The UDP Calculus: Rigorous Semantics for Real Networking

Andrei Serjantov Peter Sewell Keith Wansbrough

Computer Laboratory
University of Cambridge

{Andrei.Serjantov,Peter.Sewell,Keith.Wansbrough}@cl.cam.ac.uk

http://www.cl.cam.ac.uk/users/pes20/Netsem

July 16, 2001

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.

Table of Contents

The UDP Calculus i
Andrei Serjantov *Peter Sewell* *Keith Wansbrough*
1 Introduction 1
1.1 Background and Problem 1
1.2 Contribution 1
1.3 Experimental Semantics 1
1.4 Overview 2
1.5 Background: Networks and Protocols, Informally 2
1.6 Background: The Sockets Interface, Informally 3
1.7 Choices: What to Model? 4
1.8 Structuring the Model (and Language Independence) 5
1.9 It's Not Really So Easy 6
2 UDP – The Model 7
2.1 Statics: Types, Values, and Judgements 7
2.1.1 Messages 7
2.1.2 Hosts and Threads 8
2.1.3 Interfaces (1) 11
2.1.4 Sockets 11
2.1.5 The Sockets Interface 12
2.1.6 Networks 14
2.2 Dynamics: Interaction 14
2.2.1 Thread and Host Interaction 14
2.2.2 Host and Network Interaction 15
2.2.3 Host and Console Interaction 16
2.2.4 Host LTS Semantics 16
2.2.5 Thread LTSs and Language Independence 16
2.2.6 Network Operational Semantics 17
2.3 An Overview of the Host Semantics 18
2.3.1 Example Message Transmission 19
2.3.2 Example Message Reception 20
2.3.3 Ports: Privileged, Ephemeral, and Unused, and Autobinding 20
2.3.4 Message Delivery to the Net 21
2.3.5 Return From a Fast Call 21
2.3.6 Message Delivery from the Net 21
2.3.7 Wildcard IP Addresses and outroute 22
2.3.8 ICMP Generation 23
2.3.9 Asynchronous Errors 23
2.3.10 Local Errors 24
2.3.11 Errors and Nondeterminism 25
2.3.12 Loopback 25
2.3.13 Console Interaction 26
2.3.14 Thread Termination 26
2.3.15 Select Timeout 26
2.3.16 Message Size 26
2.3.17 Loss 27
2.3.18 The Outqueue 27
2.3.19 Dosend 28
2.3.20 Matching and Lookup 28
2.3.21 LIB Design: the Thin Abstraction Layer 29

2.3.22 Socket Options and Other Flags, and the LIB Design . . . 29
2.3.23 IP Addressing: Loopback and Martians . . . 30
2.3.24 Interfaces (2) . . . 31
2.4 Sanity Properties . . . 31
3 The Model: Detailed Operational Semantics . . . 34
3.1 SOCKET . . . 34
3.2 BIND . . . 34
3.3 CONNECT . . . 36
3.4 DISCONNECT . . . 36
3.5 GETSOCKNAME, GETPEERNAME . . . 37
3.6 GETERR, GETSOCKOPT, SETSOCKOPT . . . 37
3.7 SENDTO . . . 38
3.8 RECVFROM . . . 40
3.9 CLOSE . . . 41
3.10 SELECT . . . 41
3.11 GETIFADDRS . . . 42
3.12 CONSOLE IO . . . 43
3.13 CONVERSIONS . . . 43
3.14 EBADF/ENOTSOCK . . . 44
3.15 ENOBUFS/ENOMEM AND EINTR . . . 44
3.16 EXIT . . . 45
3.17 RET . . . 45
3.18 DELIVERY . . . 46
4 MiniCaml . . . 49
4.1 Syntax . . . 49
4.2 Typing . . . 49
4.3 Operational Semantics . . . 49
4.4 Sanity Properties . . . 50
4.5 Implementation . . . 50
5 Validation . . . 51
5.1 What We Did . . . 51
5.2 Limitations . . . 51
5.3 Idealisation and Abstraction . . . 52
5.4 MiniCaml . . . 53
6 Examples . . . 54
6.1 The Single Sender . . . 54
6.2 The Single Heartbeat . . . 55
7 Related Work . . . 56
8 Conclusion . . . 57
A MiniCaml Definition . . . 58
A.1 Syntax . . . 58
A.2 Typing . . . 59
A.3 Operational Semantics . . . 59
A.4 Sanity Properties . . . 63
B MiniCaml Implementation: `udplang.mli` . . . 64
C The Single Sender: State Spaces . . . 65

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 Sections 2 and 3 we describe the model, making these subtleties precise. Section 4 outlines the MiniCaml programming language we adopt for expressing distributed programs, a fragment of OCaml 3.00 [L+00]. The details, most of which are standard, are deferred to Appendices A and B.

Section 5 discusses our experimental setup and validation. The semantics is illustrated with a few small examples in Section 6. Finally, we discuss related work and conclude in Sections 7 and 8.

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(we discuss in the remainder of the paper how the model relates to actual systems). 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

$$\mathrm{IP}(i_1, i_2, body)$$

where i_1 and i_2 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. We can therefore abstract from routing and network topology, and depict a network as below (in fact this is our test network). We can therefore abstract from routing and network topology, and depict a network as below (in fact this is our test network). Delivery is asynchronous and unreliable – IP does not provide acknowledgments that datagrams are received, or retransmit lost messages.

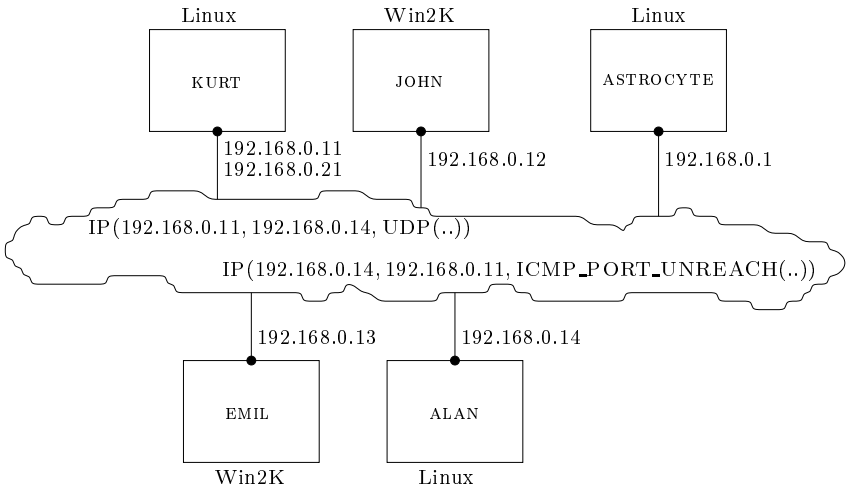
UDP (the *User Datagram Protocol*) is a thin layer above IP that provides multiplexing. It associates a set $\{1, .., 65535\}$ of *ports* to each machine; a UDP datagram

$$\mathrm{IP}(i_1, i_2, \mathrm{UDP}(ps_1, ps_2, data))$$

is an IP datagram with a body of the form $\mathrm{UDP}(ps_1, ps_2, data)$, 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:

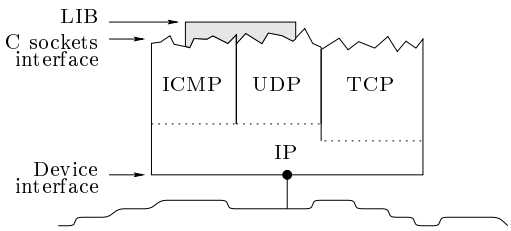
$$\mathrm{IP}(i_1, i_2, \mathrm{ICMP_PORT_UNREACH}(i_3, ps_3, i_4, ps_4)), \text{ and}$$
$$\mathrm{IP}(i_1, i_2, \mathrm{ICMP_HOST_UNREACH}(i_3, ps_3, i_4, ps_4)).$$



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 retransmission 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 §2.1.5).



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 behaviour. 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).

$e_s =$

```
let p = port_of_int 7654 in
let i = ip_of_string "192.168.0.11" in
let fd = socket() in
let _ = connect(fd, i, ↑p) in
let _ = print_endline_flush "sending" in
sendto(fd, *, "hello", FALSE)
```

$e_r =$

```
let p' = port_of_int 7654 in
let i' = ip_of_string "192.168.0.11" in
let fd' = socket() in
let _ = bind(fd', ↑i', ↑p') in
let _ = print_endline_flush "ready" in
let (_, _, v) = recvfrom(fd', FALSE) in
print_endline_flush v
```

Here the $*$ and $\uparrow$ are the constructors of option types $T\uparrow$. The example involves types fd of file descriptors, ip of IP addresses, and port of ports 1..65535, and library calls

socket	: ()	→ fd
bind	: fd * ip↑ * port↑	→ ()
connect	: fd * ip * port↑	→ ()
sendto	: fd * (ip * port)↑ * string * bool	→ ()
recvfrom	: fd * bool	→ ip * port↑ * string
port_of_int	: int	→ port
ip_of_string	: string	→ ip
print_endline_flush	: string	→ ()

(these calls may raise exceptions indicating a variety of errors). 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 (*eg.* changing IP addresses)

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 – *eg.* 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 §2.1.6, which uses the grammar below.

$N ::=$	0	empty
	$N \mid N$	parallel composition
	$\mathrm{IP}\, v$	IP datagram in transit
	$n{\cdot}\textsc{Host}\, v$	host n, with state v
	$n{\cdot}e$	thread of host n, with state e

The host semantics – the heart of the model – is outlined in §2.3. The behaviour of networks is defined in §2.2.6 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 introduce the *UDP Calculus*, our model of the network and of the sockets interface to UDP. This section gives the basic structure and definitions, and explanation of important points. The heart of the model, the detailed operational semantic rules for hosts, is given in Section 3. 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 host semantics by means of some key rules, and Section 2.4 discusses some sanity results. For the most part we speak as if the model and actual systems coincide, though some points of abstraction, idealisation and modelling choice are discussed. This section is arranged in a tutorial order rather than the strict bottom-up order of definitions; an index is provided at the end of the document.

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* $\mathrm{C} \in \mathrm{Con}$ 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 $\rightarrow$, suggestively. Values are terms of

$$v ::= \mathrm{C}\, v_1..v_n \qquad \mathrm{arity}(\mathrm{C}) = n$$

(quotiented by equations for finite set equality at $T\,\mathsf{set}$ types) with typing judgement $\vdash v : T$ defined by the rule:

$$\frac{\mathrm{C} : T_1 \rightarrow .. \rightarrow T_n \rightarrow T \quad \vdash v_i : T_i \quad i = 1..n}{\vdash \mathrm{C}\, v_1..v_n : T}$$

In addition, a number of invariants, on messages, interface descriptor sets, socket descriptors, and hosts, are expressed by judgements

$$\vdash v\ \mathsf{msg\text{-}ok} \qquad \vdash v\ \mathsf{ifd\text{-}set\text{-}ok} \qquad \vdash v\ \mathsf{socket\text{-}ok}(\mathit{ifds}) \qquad \vdash v\ \mathsf{host\text{-}ok}$$

(defined below) which strengthen the associated typing statements. The main formation judgement, for networks, has the form $\vdash N\ \mathsf{network}$.

Notation: We typically let i, p, e, tm range over values of types ip, port, error, int, and is, ps, es, tms over values of types ip↑, port↑, error↑, int↑. The infix list append operation is written @, we write $[e_1, .., e_n]$ for $e_1 :: (... :: (e_n :: \mathrm{NIL}))$, and where $e_1 : T\,\mathsf{list}$ and $e_2 : T$ we abuse notation by writing $e_1 :: e_2$ for $e_1 @ [e_2]$. We use list comprehensions $[x \mid x \in \mathit{xseq} \wedge P(x)]$. For a set X, write orderings(X) for the obvious set of lists.

2.1.1 Messages IP datagrams m, which may carry either a UDP or ICMP payload, are values of type msg, all of which are of one of the forms below.

$$\mathrm{IP}(i_1, i_2, \mathrm{UDP}(ps_1, ps_2, \mathit{data}))$$
$$\mathrm{IP}(i_1, i_2, \mathrm{ICMP_PORT_UNREACH}(i_3, ps_3, i_4, ps_4))$$
$$\mathrm{IP}(i_1, i_2, \mathrm{ICMP_HOST_UNREACH}(i_3, ps_3, i_4, ps_4))$$

Here $i_1 : \mathsf{ip}$ and $i_2 : \mathsf{ip}$ are the source and destination IP addresses. UDP datagrams contain a source and destination port $ps_1 : \mathsf{port}\uparrow$ and $ps_2 : \mathsf{port}\uparrow$, and a string of octets $\mathit{data} : \mathsf{string}$. In unusual cases the ports may be zero (see §2.3.19) so we use port↑ rather than simply port. ICMP datagrams

$T ::=$	int		TL
	bool		
	string		
	()	unit type	
	$T_1 * .. * T_n$	tuple $(n \geq 2)$	
	T list	list	
	$T\uparrow$	optional type	
	T err	T or error	
	void	empty type	
	fd	file descriptor	
	ip	IP address	
	port	port	
	error	OS error	
	netmask	netmask	
	ifid	interface descriptor	
	sockopt	socket options	
	T set	finite set	
	ipBody	body of IP datagram	
	msg	IP datagram	
	ifd	interface descriptor table entry	
	flags	flags from socket descriptor table entry	
	socket	socket descriptor table entry	
	hostid	unique identifier of a host	
	hostThreadState	the OS view of a thread	
	host	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.

Fig. 1. Types

contain the source and destination IP addresses and ports of the UDP datagram that generated them, $i_3, i_4 : \mathsf{ip}$ and $ps_3, ps_4 : \mathsf{port}\uparrow$. We expect any reasonable ICMP to have $i_1 = i_4$ and $i_2 = i_3$, but do not enforce this.

The maximum size of *data* is UDPpayloadMax = 65507 (see §2.3.16). Define $\vdash \mathrm{IP}(i_1, i_2, body)\ \mathsf{msg\text{-}ok}$ if $\vdash \mathrm{IP}(i_1, i_2, body) : \mathsf{msg}$ and also $body = \mathrm{UDP}(ps_1, ps_2, data)$ implies $\mathrm{size}(data) \leq \mathrm{UDPpayloadMax}$.

Certain pathological IP addresses, and messages containing them, are described as *martian*; we define a predicate martian(m) in §2.3.23. In certain contexts such addresses do not appear; we therefore define $\vdash \mathrm{IP}(i_1, i_2, body)\ \mathsf{msg\text{-}oq\text{-}ok}$ if $\vdash \mathrm{IP}(i_1, i_2, body)\ \mathsf{msg\text{-}ok} \wedge i_1 \notin \mathrm{MARTIAN} \wedge (body = \mathrm{ICMP_PORT_UNREACH}(i_3, ps_1, i_4, ps_2, data) \implies \{i_3, i_4\} \cap \mathrm{MARTIAN} = \emptyset) \wedge \neg(\exists v.\, body = \mathrm{ICMP_HOST_UNREACH}\ v)$.

2.1.2 Hosts and Threads We separate a running machine into two parts: the *host*, representing the machine itself and its operating system; and the *thread*, representing the application program controlling it. Threads are explained in §2.2.5. A host is of the form:

$$\textsc{Host}(ifds, t, s, oq, oqf)$$

A host has a set $ifds : \mathsf{ifd\ set}$ of interfaces, each with a set of IP addresses. Interfaces are explained in §§2.1.3, 2.3.24; 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

Partition Con into the language constructors:

$.., -1, 0, 1, 2, ..$	: int
TRUE, FALSE	: bool
octet-sequence	: string
()	: ()
(, .. ,) (mixfix)	: $T_1 \to .. \to T_n \to T_1 * .. * T_n \quad n \geq 2$
NIL	: T list
:: (infix)	: $T \to T$ list $\to T$ list
*	: $T\uparrow$
$\uparrow$	: $T \to T\uparrow$
OK	: $T \to T$ err
FAIL	: error $\to T$ err
$1..2^{32} - 1$	: ip
1..65535	: port
$\text{FD}_3, \text{FD}_4, \ldots$	: fd
LO, ETH0, ETH1, ...	: ifid
$\sum_{i \in j..31} 2^i$	: netmask for $0 \leq j \leq 31$
SO_BSDCOMPAT, SO_REUSEADDR	: sockopt
EACCES, EADDRINUSE, EADDRNOTAVAIL, EAGAIN, EBADF, ECONNREFUSED, EHOSTUNREACH, EINTR, EINVAL, EMFILE, EMSGSIZE, ENFILE, ENOBUFS, ENOMEM, ENOTCONN, ENOTSOCK	: error

and the non-language constructors:

IP	: ip * ip * ipBody	$\to$	msg
UDP	: port$\uparrow$ * port$\uparrow$ * string	$\to$	ipBody
ICMP_HOST_UNREACH	: ip * port$\uparrow$ * ip * port$\uparrow$	$\to$	ipBody
ICMP_PORT_UNREACH	: ip * port$\uparrow$ * ip * port$\uparrow$	$\to$	ipBody
HOST	: ifd set * hostThreadState * socket list * msg list * bool	$\to$	host
SOCK	: fd * ip$\uparrow$ * port$\uparrow$ * ip$\uparrow$ * port$\uparrow$ * error$\uparrow$ * flags * (msg * ifid) list	$\to$	socket
IF	: ifid * ip set * ip * netmask	$\to$	ifd
RUN	: hostThreadState		
TERM	: hostThreadState		
RET_{TL}	: TL	$\to$	hostThreadState
SENDTO2	: fd * (ip * port)$\uparrow$ * string	$\to$	hostThreadState
RECVFROM2	: fd	$\to$	hostThreadState
SELECT2	: fd list * fd list * int$\uparrow$	$\to$	hostThreadState
PRINT2	: string	$\to$	hostThreadState
FLAGS	: bool * bool	$\to$	flags
ALAN, KURT, ASTROCYTE, ...	: hostid		

Elements of T set are written $\{v_1, .. , v_n\}$. The TL subscript of RET_{TL} will usually be elided.

Fig. 2. Constructors

view of the thread state is stored in $t : \mathsf{hostThreadState}$: the thread may be running (RUN), terminated (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 fast system call (RET v) or the thread may be blocked waiting for a slow system call to complete (SENDTO2 v, RECVFROM2 v, SELECT2 v, PRINT2 v). The host's current list of sockets is given by $s : \mathsf{socket\,list}$. The *outqueue*, a queue of outbound IP messages, is given by $oq : \mathsf{msg\,list}$ and $oqf : \mathsf{bool}$, where oq is the list of messages and oqf is set when the queue is full (see §2.3.18).

The host invariant $\vdash \textsc{Host}\ v\ \mathsf{host\text{-}ok}$ requires that the interface set $ifds$, sockets s and messages in the outqueue oq satisfy their respective invariants, if the outqueue oq is empty then the outqueue full flag oqf is FALSE, that the host thread state is sensible, and that no two sockets share a file descriptor. It is defined by the rule

$$\frac{\begin{array}{l}\vdash ifds\ \mathsf{ifd\text{-}set\text{-}ok}\\ \vdash t : \mathsf{hostThreadState}\\ \vdash s : \mathsf{socket\,list} \wedge \forall \textsc{Sock}\ v \in s\,.\ \vdash \textsc{Sock}\ v\ \mathsf{socket\text{-}ok}(ifds)\\ \vdash oq : \mathsf{msg\,list} \wedge \forall m \in oq\,.\ \vdash m\ \mathsf{msg\text{-}oq\text{-}ok}\\ \vdash oqf : \mathsf{bool}\\ oq = [\,] \implies oqf = \textsc{False}\\ t = \textsc{Sendto2}\ (fd, ips, data) \implies fd \in \mathrm{sockfds}(s) \wedge (ips \neq * \vee is_2(s, fd) \neq *) \wedge ps_1(s, fd) \neq *\\ t = \textsc{Recvfrom2}\ fd \implies fd \in \mathrm{sockfds}(s) \wedge ps_1(s, fd) \neq *\\ t = \textsc{Select2}\ (readseq, writeseq, tms) \implies readseq @ writeseq \subseteq \mathrm{sockfds}(s) \wedge \neg \exists tm < 0\,.\ tms = \uparrow tm\\ \neg(s = s_1 @ \textsc{Sock}\ v_1 @ s_2 @ \textsc{Sock}\ v_2 @ s_3 \wedge \mathrm{sockfd}(\textsc{Sock}\ v_1) = \mathrm{sockfd}(\textsc{Sock}\ v_2))\end{array}}{\vdash \textsc{Host}(ifds, t, s, oq, oqf)\ \mathsf{host\text{-}ok}}$$

Here, for $\vdash \textsc{Sock}\ v : \mathsf{socket}$ and $\vdash s : \mathsf{socket\,list}$, define

$$\begin{array}{ll}\mathrm{sockfd}(\textsc{Sock}(fd, _, _, _, _, _, _)) & = fd\\ \mathrm{sockfds}(s) & = \mathrm{map\ sockfd}\ s\end{array}$$

For $s = s_1 @ \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) @ s_2$ a socket list with no repeated fd's, take $is_1(s, fd) = is_1$ *etc.*.

For convenience, we define several classes of contexts that build a host, and one that builds a socket list:

$$\begin{array}{lll}H(_) & ::= & \textsc{Host}(ifds, _, s, oq, oqf)\\ Q(__1, __2, __3) & ::= & \textsc{Host}(ifds, __1, __2, oq, __3)\\ F(__1, __2, __3) & ::= & \textsc{Host}(__1, __2, s_1 @ [__3] @ s_2), oq, oqf)\\ S(_) & ::= & s_1 @ [_] @ s_2\end{array}$$

There are no implicit typing constraints on these contexts, but we will only use well-typed instantiations of them. For F as above, define $\mathrm{socks}(F) = s_1 @ s_2$ (this will only be used where the socket in the hole of F is fully wildcarded). We abuse notation, writing $\textsc{Sock}\ v \in F$ for $\textsc{Sock}\ v \in \mathrm{socks}(F)$.

It is also useful to pick out the host thread states that are specific to a single fd:

$$\begin{array}{lll}OP_2(_) & ::= & \textsc{Sendto2}\ (_, v, data)\\ & & \textsc{Recvfrom2}\ (_)\end{array}$$

Define htsType(t), giving either RUN or a language type for each t : hostThreadState, as below. This gives the type of values (if any) to be returned to the thread. It is lifted to hosts in the obvious way.

$$\begin{array}{ll}
\text{htsType}(\textsc{Run}) & = \textsc{Run} \\
\text{htsType}(\textsc{Ret}\ TL) & = TL \\
\text{htsType}(\textsc{Sendto2}\ v) & = () \ \mathsf{err} \\
\text{htsType}(\textsc{Recvfrom2}\ v) & = (\mathsf{ip} * \mathsf{port}\uparrow * \mathsf{string})\ \mathsf{err} \\
\text{htsType}(\textsc{Select2}\ v) & = (\mathsf{fd\ list} * \mathsf{fd\ list})\ \mathsf{err} \\
\text{htsType}(\textsc{Print2}\ v) & = ()\ \mathsf{err} \\
\text{htsType}(\textsc{Term}) & = \mathsf{void}
\end{array}$$

2.1.3 Interfaces (1) Each host has a set *ifds* of interfaces, of the form

$$\{\, \text{IF}(\mathit{ifid}_j, \mathit{iset}_j, \mathit{iprimary}_j, \mathit{netmask}_j) \mid j \in J \,\},$$

where each ifid_j : ifd is an interface identifier (unique within this host), iset_j : ip set is the set of IP addresses of this interface, $\mathit{iprimary}_j$: ip is its primary IP address (which must be in iset_j), and $\mathit{netmask}_j$: netmask is its netmask. We sometimes abuse notation, writing $i \in \mathit{ifds} \iff \exists j \in J.i \in \mathit{iset}_j$. We suppose each host has a loopback interface and at least one other, and impose a number of other sanity conditions, defining $\vdash \mathit{ifds}$ ifd-set-ok in §2.3.24.

2.1.4 Sockets The central abstraction of the sockets interface is the *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

$$\textsc{Sock}(\mathit{fd}, \mathit{is}_1, \mathit{ps}_1, \mathit{is}_2, \mathit{ps}_2, \mathit{es}, f, \mathit{mq})$$

A socket is uniquely identified within the host by its file descriptor fd : fd. (Note that file descriptors of closed sockets may be reused.) The local and remote address/port pairs are is_1 : ip↑, ps_1 : port↑ and is_2 : ip↑, ps_2 : port↑ respectively; wildcards may occur. These 4-tuples are used both for filling in the details in outgoing messages and for matching incoming messages, to determine which socket (if any) they should be delivered to. Asynchronous error conditions store the pending error in the error flag es : error↑. An assortment of socket parameters are stored in f : flags. The elements of flags are $\textsc{Flags}(\mathit{bc}, \mathit{ra})$, where bc : bool and ra : bool record the SO_BSDCOMPAT and SO_REUSEADDR socket options. Finally, mq : (msg∗ifid) list is a queue of incoming messages that have been delivered to this socket but not yet received by the application.

