A Distributed Calculus with Local Areas of Communication

Tom Chothia Ian Stark
tpcc@dcs.ed.ac.uk stark@dcs.ed.ac.uk

Laboratory for Foundations of Computer Science
Division of Informatics
University of Edinburgh

August 2000

Abstract

This paper introduces a process calculus designed to capture the phenomenon of names which are known universally but always refer to local information. Our system extends the π-calculus so that a channel name can have within its scope several disjoint *local areas*. Such a channel name may be used for communication within an area, it may be sent between areas, but it cannot itself be used to transmit information from one area to another. Areas are arranged in a hierarchy of *levels*, distinguishing for example between a single application, a machine, or a whole network. We give an operational semantics for the calculus, and develop a type system that guarantees the proper use of channels within their local areas. We illustrate with models of an internet service protocol and a pair of distributed agents.

1 Introduction

Most computer programs make assumptions about the environment in which they operate: the facilities available, and how to use them. A C programmer will freely use the function `printf`, and expect that wherever their compiled code is executed an appropriate library will be loaded to print formatted text. The Java model of lightweight applets travelling over the web relies on every browser supporting a standard interface to a large collection of known libraries. Even more dynamically, the notion of "mobile agents" [9] has programs hopping from place to place: and everywhere they land, interrogating local directories and using local services through known access methods.

The common theme here is using globally-known names to access local resources. But how do names become globally known, and what counts as local? Typically this is a very static and non-computational affair: user manuals list library calls, or services are offered at "well-known" addresses. In this paper we offer a calculus that begins an investigation of the interaction between the scope over which a name is known and the local areas in which it operates.

Our system is based around the π-calculus, which provides an established framework for reasoning about names and communication. Specifically, we work with a variant that is polyadic (channels carry tuples rather than single values [10]) and asynchronous (output actions always succeed [3]). To this we add a couple of novel extensions, which we motivate here with a brief example.

One of the original observations behind the π-calculus is that many issues associated with mobile code can be studied by looking simply at mobile names. So it is here, and our example is the operation of a service protocol that directs the internet. When a browser contacts a web server to fetch a page, or a person operates `finger` to list the users on a machine, both connect to a numbered "port" on the remote host: port 80 for the web page, port 79 for the finger listing. Of course, this only works if both sides agree; and a port number becomes "well-known" when enough systems do agree on it [8].

Under Unix, the file `/etc/services` holds a list mapping numbers to services[1]. There is also a further level of indirection: most machines run only a general meta-server `inetd`, the Internet daemon, which listens on all ports. When `inetd` receives a connection, it looks up the port in `/etc/services`, and then consults a second file which identifies the program to provide that service. The `inetd` starts the program and hands it a connection to the caller. A π-calculus model of the procedure might look like this.

$$\begin{array}{rl}
\text{Client} & \mathit{Carp} = \nu c.(\overline{\mathit{pike}}\langle \mathit{finger}, c\rangle \mid c(x).\overline{\mathit{print}}\langle x\rangle) \\
\text{Server} & \mathit{Pike} = !\mathit{pike}(s,r).\bar{s}\langle r\rangle \mid !\mathit{finger}(y).\bar{y}\langle \mathit{PikeUsers}\rangle \mid !\mathit{daytime}(z).\bar{z}\langle \mathit{PikeDate}\rangle \\
\text{System} & (\mathit{Carp} \mid \mathit{Pike})
\end{array}$$

Here a client machine Carp wishes to contact a server Pike with a finger request. The client has two components: the first transmits the request, the second prepares to print the result. Server Pike comprises three replicating processes: a general Internet daemon, a Finger daemon, and a time-of-day daemon. Channel pike is the internet address of the server machine, while the free names finger and $\mathit{daytime}$ represent well-known port numbers. In operation, Carp sends its request to Pike naming the finger service and a reply channel c. The Internet daemon on Pike handles this by retransmitting the contact c over the channel named finger. The Finger daemon collects this and passes information on $\mathit{PikeUsers}$ back to the waiting process at Carp.

This is a fair model, very much in the π-calculus style, but it has some shortcomings. Because the names finger and $\mathit{daytime}$ are visible everywhere, even when the Internet daemon on Pike has collected the request there is no protection against a Finger daemon on some different server actually handling it — perhaps even one back on the "client" Carp itself. If, however, we restrict the scope of finger to host Pike, then Carp cannot formulate the request because it must know the name of the service. We break this Catch-22 by extending the π-calculus with *local areas* and assigning each channel a *level of operation*. Our system is now

$$\mathit{net}[\mathit{host}[\mathit{Carp}] \mid \mathit{host}[\mathit{Pike}]]$$

which represents the fact that Carp and Pike are separate hosts residing on a single network. Each of the names in the system is identified as operating at net or host level:

$$c@\mathit{net}, \mathit{pike}@\mathit{net}, \mathit{finger}@\mathit{host}, \mathit{daytime}@\mathit{host}, \mathit{print}@\mathit{host}\ .$$

Thus communication on the finger, $\mathit{daytime}$ or print channels can span only a single host, while channels c and pike operate over the whole network. This is distinct from the *scope* of names, given by ν-binding; that determines where a name is known, not how it is used. In particular, the finger name in this example has a wide scope, but identical Finger daemons on different hosts will never interfere.

