The Local Area π-Calculus

Tom Chothia

Doctor of Philosophy
Laboratory for Foundations of Computer Science
Division of Informatics
University of Edinburgh
2003

Abstract

All computers on the Internet are connected, but not all connections are equal. Hosts are grouped into islands of local communication. It is the agreed conventions and shared knowledge that connect these islands, just as much as the switches and wires that run between them.

The power and limitation of these conventions and shared knowledge and hence their effectiveness can be investigated by an appropriate calculus. In this thesis I describe a development of the π-calculus that is particularly well suited to express such systems. The process calculus, which I call the *local area π-calculus* or $la\pi$, 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 or 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* which distinguish, for example, between a single application, a machine, or a whole network. I present a semantics for this calculus that relies on several side-conditions which are essentially runtime level checks. I show that a suitable type system can provide enough static information to make most of these checks unnecessary.

I examine the descriptive power of the $la\pi$-calculus by comparing it to the π-calculus. I find that, perhaps surprisingly, local area communication can be encoded into the π-calculus with conditional matching. The encoding works by replacing communication inside an area with communication on a new channel created just for that area. This is analogous to replacing direct communication between two points with a system that broadcasts packets over a background ether. I show a form of operational correspondence between the behaviour of a process in $la\pi$ and its π-calculus translation.

One of my aims in developing this calculus is to provide a convenient and expressive framework with which to examine convention-laden, distributed systems. I offer evidence that the calculus has achieved this by way of an extended case study. I present a model of Internet communication based on Sockets and TCP over IP and then extend this system with Network Address Translation. I then

give a model of the File Transfer Protocol that uses TCP/IP to communicate between networks.

Traces of the model show that FTP, run in its normal mode, will fail when the client is using Network Address Translation, whereas, an alternative mode of FTP will succeed. Moreover a normal run of the model over NAT fails in the same way as the real life system would, demonstrating that the model can pick up this failure and correctly highlight the reasons behind it.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Tom Chothia*)

To my parents, and my academic parents.

Acknowledgements

I would like to thank Ian Stark. He has been an ideal supervisor. His comments and suggestions always showed his great intelligence. He often went out of his way to help me and always made himself available, for which I am eternally grateful.

My parents provided immeasurable support, both intellectual and emotional. They made helpful comments on my thesis and were always there for me when I needed them.

I am indebted to Stephen Gilmore for reading through an earlier draft of this thesis. His comments made it much more readable.

I would also like to thank my friends. They have made my time in Edinburgh enjoyable as well as worth while. I would especially like to thank Cath and Marco, without the constant diversions they provided I would either have finished my Ph.D. months earlier or more likely not finished it at all.

Think Globally, Act Locally.

Table of Contents

1 Introduction **13**

1.1 An Attempt at Local Communication in the π-calculus 14

1.2 Local Areas 16

1.3 Levels 16

1.4 Outline of Thesis 19

2 General Background and Related work **21**

2.1 Background on Process Calculi 21

2.1.1 The Calculus of Communicating Systems 21

2.1.2 The π-calculus 24

2.1.3 The Asynchronous π-calculus 26

2.2 Encoding Between Calculi 27

2.3 Related Work 32

2.3.1 The Join Calculus 33

2.3.2 The Seal Calculus 36

3 The Basic Local Area π-calculus **39**

3.1 Syntax 39

3.2 Scope and Areas 41

3.3 Operational Semantics 42

3.4 Examples 45

3.4.1 Internet Server Dæmon 45

3.4.2 Load Management System 48

3.4.3 Data Caching 49

3.5 Correctness of the Untyped Semantics 52
3.6 Non-Total Orders 57

4 Types for Local Areas 59
4.1 The Type System 60
4.1.1 Properties of Well Typed Processes 61
4.1.2 Linking Typed and Untyped Processes 67
4.2 Channel Types for the Examples 69
4.3 Correctness of the Typed Semantics 71
4.4 Sorts 79

5 Encoding the $la\pi$-calculus into the π-calculus 85
5.1 Encoding Local Areas 86
5.2 Example Encodings 90
5.2.1 Load Management System 90
5.2.2 The Inet Dæmon 95
5.3 Correctness of the Encoding 98
5.3.1 Evaluation Contexts 100
5.4 The Proof of Correctness 103
5.4.1 Proof that $la\pi$ Processes are Simulated by Their Encodings 103
5.4.2 Proof that $la\pi$ Processes Simulate Their Encodings 106

6 Modelling FTP over NAT 115
6.1 Program Syntax 116
6.1.1 Channel Coercion 117
6.2 The Internet Communication Model 118
6.2.1 Router Model 121
6.2.2 Linking IP Packets to Sockets 123
6.2.3 Network Address Translation 128
6.3 File Transfer Protocol 131
6.3.1 Introduction to FTP 131
6.4 Fitting it all Together 138
6.4.1 A Basic FTP Network 138

6.4.2 An FTP Network Client Side NAT 140
6.5 The Running System 141
6.5.1 Active FTP without NAT 142
6.5.2 Active FTP with NAT 150
6.5.3 Passive FTP with NAT 153
6.6 Extending the Model with Initialising Connections 158

7 Conclusion and Further Work 163
7.1 Conclusion 163
7.2 Further Work 165
7.2.1 The Extension of the $la\pi$-calculus with Mobility 165
7.2.2 Equivalence Methods for Local Areas 171
7.2.3 Model Checking an Extended Case Study 172
7.2.4 A Language Based on the $la\pi$-calculus 173
7.2.5 Levels and Areas in other Calculi 174

Bibliography 175

A Full Case Study Models 183
A.1 Basic FTP 183
A.2 FTP with Client Side NAT 184
A.3 Network Level 186
A.4 Transport Level 188
A.5 Application Level 191

Chapter 1

Introduction

This thesis aims to make a process calculus that is well suited to express the conventions, uniform local resources and other shared idiosyncrasies that hold large networks together. Real life systems have made use of the physical boundaries imposed by network hardware until ultimately the local conventions have become the salient features of these systems.

The calculus presented here, the *local area π-calculus*, extends the π-calculus with explicit areas and levels in such a way that a channel name may be used for communication within an area or 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* which distinguish, for example, between a single application, a machine, or a whole network.

This introduction first outlines exactly what I mean by "local resources and conventions". I then introduce the notions of *areas* and *levels*. Finally I give an outline of the thesis.

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 dynamically linked 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" [LO99] 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 the use of globally-known names to access local resources. But how do names become globally known, and what counts as local? Typically this is a static and non-computational affair: user manuals list library calls, or services are offered at "well-known" addresses. It is exactly this informal use of conventions that makes these systems so powerful but yet hard to model.

1.1 An Attempt at Local Communication in the π-calculus

The π-calculus has become the standard process calculus in which to examine distributed systems. Attempting to model local communication in this calculus will give us some insight into the problem.

Processes in the π-calculus may broadcast channel names over a given channel or listen for channel names over a given channel. Processes may also generate entirely new names. Any number of processes can be placed in parallel and, if two processes try to perform an input and an output on the same name, then the names being communicated are substituted for the names being listened for in the receiving process. Processes may also replicate themselves an arbitrary number of times. This allows recursion and so adds the computational power needed for more complex behaviours.

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. A full description of the π-calculus is given in Section 2.1.2.

An example of the kind of systems I am interested in is the operation of the service protocol that directs requests made to machines which serve a number of different applications, simultaneously. This application is known as the *Internet Dæmon Server*.

When a browser contacts a web server to fetch a page, or a person invokes `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 on the assignment of ports to services. A port number becomes "well-known" when enough systems agree on it [IAN].

Under Unix, the file `/etc/services` holds a list mapping numbers to services. This usually includes an abundant litter of port numbers which never became sufficiently "well-known". There is also a further level of indirection: most machines run only a general meta-server `inetd`, the Internet dæmon, 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:

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

This example is written in the syntax of the π-calculus that is presented in the next chapter. In English, this system consists of two processes, *Pike* and *Carp*, running in parallel. The 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. The server, *Pike*, comprises three replicating processes: a general Internet dæmon, a Finger dæmon and a time-of-day dæmon. Channel *pike* is the Internet address of the server machine, while the free names *finger* and *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 dæmon on *Pike* handles this by retransmitting the contact c over the channel named *finger*. The Finger dæmon collects this and passes information on *PikeUsers* back to the waiting process at *Carp*.

1.2 Local Areas

This is a fair model, very much in the style of the π-calculus, but it has some shortcomings. Because the names *finger* and *daytime* are visible everywhere, even when the Internet dæmon on *Pike* has collected the request there is no protection against a finger dæmon on some different server actually handling it — perhaps even one on the "client" *Carp* itself. So it is not only difficult to scale up the size of this model, it also misses many of the salient features of the Internet dæmon. 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. What we need is a restriction that stops communication on $finger$ but allows it be communicated on the channel *pike*, written below as [ ].

$$[Carp] \,|\, [Pike]$$

It is useful to nest these area restrictions inside each other. A large environment will have inside it a number of smaller sub-environments. These sub-environments will be able to access all the facilities available to the larger environment and more specialised local resources too. They may even include their own sub-environments. However, we are then left with the problem of working out which areas should block communications on which names and, at the same time, how to allow names to be sent between areas.

1.3 Levels

To state explicitly which names will be blocked by which areas, I introduce an ordering of levels and assign a level to each channel and area. A channel of a given level is then restricted within an area of that level. I also require nested areas to descend the order. This enforces a standard use of areas across a system and so allows names created in one place to be used in a uniform way in another part of the same system.

For the Inetd example given above and many of the other simple networking examples we use the ordering:

$$app < host < net$$

where *app* is short for the application level and contains the processes that model the applications. The host level will be used for uniform communication across a machine and similarly the net level will be used for communication across a whole network.

Now we have our levels, we can explicitly label the areas and channels of the Inetd example of Section 1.1. The system is now:

$$net[host[Carp] \mid host[Pike]]$$

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:

$$c@net, pike@net, finger@host, daytime@host, print@host$$

Thus communication on the *finger*, *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 dæmons on different hosts will never interfere.

It has become clear that when looking at different systems it is convenient to give the levels of the ordering names that are appropriate to whatever system we are looking at. Indeed, by looking at a few systems we can see that a multitude of orderings are already informally in use.

It is common practice to split the workings of distributed protocol into levels. One example is the *TCP/IP Reference Model* [CK74], that is used to model Internet communication. It defines 4 levels:

$$application < transport < network < physical$$

The idea of this model is that each level would deal with a different aspect of the communication process. The Application level deals with the user's programs, the Transport level processes the data into TCP packets for the Network level to send through the Physical level using IP protocol.

Communication is only allowed to take place between two adjacent levels. This means that a process can only address its surrounding layer and hence may use uniform names in all locations to pass data back and forth.

When a process uses one of these names to communicate with a different level it would automatically communicate with the appropriate handler for whatever kind of communication it was making. Communication on port numbers will be local inside any network area. Socket names inside any transport block and communication on IP addresses will be global, across the system.

The more complex OSI reference model [DZ83] is a generalisation of the TCP/IP model.

$$\begin{aligned} &application < presentation < session \\ &\qquad\qquad < transport < network < datalink < physical \end{aligned}$$

This stack aims to model all possible protocols, rather than just TCP/IP. It does this by filling out two holes in the TCP/IP model. It adds a *presentation* level, to parse data into the form expected by the applications, and a *data link* level, to control the physical connections between hosts.

Unix sets a number of variables at the start of each individual user session, such as the user's home directory, their access privileges, etc. This arrangement adds in another kind of environment. These session environments are clearly sub-environments of the computers they are running on, but each application will be run in a given session environment, so this adds a new level between the host machine and the individual applications:

$$application < user\ session < host < network$$

Levels such as *host* and *network* represent clearly distinguishable, physical boundaries. The *user session* level is a more abstract partition.

An abstract order of levels would be more useful, than a concrete set, when looking at processes that run on a single multi-user machine. In the $app < host < net$ ordering all the processes would be applications running inside a single host. A better ordering, to focus on the workings of the user sessions, would be:

$$user < session < system$$

with the processes representing the users' applications running at the *user* level, processes to handle the control of each session running at the *session* level and processes that control how the host runs and communicates with other hosts, running at the *system* level.

Introducing levels distinguishes between different uses of concurrency and communication in a single system and allows us to fine-tune the area restrictions in a uniform way. These simple extensions give us all the additional expressivity we need to begin exploring network conventions and local communication.

1.4 Outline of Thesis

The rest of this thesis is devoted to extending the π-calculus with local areas and levels and exploring the system this produces.

After going over some background material and related work in the next chapter I describe an untyped version of the local area π-calculus in Chapter 3. Having outlined its syntax and semantics, I present a number of examples that illustrate different aspects of the calculus. I also prove that the semantics given matches the idea of local areas outlined in this chapter.

However, this untyped calculus relies on several side-conditions in the operational semantics which are essentially runtime level checks. So, in Chapter 4, I show that a suitable type system can provide enough static information to make most of these checks unnecessary. Chapters 3 and 4 are an expanded version of a paper that appeared at the High-Level Concurrent Languages workshop: HLCL 00 [CS00].

In Chapter 5 I examine the expressive power of the $la\pi$-calculus by comparing it to the π-calculus. I find that, perhaps surprisingly, local area communication can be encoded into the π-calculus with conditional matching. The encoding works by replacing communication inside an area with communication on a new channel created just for that area. As well as sending the original data, an encoded output also sends the channel name. In a way, this is replacing direct communication between two points with a system that broadcasts packets over an Ethernet. There is a close match between the behaviour of a process in $la\pi$ and its π-calculus translation, that I show by way of an operational correspondence. An earlier version of this encoding was published in the proceedings of the EXPRESS '01 workshop [CS01].

In Chapter 6, I present an extended model in the $la\pi$-calculus. I provide a model of Internet communication based on sockets and TCP over IP [CK74] and then extend this system with Network Address Translation [EF94].

I then give a model of the File Transfer Protocol (FTP) and show how this protocol can sometimes fail when used with NAT and how the steps taken to correct these problems are successful to varying degrees.

My conclusions and suggestions for further work are presented in Chapter 7. One possible new direction would be to extend the $la\pi$-calculus with mobile areas. An equivalence method that encompasses local areas could also be useful. A logic or a bi-simulation that can be focused at particular areas or levels are possibilities. It might also be possible to apply the ideas of local areas at levels to extend other distributed process calculi with similar abilities.

Chapter 2

General Background and Related work

In the first section, I give details of the process calculus work on which this thesis builds. Specifically, I give details of CCS and the π-calculus. Next, I look at how the relative descriptive power of process calculi can be understood by encoding them into each other. Finally, I turn to work which is similar to my own. After giving a brief overview I focus on two calculi, the Join Calculus and the Seal Calculus, as particularly relevant.

2.1 Background on Process Calculi

2.1.1 The Calculus of Communicating Systems

Relatively speaking, CCS [Mil89] is one of the earlier theories for concurrent systems. At the time that this calculus was introduced the semantics of a program was often taken to be a function from one memory state to another. The idea was that, given a configuration of a computer's memory, running the program would have the same effect on that state as applying the function. However, this model breaks down as soon as more than one program has access to the memory at the same time.

The idea behind CCS was that concurrent processes could be modelled purely in terms of their inputs and outputs, rather than the effect that they had on the machine on which they ran. To this end the two basic actions of CCS are input and output. Processes could also branch and be placed in parallel. Names could be dynamically relabelled and communication on a name could become blocked at some point. Recursion added the additional computing power needed to handle complex or persistent systems. This led to the following syntax:

Process P, Q	$::=$	$\bar{a}.P$	output on name a
	$\mid$	$a.P$	input on name a
	$\mid$	0	inactive process
	$\mid$	$P \mid Q$	parallel composition
	$\mid$	$P + Q$	the summation of P and Q
	$\mid$	$fix(X = P)$	recursion
	$\mid$	$P \backslash a$	restriction of the name a
	$\mid$	$P[f]$	relabelling of the names by the function f

The semantics for these processes is given in Figure 2.1. These rules allow an input and an output on a channel to propagate through the system until they meet when the two actions are replaced with a τ action representing a communication. This τ action is then passed up and out of the process, indicating that a communication has taken place. The system then continues to reduce. The meaning of the recursion $fix(X = P)$ is the least fixed point of the equation $X = P$. Generally X will appear in P at least once.

Basic CCS only uses communication to release guards to other communication. However, the ability to pass data can easily be added by a simple encoding that makes a new name for every guard that one might wish to pass.

This encoding starts with a set, V, of all the channels used in a process and another D of all the pieces of data. An output is encoded as a new channel manufactured from the communication channel indexed with the data passed. The set of new channel names can be defined as $\{a_b | a \in V, b \in D\}$. This blows up the number of channels used in a process from $\#V$ to $\#V \times \#D$.

$$[\![\bar{a}(b).P]\!] \equiv \bar{a}_b.[\![P]\!]$$

IN/OUT_SEM	$\alpha.P \xrightarrow{\alpha} P$	
PAR_SEM1	$\dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$	
PAR_SEM2	$\dfrac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$	
COMM_SEM	$\dfrac{P \xrightarrow{\bar{\alpha}} P' \quad Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$	
SUM_SEM1	$\dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$	
SUM_SEM2	$\dfrac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$	
RES_SEM	$\dfrac{P \xrightarrow{\alpha} P'}{P \backslash L \xrightarrow{\alpha} P'}$	$\alpha \notin L$
REL_SEM	$\dfrac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'}$	
REC_SEM	$\dfrac{P\{fix(X = P)/X\} \xrightarrow{\alpha} P'}{fix(X = P) \xrightarrow{\alpha} P'}$	

Figure 2.1: Operational semantics for CCS

Inputs expand to listen for all possible names that might be sent by these proxy channels.

$$[\![a(x).P]\!] \equiv \sum_{v \in V} a_v.[\![P\{v/x\}]\!].$$

The other encoding rules proliferate these changes. Despite the blow up this causes in process size, this encoding tells us that the value passing form can be used as a shorthand with which to write the non-value passing processes. However, value passing CCS, as this flavour has become known, is usually taken as the standard.

CCS, along with a similar action based calculus CSP [Hoa85], started an avalanche of process calculi aimed at exploring different aspects of concurrent and distributed systems. Its great contribution was to formalize the idea of using input and output actions to look at the communications of a concurrent system to define its meaning and to blaze a path for future attempts.

2.1.2 The π-calculus

For all its power, CCS had one major drawback. It lacked the ability to evolve. There was no way for a process to pass data to another process it did not know about at the start. This makes the modelling of any system with arbitrarily changing connections, and especially any form of mobility, almost impossible.

The answer was simple: allow channels to carry channel names, not just data [Mil91, MPW92]. So a process can learn an entirely new connection. Restriction and relabeling were dropped in favour of the ability to create new names. Recursive definition was replaced with the replication operator $!P$, which would behave as an infinite number of copies of P running in parallel.

$$
\begin{array}{rcll}
\text{Process } P, Q & ::= & \bar{a}b.P & \text{output } b \text{ on name } a \\
& | & a(x).P & \text{input on name } a \\
& | & 0 & \text{inactive process} \\
& | & P \,|\, Q & \text{parallel composition} \\
& | & P + Q & \text{the summation of } P \text{ and } Q \\
& | & !P & \text{recursion} \\
& | & (\nu x)P & x \text{ in } P \text{ is a new name}
\end{array}
$$

The semantics is given in Figure 2.2. $P\{z/y\}$ denotes the process P with every occurrence of y replaced with a z, with any necessary rename to avoid variable capture. Structural Congruence, $\equiv$, equates processes that have only syntactic differences. It is defined as the smallest congruence relation containing the following equations:

$$
\begin{array}{rclrcl}
P \,|\, Q & \equiv & Q \,|\, P & (P \,|\, Q) \,|\, R & \equiv & P \,|(Q \,|\, R) \\
P \,|\, 0 & \equiv & P & a(b).P & \equiv & a(c).(P\{c/b\} \\
P + Q & \equiv & Q + P & P + 0 & \equiv & P \\
(\nu x)P & \equiv & (\nu y)(P\{y/x\}) & (\nu x)(\nu y)P & \equiv & (\nu y)(\nu x)P \\
(\nu x)0 & \equiv & 0 & & &
\end{array}
$$

It is standard practice to drop a trailing 0, so $\bar{a}b.0$, is written as just $\bar{a}b$. The ability to pass channels not only allows systems a much greater degree of flexibility, it also adds a form of mobile processes. In this abstract setting, it is possible to look at the location of a process as defined by the communication channels open to it. So by dropping one channel name and using another for communication a process can be thought of as having moved.

Milner presents a system for regulating the uses of channels [Mil91]. This system, known as *Sorts*, ensures that polyadic channels are always used to send the same kind of data and, more importantly, the same number of channels. This avoids the problematic situation where, for instance, one process is transmitting three names on a channel and another process is trying to receive five.

Each process has a set of *subject sorts*, which are effectively labels. Each name is then given a sort that describes its behaviour. *Object sorts* are sequences of

$$\frac{}{(\cdots + x(y).P) \,|\, (\cdots + \bar{x}z.Q) \to P\{z/y\} \,|\, Q}\,COMM$$

$$\frac{P \to P'}{P \,|\, Q \to P' \,|\, Q}\,PAR \qquad \frac{P \to P'}{(\nu x)P \to (\nu x)P'}\,RES$$

$$\frac{Q \equiv P \quad P \to P' \quad P' \equiv Q'}{Q \to Q'}\,STRUCT$$

Figure 2.2: π-calculus semantics

sorts and a *sorting* maps each subjective sort to an objective sort. The idea here is that each name is given a sort and the sorting will then map to the sorts of the data carried by that channel. So, to check that a process always passes the right number of values over each channel you just have to ensure that each input and output respects the sorting. An additional advantage of using a sorting function is that a channel can broadcast itself without having to resort to infinite types.

Sorts are just one example of the use of types to structure the "expected" use of π-calculus. Sangiorgi's notion of *uniform receptiveness* [San97] types the π-calculus to ensure that there will always be a process listening on a given channel and the listening processes will always treat the input in a uniform way. Dynamic *session* types [THK94, GH99] specify the exact types and order of names that will be sent and received over a channel. Abadi [Aba99] shows how types can be used to check cryptographic protocols. While Gordon and Jeffrey [GS01] show how to check authentication in security protocols. A survey paper by Sangiorgi [San99] gives a good overview of others.

2.1.3 The Asynchronous π-calculus

The π-calculus has become a very popular tool to explain distributed systems because it is so expressive. But in one particular aspect, it is too powerful.

The most common kind of output signal sent across a network is *asynchronous*, meaning that the sender will not know for sure when, or even if, the message has arrived. Communication in the π-calculus is synchronous: as soon as an output action happens, the process performs the output knows that it must have been received. This mismatch made the π-calculus an inappropriate tool for many asynchronous systems.

The deceptively simple solution, [ACS98, Bou92, HT91] was to stop the output action from prefixing any other process. The idea being that if the output process does not guard another process then no action is triggered by the output action and so it is undetectable by the sender. Summation was also removed because it allows a lack of an output action to be detected indirectly by placing it in a choice i.e. given $\bar{a} + \bar{b}$ you could tell that the a had not been transmitted if you received the b.

In the absence of running mobile processes the simple reduction of the π-calculus captures asynchronous communication perfectly. It quickly became a standard calculus on which others would build.

2.2 Encoding Between Calculi

Encodings between two calculi can give us an insight into the exact differences between them. For instance, the encoding of value passing CCS into plain CCS mentioned above tells us that in the setting of CCS the ability to pass value adds no real expressive power. It also shows us that we can think of value passing as the same as passing a guard or key to the value we wish to pass.

There is no single definition of what is or is not a correct *encoding* between calculi. Many of the calculi here are Turing complete and so it would be possible to code a Turing machine in one calculus and use it to interpret the other. This would be missing the point as it would not tell us anything about the relative expressive power of each system. What we seem to lack is a complexity theory of encodings.

Palamidessi suggests [Pal97] that the kind of encodings that tell us most about process calculi have two key properties. Firstly, they can be applied compositionally over the syntax of the calculus. This makes it impossible to use a brute force approach, such as making an interpreter for one calculus in the other, or computing every possible value and using the process you wish to encode as a lookup value.

The second property is that the encoding preserves some form of semantics from each system:

$$[\![\sigma(P)\ ]\!] = \sigma(\ [\![\ P\ ]\!]\ )$$

where σ is a sensible map that connects a process to a set of meanings. What exactly constitutes a sensible semantic map is left to the reader.

Palamidessi goes on to examine the difference between synchronous and asynchronous π-calculus. The latter can simulate both input and output guarded choice but Palamidessi shows that it is impossible to simulate a choice function that can handle both together i.e. mixed choice. As a corollary, she also shows that the π-calculus cannot be encoded in CCS.

This negative result is proved using complexity theory, to show that in a symmetric network it is impossible to solve the leader election problem in the asynchronous π-calculus whereas it is possible in the synchronous π-calculus. This tells us that the π-calculus really is more expressive than its asynchronous variant. This means that we must take special care when using any result from the π-calculus to talk about distributed systems that use asynchronous communication. In fact, a lot of work on process calculi has dropped synchronous communication altogether. Another possible conclusion is that mixed choice and truly synchronous communication would be hard to implement on a real life network. This has proved to be the case.

Sangiorgi [San93] defined the Higher Order π-calculus as a π-calculus variant in which static processes could be passed over channels in the same way as names.

$$
\begin{array}{lcll}
\text{Process } P, Q & ::= & \bar{a}(b).P & \text{output names on } a \\
& | & \bar{a}(P).P & \text{output processes on } a \\
& | & a(x).P & \text{input names on } a \\
& | & a(X).P & \text{input processes on } a \\
& | & 0 & \text{inactive process} \\
& | & P \mid Q & \text{parallel composition} \\
& | & P + Q & \text{the summation of } P \text{ and } Q \\
& | & (\nu x)P & x \text{ in } P \text{ is a new name} \\
& | & X\langle x\rangle & \text{a process place holder}
\end{array}
$$

Once a process is passed over a channel it is set running. There is no way that a running process could be *captured* and sent to a different location. So, any process that is to be moved in this way must be predefined when the process starts. A possible reduction in this system could be:

$$Q \mid \bar{a}(P) \mid a(X).X \rightarrow Q \mid P$$

This Higher Order message passing can be encoded back into the basic π-calculus by using a system of guards for each process that may be sent. Basically, each process that might be passed in the Higher Order setting is replaced by a process guarded with a token that can be used to set it running. The token is then passed instead of the process.

The key rules from the encoding are:

$$
\begin{array}{rcl}
[\![\bar{a}\langle Q\rangle.P]\!] & = & (\nu m)(\bar{a}\langle m\rangle.[\![P]\!] \mid !m(b).Q\langle b\rangle) \\
[\![a(X).P]\!] & = & a(x).[\![P]\!] \\
[\![X\langle a\rangle]\!] & = & \bar{x}\langle a\rangle
\end{array}
$$

where x is a name unique to X. So, the process given above encodes to:

$$
\begin{array}{rcl}
Q \mid (\nu m)(\bar{a}\langle m\rangle \mid !m().P\langle\rangle) \mid a(x).\bar{x}\langle\rangle & \rightarrow & Q \mid (\nu m)(!m().P\langle\rangle \mid \bar{m}\langle\rangle) \\
& \rightarrow & Q \mid (\nu m)(!m().P\langle\rangle \mid P)
\end{array}
$$

This rather neat encapsulation of mobility relies on the fact that processes can be run anywhere in the system with the same results. This uniform reduction environment is a direct result of the π-calculus approach to modelling location by connections only.

Vivas and Dam [VD98] investigate the more complicated case, where not all parts of a process offer exactly the same reduction environment. To achieve this they reintroduce the CCS restriction on a name, or *blocking* operator, into the π-calculus.

$$
\begin{array}{llll}
\text{Process } P, Q & ::= & \bar{a}b.P & \text{output names on } a \\
& | & \bar{a}P.P & \text{output processes on } a \\
& | & a(x).P & \text{input names on } a \\
& | & a(X).P & \text{input processes on } a \\
& | & 0 & \text{inactive process} \\
& | & P \,|\, Q & \text{parallel composition} \\
& | & P + Q & \text{the Summation of } P \text{ and } Q \\
& | & P \backslash a & a \text{ blocked in } P \\
& | & [a = b]P & \text{matching, reduce if } a = b \\
& | & [a \neq b]P & \text{mismatching, reduce if } a \neq b \\
& | & X\langle x \rangle & \text{an agent variable}
\end{array}
$$

The operator blocks communication on the single channel a without binding it. So, if a process is sent from inside a part of the process that is blocked on a name, to outside that blocking operator, it will be able to receive different inputs and outputs. This breaks Sangiorgi's reduction of the higher-order π-calculus to the first-order π-calculus.

Matching and mismatching rules provide equality tests on names:

$$
\frac{P \xrightarrow{\alpha} P'}{[a = b]P \xrightarrow{\alpha} P'}(x = y) \qquad \frac{P \xrightarrow{\alpha} P'}{[a \neq b]P \xrightarrow{\alpha} P'}(x \neq y)
$$

Vivas and Dam's encoding comes in two stages. The first removes the higher order processes from the calculus and the second removes the blocking operator. The second stage is the simpler of the two, so should be looked at first. The blocking operators are removed one at a time by replacing communication on a name inside a blocked area with communication on a new name created just for that blocked area. Of course, it is impossible to tell if a given variable will or will not be instantiated to a name that is blocked. So, every encoded action is given the option of using its original name or using the new name of each restriction. Match and mismatch guards then ensure that the right one reduces. This effectively doubles each term under a restriction and so leads to an exponential blow up in the size of the process.

Once the encoding meets a blocking operator, it carries two names. These are the name of the channel being blocked and the name of the channel that is used to replace it. Hence, there are two mutually recursive encodings, $[\![\_]\!]$ which removes all blocking operators and $[\![\_]\!]_{(w,z)}$ which replaces w for the blocked name z.

$$
\begin{array}{lcll}
[\![P\backslash z]\!] & = & \nu\ w.[\![P]\!]_{(w,z)} & w \notin names(P) \\
[\![\bar{x}y.P]\!]_{(w,z)} & = & [x = z]\bar{w}y.[\![P]\!]_{(w,z)} + [x \neq z]\bar{x}y.[\![P]\!]_{(w,z)} & \\
[\![x(y).P]\!]_{(w,z)} & = & [x = z]w(y').[\![P\{y'/y\}]\!]_{(w,z)} + [x \neq z]x(y').[\![P\{y'/y\}]\!]_{(w,z)} & \\
 & & & y' \notin fn(\nu\ y.P) \cup \{w, z\} \\
[\![\nu\ x.P]\!]_{(w,z)} & = & \nu\ x'.[\![P\{x'/x\}]\!]_{(w,z)} & y' \notin fn(\nu\ x.P) \cup \{w, z\} \\
[\![P\backslash x]\!]_{(w,z)} & = & [\![[\![P\backslash x]\!]]\!]_{(w,z)} &
\end{array}
$$

Following Sangiorgi's lead, a server process is spun off for each process that may be sent. The sending of a process is then replaced by sending a token to that process. Here Sangiorgi's encoding breaks down because in the presence of operators processes must reduce in the right place, not just at the right time. Vivas and Dam's solution is much more ingenious.

When the token is used, the spawning process actually sends a syntactic copy of the process being sent. A given name represents which piece of syntax is being sent, c for a parallel composition, i for an input etc. This is followed by the names needed to construct the process at the new location.

For instance:

$$send_v x(y).P = \bar{v}i.\bar{v}x.v(y).send_v(P)$$
or
$$send_v(P_1 \mid P_2) = \bar{v}c.\bar{v}v_1.\bar{v}v_2.send_{v1}(P_1) \mid send_{v1}(P_1)$$

A receiving process then rebuilds the mobile process in the right place. Effectively, this achieves mobility by sending the source code of a process that is then compiled and run at the new location.

Nestmann and Pierce give a good overview of what makes for a good encoding in their work on deconstructing choice [NP96]. Among many examples, Fournet and others have implemented Mobile Ambients in the JoCaml languages [FLS00]; this translates one notion of distributed areas into another. Their focus is on providing a basis for an implementation of Mobile Ambients; so, much attention is paid to making it run efficiently. Sangiorgi describes in great detail a rather different encoding of locations in order to express non-interleaving semantics [San96].

2.3 Related Work

There are a range of projects addressing *locations* in the π-calculus, with some similarities to local areas. On the whole their aims are complementary: for example, Sangiorgi investigates non-interleaving semantics and causality using locations [San96], and Amadio models local failure in distributed systems [Ama99]. 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 [CG98] where *Ambients* are the basic building blocks. Ambients may make subjective moves to their parent Ambient or to a child Ambient or may dissolve their boundaries and merge with the Ambient above. So the movement of Ambients becomes the basic semantic step, rather than a communication as in the π-calculus. This also means that a movement to a remote location must be negotiated step by step. Boxed Ambients [BCC01] replaces the ability to merge with another Ambient with communications that may pass to a parent or child location.

$D\pi$ of Hennessy and Riely [HR98b, HR98a] extends the π-calculus with locations and processes that can migrate between them. A type system is used to prevent agents accessing resources before they are granted the capability to do so. Also with a security theme, Control Flow Analysis, [HVY00, BDNN98] prevents communication on high security names, that might be picked up, even indirectly, on low security level names. Similarly, the Box-π of Sewell and Vitek [SV00, SV99] examines causality dependencies with explicit locations.

Cardelli, Ghelli and Gordon take a different approach to limiting communication with their notion of name *groups* [CGG00]. Ingeniously, introducing these into the type system allows one to check statically that a process never passes out certain names. In our system, by contrast, names may be passed anywhere — only their action is limited.

The word *local* is often used with a different meaning to the way in which I use it here. Sewell [Sew98] uses local to describe a channel that is restricted to a single space and Merro [Mer00] uses local to mean the ability to input names is restricted to a single space. This is a very heavy-handed way of forcing communications to stay in a given space and rules out many of the aspects I am most interested in, such as use of local resources on predefined names handling connections on ports. Sewell's approach is to type channels as *global* or *local*. Global channels can be used to communicate between areas whereas local channels are restricted to a predefined location. This approach is a two-level version of the $la\pi$-calculus but, although a local name may be globally known, any process that attempts to use it outside its predefined area will not type check.

2.3.1 The Join Calculus

The Join Calculus [FG96] allows one to state exactly where communication on each channel is handled. Instead of matching input and output actions, an output action is matched with a *definition* which describes what should happen when an output on a given channel is observed. These definitions are themselves part of the process.

$$
\begin{array}{ll}
\text{Names:} & x \ v \\
\text{Processes:} & P ::= x\langle v_1 ... v_n\rangle \ | \ def\ D\ in\ P \ | \ (P|P) \\
\text{Join Patterns:} & J ::= x\langle v_1 ... v_n\rangle \ | \ (J|J) \\
\text{Definitions:} & D ::= J \triangleright P \ | \ D \wedge D
\end{array}
$$

This is an extension of the more basic chemical framework put forward by Berry and Boudol [BB92].

Reductions take place in the presences of an environment which is used to store the current reduction definitions. The process P running with definitions D is written as $D \vdash P$. The two key semantic rules allow definitions to be put into the environment and for any pattern of outputs to be replaced by the appropriate process.

$$
\begin{array}{lrcl}
\text{str-def} & \vdash def\ D\ in\ P & \rightleftharpoons & D\sigma \ \vdash \ P\sigma \\
\text{red} & J \triangleright P \ \vdash \quad J\sigma & \rightarrow & J \triangleright P \ \vdash \ P\sigma
\end{array}
$$

where σ is an appropriate substitution that will match the output data and avoid variable capture. For example, passing on a message from x to y could be done with the following definition:

$$def\ x\langle u\rangle \triangleright y\langle u\rangle in\ P$$

The Distributed Join Calculus [FGL+96], extends the basic Join Calculus with explicit locations and models of location failure and migration. Locations are separated with a $||$ symbol and each turnstile is tagged with a location name.

$$
\begin{array}{lcll}
Term & ::= & D \vdash_a P & \text{process } P \text{ running with definitions } D \text{ at } a \\
 & | & Term \ || \ Term & \text{multiple locations in parallel}
\end{array}
$$

Syntactically, there is only one level of locations but a nested tree structure is built up using compound names.

For instance, a location name h with two sub-locations a and b would be written as:

$$D_h \vdash_h P_h \;\;||\;\; D_a \vdash_{ha} P_a \;\;||\;\; D_b \vdash_{hb} P_b$$

A locations construct is added to the definitions to allow locations to migrate and new locations to be spawned. Migration is triggered by a *go* command. Failure is triggered by a *halt* action that marks a location construct to stop it from reducing.

This system requires all channels to be located: while anyone may transmit data, only a chosen process at a single site can receive. Remote communication is handled by moving a message to the unique location where that message is defined.

$$comm \quad D_a \vdash_a x\langle\vec{v}\rangle \;||\; D_b \vdash_b P \longrightarrow D_a \vdash_a 0 \;||\; D_b \vdash_b x\langle\vec{v}\rangle \,|\, P \qquad (\ x \text{ defined in } D_b)$$

This system is excellent when each location is offering unique resources to a globally system but is inappropriate when each location can offer a different resource on the same channel. Schmitt addresses this problem [Sch02] by adding dynamically bound channels to the Distributed Join Calculus.

Schmitt's aim is very similar to my own. One of the stated motivations of his work is to be able to reason about local libraries. Channels are split into two distinct groups, *static channels*, that behave as the channels in the Distributed Join Calculus and *dynamic channels* that may be defined at a number of different locations and automatically moved to and matched with the nearest definition.

So a *print* channel, for example, could be defined at locations a and b. The process $go\ a\theta; print\langle\text{“}Hello''\rangle; go\ b\psi; print\langle\text{“}World''\rangle$ would first move to a sub-location of location a and then send “Hello” to the print function of the location a. It would then move to a sub-location of b and send “World” to b's print function.

Each process term carries around a set of imported dynamic names, a set of locally defined dynamic names, a lookup function that maps dynamic names to their locations, and the name of the current location. The reduction of a message now becomes a three-phase process. First, the lookup function is used to find the location of the nearest definition. The message is then tagged with this location. In the second step, a message that is tagged with a location is moved to that location. Finally, the message is reduced using the local definitions.

One of the principal differences between dynamic channels in the Join Calculus and the local areas presented here is that local areas give you a uniform way to declare how channels will be used at a remote location. As the Join Calculus does not use a uniform system, the semantics becomes somewhat heavy handed due to the need to maintain up to date lookup functions of all dynamic channels.

Schmitt also presents a type system for dynamic channels that ensures there is always a definition ready to reduce any dynamic message. This typing of channels to be "Safe" works well in a situation where, for instance, you wish to ensure that all hosts are able to offer all functions of a local library. It is not so helpful if you want to be able to consider systems where a process may stop listening on a name, say, to model a router that intermittently fails. With five different kinds of typing judgment and seventeen rules, the type system is quite complex.

The Join Calculus has been extended into a language: JoCaml [CF99]. This language provides a useful test bed to see how the ideas developed in the Join Calculus work out in real life.

2.3.2 The Seal Calculus

The Seal Calculus [CV99, CGN01] introduces a *seal* as an area of computation. A seal is a named group of processes and possibly other seals. Communication is limited with each seal except when an action is explicitly tagged as communicating with its parent "$\uparrow$" or a named child Seal.

$$\begin{array}{rcll}
\text{Process } P, Q & ::= & \alpha.P & \text{action} \\
& | & 0 & \text{inactive process} \\
& | & P \,|\, Q & \text{parallel composition} \\
& | & (\nu\ x).P & \text{restriction} \\
& | & !P & \text{replication} \\
& | & x[P] & \text{seal} \\
& | & x[X] & \text{abstract seal}
\end{array}$$

$$\begin{array}{rcll}
\text{Actions } \alpha & ::= & \bar{x}^n(\vec{y}) & \text{name output} \\
& | & x^n(\lambda\vec{y}) & \text{name input} \\
& | & \bar{x}^n\{\vec{y}\} & \text{seal send} \\
& | & x^n\{\vec{y}\} & \text{seal receive} \\
& | & open_n x & \text{portal open}
\end{array}$$

$$\begin{array}{rcll}
\text{Locations } n & ::= & x & \text{child seal names } x \\
& | & \uparrow & \text{the parent seal} \\
& | & * & \text{local.}
\end{array}$$

Portals are a security mechanism used to limit communication between seals. To perform a non-local communication $\bar{x}^n(\vec{y})$ a portal must be explicitly opened by the action $open_n x$. Seals can be objectively moved to a parent or child location by input and output actions distinguished with curly brackets.

In this model of communication, just as in Mobile Ambients, communication is very fine grained. To communicate with a remote location it is necessary to plot a course through an arbitrary number of named Seals. But whereas Mobile Ambients takes mobility to be the basic semantic step, the Seal calculus retains π style communications.

The semantics given in [CV99] takes advantage of this small step style and uses a reduction rule semantics, as opposed to a propagating action semantics used by CCS.

The Seal Calculus does not allow any level of abstraction when it comes to using channels. If we wish to use a channel name we must know and state exactly how it is going to be used by tagging the communication with a location. This puts a considerable burden on the user. It is not enough to give a process a channel and then to use that channel to access resources. It is necessary to know if that channel is to be used to communicate with a parent process or a child process, and if it is a child process, it must know the name of that process. It might also be necessary to open a portal to allow communication.

While Sewell's global/local calculus [Sew98] allowed a channel to be uniformly restricted across any system but with only one level of areas, seals allow for many levels of nested environments but provides no way uniformly to define the treatment of a name.

Chapter 3

The Basic Local Area π-calculus

This chapter describes the untyped local area π-calculus. After outlining its syntax and semantics I present in Section 3.4 a number of examples that illustrate different aspects of the calculus. These examples will be used to explore other features of the calculus in later chapters. In the final section I prove that, in keeping with the view of local areas outlined in the introduction, communication on a channel can never transgress that channel's predefined area.

3.1 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: they assume a prior choice of some ordered set *Level*. The examples in this chapter all use $\mathit{app} < \mathit{host} < \mathit{net}$. In the formal description of the calculus, take ℓ and m as metavariables for these levels.

Processes are given by the following syntax, based on the asynchronous polyadic

π-calculus [ACS98, Bou92, HT91].

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

Most of these are entirely standard. The last two constructions are particular to the local area π-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 contrasts with systems for locations, which are usually tagged with identifiers [CG98, FGL+96]. To ease the example processes along, output tuples may contain strings and integers as well as names.

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 $\nu a@\ell$. The set of free names of process P is written as $fn(P)$.