We define helper functions for manipulating the flags:

$$\begin{array}{ll}
\text{bsdcompat}\ \textsc{Flags}(\mathit{bc}, _) & = \mathit{bc} \\
\text{reuseaddr}\ \textsc{Flags}(_, \mathit{ra}) & = \mathit{ra} \\
\text{setbsdcompat}\ (\textsc{Flags}(_, \mathit{ra}), \mathit{bc}) & = \textsc{Flags}(\mathit{bc}, \mathit{ra}) \\
\text{setreuseaddr}\ (\textsc{Flags}(\mathit{bc}, _), \mathit{ra}) & = \textsc{Flags}(\mathit{bc}, \mathit{ra})
\end{array}$$

The socket invariant $\vdash \textsc{Sock}(\mathit{fd}, \mathit{is}_1, \mathit{ps}_1, \mathit{is}_2, \mathit{ps}_2, \mathit{es}, f, \mathit{mq})$ socket-ok(ifds) is parametric on an *ifds*. It requires the components to be well typed ($\vdash \textsc{Sock}(\mathit{fd}, \mathit{is}_1, \mathit{ps}_1, \mathit{is}_2, \mathit{ps}_2, \mathit{es}, f, \mathit{mq}) : \mathsf{socket}$); the 4-tuple $(\mathit{is}_1, \mathit{ps}_1, \mathit{is}_2, \mathit{ps}_2)$ to be in one of the forms

$$\begin{array}{l}
(*, *, *, *) \\
(*, \uparrow p_1, *, *) \\
(\uparrow i_1, \uparrow p_1, *, *) \\
(\uparrow i_1, \uparrow p_1, \uparrow i_2, *) \\
(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)
\end{array}$$

with $i_1 \in ifds$; the messages in mq to be sensible ($\forall (m, ifid) \in mq.\ \vdash m\,\mathsf{msg\text{-}ok} \wedge \neg\mathrm{martian}(m)$); and if the 4-tuple is $(*,*,*,*)$ then $mq = \textsc{nil}$ and $es = *$.

2.1.5 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 have no need for polymorphic library calls). We take a library interface LIB, shown in Figure 3, consisting of the sockets interface together with some basic OS operations.

The sockets interface:

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

Basic operating system operations:

print_endline_flush	: string	→ () err
exit	: ()	→ void

Fig. 3. The library interface LIB

All of the sockets interface calls return a value of some type T err to the thread, which can be either OK v for $v : T$ or FAIL e for a Unix error e : error. A language binding may map these error returns into exceptions, as the MiniCaml binding of §4 does. The approximate meanings of the sockets interface calls are as follows.

Calling socket() creates a new socket, returning a file descriptor that other calls can use to refer to it. Its local and remote IP/port pairs can be set by bind, connect, and disconnect, and examined with getsockname and getpeername.

A UDP datagram can be sent with sendto($fd, ips, data, b$). The destination address may be specified explicitly ($ips = \uparrow(i, p)$), or left to the socket to determine ($ips = *$). The $data$ is simply a string of octets. The b : bool specifies whether the call should return immediately if the outqueue is full, or block until the datagram can be queued.

A UDP datagram can be received with recvfrom(fd, b). If a message is available (*ie.*, is on the message queue mq of socket fd) then it will be returned, together with its originating IP address and port. Otherwise, the call will return immediately if $b = \textsc{true}$, or block if $b = \textsc{false}$.

A socket may be in a pending error state (*eg.* set by an asynchronously-arriving ICMP message). Various calls will return this error; it can also be explicitly examined and cleared by geterr.

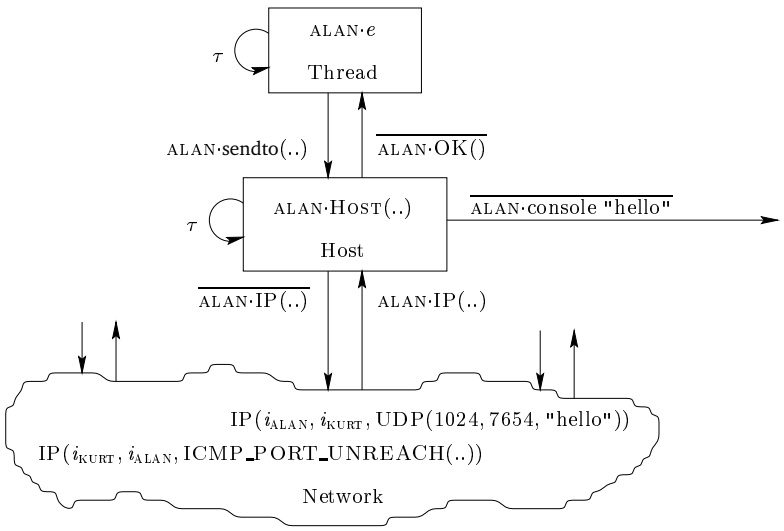


Fig. 4. Thread, Host and Network

Sockets have a number of options, accessed by getsockopt and setsockopt. Option SO_REUSEADDR allows ports to be reused by different sockets. Option SO_BSDCOMPAT affects the delivery of ICMP messages to unconnected sockets.

Calling select($readseq, writeseq, \uparrow tm$) will block until one of the sockets in $readseq$ has a message ready for reading, one of those in $writeseq$ is ready for writing, time tm elapses, or an error occurs. It returns the sublists of all those ready. The non-blocking variant select($readseq, writeseq, *$) returns immediately.

A socket can be closed using close fd, and the interfaces of a host can be examined using getifaddrs.

We include injections port_of_int and ip_of_string for constructing values of types port and ip. In this paper we have no need of injections for netmask or ifid, or the corresponding projections, so they are omitted.

It is useful to pick out the operations that are specific to a single fd: (if $\vdash fd : \mathsf{fd}$ then $OP(fd)$ is an element of the Lhost defined in §2.2.4).

$$
\begin{aligned}
OP(_) ::= \ & \mathsf{bind}(_, is, ps) \\
& \mathsf{connect}(_, i, ps) \\
& \mathsf{disconnect}\ _ \\
& \mathsf{getsockname}\ _ \\
& \mathsf{getpeername}\ _ \\
& \mathsf{geterr}\ _ \\
& \mathsf{getsockopt}(_, opt) \\
& \mathsf{setsockopt}(_, opt, b) \\
& \mathsf{sendto}(_, v, data, nb) \\
& \mathsf{recvfrom}(_, nb) \\
& \mathsf{close}\ _
\end{aligned}
$$

We discuss the design of LIB further in §§2.3.21, 2.3.22.

2.1.6 Networks A network N (a term of the grammar in §1.8) is a parallel composition of IP datagrams, hosts, and their threads. To describe partial systems, we allow hosts and their threads to be split apart. The association between them is expressed by shared names $n : \mathsf{hostid}$, which are purely semantic devices, not to be confused with IP addresses or DNS names. Here e ranges over thread labelled transition systems, defined in §2.2.5 (not to be confused with $e : \mathsf{error}$).

The formation judgement $\vdash N\ \mathsf{network}$ is below. It uses judgements $\mathsf{msg\text{-}ok}$ and $\mathsf{host\text{-}ok}$, which impose some invariants, and ensures that host identifiers and IP addresses are not repeated.

$$\frac{}{\vdash 0\ \mathsf{network}} \qquad \frac{\vdash \mathrm{IP}\ v\ \mathsf{msg\text{-}ok}}{\vdash \mathrm{IP}\ v\ \mathsf{network}} \qquad \frac{\vdash n : \mathsf{hostid} \quad \vdash \textsc{Host}\ v\ \mathsf{host\text{-}ok}}{\vdash n{\cdot}\textsc{Host}\ v\ \mathsf{network}} \qquad \frac{\vdash n : \mathsf{hostid} \quad e \text{ is a thread LTS}}{\vdash n{\cdot}e\ \mathsf{network}}$$

$$\frac{\begin{array}{l}\vdash N_j\ \mathsf{network} \quad j = 0,1 \\ \text{if } n{\cdot}\textsc{Host}\ v \in N_j \text{ then } \neg\exists n{\cdot}\textsc{Host}\ v' \text{ in } N_{1-j} \\ \text{if } n{\cdot}e \in N_j \text{ then } \neg\exists n{\cdot}e' \text{ in } N_{1-j} \\ \text{if } n{\cdot}\textsc{Host}\ v \in N_j \text{ and } n{\cdot}e \in N_{1-j} \text{ then instep}(\textsc{Host}\ v, e) \\ \text{if } n_j{\cdot}\textsc{Host}\ v_j \in N_j, j = 0,1, \text{ then disjoint_addr}(\textsc{Host}\ v_0, \textsc{Host}\ v_1)\end{array}}{\vdash N_0 \mid N_1\ \mathsf{network}}$$

where instep and disjoint_addr are defined in §2.2.5 and §2.3.24 respectively. The empty network is a network. A valid message ($\mathsf{msg\text{-}ok}$; *ie.*, well-typed and not over the maximum size) is a network. A valid host or thread tagged by a hostid is a network. Finally, two networks may be combined if the sets of host tags are disjoint, the sets of thread tags are disjoint, threads and hosts sharing the same tag are in step, and all non-loopback host IP addresses are disjoint.

Note that there is no structure on the messages in transit – they represent all IP datagrams anywhere in the network, both buffered in routers and travelling in physical links.

2.2 Dynamics: Interaction

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

$$\begin{array}{l} N = \textsc{alan}{\cdot}e \mid \textsc{alan}{\cdot}\textsc{Host}(..) \\ \quad \mid \mathrm{IP}(i_{\textsc{alan}}, i_{\textsc{kurt}}, \mathrm{UDP}(1024, 7654, \text{"hello"})) \\ \quad \mid \mathrm{IP}(i_{\textsc{kurt}}, i_{\textsc{alan}}, \mathrm{ICMP_PORT_UNREACH}(..)) \mid \ldots \end{array}$$

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.

2.2.1 Thread and Host Interaction 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 synchronisations. A thread can make a system call $f\ v$ for any $f : TL \to TL'$ in LIB and argument $v : TL$, for example $\mathsf{sendto}(..)$. The operating system may then return a value $r : TL'$, for example OK(). In the

above diagram, the host's ALAN·sendto(..) and $\overline{\text{ALAN·OK()}}$ are part of call and return synchronisations respectively. For example, for $\mathsf{socket} : () \to \mathsf{fd}\ \mathsf{err}$ the (unprefixed) thread and host may have transitions:

thread		host
$C(\mathsf{socket}())$		$\textsc{Host}(\mathit{ifds}, \textsc{Run}, s, \mathit{oq}, \mathit{oqf})$
$\xrightarrow{\overline{\mathsf{socket}()}}$		$\xrightarrow{\mathsf{socket}()}$
$C(\text{RET}_{\mathsf{fd}\ \mathsf{err}})$		$\textsc{Host}(\mathit{ifds}, \textsc{Ret}\ (\text{OK}\ \mathit{fd}), s', \mathit{oq}, \mathit{oqf})$
$\xrightarrow{\text{OK}\ \mathit{fd}}$		$\xrightarrow{\overline{\text{OK}\ \mathit{fd}}}$
$C(\mathit{fd})$		$\textsc{Host}(\mathit{ifds}, \textsc{Run}, s', \mathit{oq}, \mathit{oqf})$

Here the thread states are those of MiniCaml (see §4), with C a reduction context and the store component elided. In the host states, the $\mathsf{hostThreadState}$ component is RUN for states in which no system call is executing, and RET v for states in which a system call is executing, and is just about to return v to the thread.

Invocations of system calls may be *fast* or *slow* [Ste98, 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 §2.1.2). In the host state, the $\mathsf{hostThreadState}$ component is RUN for states in which no system call is executing. A fast system call takes this to RET r, which can immediately perform the transition $\overline{r}$ back to the RUN state. A slow system call, however, introduces additional host states for the intermediate blocked states.

For example, a blocking receive in a state which has no messages queued for socket fd has host transitions

$$\begin{aligned}
&\textsc{Host}(\mathit{ifds}, \textsc{Run}, s, \mathit{oq}, \mathit{oqf}) \\
\xrightarrow{\text{recvfrom}(\mathit{fd}, \text{FALSE})}\ &\textsc{Host}(\mathit{ifds}, \textsc{Recvfrom2}\ \mathit{fd}, s', \mathit{oq}, \mathit{oqf}) \\
\xrightarrow{\ldots \textit{a message is delivered from the network} \ldots}\ &\textsc{Host}(\mathit{ifds}, \textsc{Recvfrom2}\ \mathit{fd}, s'', \mathit{oq}, \mathit{oqf}) \\
\xrightarrow{\overline{\text{OK}(i_3, p_3, \mathit{ifid}, \mathit{data})}}\ &\textsc{Host}(\mathit{ifds}, \textsc{Run}, s''', \mathit{oq}, \mathit{oqf})
\end{aligned}$$

where $\textsc{Host}(.., \textsc{Recvfrom2}\ \mathit{fd}, s', ..)$ is the blocked state (more details are in §2.3.2). (In the absence of slow calls, one could model system calls as single transitions, carrying both argument and return values, rather than pairs.)

2.2.2 Host and Network Interaction A host interacts with the network by sending and receiving IP datagrams: $\overline{\text{ALAN·IP(..)}}$ and ALAN·IP(..) in the figure, respectively.

A host interacts with the network asynchronously to its interactions with the thread, so these host transitions are independent of the host thread state. For example, there may be transitions as below, for an output from a host and input to a host respectively.

host		network
$\textsc{Host}(\mathit{ifds}, t, s, \mathit{oq} :: \text{IP}(i_3, i_4, \mathit{body}), \mathit{oqf})$		N
$\xrightarrow{\overline{\text{IP}(i_3, i_4, \mathit{body})}}$		$\xrightarrow{\text{IP}(i_3, i_4, \mathit{body})}$
$\textsc{Host}(\mathit{ifds}, t, s, \mathit{oq}, \mathit{oqf}')$		$N \mid \text{IP}(i_3, i_4, \mathit{body})$
$\textsc{Host}(\mathit{ifds}, t, s, \mathit{oq}, \mathit{oqf})$		$N \mid \text{IP}(i_3, i_4, \mathit{body})$
$\xrightarrow{\text{IP}(i_3, i_4, \mathit{body})}$		$\xrightarrow{\overline{\text{IP}(i_3, i_4, \mathit{body})}}$
$\textsc{Host}(\mathit{ifds}, t, s', \mathit{oq}, \mathit{oqf})$		N

2.2.3 Host and Console Interaction A host may also emit strings to its console with transitions of the form $\overline{\text{ALAN·console "hello"}}$. This provides a minimal way to observe the behaviour of a network, namely by examining the output on each console. For example, there may be transitions as below.

thread		host
$C(\text{print_endline_flush } v)$		$\textsc{Host}(\mathit{ifds}, \textsc{Run}, s, \mathit{oq}, \mathit{oqf})$
	$\xrightarrow{\overline{\text{print_endline_flush } v}}$	
		$\xrightarrow{\text{print_endline_flush } v}$
$C(\text{RET}_{()\ \mathsf{err}})$		$\textsc{Host}(\mathit{ifds}, \textsc{Print2}\ v, s, \mathit{oq}, \mathit{oqf})$
		$\xrightarrow{\overline{\text{console } v}}$
		$\textsc{Host}(\mathit{ifds}, \textsc{Ret}\ (\text{OK}\ ()), s, \mathit{oq}, \mathit{oqf})$
	$\xrightarrow{\text{OK}\ ()}$	$\xrightarrow{\overline{\text{OK}\ ()}}$
$C(())$		$\textsc{Host}(\mathit{ifds}, \textsc{Run}, s, \mathit{oq}, \mathit{oqf})$

2.2.4 Host LTS Semantics Collecting these interactions together, the Host LTS has labels

$$
\begin{array}{lll}
\text{Lhost} = \{\, f\ v \mid f : TL \to TL' \in \text{LIB} \wedge \vdash v : TL \,\} & & \text{call} \\
\quad \cup \{\, \overline{r} \mid \exists TL.\ \vdash r : TL \,\} & & \text{return} \\
\quad \cup \{\, \text{IP}\ v \mid \vdash \text{IP}\ v\ \mathsf{msg\text{-}ok} \,\} & & \text{message receipt} \\
\quad \cup \{\, \overline{\text{IP}\ v} \mid \vdash \text{IP}\ v\ \mathsf{msg\text{-}ok} \,\} & & \text{message send} \\
\quad \cup \{\, \overline{\text{console } v} \mid \vdash v : \mathsf{string} \,\} & & \text{console output} \\
\quad \cup \{\tau\} & & \text{internal action}
\end{array}
$$

and states $\{\, \textsc{Host}\ v \mid \vdash \textsc{Host}\ v : \mathsf{host} \,\}$. The transition relation is defined by the rules in §3, some of which are illustrated in §2.3. It is the least relation containing

$$\textsc{Host}\ v \xrightarrow{l} \textsc{Host}\ v'$$

for each instance of a rule for which $\vdash \textsc{Host}\ v\ \mathsf{host\text{-}ok}$ and $l \in \text{Lhost}$. A number of properties of the transition relation are stated in §2.4; among them, Theorem 2 shows that all transitions preserve $\vdash \textsc{Host}\ v\ \mathsf{host\text{-}ok}$.

2.2.5 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 $\overline{f\ v}$, r and τ. 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 §4.

Take a *thread LTS* e to be (Lthread, S, $\to$, s_0) where S is a set of states, $s_0 \in S$ is the initial state, $\to\ \subseteq S \times \text{Lthread} \times S$ is the transition relation, and the labels are

$$\text{Lthread} = \{\, \overline{f\ v} \mid f : TL \to TL' \in \text{LIB} \wedge \vdash v : TL \,\} \cup \{\, r \mid \exists TL.\ \vdash r : TL \,\} \cup \{\tau\}$$

Some axioms must be imposed to give an accurate model, as in [Sew97]. 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). We require that the reachable states of S can be partitioned into sets

$\mathrm{CALL}(f)$ $f \in \mathrm{dom}(\mathrm{LIB})$	the thread is just about to make system call f
RET_{TL}	the thread is waiting for a response of language type TL
TAU	the thread is computing internally

such that

1. if $s \in \mathrm{CALL}(f)$ and $f : TL \to TL' \in \mathrm{LIB}$
   then $\exists v, s' .\ \vdash v : TL \wedge s \xrightarrow{\overline{f\ v}} s'$ and $\forall l, s'' .\ s \xrightarrow{l} s'' \implies s'' = s' \wedge l = \overline{f\ v} \wedge s'' \in \mathrm{RET}_{TL'}$
2. if $s \in \mathrm{RET}_{TL}$
   then $\forall r .\ \vdash r : TL \implies \exists_1 s' .\ s \xrightarrow{r} s'$ and $\forall l, s'' .\ s \xrightarrow{l} s'' \implies \exists r .\ \vdash r : TL \wedge l = r \wedge s'' \in \mathrm{PROG}$
3. if $s \in \mathrm{TAU}$ then $\exists s' .\ s \xrightarrow{\tau} s'$ and $\forall l, s'' .\ s \xrightarrow{l} s'' \implies l = \tau \wedge s'' \in \mathrm{PROG}$

where $\mathrm{PROG} \stackrel{def}{=} \mathrm{TAU} \cup \bigcup_{f \in \mathrm{dom}(\mathrm{LIB})} \mathrm{CALL}(f)$ (such a partition is unique if it exists).

The definition allows internal nondeterminism. This occurs in the MiniCaml semantics for technical reasons (fresh references are chosen nondeterministically), and would be important when regarding the semantics of a concurrent language such as Pict [PT00] or Join [FGL+96] as a thread LTS, where the natural operational semantics gives a loose specification of how language-level processes should be scheduled.

For a host $h : \mathsf{host}$ and a thread LTS $e = (\mathrm{Lthread}, S, \to, s_0)$ define

$$\begin{aligned}\mathrm{instep}(h, e) = (s_0 \in \mathrm{PROG} &\iff \mathrm{htsType}(h) = \textsc{Run}) \wedge \\ \forall TL .\ (s_0 \in \mathrm{RET}_{TL} &\iff \mathrm{htsType}(h) = TL)\end{aligned}$$

Often we deal with thread LTSs that are generated by an operational semantics over some expressions $\mathcal{E}$, also ranged over by e. We then confuse the thread LTS $(\mathrm{Lthread}, \mathcal{E}, \to, e)$ with e itself.

The definition of thread LTS, and that of the LIB interface, does not include any notion of marshalling – the sendto and recvfrom calls allow communcation only of strings. Marshalling could be addressed in the language semantics (though MiniCaml does not). If one wished to deal with marshallable values of extensible types (*eg.* new channel names or cryptographic keys) then the definition of thread LTS should be extended with renaming structure, along the lines of [CS00].

2.2.6 Network Operational Semantics Now we define an SOS to glue things together, making precise the synchronisations between threads, hosts, and network datagrams that were described informally above. The labels of a network are as those of threads and hosts, together with 'crash' for machine failure, but tagged with host identifiers n.

$$\begin{aligned}\mathrm{Lnetwork} = &\{\, n{\cdot}l \mid \vdash n : \mathsf{hostid} \wedge l \in \mathrm{Lthread} \setminus \tau \,\} \\ &\cup \{\, n{\cdot}l \mid \vdash n : \mathsf{hostid} \wedge l \in \mathrm{Lhost} \setminus \tau \,\} \\ &\cup \mathrm{Crash} \cup \overline{\mathrm{Crash}} \\ &\cup \{\tau\}\end{aligned}$$

where

$$\begin{aligned}\overline{\mathrm{Crash}} &= \{\, n{\cdot}\overline{\mathrm{crash}} \mid \vdash n : \mathsf{hostid} \,\} \\ \mathrm{Crash} &= \{\, n{\cdot}\mathrm{crash} \mid \vdash n : \mathsf{hostid} \,\}\end{aligned}$$

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, $\vdash n : \mathsf{hostid}$, and $\vdash N_i$ network. As usual, for a label l we write $\overline{l}$ for the label as l but overbarred iff l is not, for $l \neq \tau$, and $\overline{\tau} = \tau$. For a set L of labels $\overline{L} = \{\, \overline{l} \mid l \in L \,\}$.

$$\frac{x \xrightarrow{l} x' \quad l \neq \tau}{n{\cdot}x \xrightarrow{n\cdot l} n{\cdot}x'} \qquad \frac{x \xrightarrow{\tau} x'}{n{\cdot}x \xrightarrow{\tau} n{\cdot}x'} \qquad \frac{}{0 \xrightarrow{n\cdot \mathrm{IP}\ v} \mathrm{IP}\ v} \qquad \frac{}{\mathrm{IP}\ v \xrightarrow{n\cdot \overline{\mathrm{IP}\ v}} 0}$$

$$\frac{N_1 \xrightarrow{n\cdot l} N_1' \quad N_2 \xrightarrow{n\cdot \overline{l}} N_2'}{N_1 \mid N_2 \xrightarrow{\tau} N_1' \mid N_2'}\ \textit{par.1} \qquad \frac{N_1 \xrightarrow{n\cdot l} N_1' \quad l \in \mathrm{Lthread} \cup \mathrm{Crash} \implies n{\cdot}\textsc{Host}\ v \notin N_2 \quad l \in \overline{\mathrm{Lthread}} \cup \overline{\mathrm{Crash}} \implies n{\cdot}e \notin N_2}{N_1 \mid N_2 \xrightarrow{n\cdot l} N_1' \mid N_2}\ \textit{par.2}$$

$$\frac{}{0 \xrightarrow{n\cdot \mathrm{IP}\ v} 0}\ \textit{drop.1} \qquad \frac{k \geq 2}{0 \xrightarrow{n\cdot \mathrm{IP}\ v} \prod_{j \in 1..k} \mathrm{IP}\ v}\ \textit{dup.1}$$

$$\frac{}{n{\cdot}\textsc{Host}\ v \xrightarrow{n\cdot \overline{\mathrm{crash}}} 0}\ \textit{host.crash.1} \qquad \frac{}{n{\cdot}e \xrightarrow{n\cdot \mathrm{crash}} 0}\ \textit{host.crash.2}$$

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 π-calculus [HT91]. The *par.*$*$ rules force synchronisations between a host and a thread that share a name. IP datagrams can arrive out of order, be lost or be (finitely) duplicated. Re-ordering is built into the first two lines of rules, but for the other kinds of failure we add the rules *drop.1*, *dup.1* and *host.crash.*$*$. These are most interesting when constrained, *eg.* by fairness or timing assumptions. Hosts can fail in a variety of ways. In this paper we consider only the simplest, 'crash' failure [Mul93, §2.4]. The host failure model could be made richer in at least 3 directions: by allowing restart (and checkpointing of some non-volatile state); by allowing network partitions (or perhaps only disconnections of single machines); and by allowing Byzantine failure.

It is worth noting that delivery of an IP datagram to a host consumes the datagram, so this would not be an accurate model if we allowed hosts to share IP addresses. Note also that datagrams for IP addresses that do not belong to a host are not garbage-collected.

2.3 An Overview of the Host Semantics

We now explain the main points of the host semantics, illustrating some of the host transition axioms.