Introducing levels distinguishes between the different uses of concurrency and communication in a single system. For example, within a host there might be several applications, represented by areas at level app; or between host and net there could be a subnet level. Communication between two threads within an application will typically have a very different character to that between two hosts on a network, and a total order of levels allows us to express this concisely.

[1] This usually includes an abundant litter of port numbers which never became sufficiently "well-known".

Overview

In Section 2 we develop a formal description of this local area calculus. We give an operational semantics, and prove that it correctly limits communication to the correct local areas. We expand on the Internet service example, and present an illustration of agent-based programming in the calculus.

This operational semantics incorporates several dynamic checks to make sure that channels are used correctly. In Section 3 we propose a type system that captures this information statically. For example, our finger channel has type $(\mathit{string}@\mathit{net})@\mathit{host}$; this indicates that it is a host-level channel carrying values that themselves name net-level channels for communicating strings. We prove that "well-typed processes cannot go wrong", and deduce that we can omit most of the dynamic checks in the operational semantics. Section 4 of the paper concludes.

Related work

In earlier work, Vivas and Dam [15] investigate the behaviour of a *blocking* operator in the π-calculus: essentially CCS restriction $P \setminus a$, which blocks communication on the single channel a without binding it. Their specific observation is that this breaks Sangiorgi's reduction of the higher-order π-calculus to the first-order π-calculus. One contribution of our local areas and levels is to provide a systematic framework for this kind of static restriction.

There are a range of projects addressing *locations* in the π-calculus, with some similarities to our local areas. On the whole their aims are complementary: for example, Sangiorgi investigates non-interleaving semantics and causality using locations [11], and Amadio models local failure in distributed systems [1]. Neither of these limit the range of communication.

Systems proposed for mobile agents often use locations to curtail communication very strictly: agents may interact only with agents at the same location, and must move to talk to others. This is the case for Cardelli and Gordon's mobile ambients [4] and the system $D\pi$ of Hennessy and Riely [7, 6].

Approaches to distributed systems sometimes select particular disciplines for local and global communication. Sewell proposes a type system to distinguish between these, where channels either have universal reach or are restricted to a single local area [14]. The *Join calculus* requires all channels to be *located*: while anyone may transmit data, only a chosen process at a single site can receive it [5]. Sangiorgi's notion of *uniform receptiveness* is similar [12].

The use of types to structure the "expected" use of π-calculus channels is well-established: the survey paper by Sangiorgi [13] gives a good overview.

2 A calculus of local areas

2.1 Syntax

The calculus is built around two classes of identifiers:

$$\begin{array}{rr} \text{channels} & a, b, c, x, y, \mathit{query}, \mathit{reply}, \ldots \in \mathit{Chan} \\ \text{and levels} & \ell, m, \mathit{app}, \mathit{host}, \mathit{net}, \ldots \in \mathit{Level}. \end{array}$$

Channel names are drawn from a countably infinite supply, Chan. Syntactically, they behave exactly as in the π-calculus. Levels are rather more constrained: we assume prior choice of some finite and totally ordered set Level. The examples in this paper all use $\mathit{app} < \mathit{host} < \mathit{net}$. In the formal description of calculus, we take ℓ and m as metavariables for these levels.

Processes are given by the following syntax, based on the asynchronous polyadic π-calculus.

$$
\begin{array}{rclll}
\text{Process } P, Q & ::= & 0 & \text{inactive process} \\
& \mid & P \mid Q & \text{parallel composition} \\
& \mid & \bar{a}\langle\vec{b}\rangle & \text{output tuple} \\
& \mid & a(\vec{b}).P & \text{input} \\
& \mid & !a(\vec{b}).P & \text{replicated input} \\
& \mid & \ell[P] & \text{local area at level } \ell \\
& \mid & \nu a@\ell.P & \text{fresh channel } a \text{ at level } \ell
\end{array}
$$

Most of these are entirely standard. The last two constructions are particular to the local area calculus: thus $\ell[P]$ represents a process P running in a local area at level ℓ, and the name binding $\nu a@\ell.P$ specifies at which level channel a operates. Areas, like processes, are anonymous; this is in contrast to systems for locations, which are usually tagged with identifiers.

Definition 1. An *agent* is any process of the form $\ell[P]$: that is, a single enclosed area.

Channel names may be bound or free in any process. The binding prefixes are as usual the input prefixes $a(\vec{b})$, $!a(\vec{b})$ and restriction $\nu a@\ell$. We write $fn(P)$ for the set of free names of process P.

