosts and messages must be well-formed, and no two hosts may share an IP address.

2.2 Dynamics: Interaction

The threads, hosts, and the network itself are all labelled transition systems; they interact by means of CCS-style synchronizations. Figure 4 shows the network

$$N = \text{ALAN}\cdot e \mid \text{ALAN-Host}(..)$$

\[ IP(\text{ALAN}, \text{KURT}, \text{UDP}(1024, 7654, "hello")) \]

along with some of its possible interactions (showing the host LTS labels). Host and thread are linked by the hostid prefix on their transitions, but messages on the network are bare – messages are not tied to any particular host, other than by the IP addresses contained in their source and destination fields. As we shall see, the host and thread LTSs are defined without these prefixes, which are added when they are lifted to the network SOS.

The only interaction between a thread and its associated host is via system calls – a call and its return are both modelled by CCS-style synchronizations. A thread can make a system call \( f v \) for any \( f : T \to T' \) in LIB and argument \( v : T \), for example sendto(\( .. \)). The operating system may then return a value \( r : T' \), for example OK(). In the above diagram, the host’s ALAN-sendto(\( .. \)) and ALAN-OK(\( .. \)) are part of call and return synchronizations respectively.

Invocations of system calls may be fast or slow [Ste08, p124]. Fast calls return quickly, whereas slow calls block, perhaps indefinitely – for example until a message arrives. The labelled transitions have the same form for both, but the host states differ (as in §2.1.1). (In the absence of slow calls, one could model system calls as single transitions, carrying both argument and return values, rather than pairs.)

A host interacts with the network by sending and receiving IP datagrams; ALAN-IP(\( .. \)) and ALAN-IP(\( .. \)) in the figure, respectively.

A host may also emit strings to its console with transitions of the form ALAN-console "hello". This provides a minimal way to observe the behaviour of a network, namely by examining the output on each console.

2.2.1 Thread LTSs and Language Independence The interactions between a thread and the OS are essentially independent of the programming language the thread is written in – they exchange only values of simple types, the language types of Figure 1. Instead of taking a thread to be a syntactic program in some particular language, we can therefore take an arbitrary labelled transition system, with labels \( f v, r \) and \( \tau \). It is then straightforward to extend an operational semantics for a variety of languages to define such an LTS, as we do for MiniCaml in §3.
Take a thread LTS $e$ to be ($\text{Lthread}, S, \rightarrow, s_0$) where $S$ is a set of states, $s_0 \in S$ is the initial state $\rightarrow \subseteq S \times \text{Lthread} \times S$ is the transition relation, and the labels are 

$$\text{Lthread} = \{ f : TL \rightarrow TL' \in \text{LIB} \land \vdash v : TL \} \cup \{ r \mid \exists TL. \vdash r : TL \} \cup \{ \tau \}$$

Some axioms must be imposed to give an accurate model, as in [Sev97]. System calls are deterministic – a thread cannot offer to invoke multiple system calls simultaneously. Moreover, after making a system call, the thread must be prepared to input any of the possible return values, and its subsequent behaviour will be a function of the value. Threads may however have internal nondeterminism. A thread can always make progress, unless it has been terminated by invoking exit (the only system call with return type void). The precise statements of these properties are given in [SSW01].

### 2.2.2 Network Operational Semantics

The transitions of a network are defined by the rules below, together with a structural congruence defined by associativity, commutativity and identity axioms for $|$ and $0$. Here we let $x$ be either a host (with $\vdash x$ host-ok) or a thread LTS, $\vdash n : \text{hostid},$ and $\vdash N : \text{network}.$

$$x \xrightarrow{l} x' \quad l \neq \tau$$
$$n.x \xrightarrow{n.t} n.x'$$
$$x \xrightarrow{\tau} x'$$
$$N_1 \xrightarrow{n.t} N_1'$$
$$N_2 \xrightarrow{n.t} N_2'$$

**par.1**

$$N_1 \mid N_2 \xrightarrow{\tau} N_1' \mid N_2'$$
$$N_1 \mid N_2 \xrightarrow{n.t} N_1' \mid N_2'$$