2.3.1 Example Message Transmission We begin by outlining the host transitions involved in one successful execution of the single datagram example from §1.6. The combined system is revisited in §6.1. First, consider the transitions of the sender host ALAN, running e_s. We have

$$
\begin{aligned}
&\textsc{Host}(\mathit{ifds}_{\text{ALAN}}, \textsc{Run}, [\,], [\,], \textsc{false}) \\
&\xrightarrow{\text{port_of_int 7654}}\xrightarrow{\overline{\text{OK}\,p}} \\
&\xrightarrow{\text{ip_of_string "192.168.0.11"}}\xrightarrow{\overline{\text{OK}\,i}} \\
&\xrightarrow{\text{socket()}}\xrightarrow{\overline{\text{OK}\,fd}} \textsc{Host}(\mathit{ifds}_{\text{ALAN}}, \textsc{Run}, [\textsc{Sock}(fd, *, *, *, *, *, f, [\,])], [\,], \textsc{false}) \\
&\xrightarrow{\text{connect}(fd,i,\uparrow p)}\xrightarrow{\overline{\text{OK()}}} \textsc{Host}(\mathit{ifds}_{\text{ALAN}}, \textsc{Run}, [\textsc{Sock}(fd, \uparrow i_1, \uparrow p_1, \uparrow i, \uparrow p, *, f, [\,])], [\,], \textsc{false}) \\
&\xrightarrow{\text{print_endline_flush "sending"}} \textsc{Host}(\mathit{ifds}_{\text{ALAN}}, \textsc{Print2 "hello"}, [\textsc{Sock}(fd, \uparrow i_1, \uparrow p_1, \uparrow i, \uparrow p, *, f, [\,])], [\,], \textsc{false}) \\
&\xrightarrow{\overline{\text{console "sending"}}}\xrightarrow{\overline{\text{OK()}}} \textsc{Host}(\mathit{ifds}_{\text{ALAN}}, \textsc{Run}, [\textsc{Sock}(fd, \uparrow i_1, \uparrow p_1, \uparrow i, \uparrow p, *, f, [\,])], [\,], \textsc{false}) \\
&\xrightarrow{\text{sendto}(fd,*,\text{"hello"},\textsc{false})}\xrightarrow{\overline{\text{OK()}}} \textsc{Host}(\mathit{ifds}_{\text{ALAN}}, \textsc{Run}, [\textsc{Sock}(fd, \uparrow i_1, \uparrow p_1, \uparrow i, \uparrow p, *, f, [\,])], [m], \textsc{false}) \\
&\xrightarrow{\overline{m}} \textsc{Host}(\mathit{ifds}_{\text{ALAN}}, \textsc{Run}, [\textsc{Sock}(fd, \uparrow i_1, \uparrow p_1, \uparrow i, \uparrow p, *, f, [\,])], [\,], \textsc{false}) \\
&\xrightarrow{\text{exit ()}} \textsc{Host}(\mathit{ifds}_{\text{ALAN}}, \textsc{Term}, [\,], [\,], \textsc{false})
\end{aligned}
$$

where $fd : \mathsf{fd}$, $f = \textsc{Flags}(\textsc{false}, \textsc{false})$, $i_1 = 192.168.0.14$, $p_1 \in \{1024, .., 4999\}$, $i = 192.168.0.11$, $p = 7654$, and $m = \text{IP}(i_1, i', \text{UDP}(\uparrow p_1, \uparrow p, \text{"sending"}))$.

Here socket creates a new socket, connect sets its remote IP address and port to $\uparrow i$ and $\uparrow p$, sendto constructs an IP datagram m and adds it to the host's outqueue, and the *delivery.1* rule outputs m to the network. The connect call also fills in the socket's local IP address/port – see 'autobinding' (§2.3.3) and 'outroute' (§2.3.7) below. On exit all sockets created by the thread are closed (in our single-thread model, this covers all sockets).

This is just one trace of many – it has no errors or incoming messages, and the output $\overline{m}$ could occur one step earlier or later.

Most of the transitions are invocations and returns of library routines (interactions with the thread program), apart from the console output $\overline{\text{console "sending"}}$ and the output to the network of message m. The first 4 pairs of transitions are potential interactions with the thread, derivable using rules *convert.port.1*, *convert.ip.1*, *socket.1*, *connect.1*, *sendto.1*, and *ret.1*. The next 3 transitions are a call to print_endline_flush and the resulting console output $\overline{\text{console "sending"}}$. The following 3 are a call to sendto and its return, and the output $\overline{m}$ of message m to the network, derivable using *delivery.out.1*. The last is another interaction with the thread, from *exit.1*.

The thread LTSs for e_s includes complementary sequences of transitions, except for the console and network interactions $\overline{\text{console}}$ and $\overline{m}$. It has additional internal transitions for computation steps.

2.3.2 Example Message Reception Now we show the host transitions involved in one run of the receiver e_r from §1.6.

$$
\begin{array}{rl}
 & \textsc{Host}(\mathit{ifds}_{\textsc{kurt}}, \textsc{Run}, [], [], \textsc{false}) \\
\xrightarrow{\text{port_of_int 7654}} \xrightarrow{\overline{\text{OK}\,p'}} & \\
\xrightarrow{\text{ip_of_string "192.168.0.11"}} \xrightarrow{\overline{\text{OK}\,i'}} & \\
\xrightarrow{\text{socket}()} \xrightarrow{\overline{\text{OK}\,fd'}} & \textsc{Host}(.., \textsc{Run}, \quad [\textsc{Sock}(fd', *, *, *, *, *, f, [])], \quad ..) \\
\xrightarrow{\text{bind}(fd', \uparrow i', \uparrow p')} \xrightarrow{\overline{\text{OK}()}} & \textsc{Host}(.., \textsc{Run}, \quad [\textsc{Sock}(fd', \uparrow i', \uparrow p', *, *, *, f, [])], \quad ..) \\
\xrightarrow{\text{print_endline_flush "ready"}} & \textsc{Host}(.., \textsc{Print2 "ready"}, [\quad .. \quad], \quad ..) \\
\xrightarrow{\overline{\text{console "ready"}}} & \textsc{Host}(.., \textsc{Ret (OK())}, \quad [\quad .. \quad], \quad ..) \\
\xrightarrow{\overline{\text{OK}()}} & \textsc{Host}(.., \textsc{Run}, \quad [\quad .. \quad], \quad ..) \\
\xrightarrow{\text{recvfrom}(fd', \textsc{false})} & \textsc{Host}(.., \textsc{Recvfrom2}\ fd', [\quad .. \quad], \quad ..) \\
\xrightarrow{m} & \textsc{Host}(.., \textsc{Recvfrom2}\ fd', [\textsc{Sock}(fd', \uparrow i', \uparrow p', *, *, *, f, [m])], ..) \\
\xrightarrow{\overline{\text{OK}(i_1, \uparrow p_1, \text{"hello"})}} & \textsc{Host}(.., \textsc{Run}, \quad [\textsc{Sock}(fd', \uparrow i', \uparrow p', *, *, *, f, [])], \quad ..) \\
\xrightarrow{\text{print_endline_flush "hello"}} & \textsc{Host}(.., \textsc{Print2 "hello"}, [\quad .. \quad], \quad ..) \\
\xrightarrow{\overline{\text{console "hello"}}} \xrightarrow{\overline{\text{OK}()}} & \textsc{Host}(.., \textsc{Run}, \quad [\quad .. \quad], \quad ..) \\
\xrightarrow{\text{exit}()} & \textsc{Host}(.., \textsc{Term}, \quad [], \quad ..)
\end{array}
$$

where $fd' : \mathsf{fd}$, $f = \textsc{Flags}(\textsc{false}, \textsc{false})$, $i' = 192.168.0.11$, $p' = 7654$, and m is as before.

Again, these transitions are invocations and returns of library routines, apart from the input with label m, receiving a datagram from the network, and the console outputs $\overline{\text{console "ready"}}$ and $\overline{\text{console "hello"}}$.

2.3.3 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, unused(s), where

$$\text{unused}(s) = \{\, p \mid p \in \{1, .., 65535\} \wedge \neg \exists \textsc{Sock}(_, _, \uparrow p, _, _, _, _, _) \in s \,\}$$

One can bind the local port of a socket either to an explicit non-privileged value, *eg.* the $p' = 7654$ 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↑ argument, as in the *bind.2* rule:

bind.2 $(\uparrow i, *)$ **succeed, autobinding**

$$
\begin{array}{l}
\quad F(\mathit{ifds}, \textsc{Run}, \textsc{Sock}(fd, *, *, *, *, es, f, mq)) \\
\xrightarrow{\text{bind}(fd, \uparrow i, *)} F(\mathit{ifds}, \textsc{Ret}\ (\text{OK}()), \textsc{Sock}(fd, \uparrow i, \uparrow p'_1, *, *, es, f, mq))
\end{array}
$$

$p'_1 \in \text{unused}(F) \cap \text{ephemeral}$ and $i \in \mathit{ifds}$

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 $\textsc{Host}(__1, __2, S(__3), oq, oqf)$, where S is a socket list context, of the form $s_1 @ [_] @ s_2$. This and the other context forms H, Q and J were defined in §2.1.2. The unused function is lifted in the obvious way. 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, which use the function below.

$$\text{autobind}(ps_1, F) = (\text{if } ps_1 = \uparrow p_1 \text{ then } \{p_1\} \text{ else unused}(F) \cap \text{ephemeral})$$

2.3.4 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.

sendto.1 **succeed**

$$\text{HOST}(ifds, \text{RUN}, S(\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, *, f, mq)), oq, oqf)$$
$$\xrightarrow{\text{sendto}(fd, ips, data, nb)}$$
$$\text{HOST}(ifds, \text{RET (OK())}, S(\text{SOCK}(fd, is_1, \uparrow p'_1, is_2, ps_2, *, f, mq)), oq', oqf')$$

$p'_1 \in \text{autobind}(ps_1, S)$
and $(oq', oqf', \text{TRUE}) \in \text{dosend}(ifds, (ips, data), (is_1, \uparrow p'_1, is_2, ps_2), oq, oqf)$
and $\text{size}(data) \leq \text{UDPpayloadMax}$ and $(ips \neq * \text{ or } is_2 \neq *)$.

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 UDPpayloadMax; 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*

$$\text{HOST}(ifds, t, s, oq, oqf)$$
$$\xrightarrow{\overline{\text{IP}(i_3, i_4, body)}} \text{HOST}(ifds, t, s, oq', oqf')$$

$((\text{IP}(i_3, i_4, body)), oq', oqf') \in \text{dequeue}(oq, oqf)$
and $i_4 \notin \text{LOOPBACK} \cup \text{MARTIAN}$
and $i_3 \notin \text{MARTIAN}$

In *delivery.out.1*: the dequeue function picks a datagram off the outqueue (nondeterministically resetting the *oqf* flag), and checks the datagram has non-martian source and destination addresses [Bak95, §5.3.7] (see §2.3.23). It outputs the datagram to the network.

2.3.5 Return From a Fast Call After the invocation of a fast call (§2.2.1), *eg.* 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**

$$\text{HOST}(ifds, \text{RET } v, s, oq, oqf)$$
$$\xrightarrow{\overline{v}} \text{HOST}(ifds, \text{RUN}, s, oq, oqf)$$

2.3.6 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**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, is_1, ps_1, is_2, ps_2, *, f, \text{NIL})) \xrightarrow{\text{recvfrom}(fd, \text{FALSE})} F(ifds, \text{RECVFROM2}\ fd, \text{SOCK}(fd, is_1, \uparrow p'_1, is_2, ps_2, *, f, \text{NIL}))$$

$p'_1 \in \text{autobind}(ps_1, \text{socks}(F))$

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, *eg.* $\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data))$, arrives at a host, the 4-tuple (i_3, ps_3, i_4, ps_4) 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 $\text{SOCK}(.., is_1, ps_1, is_2, ps_2, ..)$, giving a score from 0 to 4 of how many elements match, treating a $*$ in the socket elements as a wildcard. The lookup function defined in §2.3.20 takes a list s of sockets and a datagram 4-tuple (i_3, ps_3, i_4, ps_4), 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 *delivery.in.udp.1* rule below.

delivery.in.udp.1 **get UDP from network and deliver to a matching socket**

$$\text{HOST}(ifds, t, s, oq, oqf) \xrightarrow{\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data))} \text{HOST}(ifds, t, S(\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq :: (\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data)), ifid))), oq, oqf)$$

$\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in \text{lookup}\ s\ (i_3, ps_3, i_4, ps_4)$
and $S(\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) = s$
and $(ifid, iset, _, _) \in ifds$ and $i_4 \in iset$
and $i_4 \notin \text{LOOPBACK}$ and $i_3 \notin \text{MARTIAN} \cup \text{LOOPBACK}$

After this, a blocked recvfrom will be able to complete, using the *recvfrom.6* rule.

recvfrom.6 **slow succeed**

$$F(ifds, \text{RECVFROM2}\ fd, \text{SOCK}(fd, is_1, \uparrow p_1, is_2, ps_2, *, f, (\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data)), ifid) :: mq)) \xrightarrow{\overline{\text{OK}(i_3, ps_3, data)}} F(ifds, \text{RUN}, \text{SOCK}(fd, is_1, \uparrow p_1, is_2, ps_2, *, f, mq))$$

2.3.7 Wildcard IP Addresses and outroute When a datagram $\text{IP}(i_1, i_2, body)$ is sent from a host, its source IP address i_1 must be filled in. If the socket's local IP address field is a wildcard, then one of the host's IP addresses must be chosen – in fact, the primary IP address of the interface that the host's local routing table specifies should be used for sending to i_2. This occurs in sendto and also in connect.

Our model does not contain routing information, so we take a nondeterministic approximation, distinguishing only between loopback and non-loopback addresses. We define

$\text{outroute} : \text{ifd set} * \text{ip} \to \text{ip set}$
$\text{outroute}\ (\{\, \text{IF}(ifid_j, iset_j, iprimary_j, netmask_j) \mid j \in J \,\}, i) =$
  if $i \in \text{LOOPBACK}$ then {localhost}
  else $\{\, iprimary_j \mid j \in J$ and $iset_j \neq \text{LOOPBACK} \,\}$

used, for example, in *connect.1*:

connect.1 ($is_1 = *$) **succeed, autobinding if necessary**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, *, ps_1, *, *, es, f, mq)) \xrightarrow{\text{connect}(fd,i,ps)} F(ifds, \text{RET (OK())}, \text{SOCK}(fd, \uparrow i_1, \uparrow p'_1, \uparrow i, ps, es, f, mq))$$

$i_1 \in \text{outroute}(ifds, i)$ and $p'_1 \in \text{autobind}(ps_1, F)$

Note that outroute($ifds, i$) cannot be empty, as we insist (in the host-ok invariant) that machines have both a loopback interface and at least one other interface. We do so to simplify the treatment of dosend and sendto, ensuring dosend always returns a nonempty set; it is justified by the fact that we do not wish to reason about disconnected machines here (in fact, disconnected machines can generate ENETUNREACH errors in connect and sendto).

2.3.8 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 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 enqueue described in §2.3.18 below.

delivery.in.udp.2 **get UDP from network but generate ICMP, as no matching socket**

$$\text{HOST}(ifds, t, s, oq, oqf) \xrightarrow{\text{IP}(i_3,i_4,\text{UDP}(ps_3,ps_4,data))} \text{HOST}(ifds, t, s, oq', oqf')$$

$i_4 \in ifds$ and lookup s (i_3, ps_3, i_4, ps_4) $= \emptyset$
and $(oq', oqf', ok) \in \{(oq, oqf, \text{TRUE})\} \cup$
enqueue(IP(i_4, i_3, ICMP_PORT_UNREACH(i_3, ps_3, i_4, ps_4)), oq, oqf)
and $i_4 \notin$ LOOPBACK and $i_3 \notin$ MARTIAN $\cup$ LOOPBACK

The ICMP is not always generated – the Linux kernel has per-type-of-ICMP rate-limiting to control denial-of-service attacks. We take a nondeterministic approximation.

2.3.9 Asynchronous Errors When an 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 es' of the socket is set, in this case to ↑ECONNREFUSED.

delivery.in.icmp.1 **get ICMP from the network, setting error in a matching socket**

$$\text{HOST}(ifds, t, s, oq, oqf) \xrightarrow{\text{IP}(i'_4,i'_3,\text{ICMP_X_UNREACH}(i_3,ps_3,i_4,ps_4))} \text{HOST}(ifds, t, S(\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es', f, mq)), oq, oqf)$$

$S(\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) = s$
and $\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in$ lookup $s(i_3, ps_3, i_4, ps_4)$
and $m = \text{IP}(i'_4, i'_3, \text{ICMP_X_UNREACH}(i_3, ps_3, i_4, ps_4))$
and $i'_3 \in ifds$ and $\neg(\text{loopback}(m) \vee \text{martian}(m))$
and $es' =$ if $(is_2 \neq *)$ or $\neg$(bsdcompat f) then ↑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 un-

matched ICMPs do not themselves generate new ICMPs – there is no analogue of *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.

sendto.5 **fail, as socket in an error state**

$$F(ifds, \textsc{Run}, \textsc{Sock}(fd, is_1, \uparrow p_1, is_2, ps_2, \uparrow e, f, mq)) \xrightarrow{\text{sendto}(fd, ips, data, nb)} F(ifds, \textsc{Ret}\ (\textsc{Fail}\ e), \textsc{Sock}(fd, is_1, \uparrow p_1, is_2, ps_2, *, f, mq))$$

The error flag can also be explicitly inspected and cleared by geterr .

geterr.1 **succeed**

$$F(ifds, \textsc{Run}, \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) \xrightarrow{\text{geterr } fd} F(ifds, \textsc{Ret}\ (\text{OK}\, es), \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, *, f, mq))$$

2.3.10 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:

EACCES	for a port $p \in$ privileged $= \{1, .., 1023\}$,
EADDRINUSE	for a port already in use (modulo the reuseaddr flags),
EADDRNOTAVAIL	for an IP address that is not one of the host's, and
EINVAL	if the socket already has a non-$*$ local port;

for sendto we have

ENOTCONN	if the destination is $*$ and the socket is unconnected, and
EMSGSIZE	if the *data* is bigger than UDPpayloadMax = 65507 (see §2.3.16);

and 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:

intr.1 **slow intr, as system call interrupted**

$$\textsc{Host}(ifds, op\, v, s, oq, oqf) \xrightarrow{\overline{\textsc{Fail EINTR}}} \textsc{Host}(ifds, \textsc{Run}, s, oq, oqf)$$

$op \in \{\textsc{Sendto2}, \textsc{Recvfrom2}, \textsc{Select2}, \textsc{Print2}\}$

with the expectation that most interesting reasoning will depend on an assumption that this does not occur too often.

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 ENOBUFS if the OS has run out of space or buffers. Again, these are modelled by purely nondeterministic rules, for example

badmem.2 *OP* **bad fail, as no space**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) \xrightarrow{OP(fd)} F(ifds, \text{RET}\ (\text{FAIL}\ e), v)$$

$e \in \{\text{ENOMEM}, \text{ENOBUFS}\}$ and $\vdash v$ socket-ok($ifds$) and sockfd(v) = fd

where *OP* ranges over a class of invocations of calls with a single fd argument (defined in §3). Note that the rule allows the socket state to change arbitrarily, as we do not wish to specify exactly when during the system call such failures may occur, or expect the OS to return the socket to its initial state, so an application program should not depend on the resulting socket state. Here, we expect interesting reasoning to depend on an assumption that these errors do not occur. They are included in the model as an application may have to catch them in order to do some clean-up.

We must also deal with the possibility that all the ephemeral ports are exhausted when an autobind is perfomed, in one of bind, connect, disconnect, sendto. On our test machines this does not occur, as other resources are exhausted first – we include rules for it to ensure the semantics does not get stuck. The returned error is chosen from ephemeralErrors = {EAGAIN, EADDRINUSE}.

2.3.11 Errors and Nondeterminism In cases where multiple error conditions hold, we have chosen to give a loose specification, nondeterministically allowing any of them. In any particular implementation some will take priority over others, depending on how the OS code is sequenced, but this is very likely to vary – applications should not depend on a particular priority order. Roughly, the various 'success' rules are disjoint from each other and from all the 'fail' rules, but the 'fail' rules can overlap with each other. This is made precise by Theorem 3 on page 33.

In some error cases we allow a socket state to be changed nondeterministically – again, as the actual result depends on details of the OS coding that we do not wish to fix, and that applications should not depend on.

2.3.12 Loopback A datagram sent to a loopback address $i \in$ LOOPBACK = 127/8 will be echoed back – without reaching the network. Typically, one uses the loopback address localhost = 127.0.0.1, which our ifd-set-ok condition fixes as the primary address of the unique loopback interface on each machine.

To model loopback, we use a number of additional delivery rules (*delivery.loopback.**), which are essentially the compositions of *delivery.out.** and *delivery.in.** rules. For example, the rule

delivery.loopback.udp.1 **get loopback UDP from oq and deliver to a matching socket**

$$\text{HOST}(ifds, t, s, oq, oqf) \xrightarrow{\tau} \text{HOST}(ifds, t, S(\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq :: (\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data)), ifid))), oq', oqf')$$

$(\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data)), oq', oqf') \in \text{dequeue}(oq, oqf)$
and $i_4 \in$ LOOPBACK
and $s = S(\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq))$
and $\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in$ lookup $s(i_3, ps_3, i_4, ps_4)$
and $\text{IF}(ifid, \text{LOOPBACK}, \text{localhost}, 255/8) \in ifds$

removes a loopback UDP from a host's outqueue and delivers in to a matching socket, in a single step. It is an analogue of *delivery.out.1* composed with *delivery.in.udp.1* (see §2.3.4, 2.3.6).

Together with $i \notin$ LOOPBACK conditions in various rules, this ensures that the IP packets that would otherwise be generated do not touch the network. This means they cannot be reordered, duplicated, or lost (the last is an idealisation, since in actual systems the OS could drop them for lack of space).

2.3.13 Console Interaction As a minimal form of `stdio` I/O, we add a flushing print, with host labelled transitions $\overline{\text{console}}\ v$ for strings v generated by rule

console.print.2 **print the string on console**

$$H(\textsc{Print2}\ v) \xrightarrow{\overline{\text{console } v}} H(\textsc{Ret}\ (\text{OK}()))$$

Adding input would be straightforward, but would require dealing with the fact that user input events (key presses or mouse events) can occur at any time. As in [Sew97], the host state would have to be extended with a buffer of these events.

2.3.14 Thread Termination A thread can terminate by calling exit : () → void, from which the host will not return. On termination, all sockets are closed.

exit.1 **exit**

$$\textsc{Host}(\mathit{ifds}, \textsc{Run}, s, \mathit{oq}, \mathit{oqf}) \xrightarrow{\text{exit }()} \textsc{Host}(\mathit{ifds}, \textsc{Term}, [], \mathit{oq}, \mathit{oqf})$$

This is correct only for an idealised machine which has no sockets except those opened by the thread. Note that the fact that sockets are closed on termination may be observed by other machines, as they can then receive ICMP_PORT_UNREACH messages from the associated ports. We do not treat Unix return codes (the OCaml exit allows a Unix return code to be specified, but we do not wish to model the details).

2.3.15 Select Timeout As said in §2.1.5, calling select($\mathit{readseq}, \mathit{writeseq}, \uparrow \mathit{tm}$) will block until one of the sockets in *readseq* has a message ready for reading, one of those in *writeseq* is ready for writing, time *tm* elapses, or an error occurs. With an untimed interleaving semantics, the timeout cannot be modelled very faithfully – the most we can do is allow the timer to decrement, nondeterministically:

select.5 **slow timeout decrement**

$$Q(\textsc{Select2}\ (\mathit{readseq}, \mathit{writeseq}, \uparrow \mathit{tm}), s, \mathit{oqf}) \xrightarrow{\tau} Q(\textsc{Select2}\ (\mathit{readseq}, \mathit{writeseq}, \uparrow (\mathit{tm} - 1)), s, \mathit{oqf})$$

$\mathit{tm} > 0$
$\mathit{readseq}' = [\mathit{fd} \mid \mathit{fd} \in \mathit{readseq}$ and $\exists \textsc{Sock}(\mathit{fd}, \mathit{is}_1, \mathit{ps}_1, \mathit{is}_2, \mathit{ps}_2, \mathit{es}, f, \mathit{mq}) \in s\ .\ (\mathit{mq} \neq \textsc{nil} \vee \mathit{es} \neq *)]$
$\mathit{writeseq}' = [\mathit{fd} \mid \mathit{fd} \in \mathit{writeseq}$ and $\mathit{oqf} = \textsc{false}$ and $\exists \textsc{Sock}(\mathit{fd}, \mathit{is}_1, \mathit{ps}_1, \mathit{is}_2, \mathit{ps}_2, \mathit{es}, f, \mathit{mq}) \in s]$
$\mathit{readseq}' = \textsc{nil}$ and $\mathit{writeseq}' = \textsc{nil}$

2.3.16 Message Size The maximum size of *data* in a UDP datagram is UDPpayloadMax = 65507 octets, as fixed by the IP and UDP protocols. The msg-ok, network, socket-ok and host-ok judgements require that in all UDP($\mathit{ps}_1, \mathit{ps}_2, \mathit{data}$) on the network, in socket queues, or in the outqueue, we have size(data) ≤ UDPpayloadMax. Calls to sendto with larger *data* return an error (EMSGSIZE if no other error condition applies). In general, it is not guaranteed that such large datagrams will be handled by the protocol endpoint code [Ste98, p159], but the Linux implementation appears to.