We identify process terms up to a *structural congruence* '$\equiv$', defined as the smallest congruence relation containing the following equations:

$$
\begin{array}{rcl}
a(\vec{b}).P \equiv a(\vec{c}).P\{\vec{c}/\vec{b}\} & P \mid 0 \equiv P & \\
!a(\vec{b}).P \equiv !a(\vec{c}).P\{\vec{c}/\vec{b}\} & P \mid Q \equiv Q \mid P & \\
\nu a@\ell.P \equiv \nu b@\ell.P\{b/a\} & (P \mid Q) \mid R \equiv P \mid (Q \mid R) & \\
\\
\nu a@\ell.0 \equiv 0 & \nu a@\ell.\nu b@m.P \equiv \nu b@m.\nu a@\ell.P & a \neq b \\
\ell[\nu a@m.P] \equiv \nu a@m.(\ell[P]) & (\nu a@\ell.P) \mid Q \equiv \nu a@\ell.(P \mid Q) & a \notin fn(Q)
\end{array}
$$

Here $P\{\vec{c}/\vec{b}\}$ stands for capture-avoiding simultaneous substitution. This congruence allows for alpha-conversion of bound names, algebraic properties of parallel composition '$\mid$', and flexible scope for channel names. This last point means that we can freely expand and contract the scope of any ν-binding, provided of course that it always includes every use of the name it binds.

2.2 Scope and areas

One point to note in the structural congruence above is the equation $\ell[\nu a@m.P] \equiv \nu a@m.(\ell[P])$, commuting name binding and area boundaries. A consequence of this is that the scope of a channel name, determined by ν-binding, is quite independent from the layout of areas, given by $\ell[-]$. Scope determines where a name is known, and this will change as a process evolves: areas determine how a name can be used, and these have a fixed structure.

For a process description to be meaningful, this fixed structure of nested areas must accord with the predetermined ordering of levels. For example, a net may contain a $host$, but not vice versa; similarly a $host$ cannot contain another $host$. Writing $<_1$ for the one-step relation in the total order of levels, we require that every nested area must be $<_1$-below the one above.

Definition 2. The *top-level agents* of a process P are all the subterms $m[Q]$ not themselves contained in any intermediate area $\ell[-]$.

For example, in the process $\bar{a}b \mid m[Q] \mid a(b).m[R]$ the top-level agents are $m[Q]$ and $m[R]$.

Definition 3. A process P is *well-formed at level* ℓ if for every top-level agent $m[Q]$, level $m <_1 \ell$, and Q is itself well-formed at level m, recursively. An agent $\ell[P]$ is *well-formed* if P is well-formed at level ℓ.

We can now make formal the distinction between the scope of a name and its area of operation. Consider some occurrence of a bound channel name a in a well-formed process P, as the subject of some action: $\bar{a}\langle - \rangle$, $a(-)$, or $!a(-)$. The *scope* of a is the enclosing ν-binding $\nu a@\ell.(-)$. The *local area* of this occurrence of a is the enclosing level ℓ area $\ell[-]$.

A single name may have several disjoint local areas within its scope. It is also possible for a name to occur outside any local area of the right level; in this case it can only be treated as data, not used for communication. We shall see how the operational semantics, and later the proposed type system, enforces this behaviour.

2.3 Operational semantics

We give the local area calculus a late-binding, small-step operational semantics. Much of this is standard from the regular π-calculus; the only refinement is to make sure that communication on any channel is contained within the appropriate local area.

Just what area is appropriate depends on the operating level of every channel, and we capture that information in a *level context* Λ: a finite partial map from channel names to levels. We write down level contexts using the $a@\ell$ notation from name binding. For example:

$$\Lambda = \{\mathit{pike}@\mathit{net}, \mathit{finger}@\mathit{host}, \mathit{daytime}@\mathit{host}, \mathit{print}@\mathit{host}\}$$

or, more simply:

$$\Lambda = \mathit{pike}@\mathit{net}; \mathit{finger}, \mathit{daytime}, \mathit{print}@\mathit{host}\ .$$

This declares that pike is a channel used for remote communication over the net, while finger, $\mathit{daytime}$, and print, even when globally known, are restricted to host-level interaction.

Given some level context Λ, we write $\Lambda \vdash_\ell P$ to denote that process P is well-formed at level ℓ with $\mathit{fn}(P) \subseteq \mathit{dom}(\Lambda)$. When the process is in fact a single agent we can omit the annotation on the turnstile and write this as $\Lambda \vdash \ell[P]$

Our operational semantics is given as an inductively defined relation on well-formed processes, indexed by their level ℓ and context Λ. Transitions take the form

$$\Lambda \vdash_\ell\ P \xrightarrow{\alpha} Q$$

where $\Lambda \vdash_\ell P$ and α is one of the following.

$$\begin{array}{rcll} \text{Transition } \alpha & ::= & \bar{a}\langle\vec{b}\rangle & \text{output} \\ & | & a(\vec{b}) & \text{input} \\ & | & \tau & \text{silent internal action} \end{array}$$

Transitions themselves have free and bound names, given by functions $\mathit{fn}(\alpha)$ and $\mathit{bn}(\alpha)$ respectively, where

$$\begin{aligned} \mathit{fn}(\bar{a}\langle\vec{b}\rangle) &= \{a\} \cup \vec{b} & \qquad \mathit{fn}(a(\vec{b})) &= \mathit{fn}(!a(\vec{b})) = \{a\} \\ \mathit{bn}(\bar{a}\langle\vec{b}\rangle) &= \emptyset = \mathit{fn}(\tau) = \mathit{bn}(\tau) & \qquad \mathit{bn}(a(\vec{b})) &= \mathit{bn}(!a(\vec{b})) = \vec{b}\ . \end{aligned}$$

Valid transitions are derived using the rules of Figure 1. We make several observations of these rules and the side-conditions attached to them.