**par.2**

$$N_1 \mid N_2 \xrightarrow{n.t} N_1' \mid N_2'$$

**drop.1**

$$0 \xrightarrow{n.IP} 0$$
$$0 \xrightarrow{n.IP} 0$$

**dup.1**

$$k \geq 2$$

$$N_1 \mid N_2 \xrightarrow{n.t} N_1' \mid N_2'$$

**host.cdrash.1**

$$n.\text{HOST} \xrightarrow{n.c.draft} 0$$

**host.cdrash.2**

$$n.e \xrightarrow{n.c.draft} 0$$

IP datagrams can arrive out of order, be lost or be (finely) duplicated. Reordering is built into the rules above, but for the other kinds of failure we add the rules **drop.1** and **dup.1**. These are most interesting when constrained, eg. by fairness or timing assumptions. Hosts can also fail in a variety of ways. In this paper we consider only the simplest, ‘crash’ failure [Mu93, §2.4].

Our network has no interesting topological structure. It can always receive a new datagram, and can always deliver any datagram it has, with rules similar to those of Honda and Tokoro’s asynchronous $\pi$-calculus [HT91].
2.3 Highlights of the Host Semantics

We now highlight a few of the most interesting parts of the host semantics, illustrating some (10 out of 72) of the host transition axioms. The definitions of several auxiliary functions are omitted. We aim to give some feeling for the intricacies of UDP sockets and to demonstrate that a rigorous treatment is feasible, without (for lack of space) fully explaining our semantics.

2.3.1 Ports: Privileged, Ephemeral, and Unused, and Autobinding
The ports 1...65535 of a host are partitioned into the privileged = \{1..., 1023\}, the ephemeral = \{1024..., 4999\}, and the rest (these sets are implementation-dependent; we fix on the Linux defaults). The unused ports of a host are the subset of \{1..., 65535\} that do not occur as the local port of any of its sockets. One can bind the local port of a socket either to an explicit non-privileged value, e.g., the \(p_i' = 7654\) of the \(e_r\) example in §1.6, or request the OS to choose a unused port from the set of ephemeral ports. The latter autobinding can be done by invoking bind with a \(*\) in its port\(\uparrow\) argument, as in the bind.2 rule:

\[
\text{bind.2 (\(\uparrow i, *\)) succeed, autobinding } \\
F(\text{ifds, Run, Sock}(fd, i, s, s, s, es, f, mq)) \\
\frac{\text{bind\(\downarrow\)(fd, \(\uparrow i, *\))}}{\rightarrow F(\text{ifds, Ret (OK)}, \text{Sock}(fd, \uparrow i, \uparrow p_i', s, s, es, f, mq))}
\]

\(p_i' \in \text{unused}(F) \cap \text{ephemeral and } i \in \text{ifds}\)

To reduce the syntactic clutter in rules, we define several classes of contexts that build a host. Here \(F\) ranges over contexts of the form \(\text{Host}(\omega_1, \omega_2, S(\omega_3), qo, qof)\), where \(S\) is a socket list context, of the form \(s_t \equiv \{[\ldots] s_{i_2}\}.\) The rule also requires the IP address \(i\) to be one of those of this host. Autobinding can also occur in connect (if one connects a socket that does not have a local port bound), in disconnect, in sendto, and in recvfrom.

2.3.2 Message Delivery to the Net In the simplest case, sending a UDP datagram involves two host transitions: one that constructs the datagram and adds it to the host outqueue, and one that takes it from the outqueue and outputs it to the network. These are given by the host transition axioms below.

\[
\text{sendto.1 succeed } \\
\text{Host}(\text{ifds, Run, S(Sock(fd, is, ps, ps, * f, mq), qo, qof}) \\
\frac{\text{sendto(fd, ips, data, n)}}{\rightarrow \text{Host}(\text{ifds, Ret (OK)}, S(\text{Sock(fd, is, \uparrow p_i', is, ps, f, mq), qo', qof'}) \\
p_i' \in \text{autobind}(ps, S) \\
\text{and } (qo', qof', \text{true}) \in \text{dosend(ifs, (ips, data)}, (is, \uparrow p_i', is, ps, f, mq), qo, qof') \\
\text{and } \text{size(data)} \leq \text{UDPpayloadMax and } (ips \neq * \text{ or } is \neq *)
\]