A separate issue is that of *fragmentation*: typically, a physical link will only handle packets up to a certain size (*eg.*, 1500 octets for Ethernet). The implementation of IP involves the sender (and possibly also routers) fragmenting datagrams into packets that are small enough to be transmitted, and the receiver reassembling them. This increases the potential for loss of a large datagram, as

all of its fragments must arrive successfully (and within some time window) for the datagram to be reassembled. Moreover, routers may simply drop large packets. At our level of abstraction, however, both of these are covered by the *drop.1* loss rule.

2.3.17 Loss The *drop.1* loss rule $0 \xrightarrow{n\cdot\mathrm{IP}\ v} 0$ covers many different circumstances:

- Packet loss entirely in the network, *eg.* due to routers choosing to drop, noise and collisions in the physical medium.
- Packet loss entirely in the receiver, *eg.* due to:
  - the device being full
  - the network module backlog queue being full
  - the socket message queue being full (discussed in §2.3.22 under 'receive buffer size')
  - checksum failures at various points.
- Defragmentation not succeeding, either because some of the fragments were lost, or because the receiver did not receive them close enough together in time.

2.3.18 The Outqueue A call to sendto can block or fail (according to whether the *nb* flag argument is set) if there is no space to enqueue the message. In the Linux implementation, each socket has a variable-length virtual buffer (size settable with SO_SNDBUF), and messages on the outqueue are charged to the buffer of the socket that sent them; a message cannot be sent by a socket unless there is space in this socket's virtual buffer.

We choose not to model this exactly, instead idealising as follows:

- The host's outqueue *oq* has an associated 'outqueue full' flag *oqf*.
- *oqf* is initially false, indicating that there is space in the outqueue.
- When a message is placed on the outqueue, *oqf* may nondeterministically be set to TRUE.
- When a message is removed from the outqueue, *oqf* may nondeterministically be set to FALSE; if it was the only message on the outqueue, *oqf* is deterministically set to FALSE.
- When *oqf* is set, calls to sendto block or fail.
- When *oqf* is set, any ICMPs that are generated are discarded.

Thus we maintain the host-ok invariant that $oq = [] \implies oqf = \mathrm{FALSE}$. The *sendto.** and *delivery.** rules that express this use two auxiliary functions

$$\begin{aligned} &\text{enqueue}: \mathsf{msg} * \mathsf{msg\,list} * \mathsf{bool} \to (\mathsf{msg\,list} * \mathsf{bool} * \mathsf{bool})\ \mathsf{set} \\ &\text{dequeue}: \mathsf{msg\,list} * \mathsf{bool} \to (\mathsf{msg} * \mathsf{msg\,list} * \mathsf{bool})\ \mathsf{set} \end{aligned}$$

The enqueue function takes a message to enqueue and the current *oq* and *oqf*, and returns a set of (oq', oqf', ok) triples where oq' and oqf' are the resulting outqueue, and *ok* is TRUE if the message was enqueued and FALSE if not. The dequeue function takes the current outqueue and returns a set of (m, oq', oqf') triples where m is the first element and oq' is the resulting outqueue, with flag oqf'. This set is empty if the current outqueue is empty. The functions are defined by

$$\begin{aligned} \text{enqueue}\ (m, oq, \mathrm{TRUE}) &= \{(oq, \mathrm{TRUE}, \mathrm{FALSE})\} \\ \text{enqueue}\ (m, oq, \mathrm{FALSE}) &= \{\, (m :: oq, oqf', \mathrm{TRUE}) \mid oqf' \in \{\mathrm{FALSE}, \mathrm{TRUE}\}\,\} \\ \text{dequeue}\ (\mathrm{NIL}, oqf) &= \{\} \\ \text{dequeue}\ (oq :: m, oqf) &= \{\, (m, oq, oqf') \mid oqf' \in \{(oqf \wedge oq \neq \mathrm{NIL}), \mathrm{FALSE}\}\,\} \end{aligned}$$

We further idealise by not modelling any additional network-card buffering.

2.3.19 Dosend We use an auxiliary function dosend in the sendto rules, to construct an IP datagram and enqueue it on the outqueue. It has type

$$\text{dosend} : \mathsf{ifd\,set} * ((\mathsf{ip} * \mathsf{port})\uparrow * \mathsf{string}) * (\mathsf{ip}\uparrow * \mathsf{port}\uparrow * \mathsf{ip}\uparrow * \mathsf{port}\uparrow) * \mathsf{msg\,list} * \mathsf{bool} \quad (1)$$

$$\to (\mathsf{msg\,list} * \mathsf{bool} * \mathsf{bool})\ \mathsf{set} \quad (2)$$

mapping $(ifds, (ips, data), (is_1, ps_1, is_2, ps_2), oq, oqf)$ to a set of triples (oq', oqf', ok). It is only called with ($ips \neq *$ or $is_2 \neq *$) and $ps_1 \neq *$, and always returns a nonempty set (the latter depends on the fact that enqueue and outroute always return nonempty sets).

It takes the host's $ifds$, the $(ips, data)$ part of the sendto arguments, the $(is_1, ps'_1, is_2, ps_2)$ part of the relevant socket (ps'_1 is either the original $ps_1 \neq *$, or the result of the necessary autobind), the outqueue oq and the output flag oqf, and returns the set of possible outcomes. Each outcome is a triple of the new oq, the new oqf and a bool. This bool is TRUE if there was room on the outqueue for the msg, and FALSE if not. In the latter case, $oq' = oq$ and $oqf' = oqf = \text{TRUE}$ by definition of enqueue.

$$\begin{array}{l}\text{dosend}(ifds, (*, data), (\uparrow i_1, \uparrow p_1, \uparrow i_2, ps_2), oq, oqf)\\ = \text{enqueue}(\text{IP}(i_1, i_2, \text{UDP}(\uparrow p_1, ps_2, data)), oq, oqf)\end{array}$$

$$\begin{array}{l}\text{dosend}(ifds, (\uparrow(i, p), data), (*, \uparrow p_1, *, *), oq, oqf)\\ = \bigcup_{i' \in \text{outroute}(ifds, i)} \text{enqueue}(\text{IP}(i', i, \text{UDP}(\uparrow p_1, \uparrow p, data)), oq, oqf)\end{array}$$

$$\begin{array}{l}\text{dosend}(ifds, (\uparrow(i, p), data), (\uparrow i_1, \uparrow p_1, is_2, ps_2), oq, oqf)\\ = \text{enqueue}(\text{IP}(i_1, i, \text{UDP}(\uparrow p_1, \uparrow p, data)), oq, oqf)\end{array}$$

The first case, in which a UDP datagram with a zero destination port is generated if $ips = *$ and $ps_2 = *$, may be surprising but does occur. One might expect the i_1 in the UDP datagram of the last case to be taken to be the primary IP address of the interface used for sending to i, not the local address i_1 of the socket, *ie.* with

$$\bigcup_{i' \in \text{outroute}(ifds, i)} \text{enqueue}(\text{IP}(i', i, \text{UDP}(\uparrow p_1, \uparrow p, data)), oq, oqf),$$

but it is not. Note that sendto$(fd, \uparrow(i, p), data, nb)$ on a connected socket (*ie.* with is_2, ps_2 not $*, *$) works, whereas according to [Ste98, p225] Posix specifies it should give EISCONN.

2.3.20 Matching and Lookup At various points a 4-tuple (i_3, ps_3, i_4, ps_4) is matched against each of the host's sockets, to determine which socket (if any) match it most closely (*eg.* as in *delivery.in.udp.1*, §2.3.6). We define a matching function, returning a score for each match, as follows.

$$\begin{array}{lll}
\text{match } (*, *, *, *) & (i_3, ps_3, i_4, ps_4) = 0 \\
\text{match } (*, \uparrow p_1, *, *) & (i_3, ps_3, i_4, ps_4) = \text{if } (\uparrow p_1 = ps_4) \text{ then 1 else 0} \\
\text{match } (\uparrow i_1, \uparrow p_1, *, *) & (i_3, ps_3, i_4, ps_4) = \text{if } (i_1 = i_4) \text{ and } (\uparrow p_1 = ps_4) \text{ then 2 else 0} \\
\text{match } (\uparrow i_1, \uparrow p_1, \uparrow i_2, *) & (i_3, ps_3, i_4, ps_4) = \text{if } (i_2 = i_3) \text{ and } (i_1 = i_4) \\
 & \qquad \text{and } (\uparrow p_1 = ps_4) \text{ then 3 else 0} \\
\text{match } (\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2) & (i_3, ps_3, i_4, ps_4) = \text{if } (\uparrow p_2 = ps_3) \text{ and } (i_2 = i_3) \\
 & \qquad \text{and } (i_1 = i_4) \text{ and } (\uparrow p_1 = ps_4) \text{ then 4 else 0}
\end{array}$$

Using this, we define score and lookup .

$$\begin{array}{l}\text{score } \text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)\ (i_3, ps_3, i_4, ps_4)\\ \qquad = \text{match } (is_1, ps_1, is_2, ps_2)\ (i_3, ps_3, i_4, ps_4)\\ \text{lookup } s\ (i_3, ps_3, i_4, ps_4) = \{\text{SOCK } v \mid \text{SOCK } v \in s \text{ and score } (\text{SOCK } v)\ (i_3, ps_3, i_4, ps_4) > 0\\ \qquad \text{and } \neg\exists \text{SOCK } v' \in s\,.\,\text{score } (\text{SOCK } v')\ (i_3, ps_3, i_4, ps_4) > \text{score } (\text{SOCK } v)\ (i_3, ps_3, i_4, ps_4)\}\end{array}$$

The lookup function returns the set of maximal non-zero scoring sockets, which may be empty. If there are several then one will be chosen nondeterministically (rules that use lookup take an arbitrary element).

In the Linux implementation, sockets are kept in a hash chain, which resolves this nondeterminism. The details are complex, as sockets can be added either at bind time or at 'first use' time; we expect applications will not depend on this behaviour and so choose not to model it.

2.3.21 LIB Design: the Thin Abstraction Layer Given our choice of what UDP functionality to deal with, and our decision to abstract from the intricacies of C parameter passing, the design of LIB is largely straightforward. A few small points remain – LIB also abstracts from some awkwardnesses of the C system calls underlying our bind, disconnect, getsockname and getpeername, to ensure that our socket states are an accurate abstraction of the actual OS datastructures. The details are given in comments to the rules *bind.5* and *getsockname.1*, *getpeername.1*, and *disconnect.*$*$. We omit send and recv since they are simply trivial wrappers around sendto and recvfrom. We arbitrarily ignore sendmsg and recvmsg.

A useful principle in the LIB design is that it should be stateless, so that in extremis an application could use combine LIB and a standard sockets library, if some non-LIB functionality is required.

2.3.22 Socket Options and Other Flags, and the LIB Design We considered all the socket-level socket options defined by Linux. Those we do *not* implement, because they are irrelevant to our fragment of UDP or duplicate behaviour obtainable by other means, are: SO_ATTACH_FILTER, SO_DETACH_FILTER, SO_BINDTODEVICE, SO_BROADCAST, SO_DONTROUTE, SO_NO_CHECK, SO_PASSCRED, SO_PEERCRED, SO_PRIORITY, SO_RCVLOWAT and SO_SNDLOWAT, SO_RCVTIMEO and SO_SNDTIMEO, SO_TYPE, and several options only relevant for TCP, namely SO_DEBUG, SO_KEEPALIVE, SO_LINGER, and SO_OOBINLINE. We implement the only interesting SOL_IP socket option other than perhaps IP_RECVERR, namely IP_PKTINFO, which appears to have similar functionality to the IP_RECVDSTADDR and IP_RECVIF options of [Ste98, pp198,532]. The options set via `fcntl` and `ioctl`, and the flags passed to send and receive, are either mentioned below or have been omitted for similar reasons.

The options we implement are as follows. Their semantics follows Linux except where otherwise stated.

BSD compatibility If an ICMP is received for a socket that is not connected, the BSD semantics (against [Bra89, §4.1.3.3]) is to ignore it. Linux however normally delivers the message, just as it would for a connected socket. This only affects ICMPs defined as hard errors (here ICMP_PORT_UNREACH but not ICMP_HOST_UNREACH) or sockets with the IP_RECVERR option set. The BSD semantics may be obtained by setting SO_BSDCOMPAT; by default this is false except on SunOS and SunOS32. We provide this socket option as an element of the flags component of the Sock structure.

Socket errors Reading the SO_ERROR socket option gets and clears the current socket error if present. Writing is not possible. We provide this with a system call, geterr: fd → error↑err; see §2.3.9.

Send and receive buffer size The send and receive buffer sizes may be read and written by the socket options SO_SNDBUF and SO_RCVBUF. Modelling these options fully in our calculus would complicate the semantics in annoying ways: the receive buffer would limit the per-socket message queue in some way, and the send buffer would involve tagging each message on the outqueue with an owning socket for charging purposes. Instead, we implement a nondeterministically-full outqueue as described in §2.3.18, and ignore the receive queue entirely,

considering it a part of the network and thus modelled by *drop.1*, the network packet-dropping rule. Note that this means that an assumption that *drop.1* does not occur (such as one might make when proving properties of an algorithm under a lossless network hypothesis) is only reasonable if every host does enough recvfroms that no message queue ever fills.

Address reuse Under normal circumstances, the 4-tuples (is_1, ps_1, is_2, ps_2) of distinct sockets may not overlap; that is, every incoming message must match (in the sense of match, §2.3.20) at most one socket. However, this may be overridden if *all* sockets participating in the overlap have the SO_REUSEADDR socket option set. The constraint is enforced at bind-time. We provide this socket option as an element of the flags component of the SOCK structure.

Packet information Sometimes (such as when doing virtual hosting) we want to know the destination address and/or receiving interface of an incoming packet. [Ste98, pp198,532] suggests the use of SOL_IP options IP_RECVDSTADDR and IP_RECVIF. In Linux, these are not available, but setting the IP_PKTINFO option causes a control message to be placed in the message header of each message received on the socket, containing the remote *source* address and the receiving interface. This may be obtained using `recvmsg`. In our calculus, messages on the socket message queue are tagged with the receiving interface; however it was considered sufficiently awkward in Linux to obtain the IP_PKTINFO information that it would be unrealistic to return this information from every recvfrom call. Instead, the semantics contains enough information to permit an extended recvfrom (such as a recvmsg call) to be defined, providing this information to the thread. We do not define this extension here.

Non-blocking There are several ways in which potentially-blocking operations (send, receive) can be made non-blocking, returning EAGAIN (in Linux aliased with EWOULDBLOCK) if they would otherwise block. Using an `ioctl` or `fcntl`, the socket file descriptor may be set to O_NONBLOCK so that all subsequent operations will not block; alternatively, individual sends and receives may set the MSG_DONTWAIT flag. We provide this functionality by adding a flag to sendto and recvfrom, requesting non-blocking operation per call. (The non-blocking flag does not affect delivery, hence it is not necessary to store this flag in the socket.)

Interface information There are several ways to obtain information about the host's interfaces. Among them are the `ioctl` SIOCGIFCONF, and routing sockets. Abstracting away from the details of how exactly we obtain this information, we simply add a system call $\mathsf{getifaddrs} : () \to (\mathsf{ifid} * \mathsf{ip} * \mathsf{ip\,list} * \mathsf{netmask})\,\mathsf{list}$.

2.3.23 IP Addressing: Loopback and Martians We use dotted-decimal notation for IP addresses, writing *eg.* 192.3.57.1 for $192 \times 2^{24} + 3 \times 2^{16} + 57 \times 2^8 + 1$. Trailing zero components will sometimes be omitted, allowing *eg.* 127 to denote 127.0.0.0. For $x \in 0..32$ and an address i which is zero in its least significant $(32 - x)$ bits we write i/x for the set of addresses which agree with i on their most significant x bits.

Several classes of IP addresses have special meaning. Here we pick out the *loopback* addresses, $\text{LOOPBACK} = 127/8 : \mathsf{ip\,set}$, for datagrams to be echoed internally, with conventional element $\text{localhost} = 127.0.0.1 \in \text{LOOPBACK}$, and the pathological *martian* addresses [Bak95, §5.3.7].

We take a crude idealisation of martian addresses, defining the sets of bad addresses below (in fact, sources and destinations are treated differently, and broadcast/multicast are involved).

$$\begin{aligned}
\text{MULTICAST} &= 224/4 \\
\text{BADCLASS} &= 240/4 \\
\text{ZERONET} &= 0/8 \\
\text{MARTIAN} &= \text{MULTICAST} \cup \text{BADCLASS} \cup \text{ZERONET}
\end{aligned}$$

These are lifted to a definition of martian IP datagram as below.

$$\begin{array}{l}\text{martian}(\text{IP}(i_1, i_2, \text{UDP}\ v)) = \{i_1, i_2\} \cap \text{MARTIAN} \neq \emptyset \\ \text{martian}(\text{IP}(i_1, i_2, \text{ICMP_X_UNREACH}(i_3, ps_3, i_4, ps_4))) = \{i_1, i_2, i_3, i_4\} \cap \text{MARTIAN} \neq \emptyset\end{array}$$

Similarly, we define a predicate that picks out IP datagrams containing (in any position) a loopback address.

$$\begin{array}{l}\text{loopback}(\text{IP}(i_1, i_2, \text{UDP}\ v)) = \{i_1, i_2\} \cap \text{LOOPBACK} \neq \emptyset \\ \text{loopback}(\text{IP}(i_1, i_2, \text{ICMP_X_UNREACH}(i_3, ps_3, i_4, ps_4))) = \{i_1, i_2, i_3, i_4\} \cap \text{LOOPBACK} \neq \emptyset\end{array}$$

We assume that IP addresses in datagrams (on the network and in host queues) are non-zero, expressing this by using ip rather than ip↑ in the type of the message constructor IP. This is not entirely true – at start-up, a host may need to send messages with a zero *source* address; zero destinations are still forbidden [Ste98, p891].

2.3.24 Interfaces (2) For $ifds = \{\, \text{IF}(ifid_j, iset_j, iprimary_j, netmask_j) \mid j \in J \,\}$, we say $\vdash ifds$ ifd-set-ok if:

1. $\vdash ifds$: ifd set.
2. all ifids are distinct, *ie.* $\forall j, j' \in J \,.\, j \neq j' \implies ifid_j \neq ifid_{j'}$
3. each primary IP is in the associated IP set, *ie.* $\forall j \in J \,.\, iprimary_j \in iset_j$
4. all IPs in each IP set agree with each other on the set bits of the netmask, *ie.* $\forall j \in J \,.\, \forall i, i' \in iset_j \,.\, (i \,\&\, netmask_j) = (i' \,\&\, netmask_j)$
5. all IPs appearing in *ifds* are distinct, *ie.* $\forall j, j' \in J \,.\, j \neq j' \implies iset_j \cap iset_{j'} = \emptyset$
6. there is exactly one loopback interface, *ie.* $\exists_1 j \in J \,.\, iset_j = \text{LOOPBACK} \wedge iprimary_j = \text{localhost} \wedge netmask_j = 255.0.0.0$
7. there is at least one non-loopback interface, *ie.* $\exists j \in J \,.\, iset_j \cap \text{LOOPBACK} = \emptyset$
8. all interfaces have non-martian addresses, *ie.* $\forall j \in J \,.\, iset_j \cap \text{MARTIAN} = \emptyset$

Two hosts have disjoint IP addresses if they satisfy the predicate

$$\text{disjoint_addr}(\text{HOST}\ v, \text{HOST}\ v') = \forall j, j' \,.\, (iset_j \cap iset_{j'}) \setminus \text{LOOPBACK} = \emptyset$$

2.4 Sanity Properties

This section gives 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.

Define host thread state classes – subsets of $\{\, v \mid \vdash v : \text{hostThreadState} \,\}$ – as follows (overloading the identifiers):

$\text{RUN} = \{\text{RUN}\}$
$\text{TERM} = \{\text{TERM}\}$
$\text{RET}_{TL}, \text{SENDTO2}, \text{RECVFROM2}, \text{SELECT2}, \text{PRINT2}$ the values of the corresponding forms
$\text{RET} = \{\, \text{RET}_{TL}\, v \mid \vdash v : TL \,\}$
$\text{RETOK} = \{\, \text{RET}_{TL\,\text{err}}(\text{OK}v) \mid \vdash v : TL \,\}$
$\text{RETFAIL} = \{\, \text{RET}_{TL\,\text{err}}(\text{FAIL}\ v) \mid \vdash v : \text{error} \,\}$
$\text{OP2} = \text{SENDTO2} \cup \text{RECVFROM2} \cup \text{SELECT2} \cup \text{PRINT2}$

These classes are lifted to hosts – subsets of $\{\, v \mid \vdash v : \mathsf{host} \,\}$ – in the obvious way. Let $h = \textsc{Host}(\mathit{ifds}, t, s, \mathit{oq}, \mathit{oqf})$ and $h' = \textsc{Host}(\mathit{ifds}', t', s', \mathit{oq}', \mathit{oqf}')$ range over hosts, and l range over host labels Lhost.

Theorem 1 (Axiom Classification). *The host transition axioms may be partitioned as follows:*

Transition type	From state	Under label	To state	By rules
$h \xrightarrow{f\ v} h'$	Run	$f\ v$	RetOK	succeed
	Run	$f\ v$	Op2	enter2
	Run	$f\ v$	Term	$\{\mathit{exit.1}\}$
	Run	$f\ v$	RetFail	fail
	Run	$f\ v$	RetFail	badfail
$h \xrightarrow{\overline{r}} h'$	Ret	$\overline{r}$	Run	$\{\mathit{ret.1}\}$
	Op2	$\overline{\mathrm{OK}\,v}$	Run	slowsucceed
	Op2	$\overline{\textsc{Fail}\ v}$	Run	slowfail
	Op2	$\overline{\textsc{Fail EINTR}}$	Run	$\{\mathit{intr.1}\}$
	Op2	$\overline{\textsc{Fail}\ v}$	Run	slowbadfail
$h \xrightarrow{\overline{\text{console } v}} h'$	Print2	$\overline{\text{console } v}$	RetOK	$\{\mathit{console.print.2}\}$
$h \xrightarrow{\tau} h'$	Select2	τ	Select2	$\{\mathit{select.5}\}$
	t	τ	t	internaldelivery
$h \xrightarrow{\mathrm{IP}\,v} h'$	t	$\mathrm{IP}\,v$	t	accept
$h \xrightarrow{\overline{\mathrm{IP}\,v}} h'$	t	$\overline{\mathrm{IP}\,v}$	t	emit

Here succeed, fail, badfail, slowsucceed, slowfail, slowbadfail *are the sets of rules with those words first in the rule description. The set* enter2 *is* $\{\mathit{sendto.2}, \mathit{recvfrom.2}, \mathit{select.1}, \mathit{console.print.1}\}$. *The set* internaldelivery *is* $\{\mathit{delivery.loopback.*}, \mathit{delivery.out.martian}\}$. *The set* accept *is* $\{\mathit{delivery.in.*}\}$. *The set* emit *is* $\{\mathit{delivery.out.1}\}$. *In the last three lines the* hostThreadState t *is unchanged.*

Proof. By inspection of the axioms.

Theorem 2 (Preservation: typing and host-ok). *Suppose* $\vdash h$ host-ok.

1. *If* $h \xrightarrow{f\ v} h'$, $f : TL \to TL' \in \mathrm{LIB}$ *and* $\vdash v : TL$ *then* $\mathrm{htsType}(h) = \textsc{Run}$, $\mathrm{htsType}(h') = TL'$ *and* $\vdash h'$ host-ok.
2. *If* $h \xrightarrow{\overline{r}} h'$ *then* $t \in \textsc{Ret} \cup \textsc{Op2}$, $\vdash r : \mathrm{htsType}(h)$, $t' = \textsc{Run}$, *and* $\vdash h'$ host-ok.
3. *If* $h \xrightarrow{\tau} h'$ *then* $\mathrm{htsType}(h') = \mathrm{htsType}(h)$ *and* $\vdash h'$ host-ok.
4. *If* $h \xrightarrow{\overline{\mathrm{IP}\,v}} h'$ *then* $\mathrm{htsType}(h') = \mathrm{htsType}(h)$ *and* $\vdash h'$ host-ok.
5. *If* $h \xrightarrow{\mathrm{IP}\,v} h'$ *then* $\mathrm{htsType}(h') = \mathrm{htsType}(h)$ *and* $\vdash h'$ host-ok.
6. *If* $h \xrightarrow{\overline{\text{console } v}} h'$ *then* $\vdash v : \mathsf{string}$, $\mathrm{htsType}(h') = \mathrm{htsType}(h)$ *and* $\vdash h'$ host-ok.