Definition 2. Process terms are identified up to a *structural congruence* '$\equiv$', defined as the smallest congruence relation containing the following equations:

$$
\begin{array}{rcl}
a(\vec{b}).P \equiv a(\vec{c}).P\{\vec{c}/\vec{b}\} & \qquad & P \mid 0 \equiv P \\
!a(\vec{b}).P \equiv !a(\vec{c}).P\{\vec{c}/\vec{b}\} & & P \mid Q \equiv Q \mid P \\
\nu a@\ell.P \equiv \nu b@\ell.P\{b/a\} & & (P \mid Q) \mid R \equiv P \mid (Q \mid R)
\end{array}
$$

$$
\begin{array}{rcll}
\nu a@\ell.0 \equiv 0 & \qquad & \nu a@\ell.\nu b@m.P \equiv \nu b@m.\nu a@\ell.P & \quad a \neq b \\
\ell[\nu a@m.P] \equiv \nu a@m.\ell[P] & & (\nu a@\ell.P) \mid Q \equiv \nu a@\ell.(P \mid Q) & \quad a \notin fn(Q)
\end{array}
$$

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 '$|$', 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.

3.2 Scope and Areas

One point to note in the structural congruence above is the equation $\ell[\nu a@m.P] \equiv \nu a@m.(\ell[P])$, which commutes name binding and area boundaries. A consequence of this is that the scope of a channel name, determined by ν-binding, is quite independent of 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*. Write $<_1$ for the one-step relation in the order of levels. Every nested area must be $<_1$-below the one above.

Definition 3. The *top-level agents* of a process P are all the sub-terms $m[Q]$ not themselves contained in any intermediate area $\ell[-]$. Formally, define a set of agents, $tla(P)$ by induction on the syntax of P.

$$\begin{aligned}
tla(\ 0\ ) &\stackrel{def}{=} \{\} & tla(\ P\,|\,Q\ ) &\stackrel{def}{=} tla(\ P\ ) \cup tla(\ Q\ ) \\
tla(\ \bar{a}\langle\vec{b}\rangle\ ) &\stackrel{def}{=} \{\} & tla(\ a(\vec{b}).P\ ) &\stackrel{def}{=} tla(\ P\ ) \\
tla(\ !a(\vec{b}).P\ ) &\stackrel{def}{=} tla(\ P\ ) & tla(\ \ell[P]\ ) &\stackrel{def}{=} \{\ell[P]\} \\
tla(\ \nu a@\ell.P\ ) &\stackrel{def}{=} tla(\ P\ )
\end{aligned}$$

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

Definition 4. A process P is *well-formed at level* ℓ if for every top-level agent $m[Q]$ in P it is true that $m <_1 \ell$, 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.

It is now possible to formalize the distinction between the scope of a name and its area of operation.

Definition 5. Given some occurrence of a ν-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 $\nu a@\ell.(-)$.

Definition 6. Given some occurrence of a bound channel name a of level ℓ, in a well-formed process P, as the subject of some action $\bar{a}\langle - \rangle$, $a(-)$, or $!a(-)$, 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. The operational semantics, and later the proposed type system, enforces this behaviour.

3.3 Operational Semantics

I give the local area π-calculus a late-binding, small-step operational semantics. Much of this is standard from the regular π-calculus [MPW92]; the only refinement here 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 I capture that information in a *level environment* Λ: a finite partial map from channel names to levels. I write level environments using the $a@\ell$ notation from name binding. For example:

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

or, more simply:

$$\Lambda = pike@net; finger, daytime, print@host$$

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

Definition 7. Given some level environment Λ, write $\Lambda \vdash_\ell P$ to denote that process P is well-formed at level ℓ with $fn(P) \subseteq dom(\Lambda)$.

When the process is in fact a single agent the annotation on the turnstile can be omitted and written as $\Lambda \vdash \ell[P]$

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

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

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

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

where $(\nu\vec{d}@\vec{n})$ is the set of bound names in the output action. When this is empty I will write the action as: $\bar{a}\langle\vec{b}\rangle$.

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

$$\begin{array}{ll} fn((\nu\vec{d}@\vec{n})\bar{a}\langle\vec{b}\rangle) = \{a\} \cup \vec{b}\backslash\vec{d} & fn(a(\vec{b})) = \{a\} \\ bn((\nu\vec{d}@\vec{n})\bar{a}\langle\vec{b}\rangle) = \vec{d} & bn(a(\vec{b})) = \vec{b}\,. \end{array}$$

$$fn(\tau) = bn(\tau) = \emptyset$$

Valid transitions are derived using the rules of Figure 3.1. I make the following observations of these rules and the side-conditions attached to them.

- Active use of the structural congruence '$\equiv$' is essential to make full use of the rules: a process term may need to be rearranged before it can make progress. For example, there is no symmetric form for the PAR_SEM rule (and no need for one).

- In order to communicate a bound name to a process outside its original binding it is either necessary to use structural congruence to expand the scope of communicated names to cover both sender and recipient or to use the OPEN_SEM rule to add the binder into the output action.

OUT_SEM	$\Lambda \vdash_\ell \ \bar{a}\langle\vec{b}\rangle \xrightarrow{\bar{a}\langle\vec{b}\rangle} 0$	$\ell \leq \Lambda(a)$
IN_SEM	$\Lambda \vdash_\ell \ a(\vec{b}).P \xrightarrow{a(\vec{b})} P$	$\ell \leq \Lambda(a) \quad \vec{b} \cap dom(\Lambda) = \emptyset$
IN!_SEM	$\Lambda \vdash_\ell \ !a(\vec{b}).P \xrightarrow{a(\vec{b})} P \,\vert\, !a(\vec{b}).P$	$\ell \leq \Lambda(a) \quad \vec{b} \cap dom(\Lambda) = \emptyset$
PAR_SEM	$\dfrac{\Lambda \vdash_\ell \ P \xrightarrow{\alpha} P'}{\Lambda \vdash_\ell \ P \,\vert\, Q \xrightarrow{\alpha} P' \,\vert\, Q}$	
COMM_SEM	$\dfrac{\Lambda \vdash_\ell \ P \xrightarrow{(\nu\vec{d}@\vec{n})\bar{a}\langle\vec{c}\rangle} P' \quad \Lambda \vdash_\ell \ Q \xrightarrow{a(\vec{b})} Q'}{\Lambda \vdash_\ell \ P \,\vert\, Q \xrightarrow{\tau} \nu\vec{d}@\vec{n}.P' \,\vert\, Q'\{\vec{c}/\vec{b}\}}$	
BIND_SEM	$\dfrac{\Lambda; a@m \vdash_\ell \ P \xrightarrow{\alpha} P'}{\Lambda \vdash_\ell \ \nu a@m.P \xrightarrow{\alpha} \nu a@m.P'}$	$a \notin fn(\alpha)$
OPEN_SEM	$\dfrac{\Lambda; c@m \vdash_\ell \ P \xrightarrow{(\nu\vec{d}@\vec{n})\bar{a}\langle\vec{b}\rangle} P'}{\Lambda \vdash_\ell \ \nu c@m.P \xrightarrow{(\nu\vec{d}@\vec{n}, c@m)\bar{a}\langle\vec{b}\rangle} P'}$	$c \neq a \quad c \in \vec{b} \setminus dom(\Lambda)$
AREA_SEM	$\dfrac{\Lambda \vdash_\ell \ P \xrightarrow{\alpha} P'}{\Lambda \vdash_m \ \ell[P] \xrightarrow{\alpha} \ell[P']}$	if α is $\bar{a}\langle\vec{b}\rangle$ or $a(\vec{b})$ then $m \leq \Lambda(a)$
STRUC_SEM	$\dfrac{P' \equiv P \quad \Lambda \vdash_\ell P \xrightarrow{\alpha} Q \quad Q \equiv Q'}{\Lambda \vdash_\ell \ P' \xrightarrow{\alpha} Q'}$	

Figure 3.1: Operational semantics for the local area π-calculus

- 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 the substitution $Q\{\vec{c}/\vec{b}\}$ in the COMM rule.

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_SEM, IN_SEM and IN!_SEM rules prevent channels being read or written at too high a level, for example, trying to transmit on an *app*lication-level name in a *host*-level process. Any process that attempts this becomes stuck.

- The side-condition $m \leq \Lambda(a)$ on the AREA_SEM 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.

3.4 Examples

In this section, I present three examples that illustrate the use of the local area π-calculus. In the first one, I fill out the Internet server dæmon from the introduction. The second example presents a simple, agent-style load management system. The third illustrates how a cache can be added to a system by replacing a global call for a resource with a local call to a copy of the same data.

3.4.1 Internet Server Dæmon

In the introduction we met a small model of Internet service provision. Figure 3.2 formulates this system as a term of the local area calculus. Recall that the host *Carp* wishes to contact a *Finger* dæmon running on host *Pike*, through a general

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

$$Inet = !pike(s,r).\bar{s}\langle r\rangle$$
$$Finger = !finger(y).\bar{y}\langle \text{“}PikeUsers\text{”}\rangle$$
$$Daytime = !daytime(z).\bar{z}\langle \text{“}PikeDate\text{”}\rangle$$
$$\Lambda = pike@net;\ finger, daytime, print@host$$
$$\Lambda \vdash_{net} Carp \mid Pike$$

Figure 3.2: Example of processes using local areas: an Internet server dæmon

Inet dæmon. We can now apply the operational semantics to see this in action.

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

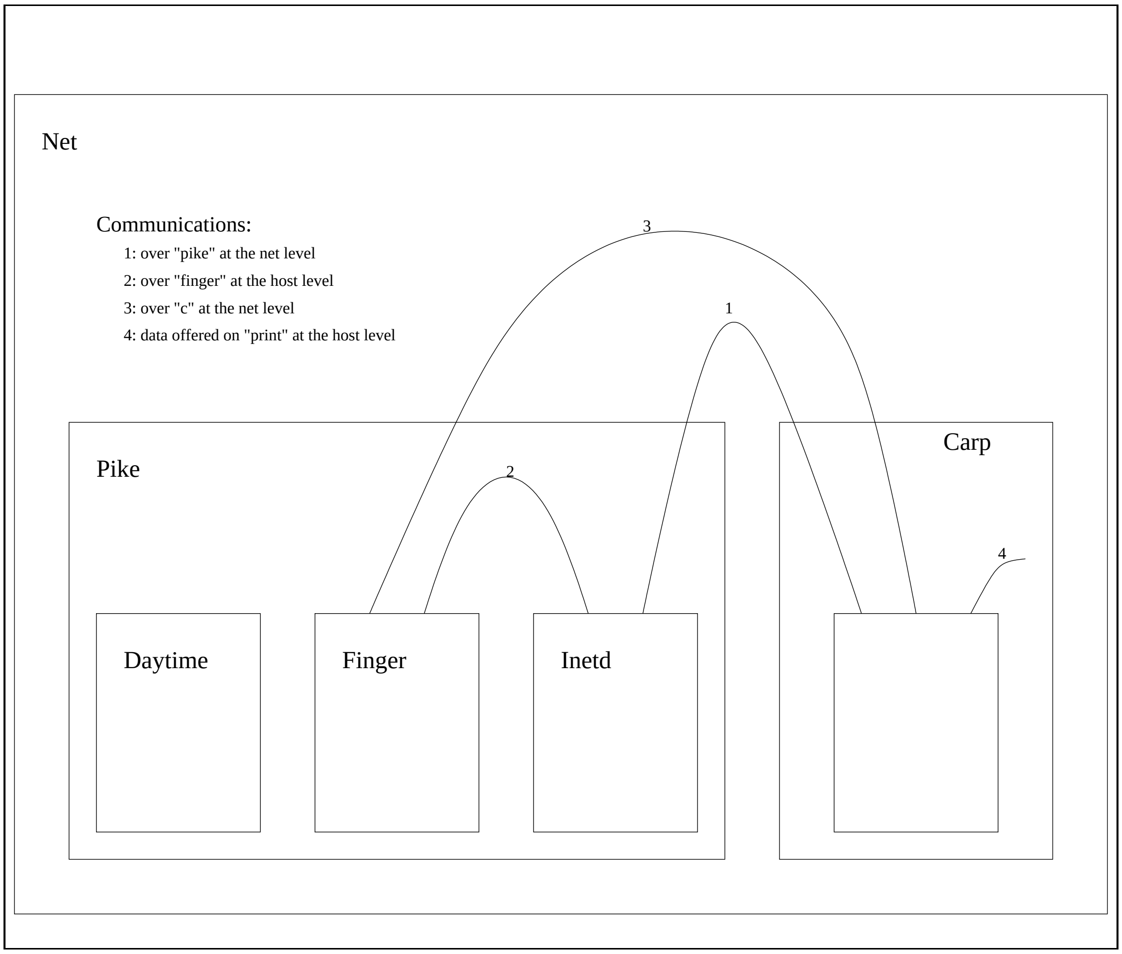


Figure 3.3: The communications made by the Internet Server Dæmon

$$\begin{array}{rl} \text{communication} & \\ \text{on } c@net & \xrightarrow{\tau} \nu c@net.\ host[\overline{print}\langle \textit{“PikeUsers”}\rangle] \\ & \qquad |host[\mathit{Inet} \mid \mathit{Finger} \mid \mathit{Daytime}] \end{array}$$

The areas of these reductions are illustrated in Figure 3.3. After a sequence of internal communications at the *net* and *host* level, the first host *Carp* is ready to print the information *“PikeUsers”* and the host *Pike* is restored to its original configuration. To aid this reduction, structural congruence was used to extend the scope of the new name c, an alternative to this would have been to use the OPEN_SEM rules and to let the new name declaration ride up on the output action.

Even this small example exhibits interesting scalability.

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

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

Figure 3.4: Example of processes using local areas: load management agents

- *Pike* can support multiple simultaneous *finger* or *daytime* requests, because freshly-created channels like c provide private communication links.

- The system can support *Finger* and *Daytime* servers on several hosts, with exactly the same agent code and protocol, because the *finger* and *daytime* names are known globally but communicate locally.

3.4.2 Load Management System

One of the original motivations of the local area π-calculus was to provide a dialect with which to explore agent systems. Figure 3.4 presents a very simple model of an agent-style load management system. 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 channel *link*.

The processes execute with the following result:

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

where k is the numerical ratio of the load on the two hosts. Output $\overline{print}\langle k\rangle$ 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 π-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.

3.4.3 Data Caching

Accessing remote data can be a time consuming and even costly affair. So it is common practice to keep a local copy of frequently requested information and provide the local copy in response to a request for remote data. This is done in most web browsers where the last few pages viewed are stored on the local hard disk and can subsequently be loaded from the disk hence avoiding an unnecessary download.

Figure 3.5 gives a model of a network in which data is not cached. Here we have three hosts: *Carp*, *Pike* and *Trout* all of which hold a string of data *"carpdata"*, *"pikedata"* and *"troutdata"* respectively. Each host will listen for a request from anywhere in the entire system and then send a copy of the data over the channel requested. This is done by the *Forward* applications.

To reduce the number of network level communications I add a cache for each piece of data at each host. These new applications will store local copies of the data pages from the other hosts and after an arbitrary time fetch a new copy of the data.

The key change is to assign the channels on which data is requested the *host* level instead of the *net* level. This single change is enough to ensure that any

$$Forward(chn, data) = app[!chn(n).\overline{n}\langle data\rangle]$$

$$Carp = host[Forward(getcarpdata, \text{``}carpdata\text{''}) \mid ...]$$
$$Pike = host[Forward(getpikedata, \text{``}pikedata\text{''}) \mid ...]$$
$$Trout = host[Forward(gettroutdata, \text{``}troutdata\text{''}) \mid ...]$$

$$\Lambda = getcarpdata, getpikedata, gettroutdata@net$$

$$\Lambda \vdash_{net} Carp \mid Pike \mid Trout$$

Figure 3.5: Example of processes passing data without caching it

processes running inside the host will seamlessly start to use the cached data. Three new channels are introduced to handle the communication between hosts, which I have called *pike*, *carp* and *trout*.

The cache process is parameterized on the name it uses to fetch data and the name on which it receives requests. The cache uses a private name, *cache*, to store the cached data. The *update* channel signals the process to update the cache.

$$\begin{aligned} Cache(get, offer) = app[\quad & \nu update, cache@app.(\overline{update} \mid \overline{cache}\langle data\rangle \\ & \mid !update.\nu m@net. \\ & \quad \overline{get}\langle m\rangle \mid m(data).(cache(\_).\overline{cache}\langle data\rangle \mid \overline{update})) \\ & \mid !offer(n).cache(data).(\overline{n}\langle data\rangle \mid \overline{cache}\langle data\rangle)] \end{aligned}$$

In this system each host starts off with a cache for all the data it might ever need. A more sophisticated system could dynamically create these caches as needed. Each host must also make copies of its local data available to these caches. This is done with the following link process.

$$Link(remote, local) = app[!remote(n).\nu m@host.(\overline{local}\langle m\rangle \mid m(data).\overline{n}\langle m\rangle)]$$

$$
\begin{aligned}
CarpCached = & host[Cache(pike, getpikedata) \mid Cache(trout, gettroutdata) \\
& \mid Link(carp, getcarpdata) \\
& \mid Forward(getcarpdata, \text{“}carpdata\text{”}) \mid ...] \\
Pikecached = & host[Cache(carp, getcarpdata) \mid Cache(trout, gettroutdata) \\
& \mid Link(pike, getpikedata) \\
& \mid Forward(getpikedata, \text{“}pikedata\text{”}) \mid ...] \\
Troutcached = & host[Cache(pike, getpikedata) \mid Cache(carp, getcarpdata) \\
& \mid Link(trout, gettroutdata) \\
& \mid Forward(gettroutdata, \text{“}troutdata\text{”}) \mid ...]
\end{aligned}
$$

$$
\begin{aligned}
Cache(get, offer) = app[\quad & \nu update, cache@app.(\overline{update} \mid \overline{cache}\langle data\rangle \\
& \mid !update.\nu m@net. \\
& \quad \overline{get}\langle m\rangle \mid m(data).(cache(\_).\overline{cache}\langle data\rangle \mid \overline{update})) \\
& \mid !offer(n).cache(data).(\overline{n}\langle data\rangle \mid \overline{cache}\langle data\rangle)]
\end{aligned}
$$

$$
Link(remote, local) = app[!remote(n).\nu m@host.(\overline{local}\langle m\rangle \mid m(data).\overline{n}\langle m\rangle)]
$$

$$
\Lambda = getcarpdata, getpikedata, gettroutdata@host \; ; carp, pike, trout@net
$$

$$
\Lambda \vdash_{net} \; Carpcached \mid Pikecached \mid Troutcached
$$

Figure 3.6: A system that caches data

The final system is given in Figure 3.6. I now have a system in which the data pages are called on the same channels from anywhere in the network and if a local copy of the data exists it will be provided and the local copy will be updated periodically.

3.5 Correctness of the Untyped Semantics

In this section I show that the operational semantics of Figure 3.1 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 8. *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 $(\nu\vec{d}@\vec{n})\bar{a}\langle\vec{b}\rangle$ or $a(\vec{b})$ then $\ell \leq \Lambda(a)$.*

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

Base Cases:

- OUT_SEM rule: the derivation is of the form $\Lambda \vdash_\ell \; \bar{a}\langle\vec{b}\rangle \xrightarrow{\bar{a}\langle\vec{b}\rangle} 0$. The process 0 contains no agents and so is well formed at any level. $fn(0) = \{\}$ therefore is trivially a subset of $dom(\Lambda) \cup bn(\alpha)$ and $\ell \leq \Lambda(a)$ follows as a side condition of OUT_SEM.
- IN_SEM rule: $P \equiv a(\vec{b}).P'$ for some P'. The derivation is of the form $\Lambda \vdash_\ell \; a(\vec{b}).P' \xrightarrow{a(\vec{b})} P'$. It can be clearly seen, that P and P' have the same set of top level agents and are at the same level. So P being well formed at level ℓ is enough to tell us that P' is also well formed.

  $\vec{b}$ are free names in P' but are bound in P, it is also possible that there are no other references to a in P' therefore:

$$fn(P') \cup \{a\} = fn(P) \cup \{\vec{b}\}$$
$$\text{and so } fn(P') \subseteq fn(P) \cup \{\vec{b}\}$$

We know that the free names of P are all in the domain of Λ because P is well formed by Λ and we also know that $bn(a(\vec{b})) = \vec{b}$. So:

$$fn(P') \subseteq dom(\Lambda) \cup bn(a).$$

Again $\ell \leq \Lambda(a)$ follows as a side condition of the reduction rule.

- IN!_SEM rule: $P \equiv\, !a(\vec{b}).P'$ for some P' and the derivation is of the form $\Lambda \vdash_\ell\, !a(\vec{b}).P' \xrightarrow{a(\vec{b})} P' \,|\, !a(\vec{b}).P'$. As before the top level agents of P' are exactly the top level agents of P. Therefore P' is well formed at level ℓ. As $fn(a(\vec{b}).P) = fn(!a(\vec{b}).P)$ and $bn(a(\vec{b}).P) = bn(!a(\vec{b}).P)$ the free names condition follows in the same way as the IN_SEM case. Also as above: $\ell \leq \Lambda(a)$ follows from the side condition on the reduction rule.

The Step cases.

- PAR_SEM rule: $P \equiv R \,|\, S$. The $tla(R) \subseteq tla(P)$ and $tla(S) \subseteq tla(P)$ therefore P being well formed is enough to tell us that R and S are well formed. The final derivation step takes the form:

$$\frac{\Lambda \vdash_\ell R \xrightarrow{\alpha} R'}{\Lambda \vdash_\ell R \,|\, S \xrightarrow{\alpha} R' \,|\, S}$$

  Therefore, by the induction hypothesis, R' is well formed at level ℓ, so, $R' \,|\, S$ is also well formed at level ℓ. The induction hypothesis also tells us that $fn(R') \subseteq dom(\Lambda) \cup bn(\alpha)$. As P is well formed in the environment Λ we know that $fn(P) = fn(R) \cup fn(S) \subseteq dom(\Lambda)$ and by dropping the free names of R we can also see that $fn(S) \subseteq dom(\Lambda)$. Combining these inequalities we get $fn(R' \,|\, S) = fn(R') \cup fn(S) \subseteq dom(\Lambda) \cup bn(\alpha)$.

  If the action α is an input or output then $\ell \leq \Lambda(a)$ will follow from the induction hypothesis.

- COMM_SEM rule: $P \equiv R \,|\, S$.

  The same argument as above implies that R and S are well formed at level ℓ. The final derivation step takes the form:

$$\frac{\Lambda \vdash_\ell \ R \xrightarrow{(\nu\vec{d}@\vec{n})\bar{a}\langle\vec{c}\rangle} R' \quad \Lambda \vdash_\ell \ S \xrightarrow{a(\vec{b})} S'}{\Lambda \vdash_\ell \ R \,|\, S \xrightarrow{\tau} \nu\vec{d}@\vec{n}.R' \,|\, S'\{\vec{c}/\vec{b}\}}$$

Therefore by the induction hypothesis R' and S' are well formed at level ℓ. Observing that well formedness is preserved under the substitution of names with the same level and when added a ν prefixing, we can see that $\nu\vec{d}@\vec{n}.R'|S'\{\vec{c}/\vec{b}\}$ is well formed at level ℓ.

The induction hypothesis tells us that:

$$fn(R') \subseteq dom(\Lambda) \cup bn((\nu\vec{d}@\vec{n})\bar{a}\langle\vec{c}\rangle) = dom(\Lambda) \cup \vec{d}$$

and

$$fn(S') \subseteq dom(\Lambda) \cup bn(a(\vec{b})) = dom(\Lambda) \cup \{\vec{b}\}$$

As $\vec{c}$ are free names of R and R is well formed, we know that $\vec{c}$ must be in the domain of Λ. So, therefore $fn(S'\{\vec{c}/\vec{b}\}) \subseteq dom(L)$ and $fn(R'|S'\{\vec{c}/\vec{b}\}) \subseteq dom(\Lambda) \cup \{\vec{d}\}$. From which we can conclude :$fn(\nu\vec{d}@\vec{n}.R'|S'\{\vec{c}/\vec{b}\}) \subseteq dom(\Lambda)$.

- BIND_SEM rule: $P \equiv \nu a@m.R$ and as above R is well formed at level ℓ. The final derivation step takes the form:

$$\frac{\Lambda; a@m \vdash_\ell \ R \xrightarrow{\alpha} R'}{\Lambda \vdash_\ell \ \nu a@m.R \xrightarrow{\alpha} \nu a@m.R'}$$

By the induction hypothesis R' is well formed at level ℓ so $\nu a@m.R'$ is also well formed. The induction hypotheses tells us that $fn(R') \subseteq dom(\Lambda; a@m) \cup bn(\alpha)$.

Therefore:

$$
\begin{aligned}
fn(\nu a@m.R') \cup \{a\} &= fn(R') \\
&\subseteq dom(\Lambda; a@m) \cup bn(\alpha) \\
&= dom(\Lambda) \cup \{a\} \cup bn(\alpha) \\
&\subseteq dom(\Lambda) \cup bn(\alpha)
\end{aligned}
$$

- AREA_SEM rule: $P \equiv m[R]$. Here, P is well formed at level ℓ therefore $m <_1 \ell$ and R is itself well-formed at level m. The final derivation step takes the form:

  $$\frac{\Lambda \vdash_m \; R \xrightarrow{\alpha} R'}{\Lambda \vdash_\ell \; m[R] \xrightarrow{\alpha} m[R']}$$

  By the induction hypothesis R' is well formed at level m and as $m <_1 \ell$ we can conclude that $m[R']$ is well formed at level ℓ.

  The free names of $m[R]$ are the same as the free names of R and the induction hypothesis tells us that the free names of R are a subset of the domain of Λ and the names bound by the action. So clearly $m[R] \subseteq dom(\Lambda) \cup bn(\alpha)$.

  Finally we can observe that the side condition of the AREA_SEM rule enforces that if α is $\bar{a}\langle\vec{b}\rangle$ or $a(\vec{b})$ then $\ell \leq \Lambda(a)$.

- OPEN_SEM rule:

  $$\frac{\Lambda; c@m \vdash_\ell \; P \xrightarrow{(\nu\vec{d}@\vec{n})\bar{a}\langle\vec{b}\rangle} P'}{\Lambda \vdash_\ell \; \nu c@m.P \xrightarrow{(\nu\vec{d}@\vec{n}, c@m)\bar{a}\langle\vec{b}\rangle} P'}$$

  with the side conditions $c \neq a \quad c \in \vec{b} \setminus dom(\Lambda)$.

  The well formedness of P' follows from the induction hypothesis. It also follows form the induction hypothesis that $\ell \leq \Lambda; c@m(a)$ and as the side condition guarantees $c \neq a$ we can conclude that $\ell \leq \Lambda(a)$. Finally:

  $$
  \begin{aligned}
  fn(P') &\subseteq dom(\Lambda; c@m) \cup bn((\nu\vec{d}@\vec{n})\bar{a}\langle\vec{b}\rangle) \\
  &\subseteq dom(\Lambda) \cup \{c\} \cup \vec{d} \cup bn(\bar{a}\langle\vec{b}\rangle) \\
  &\subseteq dom(\Lambda) \cup \{\vec{d}, c\} \cup bn(\bar{a}\langle\vec{b}\rangle) \\
  &\subseteq dom(\Lambda) \cup bn((\nu\vec{d}@\vec{n}, c@m)\bar{a}\langle\vec{b}\rangle)
  \end{aligned}
  $$

- STRUC_SEM rule:

$$\frac{P' \equiv P \quad \Lambda \vdash_\ell P' \xrightarrow{\alpha} Q' \quad Q \equiv Q'}{\Lambda \vdash_\ell \; P \xrightarrow{\alpha} Q}$$

  If $\Lambda \vdash_\ell P \xrightarrow{\alpha} Q$ by this rule we know that there must exist some Q' and P' such that $P' \equiv P, Q \equiv Q'$ and $\Lambda \vdash_\ell P' \xrightarrow{\alpha} Q'$. Therefore, by the induction hypothesis we know that:

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

  Noting that structural congruence does not change the levels or the free names of a process we can further concluded that:

  - the process Q is well-formed at level ℓ with $fn(Q) \subseteq dom(\Lambda) \cup bn(\alpha)$.

□

My first theorem states that after any number of internal communications a well formed process will still restrict a communication to its local area and will not allow channels to be used outside their areas.

Theorem 9. *If we can derive the sequence of transitions*

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

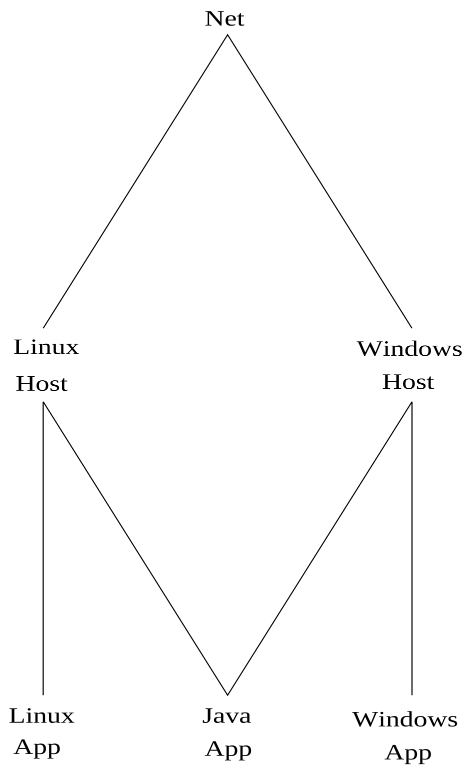
then

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

Proof. By repeated application of the Proposition 8. □

3.6 Non-Total Orders

The examples in this section have all used a total ordering of levels. This condition can be relaxed with interesting results. The semantics and the type system ensure that channels are used in the correct area by checking that the level of the channel is higher than or equal to the current level. This means that a channel that has a level that is incomparable with the current level is treated in exactly the same way as a channel that is assigned a lower level, i.e. it cannot be used. Partial orderings could be used to give a model of incompatible systems. To study a network with two different operating systems and three different types of process we could use the following partial order:



The type system will then ensure that *linux app*lication channels are only used on *linux host*s and *windows app*lications channels are only used on *windows hosts*. The *Java app* level is less than both the *linux host* and *windows host* level so *Java app*lications may be used on either type of host. If the calculus was extended with mobile agents, a partial ordering could also be used to specify where different types of agents might be sent. This could statically check that agents are never sent to the wrong kind of host, for example.

Chapter 4

Types for Local Areas

The results at the end of the last chapter 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 in Figure 3.1, of the form $\ell \leq \Lambda(a)$, which are essentially runtime level checks. In this chapter I show that a suitable type system can provide enough static information to make most of these checks unnecessary.

The rule AREA_SEM of Figure 3.1 deals with propagating actions once they have happened. Its side-condition remains. 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 $a(\vec{b}).P$ may 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, 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, so the process halts as b is only intended for communication within an application.

In the proposed type system channels are assigned types which specify not just the operating level of that channel, but also the levels of the channel names passed over it, and so on recursively. This type system is introduced in the next section. I then go on to prove some useful properties of these types, including that, well typedness is preserved by structural congruence. *Erasure*s are defined, to remove the types from a process and it is shown that the erasure of a well typed process is a well formed (untyped) process. Section 4.2 gives the channel types for the examples from the last chapter. The typed semantics is presented in Section 4.3 along with a proof of subject reduction and correctness. An extended type system, that allows recursive types is outlined in the final section of this chapter, 4.4.

4.1 The Type System

Channels are assigned types specifying not just the operating level of a channel, but also the levels of the channel names passed over it, and so on recursively. Data types allow for the use of strings and integers. Types are given by the following rather simple grammar:

$$\begin{array}{llll} \text{Type } \sigma & ::= & \vec{\sigma}@\ell & \text{Channel Type} \\ & | & \delta & \text{Data Types} \end{array} \qquad \begin{array}{lll} \text{Datatype } \delta & ::= & \mathit{string} \\ & | & \mathit{int} \end{array}$$

A type declaration of the form $a : \vec{\sigma}@\ell$ states that a is a ℓ-level channel carrying tuples of values whose types are given by the vector $\vec{\sigma}$. The type system ensures that data types are used for data only and are not used for communication. The second kind of base types are those with empty tuples: a channel of type $()@\ell$ is for synchronisation within a ℓ-area. This type system does not allow recursive types, Section 4.4 outlines how these may be added.

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

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

where σ is a channel type. 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.

Type environments Γ replace level environments Λ. These are finite partial maps from channel names to types. With these alterations, Figure 4.1 presents the rules for deriving type assertions of the form $\Gamma \vdash_\ell P$, which states that the process P is well typed at level ℓ in environment Γ.

4.1.1 Properties of Well Typed Processes

A simple induction on the type rules will show that: well-typedness is preserved by the substitution of similarly typed names.

Proposition 10. *If $\Gamma \vdash_\ell P$ and $\Gamma(a) = \sigma$ then for any name b of type σ not in Γ, it holds that $\Gamma\{b/a\} \vdash_\ell P\{b/a\}$*

A similar induction shows that if a process is well typed then any sub-term of that process is also well typed. When dealing with sub terms, it is useful to have a way to talk about the part of the process that surrounds the sub-term. This is done using contexts.

Definition 11. Context

A *la*π Context is a process with a "hole". These can be built up using the following rules:

$$\begin{array}{rcl} C\langle\!\langle \_ \rangle\!\rangle & = & \ell[C\langle\!\langle \_ \rangle\!\rangle] \\ & | & C\langle\!\langle \_ \rangle\!\rangle \mid Q \\ & | & \nu a{:}\sigma.C\langle\!\langle \_ \rangle\!\rangle \\ & | & a(\vec{b}).C\langle\!\langle \_ \rangle\!\rangle \\ & | & !a(\vec{b}).C\langle\!\langle \_ \rangle\!\rangle \\ & | & \_ \end{array}$$

A π Context can be defined by dropping the $\ell[C\langle\!\langle \_ \rangle\!\rangle]$ and the type on the ν binder from the above definition.

Any process can be placed inside a context; given a context $C\langle\!\langle \_ \rangle\!\rangle$ and a process P, define:

$$C\langle\!\langle P \rangle\!\rangle = C\langle\!\langle \_ \rangle\!\rangle\{P/\_\}$$

NULL_T	$\Gamma \vdash_\ell 0$	
OUT_T	$\Gamma \vdash_\ell \bar{a}\langle\vec{b}\rangle$	$\Gamma(a) = \vec{\sigma}@m$, $\Gamma(\vec{b}) = \vec{\sigma}$ and $\ell \leq m$
IN_T	$\dfrac{\Gamma; \vec{b} : \vec{\sigma} \vdash_\ell P}{\Gamma \vdash_\ell a(\vec{b}).P}$	$\Gamma(a) = \vec{\sigma}@m$ and $\ell \leq m$
IN!_T	$\dfrac{\Gamma; \vec{b} : \vec{\sigma} \vdash_\ell P}{\Gamma \vdash_\ell !a(\vec{b}).P}$	$\Gamma(a) = \vec{\sigma}@m$ and $\ell \leq m$
PAR_T	$\dfrac{\Gamma \vdash_\ell P \quad \Gamma \vdash_\ell Q}{\Gamma \vdash_\ell P \mid Q}$	
BIND_T	$\dfrac{\Gamma; a : \sigma \vdash_\ell P}{\Gamma \vdash_\ell \nu a{:}\sigma.P}$	
AREA_T	$\dfrac{\Gamma \vdash_\ell P}{\Gamma \vdash_m \ell[P]}$	$\ell <_1 m$

Figure 4.1: Well typed processes in the local area π-calculus

A context naturally defines an extension of an environment i.e. the new name bindings and the input name bindings that surround the *hole*. Adding the type rule $\Gamma \vdash_\ell \_$ allows us to define well typed contexts.

It is now possible to state the key proposition about sub terms.

Proposition 12. *If $\Gamma \vdash_\ell P$ and there exists a context C and process Q such that $P = C\langle\!\langle Q \rangle\!\rangle$ then there exists an environment Γ', which is an extension of Γ, and a level m that types Q i.e. $\Gamma' \vdash_m Q$.*

Proof. By structural induction on the context.

Base Cases:

- $C\langle\!\langle P \rangle\!\rangle = P$: We know that $\Gamma \vdash_\ell C\langle\!\langle P \rangle\!\rangle$ so it follows that $\Gamma \vdash_\ell P$
- $C\langle\!\langle P \rangle\!\rangle = 0$: This follows trivially as null is well typed with any environment and level.

Step cases:

- $C\langle\!\langle P \rangle\!\rangle = a(\vec{b}).D\langle\!\langle P \rangle\!\rangle$: In this case the IN_T rule must have been used, therefore we can conclude that $\Gamma; \vec{b} : \vec{\sigma} \vdash_\ell D\langle\!\langle P \rangle\!\rangle$ and apply the induction hypothesis.
- $C\langle\!\langle P \rangle\!\rangle = m[D\langle\!\langle P \rangle\!\rangle]$: In this case the AREA_T rule must have been used, therefore we can conclude that $\Gamma \vdash_m \ell[P]$ and apply the induction hypothesis.

All the other step cases follow in a similar way. $\square$

A useful follow up of this result is that well typedness is also preserved by the substitution of similarly typed sub terms.

Proposition 13. *If $\Gamma \vdash_\ell P$ and there exists a context C and processes Q and R such that $P = C\langle\!\langle Q \rangle\!\rangle$ and $\Gamma' \vdash_m Q$ and $\Gamma' \vdash_m R$ then $\Gamma \vdash_\ell C\langle\!\langle R \rangle\!\rangle$*

Proof. This proof follows the same pattern as the last proposition. Observe that the only property of Q used in the proof of Proposition 12 is that it is well typed at level m by environment Γ' therefore the proposition will hold with any similarly typed term in the place of Q. $\square$

This proposition is particularly useful when breaking down a process into a part that can perform an action and the surrounding context. In this setting, Q performs an action to become R.

The semantics of the $la\pi$-calculus includes the STRUC_SEM rule for the reduction of structure congruent terms. There is no equivalent type rule. The lack of a structural congruence rule means that the rules are type directed, i.e. there is a unique type rule for each piece of $la\pi$ syntax. So, given a well typed process, we know which rule was applied to type it, and we can deduce that the precondition of that rule must be true.

However, it is also possible to show that the well-typedness of a process is enough to imply the well-typedness of any processes structurally equivalent to it.

Proposition 14. *Well typedness is preserved by structural equivalence*

If $\Gamma \vdash_\ell P$ *and* $P \equiv R$ *then* $\Gamma \vdash_\ell R$.

Proof. I prove this by showing it holds for each structural congruence rule in turn. Transitivity and reflexivity both follow and Proposition 13 shows that the result is closed under any context and hence extends to the congruence relation.

- $a(\vec{b}).P \equiv a(\vec{c}).P\{\vec{c}/\vec{b}\}$

  If $\Gamma \vdash_\ell a(\vec{b}).P$ is typed by the IN_T rule then we know that $\Gamma; \vec{b} : \vec{\sigma} \vdash_\ell P$ is well typed. Now, by Proposition 10 well typedness is preserved under substitution of similarly typed names therefore $\Gamma; \vec{c} : \vec{\sigma} \vdash_\ell P\{\vec{c}/\vec{b}\}$ is well typed. From which we may conclude that $\Gamma \vdash_\ell a(\vec{c}).P\{\vec{c}/\vec{b}\}$ is well typed using the IN_rule.

  The reverse direction is an instance of the forward direction.

- $!a(\vec{b}).P \equiv\ !a(\vec{c}).P\{\vec{c}/\vec{b}\}$

  As the type rules IN_T and !IN_T are identical this case follows in the same way as above.

- $P \mid 0 \equiv P$

  If we know that $\Gamma \vdash_\ell P \mid 0$ is well typed by the PAR_T rule then $\Gamma \vdash_\ell P$ is well typed.

Whereas, if $\Gamma \vdash_\ell P$ is well typed, the NULL_T rule tells us that $\Gamma \vdash_\ell 0$ is also well typed, so we can use the PAR_T rule to tell us that $\Gamma \vdash_\ell P \,|\, 0$ is well typed.

- $P \,|\, Q \equiv Q \,|\, P$

  If $\Gamma \vdash_\ell P \,|\, Q$ is well typed by the typing rule, PAR_T then $\Gamma \vdash_\ell P$ and $\Gamma \vdash_\ell Q$ must also be well typed and these are exactly the preconditions needed by the PAR_T rule to type $\Gamma \vdash_\ell Q \,|\, P$.

  The reverse direction is an instance of the forward direction.

- $\nu a{:}\sigma P \equiv \nu b{:}\sigma.P\{^b/_a\}$

  If $\Gamma \vdash_\ell \nu a@\ell.P$ is well typed by the BIND_T rule then $\Gamma; a : \sigma \vdash P$ is well typed. Now, as types are preserved by substitution, Proposition 10, we know that $\Gamma; b : \sigma \vdash P\{^b/_a\}$ is well typed and so $\Gamma \vdash_\ell \nu b{:}\sigma.P\{^b/_a\}$ is well typed by the BIND_T rule.

  The reverse direction is an instance of the forward direction.

- $(P \,|\, Q) \,|\, R \equiv P \,|(Q \,|\, R)$

  If $\Gamma \vdash_\ell (P \,|\, Q) \,|\, R$ is well typed then by the PAR_T rule $\Gamma \vdash_\ell R$ is well typed and $\Gamma \vdash_\ell P \,|\, Q$ is also well typed. So by the same rule again, $\Gamma \vdash_\ell P$ is well typed, as is $\Gamma \vdash_\ell Q$.

  Observing that $\Gamma \vdash_\ell Q$ and $\Gamma \vdash_\ell R$ are well typed we can then type $\Gamma \vdash_\ell Q \,|\, R$ by the PAR_T rule. Finally observing that $\Gamma \vdash_\ell P$ is well typed lets us use the PAR_T rule to type: $\Gamma \vdash_\ell P \,|(Q \,|\, R)$

  A similar method will also prove the other direction.

- $\nu a{:}\sigma.0 \equiv 0$

  By the NULL_T rule $\Gamma \vdash_\ell 0$ is well typed for all possible Γ and ℓ therefore it is also true that $\Gamma; a : \sigma \vdash_\ell 0$ for all possible Γ, a, σ and ℓ from which we may use the BIND_T rule to conclude that $\Gamma \vdash_\ell \nu a{:}\sigma.0$ for all possible Γ and ℓ.

  So both sides of the equivalence are always well typed.

- $\nu a{:}\sigma.\nu b{:}\rho.P \equiv \nu b{:}\rho.\nu a{:}\sigma.P \qquad a \neq b$

  If $\Gamma \vdash_\ell \nu a{:}\sigma.\nu b{:}\rho.P$ is well typed by the BIND_T rule then $\Gamma; a : \sigma \vdash_\ell \nu b{:}\rho.P$ is also well typed by the BIND_T rule and hence, as $a \neq b$: $\Gamma; a : \sigma; b : \rho \vdash_\ell P$ is well typed.