In sendto.1: \(S\) is a socket list context, allowing the \(fd\) socket to be picked out; the autobind function provides a nondeterministic choice of an unused ephemeral port, if the local port of this socket has not yet been bound; the dosend function
constructs a datagram, using the *ips* argument to *sendto* and the IP addresses and ports from the socket, and adds it to the outqueue (or fails, if the queue is full); the length of *data* must be less than UDP*payloadMax*; and at least one of the *ips* argument and the socket must specify a destination IP address.

```
delivery.out.1 put UDP or ICMP to the network from *oq*
    HOST(ifds, t, s, *oq, *oq*)
    IP(*i₃, *i₄, body)\rightarrow HOST(ifds, t, s, *oq*, *oq*')
((IP(*i₃, *i₄, body)), *oq', *oq'') \in \text{dequeue}(*oq, *oq*)
and *i₃ \notin \text{LOOPBACK} \cup \text{MARTIAN} and *i₄ \notin \text{MARTIAN}
```

In *delivery.out.1*: the dequeue function picks a datagram off the outqueue (nondeterministically resetting the *oq flag*), and checks the datagram has non-martian source and destination addresses [Bak95 §5.3.7]. It outputs the datagram to the network.

### 2.3.3 Return From a Fast Call
After the invocation of a fast call, e.g. an instance of the *sendto.1* rule above, the host thread state is of the form RET *v*, recording the value *v* to be returned to the thread by *ret.1* below.

```
ret.1 return value *v* from fast system call to thread
    HOST(ifds, RET *v*, s, *oq, *oq*)
    \rightarrow HOST(ifds, RUN, s, *oq, *oq*)
```

### 2.3.4 Message Delivery from the Net
If the thread invokes *recvfrom* on a socket *fd* that does not have any queued messages, with the 'non-blocking' flag argument *false*, the thread will block until a message arrives (or until an error of some kind occurs).

```
recvfrom.2 block entering Recvfrom2 state
    F(ifds, RUN, SOCK(*fd, *is₁, *ps₁, *is₂, *ps₂, *, *f, NIL))
    \rightarrow F(ifds, RECVFROM2 *fd, SOCK(*fd, *is₁, *p₄', *is₂, *ps₂, *, *f, NIL))
    *p₄' \in \text{autobind}(*ps₁, \text{socks}(F))
```

As in *bind.2* and *sendto.1*, the local port of the socket will be automatically bound (to an unused ephemeral port) if it is not already bound.

When a UDP datagram, e.g. IP(*i₃, *i₄*, UDP(*ps₃, *ps₄, data*)), arrives at a host, the 4-tuple (*i₃, *ps₃, *i₄, *ps₄*) is matched against each of the host’s sockets, to determine which (if any) the datagram should be delivered to. This matching compares the 4-tuple with each SOCK(... *is₁, *ps₁, *is₂, *ps₂, ...), giving a score from 0 to 4 of how many elements match, treating a * in the socket elements as a wildcard. The lookup function takes a list *s* of sockets and a datagram 4-tuple (*i₃, *ps₃, *i₄, *ps₄*), returning the set of sockets with maximal non-zero scores. The datagram is delivered to one of these sockets, by adding it to the end of the
socket’s message queue \(mq\). This is expressed in the basic \texttt{delivery.in.udp.1} rule below.

![Rule](image)

After this, a blocked \texttt{recvfrom} will be able to complete, using the \texttt{recvfrom.6} rule.