Proof. By inspection of the axioms.

Corollary 1. *If* h *performs a library call transition* $f\ v$, *then the first subsequent return transition (if any) is well-typed for* f*:*

$$(f : TL \to TL' \in \mathrm{LIB} \wedge h \xrightarrow{f\ v} \cdot \to^*_{noRet} \cdot \xrightarrow{\overline{r}}) \implies \vdash r : TL'$$

where $(\to_{noRet}) = \{\, (h, h') \mid (h, l, h') \in (\dot{\to}) \wedge \forall r \,.\, l \neq \overline{r} \,\}$

Now let R range over axiom names and define an extended transition relation labelled with these, thus: $h \rightarrow^{l}_{R} h'$. Write $h \rightarrow_{R} h'$ for $\bigcup_l \rightarrow^{l}_{R}$.

Theorem 3 (Semideterminacy). *Suppose* $\vdash h$ host-ok.

1. *For a* RUN *host state and a given system call, either exactly one successful rule applies, or the call fails (several* fail *rules may be in competition): if* $h \in$ RUN, $f : TL \rightarrow TL'$ *and* $\vdash v : TL$ *then exactly one of the following two cases holds.*
   - $\left(\exists_1 R \in \text{succeed} \cup \text{enter2} \cup \{exit.1\} \,.\, h \rightarrow^{f\ v}_{R}\right) \wedge \left(\neg\exists R \in \text{fail} \,.\, h \rightarrow^{f\ v}_{R}\right)$
   - $\left(\neg\exists R \in \text{succeed} \cup \text{enter2} \cup \{exit.1\} \,.\, h \rightarrow^{f\ v}_{R}\right) \wedge \left(\exists_{\geq 1} R \in \text{fail} \,.\, h \rightarrow^{f\ v}_{R}\right)$
2. *For an* OP2 *host state, either exactly one successful rule applies, or the call fails (several* slowfail *rules may be in competition), or it is blocked: if* $h \in$ OP2 *then exactly one of the following three cases holds.*
   - $\left(\exists_1 R \in \text{slowsucceed} \cup \{console.print.2, select.5\} \,.\, h \rightarrow_{R}\right) \wedge \left(\neg\exists R \in \text{slowfail} \,.\, h \rightarrow_{R}\right)$
   - $\left(\neg\exists R \in \text{slowsucceed} \cup \{console.print.2, select.5\} \,.\, h \rightarrow_{R}\right) \wedge \left(\exists_{\geq 1} R \in \text{slowfail} \,.\, h \rightarrow_{R}\right)$
   - $\left(\neg\exists R \in \text{slowsucceed} \cup \{console.print.2, select.5\} \cup \text{slowfail} \,.\, h \rightarrow_{R}\right)$
3. *The host can accept any message destined for it, by exactly one rule:*
   $\vdash \text{IP}(i_3, i_4, body)$ msg-ok $\wedge\ i_4 \in ifds \wedge i_4 \notin$ LOOPBACK $\implies \exists_1 R \in \text{accept} \,.\, h \rightarrow^{\text{IP}(i_3,i_4,body)}_{R}$
4. *If the oq contains a message, there is exactly one rule that deals with it:*
   $oq \neq$ NIL $\implies \exists_1 R \in \text{emit} \cup \text{internaldelivery} \,.\, h \rightarrow_{R}$

Note that the first two clauses do not exclude the possibility of badfail, slowbadfail *or intr rules.*

Proof. By inspection of the axioms, using the the observation that the success of dosend (*ie.* of enqueue) is determined solely by *oqf*.

Theorem 4 (Receptiveness and Progress). *Suppose* $\vdash h$ host-ok.

1. *If the host is in a* RUN *host thread state, any well-typed library call may be performed:*
   $t =$ RUN $\wedge\ f : TL \rightarrow TL' \in$ LIB $\wedge \vdash v : TL \implies h \xrightarrow{f\ v}$
2. *If the host is in a* RET *host thread state, a unique value will be returned:*
   $t \in$ RET $\implies \exists_1 r. h \xrightarrow{\overline{r}}$
3. *The host can accept any message destined for it:*
   $\vdash \text{IP}(i_3, i_4, body)$ msg-ok $\wedge\ i_4 \in ifds \wedge i_4 \notin$ LOOPBACK $\implies h \xrightarrow{\text{IP}(i_3,i_4,body)}$

Proof. Clauses 1 and 3 follow from Theorem 3; clause 2 is immediate from Theorem 1 and the *ret.1* axiom.

Theorem 5 (Absence of Pathological Host-Thread Deadlocks). *If* $\vdash h$ host-ok, *e is a thread LTS and* instep(h, e) *then:*

1. *if* $e \xrightarrow{\overline{f\ v}}$ *then* $h \xrightarrow{f\ v}$*; and*
2. *if* $h \xrightarrow{\overline{r}}$ *then* $e \xrightarrow{r}$.

Proof. An immediate consequence of the previous results and the thread LTS axioms.

Theorem 6 (Preservation – Network). *If* $\vdash N$ network *and* $N \xrightarrow{l} N'$ *then* $\vdash N'$ network

Proof. Induction on the derivations.

3 The Model: Detailed Operational Semantics

3.1 SOCKET

socket.1 **succeed**

$$\text{HOST}(ifds, \text{RUN}, s, oq, oqf) \xrightarrow{\text{socket}()} \text{HOST}(ifds, \text{RET (OK}fd), \text{SOCK}(fd, *, *, *, *, *, \text{FLAGS}(\text{FALSE}, \text{FALSE}), \text{NIL}) :: s, oq, oqf)$$

$fd \notin \text{sockfds}(s)$

In BSD, this would be $\text{FLAGS}(\text{TRUE}, \text{FALSE})$

socket.2 **bad fail, as no file descriptors or file table space**

$$H(\text{RUN}) \xrightarrow{\text{socket}()} H(\text{RET (FAIL } e))$$

$e \in \{\text{EMFILE}, \text{ENFILE}\}$

The errors are 'Too many open files' and 'File table overflow'.

3.2 BIND

bind.1 $(*, *)$ **succeed, autobinding**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, *, *, *, *, es, f, mq)) \xrightarrow{\text{bind}(fd, *, *)} F(ifds, \text{RET (OK}()), \text{SOCK}(fd, *, \uparrow p'_1, *, *, es, f, mq))$$

$p'_1 \in \text{unused}(F) \cap \text{ephemeral}$

bind.2 $(\uparrow i, *)$ **succeed, autobinding**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, *, *, *, *, es, f, mq)) \xrightarrow{\text{bind}(fd, \uparrow i, *)} F(ifds, \text{RET (OK}()), \text{SOCK}(fd, \uparrow i, \uparrow p'_1, *, *, es, f, mq))$$

$p'_1 \in \text{unused}(F) \cap \text{ephemeral}$ and $i \in ifds$

bind.3 $(*, \uparrow p)$ **succeed**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, *, *, *, *, es, f, mq)) \xrightarrow{\text{bind}(fd, *, \uparrow p)} F(ifds, \text{RET (OK}()), \text{SOCK}(fd, *, \uparrow p, *, *, es, f, mq))$$

$p \notin$ privileged and $\forall\, \text{SOCK}(fd', is'_1, ps'_1, is'_2, ps'_2, es', f', mq') \in F.$
$(ps'_1 = \uparrow p) \implies (\text{reuseaddr } (f) = \text{TRUE} = \text{reuseaddr } (f'))$

Note that the i_1 doesn't get chosen until a datagram is sent, as in [Ste98, p.92].

bind.4 $(\uparrow i, \uparrow p)$ **succeed**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, *, *, *, *, es, f, mq)) \xrightarrow{\text{bind}(fd, \uparrow i, \uparrow p)} F(ifds, \text{RET (OK())}, \text{SOCK}(fd, \uparrow i, \uparrow p, *, *, es, f, mq))$$

$i \in ifds$ and $p \notin$ privileged and $\forall\ \text{SOCK}(fd', is'_1, ps'_1, is'_2, ps'_2, es', f', mq') \in F.$
$(ps'_1 = \uparrow p$ and $(is'_1 = *$ or $is'_1 = \uparrow i)) \implies$ (reuseaddr $(f) =$ TRUE $=$ reuseaddr $(f'))$

Bind failures

We assume that any unsuccessful bind leaves the socket state unchanged. In Linux, *bind.7* (EADDRNOTAVAIL) wins over *bind.5* (EACCES) and *bind.6* (EADDRINUSE), and *bind.5* (EACCES) wins over *bind.6* (EADDRINUSE). We choose not to specify this (see §2.3.11). If we did want to, we would add $is = *$ or $((is = \uparrow i) \implies i \in ifds)$ to *bind.5* and $(is = *$ or $((is = \uparrow i) \implies i \in ifds))$ and $p \notin$ privileged to *bind.6*. Moreover, *bind.8* (EINVAL) wins over the other three. To express this, we would replace $\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)$ in *bind.5*, *bind.6*, *bind.7* by $\text{SOCK}(fd, *, *, *, *, es, f, mq)$.

bind.5 $(is, \uparrow p)$ **fail, as we don't have access for port** p

$$F(ifds, \text{RUN}, \text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) \xrightarrow{\text{bind}(fd, is, \uparrow p)} F(ifds, \text{RET (FAIL EACCES)}, \text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq))$$

$p \in$ privileged

On Linux, *is* is written into the SOCK(...); we choose *not* to do so because it would violate our tidy socket invariant. Since the socket is not placed on the delivery list, we can hide this by letting our wrapper for getsockname test the local port, and, if it is $*$, return $*$ for local *is*.

bind.6 $(is, \uparrow p)$ **fail, as port** p **is already in use**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) \xrightarrow{\text{bind}(fd, is, \uparrow p)} F(ifds, \text{RET (FAIL EADDRINUSE)}, \text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq))$$

$\neg(\forall \text{SOCK}(fd', is'_1, ps'_1, is'_2, ps'_2, es', f', mq') \in F.(ps'_1 = \uparrow p$ and $(is'_1 = *$ or $is = *$ or $is'_1 = is))$
$\implies$ (reuseaddr $(f) =$ TRUE $=$ reuseaddr $(f'))$

bind.7 $(\uparrow i, ps)$ **fail, as** i **is not one of our ip addresses**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) \xrightarrow{\text{bind}(fd, \uparrow i, ps)} F(ifds, \text{RET (FAIL EADDRNOTAVAIL)}, \text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq))$$

$i \notin ifds$

bind.8 (is, ps) **fail, as the socket has a non-**$*$ **port already**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, is_1, \uparrow p_1, is_2, ps_2, es, f, mq)) \xrightarrow{\text{bind}(fd, is, ps)} F(ifds, \text{RET (FAIL EINVAL)}, \text{SOCK}(fd, is_1, \uparrow p_1, is_2, ps_2, es, f, mq))$$

bind.9 $(is, *)$ **fail, as there are no ephemeral ports left**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, *, *, *, *, es, f, mq)) \xrightarrow{\text{bind}(fd, is, *)} F(ifds, \text{RET}\ (\text{FAIL}\ e), \text{SOCK}(fd, *, *, *, *, es, f, mq))$$

$\text{unused}(F) \cap \text{ephemeral} = \emptyset$ and $e \in \text{ephemeralErrors}$

This error is awkward to test, as the test machines run out of files (ENFILE from socket) at about 3500, before the ephemeral ports are exhausted. The error set ephemeralErrors = {EAGAIN, EADDRINUSE} is therefore speculative.

3.3 CONNECT

Our type for connect forbids the use of $*$ for the IP address. In practice, using 0 here requests loopback transmission; it has the same effect as using 127.0.0.1.

connect.1 $(is_1 = *)$ **succeed, autobinding if necessary**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, *, ps_1, *, *, es, f, mq)) \xrightarrow{\text{connect}(fd, i, ps)} F(ifds, \text{RET}\ (\text{OK}()), \text{SOCK}(fd, \uparrow i_1, \uparrow p'_1, \uparrow i, ps, es, f, mq))$$

$i_1 \in \text{outroute}(ifds, i)$ and $p'_1 \in \text{autobind}(ps_1, F)$

connect.2 $(is_1 = \uparrow i_1)$ **succeed**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, \uparrow i_1, \uparrow p_1, is_2, ps_2, es, f, mq)) \xrightarrow{\text{connect}(fd, i, ps)} F(ifds, \text{RET}\ (\text{OK}()), \text{SOCK}(fd, \uparrow i_1, \uparrow p_1, \uparrow i, ps, es, f, mq))$$

connect.3 **fail, as there are no ephemeral ports left**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, *, *, *, *, es, f, mq)) \xrightarrow{\text{connect}(fd, i, ps)} F(ifds, \text{RET}\ (\text{FAIL}\ e), v)$$

$\text{unused}(F) \cap \text{ephemeral} = \emptyset$ and $e \in \text{ephemeralErrors}$
and $\vdash v$ socket-ok$(ifds)$ and $\text{sockfd}(v) = fd$

See remark for *bind.9*. The rule leaves the socket in a indeterminate state, as we do not want to specify exactly when the `getport` is called.

3.4 DISCONNECT

A disconnect is a C-library connect with AF_UNSPEC. [Ste98, p.226] suggests some architectures would give EAFNOSUPPORT for a successful disconnect; we would hide that in the wrapper.

disconnect.1 **succeed**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, is_1, \uparrow p_1, is_2, ps_2, es, f, mq)) \xrightarrow{\text{disconnect}\ fd} F(ifds, \text{RET}\ (\text{OK}()), \text{SOCK}(fd, *, \uparrow p_1, *, *, es, f, mq))$$

Note that this leaves the local port in place, perhaps surprisingly.

disconnect.2 **succeed, but also autobind**

$$F(ifds, \textsc{Run}, \textsc{Sock}(fd, *, *, *, *, es, f, mq)) \xrightarrow{\text{disconnect } fd} F(ifds, \textsc{Ret}\ (\text{OK}()), \textsc{Sock}(fd, *, \uparrow p_1, *, *, es, f, mq))$$

$p_1 \in \text{unused}(F) \cap \text{ephemeral}$

This autobind may be surprising, but the resulting state *can* receive msgs on $*, p_1$.

disconnect.3 **fail, as there are no ephemeral ports left**

$$F(ifds, \textsc{Run}, \textsc{Sock}(fd, *, *, *, *, es, f, mq)) \xrightarrow{\text{disconnect } fd} F(ifds, \textsc{Ret}\ (\textsc{Fail}\ e), v)$$

$\text{unused}(F) \cap \text{ephemeral} = \emptyset$ and $e \in \text{ephemeralErrors}$
and $\vdash v$ socket-ok$(ifds)$ and $\text{sockfd}(v) = fd$

See remark for *connect.3*.

3.5 GETSOCKNAME, GETPEERNAME

getsockname.1 **succeed**

$$F(ifds, \textsc{Run}, \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) \xrightarrow{\text{getsockname } fd} F(ifds, \textsc{Ret}\ (\text{OK}(is_1, ps_1)), \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq))$$

See comment at *bind.5* (EACCES). This involves a thin wrapper over the C system call; if we get $(\uparrow is, *)$ from the C system call then return $(*, *)$. This $ps_1 = *$ case of the rule covers for the weird Linux behaviour of a failing bind due to EACCES or EADDRINUSE.

getpeername.1 **succeed**

$$F(ifds, \textsc{Run}, \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) \xrightarrow{\text{getpeername } fd} F(ifds, \textsc{Ret}\ (\text{OK}(is_2, ps_2)), \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq))$$

This involves a thin wrapper over the C system call: if we get ENOTCONN then return $(*, *)$.

3.6 GETERR, GETSOCKOPT, SETSOCKOPT

A geterr is a C-library getsockopt with SOL_SOCKET and SO_ERROR.

geterr.1 **succeed**

$$F(ifds, \textsc{Run}, \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) \xrightarrow{\text{geterr } fd} F(ifds, \textsc{Ret}\ (\text{OK}es), \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, *, f, mq))$$

getsockopt.1 **succeed**

$$F(ifds, \textsc{Run}, \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) \xrightarrow{\text{getsockopt}(fd,\, opt)} F(ifds, \textsc{Ret}\ (\text{OK}\, b), \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq))$$

b = bsdcompat (f) if opt = SO_BSDCOMPAT, and
b = reuseaddr (f) if opt = SO_REUSEADDR.

setsockopt.1 **succeed**

$$F(ifds, \textsc{Run}, \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) \xrightarrow{\text{setsockopt}(fd,\, opt,\, b)} F(ifds, \textsc{Ret}\ (\text{OK}()), \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f', mq))$$

f' = setbsdcompat (f, b) if opt = SO_BSDCOMPAT, and
f' = setreuseaddr (f, b) if opt = SO_REUSEADDR.

3.7 SENDTO

Passing $*$ as the second argument to sendto corresponds in C to giving `sendto` a null `sockaddr` pointer; *cf.*[Ste98, p224]. Our typing prohibits passing $*$ for either the IP address or the port. In practice, we guess passing a zero IP sends to loopback (presumably as 127.0.0.1), and passing a zero port yields an EINVAL error.

We omit `send` since it is simply a trivial wrapper around `sendto`. We are arbitrarily ignoring `sendmsg`, even though it may have useful functionality.

sendto.1 **succeed**

$$\textsc{Host}(ifds, \textsc{Run}, S(\textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, *, f, mq)), oq, oqf) \xrightarrow{\text{sendto}(fd,\, ips,\, data,\, nb)} \textsc{Host}(ifds, \textsc{Ret}\ (\text{OK}()), S(\textsc{Sock}(fd, is_1, \uparrow p'_1, is_2, ps_2, *, f, mq)), oq', oqf')$$

$p'_1 \in \text{autobind}(ps_1, S)$
and $(oq', oqf', \textsc{true}) \in \text{dosend}(ifds, (ips, data), (is_1, \uparrow p'_1, is_2, ps_2), oq, oqf)$
and $\text{size}(data) \leq \text{UDPpayloadMax}$ and $(ips \neq *$ or $is_2 \neq *)$.

sendto.2 **block, entering Sendto2 state**

$$\textsc{Host}(ifds, \textsc{Run}, S(\textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, *, f, mq)), oq, oqf) \xrightarrow{\text{sendto}(fd,\, ips,\, data,\, \textsc{false})} \textsc{Host}(ifds, \textsc{Sendto2}\ (fd, ips, data), S(\textsc{Sock}(fd, is_1, \uparrow p'_1, is_2, ps_2, *, f, mq)), oq', oqf')$$

$p'_1 \in \text{autobind}(ps_1, S)$ and $(oq', oqf', \textsc{false}) \in \text{dosend}(ifds, (ips, data), (is_1, \uparrow p'_1, is_2, ps_2), oq, oqf)$
and $(ips \neq *$ or $is_2 \neq *)$

sendto.3 **fail, as would block**

$$\textsc{Host}(ifds, \textsc{Run}, S(\textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, *, f, mq)), oq, oqf)$$
$$\xrightarrow{\text{sendto}(fd, ips, data, \textsc{true})}$$
$$\textsc{Host}(ifds, \textsc{Ret}\ (\textsc{Fail}\ \text{EAGAIN}), S(\textsc{Sock}(fd, is_1, \uparrow p_1', is_2, ps_2, *, f, mq)), oq', oqf')$$

$p_1' \in \text{autobind}(ps_1, S)$ and $(oq', oqf', \textsc{false}) \in \text{dosend}(ifds, (ips, data), (is_1, \uparrow p_1', is_2, ps_2), oq, oqf)$
and $(ips \neq * \text{ or } is_2 \neq *)$

Note that in Linux EWOULDBLOCK and EAGAIN are aliased.

sendto.4 **fail, as socket unconnected**

$$F(ifds, \textsc{Run}, \textsc{Sock}(fd, is_1, ps_1, *, *, *, f, mq))$$
$$\xrightarrow{\text{sendto}(fd, *, data, nb)} F(ifds, \textsc{Ret}\ (\textsc{Fail}\ \text{ENOTCONN}), \textsc{Sock}(fd, is_1, \uparrow p_1', *, *, *, f, mq))$$

$p_1' \in \text{autobind}(ps_1, \text{socks}(F))$

[Ste98] states this should give EDESTADDRREQ, not ENOTCONN.

sendto.5 **fail, as socket in an error state**

$$F(ifds, \textsc{Run}, \textsc{Sock}(fd, is_1, \uparrow p_1, is_2, ps_2, \uparrow e, f, mq))$$
$$\xrightarrow{\text{sendto}(fd, ips, data, nb)} F(ifds, \textsc{Ret}\ (\textsc{Fail}\ e), \textsc{Sock}(fd, is_1, \uparrow p_1, is_2, ps_2, *, f, mq))$$

Recall that by the socket-ok invariant if the error component is $e\uparrow$ then the local port is not $*$.

sendto.6 **fail, as *data* size too big**

$$\textsc{Host}(ifds, \textsc{Run}, S(\textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)), oq, oqf)$$
$$\xrightarrow{\text{sendto}(fd, ips, data, nb)}$$
$$\textsc{Host}(ifds, \textsc{Ret}\ (\textsc{Fail}\ \text{EMSGSIZE}), S(\textsc{Sock}(fd, is_1, ps_1', is_2, ps_2, es, f, mq)), oq, oqf)$$

$\text{size}(data) > \text{UDPpayloadMax}$ and $ps_1' \in \{ps_1\} \cup \text{image}(\uparrow)\text{autobind}(ps_1, S)$

In Linux, *sendto.5* takes precedence over this, so EMSGSIZE cannot happen in an error state. In a normal state, it does not set the error flag. What happens on a SENDTO2 if an error flag is set is open. Note that we let $ps_1' = ps_1$ nondeterministically, as the size check may be before or after the autobind. On Linux this is unnecessary as the autobind is performed before the EMSGSIZE check. Note that we nondeterministically allow the message size check to fail either on entry to SENDTO2 or on exit.

sendto.7 **fail, as there are no ephemeral ports left**

$$\textsc{Host}(ifds, \textsc{Run}, S(\textsc{Sock}(fd, *, *, *, *, *, f, mq)), oq, oqf)$$
$$\xrightarrow{\text{sendto}(fd, ips, data, nb)} \textsc{Host}(ifds, \textsc{Ret}\ (\textsc{Fail}\ e), S(v), oq, oqf)$$

$\text{autobind}(*, S) = \emptyset$ and $e \in \text{ephemeralErrors}$
and $\vdash v$ socket-ok$(ifds)$ and $\text{sockfd}(v) = fd$

See remark for *connect.3*.

sendto.8 **slow succeed**

$$\text{Host}(ifds, \text{Sendto2}\ (fd, ips, data), S(\text{Sock}(fd, is_1, ps_1, is_2, ps_2, *, f, mq)), oq, oqf) \xrightarrow{\overline{\text{OK}()}} \text{Host}(ifds, \text{Run}, S(\text{Sock}(fd, is_1, ps_1, is_2, ps_2, *, f, mq)), oq', oqf')$$

$(oq', oqf', \text{TRUE}) \in \text{dosend}(ifds, (ips, data), (is_1, ps_1, is_2, ps_2), oq, oqf)$
and $\text{size}(data) \leq \text{UDPpayloadMax}$

Note that the Sendto2 state can only be entered by the *sendto.2* rule, so we have ($ips \neq *$ or $is_2 \neq *$) in the host-ok invariant.

sendto.9 **slow fail, as socket has entered an error state**

$$\text{Host}(ifds, \text{Sendto2}\ (fd, ips, data), S(\text{Sock}(fd, is_1, ps_1, is_2, ps_2, \uparrow e, f, mq)), oq, oqf) \xrightarrow{\overline{\text{Fail}\ e}} \text{Host}(ifds, \text{Run}, S(\text{Sock}(fd, is_1, ps_1, is_2, ps_2, *, f, mq)), oq, oqf)$$

sendto.10 **slow fail, as** *data* **size too big**

$$\text{Host}(ifds, \text{Sendto2}\ (fd, ips, data), s, oq, oqf) \xrightarrow{\overline{\text{Fail EMSGSIZE}}} \text{Host}(ifds, \text{Run}, s, oq, oqf)$$

$\text{size}(data) > \text{UDPpayloadMax}$

This rule leaves the socket state unchanged. One might wish to allow it to have been disturbed, nondeterministically.

3.8 RECVFROM

We omit `recv` since it is simply a trivial wrapper around `recvfrom`. We are arbitrarily ignoring `recvmsg`, even though it provide useful functionality.