  Now, as $\Gamma; a : \sigma; b : \rho \equiv \Gamma; b : \rho; a : \sigma$ it follows that $\Gamma; b : \rho; a : \sigma \vdash_\ell P$. So by the BIND_T rule $\Gamma; b : \rho \vdash_\ell \nu a{:}\sigma P$ is well typed and with another application we see that $\Gamma \vdash_\ell \nu b{:}\rho.\nu a{:}\sigma.P$ is also well typed.

  The reverse direction is an instance of the forward direction.

- $\ell[\nu a{:}\sigma.P] \equiv \nu a{:}\sigma.\ell[P]$

  If $\Gamma \vdash_m \ell[\nu a{:}\sigma.P]$ is well typed by the AREA_T rule then $\ell <_1 m$ and $\Gamma \vdash_\ell \nu a{:}\sigma.P$ is well typed by the BIND_T rule. This in turn implies that $\Gamma; a : \sigma \vdash_\ell P$.

  Now $\Gamma; a : \sigma \vdash_\ell P$ and $\ell <_1 m$ are the preconditions we need to apply the AREA_T rule to get: $\Gamma; a : \sigma \vdash_m \ell[P]$ and then the BIND_T rule can be applied to get $\Gamma \vdash_m \nu a{:}\sigma.\ell[P]$

  A similar method will also prove the other direction.

- $(\nu a{:}\sigma.P) \,|\, Q \equiv \nu a{:}\sigma.(P \,|\, Q) \qquad a \notin fn(Q)$

  If $\Gamma \vdash_\ell (\nu a{:}\sigma.P) \,|\, Q$ is well typed by the PAR_T rule then $\Gamma \vdash_\ell Q$ must be well typed and $\Gamma \vdash_\ell \nu a{:}\sigma.P$ must be well typed by the BIND_T rule. This second point also tells us that $\Gamma; a : \sigma \vdash_\ell P$ is well typed. Now, as a is not a free name of Q we can add it to Γ and still be able to type the processes: $\Gamma; a : \sigma \vdash_\ell Q$. So, by the PAR_T rule $\Gamma; a : \sigma \vdash_\ell P \,|\, Q$ and finally by the BIND_T rule: $\Gamma \vdash_\ell \nu a{:}\sigma.(P \,|\, Q)$.

  Reversing this proof will prove the reverse of the equivalence.

□

This proposition allows us to apply the type rules “backward” to show that the rule’s preconditions are true and also prove well typedness using structural congruence.

This result could also be stated in the following way:

Defining well typedness up to structural congruence is the same as defining well typedness using the typing rules alone.

This is a non-trivial result and the equivalent proposition does not hold for the semantics of most calculi, as it is often necessary to rearrange a process before it can be reduced.

4.1.2 Linking Typed and Untyped Processes

To connect the typed calculus to the untyped one I use a notion of *erasure*. This throws away the detail of type information, but keeps the basic level declaration.

Definition 15. If $\sigma = \vec{\sigma}@\ell$ is a channel type, then its *erasure* $\lfloor\sigma\rfloor$ is just the level ℓ. If σ is a data type then the erasure is the empty set.

Definition 16. 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 $\nu a@\ell.Q$.

Definition 17. The erasure of a type context Γ is the level context $\lfloor\Gamma\rfloor$ produce by erasing each of the types in the name bindings and removing the data type assignments.

As you would expect the erasure of a well typed process is a well formed process, as this following proposition shows.

Proposition 18. *If P is a well typed process at level ℓ in type environment Γ, then its erasure $\lfloor P\rfloor$ is well formed at level ℓ in the level environment $\lfloor\Gamma\rfloor$.*

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

Proof. The proof is by structural induction on the type derivation $\Gamma \vdash_\ell P$.

Recall from Definitions 7 and 4 that $\Lambda \vdash_\ell P$ if and only if all free names used in P are in the domain of Λ and the process P is well formed i.e. the areas are properly nested.

Base cases:

- NULL_T rule: If we have applied this rule, then $P \equiv 0$. The erasure of null is null, which is trivially well formed.

- OUT_T rule: If we have applied this rule, then $P \equiv \bar{a}\langle\vec{b}\rangle$. The side condition on this rule ensures that $\Gamma(a) = \vec{\sigma}@m$ for some $\vec{\sigma}$ and m, with $\Gamma(\vec{b}) = \vec{\sigma}$. Now $dom\,\Gamma = dom\,\lfloor\Gamma\rfloor$ so $a, b \in dom(\lfloor\Gamma\rfloor)$ and therefore all the free names in P are in the domain of $\lfloor\Gamma\rfloor$.

  As $\bar{a}\langle\vec{b}\rangle$ contains no agents it is trivially well formed.

Step cases

- COMP_T rule: If we have applied this rule, then $P \equiv R\,|\,Q$ where $\Gamma \vdash_\ell R$ and $\Gamma \vdash_\ell Q$. So, by the induction hypothesis $\lfloor\Gamma\rfloor \vdash_\ell \lfloor R\rfloor$ and $\lfloor\Gamma\rfloor \vdash_\ell \lfloor Q\rfloor$. This means that the free names of $\lfloor R\rfloor$ and $\lfloor S\rfloor$ are in the domain of $\lfloor\Gamma\rfloor$. So,

  $$fn(\lfloor R\,|\,Q\rfloor) = fn(\lfloor R\rfloor) \cup fn(\lfloor Q\rfloor) \subseteq dom(\lfloor\Gamma\rfloor) \cup dom(\lfloor\Gamma\rfloor) = dom(\lfloor\Gamma\rfloor)$$

  We know by the induction hypothesis that the top level agents of R and Q are well formed therefore all the top level agents of $R\,|\,Q$ are well formed.

- BIND_T rule: If we have applied this rule then $P \equiv \nu a{:}\sigma.P'$ and $\Gamma; a : \sigma \vdash_\ell P'$. So, by the induction hypothesis $\lfloor\Gamma; a : \sigma\rfloor \vdash_\ell \lfloor P'\rfloor$ and $fn(P') \subseteq dom(\lfloor\Gamma\rfloor \cup \{a{:}\lfloor\sigma\rfloor\})$ but $fn(P') \setminus \{a\} = fn(P)$ so $fn(P) \subseteq dom(\lfloor\Gamma\rfloor)$. And as P' is well formed $\nu a{:}\sigma.P'$ is well formed.

- AREA_T rule: If we have applied this rule, then $P \equiv m[P']$ and $\Gamma \vdash_m P'$. So by the induction hypothesis the free names of P' are in the domain of $\lfloor\Gamma\rfloor$ and as P has the same free names as P' we also know that $fn(P) \subseteq dom(\lfloor\Gamma\rfloor)$.

The side conditions to the type rule imply that $\ell <_1 m$ and the induction hypothesis implies that $\lfloor\Gamma\rfloor \vdash_m \lfloor P'\rfloor$ and so P' is well formed at level m. Therefore P is well formed at level ℓ.

- IN_T rule: If we have applied this rule, then $P \equiv a(\vec{b}).P'$ and $\Gamma; \vec{b} : \vec{\sigma} \vdash_\ell P'$ and $\Gamma(a) = \vec{\sigma}@m$ and $\ell \leq m$. So by the induction hypothesis $fn(P') \subseteq dom(\lfloor\Gamma\rfloor \cup \{\vec{b} : \lfloor\vec{\sigma}\rfloor\})$ but $fn(P') \setminus \{\vec{b}\} = fn(P)$ so $fn(P) \subseteq dom(\lfloor\Gamma\rfloor)$.

  It follows that the induction hypothesis also implies that P' is well formed and so $a(\vec{b}).P'$ will be well formed.

- IN!_T rule: Similar to the IN_T rule.

$\square$

4.2 Channel Types for the Examples

I will now give types to the examples from Section 3.4, which sensibly reflect their operation. First, the Internet Dæmon of Figure 3.2 (page 46). The channels have the following types:

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

The type *service* for *finger* and *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 *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.

In the second example of Section 3.4.2 (page 48), an agent compares the load on two hosts. The channels have the following types:

$$c : int@host \qquad load : (int@host)@host$$
$$link : int@net \qquad print : int@host$$

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.

The third example, presented in Section 3.4.3 (page 49), is a model of a simple data cache. The first system in this section did not cache data. There are three pieces of data, which are *string*s. The *get* channels are used to listen for a channel over which the data can then be sent.

$$\begin{aligned} getcarpdata &: (string@net)@net \\ getpikedata &: (string@net)@net \\ gettroutdata &: (string@net)@net \end{aligned}$$

The principal difference in the typing of the cached system is that the channels that get the data now operate at the *host* level. It is this change alone that ensures that data is retrieved locally rather than from a remote source. Whereas, before, the data channels would listen across the entire network for a channel on which to broadcast the data again across the entire network, now these channels listen locally for a local channel over which to send data. The network level communications are now performed on the channels *pike*, *carp* and *trout*, hence these channels can communicate at the net level.

$$\begin{aligned} getcarpdata &:(string@host)@host & carp &:(string@net)@net \\ getpikedata &:(string@host)@host & pike &:(string@net)@net \\ gettroutdata &:(string@host)@host & trout &:(string@net)@net \end{aligned}$$

The *cache* channel is used by the cache application to store a copy of a data page and so therefore has the type *string*@*app*. The *update* channel is used by the same agent to synchronise a timeout inside that application and so has the type: ()@*app*.

4.3 Correctness of the Typed Semantics

The semantics for typed processes is given in Figure 4.2. The only difference from the untyped semantics given in Figure 3.1 is that the typed semantics does not have any side conditions on the OUT, IN and IN! rules.

This section proves that my type system and typed semantics really do enforce local areas, just as the untyped semantics did. I prove the equivalent of Theorem 9 for typed processes. That is: a well typed process that is allowed to reduce under the typed semantics will not use a channel outside its area nor will it allow a communication on a channel to escape the area assigned to it.

The first step is to show that this operational semantics preserves types. I then show that if I remove the types from a process and reduce it using the untyped semantics the same reductions are possible. Finally, I apply Theorem 9 to get the result.

Proposition 19. *Subject Reduction*

If P is a well typed process at level ℓ in an environment Γ and we can derive the transition $\Gamma \vdash_\ell P \xrightarrow{\alpha} Q$, then:

1. *if $\alpha = (\nu\vec{d}{:}\vec{\theta})\bar{a}\langle\vec{b}\rangle$ then $\Gamma; \vec{d}{:}\vec{\theta} \vdash_\ell Q$ and there exists some $\vec{\sigma}$ and some m s.t. $\Gamma(a) = \vec{\sigma}@m$ and $\Gamma; d{:}\vec{\theta}(\vec{b}) = \vec{\sigma}$.*
2. *if $\alpha = a(\vec{b})$ then $\Gamma; \vec{b}{:}\vec{\sigma} \vdash_\ell Q$ where $\Gamma(a) = \vec{\sigma}@m$.*
3. *if $\alpha = \tau$ then $\Gamma \vdash_\ell Q$*

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

Base Cases

- OUT_TSEM rule: The derivation is of the form: $\Gamma \vdash_\ell \ \bar{a}\langle\vec{b}\rangle \xrightarrow{\bar{a}\langle\vec{b}\rangle} 0$ and null is trivially well typed in any context. The only rule that could be used to type P is the OUT_T rule, therefore the side condition of that rule must apply, namely: there exists some m such that $\Gamma(a) = \vec{\sigma}@m$ and $\Gamma(b) = \vec{\sigma}$.
- IN_TSEM rule: The derivation is of the form: $\Gamma \vdash_\ell \ a(\vec{b}).Q \xrightarrow{a(\vec{b})} Q$ with $P \equiv a(\vec{b}).Q$.

OUT_TSEM	$\Gamma \vdash_\ell \ \bar{a}\langle\vec{b}\rangle \xrightarrow{\bar{a}\langle\vec{b}\rangle} 0$	
IN_TSEM	$\Gamma \vdash_\ell \ a(\vec{b}).P \xrightarrow{a(\vec{b})} P$	$\vec{b} \cap dom(\Gamma) = \emptyset$
IN!_TSEM	$\Gamma \vdash_\ell \ !a(\vec{b}).P \xrightarrow{a(\vec{b})} P \,\vert\, !a(\vec{b}).P$	$\vec{b} \cap dom(\Gamma) = \emptyset$
PAR_TSEM	$\dfrac{\Gamma \vdash_\ell \ P \xrightarrow{\alpha} P'}{\Gamma \vdash_\ell \ P \,\vert\, Q \xrightarrow{\alpha} P' \,\vert\, Q}$	
COMM_TSEM	$\dfrac{\Gamma \vdash_\ell \ P \xrightarrow{(\nu\vec{d}:\vec{\theta})\bar{a}\langle\vec{c}\rangle} P' \quad \Gamma \vdash_\ell \ Q \xrightarrow{a(\vec{b})} Q'}{\Gamma \vdash_\ell \ P \,\vert\, Q \xrightarrow{\tau} \nu\vec{d}{:}\vec{\theta}.(P' \,\vert\, Q'\{\vec{c}/\vec{b}\})}$	
BIND_TSEM	$\dfrac{\Gamma; a : \sigma \vdash_\ell \ P \xrightarrow{\alpha} P'}{\Gamma \vdash_\ell \ \nu a{:}\sigma.P \xrightarrow{\alpha} \nu a{:}\sigma.P'}$	$a \notin fn(\alpha)$
OPEN_TSEM	$\dfrac{\Gamma; c : \sigma \vdash_\ell \ P \xrightarrow{(\nu\vec{d}:\vec{\theta})\bar{a}\langle\vec{b}\rangle} P'}{\Gamma \vdash_\ell \ \nu c{:}\sigma.P \xrightarrow{(\nu\vec{d}:\vec{\theta},c:\sigma)\bar{a}\langle\vec{b}\rangle} P'}$	$c \neq a \quad c \in \{\vec{b}\} \setminus dom(\Gamma)$
AREA_TSEM	$\dfrac{\Gamma \vdash_\ell \ P \xrightarrow{\alpha} P'}{\Gamma \vdash_m \ \ell[P] \xrightarrow{\alpha} \ell[P']}$	if α is $\bar{a}\langle\vec{b}\rangle$ or $a(\vec{b})$ then $m \leq \Gamma(a)$
STRUC_TSEM	$\dfrac{P' \equiv P \quad \Gamma \vdash_\ell P \xrightarrow{\alpha} Q \quad Q \equiv Q'}{\Gamma \vdash_\ell \ P' \xrightarrow{\alpha} Q'}$	

Figure 4.2: Operational semantics for the typed local area π-calculus

We know that $\Gamma \vdash_\ell P$ and as IN_T is the only rule that can type an input process we know that the side and pre-conditions of these rules must apply. I.e. $\Gamma; \vec{b} : \vec{\sigma} \vdash_\ell Q$ for $\Gamma(a) = \vec{\sigma}@m$.

- !IN_TSEM rule: The derivation is of the form: $\Gamma \vdash_\ell !a(\vec{b}).P' \xrightarrow{a(\vec{b})} P' \,|\, !a(\vec{b}).P'$ we know that $\Gamma \vdash_\ell !a(\vec{b}).P'$ and as above $\Gamma; \vec{b} : \vec{\sigma} \vdash_\ell P'$ therefore $\Gamma; \vec{b} : \vec{\sigma} \vdash_\ell P' \,|\, !a(\vec{b}).P'$.

Step Cases

- PAR_TSEM rule: To apply this rule P must be of the form $R \,|\, S$ and the derivation of the form:

$$\frac{\Gamma \vdash_\ell \; R \xrightarrow{\alpha} R'}{\Gamma \vdash_\ell \; R \,|\, S \xrightarrow{\alpha} R' \,|\, S}$$

As P is well typed and the only rule that can type a composition is the PAR_T rule this rules precondition must apply i.e. that $\Gamma \vdash_\ell R$ and $\Gamma \vdash_\ell S$. The result then follows from the induction hypothesis and the PAR_T rule.

- COMM_TSEM rule: To apply this rule P must be of the form $R \,|\, S$ and the derivation of the form:

$$\frac{\Gamma \vdash_\ell \; R \xrightarrow{(\nu\vec{d}:\vec{\theta})\bar{a}\langle\vec{c}\rangle} R' \quad \Gamma \vdash_\ell \; S \xrightarrow{a(\vec{b})} S'}{\Gamma \vdash_\ell \; R \,|\, S \xrightarrow{\tau} \nu\vec{d}:\vec{\theta}.(R' \,|\, S'\{\vec{c}/\vec{b}\})}$$

The induction hypothesis then tells us that: $\Gamma; \vec{d}:\vec{\theta} \vdash_\ell R'$ and $\Gamma; \vec{b} : \vec{\sigma} \vdash_\ell S'$ where $\Gamma(a) = \vec{\sigma}@m$. Observing that well typedness is preserved under substitution of names with the same type we see that $\Gamma; \vec{b} : \vec{\sigma} \vdash_\ell S'\{\vec{c}/\vec{b}\}$ but $\vec{b}$ does not appear in $S'\{\vec{c}/\vec{b}\}$ so $\Gamma \vdash_\ell S'\{\vec{c}/\vec{b}\}$ and by the PAR_T rule $\Gamma; \vec{d}:\vec{\theta} \vdash_\ell R' \,|\, S'\{\vec{c}/\vec{b}\}$ and so $\Gamma \vdash_\ell \nu\vec{d}:\vec{\theta}.R' \,|\, S'\{\vec{c}/\vec{b}\}$

- BIND_TSEM rule: To apply this rule P must be of the form $\nu a{:}\sigma.R$ and the derivation of the form:

$$\frac{\Gamma; a : \sigma \vdash_\ell \; R \xrightarrow{\alpha} R'}{\Gamma \vdash_\ell \; \nu a{:}\sigma.R \xrightarrow{\alpha} \nu a{:}\sigma R'}$$

The only possible rule that could be used to type P is the BIND_T rule. So, the precondition of this rule tells us that: $\Gamma; a{:}\sigma \vdash_\ell \; R$ and a is not free in α the result then follows from the induction hypothesis.

- AREA_TSEM rule: To apply this rule P must be of the form $m[R]$ and the derivation of the form:

$$\frac{\Gamma \vdash_m \; R \xrightarrow{\alpha} R'}{\Gamma \vdash_\ell \; m[R] \xrightarrow{\alpha} m[R']}$$

Again the result follows from there only being one possible type rule that could apply to this process, in this case AREA_T. This rule's precondition tells us that $\Gamma \vdash_m R$. The result then follows from the induction hypothesis and the AREA_T rule.

- OPEN_TSEM rule:

$$\frac{\Gamma; c : \sigma \vdash_\ell \; P \stackrel{(\nu \vec{d}:\vec{\theta})\bar{a}\langle\vec{b}\rangle}{\longrightarrow} P'}{\Gamma \vdash_\ell \; \nu c{:}\sigma.P \stackrel{(\nu \vec{d}:\vec{\theta},c:\sigma)\bar{a}\langle\vec{b}\rangle}{\longrightarrow} P'}$$

with the side conditions $c \neq a$ and $c \in \{\vec{b}\} \backslash \, dom(\Gamma)$

The induction hypothesis tells us that $\Gamma; c{:}\sigma; \vec{d}{:}\vec{\theta} \vdash_\ell P'$ and there exists some $\vec{\sigma'}$ and some m s.t. $\Gamma; c{:}\sigma(a) = \vec{\sigma'}@m$ and $\Gamma; c{:}\sigma; d{:}\vec{\theta}(\vec{b}) = \vec{\sigma'}$. So all that remains to prove is that a can be typed without c in the enviroment and this follows from the side condition $c \neq a$.

- STRUC_TSEM rule:

$$\frac{P' \equiv P \quad \Gamma \vdash_\ell P \xrightarrow{\alpha} Q \quad Q \equiv Q'}{\Gamma \vdash_\ell \; P' \xrightarrow{\alpha} Q'}$$

The induction hypothesis, tells us that Q is well typed for each possible action. The preservation of well typedness, by structural congruence (Proposition 14) then tells us that Q' is well typed under the same conditions.

$\square$

As expected, there is an extremely tight connection between the behaviour of typed process terms and their untyped erasures, as I show in this next proposition.

Proposition 20. *Suppose that $\Gamma \vdash_\ell P$ is some well typed process.*

1. *If $\Gamma \vdash_\ell P \xrightarrow{\alpha} P'$ then $\lfloor\Gamma\rfloor \vdash_\ell \lfloor P\rfloor \xrightarrow{\alpha} \lfloor P'\rfloor$.*
2. *If $\lfloor\Gamma\rfloor \vdash_\ell \lfloor P\rfloor \xrightarrow{\alpha} Q$ for some untyped Q, then there is a process P' such that $\Gamma \cup bn(\alpha) \vdash_\ell P'$ and $Q \equiv \lfloor P'\rfloor$ and $\Gamma \vdash_\ell P \xrightarrow{\alpha} P'$.*

Proof. By structural induction on the derivation of the transition.

1. The base cases follow from $\Gamma(a) = \vec{\sigma}@m$ and $\ell \leq m$ implies $\ell \leq \lfloor\Gamma\rfloor(a)$. The PAR_TSEM,COMM_TSEM,BIND_TSEM, STRUC_TSEM and OPEN_TSEM cases do not make use of the extra information provided by the types or levels and so follow trivially. The AREA_SEM case follows from: $\Gamma(a) = \vec{\sigma}@m$ and $\ell \leq m$ implying $\ell \leq \lfloor\Gamma\rfloor(a)$
2. Base cases:
   - IN_SEM: In this case $P \equiv a(\vec{b}).P'$ and $\alpha = a(\vec{b})$, so $\lfloor a(\vec{b}).P\rfloor \equiv a(\vec{b}).\lfloor P'\rfloor$ and the derivation tree is of the form:
   $$\lfloor\Gamma\rfloor \vdash_\ell a(\vec{b}).\lfloor P'\rfloor \xrightarrow{a(\vec{b})} \lfloor P'\rfloor$$
   The result follows by subject reduction and observing that: $\Gamma \vdash_\ell a(\vec{b}).P' \xrightarrow{a(\vec{b})} P'$
   - !IN_SEM: This case is similar to the IN_SEM case with the additional observations that:
   $$\lfloor !a(\vec{b}).P'\rfloor \equiv !a(\vec{b}).\lfloor P'\rfloor \text{ and } \lfloor P'\rfloor \,|\, \lfloor !a(\vec{b}).P'\rfloor \equiv \lfloor P' \,|\, !a(\vec{b}).P'\rfloor.$$

- OUT_SEM: Output is asynchronous so in this case $Q \equiv 0$ i.e.

$$\lfloor\Gamma\rfloor \vdash_\ell \ \bar{a}\langle\vec{b}\rangle.\lfloor P'\rfloor \xrightarrow{\bar{a}\langle\vec{b}\rangle} 0$$

As the null process contains no types our result follows from $Q \equiv 0 \equiv \lfloor 0\rfloor \equiv \lfloor Q\rfloor$.

Step Cases

- PAR_SEM: In this case $P \equiv R\,|\,S$ and observing that $\lfloor R\,|\,S\rfloor = \lfloor R\rfloor\,|\,\lfloor S\rfloor$ the derivation tree is of the form:

$$\frac{\lfloor\Gamma\rfloor \vdash_\ell \ \lfloor R\rfloor \xrightarrow{\alpha} R'}{\lfloor\Gamma\rfloor \vdash_\ell \ \lfloor R\,|\,S\rfloor \xrightarrow{\alpha} R'\,|\,\lfloor S\rfloor}$$

So by the induction hypothesis there exists a well typed T such that $\lfloor T\rfloor = R'$ and $\Gamma \vdash_\ell R \xrightarrow{\alpha} T$. So by the PAR_SEM rule $\Gamma \vdash_\ell R\,|\,S \xrightarrow{\alpha} T\,|\,S$ and $R'\,|\,\lfloor S\rfloor = \lfloor T\rfloor\,|\,\lfloor S\rfloor = \lfloor T\,|\,S\rfloor$, as required.

- COMM_SEM: In this case $P = S\,|\,R$ and the derivation tree is of the form:

$$\frac{\lfloor\Gamma\rfloor \vdash_\ell \ \lfloor S\rfloor \xrightarrow{(\nu\vec{d}@\vec{\theta})\bar{a}\langle\vec{c}\rangle} S' \quad \lfloor\Gamma\rfloor \vdash_\ell \ \lfloor R\rfloor \xrightarrow{a(\vec{b})} R'}{\lfloor\Gamma\rfloor \vdash_\ell \ \lfloor S\,|\,R\rfloor \xrightarrow{\tau} S'\,|\,R'\{\vec{c}/\vec{b}\}}$$

So by the induction hypothesis there exists a well typed T and U such that $\lfloor T\rfloor = R'$, $\lfloor U\rfloor = S'$, $\Gamma \vdash_\ell R \xrightarrow{a(\vec{b})} T$ and $\Gamma \vdash_\ell S \xrightarrow{(\nu\vec{d}@\vec{\theta})\bar{a}\langle\vec{c}\rangle} U$. So by the COMM_TSEM rule we can conclude that:

$$\Gamma \vdash_\ell S\,|\,R \xrightarrow{\tau} U\,|\,T\{\vec{c}/\vec{b}\}$$

And noting that $\lfloor U\,|\,T\{\vec{c}/\vec{b}\}\rfloor = \lfloor S'\rfloor\,|\,\lfloor R'\{\vec{c}/\vec{b}\}\rfloor = \lfloor S'\,|\,R'\{\vec{c}/\vec{b}\}\rfloor$ finishes the case.

- BIND_SEM: In this case $\lfloor P\rfloor = \nu a@m.\lfloor R\rfloor$ therefore $P = \nu a{:}\sigma.R$ for some σ' such that $\sigma = \sigma'@m$. The derivation tree is of the form:

$$\frac{\lfloor\Gamma\rfloor; a@m \vdash_\ell \ \lfloor R\rfloor \xrightarrow{\alpha} S}{\lfloor\Gamma\rfloor \vdash_\ell \ \nu a{:}m.\lfloor R\rfloor \xrightarrow{\alpha} \nu a{:}m.S}$$

where $a \notin fn(\alpha)$. Now the erasure of $\sigma'@m$ is m so from $\lfloor\Gamma\rfloor; a@m \vdash_\ell \lfloor R\rfloor \xrightarrow{\alpha} S$ it follows that $\lfloor\Gamma; a : \sigma'@m\rfloor \vdash_\ell \lfloor R\rfloor \xrightarrow{\alpha} S$.

We can then apply the induction hypothesis which tells us that there exists a well typed R' such that its erasure is S and $\Gamma; a : \sigma'@m \vdash_\ell R \xrightarrow{\alpha} R'$.

The typing rule BIND_T tells us that from $\Gamma; a : \sigma'@m \vdash_\ell R'$ we can conclude $\Gamma \vdash_\ell \nu a{:}\sigma'@m.R'$. Observing that: $\Gamma \vdash_\ell \nu a{:}\sigma'@m.R \xrightarrow{\alpha} \nu a{:}\sigma'@m.R'$ finishes the case.

- AREA_SEM: In this case $P = m[R]$ where $\ell <_1 m$. and the derivation tree is of the form:

$$\frac{\lfloor\Gamma\rfloor \vdash_\ell \ \lfloor R\rfloor \xrightarrow{\alpha} S}{\lfloor\Gamma\rfloor \vdash_m \ \ell[\lfloor R\rfloor] \xrightarrow{\alpha} \ell[S]}$$

  So the induction hypothesis tells us that there exists R' such that $\Gamma \vdash_m R'$, $\lfloor R'\rfloor = S$ and $\Gamma \vdash_m R \xrightarrow{\alpha} R'$. The type rule for areas tells us that $\Gamma \vdash_\ell m[R]$ and observing that $\Gamma \vdash_\ell m[R] \xrightarrow{\alpha} m[R']$ finishes the proof.

- OPEN_SEM: In this case $\lfloor P\rfloor = \nu c@m.\lfloor R\rfloor$ therefore $P = \nu c{:}\sigma.R$ for some σ' such that $\sigma = \sigma'@m$. The derivation tree is of the form:

$$\frac{\lfloor\Gamma\rfloor; c@m \vdash_\ell \ \lfloor P\rfloor \xrightarrow{(\nu\vec{d}@\vec{n})\bar{a}\langle\vec{b}\rangle} P'}{\lfloor\Gamma\rfloor \ \vdash_\ell \nu c@m.\lfloor P\rfloor \xrightarrow{(\nu\vec{d}@\vec{n},c@m)\bar{a}\langle\vec{b}\rangle} P'}$$

  with the side conditions $c \neq a$ and $c \in \vec{b} \setminus dom(\lfloor\Gamma\rfloor)$.

  The binding of the name c can be moving inside the erasure to tell us that: $\lfloor\Gamma; c : \sigma\rfloor \vdash_\ell \ \lfloor P\rfloor \xrightarrow{(\nu\vec{d}@\vec{n})\bar{a}\langle\vec{b}\rangle} P'$. The induction hypothesis can be applied to this to tell us that there exists a typed process R such that $\Gamma; c : \sigma; \vec{d} : \vec{\theta} \vdash_\ell R$ and $\lfloor R\rfloor \equiv P'$ and $\Gamma; c : \sigma \vdash_\ell P \xrightarrow{(\nu\vec{d}{:}\vec{n})\bar{a}\langle\vec{b}\rangle} R$

  The equivalence $\lfloor R\rfloor \equiv P$ is enough to show that $\lfloor\nu c{:}\sigma.R\rfloor \equiv \nu c@m.P'$ and the above reduction is the precondition the OPEN_TSEM rule needs to tell us that $\Gamma \vdash_\ell \nu c{:}\sigma.P \xrightarrow{(\nu\vec{d}{:}\vec{n},c{:}\sigma)\bar{a}\langle\vec{b}\rangle} R$, which finishes the case.

- STRUC_SEM:

$$\frac{\lfloor P' \rfloor \equiv P \quad \lfloor \Gamma \rfloor \ \vdash_\ell \ P \xrightarrow{\alpha} Q \quad Q \equiv Q'}{\lfloor \Gamma \rfloor \ \vdash_\ell \ \lfloor P' \rfloor \xrightarrow{\alpha} Q'}$$

A subtlety in this step is that the STRUC_SEM rule says that there exists an untyped process P that reduces to Q, however the induction hypothesis only holds for the erasures of a typed process. A simple lemma clears this up:

Lemma 1. If $\lfloor P' \rfloor \equiv P$ then there exists a typed process R such that $\lfloor R \rfloor = P$ and $R \equiv P'$.

Proof. Similar to Proposition 14. I show it holds for each structural congruence rule in turn and that this can be extended to any context. □

By this lemma $\lfloor \Gamma \rfloor \ \vdash_\ell \ \lfloor R \rfloor \xrightarrow{\alpha} Q$. So, applying the induction hypothesis we find that there exists some R' such that $\lfloor R' \rfloor \equiv Q$ and $\Gamma \vdash_\ell R \xrightarrow{\alpha} R'$. Observing that $R \equiv P'$, we can apply the STRUC_TSEM rule to show that: $\Gamma \vdash_\ell P' \xrightarrow{\alpha} R'$ and concluded by applying the transitivity of structure congruence to show that $\lfloor R' \rfloor \equiv Q'$.

□

I can now prove the second major theorem, the typed counterpart of Theorem 9, that a well typed system will never try to use a name outside its area and hence the typed semantics does indeed enforce the local area conditions.

Theorem 21. *For any well typed process P, if we can derive the sequence of transitions*

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

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

Proof. We apply Proposition 19 for each τ reduction. This shows that there exists a well typed P' such that $\Gamma \vdash_\ell P' \xrightarrow{\alpha} Q$. Proposition 20 then tells us that: $\lfloor \Gamma \rfloor \vdash_\ell \lfloor P' \rfloor \xrightarrow{\alpha} \lfloor Q \rfloor$. The result then follows from Proposition 8.

□

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.

4.4 Sorts

The aim of the type system presented above is to remove the need for some of the run time checks, while still keeping channels inside their areas. It also enforces a number of other conditions, such as making sure that all uses of the same channel have the same arity and that channels are used to transmit the same data types each time they are used. These properties are highly beneficial; essentially they guarantee that a process will be well behaved. In [Mil91] Milner conjectures that any realistic application of the π-calculus may be disciplined in this way. However, the type system also enforces a more problematic condition by limiting the types to be finite. Hence, a channel cannot pass itself, even indirectly, as this would require a recursive type.

To allow this more complex behaviour, while still removing the need for run-time checks, I have extended the notion of *sorts* [Mil91]. A *sort* is an identifier, each channel is assigned a sort and a *sorting* maps each sort to a level and the sorts that a channel may pass or a base type i.e. S is a sorting if:

$$\begin{array}{rl} S(sort) \mapsto & (sort_1, .., sort_n)@\ell \\ \text{or} & string \\ \text{or} & int \end{array}$$

Imagine that *carp* and *pike* want to exchange an arbitrary number of strings across a network and keep them ordered. One way to do this would be to send the name of a channel that would carry the next piece along with the data. This would lead to two sorts, *SEND* that is the sort of the channels used to send the data and *DATA* which is the sort of the data. The sorting would then map the sorts $SEND \mapsto (SEND, DATA)@net$ and $DATA \mapsto string$. Such a channel could not be given a finite type without knowing exactly how many pieces of data are going to be passed beforehand.

The only change needed to the syntax is to replace the type in the new name operator with a sort:

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

The sorting rules are given in Figure 4.3. The environments now include Γ which maps names to sorts and the sorting S which maps sorts to their level and the sorts they carry. The only change apart from this is to apply the sorting when finding the level of a name and what it carries. The semantic rules, given in Figure 4.4, also carry the sorting. The only difference from the typed semantic rules is a single use of the sorting to check the level of channels as they leave an area. The examples for the previous section can also be given sorts. For instance, the Inetd examples from Section 3.3 could be given the following sorts: $\{C, PIKE, SERVICE, RESPONSE, PRINT\}$ where the sorting makes the following mappings:

$$\begin{array}{rlrl} C & \mapsto string@net & PIKE & \mapsto (SERVICE, RESPONSE)@net \\ SERVICE & \mapsto (RESPONSE)@host & PRINT & \mapsto (string)@host \\ RESPONSE & \mapsto (string)@net & & \end{array}$$

The environment makes the following assignments:

$$\begin{array}{rlrl} c & : C & pike & : PIKE \\ finger & : SERVICE & daytime & : SERVICE \\ print & : PRINT & & \end{array}$$

The change from types to sorts effectively allows infinite types and so removes the ability to use induction on the type of a channel as a proof technique. Luckily none of the proofs of correctness of the type system uses induction on types, or relies on types being finite in any way.

NULL_S	$\Gamma, S \vdash_\ell 0$	
OUT_S	$\Gamma, S \vdash_\ell \bar{a}\langle\vec{b}\rangle$	$S(\Gamma(a)) = \vec{\sigma}@m$, $\Gamma(\vec{b}) = \vec{\sigma}$ and $\ell \leq m$
IN_S	$\dfrac{\Gamma; \vec{b} : \vec{\sigma}, S \vdash_\ell P}{\Gamma, S \vdash_\ell a(\vec{b}).P}$	$S(\Gamma(a)) = \vec{\sigma}@m$ and $\ell \leq m$
IN!_S	$\dfrac{\Gamma; \vec{b} : \vec{\sigma}, S \vdash_\ell P}{\Gamma, S \vdash_\ell !a(\vec{b}).P}$	$S(\Gamma(a)) = \vec{\sigma}@m$ and $\ell \leq m$
PAR_S	$\dfrac{\Gamma, S \vdash_\ell P \quad \Gamma, S \vdash_\ell Q}{\Gamma, S \vdash_\ell P \mid Q}$	
BIND_S	$\dfrac{\Gamma; a : \sigma, S \vdash_\ell P}{\Gamma, S \vdash_\ell \nu a{:}\sigma.P}$	
AREA_S	$\dfrac{\Gamma, S \vdash_\ell P}{\Gamma \vdash_m \ell[P]}$	$\ell <_1 m$

Figure 4.3: Type rules for sorted processes in the local area π-calculus

OUT_SSEM	$\Gamma, S \vdash_\ell \ \bar{a}\langle\vec{b}\rangle \xrightarrow{\bar{a}\langle\vec{b}\rangle} 0$	
IN_SSEM	$\Gamma, S \vdash_\ell \ a(\vec{b}).P \xrightarrow{a(\vec{b})} P$	$\vec{b} \cap dom(\Gamma) = \emptyset$
IN!_SSEM	$\Gamma, S \vdash_\ell \ !a(\vec{b}).P \xrightarrow{a(\vec{b})} P \,\vert\, !a(\vec{b}).P$	$\vec{b} \cap dom(\Gamma) = \emptyset$
PAR_SSEM	$\dfrac{\Gamma, S \vdash_\ell \ P \xrightarrow{\alpha} P'}{\Gamma, S \vdash_\ell \ P \,\vert\, Q \xrightarrow{\alpha} P' \,\vert\, Q}$	
COMM_SSEM	$\dfrac{\Gamma, S \vdash_\ell \ P \xrightarrow{(\nu\vec{d}:\vec{\theta})\bar{a}\langle\vec{c}\rangle} P' \quad \Gamma, S \vdash_\ell \ Q \xrightarrow{a(\vec{b})} Q'}{\Gamma, S \vdash_\ell \ P \,\vert\, Q \xrightarrow{\tau} \nu\vec{d}:\vec{\theta}.P' \,\vert\, Q'\{\vec{c}/\vec{b}\}}$	
BIND_SSEM	$\dfrac{\Gamma, S, a : \sigma \vdash_\ell \ P \xrightarrow{\alpha} P'}{\Gamma, S \vdash_\ell \ \nu a{:}\sigma.P \xrightarrow{\alpha} \nu a{:}\sigma.P'}$	$a \notin fn(\alpha)$
OPEN_SSEM	$\dfrac{\Gamma; c : \sigma, S \vdash_\ell \ P \xrightarrow{(\nu\vec{d}:\vec{\theta})\bar{a}\langle\vec{b}\rangle} P'}{\Gamma, S \vdash_\ell \ \nu c{:}\sigma.P \xrightarrow{(\nu\vec{d}:\vec{\theta},c:\sigma)\bar{a}\langle\vec{b}\rangle} P'}$	$c \neq a$ and $c \in \{\vec{b}\} \setminus dom(\Gamma)$
AREA_SSEM	$\dfrac{\Gamma, S \vdash_\ell \ P \xrightarrow{\alpha} P'}{\Gamma \vdash_m \ \ell[P] \xrightarrow{\alpha} \ell[P']}$	if α is $\bar{a}\langle\vec{b}\rangle$ or $a(\vec{b})$ then $m \leq S(\Gamma(a))$

Figure 4.4: Operational semantics for the sorted local area π-calculus

The erasure of a sort is as you would expect.

Definition 22. If $SORT$ is a sort and $S(SORT) = (\vec{SORT'})@\ell$ then its *erasure* $\lfloor SORT \rfloor$ is just the level ℓ. The sort system ensures that the level of data types ($string$, int) is never queried.

The correctness of the well sorted $la\pi$-calculus can be proved in exactly the same way as that of the well typed calculus.

Theorem 23. *For any well sorted process P, if we can derive the sequence of transitions*

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

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

Proof. Replace each application of Γ with an applications of $S.\Gamma$ in proofs Proposition 19 and Proposition 20. The result then follows from Proposition 8. $\square$

Chapter 5

Encoding the $la\pi$-calculus into the π-calculus

In this chapter, I examine the expressive power of the $la\pi$-calculus by comparing it to the asynchronous, polyadic π-calculus. I demonstrate that, perhaps surprisingly, I can encode local area communication into the π-calculus with conditional matching.

The main challenge in producing the encoding has been to find a way to prevent communication on a channel between different areas, while still preserving the identity of names. The solution is to replace communication inside an area with communication on a new channel created just for that area. The original communication channel name is then sent across this new channel, along with a name on which the original data can be found.

In a way, this replaces direct communication between two points with a system that broadcasts packets over an Ethernet. This has turned out to be a useful analogy and I will use the terminology of packet communication when motivating this encoding. In this case, the tree of nested areas (applications, machines, networks) gives rise to a hierarchy of ethers, with a process using a different ether for each level of communication (local to an application; within a machine; over the network).

In the next section I present each of the encoding rules. Section 5.2 returns to the examples from Chapter 3. I first demonstrate how a process is encoded. I then show how an encoded process reduces. There is a close match between the behaviour of a process in $la\pi$ and its π-calculus translation; in Section 5.3, I show a form of operational correspondence. However, there is some loss of information as translated terms may make additional silent moves, as packets pass over the ether, and an ether may indiscriminately accept and rebroadcast packets in which no receiver is interested.

5.1 Encoding Local Areas

The evident motivation for this model is packet communication on an Ethernet: instead of sending data directly to its destination, I drop a packet into the ether. Listening processes pick up all packets and sift out the ones they are interested in. Once a match has been found, the data is then transmitted. It might seem more natural to include the data to be transmitted in the packet, along with the channel name. But this is not possible in a polyadic setting, because the length of the tuple carried by the ether will vary for different key channels.

The exact form of the π-calculus I use is an asynchronous, polyadic, late version, similar to that used by Boudol [Bou92]. To this, I add, a conditional on names of the form: $\textit{if } a = b \textit{ then } P \textit{ else } Q$, as proposed by Parrow in *Handbook of Process Algebra* [BPS01]. Matching and mismatch are commonly used instead of the *if,then,else* syntax [VD98]. Although semantically equivalent, this would not be quite as tidy. I define recursive processes using explicit guarded recursion of the form $\mu X.P$, as used by Amadio and Boudol [ABL99], with the semantic rule:

$$\frac{P\{\mu X.P/X\} \xrightarrow{\alpha} P'}{\mu X.P \xrightarrow{\alpha} P'}$$

Using this form of recursion does not leave a litter of stopped processes and so helps to keep things tidy. It is straight forward to encode this kind of recursion using replication, see Turner [Tur96] for an example.