![Rule](image)

2.3.5 **ICMP Generation** If a UDP datagram arrives at a host (so its destination IP address is one of the host’s) but no socket matches its 4-tuple \((i_3, ps_3, i_4, ps_4)\) then the host may or may not send an \texttt{ICMP\_PORT\_UNREACH} message back to the sender. This is dealt with by the rule below (in the non-loopback case). Note that the ICMP message is added to the host’s outqueue \(oq\), not put directly on the network. This uses an auxiliary function \texttt{enqueue} which is also used by \texttt{dosed}.

![Rule](image)

2.3.6 **Asynchronous Errors** When an \texttt{ICMP\_PORT\_UNREACH} message arrives at a host, it is matched against the sockets, in roughly the same way that UDP datagrams are. If it matches a socket (which typically will be the one used to send the UDP datagram that generated this ICMP) then the error should be reported to the thread. The arrival and processing of the ICMP message is
asynchronous w.r.t. the thread activity, though, so what happens is simply that
the error flag \( e' \) of the socket is set, in this case to \( \text{ECONNREFUSED} \).

\[
\text{delivery.in.icmp.1 get ICMP from the network, setting error in a matching socket}
\]

\[
\frac{\text{IP}(i_t', i_d', \text{ICMP}_X \text{UNREACH}(i_2, i_3, i_4, i_5))}{\text{HOST}(ifds, t, s, \text{mq})} \quad \text{HOST}(ifds, t, S(\text{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) = s

and \text{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in \text{lookup } s(i_2, ps_2, i_4, ps_4)

and \( m = \text{IP}(i_t', i_d', \text{ICMP}_X \text{UNREACH}(i_3, ps_3, i_5, ps_5)) \)

and \( i_t' \in \text{ifds} \text{ and } \neg((\text{loopback}(m) \lor \text{martian}(m))) \)

and \( e' = \text{if } (is_2 \neq *) \text{ or } \neg(\text{bsdcompat } f) \text{ then } \text{ECONNREFUSED else } es \)

Here X is either HOST or PORT. There are sanity constraints on the IP addresses involved, and the behaviour differs according to whether the bsdcompat socket flag is set. Note also that unmatched ICMPs do not themselves generate new ICMPs — there is no analogue of \text{delivery.in.udp.2} for ICMPs.

The error flag may cause subsequent sendtos or recvfroms to fail, returning the error and clearing the flag, for example in the rule below:

\[
\text{sendto.5 fail, as socket in an error state}
\]

\[
\frac{\text{F}(ifds, \text{RUN, SOCK}(fd, is_1, p_1, is_2, ps_2, e, f, mq))}{\text{F}(ifds, \text{RET \ (FAIL, e), SOCK}(fd, is_1, p_1, is_2, ps_2, e, f, mq))}
\]

2.3.7 Local Errors A number of other sources of error must be dealt with. Firstly, there are straightforward erroneous parameters. Any call that takes an fd can return ENOTSOCK or EBADF if given a file descriptor that is not a socket. For bind we also have errors for a privileged port, a port already in use (modulo the reuseraddr flags), an IP address that is not one of the host’s, and a socket which already has a non-* local port. For sendto we have errors if the destination is * and the socket is unconnected, and if the data is bigger than UDPpayloadMax. Both sendto and recvfrom return EAGAIN if the non-blocking flag argument is set but the call would block.

Secondly, any of the slow calls (sendto, recvfrom, select) can return EINTR from the blocked state if the system call is interrupted. Our model does not contain the sources of such interrupts, so all we can do is include a nondeterministic rule allowing the error to occur.

Thirdly, there are pathological cases in which the OS has exhausted some resource. A call to socket can return EMFILE or ENFILE, if there are too many open files or the file table overflows, and all calls can return ENOMEM or ENOBUFFS if the OS has run out of space or buffers. Again, these are modelled by purely nondeterministic rules. We must also deal with the possibility that all the ephemeral ports are exhausted.
2.3.8 Loopback A datagram sent to a loopback address, typically 127.0.0.1, will be echoed back—without reaching the network. To model loopback, we use a number of additional delivery rules which are essentially the compositions of \texttt{delivery.out.*} and \texttt{delivery.in.*} rules. For example, a rule \texttt{delivery.loopback.udp.*} removes a loopback UDP from a host's outqueue and delivers it to a matching socket, in a single step.