We do not return the ifid component of the message queue element, as it seems sufficiently awkward to obtain under Linux that one really wouldn't want to do it all the time. We keep it in the model to support an extended recvfrom in future.

recvfrom.1 **succeed**

$$F(ifds, \text{Run}, \text{Sock}(fd, is_1, \uparrow p_1, is_2, ps_2, *, f, (\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data)), ifid) :: mq)) \xrightarrow{\text{recvfrom}(fd, nb)} F(ifds, \text{Ret}\ (\text{OK}(i_3, ps_3, data)), \text{Sock}(fd, is_1, \uparrow p_1, is_2, ps_2, *, f, mq))$$

recvfrom.2 **block, entering Recvfrom2 state**

$$F(ifds, \text{Run}, \text{Sock}(fd, is_1, ps_1, is_2, ps_2, *, f, \text{NIL})) \xrightarrow{\text{recvfrom}(fd, \text{FALSE})} F(ifds, \text{Recvfrom2}\ fd, \text{Sock}(fd, is_1, \uparrow p_1', is_2, ps_2, *, f, \text{NIL}))$$

$p_1' \in \text{autobind}(ps_1, \text{socks}(F))$

The autobinding can be observed by guessing the next ephemeral port.

recvfrom.3 **fail, as would block**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, is_1, ps_1, is_2, ps_2, *, f, \text{NIL})) \xrightarrow{\text{recvfrom}(fd, \text{TRUE})} F(ifds, \text{RET (FAIL EAGAIN)}, \text{SOCK}(fd, is_1, \uparrow p'_1, is_2, ps_2, *, f, \text{NIL}))$$

$p'_1 \in \text{autobind}(ps_1, \text{socks}(F))$

Note that in Linux EWOULDBLOCK and EAGAIN are aliased.

recvfrom.4 **fail, as socket in an error state**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, is_1, \uparrow p_1, is_2, ps_2, \uparrow e, f, mq)) \xrightarrow{\text{recvfrom}(fd, nb)} F(ifds, \text{RET (FAIL } e), \text{SOCK}(fd, is_1, \uparrow p_1, is_2, ps_2, *, f, mq))$$

recvfrom.5 **fail, as there are no ephemeral ports left**

$$F(ifds, \text{RUN}, \text{SOCK}(fd, *, *, *, *, *, f, \text{NIL})) \xrightarrow{\text{recvfrom}(fd, nb)} F(ifds, \text{RET (FAIL } e), v)$$

$\text{autobind}(*, F) = \emptyset$ and $e \in \text{ephemeralErrors}$
and $\vdash v$ socket-ok($ifds$) and $\text{sockfd}(v) = fd$

See remark for *connect.3*.

recvfrom.6 **slow succeed**

$$F(ifds, \text{RECVFROM2 } fd, \text{SOCK}(fd, is_1, \uparrow p_1, is_2, ps_2, *, f, (\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data)), ifid) :: mq)) \xrightarrow{\overline{\text{OK}(i_3, ps_3, data)}} F(ifds, \text{RUN}, \text{SOCK}(fd, is_1, \uparrow p_1, is_2, ps_2, *, f, mq))$$

Note that the RECVFROM2 state can only be entered by the *recvfrom.2* rule, so we have $ps_1 = \uparrow p_1$ in the host-ok invariant.

recvfrom.7 **slow fail, as socket has entered an error state**

$$F(ifds, \text{RECVFROM2 } fd, \text{SOCK}(fd, is_1, \uparrow p_1, is_2, ps_2, \uparrow e, f, mq)) \xrightarrow{\overline{\text{FAIL } e}} F(ifds, \text{RUN}, \text{SOCK}(fd, is_1, \uparrow p_1, is_2, ps_2, *, f, mq))$$

3.9 CLOSE

close.1 **succeed**

$$\text{HOST}(ifds, \text{RUN}, s_1@[\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)]@s_2, oq, oqf) \xrightarrow{\text{close } fd} \text{HOST}(ifds, \text{RET (OK())}, s_1@s_2, oq, oqf)$$

3.10 SELECT

select.1 **enter Select2 state**

$$H(\textsc{Run}) \xrightarrow{\mathsf{select}(readseq, writeseq, tms)} H(\textsc{Select2}\ (readseq, writeseq, tms))$$

$readseq@writeseq \subseteq \mathrm{sockfds}(H)$ and $tms = \uparrow i \implies 0 \leq i$

select.2 **fail, as timeout negative**

$$H(\textsc{Run}) \xrightarrow{\mathsf{select}(readseq, writeseq, \uparrow tm)} H(\textsc{Ret}\ (\textsc{Fail}\ \mathrm{EINVAL}))$$

$tm < 0$

select.3 **slow succeed, with at least one *fd* ready**

$$Q(\textsc{Select2}\ (readseq, writeseq, tms), s, oqf) \xrightarrow{\overline{\mathrm{OK}(readseq', writeseq')}} Q(\textsc{Run}, s, oqf)$$

$readseq' = [fd \mid fd \in readseq$ and $\exists \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in s\,.\,(mq \neq \textsc{nil} \vee es \neq *)]$
$writeseq' = [fd \mid fd \in writeseq$ and $oqf = \textsc{false}$ and $\exists \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in s]$
$readseq' \neq \textsc{nil}$ or $writeseq' \neq \textsc{nil}$

Note that (by our approximate treatment of *oqf*) all sockets become writable simultaneously.

select.4 **slow succeed, with zero timeout**

$$Q(\textsc{Select2}\ (readseq, writeseq, \uparrow 0), s, oqf) \xrightarrow{\overline{\mathrm{OK}(readseq', writeseq')}} Q(\textsc{Run}, s, oqf)$$

$readseq' = [fd \mid fd \in readseq$ and $\exists \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in s\,.\,(mq \neq \textsc{nil} \vee es \neq *)]$
$writeseq' = [fd \mid fd \in writeseq$ and $oqf = \textsc{false}$ and $\exists \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in s]$
$\wedge\ readseq' = \textsc{nil} \wedge writeseq' = \textsc{nil}$

select.5 **slow timeout decrement**

$$Q(\textsc{Select2}\ (readseq, writeseq, \uparrow tm), s, oqf) \xrightarrow{\tau} Q(\textsc{Select2}\ (readseq, writeseq, \uparrow(tm-1)), s, oqf)$$

$tm > 0$
$readseq' = [fd \mid fd \in readseq$ and $\exists \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in s\,.\,(mq \neq \textsc{nil} \vee es \neq *)]$
$writeseq' = [fd \mid fd \in writeseq$ and $oqf = \textsc{false}$ and $\exists \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in s]$
$readseq' = \textsc{nil}$ and $writeseq' = \textsc{nil}$

3.11 GETIFADDRS

getifaddrs.1 **succeed**

$$\text{HOST}(ifds, \text{RUN}, s, oq, oqf) \xrightarrow{\text{getifaddrs()}} \text{HOST}(ifds, \text{RET (OK}iflist), s, oq, oqf)$$

$iflist \in \text{orderings}(\{(ifid, ipslist, iprimary, netmask) \mid (ifid, iset, iprimary, netmask) \in ifds \text{ and } ipslist \in \text{orderings}(iset)\})$

Note this returns the interfaces in any arbitrary order each time; likewise the aliases within each interface. If one wanted otherwise, *ifds* and *iset* should be lists.

3.12 CONSOLE IO

console.print.1 **enter print2 state**

$$H(\text{RUN}) \xrightarrow{\text{print_endline_flush } v} H(\text{PRINT2 } v)$$

console.print.2 **print the string on console**

$$H(\text{PRINT2 } v) \xrightarrow{\overline{\text{console } v}} H(\text{RET (OK())})$$

3.13 CONVERSIONS

convert.port.1 **succeed**

$$H(\text{RUN}) \xrightarrow{\text{port_of_int } v} H(\text{RET (OK}v))$$

$1 \leq v \leq 65535$

convert.port.2 **fail**

$$H(\text{RUN}) \xrightarrow{\text{port_of_int } v} H(\text{RET (FAIL EINVAL)})$$

$v < 1$ or $v > 65535$

convert.ip.1 **succeed**

$$H(\text{RUN}) \xrightarrow{\text{ip_of_string } v} H(\text{RET (OK}i))$$

v is a valid non-zero IP dotted-quad with value i

convert.ip.2 **fail**

$$\frac{H(\textsc{Run})}{\text{ip_of_string } v \longrightarrow H(\textsc{Ret}\ (\textsc{Fail}\ \text{EINVAL}))}$$

v is not a valid non-zero IP dotted-quad

3.14 EBADF/ENOTSOCK

Fail as the fd is not a file descriptor, or is a file descriptor but not a socket These may apply to any of the operations that take an fd as argument, so beware when doing case analysis on operations.

notsockfd.1 select **fail, as some** *fd* **not a socket file descriptor**

$$\frac{H(\textsc{Run})}{\text{select}(\mathit{readseq}, \mathit{writeseq}, \mathit{tms}) \longrightarrow H(\textsc{Ret}\ (\textsc{Fail}\ \text{EBADF}))}$$

$\exists fd \in (\mathit{readseq} \cup \mathit{writeseq}) \setminus \text{sockfds}(H)$

The error is 'Bad file number'.

notsockfd.2 *OP* **fail, as** *fd* **not a socket file descriptor**

$$\frac{H(\textsc{Run})}{OP(fd) \longrightarrow H(\textsc{Ret}\ (\textsc{Fail}\ e))}$$

$fd \notin \text{sockfds}(H)$ and $e \in \{\text{ENOTSOCK}, \text{EBADF}\}$

The errors are 'Socket operation on non-socket' and 'Bad file number'.

3.15 ENOBUFS/ENOMEM AND EINTR

The ENOBUFS/ENOMEM rules may apply to any of the operations that take an fd as argument, and the EINTR rule may apply to any of the operations that may block, so beware when doing case analysis on operations. We conservatively allow ENOBUFS/ENOMEM in *any* of the $OP_2(fd)$ or SELECT2 v states. In the rules for $OP(fd)$ and $OP_2(fd)$ we allow the resulting socket state to be arbitrary, as we expect that a failing operation may leave the socket in bad state.

intr.1 **slow intr, as system call interrupted**

$$\frac{\textsc{Host}(\mathit{ifds}, \mathit{op}\ v, s, \mathit{oq}, \mathit{oqf})}{\overline{\textsc{Fail EINTR}} \longrightarrow \textsc{Host}(\mathit{ifds}, \textsc{Run}, s, \mathit{oq}, \mathit{oqf})}$$

$op \in \{\textsc{Sendto2}, \textsc{Recvfrom2}, \textsc{Select2}, \textsc{Print2}\}$

This rule leaves the socket state unchanged, as we presume that the interrupt occurs in a coherent state.

badmem.1 socket **bad fail, as no space**

$$H(\textsc{Run}) \xrightarrow{\mathsf{socket}()} H(\textsc{Ret}\ (\textsc{Fail}\ e))$$

$e \in \{\text{ENOMEM}, \text{ENOBUFS}\}$

The errors are (no space to allocate) and (no space for buffers) respectively.

badmem.2 OP **bad fail, as no space**

$$F(ifds, \textsc{Run}, \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) \xrightarrow{OP(fd)} F(ifds, \textsc{Ret}\ (\textsc{Fail}\ e), v)$$

$e \in \{\text{ENOMEM}, \text{ENOBUFS}\}$ and $\vdash v$ socket-ok$(ifds)$ and sockfd$(v) = fd$

The errors are (no space to allocate) and (no space for buffers) respectively.

badmem.3 OP_2 **slow bad fail, as no space**

$$F(ifds, OP_2(fd), \textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) \xrightarrow{\overline{\textsc{Fail}\ e}} F(ifds, \textsc{Run}, v)$$

$e \in \{\text{ENOMEM}, \text{ENOBUFS}\}$ and $\vdash v$ socket-ok$(ifds)$ and sockfd$(v) = fd$

The errors are (no space to allocate) and (no space for buffers) respectively.

badmem.4 **Select2 slow bad fail, as no space**

$$H(\textsc{Select2}\ v) \xrightarrow{\overline{\textsc{Fail}\ e}} H(\textsc{Run})$$

$e \in \{\text{ENOMEM}, \text{ENOBUFS}\}$

The errors are 'Out of memory' and 'No buffer space available'.

3.16 EXIT

exit.1 **exit**

$$\textsc{Host}(ifds, \textsc{Run}, s, oq, oqf) \xrightarrow{\mathsf{exit}\ ()} \textsc{Host}(ifds, \textsc{Term}, [], oq, oqf)$$

See §2.3.14. This rule removes all the sockets (matching Unix behaviour, supposing that all sockets in the machine were created by the process). It leaves the oq and oqf unchanged, so messages may still be delivered outwards.

3.17 RET

ret.1 **return value** v **from fast system call to thread**

$$\text{HOST}(ifds, \text{RET } v, s, oq, oqf) \xrightarrow{\overline{v}} \text{HOST}(ifds, \text{RUN}, s, oq, oqf)$$

3.18 DELIVERY

OUT

delivery.out.1 **put UDP or ICMP to the network from** oq

$$\text{HOST}(ifds, t, s, oq, oqf) \xrightarrow{\overline{\text{IP}(i_3, i_4, body)}} \text{HOST}(ifds, t, s, oq', oqf')$$

$((\text{IP}(i_3, i_4, body)), oq', oqf') \in \text{dequeue}(oq, oqf)$
and $i_4 \notin \text{LOOPBACK} \cup \text{MARTIAN}$
and $i_3 \notin \text{MARTIAN}$

Note that dequeue gives an empty set of possible outcomes if $oq = [\,]$.

delivery.out.martian **discard martian IP from** oq

$$\text{HOST}(ifds, t, s, oq, oqf) \xrightarrow{\tau} \text{HOST}(ifds, t, s, oq', oqf')$$

$((\text{IP}(i_3, i_4, body)), oq', oqf') \in \text{dequeue}(oq, oqf)$ and $i_4 \in \text{MARTIAN}$

This rule is speculative. Note that it applies to UDP and to ICMP, and to loopback destinations.

IN - UDP

delivery.in.udp.1 **get UDP from network and deliver to a matching socket**

$$\text{HOST}(ifds, t, s, oq, oqf) \xrightarrow{\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data))} \text{HOST}(ifds, t, S(\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq :: (\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data)), ifid))), oq, oqf)$$

$\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in \text{lookup } s\ (i_3, ps_3, i_4, ps_4)$
and $S(\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) = s$
and $(ifid, iset, _, _) \in ifds$ and $i_4 \in iset$
and $i_4 \notin \text{LOOPBACK}$ and $i_3 \notin \text{MARTIAN} \cup \text{LOOPBACK}$

delivery.in.udp.2 **get UDP from network but generate ICMP, as no matching socket**

$$\text{HOST}(ifds, t, s, oq, oqf) \xrightarrow{\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data))} \text{HOST}(ifds, t, s, oq', oqf')$$

$i_4 \in ifds$ and lookup $s\ (i_3, ps_3, i_4, ps_4) = \emptyset$
and $(oq', oqf', ok) \in \{(oq, oqf, \text{TRUE})\} \cup$
enqueue(IP$(i_4, i_3, \text{ICMP_PORT_UNREACH}(i_3, ps_3, i_4, ps_4))$, oq, oqf)
and $i_4 \notin$ LOOPBACK and $i_3 \notin$ MARTIAN $\cup$ LOOPBACK

Note that ICMP generation is unreliable; the Linux kernel has per-type-of-ICMP rate-limiting to control denial-of-service attacks. We take a nondeterministic approximation. Note also that the ICMP is dropped if the outqueue is full ($ok = \text{FALSE}$).

delivery.in.martian.3 **get martian IP from network and discard**

$$\text{HOST}(ifds, t, s, oq, oqf) \xrightarrow{\text{IP}(i_3, i_4, v)} \text{HOST}(ifds, t, s, oq, oqf)$$

$i_4 \in ifds$ and $i_4 \notin$ LOOPBACK
and (loopback(IP$(i_3, i_4, v)) \vee$ martian(IP(i_3, i_4, v)))

This is a garbage collection rule to pull martian UDP or ICMP off of the network and discard them.

IN - ICMP

We have not investigated what happens if the ICMP header source and destination do not match the embedded UDP source and destination, supposing for the time being that matching is done on the embedded data. X here is $PORT$ or $HOST$.

delivery.in.icmp.1 **get ICMP from the network, setting error in a matching socket**

$$\text{HOST}(ifds, t, s, oq, oqf) \xrightarrow{\text{IP}(i'_4, i'_3, \text{ICMP_X_UNREACH}(i_3, ps_3, i_4, ps_4))} \text{HOST}(ifds, t, S(\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es', f, mq)), oq, oqf)$$

$S(\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq)) = s$
and $\text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in$ lookup $s(i_3, ps_3, i_4, ps_4)$
and $m = \text{IP}(i'_4, i'_3, \text{ICMP_X_UNREACH}(i_3, ps_3, i_4, ps_4))$
and $i'_3 \in ifds$ and $\neg(\text{loopback}(m) \vee \text{martian}(m))$
and $es' =$ if $(is_2 \neq *)$ or $\neg$(bsdcompat f) then $\uparrow$ECONNREFUSED else es

As we are not sure which destination (i_3, i'_3) address the matching is done on, we add the LOOPBACK check to both to make the rule sensible.

delivery.in.icmp.2 **get ICMP from the network and drop, as no matching socket**

$$\text{HOST}(ifds, t, s, oq, oqf) \xrightarrow{\text{IP}(i'_4, i'_3, \text{ICMP_X_UNREACH}(i_3, ps_3, i_4, ps_4))} \text{HOST}(ifds, t, s, oq, oqf)$$

$i'_3 \in ifds$ and lookup $s(i_3, ps_3, i_4, ps_4) = \emptyset$
and $m = \text{IP}(i'_4, i'_3, \text{ICMP_X_UNREACH}(i_3, ps_3, i_4, ps_4))$
and $\neg(\text{loopback}(m) \vee \text{martian}(m))$

LOOPBACK

delivery.loopback.udp.1 **get loopback UDP from oq and deliver to a matching socket**

$$\begin{array}{l}\quad \textsc{Host}(ifds, t, s, oq, oqf) \\ \xrightarrow{\tau} \textsc{Host}(ifds, t, S(\textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, \\ \qquad\qquad mq :: (\mathrm{IP}(i_3, i_4, \mathrm{UDP}(ps_3, ps_4, data)), ifid))), oq', oqf')\end{array}$$

$(\mathrm{IP}(i_3, i_4, \mathrm{UDP}(ps_3, ps_4, data)), oq', oqf') \in \mathrm{dequeue}(oq, oqf)$
and $i_4 \in \mathrm{LOOPBACK}$
and $s = S(\textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq))$
and $\textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in \mathrm{lookup}\ s(i_3, ps_3, i_4, ps_4)$
and $\mathrm{IF}(ifid, \mathrm{LOOPBACK}, \mathrm{localhost}, 255/8) \in ifds$

Note that $\{i_3, i_4\} \cap \mathrm{MARTIAN} = \emptyset$ is guaranteed by invariants.

delivery.loopback.udp.2 **get a loopback UDP from oq but generate loopback ICMP, as no matching socket**

$$\begin{array}{l}\quad \textsc{Host}(ifds, t, s, oq, oqf) \\ \xrightarrow{\tau} \textsc{Host}(ifds, t, s, oq'', oqf'')\end{array}$$

$(\mathrm{IP}(i_3, i_4, \mathrm{UDP}(ps_3, ps_4, data)), oq', oqf') \in \mathrm{dequeue}(oq, oqf)$
and $i_4 \in \mathrm{LOOPBACK}$
and $\mathrm{lookup}\ s(i_3, ps_3, i_4, ps_4) = \emptyset$
and $(oq'', oqf'', ok) \in \mathrm{enqueue}(\mathrm{IP}(i_4, i_3, \mathrm{ICMP_PORT_UNREACH}(i_4, ps_4, i_3, ps_3)), oq', oqf')$

ICMPs are dropped if the outqueue is full (the *ok* flag is ignored). Note that $\{i_3, i_4\} \cap \mathrm{MARTIAN} = \emptyset$ is guaranteed by invariants.

delivery.loopback.icmp.1 **get a loopback ICMP from oq and deliver (by setting es) to a matching socket**

$$\begin{array}{l}\quad \textsc{Host}(ifds, t, s, oq, oqf) \\ \xrightarrow{\tau} \textsc{Host}(ifds, t, S(\textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es', f, mq)), oq', oqf')\end{array}$$

$(\mathrm{IP}(i'_4, i'_3, \mathrm{ICMP_PORT_UNREACH}(i_3, ps_3, i_4, ps_4)), oq', oqf') \in \mathrm{dequeue}(oq, oqf)$
and $i'_3 \in \mathrm{LOOPBACK}$
and $s = S(\textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq))$
and $\textsc{Sock}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \in \mathrm{lookup}\ s(i_4, ps_4, i_3, ps_3)$
and $es' = \mathrm{if}\ (is_2 \neq *)\ \mathrm{or}\ \neg(\mathrm{bsdcompat}\ f)\ \mathrm{then}\ \uparrow\mathrm{ECONNREFUSED}\ \mathrm{else}\ es$

Note that $\{i'_4, i'_3, i_3, i_4\} \cap \mathrm{MARTIAN} = \emptyset$ is guaranteed by invariants.

delivery.loopback.icmp.2 **get a loopback ICMP from oq and drop, as no socket matches**

$$\begin{array}{l}\quad \textsc{Host}(ifds, t, s, oq, oqf) \\ \xrightarrow{\tau} \textsc{Host}(ifds, t, s, oq', oqf')\end{array}$$

$(\mathrm{IP}(i'_4, i'_3, \mathrm{ICMP_PORT_UNREACH}(i_3, ps_3, i_4, ps_4)), oq', oqf') \in \mathrm{dequeue}(oq, oqf)$
and $i'_3 \in \mathrm{LOOPBACK}$
and $\mathrm{lookup}\ s(i_4, ps_4, i_3, ps_3) = \emptyset$

Note that $\{i'_4, i'_3\} \cap \mathrm{MARTIAN} = \emptyset$ is guaranteed by invariants.

4 MiniCaml

MiniCaml is designed to be a sublanguage of OCaml 3.00 [L+00]. The semantics is largely standard; the details can be found in Appendix A.

4.1 Syntax The *types* are given by the grammar marked TL in Figure 1 (except $T\,\mathsf{err}$), together with:

$$T ::= \cdots \mid T \to T' \mid T\ \mathsf{ref} \mid \mathsf{exn}$$

The nullary and unary *constructors* C_0 and C_1 are all language constructors of arity 0 and 1 of Figure 2, together with

$$\begin{array}{ll} \mathrm{UDP} & : \mathsf{error} \to \mathsf{exn} \\ \textsc{Match_failure} & : \mathsf{string} * \mathsf{int} * \mathsf{int} \to \mathsf{exn} \end{array}$$

The *expressions*, *matches*, and *patterns* are

$$\begin{array}{rcl} e & ::= & \mathrm{C}_0 \mid \mathrm{C}_1\ e \mid e :: e \mid (e_1, .., e_n) \mid x \\ & & \mid\ \mathrm{ref}\ \mid !\mid :=\mid e = e \mid \mathrm{if}\ e\ \mathrm{then}\ e\ \mathrm{else}\ e \mid \mathrm{while}\ e\ \mathrm{do}\ e\ \mathrm{done} \mid e; e \\ & & \mid \mathrm{function}\ mtch \mid e\ e \mid \mathrm{let}\ p = e\ \mathrm{in}\ e \mid \mathrm{let\ rec}\ x = \mathrm{function}\ mtch\ \mathrm{in}\ e \\ & & \mid \mathrm{raise}\ e \mid \mathrm{try}\ e\ \mathrm{with}\ mtch \\ & & \mid f \mid \mathrm{RET}_{T\,\mathsf{err}} \mid \mathrm{RET}_{\mathsf{void}} \\ mtch & ::= & p \to e \mid (p \to e \mid mtch) \\ p & ::= & _ \mid x \mid \mathrm{C}_0 \mid \mathrm{C}_1\ p \mid (p_1, .., p_n) \mid p :: p \mid (p : T) \end{array}$$

This is standard except for the last line of the e grammar. We let f range over calls of the library interface LIBEX, consisting of $f : T \to T'$ for each $f : T \to T'\,\mathsf{err} \in \mathrm{LIB}$ and $\mathsf{exit} : () \to \mathsf{void}$. $\mathrm{RET}_{T\,\mathsf{err}}$ is an expression waiting for a return value of type $T\,\mathsf{err}$ from a library call. The value will be mapped into either a value of type T or a raise of an exception. $\mathrm{RET}_{\mathsf{void}}$ is a placeholder for a terminated program (that has invoked exit, raised an uncaught exception, or reduced to a value).

4.2 Typing The typing rules are standard (though for simplicity we allow polymorphism only for constructors, not for let-bound identifiers). They define judgements:

$$E \vdash e : T \qquad E \vdash mtch : T \to T' \qquad \vdash p : T \rhd E \qquad E \vdash s\ \mathrm{store}$$

Here *stores* s are finite partial functions from identifiers to values. We say a *program* is an expression e that does not contain an occurrence of RET such that $\vdash e : T$ for some T.