OUT	$\Lambda \vdash_\ell \ \bar{a}\langle\vec{b}\rangle \xrightarrow{\bar{a}\langle\vec{b}\rangle} 0$	$\ell \leq \Lambda(a)$	
IN	$\Lambda \vdash_\ell \ a(\vec{b}).P \xrightarrow{a(\vec{b})} P$	$\ell \leq \Lambda(a)$	$\vec{b} \cap dom(\Lambda) = \emptyset$
IN!	$\Lambda \vdash_\ell \ !a(\vec{b}).P \xrightarrow{a(\vec{b})} P \mid !a(\vec{b}).P$	$\ell \leq \Lambda(a)$	$\vec{b} \cap dom(\Lambda) = \emptyset$
PAR	$\dfrac{\Lambda \vdash_\ell \ P \xrightarrow{\alpha} P'}{\Lambda \vdash_\ell \ P \mid Q \xrightarrow{\alpha} P' \mid Q}$		
COMM	$\dfrac{\Lambda \vdash_\ell \ P \xrightarrow{\bar{a}\langle\vec{c}\rangle} P' \quad \Lambda \vdash_\ell \ Q \xrightarrow{a(\vec{b})} Q'}{\Lambda \vdash_\ell \ P \mid Q \xrightarrow{\tau} P' \mid Q'\{\vec{c}/\vec{b}\}}$		
BIND	$\dfrac{\Lambda, a@m \vdash_\ell \ P \xrightarrow{\alpha} P'}{\Lambda \vdash_\ell \ \nu a@m.P \xrightarrow{\alpha} \nu a@m.P'}$	$a \notin fn(\alpha)$	
AREA	$\dfrac{\Lambda \vdash_\ell \ P \xrightarrow{\alpha} P'}{\Lambda \vdash_m \ \ell[P] \xrightarrow{\alpha} \ell[P']}$	if α is $\bar{a}\langle\vec{b}\rangle$ or $a(\vec{b})$ then $m \leq \Lambda(a)$	

Figure 1: Operational semantics for the local area calculus

- Active use of the structural congruence '$\equiv$' is essential to make full use of the rules: a process term may need to be rearranged or α-converted before it can make progress. For example, there is no symmetric form for the PAR rule (and no need for one).
- In order to apply the COMM rule it may be necessary to use structural congruence to expand the scope of communicated names to cover both sender and recipient.
- Late binding is enforced by the side-condition $\vec{b} \cap dom(\Lambda) = \emptyset$ on the input rules; this ensures that input names are chosen fresh, ready for substitution $Q\{\vec{c}/\vec{b}\}$ in the COMM rule. Again, we can always α-convert our processes to achieve this.

All of these comments are simple (and well-known) tidying of the standard π-calculus. The following are specific to local areas.

- The side-condition $\ell \leq \Lambda(a)$ on the OUT, IN and IN! rules prevent channels being read or written at too high a level. For example, trying to transmit on an application-level name in a $host$-level process. Any process that attempts this becomes stuck.
- The side-condition $m \leq \Lambda(a)$ on the AREA rule prevents communications escaping from their local area. Notice that necessarily $\ell <_1 m$ here, because of the requirement that the left-hand side $l[P]$ be well-formed at level m.

The following results show that this operational semantics does successfully capture the intuition behind areas and levels: areas retain their structure over transitions, and actions on a channel are never observed above their operating level.

Proposition 4. *If we can derive the transition $\Lambda \vdash_\ell P \xrightarrow{\alpha} Q$ then*

- *the process Q is well-formed at level ℓ with $fn(Q) \subseteq dom(\Lambda) \cup bn(\alpha)$;*
- *if the transition α is $\bar{a}\langle\vec{b}\rangle$ or $a(\vec{b})$ then $\ell \leq \Lambda(a)$.*

Proof. Structural induction on the derivation of $\Lambda \vdash_\ell P \xrightarrow{\alpha} Q$. ◻

$$
\begin{aligned}
\mathit{Carp} &= \mathit{host}[\nu c@net.(\overline{pike}\langle \mathit{finger}, c\rangle \mid c(x).\overline{print}\langle x\rangle)] \\
\mathit{Pike} &= \mathit{host}[\mathit{Inet} \mid \mathit{Finger} \mid \mathit{Daytime}]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Inet} &= !pike(s,r).\bar{s}\langle r\rangle \\
\mathit{Finger} &= !\mathit{finger}(y).\bar{y}\langle \mathit{PikeUsers}\rangle \\
\mathit{Daytime} &= !\mathit{daytime}(z).\bar{z}\langle \mathit{PikeDate}\rangle \\
\Lambda &= pike@net;\ \mathit{finger}, \mathit{daytime}, print@host
\end{aligned}
$$

$$\Lambda\ \vdash_{net}\ (\mathit{Carp} \mid \mathit{Pike})$$

Figure 2: Example of processes using local areas: an Internet server daemon

In particular if $\Lambda \vdash_\ell P \xrightarrow{\tau} P'$ then $\Lambda \vdash_\ell P'$ and P' might itself make further transitions.