Structure:

$$[\![\Gamma \vdash_\ell 0]\!]_\Delta = 0$$

$$[\![\Gamma \vdash_\ell P \mid Q]\!]_\Delta = [\![\Gamma \vdash_\ell P]\!]_\Delta \mid [\![\Gamma \vdash_\ell Q]\!]_\Delta$$

$$[\![\Gamma \vdash_\ell m[P]]\!]_\Delta = \nu e.[\![\Gamma \vdash_m P]\!]_{\Delta, m \mapsto e} \qquad e \notin fn(P) \cup \mathrm{cod}(\Delta)$$

$$[\![\Gamma \vdash_\ell \nu a{:}\sigma.P]\!]_\Delta = \nu a.[\![\Gamma, a{:}\sigma \vdash_\ell P]\!]_\Delta$$

Actions, all with $e = \Delta(m)$ where $\Gamma(a) = \vec{\sigma}@m$:

$$[\![\Gamma \vdash_\ell \bar{a}\langle\vec{b}\rangle]\!]_\Delta = \nu d.(\bar{e}\langle a, d\rangle \mid \bar{d}\langle\vec{b}\rangle)$$

$$[\![\Gamma \vdash_\ell a(\vec{b}).P]\!]_\Delta = \mu X.e(x,d).\mathit{if}\ x = a\ \mathit{then}\ d(\vec{b}).[\![\Gamma, \vec{b}{:}\vec{\sigma} \vdash_\ell P]\!]_\Delta\ \mathit{else}\ (\bar{e}\langle x, d\rangle \mid X)$$

$$[\![\Gamma \vdash_\ell !a(\vec{b}).P]\!]_\Delta = \mu X.e(x,d).(X \mid \mathit{if}\ x = a\ \mathit{then}\ d(\vec{b}).[\![\Gamma, \vec{b}{:}\vec{\sigma} \vdash_\ell P]\!]_\Delta\ \mathit{else}\ \bar{e}\langle x, d\rangle)$$

Figure 5.1: Rules for encoding $la\pi$ into the π-calculus

In the $la\pi$-calculus names act as unique identifiers and also as conduits for data. This translation makes explicit these two different roles by mapping them to two distinct sets of names. Communications are identified by the first element of each packet, names like a and b and these names are never used to broadcast data; whereas ether names and data channels like e and d, are only ever used for communication. The scope of data names manages who knows what, while the scope of ether names handles locality of communication.

The tuple output $\bar{a}\langle\vec{b}\rangle$ becomes the outputs $\bar{e}\langle a, d\rangle$ where ether e varies according to the level of a and $\vec{b}$ is made available on the fresh name d. To keep track of which ether to use, I maintain an environment Δ mapping levels to ether names. The encoding is parameterised over this and takes the form

$$[\![\Gamma \vdash_\ell P]\!]_\Delta$$

where P is a well-typed term of level ℓ in context Γ and Δ assigns ethers to levels ℓ and above.

Figure 5.1 presents the full encoding, with one clause for each constructor. I will explain these in sequence.

The encoding leaves the null process, parallel composition and name restriction unchanged.

$$
\begin{aligned}
[\![\Gamma \vdash_\ell 0]\!]_\Delta &= 0 \\
[\![\Gamma \vdash_\ell P \mid Q]\!]_\Delta &= [\![\Gamma \vdash_\ell P]\!]_\Delta \mid [\![\Gamma \vdash_\ell Q]\!]_\Delta \\
[\![\Gamma \vdash_\ell \nu a{:}\sigma.P]\!]_\Delta &= \nu a.[\![\Gamma, a{:}\sigma \vdash_\ell P]\!]_\Delta
\end{aligned}
$$

To place a process in a local area, I create a new ether name and assign it a level in the environment Δ. A side condition ensures that I do not accidentally capture any existing names when introducing the new ether.

$$[\![\Gamma \vdash_\ell m[P]]\!]_\Delta = \nu e.[\![\Gamma \vdash_m P]\!]_{\Delta, m \mapsto e} \qquad e \notin \mathit{fn}(P) \cup \mathrm{cod}(\Delta)$$

Translating an output action uses the environment and the assignment of levels to ethers to find the correct ether for the output channel. It then sends the output channel over the ether name and follows it up with the data.

$$[\![\Gamma \vdash_\ell \bar{a}\langle\vec{b}\rangle]\!]_\Delta = \nu d.(\bar{e}\langle a, d\rangle \mid \bar{d}\langle\vec{b}\rangle) \qquad \text{with } e = \Delta(m) \text{ where } \Gamma(a) = \vec{\sigma}@m$$

An encoded input also uses the environment and the assignment of levels to ether names to find which ether it should listen on. When it receives a packet over this ether, it tests the first element to see if it matches the input channel name. If it does, then the packet is meant for this input, so the data is fetched and execution continues as appropriate. If the names do not match, then this packet is meant for some other channel in the same area. The packet is sent again and the process restarts.

$$
\begin{aligned}
[\![\Gamma \vdash_\ell a(\vec{b}).P]\!]_\Delta = \mu X.e(x, d).\mathit{if}\ x = a\ \mathit{then}\ d(\vec{b}).[\![\Gamma, \vec{b}{:}\vec{\sigma} \vdash_\ell P]\!]_\Delta\ \mathit{else}\ (\bar{e}\langle x, d\rangle \mid X) \\
\text{with } e = \Delta(m) \text{ where } \Gamma(a) = \vec{\sigma}@m.
\end{aligned}
$$

Replicated input is the same, except that the process restarts whether or not the input key is correctly matched.

$$
\begin{aligned}
[\![\Gamma \vdash_\ell !a(\vec{b}).P]\!]_\Delta = \mu X.e(x; \vec{b}).(X \mid \mathit{if}\ x = a\ \mathit{then}\ [\![\Gamma, \vec{b}{:}\vec{\sigma} \vdash_\ell P]\!]_\Delta\ \mathit{else}\ \bar{e}\langle x; \vec{b}\rangle) \\
\text{with } e = \Delta(m) \text{ where } \Gamma(a) = \vec{\sigma}@m.
\end{aligned}
$$

The encoding is well-defined up to structural congruence.

Proposition 24. *For any $la\pi$ terms P and Q, if $P \equiv Q$ then $[\![P]\!]_\Delta \equiv [\![Q]\!]_\Delta$.*

Proof. Because the encoding is compositional, it is enough to check that all of the structural axioms for $la\pi$ given in Section 3.1 translate to valid π-calculus equivalences. All of these are immediate. The only significant case is that $\ell[\nu a{:}\sigma.P] \equiv \nu a{:}\sigma.(\ell[P])$ becomes exchange of name binders $\nu e.\nu a.[\![P]\!]_\Delta \equiv \nu a.\nu e.[\![P]\!]_\Delta$ for some ether e. □

As noted above, this encoding uses π-calculus channels in a stereotyped manner. Names fall into three distinct classes: names like a and b, which correspond directly to $la\pi$ channels, ether names like e and the single use names that actually transmit the data. The $la\pi$ names are never used as channels, while ether names are never transmitted, nor do they appear in match tests.

The existence of this encoding raises the question of whether the local area syntax is really needed. Would it be feasible to use the π-calculus format at all times, as a kind of "programming style"? Doing so would have advantages; there is a massive body of work on analysing the π-calculus, all this could be brought directly to bare on a $la\pi$ term written in the π-calculus. This includes a number of type systems and software tools.

Unfortunately, it seems impractical to consider anything but the simplest $la\pi$ terms in their π encoded form, as can be seen from comparing a system and its encoding, Figures 3.2 and 5.3, for example. The π-calculus form is much more complicated, bigger systems will quickly become unreadable. A further problem is that processes are encoded with a mapping of levels to ether names and hence when composing two processes it is not enough simply to place them in parallel as it is in the plain π-calculus. For the process to behave as expected it is also necessary to make sure the ether names match for all levels above the level of encoding.

The most important aspect of the encoding process is that it introduces divergence. As any input may continually pick and drop outputs no, non trivial, process can ever be guaranteed to terminate. This will invalidate any analysis that attempts to account for silent steps, such as strong bi-simulations, or examining traces. It will, also make the problem of finding properties of systems using a model checker intractable.

For this reason, the encoding sheds light on how the calculus functions and, in some limited cases, may assist in the analysis of the calculus and systems written in it, but it does not replace the calculus.

Carbone and Maffeis, suggest an alteration of the π-calculus to allow $c$ompound names to be build from two other names [CM02]. For instance, the names a and b may be combined to make a name $a.b$ which can be used form input and output in the normal way. They point out, also in [CM02], that this allows a non-divergent encoding of the $la\pi$-calculus. Any analysis on the π-calculus with compound names would have direct relevance to the $la\pi$-calculus.

5.2 Example Encodings

There are two distinct aspects of this encoding, both of which I demonstrate. The first is, how the encoding actually works i.e. the way in which it turns a $la\pi$ process into a π process. The second is how an encoded process will then go on to behave.

5.2.1 Load Management System

To illustrate the process of encoding I will work through the encoding of the load management system, from Section 3.4.2 (page 48), step by step. This example starts off with the channels *load*, *print* and *link* in the environment. This encoding is summarised in Figure 5.2.

I start encoding at the *net* level so I need an ether name for net level communications. Define:

$$\Gamma = \{print : int@host, link : int@net, load : (int@host)@host\}$$
$$\Delta = \{net \mapsto n\}$$

$$\begin{aligned}
System \equiv \quad & \nu p.(Main \mid Load1) \\
& \nu q.(Probe \mid Load2)
\end{aligned}$$

$$\begin{aligned}
Main =& \nu main.\nu c.\nu d.(\bar{p}\langle load, d\rangle \mid \bar{d}\langle c\rangle) \mid \mu X.p(x,f).(if\ \ x = c\ then \\
& \qquad f(y).\mu Y.n(u,g).(if\ \ u = link\ then \\
& \qquad\quad g(z).\nu d.(\bar{p}\langle print, d\rangle \mid \bar{d}\langle y/z\rangle)\ else\ \bar{n}\langle u,g\rangle \mid Y) \\
& \quad else\ \bar{p}\langle x,f\rangle \mid X)
\end{aligned}$$

$$\begin{aligned}
Probe =& \nu probe.\nu c.(\nu d.(\bar{q}\langle load, d\rangle \mid \bar{d}\langle c\rangle) \mid \mu X.q(x,f).(\\
& \qquad if\ \ x = c\ then\ f(w).\nu g.(\bar{n}\langle link, g\rangle \mid \bar{g}\langle w\rangle)\ else\ \bar{q}\langle x,f\rangle \mid X)
\end{aligned}$$

$$\begin{aligned}
LoadMain =& \nu loadM.\mu X.p(x,f).(X \mid if\ \ x = load\ then \\
& \qquad f(v).\nu d.(\bar{p}\langle v,d\rangle . \mid \bar{d}\langle load1\rangle)\ else\ \bar{p}\langle x,v\rangle
\end{aligned}$$

$$\begin{aligned}
LoadProbe =& \nu loadP.\mu X.p(x,f).(X \mid if\ \ x = load\ then \\
& \qquad f(v).\nu d.(\bar{p}\langle v,d\rangle . \mid \bar{d}\langle load2\rangle)\ else\ \bar{p}\langle x,v\rangle
\end{aligned}$$

Figure 5.2: Encoding of the load management agents

The load management system can then be encoded with this environment and ether mapping.

$$[\![\Gamma \vdash_{net} host[Main \mid Load] \mid host[Probe \mid Load]]\!]_{\Delta}$$

This term is reduced using the composition encoding rule.

$$[\![\Gamma \vdash_{net} host[Main \mid Load]]\!]_{\Delta} \mid [\![\Gamma \vdash_{net} host[Probe \mid Load]]\!]_{\Delta}$$

The second step adds two new ether names for the host areas. So, defining Δ_M as $\{net \mapsto n, host \mapsto p\}$ and Δ_P as $\{net \mapsto n, host \mapsto q\}$. The encoding becomes:

$$\nu p.[\![\Gamma \vdash_{host} Main \mid Load]\!]_{\Delta_M} \mid \nu q.[\![\Gamma \vdash_{host} Probe \mid Load]\!]_{\Delta_P}$$

Next, I apply the composition rule two more times to get:

$$\begin{array}{l}\nu p.([\![\Gamma \vdash_{host} Main]\!]_{\Delta_M} \mid [\![\Gamma \vdash_{host} Load]\!]_{\Delta_M}) \\ \mid \nu q.([\![\Gamma \vdash_{host} Probe]\!]_{\Delta_P} \mid [\![\Gamma \vdash_{host} Load]\!]_{\Delta_P})\end{array}$$

I will now encode each of the four areas, individually.

- The $Probe$ agent:

  $$[\![\Gamma \vdash_{host} app[\nu c : int@host(\overline{load}\langle c\rangle \mid c(w).\overline{link}\langle w\rangle)]]\!]_{\Delta_P}$$

  I apply the area encoding rule to this application. This rule will create a new ether name even though it will never be used. The ether mapping is extended with a new name and that name is declared in the process.

  $$\nu probe.[\![\Gamma \vdash_{app} \nu c : int@host(\overline{load}\langle c\rangle \mid c(w).\overline{link}\langle w\rangle)]\!]_{\Delta_{PP}}$$

  Where $\Delta_{PP} = \{net \mapsto n, host \mapsto q, app \mapsto probe\}$. The new name c has to be added as a new name in the π term and its type has to be added into the environment.

  $$\nu probe.\nu c.[\![\Gamma; c : int@host \vdash_{app} \overline{load}\langle c\rangle \mid c(w).\overline{link}\langle w\rangle]\!]_{\Delta_{PP}}$$

Applying the composition rule again, this becomes:

$$\begin{aligned}\nu probe.\nu c.\quad & ([\![\Gamma; c : int@host \vdash_{app} \overline{load}\langle c\rangle]\!]_{\Delta_{PP}} \\ & |[\![\Gamma; c : int@host \vdash_{app} c(w).\overline{link}\langle w\rangle]\!]_{\Delta_{PP}})\end{aligned}$$

The output on *load* becomes a broadcast on the host's ether channel

$$\begin{aligned}\nu probe.\nu c.\quad & (\nu d.(\bar{q}\langle load, d\rangle \mid \bar{d}\langle c\rangle) \\ & |[\![\Gamma; c : int@host \vdash_{app} c(w).\overline{link}\langle w\rangle]\!]_{\Delta_{PP}})\end{aligned}$$

and the $la\pi$ input becomes a π process that listens for inputs.

$$\begin{aligned}\nu probe.\nu c.\quad & (\nu d.(\bar{q}\langle load, d\rangle \mid \bar{d}\langle c\rangle) \\ & \mid \mu X.q(x, f).(if\ \ x = c\ then \\ & \qquad\qquad\qquad f(w).[\![\Gamma; c : int@host \vdash_{app} \overline{link}\langle w\rangle]\!]_{\Delta_{PP}} \\ & \qquad\quad else\ \bar{q}\langle x, f\rangle \mid X)\end{aligned}$$

Finally, I encode the output on *link* as another packet broadcast to give us the true π-calculus term:

$$\begin{aligned}\nu probe.\nu c.\quad & (\nu d.(\bar{q}\langle load, d\rangle \mid \bar{d}\langle c\rangle) \\ & \mid \mu X.q(x, f).(if\ \ x = c\ then \\ & \qquad\qquad\qquad f(w).\nu g.(\bar{n}\langle link, g\rangle \mid \bar{g}\langle w\rangle) \\ & \qquad\quad else\ \bar{q}\langle x, f\rangle \mid X)\end{aligned}$$

- The $Main$ agent:

$$[\![\Gamma \ \vdash_{host}\ app[\nu c : int@host.(\overline{load}\langle c\rangle \mid c(y).link(z).\overline{print}\langle y/z\rangle)]]\!]_{\Delta_M}$$

The first five steps used to encode the *Main* agent are the same as the first five steps used to encode the *Probe* agent. That is, firstly an ether name is added for the *app* level, secondly, the new name c is created, thirdly the composition rule is applied, and finally, the output and input signals are encoded:

$$\begin{array}{ll} \nu main.\nu c. & (\nu d(\bar{p}\langle load, d\rangle \mid \bar{d}\langle c\rangle) \mid \mu X.p(x,f).(\\ & \mathit{if}\ x = c\ then \\ & \quad f(y).[\![\Gamma; c : int@host\ \vdash_{app} link(z).\overline{print}\langle y/z\rangle]\!]_{\Delta_{MM}} \\ & \ else\ \bar{p}\langle x,f\rangle \mid X) \end{array}$$

where $\Delta_{MM} = \{net \mapsto n, host \mapsto p, app \mapsto main\}$. The second input is then encoded to give:

$$\begin{array}{ll} \nu main.\nu c. & (\nu d.(\bar{p}\langle load, d\rangle \mid \bar{d}\langle c\rangle) \mid \mu X.p(x,f).(\\ & \mathit{if}\ x = c\ then \\ & \quad f(y).\mu Y.n(u,g).(\mathit{if}\ u = link\ then \\ & \qquad\quad g(z)[\![\Gamma; c : int@host\ \vdash_{app} \overline{print}\langle y/z\rangle]\!]_{\Delta_{MM}} \\ & \qquad else\ \bar{n}\langle u,g\rangle \mid Y) \\ & \ else\ \bar{p}\langle x,f\rangle \mid X) \end{array}$$

The final encoding step translates the output on *print* into a packet broadcast:

$$\begin{array}{ll} \nu main.\nu c. & \nu d.(\bar{p}\langle load, d\rangle \mid \bar{d}\langle c\rangle) \mid \mu X.p(x,f).(\\ & \mathit{if}\ x = c\ then \\ & \quad f(y).\mu Y.n(u,g).(\mathit{if}\ u = link\ then \\ & \qquad\quad g(z).\nu d.(\bar{p}\langle print, d\rangle \mid \bar{d}\langle y/z\rangle)\ else\ \bar{n}\langle u,g\rangle \mid Y) \\ & \ else\ \bar{p}\langle x,f\rangle \mid X) \end{array}$$

- The main *Load* agent:

$$[\![\Gamma\ \vdash_{host}\ app[!load(x).\bar{x}\langle LocalLoad\rangle]]\!]_{\Delta_P}$$

As with the other applications the first step of the encoding sequence is the generation of an ether name:

$$\nu loadM.[\![\Gamma\ \vdash_{app}\ !load(x).\bar{x}\langle LocalLoad\rangle]\!]_{\Delta_{ML}}$$

where $\Delta_{ML} = \{net \mapsto n, host \mapsto p, app \mapsto loadM\}$. The next step is to encode the replicated input.

$$\nu loadM.\mu X.p(x,f).(X \mid \mathit{if}\ x = load\ then\ f(v).[\![\Gamma\ \vdash_{app}\ \bar{v}\langle load1\rangle]\!]_{\Delta_{ML}}$$
$$else\ \bar{p}\langle x,v\rangle$$

Finally, the output is encoded.

$$\nu loadM.\mu X.p(x,f).(X \mid \mathit{if}\ x = load\ then\ f(v).\nu d.(\bar{p}\langle v,d\rangle. \mid \bar{d}\langle load1\rangle)$$
$$else\ \bar{p}\langle x,v\rangle$$

- The probe's *Load* agent. This is encoded in the same way as the main load agent. The difference is produced by encoding with an ether mapping that maps *host* to q rather than p.

5.2.2 The Inet Dæmon

To illustrate how an encoding reduces, I return to the `inetd` example from Section 3.4.1. Translating that example results in the process shown in Figure 5.3. The encoding uses three ethers, for which I take names n, p and q to cover the network, the server host *Pike* and the client host *Carp* respectively. All $la\pi$ channel names like *finger* map to themselves.

Figure 5.4 graphically represents the behaviour of the translated system. Grey bars indicate the local ethers. This differs from the representation of the unencoded example, given in Figure 3.3, which has direct links between processes. Reductions of the translated process closely match those of the original given earlier.

$$
\begin{array}{lll}
Carp' \mid Pike' & \equiv & (\ \nu q.\nu c.(\bar{n}\langle pike,d\rangle \mid \mu X.n(x,d)\ldots \\
& & \mid \nu p.(Inet' \mid Finger' \mid Daytime')\ ) \\
\text{extend scope} & \equiv & \nu p,q,c.(\ \bar{n}\langle pike,d\rangle \mid \mu X.n(x,d)\ldots \\
\text{of } p,\ q \text{ and } c & & \qquad \mid Inet' \mid Finger' \mid Daytime'\ ) \\
\text{unroll } Inet' & \equiv & \nu p,q,c.(\ \bar{n}\langle pike, finger, c\rangle \mid \mu X.n(x,d)\ldots \\
& & \qquad \mid n(x,d).(Inet' \mid \mathit{if}\ x = pike\ then\ d(s,r).\nu f.\ldots \\
& & \qquad \mid Finger' \mid Daytime'\ )
\end{array}
$$

$$[\![\, \Gamma \vdash_{net} Carp \mid Pike \,]\!]_{\{net \mapsto n\}} \;=\; Carp' \mid Pike'$$

$$\begin{aligned}
Carp' = \nu q.\nu c.(\ & \nu d.(\bar{n}\langle pike, d\rangle \mid \bar{d}\langle finger, c\rangle) \\
& \mid \mu X.n(x,d).if\ x = c\ then\ d(y).\nu f.(\bar{q}\langle print, f\rangle \mid \bar{f}\langle y\rangle) \\
& \qquad\qquad\qquad else\ (\bar{n}\langle x, d\rangle \mid X)\ )
\end{aligned}$$

$$Pike' = \nu p.(Inet' \mid Finger' \mid Daytime')$$

$$\begin{aligned}
Inet' &= \mu X.n(x,d).(X \mid if\ x = pike\ then\ d(s,r).\nu f.(\bar{p}\langle s, f\rangle \mid \bar{f}\langle r\rangle) \\
&\qquad\qquad else\ \bar{n}\langle x, d\rangle) \\
Finger' &= \mu X.p(x,d).(X \mid if\ x = finger\ then\ d(y).\nu f.(\bar{n}\langle y, f\rangle \mid \bar{f}\langle \text{“}PikeUsers''\rangle) \\
&\qquad\qquad else\ \bar{p}\langle x, d\rangle) \\
Daytime' &= \mu X.p(x,d).(X \mid if\ x = daytime\ then\ d(z)\nu f.(\bar{n}\langle y, f\rangle.\bar{f}\langle \text{“}PikeDate''\rangle) \\
&\qquad\qquad else\ \bar{p}\langle s, r\rangle)
\end{aligned}$$

Ether names: n, p, q

Data names: $pike, finger, daytime, print, c, r, s$

Figure 5.3: A π encoding of the $la\pi$ Internet server dæmon

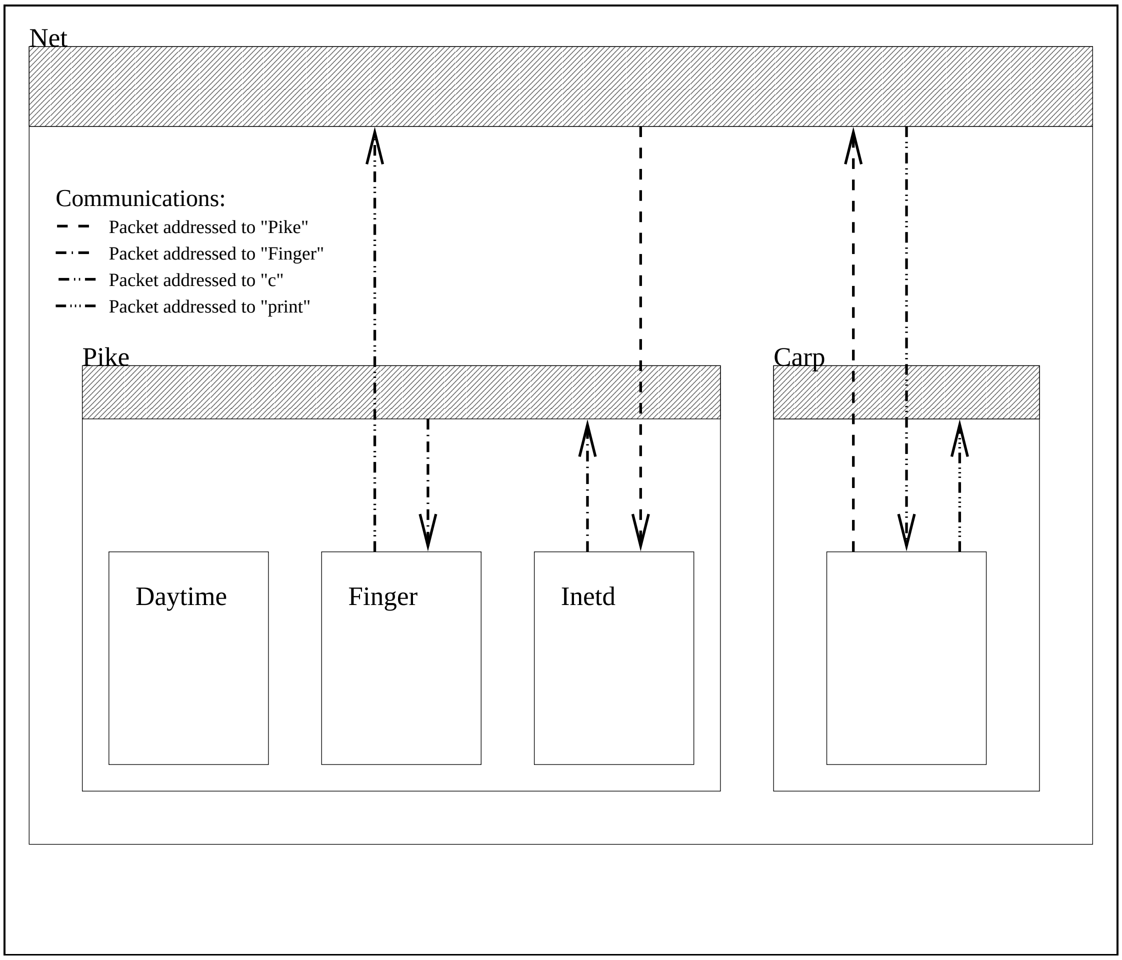


Figure 5.4: Ether-based encoding of `inet` dæmon relaying `finger` service

$$
\begin{array}{rl}
\text{communication} & \xrightarrow{\tau\tau} \nu p,q,c.(\ \mu X.n(x,d)\ldots \\
\text{of } \mathit{pike} \text{ over } n & \qquad\qquad \mid \nu f.(\bar{p}\langle finger, f\rangle \mid \bar{f}\langle c\rangle) \\
\text{and data over } d & \qquad\qquad \mid \mathit{Inet}' \mid \mathit{Finger}' \mid \mathit{Daytime}'\ ) \\
& \\
\text{unroll } \mathit{Finger}' & \equiv \nu p,q,c.(\ \mu X.n(x,d)\ldots \\
& \qquad\quad \mid \nu f.(\bar{p}\langle finger, f\rangle \mid \bar{f}\langle c\rangle) \\
& \qquad\quad \mid p(x,d).(\mathit{Finger}' \mid \mathit{if}\ \ x = \mathit{finger}\ \ \mathit{then}\ \ \bar{d}(y).\nu f.\ldots \\
& \qquad\quad \mid \mathit{Inet}' \mid \mathit{Daytime}'\ ) \\
& \\
\text{communication} & \xrightarrow{\tau\tau} \nu p,q,c.(\ \mu X.n(x,d).\mathit{if}\ \ x = c\ \ \mathit{then}\ \ d(y).\nu f.(\bar{q}\langle print, f\rangle\ldots \\
\text{of } \mathit{finger} \text{ over } p & \qquad\qquad \mid \nu f.(\bar{n}\langle c, f\rangle \mid \bar{f}\langle \text{``}\mathit{PikeUsers}''\rangle) \\
\text{and data over } f & \qquad\qquad \mid \mathit{Inet}' \mid \mathit{Finger}' \mid \mathit{Daytime}'\ ) \\
& \\
\text{communication} & \xrightarrow{\tau\tau} \nu p,q,c.(\ \nu f.(\bar{q}\langle print, f\rangle \mid \bar{f}\langle \text{``}PikeUsers''\rangle) \\
\text{of } c \text{ over } n & \qquad\qquad \mid \mathit{Inet}' \mid \mathit{Finger}' \mid \mathit{Daytime}'\ ) \\
\text{and data over } f &
\end{array}
$$

Comparison with the reduction given in Section 3.4.1, shows that communication restricted to a local area ("communication on *finger*@*host*") is replaced by communication on a local ether ("communication of *finger* over p").

Unlike the original $la\pi$ term, other reduction sequences are possible, though they will only add extra τ-transitions. For example, the $\mathit{Daytime}'$ server may mistakenly pick up the *finger* request, but will always immediately rebroadcast it.

5.3 Correctness of the Encoding

An $la\pi$ process and its encoding behave in very similar ways. For each step an $la\pi$ process may take its encoding can take two steps to match it. The six parts of the following proposition characterise exactly the possible actions of encoded processes.

Proposition 25. *For any well-typed process $\Gamma \vdash_\ell P$ in the local area π-calculus and any Δ, that maps all levels higher that ℓ to different names that do not appear in P, the following hold: In each case $Q = [\![\Gamma \vdash_\ell P]\!]_\Delta$ and $e = \Delta(m)$ where $\Gamma(a) = \vec{\sigma}@m$.*

1. *If $\Gamma \vdash_\ell P \xrightarrow{(\nu \vec{f}:\vec{n})\bar{a}\langle \vec{b}\rangle} P'$ then $Q \xrightarrow{(\nu d)\bar{e}\langle a,d\rangle} \xrightarrow{(\nu \vec{f})\bar{d}\langle \vec{b}\rangle} Q'$ where $Q' \equiv [\![\Gamma; \vec{f}{:}\vec{n} \vdash_\ell P']\!]_\Delta$*

2. *If $\Gamma \vdash_\ell P \xrightarrow{a(\vec{b})} P'$ then $Q \xrightarrow{e(x,d)} Q'$ such that $Q'\{a, f/x, d\} \xrightarrow{f(\vec{b})} Q''$ and for any vector of names $\vec{c}$ we have $Q''\{\vec{c}/\vec{b}\} \equiv [\![\Gamma' \vdash_\ell P'\{\vec{c}/\vec{b}\}]\!]_\Delta$.*

3. *If $\Gamma \vdash_\ell P \xrightarrow{\tau} P'$ then $Q \xrightarrow{\tau\tau} Q'$ where $Q' \equiv [\![\Gamma \vdash_\ell P']\!]_\Delta$.*

4. *If $Q \xrightarrow{(\nu \vec{f})\bar{e}\langle a,d\rangle} Q'$ then either $a \in \vec{f}$ or $Q' \xrightarrow{(\nu \vec{g})\bar{d}\langle \vec{b}\rangle} Q''$ and*
   $$\Gamma \vdash_\ell P \xrightarrow{(\nu \vec{g}:\vec{n})\bar{a}\langle \vec{b}\rangle} P' \text{ with } [\![\Gamma; \vec{g} : \vec{n} \vdash_\ell P']\!]_\Delta \equiv Q'' \text{ where } \Gamma(\vec{g}) = \vec{n}.$$

5. *If $Q \xrightarrow{e(x,d)} Q'$ then for all names a and f either $Q'\{a, f/x, d\} \equiv \bar{e}\langle a, f\rangle \mid Q$ or $\Gamma \vdash_\ell P \xrightarrow{a(\vec{b})} P'$ and $Q'\{a, f/x, d\} \xrightarrow{f(\vec{b})} Q''$ where for any vector of names $\vec{c}$ we have $Q''\{\vec{c}/\vec{b}\} \equiv [\![\Gamma \vdash_\ell P'\{\vec{c}/\vec{b}\}]\!]_\Delta$*

6. *If $Q \xrightarrow{\tau} Q'$ then either $Q \equiv Q'$ or $\Gamma \vdash_\ell P \xrightarrow{\tau} P'$ and $Q' \xrightarrow{\tau} Q''$ with*
   $$[\![\Gamma \vdash_\ell P']\!]_\Delta \equiv Q''.$$

This makes clear the close connection between $la\pi$ transitions and ether packets. For output, it is possible for actions that should be bound in the $la\pi$ term to be observed in the π term. These actions can be picked out by noticing that their output bindings include the first piece of data carried, i.e. the $la\pi$ channel name. Further, it is impossible for a translated $la\pi$ process to pick up these communications, as no process will be able to match the channel name. Apart from these packet snooping outputs, the correspondence is exact: a process can perform an output if and only if its translation can.

For input, an encoded process reads a packet; it tests this packet and if unsuitable it retransmits the packet again and continues as before. This means that an input in the encoded system may perform an output and revert to the original process. Alternatively the input will be matched by a similar input in the system it encodes, which is then followed up by the communication of the data.

This choice of two possible responses to any input action is carried over to the case of an encoded process performing a τ. This may either reflect a τ in the system it encodes or it may be a rejected communication, in which case the process to which it reduces is congruent to the original.

In the terminology of Nestmann and Pierce [NP96], this is an *operational correspondence* between the calculi. Transitions match, with the encoding sending a follow up packet some time later, except that a single internal π transition may map to zero $la\pi$ transitions. Unfortunately this does introduce the possibility of divergence: most translated terms can perform an unbounded sequence of τ steps as they collect and return ether packets. Divergence also arises in Nestmann and Pierce's choice encoding, except that there it is inserted by design, to give a more convenient full abstraction result; their initial encoding is divergence free. In my system, divergence arises rather naturally from the mechanism of ethers.

5.3.1 Evaluation Contexts

If we know what action a process can perform, we can derive a little about its structure, for instance if a process can output a value b on a channel a we know that somewhere inside P there must be an $\overline{a}\langle b\rangle$ action and furthermore there must be such an action that is not under a guard.

To formalise this I use a version of *Evaluation contexts* (these are known to other authors as reduction contexts [FF86], static contexts [Mil89], as well as evaluation contexts [FG98]). Evaluation contexts are the same as normal contexts, defined in Definition 11, except that the *hole* is not prefixed by any guard actions.

Definition 26. Evaluation Context

A $la\pi$ Evaluation Context is a process with a "hole" that does not appear under a guard. These can be built up using the following rules:

$$\begin{array}{rcl} C\langle\!\langle \_ \rangle\!\rangle & = & \ell[C\langle\!\langle \_ \rangle\!\rangle] \\ & | & C\langle\!\langle \_ \rangle\!\rangle \mid Q \\ & | & \nu a{:}\sigma.C\langle\!\langle \_ \rangle\!\rangle \\ & | & \_ \end{array}$$

A π Evaluation Context can be defined by dropping the $\ell[C\langle\!\langle \_ \rangle\!\rangle]$ and the type on the ν binder from the above definition.

Evaluation contexts are used in the main proof to break down a *la*π process into the part that performs the action and whatever surrounds it. I then encode these two parts so that the part that performs the action in the *la*π process encodes into the part that performs the equivalent action in the π process. It is useful to refer to particular properties of contexts. To avoid long-winded explanations I formalise them here:

Definition 27. Properties of evaluation contexts.

- The *context level of* C is the level at which the hole occurs.
- The *levels of* C is the set of all the levels of the agents in which the hole is nested.
- The *variables bound/free in* C are the variables which are bound/free by the context, at the hole.

Definition 28. Encoding of a context

laπ contexts can be encoded into π contexts by the encoding rules above and encoding a *la*π hole to a π hole: $[\![\Gamma \vdash_\ell \ \_ \ ]\!]_\Delta = \_$

I can now formally state my intuitions about actions and processes. The first result breaks down a process that can perform an output into an evaluation context and an output action.

Lemma 2. If $\Gamma \vdash_\ell P \overset{(\nu\vec{g}:\vec{\sigma})\overline{a}\langle\vec{b}\rangle}{\rightarrow} P'$ then there exists an evaluation context $C\langle\!\langle \_ \rangle\!\rangle$ such that $P = C\langle\!\langle \overline{a}\langle\vec{b}\rangle \rangle\!\rangle$ and $P' = C'\langle\!\langle 0 \rangle\!\rangle$, where $C' = C$ with the binders of $\vec{g}$ removed.

Proof. By induction on the derivation of: $\Gamma \vdash_\ell P \overset{(\nu\vec{g}:\vec{\sigma})\overline{a}\langle\vec{b}\rangle}{\rightarrow} P'$ $\square$

A process that performs an input action can be broken down in the same way. The only difference is the addition of the process that receives the input. To allow for this I have to make a minor change to the base case.

Lemma 3. If $\Gamma \vdash_\ell P \stackrel{a(\vec{b})}{\rightarrow} P'$ then there exists an evaluation context $C\langle\!\langle \_ \rangle\!\rangle$ and a process Q such that $P = C\langle\!\langle a(\vec{b}).Q \rangle\!\rangle$ and $P' = C\langle\!\langle Q \rangle\!\rangle$ with a free in C.

Proof. By induction on the derivation of: $\Gamma \vdash_\ell P \stackrel{a(\vec{b})}{\rightarrow} P'$

□

The next logical case to consider is that of a process that performs a τ. Here I break down such a process into two further processes, that perform an input and output and an evaluation context that surrounds them. The two lemmas above could then be used to break down these sub-processes and get at the actual guards that perform the actions. This proof is on the derivation of the silent, τ action, so the base case is that of two communication processes and the step cases consider any reduction rule that can pass up a τ action.

Lemma 4. If $\Gamma \vdash_\ell P \stackrel{\tau}{\rightarrow} P'$ then there exists an evaluation context $C\langle\!\langle \_ \rangle\!\rangle$ and processes R and T such that $P = C\langle\!\langle R \,|\, T \rangle\!\rangle$ and

$$\Gamma' \vdash_m R \stackrel{\overline{a}\langle\vec{b}\rangle}{\rightarrow} R' \qquad\qquad \Gamma' \vdash_m T \stackrel{a(\vec{c})}{\rightarrow} T'$$
$$P' = C\langle\!\langle R \,|\, T[\vec{b}/\vec{c}] \rangle\!\rangle$$

where m is the level of C and $\Gamma' = \Gamma \oplus$ *the variables bound in* C

Proof. By induction on the derivation of: $\Gamma \vdash_\ell P \stackrel{\tau}{\rightarrow} P'$, α-converting any binders of $\vec{b}$. □

I can also define evaluation contexts for the π-calculus by removing the area rule from the $la\pi$ definition. It is gratifying to note that the encoding would then translate evaluation $la\pi$ contexts to evaluation π contexts. As I will be encoding contexts, it is helpful to note that encoding is transitive over contexts, which I use in my final result.

Lemma 5. $[\![\ \Gamma \vdash_\ell C\langle\!\langle P \rangle\!\rangle]\!]_\Delta \equiv [\![\ \Gamma \vdash_\ell C]\!]_\Delta \ \langle\!\langle\ [\![\ \Gamma' \vdash_m P]\!]_{\Delta'} \rangle\!\rangle$

where Δ' is Δ extended with the assignment of new variables to the levels of C, m is the level of C and Γ' is Γ extended with any new names declared in C.

Proof. By induction on the structure of the context C.

□

5.4 The Proof of Correctness

In this section, I prove the main result of operational correspondence. The first three parts of this theorem show that for every possible action a $la\pi$ process can perform, its encoding can perform the equivalent packet action and reduce to an encoding of the new $la\pi$ process. The last three propositions consider the actions an encoded process can perform and how these are simulated in the $la\pi$ calculus. It is not necessary to consider all possible actions. A simple induction shows that:

Lemma 6. A π process that is an encoding of a $la\pi$ process can only perform a packet output: $\overline{e}\langle a, d\rangle$, a packet input: $e(a, d)$ or a communication: τ.

So, when showing that $la\pi$ processes simulate their encodings, it is only necessary to consider these packet communications.

The Propositions, 29 - 34, each make up part of Proposition 25 and so take P to be a well-typed local area π-calculus process,Δ to be a mapping of levels to ether names, $Q = [\![\Gamma \vdash_\ell P]\!]_\Delta$ and $e = \Delta(m)$ where $\Gamma(a) = \vec{\sigma}@m$.

5.4.1 Proof that $la\pi$ Processes are Simulated by Their Encodings

The proofs for each clause in this section follow a similar pattern. For clauses (i)–(iii), I break down a process into the part that performs the action and a surrounding context. Next, I use the encoding rules to encode these parts. Then I show how the encoding of the part of the $la\pi$ process that performs the action can perform a matching π action. Finally, I show that the encoding of the context allows this similar action to escape. There is a dependency, in that I must prove parts (i) and (ii) before (iii).

Proposition 29. *If* $\Gamma \vdash_\ell P \xrightarrow{(\nu \vec{f}:\vec{n})\bar{a}\langle\vec{b}\rangle} P'$ *then* $Q \xrightarrow{(\nu d)\bar{e}\langle a,d\rangle}\xrightarrow{(\nu \vec{f})\bar{d}\langle\vec{b}\rangle} Q'$ *where*

$$Q' \equiv [\![\Gamma; \vec{f}:\vec{n} \vdash_\ell P']\!]_\Delta.$$

Proof. Assume $\Gamma \vdash_\ell P \xrightarrow{(\nu \vec{f}:\vec{n})\bar{a}\langle\vec{b}\rangle} P'$. P is well typed and the level of P is less than or equal to $\Gamma(a)$. By Lemma 2, we know that there exists an evaluation context C such that:

$$P \equiv C\langle\!\langle \overline{a}\langle\vec{b}\rangle \rangle\!\rangle \quad , \quad \Gamma \vdash_\ell C\langle\!\langle \overline{a}\langle\vec{b}\rangle \rangle\!\rangle \overset{(\nu \vec{f}:\vec{n})\overline{a}\langle\vec{b}\rangle}{\rightarrow} C'\langle\!\langle 0 \rangle\!\rangle \;\; \textit{and} \;\; C'\langle\!\langle 0 \rangle\!\rangle \equiv P'$$

As P can output on a, a must be free in P therefore Δ must assign some name, e, to $\Gamma(a)$. Further, Q is the encoding of P therefore:

$$\begin{array}{rcll}
Q & \equiv & [\![\Gamma \vdash_\ell P]\!]_\Delta & \\
 & \equiv & [\![\Gamma \vdash_\ell C\langle\!\langle \overline{a}\langle\vec{b}\rangle \rangle\!\rangle]\!]_\Delta & \\
 & \equiv & [\![\Gamma \vdash_\ell C]\!]_\Delta \langle\!\langle [\![\overline{a}\langle\vec{b}\rangle]\!]_\Delta \rangle\!\rangle & \text{by Lemma 5} \\
 & \equiv & [\![\Gamma \vdash_\ell C]\!]_\Delta \langle\!\langle \nu d.\overline{e}\langle a, d\rangle \mid \bar{d}\langle\vec{b}\rangle \rangle\!\rangle & \text{by encoding rule} \\
 & \overset{(\nu d)\overline{e}\langle a,d\rangle}{\rightarrow} & [\![\Gamma \vdash_\ell C]\!]_\Delta \langle\!\langle \bar{d}\langle\vec{b}\rangle \rangle\!\rangle & \text{by semantic rules}
\end{array}$$

As the output on the ether name has opened the new name declarations of d we know that d must be free and unguarded in C. As encoding preserves free and bound names, we know that the bound names of C are also bound names of its encoding. So, by applying the semantic rules of the π-calculus we get:

$$\begin{array}{rcl}
[\![\Gamma \vdash_\ell C]\!]_\Delta \langle\!\langle \bar{d}\langle\vec{b}\rangle \rangle\!\rangle & \overset{(\nu \vec{f})\bar{d}\langle\vec{b}\rangle}{\rightarrow} & [\![\Gamma; \vec{f} : \vec{n} \vdash_\ell C']\!]_\Delta \langle\!\langle 0 \rangle\!\rangle \\
\text{and } \; [\![\Gamma; \vec{f} : \vec{n} \vdash_\ell C']\!]_\Delta \langle\!\langle 0 \rangle\!\rangle & \equiv & [\![\Gamma; \vec{f} : \vec{n} \vdash_\ell C'\langle\!\langle 0 \rangle\!\rangle]\!]_\Delta \equiv [\![\Gamma \vdash_\ell P']\!]_\Delta .
\end{array}$$

□

The second case, of input similarity, is only very slightly more complicated. It follows the same pattern as the above proof with the additional requirement to show that the two processes the inputs guard will also behave in the same way. Both calculi use a late semantics, i.e. the input value is only provided after the communication takes place. Therefore, I must show that those input guarded processes will behave in the same way for any value inputted.