4.3 Operational Semantics The operational semantics defines a transition relation over pairs e, s of an expression and a store, where $E \vdash e : T \wedge E \vdash s\ \mathrm{store}$ for some E, T. Stores are sometimes elided. It is standard, using reduction axioms and evaluation contexts (taking the evaluation order of the `ocamlopt` native-code compiler), except for transitions with non-τ labels, for which we take the axioms

$$\begin{array}{lcll} f\ v, s & \xrightarrow{\overline{f\ v}} & \mathrm{RET}_{T'}, s & \text{if } f : T \to T' \text{ in LIB} \\ \mathrm{RET}_{T\,\mathsf{err}}, s & \xrightarrow{\mathrm{OK}(v)} & v, s & \text{if } \vdash v : T \\ \mathrm{RET}_{T\,\mathsf{err}}, s & \xrightarrow{\textsc{Fail}\ (v)} & \mathrm{raise\ UDP}(v), s & \text{if } \vdash v : \mathsf{error} \end{array}$$

(closed under evaluation contexts), together with top-level axioms for exit of a terminated value or unhandled exception:

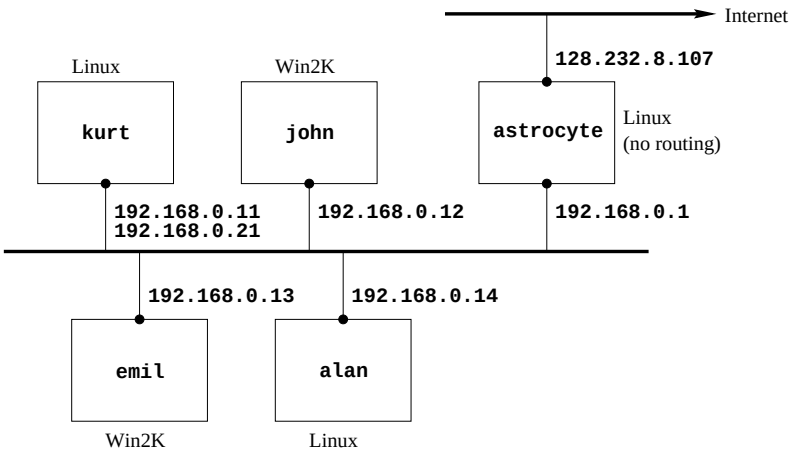
$$v, s \xrightarrow{\overline{\text{exit ()}}} \text{RET}_{\mathsf{void}}, s$$
$$\text{raise } v, s \xrightarrow{\overline{\text{exit ()}}} \text{RET}_{\mathsf{void}}, s$$

4.4 Sanity Properties We prove theorems stating that types are preserved by transitions, that runtime errors do not occur, and that the semantics satisfies the thread LTS axioms of §2.2.5.

4.5 Implementation We have implemented an OCaml module `Udplang` that provides all the language types of Figure 1 that are not OCaml types (except T err), the language constructors of Figure 2 for types $T\uparrow$, error and sockopt, and (almost all of) the library LIBEX. Its signature can be found in Appendix B. The example programs in this paper are automatically typeset from working code, omitting an `open Udplang;;` at the beginning of each program and using mathematized concrete syntax, writing $(), T\uparrow, \uparrow e, *, \rightarrow$ for `unit`, `T lift`, `Lift e`, `Star` and `->`. unit, T lift, LIFT e, STAR and `->`. We also typeset sugared lists $[e_1; ..; e_n]$ as $[e_1, .., e_n]$. We have not implemented any automated check that an OCaml program lies within MiniCaml.

5 Validation

5.1 What We Did To develop and validate our host semantics, we set up a test network: a non-routed 10BaseT-Ethernet subnet with four dedicated machines (two Linux and two Win2K), accessible via an additional interface on one of our Linux workstations.



This gives us a quiet and predictable network over which we are able to send and receive datagrams and observe the behaviour of the various UDP and sockets implementations. In a few cases (mainly in order to generate ICMPs with more significant delays) we also sent datagrams into the wider Internet.

The machines KURT, JOHN, EMIL, ALAN are all Intel Pentium 150s. KURT and ALAN run Linux 2.2.16-22, and JOHN and EMIL run Windows 2000 5.00.2195. The workstation ASTROCYTE is an Athlon 800 running a departmentally-modified version of Linux 2.2.16-4. All machines use 3Com network interface cards.

Tests were written in C, using the `glibc` 2.1.92 sockets library. Initially we wrote a large number of *ad hoc* tests, C programs that display the results of short sequences of socket calls, and also observed the resulting network traffic with the `tcpdump` utility. We wrote a daemon, `udpdaemon`, which allowed us to remotely send and receive UDP datagrams and test for returned ICMPs. Certain hard-to-test issues (*eg.*, the range of ephemeral ports, the reliability of ICMP generation) were resolved by inspecting the Linux kernel source code, which also helped us build an intuition for the OS's view of socket interaction.

Later, to more thoroughly validate the semantics as a whole, we translated the host operational semantics into C; we wrote an automatic tool, `udpautotest`, that simulates the model in parallel with the real socket calls (augmented with a wrapper providing tracing and the thin LIB abstraction of Section 1.7). 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.

5.2 Limitations The testing has a number of limitations, however. We only test the sockets library as a closed box: the internal socket state is not directly observable or manipulable, and so all our observations are via the sockets interface itself. Our automated test framework relies heavily on first using a sequence of socket, bind, connect to put a socket in a known state, then invoking the socket call under test, and finally testing the state of the socket afterwards using getsockname, getpeername, and getsockopt. UDP messages and, indirectly, ICMPs are generated by sendto and

tested for by recvfrom. This means that we have to trust that these calls are 'telling the truth'. We can gain some confidence of this by direct observation of the network using `tcpdump`, and by inspecting the implementation source code where available.

Our testing has other limitations.

- It is clearly impossible to exhaust all cases – every possible port, every possible IP address, every possible file descriptor – and so we chose a set of representative values: for ports, for example, we chose a privileged port, an already-bound port, the wildcard port, an arbitrary unbound port, and a known unbound port.
- While `tcpdump` did allow us to inspect arbitrary packets on the wire, we had no way of *injecting* arbitrary packets onto the wire; while UDP datagrams are easily generated, we were only able to generate ICMP messages by sending UDP datagrams to unbound ports on known hosts; we did not test ICMP_HOST_UNREACH behaviour. Similarly, we were unable to generate martian or loopback packets on the wire to validate their treatment.
- We were unable to test most of the 'bad errors', such as running out of kernel memory or ephemeral ports; similarly some details of the behaviour of blocking calls, slow returns, and interrupted system calls; the exact behaviour of delivery and loopback rules (unobservable without examining internal state); multiple interfaces and/or multiple IPs per interface. We did not thoroughly test the reuseaddr behaviour, or calls with non-socket file descriptors.
- We did not look for Byzantine behaviour – if we attempted to send a datagram on port 1745, we did not check for datagrams on ports 1746 or 42420, for example.
- Loss is very rare on our single subnet, and as far as we are aware reordering and duplication never occur. To test these we will need to extend our tests through one or more routers, probably using the Internet rather than our own isolated network.

We therefore cannot regard the semantics as definitive, and would be interested to hear of discrepancies between it and real system behaviour.

5.3 Idealisation and Abstraction 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.

In the model, the library call $(f\ v)$ and return $(\overline{r})$ host transitions are atomic, as are the delivery transitions, and delivery transitions can be arbitrarily interleaved with calls and returns. It is tempting to think of the calls and returns as occuring exactly when control is passed from user to library code (well-defined up to a few cycles). To be strictly accurate, however, we would have to take into account when the Linux kernel takes locks, and how the scheduler operates. It would be interesting to develop a precise statement of the sense in which our model is an abstraction of such a lower-level model, though useful properties of distributed infrastructures may well not depend on such details.

Using an untimed semantics restricts the possible treatments of message delivery. Our model is asynchronous, ensuring that it does not have unrealistic message ordering properties, but consequently messages do persist in a network until either they are delivered or the loss rule fires. (The alternative, of a synchronous model, would be more seriously flawed.) In an actual network, an ethernet-connected machine must be up for the interval during which a packet passes its interface in order to receive it; to reflect these details requires a more complex timed semantics.

5.4 MiniCaml 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.

6 Examples

6.1 The Single Sender We first show the possible traces of the single sender and single receiver from §1.6. Their state spaces and transition relations are illustrated in Appendix C. Consider

$$\begin{aligned} N = \;& \text{ALAN}{\cdot}e_s \mid \text{ALAN}{\cdot}\text{HOST}(\mathit{ifds}_{\text{ALAN}}, \text{RUN}, [\,], [\,], \text{FALSE}) \\ & \mid \text{KURT}{\cdot}e_r \mid \text{KURT}{\cdot}\text{HOST}(\mathit{ifds}_{\text{KURT}}, \text{RUN}, [\,], [\,], \text{FALSE}) \end{aligned}$$

and discount rules modelling interrupted system calls or the OS running out of file descriptors or kernel memory(the badfail, *intr.1* and slowbadfail rules). Suppose loss (*drop.1*) may occur, but duplication (*dup.1*) and host failure (*host.crash.*$*$) do not.

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

$$N \xrightarrow{\overline{\text{KURT}{\cdot}\text{console "ready"}}} \xrightarrow{\overline{\text{ALAN}{\cdot}\text{console "sending"}}} \xrightarrow{\overline{\text{KURT}{\cdot}\text{console "hello"}}} N'$$

and resulting state

$$\begin{aligned} N' = \;& \text{ALAN}{\cdot}\text{RET}_{\mathsf{void}} \mid \text{ALAN}{\cdot}\text{HOST}(\mathit{ifds}_{\text{ALAN}}, \text{TERM}, [\,], [\,], \text{FALSE}) \\ & \mid \text{KURT}{\cdot}\text{RET}_{\mathsf{void}} \mid \text{KURT}{\cdot}\text{HOST}(\mathit{ifds}_{\text{KURT}}, \text{TERM}, [\,], [\,], \text{FALSE}) \end{aligned}$$

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

$$N \xrightarrow{\overline{\text{ALAN}{\cdot}\text{console "sending"}}} \xrightarrow{\overline{\text{KURT}{\cdot}\text{console "ready"}}} \xrightarrow{\overline{\text{KURT}{\cdot}\text{console "hello"}}} N'$$

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

$$N \xrightarrow{\overline{\text{ALAN}{\cdot}\text{console "sending"}}} \xrightarrow{\overline{\text{KURT}{\cdot}\text{console "ready"}}} N''$$

ending with ALAN's state terminated as before but KURT in a blocked RECVFROM2 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.

6.2 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):

```
e_A =
 let p = port_of_int (7655) in
 let i = ip_of_string ("192.168.0.11") in
 let fd = socket() in
 let _ = bind(fd, *, ↑p) in
 let _ = connect(fd, i, ↑p) in
 let _ = print_endline_flush "pinging" in
 let _ = sendto(fd, *, "ping", FALSE) in
 let (fds, _) = select([fd], [], ↑5000000) in
 if fds = [] then
  print_endline_flush "dead"
 else
  try
   let (_, _, v) = recvfrom(fd, FALSE) in
   print_endline_flush v
  with
   UDP(ECONNREFUSED)
     → print_endline_flush "down"
```

```
e_B =
 let p = port_of_int (7655) in
 let i = ip_of_string ("192.168.0.14") in
 let fd = socket() in
 let _ = bind(fd, *, ↑p) in
 let _ = connect(fd, i, ↑p) in
 let _ = print_endline_flush "ready" in
 let _ = recvfrom(fd, FALSE) in
 let _ = sendto(fd, *, "ack", FALSE) in
 print_endline_flush "done"
```

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_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.

It is straightforward to apply the MiniCaml axioms to determine the thread-LTS behaviour of each program. One may then determine the host's response to each transition and thus the behaviour of the system as a whole. 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.

A more useful version of this would have both e_A and e_B repeat. This is easily expressible, but the desired properties require a fair and/or timed semantics. More sophisticated examples often require multiple threads per host, so that one thread can be dedicated to a protocol.

7 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, JoCaml, and Nomadic Pict [INM87,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 Dam [AD99] is similar to ours: they aim to prove properties of real concurrent programs written in Erlang. They describe an operational 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].

8 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.

We hope that such models, giving precise behavioural descriptions (albeit only for a fragment of the full OS library interfaces), will usefully complement the more usual informal descriptions. Developing the semantics has required us to consider many erroneous and pathological cases; these may be uncommon but are also likely to be poorly understood, and hence may lead to coding errors.

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, *eg.* for a Java fragment. Incorporating fairness and time is required to capture interesting properties of algorithms. As discussed in §5, 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*.

A MiniCaml Definition

A.1 Syntax

The *types* T are given by the clauses of the Figure 1 grammar marked TL, except $T\,\mathsf{err}$, together with:

$$T ::= \cdots \mid T \to T' \mid T\ \mathsf{ref} \mid \mathsf{exn}$$

The nullary and unary *constructors* C_0 and C_1 are all language constructors of arity 0 and 1 of Figure 2, together with

$$\begin{array}{ll} \text{UDP} & : \mathsf{error} \to \mathsf{exn} \\ \text{MATCH_FAILURE} & : \mathsf{string} * \mathsf{int} * \mathsf{int} \to \mathsf{exn} \end{array}$$

Note the language constructors are all of arity 0, arity 1, or equal to :: or $(_,..,_)$. We let f range over calls of the library interface LIBEX, consisting of $f : T \to T'$ for each $f : T \to T' \,\mathsf{err} \in \text{LIB}$ and $\mathsf{exit} : () \to \mathsf{void}$. Take *operators* o

$$\begin{array}{lll} \text{ref} & : & T \to T\ \mathsf{ref} \\ ! & : & T\ \mathsf{ref} \to T \\ := & : & (T\ \mathsf{ref}) * T \to () \\ = & : & T * T \to \mathsf{bool} \qquad T \text{ not involving } \to, \mathsf{ref} \text{ or } \mathsf{string} \end{array}$$

We write $e := e'$ for $:= (e, e')$ and $e = e'$ for $= (e, e')$. The examples in this paper require equality, but do not require any other operators, *eg.*. for booleans, integers, or strings, so we omit them.

Expresssions

$$\begin{array}{lll} e ::= & \mathrm{C}_0 & \mathrm{C}_0 \text{ a constructor of arity 0} \\ & \mathrm{C}_1\ e & \mathrm{C}_1 \text{ a constructor of arity 1} \\ & e :: e & \text{Cons} \\ & (e_1, .., e_n) & \text{Tuple } (n \geq 2) \\ & f & f \in \text{LIB} \\ & o & o \text{ an operator} \\ & x & \text{Identifier} \end{array} \Bigg\} \; v$$

$$\begin{array}{ll} \text{if } e \text{ then } e \text{ else } e & \text{Conditional} \\ \text{while } e \text{ do } e \text{ done} & \text{Loop} \\ e; e & \text{Sequence} \\ \text{function } mtch & \text{Function} \\ e\ e & \text{Application} \\ \text{let } p = e \text{ in } e & \text{Local definition} \\ \text{let rec } x = \text{function } mtch \text{ in } e & \text{Recursive definition} \\ \text{raise } e & \text{Raise exception} \\ \text{try } e \text{ with } mtch & \text{Handle exception(s)} \\ \mathrm{RET}_{T\,\mathsf{err}} & \text{Await return from OS routine} \\ \mathrm{RET}_{\mathsf{void}} & \text{Terminated} \end{array}$$

Values The values v are the elements of the subgrammar of expressions marked 'v' above, together with all expressions of the form function *mtch*.

Matches

$$\begin{array}{rl} mtch ::= & p \to e \\ & p \to e | mtch \end{array}$$

Patterns

$p ::=$	_	Wildcard
	x	Variable
	C_0	C_0 a constructor of arity 0
	$\mathrm{C}_1\, p$	C_1 a constructor of arity 1
	$p :: p$	Cons
	$(p_1, .., p_n)$	Tuple ($n \geq 2$)
	$(p : T)$	Typed pattern

These are subject to the condition that all variables occuring in a pattern are distinct.

Binding: We work up to alpha equivalence throughout, with the obvious scoping of the constructs above. Note that MiniCaml does not allow rebinding of library functions or operators, contrary to OCaml. The syntactic categories relate as follows.

MiniCaml	OCaml
T	*typexpr*
e	*expr*
p	*pattern*
mtch	*pattern-matching*

A.2 Typing

Type Environments E are finite partial functions from identifiers to types. We write E, E' for their union, thereby asserting also that E and E' have disjoint domain. We say E *closed* if it declares only references, *ie.* if $\forall x \in \mathrm{dom} E\,.\, E(x) = T \implies \exists T'\,.\, T = T'$ ref . *Stores* s are finite partial functions from identifiers to values. The typing judgements are:

$E \vdash e : T$	under assumptions E, expression e has type T
$E \vdash mtch : T \to T'$	under assumptions E, *mtch* has type $T \to T'$
$\vdash p : T \rhd E$	pattern p matches type T, giving additional bindings E
$E \vdash s\,\mathrm{store}$	under assumptions E, store s is well-formed

The typing rules are in Figure 5. We say a *program* is an expression e that does not contain an occurrence of RET such that $\vdash e : T$ for some T.

A.3 Operational Semantics

For any program e, we define a thread LTS with labels Lthread (as defined in §2.2.5), states

$$\{\, e', s \mid \exists E, T\,.\, E \vdash e' : T \wedge E \vdash s\,\mathrm{store} \,\}$$

initial state $e, empty$, and transitions as defined below. Note that the $E \vdash s\,\mathrm{store}$ rule requires E closed; we are only interested in behaviour w.r.t. such closed E. Strictly, the transition relations are defined as the least relations over

$$\{\, e, s \mid \exists E, T\,.\, E \vdash e : T \wedge E \vdash s\,\mathrm{store} \,\} \times \{\, e, s \mid e, s \text{ an arbitrary expression and store} \,\}$$

satisfying the rules below.

$\boxed{E \vdash e : T}$

$$\frac{\mathrm{C}_0 : T}{E \vdash \mathrm{C}_0 : T} \qquad \frac{\mathrm{C}_1 : T \to T' \quad E \vdash e : T}{E \vdash \mathrm{C}_1\ e : T'} \qquad \frac{E \vdash e_1 : T \quad E \vdash e_2 : T\ \mathsf{list}}{E \vdash e_1 :: e_2 : T\ \mathsf{list}} \qquad \frac{E \vdash e_i : T_i\ \ i \in 1..n \quad n \geq 2}{E \vdash (e_1, .., e_n) : T_1 * .. * T_n}$$

$$\frac{f : T \to T' \in \mathrm{LIBEX}}{E \vdash f : T \to T'} \qquad \frac{o : T \to T'}{E \vdash o : T \to T'} \qquad \frac{}{E, x : T \vdash x : T}$$

$$\frac{E \vdash e_1 : \mathsf{bool} \quad E \vdash e_2 : T \quad E \vdash e_3 : T}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \qquad \frac{E \vdash e_1 : \mathsf{bool} \quad E \vdash e_2 : ()}{E \vdash \text{while } e_1 \text{ do } e_2 \text{ done} : ()} \qquad \frac{E \vdash e_1 : () \quad E \vdash e_2 : T}{E \vdash e_1 ; e_2 : T}$$

$$\frac{E \vdash mtch : T \to T'}{E \vdash \text{function } mtch : T \to T'} \qquad \frac{E \vdash e_1 : T \to T' \quad E \vdash e_2 : T}{E \vdash e_1\, e_2 : T'}$$

$$\frac{\vdash p : T_1 \rhd E' \quad E \vdash e_1 : T_1 \quad E, E' \vdash e_2 : T_2}{E \vdash \text{let } p = e_1 \text{ in } e_2 : T_2} \qquad \frac{E, x : T \to T' \vdash mtch : T \to T' \quad E, x : T \to T' \vdash e_2 : T_2}{E \vdash \text{let rec } x = \text{function } mtch \text{ in } e_2 : T_2} \qquad \frac{}{E \vdash \mathrm{RET}_{\mathsf{void}} : T}$$

$$\frac{E \vdash e : \mathsf{exn}}{E \vdash \text{raise } e : T} \qquad \frac{E \vdash e : T \quad E \vdash mtch : \mathsf{exn} \to T}{E \vdash \text{try } e \text{ with } mtch : T} \qquad \frac{}{E \vdash \mathrm{RET}_{T\,\mathsf{err}} : T}$$

$\boxed{E \vdash mtch : T \to T'}$

$$\frac{\vdash p : T \rhd E' \quad E, E' \vdash e : T'}{E \vdash p \to e : T \to T'} \qquad \frac{E \vdash p \to e : T \to T' \quad E \vdash mtch : T \to T'}{E \vdash p \to e | mtch : T \to T'}$$

$\boxed{\vdash p : T \rhd E'}$

$$\vdash _ : T \rhd empty \qquad \vdash x : T \rhd x : T \qquad \frac{\mathrm{C}_0 : T}{\vdash \mathrm{C}_0 : T \rhd empty} \qquad \frac{\mathrm{C}_1 : T \to T' \quad \vdash p : T \rhd E'}{\vdash \mathrm{C}_1\, p : T' \rhd E'}$$

$$\frac{\vdash p_1 : T \rhd E_1 \quad \vdash p_2 : T\ \mathsf{list} \rhd E_2}{\vdash p_1 :: p_2 : T\ \mathsf{list} \rhd E_1, E_2} \qquad \frac{\vdash p_i : T_i \rhd E_i\ \ i \in 1..n \quad n \geq 2}{\vdash (p_1, .., p_n) : T_1 * .. * T_n \rhd E_1, .., E_n} \qquad \frac{\vdash p : T \rhd E'}{\vdash (p : T) : T \rhd E'}$$

$\boxed{E \vdash s\ \mathrm{store}}$

$$\frac{\begin{array}{l} E \text{ closed} \\ \mathrm{dom}E = \mathrm{dom}s \\ \forall x : T\ \text{ref}\ \in \mathrm{dom}E\ .\ E \vdash s(x) : T \end{array}}{E \vdash s\ \mathrm{store}}$$

Fig. 5. MiniCaml Typing Rules

Evaluation contexts

$$
\begin{array}{lll}
C ::= & \mathrm{C}_1\ _ & \mathrm{C}_1 \text{ a constructor of arity 1} \\
& _ :: e & \\
& v :: _ & \\
& (e_1, .., e_{m-1}, _, v_{m+1}, .., v_n) & n \geq 2 \\
& \text{if } _ \text{ then } e_1 \text{ else } e_2 & \\
& _; e & \\
& _\ e & \\
& v\ _ & \\
& \text{let } p = _ \text{ in } e & \\
& \text{raise } _ & \\
& \text{try } _ \text{ with } mtch &
\end{array}
$$

The evaluation order is chosen to match that of the `ocamlopt` native code compiler, not the bytecode compiler. Note that in both, tuples are evaluated right-to-left.

Matching Define a partial function match(_, _) taking a value and a pattern (in which all variables are distinct) and giving a set of substitutions:

$$
\begin{array}{ll}
\text{match}(v, _) & = \{\} \\
\text{match}(v, x) & = \{v/x\} \\
\text{match}(v, (p : T)) & = \text{match}(v, p) \\
\text{match}(\mathrm{C}_0, \mathrm{C}_0) & = \{\} \\
\text{match}(\mathrm{C}_1\ v, \mathrm{C}_1\ p) & = \text{match}(v, p) \\
\text{match}(v_1 :: v_2, p_1 :: p_2) & = \text{match}(v_1, p_1) \cup \text{match}(v_2, p_2) \\
\text{match}((v_1, .., v_n), (p_1, .., p_n)) & = \text{match}(v_1, p_1) \cup ... \cup \text{match}(v_n, p_n)\ n \geq 2 \\
\text{match}(v, p) & \text{undefined otherwise}
\end{array}
$$

Reduction Axioms In the following, the store component is elided from left and right hand sides.

$$
\begin{array}{llll}
\text{if TRUE then } e_1 \text{ else } e_2 & \xrightarrow{\tau} e_1 & & 1 \\
\text{if FALSE then } e_1 \text{ else } e_2 & \xrightarrow{\tau} e_2 & & 2 \\
(); e & \xrightarrow{\tau} e & & 3 \\
\text{while } e_1 \text{ do } e_2 \text{ done} & \xrightarrow{\tau} \text{if } e_1 \text{ then } (e_2; \text{while } e_1 \text{ do } e_2 \text{ done}) \text{ else } () & & 4 \\
(\text{function } p_1 \to e_1 | ... | p_n \to e_n)\, v & \xrightarrow{\tau} \text{match}(v, p_i) e_i & (a) & 5 \\
(\text{function } p_1 \to e_1 | ... | p_n \to e_n)\, v & \xrightarrow{\tau} \text{raise MATCH_FAILURE } v' & (b) & 6 \\
\text{let } p = v \text{ in } e & \xrightarrow{\tau} \text{match}(v, p) e & \text{match}(v, p) \text{ defined} & 7 \\
\text{let } p = v \text{ in } e & \xrightarrow{\tau} \text{raise MATCH_FAILURE } v' & \text{match}(v, p) \text{ not defined and (d)} & 8 \\
\text{let rec } x = \text{function } mtch \text{ in } e & \xrightarrow{\tau} \{(\{\text{let rec } x = \text{function } mtch \text{ in } x/x\}\text{function } mtch)/x\} e & & 9 \\
C(\text{raise } v) & \xrightarrow{\tau} \text{raise } v & (c) & 10 \\
\text{try raise } v \text{ with } p_1 \to e_1 | .. | p_n \to e_n & \xrightarrow{\tau} \text{match}(v, p_i) e_i & (a) & 11 \\
\text{try } v \text{ with } p_1 \to e_1 | .. | p_n \to e_n & \xrightarrow{\tau} v & & 12
\end{array}
$$

(a) match(v, p_i) is defined and there is no $j < i$ with match(v, p_j) defined
(b) where not exists $i \in 1..n$ such that match(v, p_i) is defined, and v' is an arbitrary value such that $\vdash v' : \mathsf{string} * \mathsf{int} * \mathsf{int}$
(c) if not exists $p_1 \to e_1 | .. | p_n \to e_n$ and i st $C = \text{try } _ \text{ with } p_1 \to e_1 | .. | p_n \to e_n$ and match(v, p_i) defined
(d) $\vdash v' : \mathsf{string} * \mathsf{int} * \mathsf{int}$

Note the match failure rule introduces nondeterminism. In the OCaml implementation, the $\mathsf{string} * \mathsf{int} * \mathsf{int}$ gives the position in the source file of this code; we do not want to model that precisely.