Corollary 5. *If we can derive the sequence of transitions*

$$\Lambda \vdash_\ell\ P \xrightarrow{\tau} P_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_k \xrightarrow{\alpha} Q$$

then the same properties hold of Q as in Proposition 4

Proof. Repeated application of the preceding Proposition. □

2.4 Examples

In the introduction we met a small model of Internet service provision. Figure 2 formulates this system as a term of the local area calculus, with the following structure:

$$\Lambda\ \vdash_{net}\ (\mathit{Carp} \mid \mathit{Pike})\ .$$

Recall that the host Carp wishes to contact a Finger daemon running on host Pike, through a general Inet daemon. We can now apply our operational semantics to see this in action.

$$
\begin{array}{rl}
\Lambda\ \vdash_{net}\ (\mathit{Carp} \mid \mathit{Pike}) & \equiv\ (\mathit{host}[\nu c@net.(\overline{pike}\langle \mathit{finger}, c\rangle \mid c(x).\overline{print}\langle x\rangle)] \\
 & \qquad \mid \mathit{host}[\mathit{Inet} \mid \mathit{Finger} \mid \mathit{Daytime}]) \\
\text{extend scope of } \nu c@net & \equiv \nu c@net.\,(\mathit{host}[\overline{pike}\langle \mathit{finger}, c\rangle \mid c(x).\overline{print}\langle x\rangle] \\
 & \qquad\qquad \mid \mathit{host}[\mathit{Inet} \mid \mathit{Finger} \mid \mathit{Daytime}]) \\
\text{expand } \mathit{Inet} & \equiv \nu c@net.\,(\mathit{host}[\overline{pike}\langle \mathit{finger}, c\rangle \mid c(x).\overline{print}\langle x\rangle] \\
 & \qquad\qquad \mid \mathit{host}[!pike(s,r).\bar{s}\langle r\rangle \mid \mathit{Finger} \mid \mathit{Daytime}]) \\
\text{communication on } pike@net & \xrightarrow{\tau} \nu c@net.\,(\mathit{host}[c(x).\overline{print}\langle x\rangle] \\
 & \qquad\qquad \mid \mathit{host}[\overline{\mathit{finger}}\langle c\rangle \mid \mathit{Inet} \mid \mathit{Finger} \mid \mathit{Daytime}]) \\
\text{expand } \mathit{Finger} & \equiv \nu c@net.\,(\mathit{host}[c(x).\overline{print}\langle x\rangle] \\
 & \qquad\qquad \mid \mathit{host}[\overline{\mathit{finger}}\langle c\rangle \mid \mathit{Inet} \mid !\mathit{finger}(y).\bar{y}\langle \mathit{PikeUsers}\rangle \mid \mathit{Daytime}])
\end{array}
$$

$$
\begin{aligned}
\mathit{Main} &= \mathit{app}[\nu c@\mathit{host}.(\overline{\mathit{load}}\langle c\rangle \mid c(y).\mathit{link}(z).\overline{\mathit{print}}\langle y/z\rangle)]\\
\mathit{Probe} &= \mathit{app}[\nu c@\mathit{host}.(\overline{\mathit{load}}\langle c\rangle \mid c(w).\overline{\mathit{link}}\langle w\rangle)]\\
\mathit{Load} &= \mathit{app}[!\mathit{load}(x).\bar{x}\langle \mathit{LocalLoad}\rangle]\\
\Lambda &= \mathit{load}@\mathit{host}, \mathit{link}@\mathit{net}, \mathit{print}@\mathit{host}
\end{aligned}
$$

$$
\Lambda \vdash_{net} \mathit{host}[\mathit{Main} \mid \mathit{Load}] \mid \mathit{host}[\mathit{Probe} \mid \mathit{Load}]
$$

Figure 3: Example of processes using local areas: load management agents

$$
\begin{array}{rl}
\text{communication on } \mathit{finger}@\mathit{host} & \xrightarrow{\tau} \nu c@\mathit{net}.\,(\mathit{host}[c(x).\overline{\mathit{print}}\langle x\rangle] \\
& \qquad \mid \mathit{host}[\mathit{Inet} \mid \bar{c}\langle \mathit{PikeUsers}\rangle \mid \mathit{Finger} \mid \mathit{Daytime}]) \\
\text{communication on } c@\mathit{net} & \xrightarrow{\tau} \nu c@\mathit{net}.\,(\mathit{host}[\overline{\mathit{print}}\langle \mathit{PikeUsers}\rangle] \\
& \qquad \mid \mathit{host}[\mathit{Inet} \mid \mathit{Finger} \mid \mathit{Daytime}])
\end{array}
$$

After a sequence of internal communications at the net and $host$ level, the first host $Carp$ is ready to print the information $\mathit{PikeUsers}$, and host $Pike$ is restored to its original configuration.