2.4 Sanity Properties

We have proved type preservation and progress theorems for the model, and a semideterminacy result. The latter states roughly that for a given system call and host state, either the call succeeds (and exactly one rule applies) or it fails (several error rules may be in competition). The combination of the progress result, the thread LTS axioms and the network SOS rules exclude pathological deadlocks.

3 MiniCaml

MiniCaml is designed to be a sublanguage of OCaml 3.00 [L+00]. Its \textit{types} (with corresponding constructors) are given by the grammar marked $T_{L}$ in Figure 1 (except \texttt{T err}), together with:

$$T ::= \cdots \mid T \rightarrow T' \mid T \text{ ref} \mid \text{con}$$

The \textit{syntax}, \textit{typing rules} and \textit{reduction rules} are standard, with additions to define an LTS satisfying the axioms of §2.2.1. We also prove theorems stating type preservation and absence of runtime errors.

We have written an OCaml module \texttt{udplang} which implements almost all of LIB (together with the required types and constructors). The example programs in this paper are automatically typeset from working code omitting an \texttt{open udplang;}; at the beginning of each program and using mathematically concrete syntax, writing \{. $T_{l}$, \texttt{e}. $s$, \texttt{e}. $\rightarrow$ for unit, \texttt{T lift}, \texttt{Lift e}. \texttt{Star} and \texttt{->}.

4 Validation

To develop and validate our host semantics, we set up a test network; a non-routed subnet with four dedicated machines (two Linux and two Win2K), accessible via an additional interface on one of our Linux workstations. In a few cases we ran tests further afield. Tests were written in C, using the glibc socket library. Initially we wrote a large number of \textit{ad hoc} tests. C programs that display the results of short sequences of socket calls, and also observed the resulting network traffic with the \texttt{tcpdump} utility. Certain hard-to-test issues were resolved by inspecting the Linux kernel source code.

Later, to more thoroughly validate the semantics as a whole, we translated the host operational semantics into C; we wrote an automatic tool \texttt{udpautotest}. 
that simulates the model in parallel with the real socket calls. This tests representatives of most cases of the semantic rules, giving us a high level of confidence in our model. It helped us greatly in correctly stating the more subtle corners of the semantics, and will hopefully make determining the semantics of other implementations (such as Win2K or BSD) relatively routine.

The closed-box testing has a number of limitations, however (which we discuss further in [SSW01]). We do not directly observe the internal socket state (of which our SOCK structures are an abstraction), some pathological cases are hard to set up, and it is clearly impossible to exhaust all cases. Loss is very rare on our single subnet, and as far as we are aware reordering and duplication never occur. We therefore cannot regard the semantics as definitive, and would be interested to hear of discrepancies between it and real system behaviour.

We have endeavoured to make the model as accurate as possible, for the fragment of socket programming and the level of abstraction chosen in §1.7, and as far as one can with an untimed interleaving semantics. Nonetheless, it is in some respects idealised. Some of these are resource issues – we do not bound the MiniCaml space usage, and have a purely nondeterministic semantics for OS allocation failures. We simplify the real full-outqueue behaviour, and use an approximation to the treatment of ’martian’ datagrams. We also assume unbounded integers and perfect UDP checksums, and have atomic transitions that have a subtle relationship to the detailed OS process scheduling.

No attempt was made to validate either the language semantics for MiniCaml (other than to check the evaluation order, which differs between the native-code generator and the bytecode interpreter), or the Udplang OCaml binding we used to test our examples. In the latter case, we assume the OCaml Unix module is a trivial binding to the C sockets interface; our Udplang module does little more.

5 Examples

5.1 The Single Sender We first show the possible traces of the single sender and single receiver from §1.6. Consider

\[
N = \text{ALAN-}e_s | \text{ALAN-HOST}(jfds_{\text{ALAN}}, \text{RUN}, [], [], \text{FALSE})
| \text{KURT-}e_p | \text{KURT-HOST}(jfds_{\text{KURT}}, \text{RUN}, [], [], \text{FALSE})
\]

and discount rules modelling interrupted system calls or the OS running out of file descriptors or kernel memory. Suppose loss (\textit{drop 1}) may occur, but duplication (\textit{dup 1}) and host failure (\textit{host.crash s}) do not.