Now rules for operators:

$$\begin{array}{llll}
\text{ref } v, s & \xrightarrow{\tau} x, (s, x \mapsto v) & x \notin \mathrm{dom} s & 13 \\
!x, s & \xrightarrow{\tau} s(x), s & x \in \mathrm{dom} s & 14 \\
x := v, s & \xrightarrow{\tau} (), (s \oplus x \mapsto v) & x \in \mathrm{dom} s & 15 \\
v = v & \xrightarrow{\tau} \text{TRUE} & & 16 \\
v = v' & \xrightarrow{\tau} \text{FALSE} & v \neq v' & 17
\end{array}$$

Note that the rule for ref introduces nondeterminism. That could be avoided by working up to cyclic bindings, but we choose not to for simplicity. Transitions with τ labels are closed under reduction contexts:

$$\frac{e, s \xrightarrow{\tau} e', s'}{C(e), s \xrightarrow{\tau} C(e'), s'}$$

Input and Output Transition axioms The transition relations with non-τ labels are the least containing the closure of axioms

$$\begin{array}{llll}
fv, s & \xrightarrow{\overline{fv}} \mathrm{RET}_{T'}, s & \text{if } f : T \to T' \text{ in LIB} & 18 \\
\mathrm{RET}_{T\,\mathsf{err}}, s & \xrightarrow{\mathrm{OK}(v)} v, s & \text{if } \vdash v : T & 19 \\
\mathrm{RET}_{T\,\mathsf{err}}, s & \xrightarrow{\textsc{Fail}\ (v)} \text{raise } \mathrm{UDP}(v), s & \text{if } \vdash v : \mathsf{error} & 20
\end{array}$$

under reduction contexts:

$$\frac{e, s \xrightarrow{l} e', s'}{C(e), s \xrightarrow{l} C(e'), s'}$$

together with

$$\begin{array}{lll}
v, s & \xrightarrow{\overline{\text{exit ()}}} \mathrm{RET}_{\mathsf{void}}, s & 21 \\
\text{raise } v, s & \xrightarrow{\overline{\text{exit ()}}} \mathrm{RET}_{\mathsf{void}}, s & 22
\end{array}$$

(not closed under reduction contexts).

In the implementation, the user can distinguish the two exit cases, both by an uncaught exception being printed and by the return status. The return code varies with the exception, though – we choose not to model this exactly.

Runtime errors and Termination

$$\begin{array}{lll}
\mathrm{iserr}(\text{if } v \text{ then } e_1 \text{ else } e_2, s) & v \notin \{\text{TRUE}, \text{FALSE}\} & 23 \\
\mathrm{iserr}(v; e, s) & v \neq () & 24 \\
\mathrm{iserr}(v_1\, v_2, s) & v_1 \text{ not of the forms function } \mathit{mtch},\ f, \text{ or } o & 25 \\
\mathrm{iserr}(!v, s) & v \notin \mathrm{dom} s & 26 \\
\mathrm{iserr}(v := v', s) & v \notin \mathrm{dom} s & 27
\end{array}$$

$$\mathrm{isterm}(\mathrm{RET}_{\mathsf{void}}) \qquad 28$$

$$\frac{\mathrm{iserr}(e, s)}{\mathrm{iserr}(C(e), s)} \qquad \frac{\mathrm{isterm}(e, s)}{\mathrm{isterm}(C(e), s)}$$

A.4 Sanity Properties

Lemma 1 (Matching yields well-typed substitutions).

$$\frac{\begin{array}{l}E \vdash v : T \\ \vdash p : T \rhd E' \\ \mathrm{match}(v,p)\ \textit{defined}\end{array}}{\begin{array}{l}\mathrm{dom}(\mathrm{match}(v,p)) = \mathrm{dom}(E') \\ \forall x : T' \in E' \,.\, E \vdash \mathrm{match}(v,p)(x) : T'\end{array}}$$

Lemma 2 (Substitution).

$$\frac{\begin{array}{l}E, E' \vdash e : T \\ \mathrm{dom}(\sigma) = \mathrm{dom}(E') \\ \forall x : T' \in E' \,.\, E \vdash \sigma(x) : T'\end{array}}{E \vdash \sigma\, e : T}$$

Theorem 7 (Type Preservation).

$$\frac{\begin{array}{l}E \vdash e : T \\ E \vdash s\ \mathrm{store} \\ e, s \xrightarrow{l} e', s'\end{array}}{\begin{array}{l}\exists E' \,.\, E' \vdash e' : T \wedge E' \vdash s'\ \mathrm{store} \\ l = \overline{fv} \implies \exists T_1, T_2 \,.\, f : T_1 \to T_2 \text{ in LIB and } \vdash v : T_1\end{array}}$$

Theorem 8 (Absence of runtime errors).

$$\frac{\begin{array}{l}E \vdash e : T \\ E \vdash s\ \mathrm{store}\end{array}}{\neg \mathrm{iserr}(e,s)}$$

Let r range over the set $\{1, .., 28\}$ of transition axioms, runtime error axioms, and termination axiom.

Lemma 3 (Unique Decomposition). *For any (expression,state) pair e, s there exists a unique sequence of evaluation contexts $\boldsymbol{C}$, expression e_0, and rule r such that*

$e = \boldsymbol{C}[e_0]$
e_0, s *is an instance of rule* r
$r \in \{21, 22\} \implies \boldsymbol{C} = _$

Theorem 9 (Thread LTS). *For any program e, the LTS above satisfies the thread LTS axioms of §2.2.5.*

Proof. (Sketch) Take the partition of states below.

$$\begin{array}{ll}
\mathrm{CALL}(f) & = \{\, \boldsymbol{C}[f\ v], s \mid \exists E, T' \,.\, E \vdash \boldsymbol{C}[f\ v] : T' \wedge E \vdash s\ \mathrm{store} \,\} \quad f \neq \mathsf{exit} \\
\mathrm{CALL}(\mathsf{exit}) & = \{\, \boldsymbol{C}[\mathsf{exit}\ v], s \mid \exists E, T' \,.\, E \vdash \boldsymbol{C}[\mathsf{exit}\ v] : T' \wedge E \vdash s\ \mathrm{store} \,\} \\
 & \quad \cup \{\, v, s \mid \exists E, T' \,.\, E \vdash v : T' \wedge E \vdash s\ \mathrm{store} \,\} \\
 & \quad \cup \{\, \mathrm{raise}\ v, s \mid \exists E, T' \,.\, E \vdash \mathrm{raise}\ v : T' \wedge E \vdash s\ \mathrm{store} \,\} \\
\mathrm{RET}_{T\,\mathsf{err}} & = \{\, \boldsymbol{C}[\mathrm{RET}_{T\,\mathsf{err}}], s \mid \exists E, T' \,.\, E \vdash \boldsymbol{C}[\mathrm{RET}_{T\,\mathsf{err}}] : T' \wedge E \vdash s\ \mathrm{store} \,\} \\
\mathrm{RET}_{\mathsf{void}} & = \{\, \boldsymbol{C}[\mathrm{RET}_{\mathsf{void}}], s \mid \exists E, T' \,.\, E \vdash \boldsymbol{C}[\mathrm{RET}_{\mathsf{void}}] : T' \wedge E \vdash s\ \mathrm{store} \,\} \\
\mathrm{RET}_{T} & = \emptyset \qquad T \text{ not of the forms } T\,\mathsf{err} \text{ or } \mathsf{void} \\
\mathrm{TAU} & = \text{all other states}
\end{array}$$

B MiniCaml Implementation: `udplang.mli`

```
type fd
type ip
type port
type error =   EACCES | EADDRINUSE | EADDRNOTAVAIL | EAGAIN | EBADF
             | ECONNREFUSED | EHOSTUNREACH | EINTR | EINVAL | EMFILE
             | EMSGSIZE | ENFILE | ENOBUFS | ENOMEM | ENOTCONN
             | ENOTSOCK
type netmask
type ifid
type sockopt = (* SO_BSDCOMPAT | *)  SO_REUSEADDR
type 'a lift = Star | Lift of 'a
type void

exception UDP of error

val socket       : unit                            -> fd
val bind         : fd*ip lift *port lift           -> unit
val connect      : fd*ip*port lift                 -> unit
(* val disconnect: fd                              -> unit                    *)
val getsockname  : fd                              -> (ip lift * port lift )
val getpeername  : fd                              -> (ip lift * port lift )
val sendto       : fd* (ip*port) lift *string*bool -> unit
val recvfrom     : fd*bool                         -> (ip*(port lift)*string)
(* val geterr    : fd                              -> error lift              *)
val getsockopt   : fd*sockopt                      -> bool
val setsockopt   : fd*sockopt*bool                 -> unit
val close        : fd                              -> unit
val select       : fd list * fd list * int lift    -> (fd list * fd list)
val port_of_int  : int                             -> port
val ip_of_string : string                          -> ip
(* val getifaddrs: unit   -> (ifid * ip * ip list * netmask) list             *)

val print_endline_flush : string -> unit
val exit                : unit   -> void

(* For select, the int measures microseconds. *)

(* Some LIB functionality is not yet implemented, as it is not easy to
do so above the OCAML 3.00 Unix library: geterr, getifaddrs, and
get/setsockopt with the SO_BSDCOMPAT socket option.  These are
commented out, therefore.  We add two exceptions, not in the model.
If the OS (via the OCaml Unix library) returns an error that is not
one of those above, BadUnixError is raised.  If it is detected that
the abstraction is broken in some other surprising way, BadError is
raised. *)

exception BadUnixError of Unix.error  * string
exception BadError of string
```

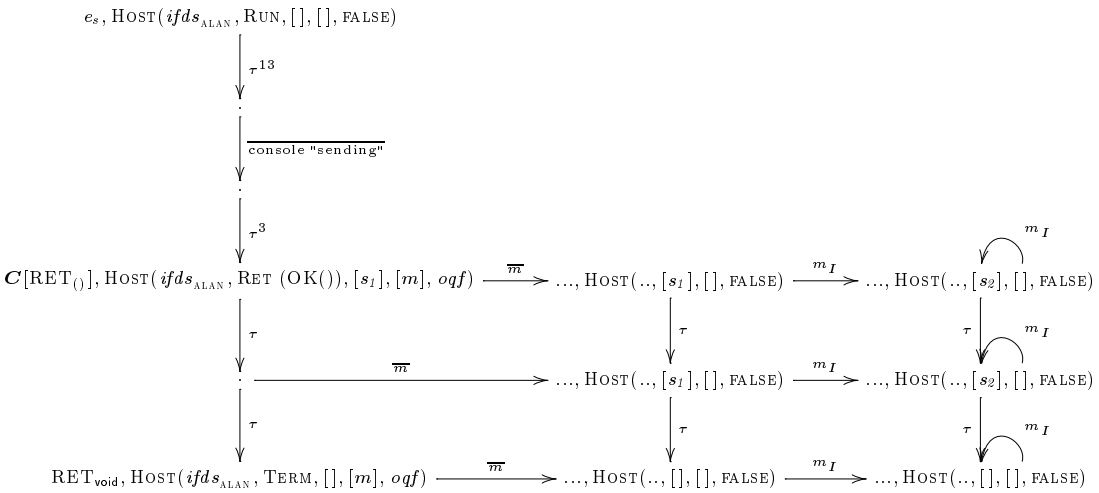
C The Single Sender: State Spaces

In this appendix we illustrate the state spaces of the two host/thread pairs in the single sender example of §6.1. In larger examples this quickly becomes impractical; it may be convenient to work with explicit symbolic characterisations of the state spaces. There are clearly many confluent pairs, which reasoning methods should take advantage of.

As the example is of a quiet network, with only the two hosts, we need not consider any incoming datagrams to ALAN until after m has been sent, after which there might be an ICMP coming back. Looking at this part of the transition system of

$$\text{ALAN}{\cdot}e_s \mid \text{ALAN}{\cdot}\text{HOST}(\mathit{ifds}_{\text{ALAN}}, \text{RUN}, [\,], [\,], \text{FALSE}),$$

therefore, we have:



where

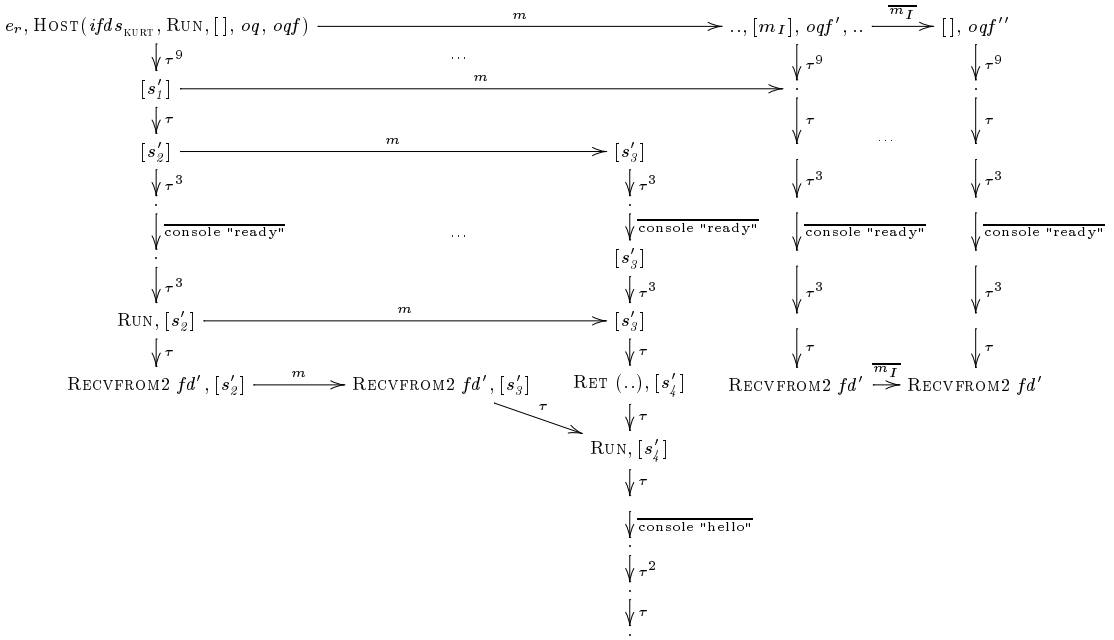
$$\begin{aligned}
s_0 &= \text{SOCK}(\mathit{fd}, *, *, *, *, *, f, [\,]) \\
s_1 &= \text{SOCK}(\mathit{fd}, \uparrow i_{\text{ALAN}}, \uparrow p_1, \uparrow i_{\text{KURT}}, \uparrow p, *, f, [\,]) \\
s_2 &= \text{SOCK}(\mathit{fd}, \uparrow i_{\text{ALAN}}, \uparrow p_1, \uparrow i_{\text{KURT}}, \uparrow p, \uparrow \text{ECONNREFUSED}, f, [\,]) \\
m &= \text{IP}(i_{\text{ALAN}}, i_{\text{KURT}}, \text{UDP}(\uparrow p_1, \uparrow 7654, \text{"hello"})) \\
m_{\text{I}} &= \text{IP}(i_{\text{KURT}}, i_{\text{ALAN}}, \text{ICMP_PORT_UNREACH}(i_{\text{ALAN}}, \uparrow p_1, i_{\text{KURT}}, \uparrow p))
\end{aligned}$$

and many state components are elided, as is the ALAN labelling.

For the receiver

$$\text{KURT}{\cdot}e_r \mid \text{KURT}{\cdot}\text{HOST}(ifds_{\text{KURT}}, \text{RUN}, [\,], [\,], \text{FALSE})$$

we have the transitions below, considering just a single incoming message m and again eliding many state components and the KURT labelling. Here we also omit (for lack of space!) transitions in which m is received before the socket is bound but an ICMP is *not* generated; these are from the initial state and its first 9 τ successors. The ellipses $\cdots$ indicate blocks of left-to-right m and $\overline{m_I}$ transitions.



where

$s'_1 = \text{SOCK}(fd', *, *, *, *, *, f, [\,])$
$s'_2 = \text{SOCK}(fd', \uparrow i_{\text{KURT}}, \uparrow p, *, *, *, f, [\,])$
$s'_3 = \text{SOCK}(fd', \uparrow i_{\text{KURT}}, \uparrow p, *, *, *, f, [m])$
$s'_4 = \text{SOCK}(fd', \uparrow i_{\text{KURT}}, \uparrow p, *, *, *, f, [\,])$

References

[AD99] Thomas Arts and Mads Dam. Verifying a distributed database lookup manager written in Erlang. In *World Congress on Formal Methods (1)*, pages 682–700, 1999.

[AP94] R. Amadio and S. Prasad. Localities and failures. In *Foundations of Software Technology and Theoretical Computer Science*, volume 880 of *LNCS*. Springer-Verlag, 1994.

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

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

[BH00] Martin Berger and Kohei Honda. The two-phase commit protocol in an extended π-calculus. In *Proceedings of the 7th International Workshop on Expressiveness in Concurrency, EXPRESS '00*, 2000.

[Bia94] Edoardo Biagioni. A structured TCP in standard ML. In *Proceedings of SIGCOMM 94*, 1994.

[Bra89] R. Braden. Requirements for internet hosts – communication layers, STD 3, RFC 1122. Internet Engineering Task Force, October 1989. `http://www.ietf.org/rfc.html`.

[CS00] Gian Luca Cattani and Peter Sewell. Models for name-passing processes: Interleaving and causal (extended abstract). In *Proceedings of LICS 2000: the 15th IEEE Symposium on Logic in Computer Science (Santa Barbara)*, pages 322–333, June 2000.

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

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of CONCUR '96*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, August 1996.

[GLV00] S. J. Garland, N. Lynch, and Mandana Vaziri. IOA: A language for specifying, programming and validating distributed systems. Reference guide, December 2000. Available `http://nms.lcs.mit.edu/~garland/IOA/`.

[Hay98] Mark Hayden. *The Ensemble System*. PhD thesis, Cornell University, January 1998. Technical Report TR98-1662.

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

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

[INM87] INMOS. *Occam2 Reference Manual*. Prentice-Hall, 1987.

[L+00] Xavier Leroy et al. *The Objective-Caml System, Release 3.00*. INRIA, April 27 2000. Available `http://caml.inria.fr/ocaml/`.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lyn96] Nancy A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

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

[Mul93] Sape J. Mullender. *Distributed Systems*. ACM Press, 1993.

[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 6, RFC 791. Internet Engineering Task Force, September 1981. `http://www.ietf.org/rfc.html`.

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[Rep91] John Reppy. CML: A higher-order concurrent language. In *Programming Language Design and Implementation*, pages 293–259. SIGPLAN, ACM, June 1991.

[RH97] James Riely and Matthew Hennessy. Distributed processes and location failures. In *Automata, Languages and Programming*, volume 1256 of *LNCS*, pages 471–481, Bologna, Italy, 7–11 July 1997. Springer-Verlag.

[Sew97] Peter Sewell. On implementations and semantics of a concurrent programming language. In *Proceedings of CONCUR '97. LNCS 1243*, pages 391–405, 1997.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated: The Protocols*, volume 1 of *Addison–Wesley Professional Computing Series*. Addison–Wesley, 1994.

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

[Swi01] The SwitchWare project. `http://www.cis.upenn.edu/~switchware`, 2001.

[TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In *Proceedings of CONCUR '96. LNCS 1119*, pages 278–298. Springer-Verlag, August 1996.

[WS00] Paweł T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, April–June 2000.

Index

() (constructor), 9
() (type), 8
(, .. ,), 9
∗, 9
∗, 8
:: , 7, 9
⊢ $v : T$, 7
@, 7
{..}, 9
↑, 9
↑, 8

ALAN, 9
append, 7
ASTROCYTE, 9
autobind, 21
autobinding, 20, 21

BADCLASS, 30
bc, 11
bind, 12
blocking, 15, 21, 30
bool, 8
bsdcompat , 11

C, 9
CALL, 17
close , 12, 13
Con, 9
connect, 12
constructor, 7
context, 10
conversion, 13

data, 7, 26
datagram, 7
dequeue, 27
disconnect , 12
disjoint_addr, 31
dosend, 23, 28
drop.1, 18, 29
dup.1, 18

e, 7, 16
E* (errors), 9
enqueue, 27
ephemeral, 20
ephemeralErrors, 25
err, 8, 12
error, 8
error flag, 11, 23
es, 7, 11
ETH0, 9
exit , 12

F, 10
f, 11
FAIL , 9
FALSE, 9
fast system call, 15, 21
fd, 8, 11
fd_3, 9
file descriptor, 11
FLAGS, 9, 11
flags, 8, 11
fragmentation, 26

geterr , 12, 29
getifaddrs, 12, 13, 30
getpeername , 12
getsockname , 12
getsockopt, 12, 13

H, 10
HOST, 8, 9
host, 8, 14
host, 8
host LTS, 16
⊢ h host-ok, 10
host.crash.∗, 18
hostid, 8, 14
hostThreadState, 8, 10
htsType, 11

i, 7
$i \in ifds$, 11
i/x, 30
ICMP_HOST_UNREACH, 7, 9
ICMP_PORT_UNREACH, 7, 9, 23
ICMP_X_UNREACH, 23
IF, 9, 11
ifd, 8
⊢ *ifds* ifd-set-ok, 31
ifds, 8, 11, 31
ifid, 8, 11
injection, 13
instep, 17
int, 8
integer, 9
interface, 11, 31
IP, 7, 9
ip, 8
ip_of_string , 12, 13
ipBody, 8
iprimary, 11
is, 7
iset, 11

KURT, 9

Lhost, 16
LIB, 12
library interface, 12
list, 8
list comprehension, 7
Lnetwork, 17
LO, 9
localhost, 30
lookup , 22, 28
LOOPBACK, 30
loopback, 31
Lthread, 16

m, 7
marshalling, 17
MARTIAN, 30
martian, 31
martian address, 30
match, 28
message, 7
message queue, 11, 29
mq, 11
msg, 7, 8
⊢ m msg-ok, 8
⊢ m msg-oq-ok, 8
MULTICAST, 30

N, 14
n, 14
name, 14, 17
netmask, 8, 11
network, 14
⊢ N network, 14
network operational semantics, 17
NIL, 9

octet-sequence, 9
OK, 9
OP, 13
OP_2, 10

option type, 8
oq, 10, 27
oqf, 10, 27
orderings, 7
outqueue, 10, 27, 29
outroute, 22

p, 7
$par.*$, 18
partial system, 14
port, 8
port_of_int , 12, 13
PRINT2 , 9, 10
print_endline_flush , 12
privileged, 20
PROG, 17
projection, 13
ps, 7

Q, 10

ra, 11
rate limiting, 23
recv, 29
recvfrom, 12
RECVFROM2 , 9, 10
recvmsg, 29
RET, 17
RET , 9, 10
RET_{TL}, 9
RET_{TL}, 17
return, 14
reuseaddr , 11
routing, 22
RUN, 9, 10

S, 10
s, 10
score, 28
select, 12, 13
SELECT2 , 9, 10
send, 29
send, 29
sendto, 12
SENDTO2 , 9, 10
set, 8
setbsdcompat , 11
setreuseaddr , 11
setsockopt, 12, 13
slow system call, 15
SO_BSDCOMPAT, 9, 13, 29
SO_REUSEADDR, 9, 13, 30
SOCK, 9, 11
socket, 11
socket, 8, 12
socket list, 10
socket option, 11, 13, 29
$\vdash s$ socket-ok, 11
sockets interface, 12
sockopt, 8
socks, 10
string, 8
synchronisation, 14, 17
system call, 14

T, 8
t, 10
TAU, 17
TERM, 9, 10
thread, 8, 14, 16
thread LTS, 16
TL, 8
tm, 7
tms, 7
transition relation, host, 16, 31–48
TRUE, 9
tuple constructor, 9
tuple type, 8
type, 7

UDP, 7, 9
UDPpayloadMax, 26
unit constructor, 9
unit type, 8
unused, 20

v, 7
value, 7
void, 8

zero port, 7, 28
ZERONET, 30