Even this small example exhibits interesting scalability.

- $Pike$ can support multiple simultaneous finger or $\mathit{daytime}$ requests, because freshly-created channels like c provide private communication links.
- The system can support Finger and $\mathit{Daytime}$ servers on several hosts, with exactly the same agent code and protocol, because the finger and $\mathit{daytime}$ names are known globally but act locally.

Figure 3 presents another example, this time a very simple model of agent-style programming. Two hosts both carry a load-monitoring agent $Load$, which will report the current system load to any other agent on the same host. A $Main$ program on one host wants to compare the load on the two machines, and does this using a $Probe$ agent with which it shares a private channel $link$.

The processes execute with the following result:

$$
\Lambda \vdash_{net} \mathit{host}[\mathit{Main} \mid \mathit{Load}] \mid \mathit{host}[\mathit{Probe} \mid \mathit{Load}] \xrightarrow{\tau}{}^{*} \mathit{host}[\overline{\mathit{print}}\langle k\rangle \mid \mathit{Load}] \mid \mathit{host}[\mathit{Load}]
$$

where k is the numerical ratio of the load on the two hosts. Output $\overline{\mathit{print}}\langle k\rangle$ is the residue of the $Main$ agent, and the $Probe$ is discharged entirely.

One purpose of a system arranged like this is the simplifications it allows in the $Load$ agent:

- The two $Load$ agents are actually identical: no parameters, no distinguishing identifiers.
- Both are addressed using the same globally-known channel name $load$.
- They only require $host$-level communication capabilities, and can operate independently of firewalls or authentication.

These are the kind of advantages put forward for agent-based programming: the example shows how the local area calculus can represent them. Of course, they really take off when agents become mobile, but we can begin to evaluate their properties even in static systems like these.

3 Types for areas

The results at the end of Section 2.3 showed that local communications do remain local: an action on a channel is never observed above its level of operation. However, this relies on several side-conditions in the operational semantics of Figure 1, of the form $\ell \leq \Lambda(a)$, which are essentially runtime level checks. In this section we show that a suitable type system can provide enough static information to make these checks unnecessary.

The rule AREA of Figure 1 deals with propagating actions once they have happened, and its side-condition remains essential. The level tests accompanying OUT, IN and IN! are different: they check to see if an action should be attempted at all. For example, the process $a(\vec{b}).P$ should not proceed if it is above a's level of operation. Arguably, such processes should never be written: the reason it is not entirely trivial to eliminate them is that they can arise during execution as a result of substitution. For example, consider the following system:

$$a@host, b@app \vdash \ host[\ app[\bar{a}\langle b\rangle] \mid a(x).\bar{x}\langle\rangle\ ] \xrightarrow{\tau} host[\ \bar{b}\langle\rangle\ ] \not\longrightarrow$$

Here an application sends name b to a host-level process; this is fine as data, but the host then tries to transmit on it, and the process halts as b is only intended for communication within an application.

The type system we propose handles this by specifying not just the operating level of a channel, but also the levels of the channel names passed over it, and so on recursively.

3.1 Type system

Types are given by the following rather simple grammar.

$$\text{Type } \sigma \quad ::= \quad \vec{\sigma}@\ell$$

A type declaration of the form $a : \vec{\sigma}@\ell$ states that a is an ℓ-level channel carrying tuples of values whose types are given by the vector $\vec{\sigma}$. The base types are those with empty tuples: a channel of type $()@\ell$ is for synchronisation within an ℓ-area. In concrete examples we shall assume additional base datatypes like int or $string$ as convenient; these can be incorporated without difficulty into the formal presentation.

The only syntactic change required to introduce types into processes is at ν-binding:

$$\text{Process } P, Q \quad ::= \quad \cdots \mid \nu a{:}\sigma.P \quad \text{fresh channel } a \text{ of type } \sigma.$$

The other binding operation, input prefix $a(\vec{b}).P$, does not need any explicit type annotation, as the types of the $\vec{b}$ are fixed by the type of the channel a.

We also replace level contexts Λ with type contexts Γ, finite partial maps from channel names to types. With these alterations, Figure 4 presents the rules for deriving type assertions of the form $\Gamma \vdash_\ell P$, which states that process P is well-typed at level ℓ in context Γ.

To connect the typed calculus to the untyped one we use a notion of *erasure*. If $\sigma = \vec{\sigma}@\ell$ is a type, then its erasure $\lfloor\sigma\rfloor$ is just the level ℓ. If P is a typed process, then its erasure $\lfloor P\rfloor$ is the same process with all types replaced by their erased versions: in particular name binding $\nu a{:}\vec{\sigma}@\ell.Q$ is replaced by $\nu a@\ell.Q$. This throws away the detail of type information, but keeps the basic level declaration. Finally, erasing a type context Γ gives a level context $\lfloor\Gamma\rfloor$.