Proposition 30. *If $\Gamma \vdash_\ell P \xrightarrow{a(\vec{b})} P'$ then $Q \xrightarrow{e(x,d)} Q'$ such that $Q'\{a, f/x, d\} \xrightarrow{f(\vec{b})} Q''$ and for any vector of names $\vec{c}$ we have $Q''\{\vec{c}/\vec{b}\} \equiv [\![\Gamma' \vdash_\ell P'\{\vec{c}/\vec{b}\}]\!]_\Delta$.*

Proof. Assume $\Gamma \vdash_\ell P \overset{a(\vec{b})}{\rightarrow} P'$ so, by Lemma 3, there exists an evaluation context C such that:

$$P = C\langle\!\langle a(\vec{b}).S \rangle\!\rangle \quad , \quad \Gamma \vdash_\ell C\langle\!\langle a(\vec{b}).S \rangle\!\rangle \overset{\overline{a}\langle\vec{b}\rangle}{\rightarrow} C\langle\!\langle S \rangle\!\rangle \;\; \textit{and} \;\; C\langle\!\langle S \rangle\!\rangle = P'$$

$$\begin{aligned}
Q &\equiv [\![\Gamma \vdash_\ell P]\!]_\Delta \\
&\equiv [\![\Gamma \vdash_\ell C\langle\!\langle a(\vec{b}).S\rangle\!\rangle]\!]_\Delta \\
&\equiv [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle [\![\Gamma' \vdash_m a(\vec{b}).S]\!]_{\Delta'}\rangle\!\rangle && \text{by Lemma 5} \\
&\equiv [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle \mu X.e(x,d).if\ (x=a)\ then && \text{by the encoding rule} \\
&\qquad d(\vec{b}).[\![\Gamma;\vec{b}{:}\vec{\sigma} \vdash_m S]\!]_{\Delta'}\ else\ (\overline{e}\langle x,d\rangle \mid X)\rangle\!\rangle && \text{for input}
\end{aligned}$$

$[\![\Gamma \vdash_\ell C]\!]_\Delta$ is an evaluation π-calculus context. P can input on a hence the level of a is greater than or equal to the level of P. Therefore, the level of a will be assigned a variable, e, in Δ and which is not bound in the encoding of C. So:

$$\begin{aligned}
Q \stackrel{e(x,d)}{\rightarrow}\ & [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle if\ (x=a)\ then\ d(\vec{b}).[\![\Gamma,\vec{b}{:}\vec{\sigma} \vdash_m S]\!]_{\Delta'}\ else \\
&\quad (\overline{e}\langle x,d\rangle \mid \mu X.if\ (x=a)\ then\ d(\vec{b}).[\![\Gamma,\vec{b}{:}\vec{\sigma} \vdash_m S]\!]_{\Delta'}\ else\ (\overline{e}\langle x,d\rangle \mid X))\rangle\!\rangle
\end{aligned}$$

Which gives us our Q'. We can now substitute a in for x and f in for d. Making the same observations as in the previous case, I continue the reduction.

$$\begin{array}{rcl}
Q'\{a,f/x,d\} & = & [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle f(\vec{b}).[\![\Gamma,\vec{b}{:}\vec{\sigma} \vdash_m S]\!]_{\Delta'}\rangle\!\rangle \\
& \xrightarrow{f(\vec{b})} Q'' \equiv & [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle [\![\Gamma,\vec{b}{:}\vec{\sigma} \vdash_m S]\!]_{\Delta'}\rangle\!\rangle \\
Q'' & \equiv & [\![\Gamma,\vec{b}{:}\vec{\sigma} \vdash_\ell C\langle\!\langle S\rangle\!\rangle]\!]_\Delta
\end{array}$$

Substituting $\vec{c}$ for $\vec{b}$ finishes the proof.

□

To prove that an internal communication in a $la\pi$-calculus process is matched by an internal communication in its encoding, I first use Lemma 2 to break the τ action into its constituent input and output parts. I then apply the above two propositions to these two parts to show how the encoding can perform the required action.

Proposition 31. *If* $\Gamma \vdash_\ell P \xrightarrow{\tau} P'$ *then* $Q \xrightarrow{\tau\tau} Q'$ *where* $Q' = [\![\Gamma \vdash_\ell P']\!]_\Delta$.

Proof. Assume $\Gamma \vdash_\ell P \stackrel{\tau}{\rightarrow} P'$ therefore, by Lemma 4, there exists a context C and $la\pi$ processes S and T such that $P \equiv C\langle\!\langle S \mid T\rangle\!\rangle$ and $\Gamma' \vdash_m S \stackrel{\overline{a}\langle\vec{b}\rangle}{\rightarrow} U$ and $\Gamma' \vdash_m T \stackrel{a(\vec{c})}{\rightarrow} V$ with $P' = C\langle\!\langle U \mid V\{\vec{b}/\vec{c}\}\rangle\!\rangle$.

$$
\begin{aligned}
Q &\equiv [\![\Gamma \vdash_\ell C\langle\!\langle T \mid S\rangle\!\rangle]\!]_\Delta \\
&\equiv [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle[\![\Gamma' \vdash_\ell T \mid S]\!]_{\Delta'}\rangle\!\rangle && \text{by Lemma 5} \\
&\equiv [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle[\![\Gamma' \vdash_\ell T]\!]_{\Delta'} \mid [\![\Gamma' \vdash_\ell S]\!]_{\Delta'}\rangle\!\rangle && \text{by encoding rule}
\end{aligned}
$$

We know from the two previous propositions that the encodings of T and S can perform inputs and outputs on the ether name for a. Matching these communications we get:

$$Q \xrightarrow{\tau} [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle T'\{a, d/x, d'\} \mid S'\rangle\!\rangle$$

The previous two propositions also tell us that T' and S' can perform further inputs and outputs on the channel d.

$$Q \xrightarrow{\tau\tau} Q'' \equiv [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle T''\{a, d, \vec{b}/x, d', \vec{c}\} \mid S''\rangle\!\rangle$$

There are no occurrences of d' or x free in T'', as these disappear once the data has been passed over d'. So I can drop the substitutions for these names. Finally, the last two propositions tell us that T'' and S'' are equivalent to the reduced $la\pi$ terms.

$$S'' \equiv [\![\Gamma' \vdash_m U]\!]_{\Delta'} \quad \textit{and} \quad T'' \equiv [\![\Gamma' \vdash_m V]\!]_{\Delta'}$$

and so:

$$
\begin{aligned}
Q' &\equiv [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle T''\{\vec{b}/\vec{c}\} \mid S''\rangle\!\rangle \\
&\equiv [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle[\![\Gamma' \vdash_m V]\!]_{\Delta'}\{\vec{b}/\vec{c}\} \mid [\![\Gamma' \vdash_m U]\!]_{\Delta'}\rangle\!\rangle \\
&\equiv [\![\Gamma \vdash_\ell C\langle\!\langle V \mid U\rangle\!\rangle]\!]_\Delta \\
&\equiv [\![\Gamma \vdash_\ell P']\!]_\Delta
\end{aligned}
$$

□

5.4.2 Proof that $la\pi$ Processes Simulate Their Encodings

Given $[\![\Gamma \vdash_\ell P]\!]_\Delta \equiv Q$ our proofs that P simulates Q all follow a similar pattern to the proofs of Q simulating P, but in the reverse direction. First I break down Q, into the parts that perform the action and a context, then using this decomposition I characterise P, finally I show that this P can perform the required action and reduce to a process matching the reduction of Q.

The output case is again the simplest. I make use of the fact that if the encoding can perform a packet output then there must be an output action somewhere in the original process. This could easily be proved by induction on the encoding rules. Then all that remains is to show that this action can be performed and that the resulting processes are also related.

Proposition 32. *If* $Q \stackrel{(\nu \vec{f})\bar{e}\langle a,d\rangle}{\longrightarrow} Q'$ *then either* $a \in \vec{f}$ *or* $Q' \stackrel{(\nu \vec{g})\bar{d}\langle \vec{b}\rangle}{\longrightarrow} Q''$ *and*
$\Gamma \vdash_\ell P \stackrel{(\nu \vec{g}:\vec{n})\bar{a}\langle \vec{b}\rangle}{\longrightarrow} P'$ *with* $[\![\Gamma; \vec{g} : \vec{n} \vdash_\ell P']\!]_\Delta \equiv Q''$ *where* $\Gamma(\vec{g}) = \vec{n}$.

Proof. Q is an encoding of P. Only an output action on a channel of the level assigned to e can be encoded to an output on e. Therefore there exists a context C such that $P = C\langle\!\langle \overline{a}(\vec{b})\rangle\!\rangle$ for some $\vec{b}$ and $\Delta(\Gamma(a)) = e$

As e is free in Q, e must be assigned a level by Δ that is greater than or equal to that of P and hence the level of a is greater than or equal to ℓ.

A binder in $la\pi$ is encoded into a binder in π so if a were bound in C it would also be bound in Q and hence in $\vec{f}$. Guards are also translated into guards. So, as the encoding of C is an evaluation context, C must also be an evaluation context. Therefore, $\Gamma \vdash_\ell P \stackrel{(\nu \vec{g})\overline{a}\langle \vec{b}\rangle}{\to} C\langle\!\langle 0\rangle\!\rangle$ and:

$$\begin{array}{lcll}
Q & \equiv & [\![\Gamma \vdash_\ell P]\!]_\Delta & \\
 & \equiv & [\![\Gamma \vdash_\ell C\langle\!\langle \bar{a}\langle \vec{b}\rangle\rangle\!\rangle]\!]_\Delta & \\
 & \equiv & [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle \nu d.\overline{e}\langle a, d\rangle.\bar{d}\langle \vec{b}\rangle\rangle\!\rangle & \text{Lemma 5 and encoding rule} \\
 & \stackrel{(\nu d)\overline{e}\langle a,d\rangle}{\to} & Q' \equiv [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle \bar{d}\langle \vec{b}\rangle\rangle\!\rangle & \text{by semantics of output} \\
Q' & \stackrel{(\nu \vec{g})\bar{d}\langle \vec{b}\rangle}{\to} & Q'' \equiv [\![\Gamma; \vec{g} : \vec{n} \vdash_\ell C'\langle\!\langle 0\rangle\!\rangle]\!]_\Delta & \text{by Lemma 5} \\
Q'' & \equiv & [\![\Gamma; \vec{g} : \vec{n} \vdash_\ell P']\!]_\Delta &
\end{array}$$

as required.

□

We do not get a strong bi-simulation result because an input action performed by an encoding may or may not represent an input action in the process it encodes. When an encoding process receives an input it checks the first argument to see if it is being passed over the right channel, if it is not it will send it back out and listen again. So, this leads to two possible cases for a process that performs an input.

Proposition 33. *If $Q \xrightarrow{e(x,d)} Q'$ then for all names a and f either:*

- $Q'\{a, f/x, d\} \equiv \bar{e}\langle a, f\rangle \mid Q$
  or
- $\Gamma \vdash_\ell P \xrightarrow{a(\vec{b})} P'$ *and* $Q'\{a, f/x, d\} \xrightarrow{f(\vec{b})} Q''$ *where for any vector of names* $\vec{c}$ *we have* $Q''\{\vec{c}/\vec{b}\} \equiv [\![\Gamma \vdash_\ell P'\{\vec{c}/\vec{b}\}]\!]_\Delta$

Proof. Q is an encoding of P and only an input action on a channel of the level assigned to e can produce an input action on e. So, there must exist a context C, process S and names $c, \vec{b}$ such that $P = C\langle\!\langle c(\vec{b}).S\rangle\!\rangle$ and $\Delta(\Gamma(a)) = e$ Where, for the same reasons as the last case, the encoding of C is an evaluation context that does not bind a.

$$
\begin{array}{rl}
Q & \equiv \quad [\![\Gamma \vdash_\ell P]\!]_\Delta \\
& \equiv \quad [\![\Gamma \vdash_\ell C\langle\!\langle c(\vec{b}).S\rangle\!\rangle]\!]_\Delta \\
& \equiv \quad [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle \mu X.e(x,d).\mathit{if}\ x = c\ \mathit{then}\ d(\vec{b}).[\![\Gamma, \vec{b}{:}\vec{\sigma} \vdash_\ell S]\!]_{\Delta'}\ \mathit{else}\ (\bar{e}\langle x, d\rangle \mid X) \\
& \xrightarrow{e(x,d)} \quad Q' \equiv [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle \mathit{if}\ x = c\ \mathit{then}\ d(\vec{b}).[\![\Gamma, \vec{b}{:}\vec{\sigma} \vdash_\ell S]\!]_{\Delta'}\ \mathit{else} \\
& \qquad (\bar{e}\langle x, d\rangle \mid \mu X.e(x,d).\mathit{if}\ x = c\ \mathit{then}\ d(\vec{b}).[\![\Gamma, \vec{b}{:}\vec{\sigma} \vdash_\ell S]\!]_{\Delta'}\ \mathit{else}\ (\bar{e}\langle x, d\rangle \mid X))
\end{array}
$$

So:

$$
\begin{array}{l}
Q'\{a, f/x, d\} \equiv [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle \mathit{if}\ a = c\ \mathit{then}\ f(\vec{b}).[\![\Gamma, \vec{b}{:}\vec{\sigma} \vdash_\ell S]\!]_{\Delta'}\ \mathit{else} \\
\qquad (\bar{e}\langle a, f\rangle \mid \mu X.e(x,d).\mathit{if}\ x = c\ \mathit{then}\ d(\vec{b}).[\![\Gamma, \vec{b}{:}\vec{\sigma} \vdash_\ell S]\!]_{\Delta'}\ \mathit{else} \\
\qquad\qquad (\bar{e}\langle x, d\rangle \mid X))
\end{array}
$$

where the internal d is not replaced with f because it is bound by the $e(x, d)$ input action. There are now two cases:

1. $c \neq a$ in which case:

$$
\begin{array}{rcl}
Q'\{a, f/x, d\} & \equiv & [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle (\bar{e}\langle a, f\rangle \mid \mu X.e(x,d).\ \mathit{if}\ x = c\ \mathit{then}\ d(\vec{b}).\ \ldots \\
& \equiv & [\![\Gamma \vdash_\ell C]\!]_\Delta\langle\!\langle \mu X.e(x,d).\mathit{if}\ x = a\ \mathit{then}\ d(\vec{b}).[\![\Gamma, \vec{b}{:}\vec{\sigma} \vdash_\ell S]\!]_{\Delta'} \\
& & \qquad \mathit{else}\ (\bar{e}\langle x, d\rangle \mid X)\rangle\!\rangle \mid \bar{e}\langle a, f\rangle \\
& \equiv & [\![\Gamma \vdash_\ell C\langle\!\langle a(\vec{b}).S\rangle\!\rangle]\!]_\Delta \mid \bar{e}\langle a, f\rangle \\
& \equiv & Q \mid \bar{e}\langle a, f\rangle
\end{array}
$$

2. $c = a$ in which case:

$$\begin{array}{lcl} Q' & \equiv & [\![\Gamma \vdash_\ell C]\!]_\Delta \langle\!\langle f(\vec{b}).[\![\Gamma, \vec{b}{:}\vec{\sigma} \vdash_\ell S]\!]_{\Delta'} \rangle\!\rangle \\ & \xrightarrow{f(\vec{b})} & Q'' \equiv [\![\Gamma \vdash_\ell C]\!]_\Delta \langle\!\langle [\![\Gamma, \vec{b}{:}\vec{\sigma} \vdash_\ell S]\!]_{\Delta'} \rangle\!\rangle \\ Q'' & \equiv & [\![\Gamma \vdash_\ell C\langle\!\langle S \rangle\!\rangle]\!]_\Delta \\ & \equiv & [\![\Gamma \vdash_\ell P']\!]_\Delta \end{array}$$

and so: $Q''\{\vec{c}/\vec{b}\} \equiv [\![\Gamma \vdash_\ell P'\{\vec{c}/\vec{b}\}]\!]_\Delta$.

□

In the above two proofs I have broken down the encoded process in the main proof instead of using a lemma, as I did in the proofs of encodings simulating their $la\pi$ processes. This is because the decomposition could be done quite simply. The case of breaking up an encoding process that performs a τ is slightly more complicated so I present it first as a lemma.

Given $Q \xrightarrow{\tau} Q'$ it is easy enough to break it down into a part that can perform some output action and another part that can perform the reciprocal input action. However, it is also necessary to show that these parts are also encodings of some $la\pi$ process, in order to allow us to use lemmas 3 and 2 to in turn decompose these parts.

Lemma 7. Given $Q \equiv [\![\Gamma \vdash_\ell P]\!]_\Delta$, if $Q \xrightarrow{\tau} Q'$ then there exist $la\pi$ processes P_1 and P_2, an evaluation context C and a π process M such that

$$P = C\langle\!\langle P_1 \mid P_2 \rangle\!\rangle$$

and

$$Q = \nu\vec{a}.([\![\Gamma' \vdash_m P_1]\!]_{\Delta'} \,|\, [\![\Gamma' \vdash_m P_2]\!]_{\Delta'} \mid M)$$

where Δ' is Δ extended with new variable assignments for the levels of C, m is the level of C, Γ' is Γ extended by the variables bound by C and $\vec{a}$ are the bound names shared between all processes. The encodings of P_1 and P_2 can then perform the following actions:

$$[\![\Gamma \vdash_m P_1]\!]_{\Delta'} \xrightarrow{\overline{e}\langle a,d\rangle} Q'_1 \quad \text{and} \quad [\![\Gamma \vdash_m P_2]\!]_{\Delta'} \xrightarrow{e(x,f)} Q'_2$$

Proof. By induction on the derivation of the τ action.

Base Case:

- COMM_TSEM: $P \equiv R \,|\, S$ therefore $Q \equiv [\![\Gamma \vdash_\ell R]\!]_\Delta \,|\, [\![\Gamma \vdash_\ell S]\!]_\Delta$. One of $[\![\Gamma \vdash_\ell R]\!]_\Delta$ and $[\![\Gamma \vdash_\ell S]\!]_\Delta$ performs an input and one an output.

  For simplicity, let us assume that $[\![\Gamma \vdash_\ell S]\!]_\Delta$ performs the output and $[\![\Gamma \vdash_\ell R]\!]_\Delta$ the input. Lemma 6 states that the only output action that can be performed by S is of the form $\overline{e}\langle a, d\rangle$ and that $e(\alpha, \vec{\delta})$ are the only possible input actions for R. Therefore:

  $$[\![\Gamma \vdash_\ell S]\!]_\Delta \overset{\overline{e}\langle a,d\rangle}{\rightarrow} Q_1' \quad \text{and} \quad [\![\Gamma \vdash R]\!]_\Delta \overset{e(\alpha,f)}{\rightarrow} Q_2'$$

  and the surrounding context contains any necessary binders.

Step Cases:

- AREA_TSEM: $P \equiv n[R]$

  Therefore $Q \equiv \nu e.[\![\Gamma \vdash_n R]\!]_{\Delta \oplus \{\ell \mapsto e\}}$. So by the induction hypothesis there exists an evaluation context D and $la\pi$ processes R_1 and R_2 and a π process M such that:

  $$R \equiv D\langle\!\langle R_1 \,|\, R_2 \rangle\!\rangle$$
  $$\text{and}$$
  $$[\![\Gamma \vdash_m R]\!]_{\Delta \oplus \{\ell \mapsto e\}} \equiv \nu\vec{a}.([\![\Gamma' \vdash_m R_1]\!]_{\Delta' \oplus \{\ell \mapsto e\}} \,|\, [\![\Gamma' \vdash_m R_2]\!]_{\Delta' \oplus \{\ell \mapsto e\}} \,|\, M)$$
  $$[\![\Gamma' \vdash_m R_1]\!]_{\Delta' \oplus \{\ell \mapsto e\}} \overset{\overline{e}\langle a,d\rangle}{\rightarrow} Q_1' \text{ and } [\![\Gamma' \vdash_m R_2]\!]_{\Delta' \oplus \{\ell \mapsto e\}} \overset{e(x,f)}{\rightarrow} Q_2'$$

  so $Q \equiv \nu e.\nu\vec{a}.([\![\Gamma' \vdash_m R_1]\!]_{\Delta' \oplus \{\ell \mapsto e\}} \,|\, [\![\Gamma' \vdash_m R_2]\!]_{\Delta' \oplus \{\ell \mapsto e\}} \,|\, M)$

  and we can define $C \equiv m[D]$ so that $P = C\langle\!\langle R_1 \,|\, R_2 \rangle\!\rangle = m[D\langle\!\langle R_1 \,|\, R_2 \rangle\!\rangle]$

- BIND_TSEM: $P \equiv \nu a{:}\sigma.R$.

  In this case $Q \equiv \nu a.[\![\Gamma; a{:}\sigma \vdash_\ell R]\!]_\Delta$

  So by the induction hypothesis there exists an evaluation context D and $la\pi$ processes R_1 and R_2 and a π process M such that:

$$R \equiv D\langle\!\langle R_1 \mid R_2 \rangle\!\rangle$$
$$\text{and}$$
$$[\![\Gamma; a{:}\sigma \vdash_m R]\!]_\Delta \equiv \nu\vec{a}.([\![\Gamma\ ; a : \sigma \vdash_m R_1]\!]_{\Delta'} \mid [\![\Gamma'; a : \sigma \vdash_m R_2]\!]_{\Delta'} \mid M)$$
$$[\![\Gamma'; a : \sigma \vdash_m R_1]\!]_{\Delta'} \overset{\overline{e}\langle a,d\rangle}{\rightarrow} Q_1' \text{ and } [\![\Gamma'; a : \sigma \vdash_m R_2]\!]_{\Delta'} \overset{e(x,f)}{\rightarrow} Q_2'$$

so $Q \equiv \nu a.\nu\vec{a}.([\![\Gamma'; a : \sigma \vdash_m R_1]\!]_{\Delta'} \mid [\![\Gamma'; a : \sigma \vdash_m R_2]\!]_{\Delta'} \mid M)$

and we can define $C \equiv \nu a{:}\sigma.D$ so that $P = C\langle\!\langle R_1 \mid R_2 \rangle\!\rangle = \nu a{:}\sigma.D\langle\!\langle R_1 \mid R_2 \rangle\!\rangle]$

- PAR_TSEM: $P \equiv R \mid S$ therefore $Q \equiv [\![\Gamma \vdash_\ell R]\!]_\Delta \mid [\![\Gamma \vdash_\ell S]\!]_\Delta$.

  Now, either $[\![\Gamma \vdash_\ell R]\!]_\Delta$ performs a τ or $[\![\Gamma \vdash_\ell S]\!]_\Delta$ performs a τ. In the first case we apply the induction hypothesis. This tells us that there exists an evaluation context D and $la\pi$ processes R_1 and R_2 and a π process M such that:

$$R \equiv D\langle\!\langle R_1 \mid R_2 \rangle\!\rangle$$
$$\text{and}$$
$$[\![\Gamma \vdash_\ell R]\!]_{\Delta'} \equiv \nu\vec{a}.([\![\Gamma' \vdash_m R_1]\!]_{\Delta'} \mid [\![\Gamma' \vdash_m R_2]\!]_{\Delta'} \mid M)$$
$$[\![\Gamma' \vdash_m R_1]\!]_{\Delta'} \overset{\overline{e}\langle a,d\rangle}{\rightarrow} Q_1' \text{ and } [\![\Gamma' \vdash_m R_2]\!]_{\Delta'} \overset{e(x,f)}{\rightarrow} Q_2'$$

  and so:

$$\begin{aligned} Q &\equiv \nu\vec{a}.([\![\Gamma' \vdash_m R_1]\!]_{\Delta'} \mid [\![\Gamma' \vdash_m R_2]\!]_{\Delta'} \mid M) \mid [\![\Gamma \vdash_\ell S]\!]_\Delta \\ &\equiv \nu\vec{a}.([\![\Gamma' \vdash_m R_1]\!]_{\Delta'} \mid [\![\Gamma' \vdash_m R_2]\!]_{\Delta'} \mid (M \mid [\![\Gamma \vdash_\ell S]\!]_\Delta)) \end{aligned}$$

  and we can define $C \equiv D \mid [\![\Gamma \vdash_\ell S]\!]_\Delta$ so that $P = C\langle\!\langle R_1 \mid R_2 \rangle\!\rangle = D\langle\!\langle R_1 \mid R_2 \rangle\!\rangle \mid [\![\Gamma \vdash_\ell S]\!]_\Delta$

  The second case is symmetric.

The semantic rules for input, replicated input, output and null cannot be part of the derivation of a τ so they are not considered. $\square$

So, given an encoded process that performs a τ we can use the above lemma to break it into two parts that perform an input and output. We can then use Lemmas 3 and 2, to show how the $la\pi$ processes these parts encode perform similar actions. Finally we recombine these $la\pi$ parts to finish the proof.

Proposition 34. *If $Q \xrightarrow{\tau} Q'$ then either*

- $Q \equiv Q'$, *so trivially* $[\![\Gamma \vdash_\ell P]\!]_\Delta \equiv Q'$
  or

- $\Gamma \vdash_\ell P \xrightarrow{\tau} P'$ *and* $Q' \xrightarrow{\tau} Q''$ *with* $[\![\Gamma \vdash_\ell P']\!]_\Delta \equiv Q''$.

Proof. Lemma 7 tells us that we can find the subprocesses of Q and P that perform the input and output actions that make up the τ communication and that the behaviour of these two parts will correspond i.e. there exist π processes Q_1, Q_2 and M such that

$$Q = \nu\vec{a}.(Q_1 \mid Q_2 \mid M)$$
$$Q_1 \overset{(\nu\vec{f})\overline{e}\langle a,d\rangle}{\rightarrow} Q_1' \text{ and } Q_2 \overset{e(\alpha,\delta)}{\rightarrow} Q_2'$$
$$\text{so } Q \xrightarrow{\tau} \nu\vec{a}.(Q_1' \mid Q_2'\{a,d/\alpha,\delta\} \mid M)$$

and Q_1 and Q_2 are encodings of evaluation subprocesses of P, P_1 and P_2 i.e. there exists an evaluation context C such that for Δ' that extends Δ with new variable assignments for the levels of C, m that is the level of C and Γ' that extends Γ with the variables bound by C:

$$Q_1 \equiv [\![\Gamma' \vdash_m P_1]\!]_{\Delta'} \text{ and } Q_2 \equiv [\![\Gamma' \vdash_m P_1]\!]_{\Delta'}$$
$$\text{and } P \equiv C\langle\!\langle P_1 \mid P_2 \rangle\!\rangle$$

So, by Proposition 32, we know that either $a \in \vec{f}$ or $Q_1' \overset{(\nu\vec{g})\bar{d}\langle\vec{b}\rangle}{\longrightarrow} Q_1''$ and $\Gamma \vdash_\ell P_1 \overset{(\nu\vec{g}:\vec{n})\bar{a}\langle\vec{b}\rangle}{\longrightarrow} P_1'$ with $[\![\Gamma \vdash_\ell P_1']\!]_\Delta \equiv Q_1''$ and $\Gamma(\vec{g}) = \vec{n}$. And by Proposition 33 either

- (1) $Q_2'\{a,d/\alpha,\delta\} \equiv \bar{e}\langle a,d\rangle \mid Q_2$
  or

- (2) $\Gamma \vdash_\ell P_2 \xrightarrow{a(\vec{\beta})} P_2'$ and $Q_2'\{a, d/\alpha, \delta\} \xrightarrow{d(\vec{\beta})} Q''$ where for any vector of names $\vec{c}$ we have $Q_2''\{\vec{c}/\vec{b}\} \equiv [\![\Gamma \vdash_\ell P_2'\{\vec{c}/\vec{b}\}]\!]_\Delta$

If the packet key a is bound in the output, i.e. in $\vec{f}$ then the process Q_2' cannot know about this name and so the packet cannot pass the input test. So, Q_2' will rebroadcast the packet. So, $a \in \vec{f} \implies Q_2'\{a, d/\alpha, \delta\} \equiv \bar{e}\langle a, d\rangle \,|\, Q_2$. So, it is enough to consider the two possible outcomes of Proposition 33.

- (1) $Q_2'\{a, d/\alpha, \delta\} \equiv \bar{e}\langle a, d\rangle \,|\, Q_2$, In this case, as Q_1 and Q_2 are asynchronous π-calculus terms we know that:

$$Q_1 \xrightarrow{\bar{e}\langle a,d\rangle} Q_1' \text{ therefore } Q_1 \equiv Q_1' \,|\, \bar{e}\langle a, d\rangle \text{ (i)}$$

$$\begin{array}{lll} Q \xrightarrow{\tau} & Q' \equiv \nu\vec{a}.\nu\vec{f}.(Q_1' \,|\, Q_2'\{a, d/\alpha, \delta\} \,|\, M) & \text{by Lemma 7} \\ \equiv & \nu\vec{a}.\nu\vec{f}.(Q_1' \,|\, Q_2 \,|\, \bar{e}\langle a, d\rangle \,|\, M) & \\ \equiv & \nu\vec{a}.(Q_1 \,|\, Q_2 \,|\, M) & \text{by (i)} \\ \equiv & Q & \text{by Lemma 7} \end{array}$$

- (2) In the second case, $\Gamma' \vdash_m P_2 \xrightarrow{a\langle\vec{\beta}\rangle} P_2'$ which means that $\Gamma' \vdash_m P_1 \,|\, P_2 \xrightarrow{\tau} P_1' \,|\, P_2'[\vec{b}/\vec{\beta}]$. And as C is an evaluation context: $\Gamma \vdash_\ell C\langle\!\langle P_1 \,|\, P_2 \rangle\!\rangle \xrightarrow{\tau} C\langle\!\langle P_1' \,|\, P_2'[\vec{b}/\vec{\beta}]\rangle\!\rangle$

$$\begin{array}{ll} Q' & \equiv \nu\vec{a}.\nu\vec{f}.(Q_1' \,|\, Q_2'\{a, d/\alpha, \delta\} \,|\, M) \\ Q' & \xrightarrow{\tau} Q'' \equiv \nu\vec{a}.\nu\vec{f}.\nu\vec{g}.(Q_1'' \,|\, Q_2''\{a, d, \vec{b}/\alpha, \delta, \vec{\beta}\} \,|\, M) \\ & \qquad\qquad\qquad \text{by reduction of } Q_1' \;\&\; Q_2' \\ Q'' & \equiv \nu\vec{a}.\nu\vec{f}.\nu\vec{g}.(Q_1'' \,|\, Q_2''\{\vec{b}/\vec{\beta}\} \,|\, M) \\ & \qquad\qquad\qquad \text{remove unused names.} \end{array}$$

and by Propositions 33 and 32, Q_1'' and Q_2'' are the encodings of P_1' and P_2'.

$$\begin{array}{ll} Q'' & \equiv \nu\vec{a}.\nu\vec{f}.\nu\vec{g}.([\![\Gamma' \vdash_m P_1']\!]_\Delta' \,|\, [\![\Gamma'' \vdash_m' P_1']\!]_\Delta''\{\vec{b}/\vec{\beta}\} \,|\, M) \\ Q'' & \equiv [\![\Gamma \vdash_\ell P']\!]_\Delta \end{array}$$

□

Chapter 6

Modelling FTP over NAT

My main reason for investigating local areas is the belief that they are natural notions with which to model any distributed systems that make use of conventions.

In this chapter, I offer evidence for this by presenting a large-scale model of Internet communication and use it to examine the liveness properties of File Transfer Protocol in the presence of Network Address Translation. Specifically I investigate why and how the application may fail to make the required connection.

In the next section, I introduce an alternative syntax for the $la\pi$-calculus that aims to make larger processes easy to read. The first model is of Internet communication based on sockets and TCP over IP [CK74]. This gives me a reusable framework in which to carry out further investigations. Next, I extend the Internet communication model with Network Address Translation [EF94]. In Section 6.3 I build a model of an FTP client and server that uses the TCP/IP model to communicate. The results of running this model are given in Section 6.5. They show how FTP can sometimes fail when used with NAT and how the steps taken to correct the problems are successful. In the final Section, 6.6, I look at one of the ways in which the model may be extended.

6.1 Program Syntax

The models in this chapter are significantly larger than those presented earlier. The syntax of the $la\pi$-calculus, like that of the π-calculus is wonderfully concise when it comes to describing small systems but can become somewhat unreadable with anything larger than half a side of A4. To remedy this I use a pinch of syntactic sugar to make the calculus clearer. I refer to this new form as *program syntax*. This is similar to the form used for the spi calculus from [GJ01].

Process P,Q ::= a list of terms separated by ";"

Term	::=	**in**	$a(\vec{b})$	input tuple
	$\mid$	**out**	$a\langle\vec{b}\rangle$	output tuple
	$\mid$	**new**	$a{:}\sigma$	fresh channel
	$\mid$	**spawn**	{**in** $a(\vec{b})$ } { Q }	spawn new thread
	$\mid$	**spawn**	{**in** $a(\vec{b})$ } **repeat** { Q }	spawn new persistent thread
	$\mid$	**area**	$\ell\{Q\}$	local area at level ℓ

Input on a name is written as: **in** $a(\vec{b})$ and output **out** $a\langle\vec{b}\rangle$. A subtlety here is that the input is a guard and so is a prefix to a continuation P whereas the output is asynchronous and does not block the process P and so it is in parallel with P. This gives ";" a dual meaning: when following an input it represents a prefix, and when following an output it represents a parallel composition. Indeed, the fact that asynchronous output could be modelled in this way was one of the original observations behind asynchronous π-calculus. The ";" also represents prefixing in the new name declaration: " **new** $a{:}\sigma$ ".

The **spawn** {**in** $a(\vec{b})$ } { Q } ; P command starts a parallel process. This can be interpreted as: spin off a parallel process which will perform Q when triggered by an input on a. While waiting for this trigger, the process continues with the "program" P. This command really has two uses, both of which relate to the use of parallel, input guarded processes. Firstly, it can be used to spin off a thread to perform a given job allowing the main program to continue. This is exactly what is done with servers that will spin off a new thread to deal with each connection.

$[\![$ **<u>in</u>** a $(\vec{b})$;P $]\!]$	$\equiv$	$a(\vec{b}).[\![P]\!]$
$[\![$ **<u>out</u>** a $(\vec{b})$;P $]\!]$	$\equiv$	$\overline{a}\langle\vec{b}\rangle \mid [\![P]\!]$
$[\![$ **<u>new</u>** a **<u>at</u>** σ;P $]\!]$	$\equiv$	$\nu a : \sigma.P$
$[\![$ **<u>spawn</u>** { α }{ Q };P $]\!]$	$\equiv$	$[\![\alpha.Q]\!] \mid [\![P]\!]$
$[\![$ **<u>spawn</u>** { α } **<u>repeat</u>**{ Q };P $]\!]$	$\equiv$	$[\![!\alpha.Q]\!] \mid [\![P]\!]$
$[\![$ **<u>area</u>** ℓ { Q } ;P $]\!]$	$\equiv$	$\ell[[\![Q]\!]] \mid [\![P]\!]$
$[\![$ empty list $]\!]$	$\equiv$	0

Figure 6.1: Rules for translating program syntax into standard $la\pi$syntax

Secondly, it can be used as a kind of switch statement. Where a name can be compared to a number of others by outputting it, while spinning off processes to listen and act on each of its possible values.

Adding **<u>repeat</u>** after the **<u>spawn</u>** command will make this a persistent process that will keep going, after it gets the first trigger. Finally, areas are declared using the key word **<u>area</u>** followed by the process for that area in curly brackets.

The translation between the $la\pi$ syntax used in previous chapters and the program syntax is given in Figure 6.1.

6.1.1 Channel Coercion

Polyadic channels can be used to pass a myriad of other channels. It is often the case that I am only interested in the first few. Likewise, it is often the case that I wish to communicate with a remote process in the knowledge that only the first few values of the tuple passed will have an effect on the resulting process.

To avoid cluttering the process terms, in this extended example, I introduce a shorthand, referred to here as channel coercion. This allows me to leave out unnecessary channel names. If, for example, I have a channel name, a, which carries five other names but I am only interested in the first three then I write "**<u>in</u>** a (b_1, b_2, b_3);P " for "$a(b_1, b_2, b_3, c_1, c_2).[\![P]\!]$" where c_1 and c_2 do not occur in P.

Formalising this for the general case of well sorted processes, I get:

$$[\![\ \underline{\textbf{in}}\quad \text{a}\ (\vec{b});\text{P}\ ]\!] \ \equiv\ a(\vec{b},\vec{c}).[\![P]\!]$$

where $S(a) = (S(\vec{b}), S(\vec{c}))$ extending S with $\vec{c}$ if neseccary
and $\vec{c} \cap fn(P) = \emptyset$

I can do the same thing with output actions but must first declare the channels I am going to use as padding.

$$[\![\ \underline{\textbf{out}}\quad \text{a}\ (\vec{b});\text{P}\ ]\!] \ \equiv\ \nu\vec{c}{:}SORTS.\bar{a}\langle\vec{b},\vec{c}\rangle.[\![P]\!]$$

where $S(a) = (S(\vec{b}), SORTS)$
and $\vec{c} \cap fn(P) = \emptyset$

I will also use brackets to mark out natural groups inside tuples. These brackets are only used to make the longer tuples more readable and have no semantic meaning in themselves. For example, the following IP packet consists of a destination IP address and port, a source IP address and port and a data payload:

$$(155.246.7.5\ ,\ 129.215.98.54\ ,\ 21\ ,\ 3022\ ,\ (port\ ,\ 129.215.98.54\ ,\ 3021))$$

This is a tuple with seven elements but brackets are placed round the pay load to make it easier to read.

Finally, when I am going to ignore a name received on an input, I will write the channel name it is to be bound to as "_", to avoid cluttering processes with unused names.

6.2 The Internet Communication Model

Internet communication is built on a four level model. At the base is the link level of real physical connections. On top of this the second level, the network level, provides basic communication. This is done using *Internet Protocol* (IP)

[Pos81a] to transmit small packets of data between hosts. The use of packets to communicate has many advantages. It makes multiplexing easy and it improves the resilience of a network. If part of the network breaks, the packets can simply take a different route to avoid the fault. However, applications often require a steady stream of data. Packets may also be lost, or accidentally replicated. This is where the third level of Internet communication comes in. This level implements another protocol on top of IP that will provide user friendly connections to the applications. While use of Internet Protocol is compulsory at the second level, the third level may use a number of protocols. The most common is Transmission Control Protocl [Pos81b] but User Datagram Protocol [Pos80] is not uncommon. The last level in the Internet communication model is the application level; this is the natural habitat of processes that may wish to use the connections provided by the three other levels.

The primary aim of the model presented in this chapter is to establish liveness properties of single runs of the FTP protocol showing that connections are correctly made or that data is lost. To this end, I have simplified the systems I am looking at, removing features that are not relevant. For instance, my routers do not lose packets as real life routers may do, therefore I do not need error correction at the protocol level. Also, at the protocol level I have removed the size restriction on IP packets, and therefore removed the need to keep track of fragmented data.

I only look at the basic core of the protocols, therefore I do not consider multiple sessions, multiplexing or any of the advanced features offered by FTP. Also as I am only interested in the correct transfer of data I stop as soon as the transfer has taken place or failed. So, I also ignore the ways in which connections are terminated at the protocol level and I do not examine the ways in which sessions end at the application level.

I drop all the error correction information from the IP packet, i.e. the check sum, the length of the header, time-to-live fields, etc. I do the same with the TCP packet. As a result, my data packets only contain the socket information and the data payload:

Packet $\equiv$ (Source IP, Destination IP, Source port, Destination port, Data)

As with any $la\pi$ model I must decide on the levels the model will use. The previous examples have used three levels: *app*, *host* and *net*. In these models, all the processes have been at the *app* level and the *host* and *net* levels have just been used to provide boundaries. In this model, I use more abstract levels. Processes that deal with specific jobs run at the level at which those job take place. The levels are:

$$app < transport < network < link$$

The application processes all run at the *app* level. The *transport* level handles TCP command based communication for *app* level processes. The *transport* level, in turn, uses the *network* level to handle the passing of IP packets to their destinations. Assigning the *transport* or *network* level to the channels used by these processes, ensures that all applications running inside these levels always talk to the right handler. A single router process runs at each *network* level and forwards packets to local hosts or other routers. The link level is only used by the channel that the routers use to pass packets between networks.

In work similar to this, Serjantov, Sewell and Wansbrough [SSW01] have built a more complex model of the User Data Protocol. This model is aimed at providing realistic models of real software and so addresses many of the issues I do not, such as packet loss, host failure and timeouts. These grander ambitions lead to a more complex calculus. Amadio and Prasad [AP98] have made a model of the next generation version of the IP protocol (IP version 6 [DH98]). This is a single level model, which looks at just one layer of the TCP/IP stack. Indeed, with a rearrangement into the π-calculus this model of IP version 6 could replace the model of IP version 4 that runs at the *network* level of the model presented here. Both of these models use purpose built calculi that lack the flexibility of the $la\pi$-calculus.

6.2.1 Router Model

The routers handle the lowest level of communications between networks. My routers listen for a packet being sent out on their names. Once a packet is received the router refers to a lookup table to find where that packet should be passed. Packets destined for local machines are then sent directly to those machines on the machine's name. Packets for machines on remote networks are broadcast to another router.

```
Router(name) =
        spawn {in name (dip,sip,dport,sport,data)} repeat
          {
            new replyRouter:LOOKUPREP;
            out lookup (replyRouter, dip);
            in  replyRouter (forward);
            out forward (dip,sip,dport,sport,data)
          }
```

The channels *name* and *forward* are router names. They communicate at the link level, so they have the sorting:

$$S(ROUTERNAME) \mapsto (IP, IP, PORT, PORT, DATA)@link$$

The reply channel carries a router name between the lookup table and a router and so must travel at the network level:

$$S(LOOKUPREP) \mapsto ROUTERNAME@net$$

The lookup channel's sorting is also easily derived:

$$S(LOOKUP) \mapsto (IP, LOOKUPREP)@net$$

The lookup table simply links a router or host name to each IP. Different routers work off different tables of data.