One behaviour involves message \( m = \text{IP}(i_{\text{ALAN}}, k_{\text{KURT}}, \text{UDP}(\uparrow p_1, \uparrow 7654, \text{"hello"})) \) (for \( p_1 \) ephemeral) being successfully sent, with observable trace

\[
N \xrightarrow{\text{KURT-console "ready"}} \xrightarrow{\text{ALAN-console "sending"}} \xrightarrow{\text{KURT-console "hello"}} N'
\]

and resulting state

\[
N' = \text{ALAN-RET\text{void}} | \text{ALAN-HOST}(jfds_{\text{ALAN}}, \text{TERM}, [], [], \text{FALSE})
| \text{KURT-RET\text{void}} | \text{KURT-HOST}(jfds_{\text{KURT}}, \text{TERM}, [], [], \text{FALSE})
\]
It is also possible for the "hello" to be received and printed with the message $m$ arriving at KURT after KURT’s bind but before the output of "ready", giving trace

\[ N \xrightarrow{\text{ALAN-console "sending"}} \text{KURT-console "ready"} \xrightarrow{\text{KURT-console "hello"}} N' \]

ending in the same state. If message $m$ arrives at KURT before KURT’s bind, however, it will be discarded, giving a trace

\[ N \xrightarrow{\text{ALAN-console "sending"}} \text{KURT-console "ready"} \xrightarrow{\text{N''}} \]

ending with ALAN’s state terminated as before but KURT in a blockedRecvFrom2 state. Here KURT may or may not generate an ICMP, which may or may not be delivered to ALAN in time to set the socket error flag, but as the socket is not used again and is removed on exit, this is not visible.

Finally, there are two observable traces if message $m$ is lost: the trace above and its permutation. In both ALAN runs to completion and KURT remains blocked; no ICMPs are generated.

### 5.2 The Single Heartbeat

As a more realistic example, we present code for a simple heartbeat algorithm, a program $e_A$ that checks the status of another program $e_B$ (which one might think of running as part of a large application):

```plaintext
$e_A =$
let $p = \text{port of int (7655)}$ in
let $i = \text{ip of string ("192.168.0.11")}$ in
let $fd = \text{socket()}$ in
let $\_ = \text{bind}(fd, *, \_\_p)$ in
let $\_ = \text{connect}(fd, i, \_\_p)$ in
let $\_ = \text{print_endline_flush "pinging"}$ in
let $\_ = \text{sendto}(fd, *, "ping", \_\_false)$ in
if $fds = []$ then
  \text{print_endline_flush "dead"}
else try
  let $(\_\_ v) = \text{recvfrom}(fd, \_\_false)$ in
  \text{print_endline_flush v}
  with UDP(ECONNREFUSED)
    \rightarrow \text{print_endline_flush "down"}
```

Program $e_B$, which should be run on KURT, displays "ready" on the console, waits for a message from ALAN on a known port, and responds with an "ack" message when the message arrives.

Program $e_A$, which should be run on ALAN, displays "pinging" and checks the status of the remote machine KURT by sending a message on the known port. It then waits up to five seconds for a response (either a UDP reply datagram
or an ICMP\textunderscore PORT\_UNREACH error). If there is none, it displays "dead"; if the response is a UDP datagram it displays its contents to indicate KURT is alive; and if the response is an ICMP it displays "down" to indicate that KURT is running but the responder thread $e_B$ is down. Note that $e_A$ will print "dead" if KURT is really dead, but it may also do so if the initial datagram is lost, or if the reply datagram or ICMP is lost, or if the reply ICMP is not generated.