Proposition 6. *If P is a well-typed process at level ℓ in type context Γ, then its erasure $\lfloor P\rfloor$ is well-formed at level ℓ in the level context $\lfloor\Gamma\rfloor$.*

$$\Gamma \vdash_\ell P \quad \Longrightarrow \quad \lfloor\Gamma\rfloor \vdash_\ell \lfloor P\rfloor$$

Proof. Structural induction on the type derivation $\Gamma \vdash_\ell P$. ☐

$$\Gamma \vdash_\ell 0 \qquad \frac{\Gamma \vdash_\ell P \quad \Gamma \vdash_\ell Q}{\Gamma \vdash_\ell P \mid Q} \qquad \frac{\Gamma, \vec{b} : \vec{\sigma} \vdash_\ell P}{\Gamma \vdash_\ell a(\vec{b}).P} \quad \begin{array}{l}\Gamma(a) = \vec{\sigma}@m \\ \text{and } \ell \leq m\end{array}$$

$$\frac{\Gamma \vdash_\ell P}{\Gamma \vdash_m \ell[P]} \quad \ell <_1 m \qquad \frac{\Gamma, a : \sigma \vdash_\ell P}{\Gamma \vdash_\ell \nu a{:}\sigma.P} \qquad \frac{\Gamma, \vec{b} : \vec{\sigma} \vdash_\ell P}{\Gamma \vdash_\ell !a(\vec{b}).P} \quad \begin{array}{l}\Gamma(a) = \vec{\sigma}@m \\ \text{and } \ell \leq m\end{array}$$

$$\Gamma \vdash_\ell \bar{a}\langle\vec{b}\rangle \quad \Gamma(a) = \vec{\sigma}@m,\ \Gamma(\vec{b}) = \vec{\sigma} \text{ and } \ell \leq m$$

Figure 4: Types for processes in the local area calculus

3.2 Examples

We can give types to both of our examples from Section 2.4, which sensibly reflect their operation. First, for the internet daemon of Figure 2.

$$\begin{aligned} c &: \mathit{string}@\mathit{net} & \mathit{pike} &: (\mathit{service}, \mathit{response})@\mathit{net} \\ \mathit{finger} &: \mathit{service} \\ \mathit{daytime} &: \mathit{service} & \mathit{service} &= \mathit{response}@\mathit{host} \\ \mathit{print} &: \mathit{string}@\mathit{host} & \mathit{response} &= \mathit{string}@\mathit{net} \end{aligned}$$

The type $\mathit{service}$ for finger and $\mathit{daytime}$ expands to $(\mathit{string}@\mathit{net})@\mathit{host}$. This means that the channels can be used only for host-level communication, but the values carried will themselves be net-level names. The host-level communication is between Inet and Finger or $\mathit{Daytime}$; the net-level communication is the response sent out to the original enquirer, in this case machine Carp.

Channel pike has a net-level type that acts as a gateway to this, reading the name of a service and a channel where that service should send its reply.

The second example, of agents comparing the load on two hosts, has the following typing.

$$\begin{aligned} c &: \mathit{int}@\mathit{host} & \mathit{load} &: (\mathit{int}@\mathit{host})@\mathit{host} \\ \mathit{link} &: \mathit{int}@\mathit{net} & \mathit{print} &: \mathit{int}@\mathit{host} \end{aligned}$$

The most interesting type here is that for load: it captures the fact that not only must requests to load come from agents on the same host, but replies are also host-limited. This characterises a purely local procedure call used within a larger distributed environment.

3.3 Correctness

The operational semantics for well-typed processes replaces Λ with Γ in all the rules of Figure 1 and omits the side-condition $\ell \leq \Lambda(a)$ from OUT, IN, and IN! What we show in this section is that it is safe to make these omissions. The first step is to show that this operational semantics preserves types.

Proposition 7. *If P is a well-typed process at level ℓ in context Γ and we can derive the transition $\Gamma \vdash_\ell P \xrightarrow{\alpha} Q$, then Q is well typed:*

- *if $\alpha = \bar{a}\langle\vec{b}\rangle$ or τ then $\Gamma \vdash_\ell Q$*
- *if $\alpha = a(\vec{b})$ then $\Gamma, \vec{b} : \vec{\sigma} \vdash_\ell Q$ where $\Gamma(a) = \vec{\sigma}@m$*

Proof. Structural induction on the derivation of the transition $\Gamma \vdash_\ell P \xrightarrow{\alpha} Q$. □

As expected, there is an extremely tight connection between the behaviour of typed process terms and their untyped erasures.

Proposition 8. *Suppose that* $\Gamma \vdash_\ell P$ *is some well-typed process.*

(i) *If* $\Gamma \vdash_\ell P \xrightarrow{\alpha} P'$ *then* $\lfloor\Gamma\rfloor \vdash_\ell \lfloor P\rfloor \xrightarrow{\alpha} \lfloor P'\rfloor$.

(ii) *If* $\lfloor\Gamma\rfloor \vdash_\ell \lfloor P\rfloor \xrightarrow{\alpha} Q$ *for some untyped* Q*, then there is a typed process* P' *such that* $Q = \lfloor P'\rfloor$ *and* $\Gamma \vdash_\ell P \xrightarrow{\alpha} P'$.