LookupTable =

    spawn { **in** lookup (replyRouter, dip) } **repeat**
      { **out** dip (replyRouter) }

In my model of FTP, I have a client machine fetch a file from a server on a different network. The client machine's IP address is 129.215.98.54 and it is addressed with the name *ftpCliHost*. The host's address is 155.246.7.5 with the name *ftpSerHost*. To make the journey between these two machines each host first talks to the router on its own network. This router passes the packet to the router on the other network. The second router then passes the packet to its destination.

So, the client router passes packets for the client directly to the client and packets for the server to the server's network router.

ClientTableData =

    spawn { **in** 155.246.7.5 (replyRouter) } **repeat**
      { **out** replyRouter (routerFtpSer) };
    spawn { **in** 129.215.98.54 (replyRouter) } **repeat**
      { **out** replyRouter (ftpCliHost) }

We can now see that IP addresses carry router reply channels. So, they will have the sorting: $S(IP) \mapsto (LOOKUPREP)@net$. In the same way, the server router passes packets for the server directly to the server and packets for the client to the client's network router.

ServerTableData =

    spawn { **in** 155.246.7.5 (replyRouter) } **repeat**
      {**out** replyRouter (ftpSerHost) };
    spawn { **in** 129.215.98.54 (replyRouter) } **repeat**
      { **out** replyRouter (routerFtpCli) }

6.2.2 Linking IP Packets to Sockets

Now that IP packets can be sent and received a wrapper is needed to turn these packets into sockets.

In the TCP/IP model, the transport level reassembles fragmented IP packets. An entry in the IP packet header then dictates which network level protocol the packet is using and hence the process the reassembled packet should be passed to. The model presented here doesn't fragment IP packets. So, the *transport* level's first task is to identify which port the incoming packet is destined for. So, the whole data packet is rebroadcast on the destination port:

IPlinkSOCKET(routerIn,routerOut) =
spawn { **in** routerIn (dip,sip,dport,sport,data) } **repeat**
{ **out** dport (dip,sip,dport,sport,data) }

TCP provides a reliable host to host protocol that will work over a possibly unreliable packet switching network, usually IP. It presents six commands to the application level with which to control these connections. The commands are: Open, Close, Status, Abort, Send and Receive.

In my examination of FTP, I am only looking for the correct transfer of a file, or rather a channel name representing that file. Therefore, I am not interested in how the protocols terminate after that file has been transferred and so I have no need to add the "close" or "abort" commands to my system, similarly the "status" command is not used and so is not needed in the model. I will now go over the implementation of the commands "open", "send" and "receive".

6.2.2.1 Open

Format:

```
OPEN (local port, foreign socket,active/passive [, timeout]
                    [,precedence] [,security/compartment] [,option])
```

Open performs the twin function of actively connecting to a remote computer or passively listening for a connection. If the foreign socket is specified for a passive call, the process will listen for a connection from that location only, ignoring all others.

In my model, I remove all the optional features and split the format into two commands: $listen(local\ port, reply\ channel)$ listens for a connection on the given local port and then returns the socket name on the reply channel;

$$connect(destination\ IP,\ destination\ port,\ source\ port,\ reply\ channel)$$

connects to a remote machine. So these names have the sorts:

$$S(LISTEN) \mapsto (PORT, SOCKETREP)@transport$$
$$S(CONNECT) \mapsto (IP, PORT, PORT, SOCKETREP)@transport$$

Once a connection is opened, the full socket information is made available on the socket's name. Sockets are defined by the destination's IP number and port and the source's IP number and port. This is exactly the information needed by the "send" command to make up the IP packets to send to the router.

```
Open(sip) = spawn { in listen (portNo, reply) } repeat
              {
                new socket : SOCKET;
                in   portNo (dip, sip, dport, sport, data);
                out  reply   (socket);
                out  portNo (dip, sip, dport, sport, data);
                spawn { in socket (n) } repeat
                  {out n (sip, dip, sport, dport) }
              };
            spawn { in connect (dip, dport, sport, reply) } repeat
              {
                 new socket : SOCKET;
                 out  reply (socket);
                 spawn { in socket (n) } repeat
                   { out n (dip, sip, dport, sport) }
              }
```

This process shows how the port name is used to identify the packet and how the socket name is used to show the information needed to make IP packets for communication over that socket. These abilities are characterised by the following sorts:

$$\begin{aligned}
\mathrm{S(PORT)} &\mapsto \mathrm{(IP,IP,PORT,PORT,DATA)@transport}\\
\mathrm{S(SOCKET)} &\mapsto \mathrm{SOCKETREP@transport}\\
\mathrm{S(SOCKETREP)} &\mapsto \mathrm{(IP,IP,PORT,PORT)@transport}
\end{aligned}$$

In this simplified setting the *connect* command does not initiate the connection to the remote host. The connection is initiated when the first piece of data is sent across the connection. This is not strictly in keeping with the specification of TCP, as it means that the connection is not properly rejected if the destination is not listening at the time the connection is made. It is also impossible for a connection to be made and then for the host that is listening for the connection to send the first piece of data. However, as I am assuming that the protocols are always used correctly, these kinds of error will not arise. The *connect* command can be extended to initiate the connection at the cost of a slightly more involved model. I spell out the details in Section 6.6.

Sockets are identified by just the destination port, rather than the full destination port, IP address, source port and IP address as is specified by the TCP specification. This means that no host can have more than one connection per port. This is not a problem for the limited runs I am looking at here. The full ability to multiplex connections over ports, as long as some part of the socket is unique, could be added by an additional three checks on each incoming packet to match up all the port and IP numbers.

6.2.2.2 Send

Format:

```
SEND (local socket name, buffer address, byte count, PUSH flag,
                                           URGENT flag [,timeout])
```

This model uses the simplified form: $S(SEND) \mapsto (SOCKET, DATA)@transport$ to send the data tuple, over the named socket. The send process gets the socket information from the socket name. It then uses this information to build the data packet and pass the fully formed packet to the router:

```
Send(routerOut) =
    spawn { in send (socket, data) } repeat
      {
        new n : SOCKETREP;
        out socket     (n);
        in  n          (dip,sip,dport,sport);
        out routerOut  (dip,sip,dport,sport,data)
      }
```

6.2.2.3 Receive

Format:

`RECEIVE (local socket name, buffer address, byte count)`

The *receive* command finds the socket information from the socket name, just as the send command did. It then listens on the local port for a data packet. Once it receives a packet, it strips off the header and sends the data to the user, over the reply channel:

```
Receive =
    spawn { in receive (socket, reply)}  repeat
      {
        new n : SOCKETREP
        out socket  (n);
        in  n       ( _,_,_ ,sport);
        in  sport   ( _,_,_ ,_ , data);
        out reply   (data)
      }
```

Model call	Equivalent TCP call	Function
connect	OPEN(active)	connect to a remote machine
listen	OPEN(passive)	listen for a connection
send	SEND	send data over an already open socket
receive	RECEIVE	receive data from a remote host over an already open socket

Figure 6.2: TCP calls and their model counterparts

The receive channel has the sorting:

$$S(RECEIVE) \mapsto (SOCKET, DATAREP)@transport$$

6.2.2.4 The Transport Level

The TCP calls and their model counterparts are summarised in Figure 6.2. Each piece of the transport level runs in parallel. A host's entire transport level is parameterised by its own IP address, the name on which packets will be sent to it, and the name of a router to which it can send packets.

TCP(sip,routerIn,routerOut) =
    IPlinkSOCKET(routerIn);
    Open(sip);
    Send(routerOut);
    Receive

6.2.3 Network Address Translation

The 32 bit IP address length allows for over four billion possible addresses. At the time the Internet was conceived this was considered more than enough. Even with rapid growth of the Internet the addition of the 4 billionth host is a long way off. However, IP addresses have been handed out in a manner designed for efficient routing, rather than efficient usage. The system of handing out addresses, used until recently was known as the *classfull* addressing system. This used the first few bits of an address as a label for the network and the remaining bits as a number for a host on that network.

There were three different sizes of “network” The biggest were the Class A addresses, which are marked with a 0 as the first bit followed by a 7 bit network address. This allowed for 127 possible networks, with slightly over 16 million hosts. Class A addresses have been assigned to major organisations, such as the US navy that has the addresses 26.0.0.0 to 26.255.255.255. Also using Class A addresses are the original founding sights of the Internet, such as M.I.T., 18.0.0.0 to 18.255.255.255, and any company lucky enough to get in on the act early, such as IBM 9.0.0.0 to 9.255.255.255 and quite bizarrely the British Department of Social Security, which has the addresses 51.0.0.0 to 51.255.255.255 to share between its 93,000 staff.

The second largest address group is the Class B network addresses. These start with the bits 10, hence the addresses are 128.0.0.0 to 191.255.255.255. These use 14 bits for the network number and so allow for roughly sixteen thousand networks each with sixty five thousand hosts. The smallest network group, Class C, allows for roughly 2 million networks with up to 256 hosts each.

Addresses starting with the bits 1110 were used for multicasting and those starting 1111 were reserved for future uses, hence stopping five hundred million addresses being used as host names.

It is easy to see how this system leads to inefficiencies, M.I.T. has nowhere near 16 million computers, but still has 16 million addresses reserved for its personal use. The real problem however, is the Class B addresses. Almost no small company or organisation starts life planning to remain small for long, so very few ever opted for a Class C network, instead going for the much larger Class B network. This has led to many Class B networks using less than 100 of their 64,000 address space.

This system has recently been replaced by a *classless* addressing system known as Classless Inter-Domain Routing (CIDR), [FLYK93]. This system hands out addresses more efficiently, at the cost of a very slight increase in routing complexity. But much of the damage is already done; we are running short of addresses. A permanent fix is planned by way of a complete new Internet Protocol: IP version 6. But as it is incompatible with the current IP version 4 many authors [Wei01, Duc02] suggest it may never be rolled out.

At the same time, it is common for machines on some networks to have no direct contact with the outside world, especially when webpages are delivered through a proxy and email through a local mail server. Most of the time these machines are just using up an IP address that is not needed.

Network Address Translation, as defined in [EF94, SE01], increases the number of hosts that can have an IP address by allowing a restricted set of IP addresses to be reused in mutually exclusive networks. Three sets of IP addresses have been put aside for local use:

10.0.0.0	-	10.255.255.255
172.16.0.0	-	172.31.255.255
192.168.0.0	-	192.168.255.255

The local routers are the only ones to have the local routing information for these addresses. Although it is possible for these machines to transmit IP packets it will be impossible for any packet to find its way back.

Generally, a network that is running most of its machines with internal addresses would have a small number of normal IP addresses that it had not assigned to any given host. When one of the hosts that had been configured to use a local address wished to access the outside world, it would be assigned one of the external addresses. It could then use this address for as long as the connection was needed. Once the process was finished it would return the external IP address for another host to use and go back exclusively to using its internal address.

A more advanced version of NAT allows a number of different hosts to communicate over the same IP address and use their source port as identification. Known as Network Address Port Translation or NAPT [SE01], this has the advantage of allowing a large number of hosts to connect through the same address at the same time. However, it has several disadvantages such as not working with transport protocols that do not use port numbers. So breaking the "Internet model" that the workings of one level should not effect another.

The following model examines the basic version of NAT, I also assume that all necessary address translation has been set up. When a packet arrives at a router on a network that is running NAT the router gives the lookup table the option of substituting the IP addresses in the packet's header.

```
NATLookupTable =
  spawn { in lookup (replyRouter, dip, sip) } repeat
    {
      new same : LOOKUPREP_NAT;
      new swap : LOOKUPREP_NAT;
      out  dip (same, swap );
      spawn { in same (forward, newdip) }
        { out replyRouter (forward, newdip, sip) };
      spawn { in swap (forward, newdip) }
        {
          out  sip    (same);
          in   same  ( _, newsip);
          out  replyRouter (forward, newdip, newsip)
        }
    }
```

ClientNATtableData =
spawn { **in** 155.246.7.5 (same, swap) } **repeat**
{ **out** swap (routerFtpSer, 155.246.7.5) };
spawn { **in** 129.215.98.54 (same, swap) } **repeat**
{ **out** same (ftpCliHost, 192.168.32.26) };
spawn { **in** 192.168.32.26 (same, swap) } **repeat**
{ **out** same (ftpCliHost, 129.215.98.54) }

ServerNATtableData = ServerTableData

The added functionality requires extended sorts for IP addresses and look up channels:

$$\begin{array}{rcl} \text{S(LOOKUP\_NAT)} & \mapsto & \text{(LOOKUPREP,IP,IP)@net} \\ \text{S(IP\_NAT)} & \mapsto & \text{(LOOKUPFLAG,LOOKUPFLAG)@net} \\ \text{S(LOOKUPREP\_NAT)} & \mapsto & \text{(ROUTERNAME,IP)@net} \end{array}$$

These new sorts can be coerced into the old sorts. What is more, the behaviour of the channels with or without NAT is exactly the same when no actual address translation is taking place. Therefore, NAT routers and normal routers can, quite happily, coexist in the same system.

6.3 File Transfer Protocol

6.3.1 Introduction to FTP

File Transfer Protocol is one of the original Internet protocols that dates back to the days of the APRAnet. As such, it has been revised and updated many times leading to a system of downloading that isn't exactly instinctive. From a design point of view the protocol's most interesting feature is that it uses separate socket connections to send the protocol commands and to transfer the file. These connections can even be between different hosts making it possible to use FTP to transfer files between two remote sites. This little-used feature dates from a time when remotely logging into sites was complicated and when it was also considered useful to have the ability to send files directly to "dumb" remote devices such as printers.

FTP provides many commands to identify and authenticate users, navigate remote file structures, transfer files and even provide help. The three key commands involved in file transfer are data port (PORT), passive (PASV) and retrieve (RETR).

The data port command is used to specify the IP address and port number that the data connection is to use. The command also indicates that the server should be running in active mode. It is common practice for FTP clients to dispense with this command and reply on a series of defaults but in this simplified system, I have chosen to make it compulsory.

The passive command requests that the server listens for a data connection. The client can then initiate the connection to the server. The port the server will listen on is returned in reply to this command.

Once the information necessary to make a data connection has been provided, the client can then ask for a given file to be sent by using the retrieve command.

FTP requires that an explicit acknowledgment is given to each command. This dramatically reduces the number of states any given run of the protocol may reach. It also allows both sides to keep a fairly good idea of the exact state of the other side. These acknowledgments take the form of a three digit number followed by a string. The number defines the type of response whereas the string is intended to be a friendly message to the user, or may carry additional information, if required.

There are numerous commonly used acknowledgements including many to help recover from erroneous, out of order or incomplete commands. Given that I am assuming that the protocol is always used correctly only three are useful. They are shown in the following table.

Number	String	Meaning
200	Command ok	used in reply to the port command
227	Switch to passive mode using IP:PORT	used in reply to the pasv command
150	File status ok, about to open data socket	used in reply to the retr command

6.3.1.1 FTP Client Model

The FTP client is started by a call on the channel name *ftp*. This call specifies the IP address of the FTP server, the name of the file that is to be downloaded, a flag to indicate if the protocol is to be run in passive or active mode and a return channel for the downloaded file.

The client's first job is to connect to the FTP server. It does this by requesting a connection from its transport level on the channel *connect*. It then selects either active or passive mode.

```
FTPclient = area app { in  ftp  (ip, filename, mode, filerep);
                       new socketReply at SOCKETREP;
                       out connect (ip , 21, 3022, socketReply);
                       in  socketReply (socket); new replyCli at DATAREP;
                       out mode ();
                       spawn { in active () }
                         Active Mode;
                       spawn { in passive () }
                         Passive Mode
                     }
```

The sort of the ftp channel provides all the information needed to find and return the file, $S(FTP) \mapsto (IP, FILENAME, MODE, DATAREP)@transport$. The mode indicators *passive* and *active* are used for synchronisation at the application level: $S(MODE) \mapsto ()@app$.

In active mode the client issues the port command and instructs the transport level to listen on a port. Finally, both modes then send the server the *retr* command and listen on this data connection for the file.

$$
\begin{array}{llll}
\text{Active Mode} = & \underline{\textbf{out}} & \text{send} & (\text{socket}, (\text{port}, \text{ip}, 3021)); \\
 & \underline{\textbf{out}} & \text{receive} & (\text{socket}, \text{replyCli}); \\
 & \underline{\textbf{in}} & \text{replyCli} & (200); \\
 & \underline{\textbf{out}} & \text{send} & (\text{socket}, (\text{retr}, \text{filename})); \\
 & \underline{\textbf{out}} & \text{receive} & (\text{socket}, \text{replyCli}); \\
 & \underline{\textbf{in}} & \text{replyCli} & (150); \\
 & \underline{\textbf{out}} & \text{listen} & (\text{socketReply}, 3021); \\
 & \underline{\textbf{in}} & \text{socketReply} & (\text{dataSocket}); \\
 & \underline{\textbf{out}} & \text{receive} & (\text{dataSocket}, \text{filerep})
\end{array}
$$

This is the first time I have specified the data that is sent across the network. It consists of an FTP command, *port*, *pasv* or *retr*, and arguments that might be needed by these commands, i.e. an IP address and a port number, more formally: $DATA = (COMMAND, IP, PORT)$. The reply channels are therefore sorted as carrying DATA at the transport level. To avoid further coercion I give the filename the same sort as IP addresses, so it can be carried as the second part of the data.

If the client is in passive mode it sends the *passive* command and then opens the data connection to the server.

```
Passive mode = out send          (socket , ( passive ));
               out receive       (socket , replyCli );
               in  replyCli      (227, serverIP , serverPort );
               out send          (socket , ( retr , filename ));
               out receive       (socket , replyCli );
               in  replyCli      (150);
               out connect       (serverIP , serverPort , 3020, socketReply);
               in  socketReply   (dataSocket);
               out send          (dataSocket);
               out receive       (dataSocket, filerep )
```

With a little more coercion I can match up the sorts of the file reply and data reply channels by giving the file the sort COMMAND.

6.3.1.2 FTP Server Model

The server's job is more complicated than the client's. It must be able to handle each of the three commands in the reduced File Transfer Protocol. To simplify things, this server only handles one connection at a time.

The IP address and the port number for the data connection are stored on the names *storeIP* and *storePort* and the mode the server is running in is represented by the flag *passive* or *active*.

When a message is received by the server, it is broadcast on the command's name. It will then be picked up by whichever process deals with that command.

```
Command Link = spawn { in replySer (command, arg1, arg2) } repeat
               {
                 out command (arg1, arg2);
                 out receive  (serverSocket , replySer)
               }
```

The Data Port Command When the PORT command is received the IP address and port number for the data connection are set, as is the flag *active*.

Port Command = **spawn** { **in** port (newIP, newPort) } **repeat**
{
out storeIP (newIP);
out storePort (newPort);
out active ();
out send (serverSocket, 200)
}

The Passive Command When the PASV command is received the new port number is chosen and set as the stored value; the passive flag is set, and finally the data port number and IP address are returned to the client.

Passive Command = **spawn** { **in** pasv () } **repeat**
{
out storeIP (3021);
out passive ();
out send (serverSocket , (227, ftpip, 3021))
}

The Retrieve Command The retrieval process can run in two possible ways depending on whether the passive or active flag is set. Either way the process first finds the stored port number and then checks the mode flag to see which mode to use. Running in active mode, the process finds the stored IP address and requests the transport level for a connection to the client and then sends the file across that connection. In passive mode it listens on the stored port and then sends the file once the client has connected.

Retrieve Command = **spawn** { **in** retr (filename) } **repeat**
{
in storePort (port);
new rep at SOCKETREP ;

```
out send (serverSocket , 150);
spawn { in active () }
  {
    in   storeIP      (ip);
    out  connect      (ip, port, socketReply);
    in   socketReply  (dataSocket);
    out  send         (dataSocket, datafile)
  }
spawn { in passive () }
  {
    out  listen       (port, socketReply);
    in   socketReply  (dataSocket);
    out  receive      (dataSocket, replySer);
    out  send         (dataSocket, datafile)
  }
}
```

Combining the three command-handling processes above, the command forwarding process and the commands to listen on the assigned FTP server port, finishes the server:

```
FTPserver = area app { new socketReply at SOCKETREP;
                       new replySer at DATAREP;
                       out  listen       (21, socketReply);
                       in   socketReply  (serverSocket);
                       out  receive      (serverSocket, replySer);
                       Command Link;
                       Port Command;
                       Passive Command;
                       Retrieve Command
```

6.4 Fitting it all Together

The components presented in the last few sections comprise a building set from which I can construct many different systems. Having one router per network and a properly configured transport level is enough to give me TCP style communication between hosts on any network topology. Clients and servers can then be dropped into any host and run, seamlessly.

6.4.1 A Basic FTP Network

FTP is most commonly used to transfer a file from a server on one network to a client on another.

$$\begin{aligned} \text{System} \equiv\ & \text{FTPServerNetwork;} \\ & \text{FTPClientNetwork} \end{aligned}$$

Each network must contain a router with a unique router name: *routerFtpCli* is used for the FTP client's network router. To support the router each network must also have a lookup table with properly configured data.

$$\begin{aligned} &\text{FTPClientNetwork} \equiv \\ &\underline{\textbf{area}}\ \text{net}\ \{\quad \text{Router(routerFtpCli);} \\ &\qquad\qquad \text{LookupTable;} \\ &\qquad\qquad \text{ClientTableData;} \\ &\qquad\qquad \underline{\textbf{area}}\ \text{transport}\ \{\ \text{TCP(129.215.98.54,ftpCliHost,routerFtpCli);} \\ &\qquad\qquad\qquad\qquad \text{FTPclient}\ \}\ \} \end{aligned}$$

There can be any number of hosts on a network, but in this example, I only have one on each network. The TCP process needs to know its own IP address and the name it should respond to as well as the name of a router that will deliver packets for it. The server network differs only by a slight change in the configuration data and the replacement of the FTPclient application with the FTPserver.

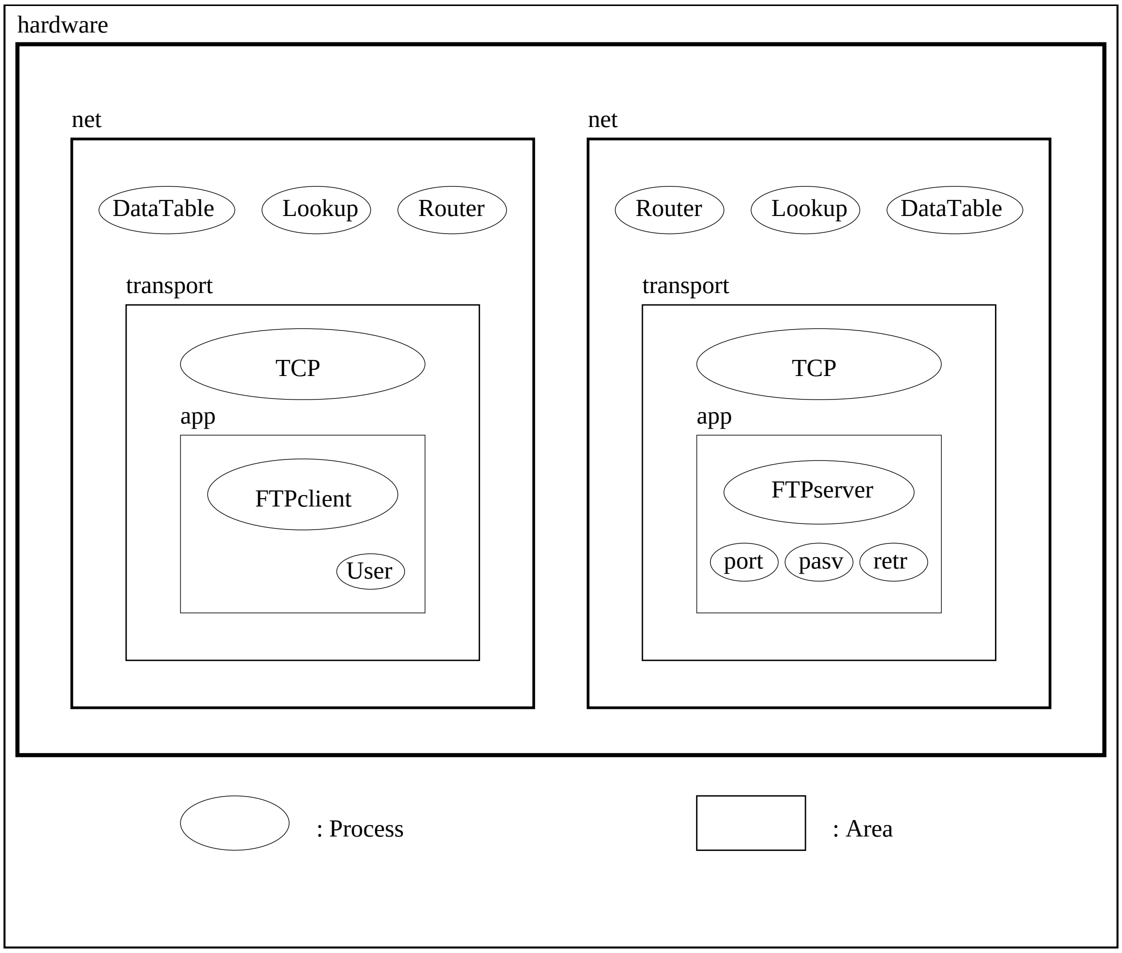


Figure 6.3: FTP on TCP/IP

FTPServerNetwork $\equiv$

area net { Router(routerFtpSer);
LookupTable;
ServerTableData;
area transport { TCP(155.246.7.5),ftpSerHost,routerFtpSer);
FTPserver } }

The overall set up is displayed graphically in Figure 6.3. All that remains to make a running process is to define the sorts of the free names.

$$\Gamma \equiv \{ \quad \text{Network level names,} \\ \text{Transport level names,} \\ \text{Application level names } \}$$

Network level names

$$\begin{array}{rlrl}
\text{lookup} & : \text{LOOKUP} & 155.246.7.5: & \text{IP} \\
129.215.98.54: & \text{IP} & 192.168.32.26: & \text{IP}
\end{array}$$

$$\begin{array}{rlrl}
\text{routerFtpSer} & :\text{ROUTERNAME} & \text{ftpCliHost} & :\text{ROUTERNAME} \\
\text{ftpSerHost} & :\text{ROUTERNAME} & \text{routerFtpCli} & :\text{ROUTERNAME}
\end{array}$$

Transport level names

$$\begin{array}{rlrl}
21 & : \text{PORT} & 3020: & \text{PORT} \\
3021 & : \text{PORT} & 3022: & \text{PORT}
\end{array}$$

$$\begin{array}{rlrl}
\text{listen} & :\text{LISTEN} & \text{connect} & :\text{CONNECT} \\
\text{send} & : \text{SEND} & \text{receive} & : \text{RECEIVE}
\end{array}$$

Application level names

$$\begin{array}{rlrl}
\text{ftp} & : \text{FTP} & \text{pasv} & :\text{COMMAND} \\
\text{port} & :\text{COMMAND} & \text{retr} & :\text{COMMAND}
\end{array}$$

$$\begin{array}{rlrl}
\text{data.tex}: & \text{IP} & \text{datafile}: & \text{COMMAND} \\
\text{active}: & \text{MODE} & \text{passive}: & \text{MODE}
\end{array}$$

$$\begin{array}{rlrl}
200 & :\text{COMMAND} & 227 & :\text{COMMAND} \\
150 & :\text{COMMAND} & &
\end{array}$$

6.4.2 An FTP Network Client Side NAT

The simplest possible instance of Network Address Translation in this network would be for the client to use NAT while the server sticks to the normal router and IP address.

$$\begin{aligned}
\text{NatSystem} \equiv\ & \text{FTPServerNetwork}; \\
& \text{NatFTPClientNetwork}
\end{aligned}$$

A NAT router and lookup table replace their counterparts in the client network and the transport level is run off the internal IP address: 192.168.32.26. What really makes this address internal to the client network is not the level binding of the name but, rather, that only the client's lookup table has an entry that passes packets for this IP address towards the client's host. So, packets sent to 192.168.32.26 from external hosts will not make it to the client.

NatFTPClientNetwork ≡

area net { NatRouter(routerFtpCli);
NatLookupTable;
NatClientTableData;
area transport { TCP(192.168.32.26,ftpCliHost,routerFtpCli);
FTPclient } }

Although the new server network is syntactically the same as the old one, there is in fact a subtle change. The NAT router extends the lookup channel to carry the source IP address as well as the destination. The IP address carries two reply channels and a lookup reply channel now also carries an IP address. However, all these changes can be coerced out of the process so the basic non-NAT router works quite happily with the extended channels. The free names for the NAT process are the same as those of the basic process except with IP, LOOKUP and LOOKUPREP updated to their extended NAT versions.

6.5 The Running System

In this section, I present three traces of the FTP model that show how the protocol works with and without Network Address Translation. The first trace shows an active request for a file running over a network without address translation. Address translation is added in the second trace, and it is seen that the attempt to open the data connection fails. In the final trace, the FTP client switches to passive mode and the file transfer succeeds.

In this simplified model, only one outcome is possible for each run. However, the order in which certain acts take place may vary. For instance, when a connection is being opened the client and the server will both be taking their own steps to establish the connection at the same time. As these steps take place in different areas and as they do not interfere with each other, they may interleave until the connection is made and the traces synchronise. For this reason, and for reasons of clarity, I present the traces as a number of small sections, some of which may overlap.

If these processes were encoded into the π-calculus with the encoding of Chapter 5 there would no longer be a single outcome for each trace, since packets could be endlessly picked up and rejected. As pointed out in Section 5.1, this would mean that these systems could no longer be analysis by examining all possible traces and so some other way would have to be found to explore these systems.

Much of the trace of each system is repetitive, especially the passing of a packet from one host to another which only differs by the packet's payload. So, I just present the highlights of each trace. The full version of each trace can be found in Appendix B.

6.5.1 Active FTP without NAT

This is the classic case: the network is as shown in Figure 6.3. A normal router is being used and the FTP user issues an active request for a file. Figure 6.4 shows a sketch of the connections made during this run.

The first step is for the FTP user to issue the command to get the file and for the FTP client to make its connection to the server.

1	FTPuser	$\rightarrow$	client App
			: ftp(155.246.7.5,data.tex,active,filerep)
2	client App	$\rightarrow$	client Transport
			: connect(155.246.7.5,21,3022,socketReply)
3	client Transport	$\rightarrow$	client App
			: socketReply(ftpCliSocket)

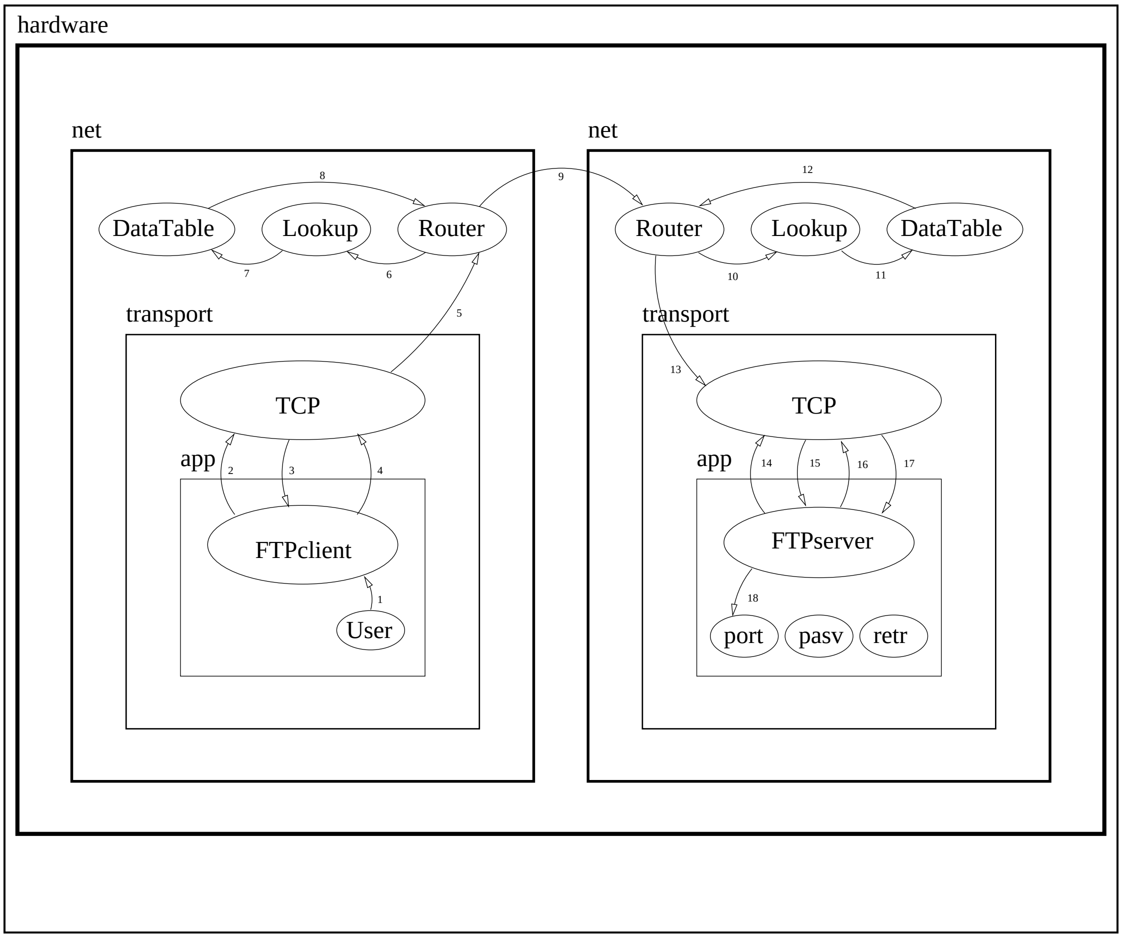


Figure 6.4: FTP running on TCP/IP

The line marked 2 spins off a process to link the name *ftpCliSocket* to the socket information *(155.246.7.5 , 129.215.98.54 , 21 , 3022)*. The client then selects its mode and issues the first command.

	client App	$\rightarrow$	client App	: active
♣♣ 4	client App	$\rightarrow$	client Transport	
			: send(ftpCliSocket,(port , 129.215.98.54 , 3021))	

We now come to the first possible branch in the trace. After the port command is passed to the transport level, by the line marked ♣♣, the transport process will carry this message to the server. At the same time, the client will request that the transport level listens for a reply. The request to listen is made up of three actions.

client App	$\rightarrow$	client Transport	: receive(ftpCliSocket,replyCli)
client Transport	$\rightarrow$	client Transport	: ftpCliSocket(n)
client Transport	$\rightarrow$	client Transport	
		: n(155.246.7.5 , 129.215.98.54 , 21 , 3022)	

This part of the trace will then block while waiting for the reply to the port command, when the traces will synchronise. While this is happening, the packet containing the port command is passed to the server. The packet's first stop is the FTP client's network router. After consulting its lookup table, the client's router passes the packet across to the server's router. This router then hands off the packet to the FTP server's host.

client Transport → client Transport : ftpCliSocket(n)
client Transport → client Transport
: n(155.246.7.5 , 129.215.98.54 , 21 , 3022)
5 client Transport → client Router
: routerFtpCli(155.246.7.5 , 129.215.98.54 , 21 , 3022 ,
(port , 129.215.98.54 , 3021))

6 client Router → client Lookup
: lookup(replyRouter , 155.246.7.5)
7 client Lookup → client Lookup
: 155.246.7.5(replyRouter)
8 client Lookup → client Router
: replyRouter(routerFtpSer)
9 client Router → server Router
: routerFtpSer(155.246.7.5 , 129.215.98.54 , 21 , 3022 ,
(port , 129.215.98.54 , 3021))
10 server Router → server Lookup
: Lookup(replyRouter , 155.246.7.5)
11 server Lookup → server Lookup
: 155.246.7.5(replyRouter)
12 server Lookup → server Router
: replyRouter(ftpSerHost)
13 server Router → server Transport
: ftpSerHost(155.246.7.5 , 129.215.98.54 , 21 , 3022 ,
(port , 129.215.98.54 , 3021))
A server Transport → server Transport
: 21(155.246.7.5 , 129.215.98.54 , 21 , 3022 ,
(port , 129.215.98.54 , 3021))

Before this packet can be received, the server must listen on port 21. Indeed, the line marked $\diamondsuit$, where this happens, can occur at anytime from the start of the trace. The server then picks up the packet and spins off a thread to execute the port command.

♢ 14	server App	→	server Transport	: listen(21 , replySocket)
†	server Transport	→	server Transport	
			: 21(155.246.7.5 , 129.215.98.54 , 21 , 3022 , (port , 129.215.98.54 , 3021))	
15	server Transport	→	server App	: replySocket(ftpSerSocket)
16	server App	→	server Transport	
			: receive(ftpSerSocket , replySer)	
	server Transport	→	server Transport : ftpSerSocket(n)	
	server Transport	→	server Transport	
			: n(129.215.98.54 , 155.246.7.5 , 3022 , 21)	
	server Transport	→	server Transport	
			: 21(155.246.7.5 , 129.215.98.54 , 21 , 3022 , (port , 129.215.98.54 , 3021))	
17	server Transport	→	server App	
			: replySer(port , 129.215.98.54 , 3021)	
18	server App	→	server App : port(129.215.98.54 , 3021)	

The line marked † spins off a process to link the name *ftpSerSocket* to the socket information *(129.215.98.54 , 155.246.7.5 , 3022 , 21)*. At this point, the command connection has been established between the client and the server. The client can send and receive data on the socket name *ftpCliSocket* and the server can do the same on the socket name *ftpSerSocket*.

The port command does not produce any visible signs of execution but it does output the IP address and port number of the client and a flag to indicate that the server should run in active mode. These three outputs are picked up by inputs much later in the trace, after the *retr* command has been issued. These outputs are being used as a memory with which to store the values needed later by another process. The server passes back the acknowledgement 200, “command ok”.

server App	$\rightarrow$	server Transport : send(ftpSerSocket , 200)
server Transport	$\rightarrow$	server Transport : ftpSerSocket(n)
server Transport	$\rightarrow$	server Transport : n(129.215.98.54 , 155.246.7.5 , 3022 , 21)

server Transport	$\rightarrow$	server Router : routerFtpSer(129.215.98.54 , 155.246.7.5 , 3022 , 21 , 200)
server Router	$\rightarrow$	server Lookup : Lookup(replyRouter , 129.215.98.54)
server Lookup	$\rightarrow$	server Lookup : 129.215.98.54(21 , replyRouter)
server Lookup	$\rightarrow$	server Router : replyRouter(routerFtpCli)
server Router	$\rightarrow$	client Router : routerFtpCli(129.215.98.54 , 155.246.7.5 , 3022 , 21 , 200)
client Router	$\rightarrow$	client Lookup : lookup(replyRouter , 129.215.98.54)
client Lookup	$\rightarrow$	client Lookup : 129.215.98.54(replyRouter)
client Lookup	$\rightarrow$	client Router : replyRouter(ftpCliHost)
client Router	$\rightarrow$	client Transport : ftpCliHost(129.215.98.54 , 155.246.7.5 , 3022 , 21 , 200)
client Transport	$\rightarrow$	client Transport : 3022(129.215.98.54 , 155.246.7.5 , 3022 , 21 , 200)
client Transport	$\rightarrow$	client App : replyCli(200)

After receiving this acknowledgement the FTP client sends the *retr* command. This is the actual request for the file. Again, much of this trace is taken up with the transport and host levels passing the packet between the client and the server. As I have already shown this trace for the *port* command above, I cut it out here.

♠♠ client App → client Transport
: send(ftpCliSocket , (retr , data.tex))
. . .
server Transport → server App : replySer(retr , data.tex)
server App → server App : retr(data.tex)

From this point or at anytime later in the trace, the FTP server can ask the transport level for another FTP command, even though another command will never arrive.

server App → server Transport : receive(ftpSerSocket , replySer)
server Transport → server Transport : ftpSerSocket(n)
server Transport → server Transport
: n(129.215.98.54 , 155.246.7.5 , 3022 , 21)

The FTP server processes the *retr* command by first picking up the information on the data connection, stored for it by the *port* command.

server App → server App : storePort(3021)
QQ server App → server Transport : send(ftpSerSocket , 150)

The server may enter active mode at any time after QQ. But while it is entering active mode the acknowledgement 150, “File status ok; about to open data connection ” will be passed back to the client.

server Transport → server Transport : ftpSerSocket(n)
server Transport → server Transport
: n(129.215.98.54 , 155.246.7.5 , 3022 , 21)
server Transport → server Router
: routerFtpSer(129.215.98.54 , 155.246.7.5 ,
3022 , 21 , 150)
. . .
client Transport → client Transport
: 3022(129.215.98.54 , 155.246.7.5 , 3022 , 21 , 150)
client Transport → client App : replyCli(150)

The server now opens the data connection and sends the file while the client listens for the connection.

server App → server App : active()
server App → server App : storeIP(129.215.98.54)
server App → server Transport
: connect(129.48.32.2 , 3021 , 3020 , reply)
server Transport → server App : reply(ftpSerDataSocket)
server App → server Transport
: send(ftpSerDataSocket , dataFile)
...
client Router → client Transport
: ftpCliHost(129.215.98.54 , 155.246.7.5 ,
3021 , 3020 , dataFile)
client App → client Transport : listen(3021 , socketReply)
client Transport → client App : socketReply(ftpCliDataSocket)

client App	$\rightarrow$	client Transport
		: receive(ftpCliDataSocket , filerep)
client Transport	$\rightarrow$	client Transport : ftpCliDataSocket(n)
client Transport	$\rightarrow$	client Transport
		: n(155.246.7.5 , 129.215.98.54 , 3020 , 3021)
client Transport	$\rightarrow$	client Transport
		: 3021(129.215.98.54 , 155.246.7.5 ,
		3021 , 3020 , dataFile)
client Transport	$\rightarrow$	outside processes : fileReply(dataFile)

Of course, the listen command may be issued as soon as the client receives the 150 acknowledgement, but it cannot happen later than at the point indicated in the trace above.

The final action successfully passes the file to the user as requested.