Again discount rules modelling interrupted system calls or the OS running out of resources, but now allow loss, duplication and failure. Assuming further that only $e_A$ and $e_B$ run, on an otherwise-quiet network, we can prove that no uncaught exceptions arise during the execution of $e_A$. No errors can arise from any line of $e_A$ apart from the recvfrom call, and the only error this may return is ECONNREFUSED. This means we are justified in omitting all error handling from the code of $e_A$. Further, we can show that the sendto and recvfrom calls in $e_A$ will never block. On the other hand, the message duplication rule dup.1 means that $e_B$ might block temporarily in the sendto call, if the output queue has been filled with ICMP\_PORT\_UNREACH messages generated by "ping" messages arriving before the bind call, but at least one "ping" arrives after the bind. It is still guaranteed that no system call in $e_B$ will fail.

6 Related Work

Work on the mathematical underpinnings of distributed systems has been carried out in the fields of distributed algorithms, process calculi, and programming language semantics. Distributed algorithms research has developed sophisticated algorithms, often dealing with failure, and proofs of their properties, for example using the IO automata of Lynch et al. [Lyn96] and the TLA of Lamport [Lam94]. Work on process calculi has emphasised operational equivalences and compositional descriptions of processes, and recently systems with dynamic local name generation - with calculi based on the π-calculus of Milner, Parrow and Walker [MPW92]. A few calculi have dealt with failure, including [AP94, FGL+96, RH97, BH00]. Building on process calculi, a number of concurrent or distributed programming languages have been designed, with associated semantic work, including among others Occam, Facile, CML, Pict, JoCam, and Nomadic Pict [INMS87, TLK96, Rep91, PT00, FGL+96, WS00]. Little of this work, however, deals with the core network protocols, and as far as we are aware none addresses the level of abstraction of the sockets interface. Further, most does not support reasoning about executable code (or adopts a much higher level of abstraction). The most relevant work is discussed below.

The IOA Language [GLV00] is a language for expressing IO automata directly. Work on proof tools and compilation is ongoing. This will allow reasoning about executable sophisticated distributed algorithms that interact with the network using higher-level abstractions than the sockets library, modulo correctness of the compiler. Using IOA rather than conventional programming languages aids reasoning, but may reduce the applicability of the method.

The approach of Arts and Dann [AD99] is similar to ours: they aim to prove properties of real concurrent programs written in Erlang. They describe an oper-
ational semantics for a subset of Erlang, a logic for reasoning about this subset, and use an automated tool to verify that a program satisfies properties expressed in the logic.

Less closely related, Biagioni implemented TCP/IP in ML [Bia94] as part of the Fox project, and the Ensemble system of [Hay98] provides group communication facilities above UDP. The latter is implemented in OCaml; some verification of optimisations to the Ensemble protocol endpoint code has been carried out. Neither involve a semantics of the network (or, for Ensemble, the underlying sockets implementation), however. At a lower level, work on the semantics of active networks [Swi01] has developed proofs of routing algorithms. Related work on monitoring protocol implementations – TCP in particular – from outside the hosts is presented in [BCMG01].

7 Conclusion

We have described a model that gives a rigorous understanding of programming with sockets and UDP, validated against actual systems. This demonstrates that an operational treatment of this level of network programming – traditionally regarded as beyond the scope of formal semantics – is feasible.

The model provides a basis for two directions of future work. Firstly, we plan to investigate the verification of more interesting examples, developing proof techniques that build on those of both the distributed algorithm and process calculus communities. Secondly, we plan to extend the model to cover a larger fragment of network programming, in a number of ways; we are considering machine support for managing the large definitions that will certainly result. We intend to define other language bindings, e.g. for a Java fragment. Incorporating fairness and time is required to capture interesting properties of algorithms. As discussed in §4, we plan to apply our validation tools to other operating systems, to identify a common semantic core. Finally, we would like to address more of the points listed in §1.7, especially aspects of TCP and multi-threaded hosts.

Acknowledgements Sewell is funded by a Royal Society University Research Fellowship. Serjantov and Wansbrough are funded by EPSRC research grant GRN24872 Wide-area programming: Language, Semantics and Infrastructure Design.

References

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