Proof. Structural induction on the derivation of the transitions. □

Combining Propositions 7 and 8(i) with Corollary 5, we obtain a demonstration that "well-typed terms cannot go wrong".

Theorem 9. *For any well-typed process* P*, if we can derive the sequence of transitions*

$$\Gamma \vdash_\ell \; P \xrightarrow{\tau} \ldots \xrightarrow{\tau}\xrightarrow{\alpha} Q$$

where α *is* $\bar{a}\langle\vec{b}\rangle$ *or* $a(\vec{b})$ *and* $\Gamma(a) = \vec{\sigma}@m$ *then level* $\ell \leq m$.

This establishes that a well-typed process will never attempt to use a channel above its level of operation, without the need for explicit checks in the operational semantics.

4 Conclusion and further work

The local area calculus provides a reasonable setting to explore the use of names that are known globally but act locally. We have given an operational semantics and proved that it correctly captures this intuition. Illustrative examples include an internet service protocol and a pair of distributed agents. We propose a type system for channels in the calculus, and prove that it removes the need for some run-time locality checks.

A further area of application that we are exploring is layered network protocols: where each level communicates with the next on a local name, and only the outermost layer engages in actual long-distance communication. For example, TCP/IP is often used with an ordering of levels as $\mathit{application} < \mathit{transport} < \mathit{network} < \mathit{link}$.

We have an encoding of local areas into the pure π-calculus, using an explicit apparatus of controller processes to enforce level constraints: every communication is marshalled through routers, one for each local area. We have also built components of an implementation in MLj [2], which has provided useful insight on various design choices for the calculus.

With channels operating at distinct levels — network, host, application — the possibility arises of tuning observations of a process to inspect a single level of interest. We are working on a corresponding notion of bisimulation that filters out actions at some levels and focuses attention on others. Local areas give an opportunity for this to capture spatial information too.

The fact that communication in the calculus may be restricted to certain areas provides a form of security, though rather a weak one, and it may be possible to make a connection here to relevant versions of the π-calculus. For example, if encryption and decryption of messages are represented by communication on channels named by a private key, then carrying these out within a small "sandbox" area prevents eavesdropping. See Vivas and Dam [15] for an illustration of this using blocking.

The ordering of levels immediately suggests notions of subtyping on channels: for example, a net-level name might be used in place of a host-level name. However, even if this were desirable, it breaks down as soon as we pass around names themselves. The type constructor '@' in $\vec{\sigma}@\ell$ turns out to be invariant in its left argument, essentially because channels are used for both input and output.

One direction that is certainly worth pursuing is the step from static agents to properly mobile ones, and we are looking at various ways to incorporate these into the calculus while retaining the separate handling of scope and area. The π-calculus encoding mentioned above provides some hints: in classic π-calculus style, it can be subverted to emulate mobile areas by dynamically rewiring the attendant router processes.

References

[1] R. Amadio. On modelling mobility. *Theoretical Computer Science, to appear*. Available electonically from Elsevier Preprint.

[2] P. N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 1998.

[3] G. Boudol. Asynchrony and the π-calculus. Rapport de recherche 1702, INRIA, Sophia Antipolis, 1992.

[4] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structure: Proceedings of FoSSaCS '98*, Lecture Notes in Computer Science 1378, pages 140–155. Springer-Verlag, 1998.

[5] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Conference Record of POPL '96: 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996.

[6] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *Proceedings of HLCL '98: High-Level Concurrent Languages*, Electronic Notes in Theoretical Computer Science 16.3, pages 3–17. Elsevier, 1998.

[7] M. Hennessy and J. Riely. A typed language for distributed mobile processes. In *Conference Record of POPL '98: 25th ACM Symposium on Principles of Programming Languages*. ACM Press, 1998.

[8] IANA, the Internet Assigned Numbers Authority. Protocol numbers and assignment services: Port numbers. `http://www.iana.org/numbers.html#P`.

[9] D. B. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, March 1999.

[10] R. Milner. The polyadic π-calculus — a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[11] D. Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processe. *Theoretical Computer Science*, 155:39–83, 1996.

[12] D. Sangiorgi. The name discipline of receptiveness. In *Automata, Languages and Programming: Proceedings of the 24th International Colloquium ICALP 97*, Lecture Notes in Computer Science 1256. Springer-Verlag, 1997.

[13] D. Sangiorgi. Reasoning about concurrent systems using types. In *Foundations of Software Science and Computation Structure: Proceedings of FoSSaCS '99*, Lecture Notes in Computer Science 1578, pages 31–40. Springer-Verlag, March 1999.

[14] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Automata, Languages and Programming: Proceedings of the 25th International Colloquium ICALP 98*, Lecture Notes in Computer Science 1442. Springer-Verlag, 1998.

[15] J.-L. Vivas and M. Dam. From higher-order π-calculus to π-calculus in the presence of static operators. In *CONCUR '98: Concurrency Theory. Proceedings of the 9th International Conference*, Lecture Notes in Computer Science 1466. Springer-Verlag, 1998.