6.5.2 Active FTP with NAT

Section 6.5.1 presented a successful, normal run of FTP. This section shows what happens when an active FTP request is made on a network that is using Network Address Translation. In this model the client's router has been replaced by the NAT router of Section 6.2.3 and the lookup table has been updated with the new internal and external addresses. The client now uses the internal IP address 192.168.32.26 which is mapped to the external address 129.215.98.54. The server remains on 155.246.7.5.

The trace starts in the same way as the last one, with the user issuing the *ftp* command to the FTP client and the client opening a connection to the server. But this time the name *ftpCliSocket* is linked to the socket information *(155.246.7.5 , 192.168.32.26 , 21 , 3022)*.

$$\begin{array}{lll}
\text{client App} & \to & \text{client Transport} \\
 & & \quad : \ \text{send(ftpCliSocket , (port , 192.168.32.26 , 3021))} \\
\text{client Transport} & \to & \text{client Transport} \ : \text{ftpCliSocket(n)} \\
\text{client Transport} & \to & \text{client Transport} \\
 & & \quad : \ \text{n(155.246.7.5 , 192.168.32.26 , 21 , 3022)}
\end{array}$$

When the packet containing the *port* command is passed to the client's router the router replaces the internal IP with 129.48.32.2:

$$\begin{array}{lll}
\text{client Transport} & \to & \text{client Router} \\
 & & \quad : \ \text{routerFtpCli(155.246.7.5 , 192.168.32.26 , 21 , 3022 ,} \\
 & & \qquad\qquad\qquad\qquad \text{(port , 192.168.32.26 , 3021))} \\
\text{client Router} & \to & \text{client Lookup} \\
 & & \quad : \ \text{lookup(replyRouter , 155.246.7.5 , 192.168.32.26)} \\
\text{client Lookup} & \to & \text{client Lookup} \ : \ \text{155.246.7.5(same , swap)} \\
\text{client Lookup} & \to & \text{client Lookup} : \ \text{swap(routerFtpSer , 155.246.7.5)} \\
\text{client Lookup} & \to & \text{client Lookup} \ : \ \text{192.168.32.26(same)} \\
\text{client Lookup} & \to & \text{client Lookup} \ : \ \text{same(_129.215.98.54)} \\
\text{client Lookup} & \to & \text{client Router} \\
 & & \quad : \ \text{replyRouter(routerFtpSer , 155.246.7.5 , 129.215.98.54)}
\end{array}$$

$$\begin{array}{lll}
\text{client Router} & \to & \text{server Router} \\
 & & \quad : \ \text{routerFtpSer(155.246.7.5 , 129.215.98.54 , 21 , 3022 ,} \\
 & & \qquad\qquad\qquad\qquad \text{(port , 192.168.32.26 , 3021))}
\end{array}$$

The server's router processes the packet as normal.

$$
\begin{array}{lcl}
\text{server Router} & \rightarrow & \text{server Lookup} \\
 & & \ : \text{lookup(replyRouter , 155.246.7.5)} \\
\text{server Lookup} & \rightarrow & \text{server Lookup} \ : \ \text{155.246.7.5(replyRouter)} \\
\text{server Lookup} & \rightarrow & \text{server Router} \ : \ \text{replyRouter(ftpSerHost)} \\
\text{server Router} & \rightarrow & \text{server Transport} \\
 & & \ : \ \text{ftpSerHost(155.246.7.5 , 129.215.98.54 , 21 , 3022 ,} \\
 & & \qquad\qquad\qquad \text{(port , 192.168.32.26 , 3021))} \\
\text{server Transport} & \rightarrow & \text{server Transport} \\
 & & \ : \text{21(155.246.7.5 , 129.215.98.54 , 21 , 3022 ,} \\
 & & \qquad\qquad\qquad \text{(port , 192.168.32.26 , 3021))}
\end{array}
$$

Although the packet has been successfully delivered to the server, we can now catch our first glimpse of the forthcoming problem. The server now believes that it should use the client's private internal address, an address unknown to the server network's router. However, the command connection is fully operational so the server can use this socket to return the acknowledgement. The client then requests the file.

I pick up the trace again as the server selects active mode and attempts to send the file.

$$
\begin{array}{lcl}
\text{server App} & \rightarrow & \text{server App : active()} \\
\text{server App} & \rightarrow & \text{server App} \ : \text{storeIP(192.168.32.26)} \\
\text{server App} & \rightarrow & \text{server Transport} \\
 & & \ : \text{connect(192.168.32.26 , 3021 , 3020 , reply)} \\
\text{server Transport} & \rightarrow & \text{server App : reply(ftpSerDataSocket)} \\
\text{server App} & \rightarrow & \text{server Transport} \\
 & & \ : \ \text{send(ftpSerDataSocket , dataFile)} \\
\text{server Transport} & \rightarrow & \text{server Transport : ftpSerDataSocket(n)} \\
\text{server Transport} & \rightarrow & \text{server Transport} \\
 & & \ : \text{n(192.168.32.26 , 155.246.7.5 , 3021 , 3020)} \\
\text{server Transport} & \rightarrow & \text{server Router} \\
 & & \ : \ \text{routerFtpSer(192.168.32.26 , 155.246.7.5 , 3021 ,} \\
 & & \qquad\qquad\qquad\qquad \text{3020 , dataFile)} \\
\text{server Router} & \rightarrow & \text{server Lookup} \ : \ \text{lookup(replyRouter , 192.168.32.26)}
\end{array}
$$

At this point, we see that the FTP server has requested a connection to the FTP client using its internal address. Once this packet reaches the router, it is found that this address is not in the lookup table and the model blocks. A real life system would issue an Internet Control Message Protocol (ICMP) [Pos81a] message stating that the network is unreachable. However, the problem would be exactly the same as the one illustrated here. The client is unaware of the IP address that should be used to talk to it from outside the network and so it has no way of giving this information to an external process. Therefore, any attempt by the server to open a data connection is doomed to failure.

6.5.3 Passive FTP with NAT

In the last section, we saw the problems caused by adding NAT to a network on which users were running FTP clients. Now we can see what is done about it. This time the user issues a *passive* command rather than an *active* one.

The set up is the same as before; the client uses the internal address 192.168.32.26 that is mapped to the external address 129.215.98.54. The server remains at 155.246.7.5. The only difference this time is that the FTP user requests that the client uses passive mode.

The FTP clients starts and connects:

	FTPuser	→	client App
			: ftp(155.246.7.5 , data.tex , passive , filerep)
1	client App	→	client Transport
			: connect(155.246.7.5 , 21 , 3022 , socketReply)
	client Transport	→	client App : socketReply(ftpCliSocket)
	client App	→	client App : passive

The selection of passive mode is the first difference between this trace and the last. As a result, the client will send the *pasv* command rather than the *port* command.

♣♣ client App → client Transport : send(ftpCliSocket,pasv)
client Transport → client Transport : ftpCliSocket(n)
client Transport → client Transport
: n(155.246.7.5 , 192.168.32.26 , 21 , 3022)
client Transport → client Router
: routerFtpCli(155.246.7.5 , 192.168.32.26 , 21 ,
3022 , pasv)

...

server Router → server Transport
: ftpSerHost(155.246.7.5 , 129.215.98.54 , 21 ,
3022 , pasv)
A server Transport → server Transport
: 21(155.246.7.5 , 129.215.98.54 , 21 ,
3022 , pasv)

At ♣♣ the client may start to listen for a reply. The server then listens for the command. After processing it the 227 acknowledgement is returned, with the string: “entering passive mode using 155.246.7.5:3021”.

$$\begin{array}{lll}
\text{server App} & \rightarrow & \text{server Transport} \\
& & \quad : \ \text{send(ftpSerSocket , (227 , 155.246.7.5 , 3021))} \\
\text{server Transport} & \rightarrow & \text{server Transport} \ : \ \text{ftpSerSocket(n)} \\
\text{server Transport} & \rightarrow & \text{server Transport} \\
& & \quad : \text{n(129.215.98.54 , 155.246.7.5 , 3022 , 21)} \\
\text{server Transport} & \rightarrow & \text{server Router} \\
& & \quad : \ \text{routerFtpSer(129.215.98.54 , 155.246.7.5 , 3022 ,} \\
& & \qquad\qquad\qquad \text{21 , (227 , 155.246.7.5 , 3021))}
\end{array}$$

...

$$\begin{array}{lll}
\text{client Router} & \rightarrow & \text{client Transport} \\
& & \quad : \ \text{ftpCliHost(192.168.32.26 , 155.246.7.5 , 3022 ,} \\
& & \qquad\qquad\qquad \text{21 , (227 , 155.246.7.5 , 3021))} \\
\text{client Transport} & \rightarrow & \text{client Transport} \\
& & \quad : \ \text{3022(192.168.32.26 , 155.246.7.5 , 3022 ,} \\
& & \qquad\qquad\qquad \text{21 , (227 , 155.246.7.5 , 3021))}
\end{array}$$

After receiving the information for the data socket the client sends the *retr* command to the server. The first thing the server does after getting this command is to read off the port information and then send the acknowledgement.

$$\begin{array}{llll}
& \text{server App} & \rightarrow & \text{server App} \quad : \text{storePort(3021)} \\
\text{QQ} & \text{server App} & \rightarrow & \text{server Transport} : \ \text{send(ftpSerSocket , 150)} \\
& \text{server Transport} & \rightarrow & \text{server Transport} : \text{ftpSerSocket(n)} \\
& \text{server Transport} & \rightarrow & \text{server Transport} \\
& & & \quad : \text{n(129.215.98.54 , 155.246.7.5 , 3022 , 21)} \\
& \text{server Transport} & \rightarrow & \text{server Router} \\
& & & \quad : \ \text{routerFtpSer(129.215.98.54 , 155.246.7.5 ,} \\
& & & \qquad\qquad\qquad \text{3022 , 21 , 150)}
\end{array}$$

...

$$\begin{array}{llll}
& \text{client Router} & \rightarrow & \text{client Transport} \\
& & & \quad : \ \text{ftpCliHost(192.168.32.26 , 155.246.7.5 , 3022 , 21 , 150)} \\
D & \text{client Transport} & \rightarrow & \text{client Transport} \\
& & & \quad : \ \text{3022(192.168.32.26 , 155.246.7.5 , 3022 , 21 , 150)}
\end{array}$$

The server may enter passive mode at any time after QQ. On receipt of this acknowledgement the FTP client opens the data socket and sends a packet to initiate the connection.

$$
\begin{array}{lllll}
 & \text{client App} & \rightarrow & \text{client Transport} \\
 & & & :\ \text{connect}(155.246.7.5\ ,\ 3021\ ,\ 3021\ ,\ \text{socketReply}) \\
 & \text{client Transport} & \rightarrow & \text{client App} \quad :\ \text{socketReply}(\text{dataSocket}) \\
 & \text{client App} & \rightarrow & \text{client Transport} : \text{send}(\text{dataSocket}) \\
\ldots & & & \\
 & \text{server Router} & \rightarrow & \text{server Transport} \\
 & & & : \text{ftpSerHost}(155.246.7.5\ ,\ 129.215.98.54\ ,\ 3021\ ,\ 3020) \\
A & \text{server Transport} & \rightarrow & \text{server Transport} \\
 & & & : 21(155.246.7.5\ ,\ 129.215.98.54\ ,\ 3021\ ,\ 3020)
\end{array}
$$

The FTP server listens for the data connection:

$$
\begin{array}{llll}
\Diamond & \text{server App} & \rightarrow & \text{server Transport} \\
 & & & :\ \text{listen}(3021\ ,\ \text{replySocket}) \\
5 & \text{server Transport} & \rightarrow & \text{server Transport} \\
 & & & : 3021(155.246.7.5\ ,\ 129.215.98.54\ ,\ 3021\ ,\ 3020) \\
 & \text{server Transport} & \rightarrow & \text{server App} \\
 & & & :\ \text{replySocket}(\text{ftpSerDataSocket})
\end{array}
$$

The server then discards the initiation packet and sends the file.

$$
\begin{array}{lll}
\text{server App} & \rightarrow & \text{server Transport} \\
 & & :\ \text{receive}(\text{ftpSerSocket}\ ,\ \text{replySer})
\end{array}
$$

G	server Transport	$\rightarrow$	server Transport
			: fpSerSocket(n)
G	server Transport	$\rightarrow$	server Transport
			: n(129.215.98.54 , 155.246.7.5 , 3022 , 21)
G	server Transport	$\rightarrow$	server Transport
			: 3021(155.246.7.5 , 129.215.98.54 , 3021 , 3020)
	server App	$\rightarrow$	server Transport
			: send(ftpSerDataSocket , dataFile)
...			
	client Router	$\rightarrow$	client Transport
			: ftpCliHost(192.168.32.26 , 155.246.7.5 , 3020 , 21 , dataFile)
E	client Transport	$\rightarrow$	client Transport
			: 3020(192.168.32.26 , 155.246.7.5 , 3020 , 3021 , dataFile)

The lines marked G discard the packet used to open the connection and so the communications may happen later than given. The client listens for the data file:

	client App	$\rightarrow$	client Transport
			: receive(ftpCliDataSocket , filerep)
	client Transport	$\rightarrow$	client Transport
			: ftpCliDataSocket(n)
	client Transport	$\rightarrow$	client Transport
			: n(155.246.7.5 , 129.215.98.54 , 3020 , 3021)
E	client Transport	$\rightarrow$	client Transport
			: 3020(129.215.98.54 , 155.246.7.5 , 3020 , 3021 , dataFile)
	client Transport	$\rightarrow$	outside processes
			: fileReply(dataFile)

With the final action above, the requested file is passed to the FTP user. This signifies that, within the bounds of this model, the theoretical problems of running FTP on a network using NAT are, indeed, overcome by using the passive mode.

6.6 Extending the Model with Initialising Connections

The main limit on the expressiveness of this model is the need to make the process terms easily presentable and to work the traces through by hand. One example of the kind of extensions that could easily be added would be to identify incoming packets on the destination and source IP addresses and ports, not just the destination port. This would automatically allow multiplexing over ports.

Perhaps a more natural extension would be to make the TCP connect command (active OPEN) initiate the connection. A result of this would be that a process using the connect command would not get a socket name back before the remote host had performed the listen command and that the listening process could send the first piece of data down the connection. This would bring the model nearer real life TCP at the cost of tens of extra actions per trace.

To see how the connections can be developed I return to the TCP specification. When establishing a connection to a remote host the connecting process will send a packet to that host to ask if an appropriate process is listening. On receiving this query, the listening process will assume the connection is ready to be used and report its success to who or what ever requested the connection. At the same time, an acknowledgement is sent back to the host that requested the connection. Only once this acknowledgement of success has arrived back will the connecting process consider the connection open.

A problem faced both by the designers of TCP and by me was that the listening process considers the connection ready to use before the acknowledgement has reached the connecting process. Therefore, it is possible that the first real data packet may overtake the acknowledgement packet. TCP overcomes this problem by placing an (almost) unique identifier in the header of each packet and by explicitly flagging the header of all acknowledgement packets. If a data packet is then received when an acknowledgement was expected the data is discarded. The packet will then be re-requested if and when the connection is properly established.

Following this same pattern, I have added a packet identifier and an acknowledgement flag to the header of my packets.

Packet ≡ (Source IP, Destination IP, Source port, Destination port, Packet identifier, Acknowledgement flag, Data)

To make processing of a packet easy in the absence of Booleans, I set the acknowledgement flag to equal the packet identifier if the packet carries data. Whereas, if the packet is an acknowledgement to a previous packet, the flag is set to the identifier of that previous packet.

The connection process now sends a packet to the remote machine to establish the connection. After sending this packet off, the process waits for a signal on the acknowledgement flag before returning the socket name to the user and making the socket information available to other processes.

```
Connect = spawn { in connect (dip, dport, sport, rep) } repeat
          {
            new socket at Socket;
            new ack at ACK;
            out router  (dip, sip, dport, sport, ack, ack, _ );
            in  ack     ();
            out rep     (socket);
            spawn { in socket (n) } repeat
              { out n (dip, sip, dport, sport) }
          }
```

where ACK has the sorting ()@*transport*. On receiving this first packet, the listening process returns the socket name to its user. It also sends its own acknowledgement to the connecting process using that process's acknowledgment flag.

```
Listen = spawn { in listen (portNo, rep) } repeat
         {
           new socket at SOCKET;
           in   portNo (dip, sip , dport, sport , ack , ackflag );
           new ack at ACT ;
           out  router  (dip , sip , dport, sport , ack , ackflag );
           out  reply   (socket );
           spawn { in socket (n) } repeat
             {out n (sip , dip , sport , dport) }
         }
```

The link process checks the acknowledgement flag of any incoming packet. If this flag is the same as the packet ID the whole packet is passed to the port in the usual way. Otherwise, the *ack* channel is used to release the blocked part of the connect process.

```
IPlinkSOCKET(router) =
  spawn { in router (dip, sip , dport, sport,ack , ackflag , data) } repeat
    {
      out  ackflag  ();
      in   ack      ();
      out  dport    (dip , sip , dport, sport , ack , ackflag , data)
    }
```

The only changes required to the send and receive commands are the addition of the packet ID and acknowledgement flag to the packet header.

Send(router) = **spawn** { **in** send (socket, data) } **repeat**
{
new n at SOCKETREP;
out socket (n);
in n (dip, sip, dport, sport);
new ack at ACK;
out router (dip, sip, dport, sport, ack, ack, data)
}

Receive = **spawn** { **in** receive (socket, rep) } **repeat**
{
new n at SOCKETREP;
out socket (n);
in n (_, _, _ , sport);
in sport (_, _ , _ , _ , _ , _ , data);
out rep (data)
}

Chapter 7

Conclusion and Further Work

7.1 Conclusion

The motivation for this work was to develop a system in which it is possible to express globally known names that are used to access local resources. It was also my aim to make this system as simple as possible without sacrificing any expressiveness. These ideas are filled out in the introduction.

In Chapter 3, I expand on the differences between *scope* and *area* and introduce the local area π-calculus. This chapter contains a simple, untyped operational semantics and I prove that it correctly captures the ideas outlined in the introduction. This operational semantics makes use of runtime checks to stop channels being used outside the area to which they are restricted. Illustrative examples include an Internet Service Protocol, a pair of distributed agents and a data cache.

The runtime checks are removed by a type system proposed in Chapter 4. This type system types a channel with the level at which it operates and the types of the channels it may transmit. I prove that this type system does indeed remove the need for runtime checks on some of the rules. Subject reduction holds for this type system, i.e. well typed processes always reduce to well typed processes. The channel types from the examples in Chapter 3 are given in full.

It is observed that the type system forbids channels that transmit their own name, even indirectly by way of another channel. This ability is useful when identifying a message by some data it carries or when two remote parties are negotiating for a connection. A second type system is then presented that uses recursive sorts, in the style of [Mil91]. This type system allows channels to pass themselves recursively. The correctness proofs for the original type system do not make use of the fact that types are finite. So, the correctness of this second type system follows easily from the proofs of the correctness of the first.

I have translated the $la\pi$-calculus back into the π-calculus. At the core of this encoding is the technique of replacing communication on a channel name with communication over an ether associated with the appropriate local area for that name. Two of the examples that have run through the previous two chapters are encoded and an illustration given of how an encoded process reduces in the π-calculus. It is shown that a single direct communication in the $la\pi$-calculus system is replaced by a packet communication on an ether, using the original channel name as a key.

An operational correspondence result shows that there is a close relation between the actions of processes and their translations.

This result is composed of six parts, three of which show how an encoded process behaves when the process it encoded performs an input, output or a communication. The other three describe how a $la\pi$ process behaves when its encoding performs one of these 3 actions.

My main motivation for investigating local communication on global names was that it provides a useful abstraction to model the kind of distributions that are used in real life networks. I offered evidence for this in Chapter 6 by presenting a model of TCP/IP network communication protocol and the File Transfer Protocol, FTP. I investigated how this protocol works in the presence of Network Address Translation.

This model makes use of several key features of the calculus. Levels are used to distinguish physical boundaries such as networks and hosts as well as more abstract boundaries, such as the levels of the TCP/IP hierarchy. The result is a plug and play arrangement where highly modular processes can be happily swapped as long as they keep the same level, i.e. an application can use the TCP/IP commands on any host and automatically get its local handler and the only change needed to switch from normal routing to Network Address Translation is to swap the routers.

I examine traces of the different modes of FTP and find that passive mode FTP works with network address translation but active mode does not. Further, I find that the active mode fails because the FTP server tries to open the data connection to an IP address that is only accessible from inside the client's own network. Although these properties of the FTP protocol are well known, the fact that they are so easily picked up in a $la\pi$-calculus model suggests that it is a useful system with which to examine and check distributed protocols.

7.2 Further Work

Well-known names that mean different things in different places are reminiscent of dynamic binding in programming languages; that slippery concept whereby the meaning of a local variable at a program point depends on how we got there. While there seems to be no direct connection, it would be interesting to know how local areas affect the classic encoding of functions as π-calculus processes [San95].

Another possible direction would be to investigate the degree to which the encoding of Chapter 5 preserves and reflects equivalences between processes. I would expect adequacy, but not full abstraction, as encoding local areas by ethers exposes them to probing by general π-calculus terms. For example, it is possible to eavesdrop on all top-level communications, even ones involving private names ("packet-snooping").

As well as extending the work presented in this thesis, many new directions suggest themselves.

7.2.1 The Extension of the $la\pi$-calculus with Mobility

One direction that may be worth pursuing is the step from static agents to properly mobile ones. The fixed arrangement of local areas in $la\pi$ does not lend itself to a dynamic runtime structure. There is however some flexibility: where areas appear under replication, they will be freshly created during execution, and empty areas are indistinguishable from the null process. I have looked at various ways to incorporate full mobility into the calculus while retaining the separate handling of scope and area. An advantage of using a system with such clear levels to examine mobile agents is that mobile agents are intrinsically level based. It makes no sense to send a process designed to run on a computer to run on a network.

Syntax, Semantics and Types

There are many important issues that I would need to consider when deciding on the exact form of mobility that I might use. Among them would be...

- ...whether processes should be passed over channels or by an explicit "*go*" command. The passing of processes over channels has been extensively used for the Higher Order π-calculus and to my knowledge has not been used in a distributed setting.
- ...whether the dynamic cloning of agents should be allowed. Static agents can be cloned using replication (!) but allowing a running agent to be cloned could add a new level of complexity to the system.
- ... how I might mark out the location I wish to move. This leads to the question of whether I should use subjective or objective forms of movement and if they are interchangeable in my system.

Introducing a "go" command would add another method of communication beyond that of passing data over channels. So to keep the model as simple as possible the best option maybe to pass areas over channels.

The mobile primitives I have considered are $send\ \ell\ a$ and $receive\ a$. The first orders the enclosing ℓ level agent to be sent over the channel a and the $receive\ a$ receives an agent that has been sent over the channel a and then runs it.

$$\text{Process } P, Q ::= \ldots \,|\, send\ \ell\ a \,|\, receive\ a$$

$$\text{AREA1\_MTS} \quad \frac{\Gamma \vdash_\ell P \xrightarrow{\alpha} P'}{\Gamma \vdash_m \ell[P] \xrightarrow{\alpha} \ell[P']} \quad \begin{array}{l}\text{if } \alpha \text{ is } \bar{a}\langle\vec{b}\rangle \text{ or } a(\vec{b})\\ \text{then } m \leq \Gamma(a)\end{array}$$

$$\text{SEND\_MTS} \quad \frac{}{\Gamma \vdash_\ell \mathit{send}\ m\ a \xrightarrow{\mathit{send}\ m\ a} 0}$$

$$\text{AREA2\_MTS} \quad \frac{\Gamma \vdash_\ell P \xrightarrow{\mathit{send}\ \ell\ a} P'}{\Gamma \vdash_m \ell[P] \xrightarrow{\bar{a}\langle \ell[P]\rangle} 0}$$

$$\text{RECE\_MTS} \quad \frac{}{\Gamma \vdash_\ell \mathit{receive}\ a \xrightarrow{a(P)} P} \quad \Gamma \vdash_\ell P$$

Figure 7.1: Mobile Typed Semantics for the local area π-calculus

I have a first draft of semantics for mobile primitives with an extended type system that preserves the correctness of the calculus with regard to local areas. The types are extended with levels. An area of level ℓ has type ℓ and a channel that can pass a ℓ level area across the host level has type $\ell @ host$. The type system presented in Chapter 4 extends effortlessly to cover mobility. The two extra rules needed are given in Figure 7.2. All the other type rules remain unchanged because the mutually exclusive types of channels that pass processes and the channels that pass names keep them apart.

The two new rules are for the *send* and *receive* actions. The send type rule checks that the channel being used is of the right type to transport the enclosing area. The receive type rule restricts the names on which the receive command can listen to names which can only receive areas of the appropriate level.

The semantics uses a two-stage process of mobility. When a send action is performed, it works its way up until it reaches the level it wishes to send. That area is then packaged up and broadcast over the named channel. The *receive* command uses an early communication to match a process being sent. The use of early communication avoids the need to substitute processes. The three extra semantic rules are given in Figure 7.1.

$$\frac{}{\Gamma \vdash_\ell \mathit{send}\ m\ a}\mathit{Send}_T \qquad \Gamma(a) = o@n \text{ and } \ell \leq n \leq n \text{ and } m <_1 o$$

$$\frac{}{\Gamma \vdash_\ell \mathit{receive}\ a}\mathit{Rec}_T \qquad \Gamma(a) = \ell@m \text{ and } \ell <_1 m$$

Figure 7.2: Types for mobile processes in the local area π-calculus

Further work along these lines would first have to establish that the *send* and *receive* commands outlined above would provide a useful form of mobility. Truly mobile processes are still at a fledgling stage so great care must be taken when modelling a technology that is still to come into its own. It would also be necessary to prove the correctness of any system with mobility.

Encoding Mobile Areas into the π-Calculus

The encoding given in Chapter 5 does not extend to handle mobility, because the encoding assumes that each process has direct access to the ethers for every containing level. A possible solution here may be to use an encoding with a network of controllers. Within an area, each process communicates only through its immediate local area controller. Each controller knows the name of the controller for its parent location and packets are routed from one controller to another until they reach their destination. Mobile areas can be represented by reprogramming the parent location of the area's controllers.

$$
\begin{aligned}
[\![\Gamma \vdash_\ell m[Q]]\!]_{\{n',nUp'\}} &\equiv \nu\ n, nUp.(Control \mid \overline{nUp}\langle n' \rangle \mid [\![\Gamma \vdash_m Q]\!]_{\{n,nUp\}}) \\
[\![\Gamma \vdash_\ell \overline{a}\langle b \rangle]\!]_{\{n,nUp\}} &\equiv \overline{n}\langle a; \Gamma(a); up; b; out; n; n \rangle \\
[\![\Gamma \vdash_\ell a(b).P]\!]_{\{n,nUp\}} &\equiv \mu\ X.n(\alpha; l_a; dir; b; data).([a = \alpha].([\![\Gamma \vdash_\ell P]\!]_{\{n,nUp\}}) \\
&\qquad \mid [a \neq \alpha].(\overline{n}\langle \alpha; l_a; dir; b; data \rangle \mid X)) \\
[\![\Gamma \vdash_\ell \nu a{:}\sigma.P]\!]_{\{n,nUp\}} &\equiv \nu a.[\![\Gamma \oplus \{a@\sigma\} \vdash_\ell P]\!]_{\{n,nUp\}} \\
[\![\Gamma \vdash_\ell P \mid Q]\!]_{\{n,nUp\}} &\equiv [\![\Gamma \vdash_\ell P]\!]_{\{n,nUp\}} \mid [\![\Gamma \vdash_\ell Q]\!]_{\{n,nUp\}} \\
[\![\Gamma \vdash_\ell 0]\!]_{\{n,nUp\}} &\equiv 0
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash_\ell send\ m\ a]\!]_n &\equiv \overline{n}(a; m; up; \Gamma(a); send; n; n) \\
[\![\Gamma \vdash_\ell receive\ a]\!]_n &\equiv \mu\ X.n(\alpha; l_\alpha; dir; nUp'; data). \\
&\qquad (\ [a = \alpha]nUp(x).(\overline{nUp}\langle x \rangle \mid \overline{nUp'}\langle x \rangle) \\
&\qquad \mid [a \neq \alpha]\overline{n}\langle \alpha; l_\alpha; dir; nUp'; data \rangle \mid X)
\end{aligned}
$$

A controller may get a packet from its parent agent. It may also send a packet up to its parent, given the correct level checks. A controller can time out any communication by sending the action back to its agent and replacing the lock. Finally, a controller can redirect to a new location.

$$
\begin{aligned}
Control = \quad & GetFromAbove \\
& \mid \mu\ X.n'(a; \ell_a; dir; data; type; tOut; n).(\quad [dir = up][l_a \neq lv]SendUp \\
&\qquad \mid [dir = up][l_a = lv]TryMove \\
&\qquad \mid [dir = down]Timeout) \mid X
\end{aligned}
$$

Send signals only move up to their level therefore it is only necessary to look for move signals and output at the level above:

$$
\begin{aligned}
GetFromAbove = \quad & \mu X.nUp'(above).(\overline{nUp'}\langle above\rangle \\
& \quad |\, above(a;\ell_a;dir;data).nUp'(check).(\overline{nUp'}\langle check\rangle \\
& \qquad |[check = above]\overline{n}\langle a;\ell_a;down;data\rangle \\
& \qquad |[check \neq above]\overline{above}\langle a;\ell_a;down;data\rangle) \,|\, X)
\end{aligned}
$$

$$
\begin{aligned}
SendUp = \quad & nUP'(above).(\overline{nUP'}(above) \\
& \qquad\qquad\qquad |\, \overline{above}\langle a;\ell_a;dir;data;type;tOut;n\rangle)
\end{aligned}
$$

$$
\begin{aligned}
TimeOut = \quad & [tOut = n]\overline{n}(a;\ell_a;up;data;type;tOut;n) \\
& [tOut \neq n]\overline{n}(a;tOut;up;\ell_a;send;n;n) \,|\, \overline{data}\langle tOut\rangle
\end{aligned}
$$

$$
\begin{aligned}
Move = \quad & [type = send]nUp(above).\overline{n'}(a;data;up;nUp;above;lv;n) \\
& [type \neq send]\overline{n'}(a;\ell_a;dir;data;type;tOut;n)
\end{aligned}
$$

The encoding uses a *controller process* for each location. All communications in a given agent in the local area π-calculus are replicated by communications to and from the controller for this agent in the π-calculus. The controller may pass a communication up to the controller for the agent above it or it may match two communications having first made the appropriate checks. Packets travel up the routers and then switch and travel down the location tree. This ensures that a packet cannot traverse a network forever without timing out and returning to its original area. The packets include a channel they can use to look up the location of the area it started out from, rather than the actual name of the area. This means that the packet can look up the current location of its area, even if it has moved, and so does not get lost when their agents move. Encoding the typed calculus would allow me to dispense with some of the runtime checks I would have had to perform for the untyped calculus.

In this encoding an output packet may be dropped into the ether, the area may then move before the output packet finishes its communication. It is even possible that the output packet may synchronise with an input that is only available because the area moved. This is different to the purposed behaviour for the mobile $la\pi$-calculus. Where, although communication is asynchronous, an output signal must remain in its area until the communication has happened. Which of these is a better model of asynchronous communication in the presence of mobility would have to be investigated and the mobile calculus and the encoding made to match.

Even with a completed mobile calculus and encoding, a proof of correctness will be no easy feat. A large bi-simulation relation would seem to be the best possibility, with a mobile $la\pi$-calculus process related to its encoding and all the possible *house-keeping* stages that the encoded process might reach.

An encoding of a system of cloneable mobile agents would be not as easy but may be more interesting to look into.

A Mobile Case Study

One of the reasons for making these mobility extensions of the calculus would be to produce a convenient and powerful system that could be used to examine real life systems. This would call for a case study, like the one for the basic $la\pi$-calculus presented in Chapter 6, as this would be a good way to justify this work. Part of the motivation of the work in this thesis has been to examine the way in which a mobile process can interact with the environment when arriving at a new location. Therefore, it might be possible to make a detailed model of an industrial mobility package, such as General Magic's Odyssey [LSML00], Sun's JavaSpaces [Mic98] or IBM's Aglets [Lea97]. The Local Area calculus is ideally suited to this, as all of these packages tend to make heavy use of standard libraries and uniform system resources.

7.2.2 Equivalence Methods for Local Areas

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. It would be useful to have a corresponding notion of bi-simulation that filters out actions at some levels and focuses attention on others. Local areas give an opportunity for this to capture spatial information too.

In Chapter 6 we saw that the passive model of FTP behaved in a similar way whether or not it was run over a NAT router. In a way the process with the NAT router is bi-similar to the process with the normal router when you only look at actions that take place at the *app* and *protocol* levels. It would also be useful to detect properties such as a protocol level successfully providing a communication medium for a level above it, or two given levels behaving in the same way and hence being interchangeable.

The major problem with using bi-simulation for a distributed system that encloses some of its actions is that, in any large enough distributed system, all the interesting transitions will be internal. This means that the traditional notions of bi-simulation will miss the very details I am most interested in.

A solution to this may be to let the relation choose which of the communications to observe and which to let happen as silent communication, perhaps tagging τs with the channel and data they pass.

Another alternative might be to look at the actions that could be observed at a number of given points inside the system. This is similar to the idea of Modal Logic that has been applied to the Ambient Calculus [CG00]. If it were possible to get the right level of abstraction, a comparable logic might be interesting.

7.2.3 Model Checking an Extended Case Study

The case study in Chapter 6 was limited in size by what could be presented easily and what could be worked through by hand. Hence, some features had to be cut out of the model. For instance, sockets were identified by their destination port rather than the destination IP address, destination port, source IP address and source port. The application protocols were also cut down in a similar way.

Model Checking software, can automatically check properties of systems modelled using process calculi. Using such an automated system, it would be possible to check more interesting properties of more realistic models. Modifying an existing model checker to handle the $la\pi$-calculus could provide a useful tool with which to examine distributed systems. An excellent starting place might be the SPIN model check [Hol97]. The specification language, PROMELA, has the ability to send and receive over channels and to run processes in parallel, and so seems to be a good match for π like calculi.

An alternative to adapting a model checker to use the $la\pi$-calculus would be to use the encoding of Chapter 5 to translate models into the π-calculus and then to check these π-calculus models, in a model checker, such as Edinburgh Concurrency Workbench [CPS93]. However, this would require adaptation to avoid the inherent divergence in the encoding.

7.2.4 A Language Based on the $la\pi$-calculus

The $la\pi$-calculus aims to capture some of the fundamental ideas used to make distributed computing manageable and convenient. This would seem to make it an excellent basis for a language that would make distributed computing manageable and convenient. The calculus could then be used as a rigorous framework with which to examine programs written in this language. Similar work has been done for the join calculus to make JoCaml [CF99] and the π like Pict [PT00]

The program syntax of Section 6.1 could show the way to turn the calculus into a language. The command like syntax could either form a basis of a language, or, on a smaller scale, an extension to an already existing package. The most interesting question here would be to determine how levels would be classified. One possibility would be to split the levels above and below the host level. A single $la\pi$ server would run on each host. Levels below the host level would be administered by this server and could be used to control threads, resource access, user sessions etc. Levels above the host level would be used to control network structure. These levels could be used to control communication to local and remote hosts in a uniform way. A single local area network could be split into a number of areas that defined, say, which printer is local to each host, which remote backup device a host will use by default. A group of people working on the same project could be defined as being in the same area to give each of their computers access to the same local environment. Areas above the host level could also be used to define Virtual Private Networks or router paths.

7.2.5 Levels and Areas in other Calculi

The local area π-calculus is the π-calculus extended with ideas of areas organised into levels. It is possible to see areas and levels as separate from the π-calculus so it might be interesting to see how this idea could be used with other calculi. Such work might be able to build up Levels and Areas as a design paradigm that could stand alone. It might also provide greater insight into what design decisions should be made for future process calculi.

Bibliography

[Aba99] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[ABL99] Roberto M. Amadio, Gerard Boudol, and Cedric Lhoussaine. The receptive distributed π-calculus. In *Proc of FST-TCS99*, volume 5th Mobile Objects Systems Workshop,LNCS 1738. Springer, 1999.

[ACS98] Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. *Theoretical Computer Science*, 195:291–324, 1998.

[Ama99] Roberto Amadio. On Modelling Mobility. *Theoretical Computer Science*, 1999. Available electonically from Elsevier Preprint.

[AP98] Roberto M. Amadio and Sanjiva Prasad. Modelling IP mobility. In *International Conference on Concurrency Theory*, pages 301–316, 1998.

[BB92] G Berry and G Boudol. The Chemical Abstact Machine. *Theoretical Computer Science*, 1992.

[BCC01] M. Bugliesi, G. Castagna, and S. Crafa. Boxed Ambients. *TACS '01*, 2001.

[BDNN98] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the pi-calculus. In *International Conference on Concurrency Theory*, pages 84–98, 1998.

[Bou92] Gérard Boudol. Asynchrony and the π-Calculus. Rapport de recherche 1702, INRIA, Sophia Antipolis, 1992.

[BPS01] Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors. *Handbook of Process Algebra.* Elsevier, 2001.

[CF99] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile Agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile Ambients. In *Foundations of Software Science and Computation Structure: Proceedings of FoSSaCS '98*, number 1378 in Lecture Notes in Computer Science, pages 140–155. Springer-Verlag, 1998.

[CG00] Luca Cardelli and Andrew D. Gordon. Anytime, Anywhere, Modal Logics for Mobile Ambient. In *POPL'00, Boston, Massachusetts*, pages 365–377. ACM Press, 2000.

[CGG00] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and Group Creation. In *CONCUR '2000: Concurrency Theory. Proceedings of the 11th International Conference*, number 1877 in Lecture Notes in Computer Science, pages 365–379. Springer-Verlag, 2000.

[CGN01] Giuseppe Castagna, Giorgio Ghelli, and Francesco Zappa Nardelli. Typing mobility in the Seal Calculus. *Lecture Notes in Computer Science*, 2154:82–101, 2001.

[CK74] V. Cerf and R. Kahn. A Protocol for Packet Network Interconnection. *IEEE trans. on commun. vol COMM-22*, 1974.

[CM02] Marco Carbone and Sergio Maffeis. On the expressive power of polyadic synchronisation in π-calculus. In *Proceedings of EXPRESS '02: Expressiveness in Concurrency*, Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[CS00] Tom Chothia and Ian Stark. A Distributed π-Calculus with Local Areas of Communication. In *Proceedings of HLCL '00: High-Level Concurrent Languages*, Electronic Notes in Theoretical Computer Science. Elsevier, 2000.

[CS01] Tom Chothia and Ian Stark. Encoding Distributed Areas and Local Communication into the π-calculus. In *Proceedings of EXPRESS '01: Expressiveness in Concurrency*, Electronic Notes in Theoretical Computer Science. Elsevier, 2001.

[CV99] Giuseppe Castagna and Jan Vitek. Seal: A Framework for Secure Mobile Computations. In *Internet Programming Languages*, number 1686 in Lecture Notes in Computer Science, pages 47–77. Springer-Verlag, 1999.

[DH98] S. Deering and R. Hinden. Internet protocol, version 6 (ipv6) specification. *RFC 2460*, 1998.

[Duc02] Daniel Duchamp. The Discrete Internet and What to do About It. In *Proceedings of the Second New York Metro Area Networking Workshop*, September 2002.

[DZ83] J.D. Day and H. Zimmermann. The OSI Reference Model. *Proc. of the IEEE, vol 71*, 1983.

[EF94] K. Egevang and P. Francis. The IP Network Address Translator (NAT). *RFC 1631*, 1994.

[FF86] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217, 1986.

[FG96] Cédric Fournet and Georges 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.

[FG98] Cedric Fournet and Georges Gonthier. A hierarchy of equivalences for asynchronous calculi (extended abstract). In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, volume LNCS 1443, pages pages 844–855. Springer Verlag, July 1998.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A Calculus of Mobile Agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421. Springer-Verlag, 1996.

[FLS00] Cédric Fournet, Jean-Jacques Lévy, and Alain Schmitt. An Asynchronous Distributed Implementation for Mobile Ambients. In *Theoretical Computer Science: Proceedings of TCS 2000*, number 1872 in Lecture Notes in Computer Science, pages 348–364. Springer-Verlag, August 2000.

[FLYK93] V. Fuller, T. Li, J. Yu, and K.Varadhan. Classless Inter-Domain Routing (CIDR). *RFC 1519*, 1993.

[GH99] Simon J. Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *European Symposium on Programming*, volume LNCS 1576, pages 74–90, 1999.

[GJ01] Andrew D. Gordon and Alan Jeffrey. Authenticity by Typing for Security Protocols. In *n Proc. IEEE Computer Security Foundations Workshop,*, pages 145–159. IEEE Press, 2001.

[GS01] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. *ACM SIGPLAN Notices*, 36(3):248–260, 2001.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[HR98a] Matthew Hennessy and James Riely. Resource Access Control in Systems of Mobile Agents. In *Proceedings of HLCL '98: High-Level Concurrent Languages*, number 16.3 in Electronic Notes in Theoretical Computer Science, pages 3–17. Elsevier, 1998.

[HR98b] Matthew Hennessy and James 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.

[HT91] Kohei Honda and Mario Tokoro. An Object Calculus for Asynchronous Communication. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, number 512 in Lecture Notes in Computer Science, pages 133–147. Springer-Verlag, July 1991.

[HVY00] Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In *European Symposium on Programming*, pages 180–199, 2000.

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

[Lea97] Danny B. Lange et al. Aglets:programming mobile agents in java. In *Worldwide Computing and Its Applications*, volume 1274 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[LO99] Danny B. Lange and Mitsuru Oshima. Seven Good Reasons for Mobile agents. *Communications of the ACM*, 42(3):88–89, March 1999.

[LSML00] Jeffrey Lewis, Mark Shields, Erik Meijer, and John Launchbury. Implicit Parameters: Dynamic Scoping with Static Types. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts*, pages 108–118, Jan 2000.

[Mer00] Massimo Merro. *Locality in the π-Calculus and Applications to Distributed Objects*. PhD thesis, Ecole des Mines, France, October 2000.

[Mic98] Sun Microsystems. Javaspace specification, 1998.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil91] Robin Milner. The Polyadic π-Calculus — a Tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Parts I and II. *Information and Computation*, 100:1–77, 1992. Also Laboratory for Foundations of Computer Science Technical Report, ECS-LFCS-89-85 & ECS-LFCS-89-86, 1989.

[NP96] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119, pages 179–194, Pisa, Italy, 1996. Springer-Verlag. Also BRICS Technical Report RS-99-42.

[Pal97] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *Symposium on Principles of Programming Languages*, pages 256–265, 1997.

[Pos80] Jon Postel. User Datagram Protocl. *RFC 768*, 1980.

[Pos81a] Jon Postel. Internet Protocl. *RFC 791*, 1981.

[Pos81b] Jon Postel. Transmission Control Protocl. *RFC 793*, 1981.

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language Based on the Pi-Calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[San93] Davide Sangiorgi. From pi-calculus to higher-order pi-calculus - and back. In *TAPSOFT*, pages 151–166, 1993.

[San95] Davide Sangiorgi. Lazy Functions and Mobile Processes. Rapport de Recherche 2515, INRIA, Sophia Antipolis, November 1995.

[San96] Davide Sangiorgi. Locality and Non-Interleaving Semantics in Calculi for Mobile Processes. *Theoretical Computer Science*, 155:39–83, 1996.

[San97] Davide Sangiorgi. The Name Discipline of Receptiveness. In *Automata, Languages and Programming: Proceedings of the 24th International Colloquium ICALP 97*, number 1256 in Lecture Notes in Computer Science. Springer-Verlag, 1997.

[San99] Davide Sangiorgi. Reasoning about Concurrent Systems using Types. In *Foundations of Software Science and Computation Structure: Proceedings of FoSSaCS '99*, number 1578 in Lecture Notes in Computer Science, pages 31–40. Springer-Verlag, March 1999.

[Sch02] A. Schmitt. Safe Dynamic Binding in the Join Calculus. In *2nd IFIP International Conference on Theoretical Computer Science*. Kluwer, 2002.

[SE01] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). *RFC 3022*, 2001.

[Sew98] Peter 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*, number 1442 in Lecture Notes in Computer Science. Springer-Verlag, 1998.

[SSW01] Andrei Serjantov, Peter Sewell, and Keith Wansbrough. The UDP Calculus: Rigorous Semantics for Real Networking. *TACS '01*, 2001.

[SV99] Peter Sewell and Jan Vitek. Secure composition of insecure components. In *PCSFW: Proceedings of The 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[SV00] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Wrappers and causality types. In *PCSFW: Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Parallel Architectures and Languages Europe*, pages 398–413, 1994.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, Edinburgh University, 1996.

[VD98] José-Luis Vivas and Mads 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*, number 1466 in Lecture Notes in Computer Science. Springer-Verlag, 1998.

[Wei01] Mark Weiser. Whatever happened to the next-generation Internet? *Communications of the ACM*, 44(9):61–69, 2001.

Appendix A

Full Case Study Models

A.1 Basic FTP

$$\text{System} \equiv \begin{array}{l}\text{FTPServerNetwork;}\\ \text{FTPClientNetwork}\end{array}$$

$$\text{FTPClientNetwork} \equiv \quad \underline{\textbf{area}}\ \text{net}\ \{ \quad \begin{array}{l}\text{Router(routerFtpCli);}\\ \text{LookupTable;}\\ \text{ClientTableData;}\\ \underline{\textbf{area}}\ \text{transport}\ \{\ \text{TCP(129.48.32.3,ftpCliHost,routerFtpCli);}\\ \qquad\qquad\qquad\quad \text{FTPclient}\ \}\ \}\end{array}$$

$$\text{FTPServerNetwork} \equiv \quad \underline{\textbf{area}}\ \text{net}\ \{ \quad \begin{array}{l}\text{Router(routerFtpSer);}\\ \text{LookupTable;}\\ \text{ServerTableData;}\\ \underline{\textbf{area}}\ \text{transport}\ \{\ \text{TCP(168.32.16.5),ftpSerHost,routerFtpSer);}\\ \qquad\qquad\qquad\quad \text{FTPserver}\ \}\ \}\end{array}$$

$$\Gamma \equiv \{ \quad \begin{array}{l}\text{Network level names,}\\ \text{Transport level names,}\\ \text{Application level names}\ \}\end{array}$$

Network level names	lookup : LOOKUP	168.32.16.5 : IP	
	129.48.32.3 : IP	255.0.0.1 : IP	
	routerFtpSer:ROUTERNAME	ftpCliHost :ROUTERNAME	
	ftpSerHost :ROUTERNAME	routerFtpCli:ROUTERNAME	

Transport level names	21 : PORT	3020 : PORT
	3021 : PORT	3022 : PORT
	listen:LISTEN	connect:CONNECT
	send : SENT	receive : RECEIVE

Application level names

ftp	: FTP	pasv	:COMMAND
port	:COMMAND	retr	:COMMAND
data.tex:	IP	datafile:	COMMAND
active :	MODE	passive:	MODE
200	:COMMAND	227	:COMMAND
150	:COMMAND		

A.2 FTP with Client Side NAT

NatSystem ≡ FTPServerNetwork;
NatFTPClientNetwork

FTPClientNetwork ≡ **area** net { NatRouter(routerFtpCli);
NatLookupTable;
NatClientTableData;
area transport { TCP(255.0.0.1,ftpCliHost,routerFtpCli);
FTPclient } }

Γ ≡ { Network level names,
Transport level names,
Application level names }

Network level names

lookup	: LOOKUP_NAT	168.32.16.5 :	IP_NAT
129.48.32.3 :	IP_NAT	255.0.0.1 :	IP_NAT
routerFtpSer:	ROUTERNAME	ftpCliHost :	ROUTERNAME
ftpSerHost :	ROUTERNAME	routerFtpCli:	ROUTERNAME

Transport level names

21 :	PORT	3020 :	PORT
3021 :	PORT	3022 :	PORT
listen:	LISTEN	connect:	CONNECT
send :	SENT	receive :	RECEIVE

Application level names

ftp	: FTP	pasv	:COMMAND
port	:COMMAND	retr	:COMMAND
data.tex:	IP_NAT	datafile:	COMMAND
active :	MODE	passive:	MODE
200	:COMMAND	227	:COMMAND
150	:COMMAND		

A.3 Network Level

Router(name) ≡ **spawn** { **in** name ($dip, sip, dport, sport, data$) } **repeat**
{
  new replyRouter **at** $ROUTERNAME$;
  out lookup ($replyRouter, dip$);
  in replyRouter ($forward$);
  out forward ($dip, sip, dport, sport, data$)
}

$LookupTable$ ≡ **spawn** { **in** lookup ($replyRouter, dip, sip$) } **repeat**
{ **out** dip ($replyRouter$) };
TableData

$ClientTableData$ ≡ **spawn** { **in** 168.32.16.5 ($replyRouter$) } **repeat**
{ **out** replyRouter ($routerFtpSer$) };
spawn { **in** 129.48.32.3 ($replyRouter$) } **repeat**
{ **out** replyRouter ($ftpCliHost$) }

$ServerTableData$ ≡ **spawn** { **in** 168.32.16.5 ($replyRouter$) } **repeat**
{ **out** replyRouter ($ftpSerHost$) } ;
spawn { **in** 129.48.32.3 ($replyRouter$) } **repeat**
{ **out** replyRouter ($routerFtpCli$)}

$NATLookupTable \equiv$ **spawn** { **in** lookup (*replyRouter*, *dip*, *sip*) } **repeat**
new same **at** *LOOKUPREP_NAT*;
new swap **at** *LOOKUPREP_NAT*;
out dip (*same*, *swap*, _);
spawn { **in** same (*forward*, *newdip*) }
{ **out** replyRouter (*forward*, *newdip*, *sip*) } ;
spawn { **in** swap (*forward*, *newdip*) }
{
out sip (*same*);
in same (_, *newsip*);
out replyRouter (*forward*, *newdip*, *newsip*)
};
TableData;

$ClientNATtableData \equiv$ **spawn** { **in** 168.32.16.5 (*same*, *swap*) } **repeat**
{ **out** swap (*routerFtpSer*, *168.32.16.5*) };
spawn { **in** 129.48.32.3 (*same*, *swap*) } **repeat**
{ **out** same (*ftpCliHost*, *255.0.0.1*) };
spawn { **in** 255.0.0.1 (*same*, *swap*) } **repeat**
{ **out** same (*ftpCliHost*, *129.48.32.3*) }

$ClientNATtableData \equiv$ $ServerTableData$

A.4 Transport Level

IPlinkSOCKET(routerIn,routerOut) ≡ **spawn** { **in** routerIn (*dip*, *sip*, *dport*, *sport*, *data*) } **repeat**
{
out dport (*dip*, *sip*, *dport*, *sport*, *data*)
}

Open(sip) ≡ **spawn** { **in** listen (*portNo*, *reply*) } **repeat**
{
new socket **at** *SOCKET*;
in portNo (*dip*, *sip*, *dport*, *sport*, *data*);
out reply (*socket*);
out portNo (*dip*, *sip*, *dport*, *sport*, *data*);
spawn { **in** socket (*n*) } **repeat**
{**out** n (*sip*, *dip*, *sport*, *dport*)}
};
spawn { **in** connect (*dip*, *dport*, *sport*, *reply*) } **repeat**
{
new socket **at** *SOCKET*;
out reply (*socket*);
spawn { **in** socket (*n*) } **repeat**
{**out** n (*dip*, *sip*, *dport*, *sport*)}
}

```
Send(routerOut) ≡   spawn {  in   send (socket, data)  } repeat
                      {
                          new   n at SOCKETINFO;
                          out   socket (n);
                          in    n (dip, sip, dport, sport);
                          out   routerOut (dip, sip, dport, sport, data)
                      }

Receive ≡   spawn {  in   receive (socket, reply)  } repeat
              {
                  new   n at SOCKETINFO
                  out   socket (n);
                  in    n (_, _, _, sport);
                  in    sport (_, _, _, _, data);
                  out   reply (data)
              }

TCP(sip,routerIn,routerOut) ≡   IPlinkSOCKET(routerIn);
                                Open(sip);
                                Send(routerOut);
                                Receive
```

A.5 Application Level

```
FTPclient ≡ area app {   in     ftp (ip, filename, mode, filerep);
                         new    socketReply at SOCKETREP;
                         out    connect (ip, 21, 3022, socketReply);
                         in     socketReply (socket);
                         new    replyCli at DATAREP;
                         out    mode ();
                         spawn {  in   active ()  }
                           {
                               out   send (socket, (port, ip, 3021));
                               out   receive (socket, replyCli);
                               in    replyCli (200);
                               out   send (socket, (retr, filename));
                               out   receive (socket, replyCli);
                               in    replyCli (150);
                               out   listen (socketReply, 3021);
                               in    socketReply (dataSocket);
                               out   receive (dataSocket, filerep)
                           };
                         spawn {  in   passive ()  }
                           {
                               out   send (socket, (passive));
                               out   receive (socket, replyCli);
                               in    replyCli (227, serverIp, serverPort);
                               out   send (socket, (retr, filename));
                               out   receive (socket, replyCli);
                               in    replyCli (150);
                               out   connect (serverIp, serverPort, 3020, socketReply);
                               in    socketReply (dataSocket);
                               out   send (dataSocket);
                               out   receive (dataSocket, filerep)
                           }
                       }
```

```
FTPserver(ftpip) ≡ area app {
    new    socketReply at SOCKETREP;
    new    replySer at DATAREP;
    out    listen (21, socketReply);
    in     socketReply (serverSocket);
    out    receive (serverSocket, replySer);
    spawn { in   replySer (command, arg1, arg2) }   repeat
      {
        out   command (arg1, arg2);
        out   receive (serverSocket, replySer)
      };
    spawn { in   port (newIp, newPort) }   repeat
      {
        out   storeIp (newIP);
        out   storePort (newPort);
        out   active ();
        out   send (serverSocket, 200)
      };
    spawn { in   pasv () }   repeat
      {
        out   pastive ();
        out   send (serverSocket, (227, ftpip, 3021))
      };
    spawn { in   retr (filename) }   repeat
      {
        in    storeIp (ip);
        in    storePort (port);
        new   rep at SOCKETREP;
        out   send (serverSocket, 150);
        spawn { in   active () }
            {
              out   storeIP (3021);
              out   connect (ip, port, 3020, socketReply);
              in    socketReply (dataSocket);
              out   send (dataSocket, file)
            };
        spawn { in   passive () }
            {
              out   listen (port, socketReply);
              in    socketReply (dataSocket);
              out   receive (replySer);
              out   send (dataSocket, file);
            };
      };
}
```

Appendix B

Full Model Traces

B.1 Active Mode without NAT

The following is the full trace of the file transfer between a client running on host 168.32.16.5 and a server at 129.48.32.3

Numbers indicate output actions and spawned processes that are picked up later in the trace. Letters are points of synchronisation between traces and mark the order in which the traces may happen. The FTP client starts and connects:

	FTP user	$\rightarrow$	client App	: ftp(168.32.16.5 , data.tex , active , filerep)
1	client App	$\rightarrow$	client Transport	: connect(168.32.16.5 , 21 , 3022 , socketReply)
	client Transport	$\rightarrow$	client App	: socketReply(ftpCliSocket)
	client App	$\rightarrow$	client App	: active
♣♣	client App	$\rightarrow$	client Transport	: send(ftpCliSocket , (port , 129.48.32.3 , 3021))
	client Transport	$\rightarrow$	client Transport	: ftpCliSocket(n)
	client Transport	$\rightarrow$	client Transport	: n(168.32.16.5 , 129.48.32.3 , 21 , 3022)
	client Transport	$\rightarrow$	client Router	: routerFtpCli(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (port , 129.48.32.3 , 3021))
	client Router	$\rightarrow$	client Lookup	: lookup(replyRouter , 168.32.16.5)
	client Lookup	$\rightarrow$	client Lookup	: 168.32.16.5(replyRouter)
	client Lookup	$\rightarrow$	client Router	: replyRouter(routerFtpSer)
	client Router	$\rightarrow$	Server router	: routerFtpSer(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (port , 129.48.32.3 , 3021))
	server Router	$\rightarrow$	server Lookup	: lookup(replyRouter , 168.32.16.5)
	server Lookup	$\rightarrow$	server Lookup	: 168.32.16.5(replyRouter)
	server Lookup	$\rightarrow$	server Router	: replyRouter(ftpSerHost)
	server Router	$\rightarrow$	server Transport	: ftpSerHost(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (port , 129.48.32.3 , 3021))
A	server Transport	$\rightarrow$	server Transport	: 21(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (port , 129.48.32.3 , 3021))

At ♣♣ the client may start to listen for a reply. The server listens for a connection:

♢	server App	→	server Transport	: listen(21 , replySocket)
$A2$	server Transport	→	server Transport	: 21(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (port , 129.48.32.3 , 3021))
	server Transport	→	server App	: replySocket(ftpSerSocket)
	server App	→	server Transport	: receive(ftpSerSocket , replySer)
	server Transport	→	server Transport	: ftpSerSocket(n)
	server Transport	→	server Transport	: n(129.48.32.3 , 168.32.16.5 , 3022 , 21)
	server Transport	→	server Transport	: 21(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (port , 129.48.32.3 , 3021))
	server Transport	→	server App	: replySer(port , 129.48.32.3 , 3021)
♡♡3	server App	→	server App	: port(129.48.32.3 , 3021)

♢: may happen at any time between the start and when it is given. The two traces above synchronise on A. At ♡♡ the server may start to listen for new command. The acknowledgement is then passed back to the client:

	server App	→	server Transport	: send(ftpSerSocket , 200)
	server Transport	→	server Transport	: ftpSerSocket(n)
	server Transport	→	server Transport	: n(129.48.32.3 , 168.32.16.5 , 3022 , 21)
	server Transport	→	server Router	: routerFtpSer(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 200)
	server Router	→	server Lookup	: lookup(replyRouter , 129.48.32.3)
	server Lookup	→	server Lookup	: 129.48.32.3 , 21(replyRouter)
	server Lookup	→	server Router	: replyRouter(routerFtpCli)
	server Router	→	client Router	: routerFtpCli(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 200)
	client Router	→	client Lookup	: lookup(replyRouter , 129.48.32.3)
	client Lookup	→	client Lookup	: 129.48.32.3(replyRouter)
	client Lookup	→	client Router	: replyRouter(ftpCliHost)
	client Router	→	client Transport	: ftpCliHost(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 200)
B	client Transport	→	client Transport	: 3022(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 200)

The client listens for an acknowledgement:

♣	client App	→	client Transport	: receive(ftpCliSocket , replyCli)
♣	client Transport	→	client Transport	: ftpCliSocket(n)
♣	client Transport	→	client Transport	: n(168.32.16.5 , 129.48.32.3 , 21 , 3022)
B	client Transport	→	client Transport	: 3022(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 200)
	client Transport	→	client App	: replyCli(200)

♣ : may happen at any time between ♣♣ and when it is given. The two traces above synchronise on B. The client sends the *retr* command:

♠♠	client App	→	client Transport	:	send(ftpCliSocket , (retr , data.tex))
	client Transport	→	client Transport	:	ftpCliSocket(n)
	client Transport	→	client Transport	:	n(168.32.16.5 , 129.48.32.3 , 21 , 3022)
	client Transport	→	client Router	:	routerFtpCli(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				:	(retr , data.tex))
	client Router	→	client Lookup	:	lookup(replyRouter , 168.32.16.5)
	client Lookup	→	client Lookup	:	168.32.16.5(replyRouter)
	client Lookup	→	client Router	:	replyRouter(routerFtpSer)
	client Router	→	server Router	:	routerFtpSer(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				:	(retr , data.tex))
	server Router	→	server Lookup	:	lookup(replyRouter , 168.32.16.5)
	server Lookup	→	server Lookup	:	168.32.16.5(replyRouter)
	server Lookup	→	server Router	:	replyRouter(ftpSerHost)
	server Router	→	server Transport	:	ftpSerHost(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				:	(retr , data.tex))
C	server Transport	→	server Transport	:	21(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				:	(retr , data.tex))

At ♠♠ the client may start to listen for a reply. The server listens for a FTP command:

♡	server App	→	server Transport	:	receive(ftpSerSocket , replySer)
♡	server Transport	→	server Transport	:	ftpSerSocket(n)
♡	server Transport	→	server Transport	:	n(129.48.32.3 , 168.32.16.5 , 3022 , 21)
C	server Transport	→	server Transport	:	21(168.32.16.5 , 129.48.32.3 , 21 , 3022(retr , data.tex))
	server Transport	→	server App	:	replySer(retr , data.tex)
	server App	→	server App	:	retr(data.tex)

♡ : may happen at any time between ♡♡ and when it is given. The two traces above synchronise on C. At anytime from this point on the FTP server can ask the transport level for another command, that never comes:

server App	→	server Transport	:	receive(ftpSerSocket , replySer)
server Transport	→	server Transport	:	ftpSerSocket(n)
server Transport	→	server Transport	:	n(129.48.32.3 , 168.32.16.5 , 3022 , 21)

The FTP server now processes the *retr* command:

	server App	→	server App	:	storePort(3021)
QQ	server App	→	server Transport	:	send(ftpSerSocket , 150)

The server may enter active mode at any time after QQ. The acknowledgement is passed back to the client:

	server Transport	→	server Transport	:	ftpSerSocket(n)
	server Transport	→	server Transport	:	n(129.48.32.3 , 168.32.16.5 , 3022 , 21)
	server Transport	→	server Router	:	routerFtpSer(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 150)
	server Router	→	server Lookup	:	lookup(replyRouter , 129.48.32.3)
	server Lookup	→	server Lookup	:	129.48.32.3(replyRouter)
	server Lookup	→	server Router	:	replyRouter(routerFtpCli)
	server Router	→	client Router	:	routerFtpCli(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 150)
	client Router	→	client Lookup	:	lookup(replyRouter , 129.48.32.3)
	client Lookup	→	client Lookup	:	129.48.32.3(replyRouter)
	client Lookup	→	client Router	:	replyRouter(ftpCliHost)
	client Router	→	client Transport	:	ftpCliHost(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 150)
D	client Transport	→	client Transport	:	3022(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 150)

The client listens for an acknowledgement:

♠	client App	→	client Transport :	receive(ftpCliSocket , replyCli)
♠	client Transport	→	client Transport :	ftpCliSocket(n)
♠	client Transport	→	client Transport :	n(168.32.16.5 , 129.48.32.3 , 21 , 3022)
D	client Transport	→	client Transport :	3022(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 150)
	client Transport	→	client App :	replyCli(150)
	client App	→	client Transport :	listen(3021 , socketReply)

♠ : may happen at any time between ♠♠ and when it is given. The two traces above synchronise on D. The server (which is processing the retr command) now selects active mode. This trace can happen at anytime between now and QQ:

	server App	→	server App :	active()
	server App	→	server App :	storeIp(129.48.32.3)
4	server App	→	server Transport :	connect(129.48.32.2 , 3021 , 3020 , reply)
	server Transport	→	server App :	reply(ftpSerDataSocket)

The file is now sent to the client:

	server App	→	server Transport :	send(ftpSerDataSocket , dataFile)
	server Transport	→	server Transport :	ftpSerDataSocket(n)
	server Transport	→	server Transport :	n(129.48.32.3 , 168.32.16.5 , 3021 , 3020)
	server Transport	→	server Router :	routerFtpSer(129.48.32.3 , 168.32.16.5 ,
			:	3021 , 3020 , dataFile)
	server Router	→	server Lookup :	lookup(replyRouter , 129.48.32.3)
	server Lookup	→	server Lookup :	129.48.32.3(replyRouter)
	server Lookup	→	server Router :	replyRouter(routerFtpCli)
	server Router	→	client Router :	routerFtpCli(129.48.32.3 , 168.32.16.5 ,
			:	3021 , 3020 , dataFile)
	client Router	→	client Lookup :	lookup(replyRouter , 129.48.32.3)
	client Lookup	→	client Lookup :	129.48.32.3(replyRouter)
	client Lookup	→	client Router :	replyRouter(ftpCliHost)
	client Router	→	client Transport :	ftpCliHost(129.48.32.3 , 168.32.16.5 ,
			:	3021 , 3020 , dataFile)
E	client Transport	→	client Transport :	3021(129.48.32.3 , 168.32.16.5 , 3021 , 3020 , dataFile)

The data connection is opened and the file received:

5	client Transport	→	client App :	socketReply(ftpCliDataSocket)
	client App	→	client Transport :	receive(ftpCliDataSocket , filerep)
	client Transport	→	client Transport :	ftpCliDataSocket(n)
	client Transport	→	client Transport :	n(168.32.16.5 , 129.48.32.3 , 3020 , 3021)
E	client Transport	→	client Transport :	3021(129.48.32.3 , 168.32.16.5 , 3021 , 3020 , dataFile)
	client Transport	→	outside processes :	fileReply(dataFile)

The two traces above synchronise on E.

Output and links:

- 1 links ftpCliSocket to (168.32.16.5 , 129.48.32.3 , 21 , 3022)
- 2 links ftpSerSocket to (129.48.32.3 , 168.32.16.5 , 3022 , 21)
- 3 outputs 129.48.32.2 on storeIP , 3021 on storePort and sets the active flag.
- 4 links ftpSerDataSocket to (129.48.32.2 , 168.32.16.5 , 3021 , 3020)
- 5 links ftpCliDataSocket to (168.32.16.5 , 129.48.32.2 , 3020 , 3021)

B.2 Active Mode with NAT

In this section, the client network is using Network Address Transation. The internal address 255.0.0.1 is mapped to the external address 129.48.32.3. The server remains at 168.32.16.5. Numbers indicate output actions and spawned processes that are picked up later in the trace. Letters are points of synchronisation between traces and mark the order in which the traces may happen. The clients starts and connects:

	FTP user	→	client App	: ftp(168.32.16.5 , data.tex , active)
1	client App	→	client Transport	: connect(168.32.16.5 , 21 , 3022 , socketReply)
	client Transport	→	client App	: socketReply(ftpCliSocket)
	client App	→	client App	: active
♣♣	client App	→	client Transport	: send(ftpCliSocket , (port , 255.0.0.1 , 3021))
	client Transport	→	client Transport	: ftpCliSocket(n)
	client Transport	→	client Transport	: n(168.32.16.5 , 255.0.0.1 , 21 , 3022)
	client Transport	→	client Router	: routerFtpCli(168.32.16.5 , 21 , 255.0.0.1 , 21 , 3022 ,
				: (port , 255.0.0.1 , 3021))
	client Router	→	client Lookup	: lookup(replyRouter , 168.32.16.5 , 255.0.0.1)
	client Lookup	→	client Lookup	: 168.32.16.5(same , swap)
	client Lookup	→	client Lookup	: swap(routerFtpSer , 168.32.16.5)
	client Lookup	→	client Lookup	: 255.0.0.1(same)
	client Lookup	→	client Lookup	: same(_129.48.32.3)
	client Lookup	→	client Router	: replyRouter(routerFtpSer , 168.32.16.5 , 129.48.32.3)
	client Router	→	server Router	: routerFtpSer(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (port , 255.0.0.1 , 3021))
	server Router	→	server Lookup	: lookup(replyRouter , 168.32.16.5)
	server Lookup	→	server Lookup	: 168.32.16.5(replyRouter)
	server Lookup	→	server Router	: replyRouter(ftpSerHost)
	server Router	→	server Transport	: ftpSerHost(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (port , 255.0.0.1 , 3021))
A	server Transport	→	server Transport	: 21(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (port , 255.0.0.1 , 3021))

At ♣♣ the client may start to listen for a reply. The server listens for a connection:

◇	server App	→	server Transport	: listen(21 , replySocket)
$A2$	server Transport	→	server Transport	: 21(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (port , 255.0.0.1 , 3021))
	server Transport	→	server App	: replySocket(ftpSerSocket)
	server App	→	server Transport	: receive(ftpSerSocket , replySer)
	server Transport	→	server Transport	: ftpSerSocket(n)
	server Transport	→	server Transport	: n(129.48.32.3 , 168.32.16.5 , 3022 , 21)
	server Transport	→	server Transport	: 21(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (port , 255.0.0.1 , 3021))
	server Transport	→	server App	: replySer(port , 255.0.0.1 , 3021)
♡♡3	server App	→	server App	: port(129.48.32.3 , 3021)

◇: may happen at any time between the start and when it is given. The two traces above synchronise on A. At ♡♡ the server may start to listen for a new command. The acknowledgement is then passed back to the client:

	server App	→	server Transport	: send(ftpSerSocket , 200)
	server Transport	→	server Transport	: ftpSerSocket(n)
	server Transport	→	server Transport	: n(129.48.32.3 , 168.32.16.5 , 3022 , 21)
	server Transport	→	server Router	: routerFtpSer(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 200)
	server Router	→	server Lookup	: lookup(replyRouter , 129.48.32.3)
	server Lookup	→	server Lookup	: 129.48.32.3 , 21(replyRouter)
	server Lookup	→	server Router	: replyRouter(routerFtpCli)
	server Router	→	client Router	: routerFtpCli(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 200)
	client Router	→	client Lookup	: lookup(replyRouter , 129.48.32.3 , 168.32.16.5)
	client Lookup	→	client Lookup	: 129.48.32.3(same , swap)
	client Lookup	→	client Lookup	: same(ftpCliHost , 255.0.0.1)
	client Lookup	→	client Router	: replyRouter(ftpCliHost , 255.0.0.1 , 168.32.16.5)
	client Router	→	client Transport	: ftpCliHost(255.0.0.1 , 168.32.16.5 , 3022 , 21 , 200)
B	client Transport	→	client Transport	: 3022(255.0.0.1 , 168.32.16.5 , 3022 , 21 , 200)

The client listens for the acknowledgement:

♣	client App	→	client Transport	: receive(ftpCliSocket , replyCli)
♣	client Transport	→	client Transport	: ftpCliSocket(n)
♣	client Transport	→	client Transport	: n(168.32.16.5 , 255.0.0.1 , 21 , 3022)
B	client Transport	→	client Transport	: 3022(255.0.0.1 , 168.32.16.5 , 3022 , 21 , 200)
	client Transport	→	client App	: replyCli(200)

♣ : may happen at any time between ♣♣ and when it is given. The two traces above synchronise on B. The client then sends the *retr* command:

♠♠	client App	→	client Transport	: send(ftpCliSocket , (retr , data.tex))
	client Transport	→	client Transport	: ftpCliSocket(n)
	client Transport	→	client Transport	: n(168.32.16.5 , 255.0.0.1 , 21 , 3022)
	client Transport	→	client Router	: routerFtpCli(168.32.16.5 , 255.0.0.1 , 21 , 3022 ,
				: (retr , data.tex))
	client Router	→	client Lookup	: lookup(replyRouter , 168.32.16.5 , 255.0.0.1)
	client Lookup	→	client Lookup	: 168.32.16.5(same , swap)
	client Lookup	→	client Lookup	: swap(routerFtpSer , 168.32.16.5)
	client Lookup	→	client Lookup	: 255.0.0.1(same)
	client Lookup	→	client Lookup	: same(_129.48.32.3)
	client Router	→	client Lookup	: lookup(replyRouter , 168.32.16.5)
	client Lookup	→	client Router	: replyRouter(routerFtpSer)
	client Router	→	server Router	: routerFtpSer(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (retr , data.tex))
	server Router	→	server Lookup	: lookup(replyRouter , 168.32.16.5)
	server Lookup	→	server Lookup	: 168.32.16.5(replyRouter)
	server Lookup	→	server Router	: replyRouter(ftpSerHost)
	server Router	→	server Transport	: ftpSerHost(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (retr , data.tex))
C	server Transport	→	server Transport	: 21(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				: (retr , data.tex))

At ♠♠ the client may start to listen for a reply. The server listens for a FTP command:

$\heartsuit$	server App	$\rightarrow$	server Transport	: receive(ftpSerSocket , replySer)
$\heartsuit$	server Transport	$\rightarrow$	server Transport	: ftpSerSocket(n)
$\heartsuit$	server Transport	$\rightarrow$	server Transport	: n(129.48.32.3 , 168.32.16.5 , 3022 , 21)
C	server Transport	$\rightarrow$	server Transport	: 21(168.32.16.5 , 129.48.32.3 , 21 , 3022 , (retr , data.tex))
	server Transport	$\rightarrow$	server App	: replySer(retr , data.tex)
	server App	$\rightarrow$	server App	: retr(data.tex)

$\heartsuit$: may happen at any time between $\heartsuit\heartsuit$ and when it is given. The two traces above synchronise on C. At this point or at anytime later in the trace the FTP server can ask the transport level for another command, that never comes:

server App	$\rightarrow$	server Transport	: receive(ftpSerSocket , replySer)
server Transport	$\rightarrow$	server Transport	: ftpSerSocket(n)
server Transport	$\rightarrow$	server Transport	: n(129.48.32.3 , 168.32.16.5 , 3022 , 21)

The FTP server now processes the *retr* command:

	server App	$\rightarrow$	server App	: storePort(3021)
QQ	server App	$\rightarrow$	server Transport	: send(ftpSerSocket , 150)

The server may enter active mode at any time after QQ. The acknowledgement is then passed back to the client:

	server Transport	$\rightarrow$	server Transport	: ftpSerSocket(n)
	server Transport	$\rightarrow$	server Transport	: n(129.48.32.3 , 168.32.16.5 , 3022 , 21)
	server Transport	$\rightarrow$	server Router	: routerFtpSer(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 150)
	server Router	$\rightarrow$	server Lookup	: lookup(replyRouter , 129.48.32.3)
	server Lookup	$\rightarrow$	server Lookup	: 129.48.32.3(replyRouter)
	server Lookup	$\rightarrow$	server Router	: replyRouter(routerFtpCli)
	server Router	$\rightarrow$	client Router	: routerFtpCli(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 150)
	client Router	$\rightarrow$	client Lookup	: lookup(replyRouter , 129.48.32.3 , 168.32.16.5)
	client Lookup	$\rightarrow$	client Lookup	: 129.48.32.3(same , swap)
	client Lookup	$\rightarrow$	client Lookup	: same(ftpCliHost , 255.0.0.1)
	client Lookup	$\rightarrow$	client Router	: replyRouter(ftpCliHost , 255.0.0.1 , 168.32.16.5)
	client Router	$\rightarrow$	client Transport	: ftpCliHost(255.0.0.1 , 168.32.16.5 , 3022 , 21 , 150)
D	client Transport	$\rightarrow$	client Transport	: 3022(255.0.0.1 , 168.32.16.5 , 3022 , 21 , 150)

The client listens for an acknowledgement:

$\spadesuit$	client App	$\rightarrow$	client Transport	: receive(ftpCliSocket , replyCli)
$\spadesuit$	client Transport	$\rightarrow$	client Transport	: ftpCliSocket(n)
$\spadesuit$	client Transport	$\rightarrow$	client Transport	: n(168.32.16.5 , 255.0.0.1 , 21 , 3022)
D	client Transport	$\rightarrow$	client Transport	: 3022(255.0.0.1 , 168.32.16.5 , 3022 , 21 , 150)
	client Transport	$\rightarrow$	client App	: replyCli(150)

$\spadesuit$: may happen at any time between $\spadesuit\spadesuit$ and when it is given. The two traces above synchronise on D. The server selects active mode and attempts to send the file:

server App	→	server App	: active()
server App	→	server App	: storeIP(255.0.0.1)
server App	→	server Transport	: connect(255.0.0.1 , 3021 , 3020 , reply)
server Transport	→	server App	: reply(ftpSerDataSocket)
server App	→	server Transport	: send(ftpSerDataSocket , dataFile)
server Transport	→	server Transport	: ftpSerDataSocket(n)
server Transport	→	server Transport	: n(255.0.0.1 , 168.32.16.5 , 3021 , 3020)
server Transport	→	server Router	: routerFtpSer(255.0.0.1 , 168.32.16.5 ,
			: 3021 , 3020 , dataFile)
server Router	→	server Lookup	: lookup(replyRouter , 255.0.0.1)

The server now blocks on the output of 255.0.0.1. The client may still execute the 3 actions needed to listen for the connection, but the packet from the server will never arrive.

Output and links:

- 1 links ftpCliSocket to (168.32.16.5 , 255.0.0.1 , 21 , 3022)
- 2 links ftpSerSocket to (129.48.32.3 , 168.32.16.5 , 3022 , 21)
- 3 outputs 255.0.0.1 on storeIP , 3021 on storePort and sets the active flag.
- 4 links ftpSerDataSocket to (255.0.0.1 , 168.32.16.5 , 3021 , 3020)

B.3 Passive Mode with NAT

In this section, the client uses passive mode to overcome the problems posed by NAT. The set up is the same as before; the client uses the internal address 255.0.0.1 which is mapped to the external address 129.48.32.3. The server remains at 168.32.16.5. The only difference, this time, is that the user request that the client uses passive mode. The clients starts and connects:

	user	→	client App	: ftp(168.32.16.5 , data.tex , passive , filerep)
1	client App	→	client Transport	: connect(168.32.16.5 , 21 , 3022 , socketReply)
	client Transport	→	client App	: socketReply(ftpCliSocket)
	client App	→	client App	: passive

The selection of passive mode is the fist difference between this trace and the last. As a result the client will send the *pasv* command rather than the *port* command:

♣♣	client App	→	client Transport	: send(ftpCliSocket , pasv)
	client Transport	→	client Transport	: ftpCliSocket(n)
	client Transport	→	client Transport	: n(168.32.16.5 , 255.0.0.1 , 21 , 3022)
	client Transport	→	client Router	: routerFtpCli(168.32.16.5 , 21 , 255.0.0.1 ,
				: 21 , 3022 , pasv)
	client Router	→	client Lookup	: lookup(replyRouter , 168.32.16.5 , 255.0.0.1)
	client Lookup	→	client Lookup	: 168.32.16.5(same , swap)
	client Lookup	→	client Lookup	: swap(routerFtpSer , 168.32.16.5)
	client Lookup	→	client Lookup	: 255.0.0.1(same)
	client Lookup	→	client Lookup	: same(_ , 129.48.32.3)
	client Lookup	→	client Router	: replyRouter(routerFtpSer , 168.32.16.5 , 129.48.32.3)
	client Router	→	server Router	: routerFtpSer(168.32.16.5 , 129.48.32.3 ,
				: 21 , 3022 , pasv)
	server Router	→	server Lookup	: lookup(replyRouter , 168.32.16.5)
	server Lookup	→	server Lookup	: 168.32.16.5(replyRouter)
	server Lookup	→	server Router	: replyRouter(ftpSerHost)
	server Router	→	server Transport	: ftpSerHost(168.32.16.5 , 129.48.32.3 , 21 , 3022 , pasv)
A	server Transport	→	server Transport	: 21(168.32.16.5 , 129.48.32.3 , 21 , 3022 , pasv)

At ♣♣ the client may start to listen for a reply. The server listens for a connection:

♢	server App	→	server Transport	:	listen(21 , replySocket)
A , 2	server Transport	→	server Transport	:	21(168.32.16.5 , 129.48.32.3 , 21 , 3022 , pasv)
	server Transport	→	server App	:	replySocket(ftpSerSocket)
	server App	→	server Transport	:	receive(ftpSerSocket , replySer)
	server Transport	→	server Transport	:	ftpSerSocket(n)
	server Transport	→	server Transport	:	n(129.48.32.3 , 168.32.16.5 , 3022 , 21)
	server Transport	→	server Transport	:	21(168.32.16.5 , 129.48.32.3 , 21 , 3022 , pasv)
	server Transport	→	server App	:	replySer(pasv)
♡♡ , 3	server App	→	server App	:	pasv()

♢: may happen at any time between the start of the trace and when it is given. The two traces above synchronise on A. At ♡♡ the server may start to listen for a new command. The server then stores 3021 as the future data port choice, and sends back the 227 acknowledgement: "entering passive mode using 168.32.16.5:3021":

	server App	→	server Transport	:	send(ftpSerSocket ,
				:	(227 , 168.32.16.5 , 3021))
	server Transport	→	server Transport	:	ftpSerSocket(n)
	server Transport	→	server Transport	:	n(129.48.32.3 , 168.32.16.5 , 3022 , 21)
	server Transport	→	server Router	:	routerFtpSer(129.48.32.3 , 168.32.16.5 , 3022 , 21 ,
				:	(227 , 168.32.16.5 , 3021))
	server Router	→	server Lookup	:	lookup(replyRouter , 129.48.32.3)
	server Lookup	→	server Lookup	:	129.48.32.3 , 21(replyRouter)
	server Lookup	→	server Router	:	replyRouter(routerFtpCli)
	server Router	→	client Router	:	routerFtpCli(129.48.32.3 , 168.32.16.5 , 3022 , 21 ,
				:	(227 , 168.32.16.5 , 3021))
	client Router	→	client Lookup	:	lookup(replyRouter , 129.48.32.3 , 168.32.16.5)
	client Lookup	→	client Lookup	:	129.48.32.3(same , swap)
	client Lookup	→	client Lookup	:	same(ftpCliHost , 255.0.0.1)
	client Lookup	→	client Router	:	replyRouter(ftpCliHost , 255.0.0.1 , 168.32.16.5)
	client Router	→	client Transport	:	ftpCliHost(255.0.0.1 , 168.32.16.5 , 3022 , 21 ,
				:	(227 , 168.32.16.5 , 3021))
B	client Transport	→	client Transport	:	3022(255.0.0.1 , 168.32.16.5 , 3022 , 21 ,
				:	(227 , 168.32.16.5 , 3021))

The client listens for the acknowledgement:

♣	client App	→	client Transport	:	receive(ftpCliSocket , replyCli)
♣	client Transport	→	client Transport	:	ftpCliSocket(n)
♣	client Transport	→	client Transport	:	n(168.32.16.5 , 255.0.0.1 , 21 , 3022)
B	client Transport	→	client Transport	:	3022(255.0.0.1 , 168.32.16.5 , 3022 , 21 ,
				:	(227 , 168.32.16.5 , 3021))
	client Transport	→	client App	:	replyCli(227 , 168.32.16.5 , 3021)

♣ : may happen at any time between ♣♣ and when it is given. The two traces above synchronise on B. The client sends the *retr* command:

♠♠	client App	→	client Transport	:	send(ftpCliSocket , (retr , data.tex))
	client Transport	→	client Transport	:	ftpCliSocket(n)
	client Transport	→	client Transport	:	n(168.32.16.5 , 255.0.0.1 , 21 , 3022)
	client Transport	→	client Router	:	routerFtpCli(168.32.16.5 , 255.0.0.1 , 21 , 3022 ,
				:	(retr , data.tex))
	client Router	→	client Lookup	:	lookup(replyRouter , 168.32.16.5 , 255.0.0.1)
	client Lookup	→	client Lookup	:	168.32.16.5(same , swap)
	client Lookup	→	client Lookup	:	swap(routerFtpSer , 168.32.16.5)
	client Lookup	→	client Lookup	:	255.0.0.1(same)
	client Lookup	→	client Lookup	:	same(_ , 129.48.32.3)
	client Router	→	client Lookup	:	lookup(replyRouter , 168.32.16.5)
	client Lookup	→	client Router	:	replyRouter(routerFtpSer)
	client Router	→	server Router	:	routerFtpSer(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				:	(retr , data.tex))
	server Router	→	server Lookup	:	lookup(replyRouter , 168.32.16.5)
	server Lookup	→	server Lookup	:	168.32.16.5(replyRouter)
	server Lookup	→	server Router	:	replyRouter(ftpSerHost)
	server Router	→	server Transport	:	ftpSerHost(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				:	(retr , data.tex))
C	server Transport	→	server Transport	:	21(168.32.16.5 , 129.48.32.3 , 21 , 3022 ,
				:	(retr , data.tex))

At ♠♠ the client may start to listen for a reply. The server listens for a FTP command:

♡	server App	→	server Transport	:	receive(ftpSerSocket , replySer)
♡	server Transport	→	server Transport	:	ftpSerSocket(n)
♡	server Transport	→	server Transport	:	n(129.48.32.3 , 168.32.16.5 , 3022 , 21)
C	server Transport	→	server Transport	:	21(168.32.16.5 , 129.48.32.3 , 21 , 3022(retr , data.tex))
	server Transport	→	server App	:	replySer(retr , data.tex)
	server App	→	server App	:	retr(data.tex)

♡ : may happen at any time between ♡♡ and when it is given. The two traces above synchronise on C. At this point or at anytime later in the trace, the FTP server can ask the transport level for another command, that never comes:

server App	→	server Transport	:	receive(ftpSerSocket , replySer)
server Transport	→	server Transport	:	ftpSerSocket(n)
server Transport	→	server Transport	:	n(129.48.32.3 , 168.32.16.5 , 3022 , 21)

The FTP server now processes the *retr* command:

	server App	→	server App	:	storePort(3021)
QQ	server App	→	server Transport	:	send(ftpSerSocket , 150)

The server may enter passive mode at any time after QQ. The acknowledgement is then passed back to the client:

	server Transport	→	server Transport	:	ftpSerSocket(n)
	server Transport	→	server Transport	:	n(129.48.32.3 , 168.32.16.5 , 3022 , 21)
	server Transport	→	server Router	:	routerFtpSer(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 150)
	server Router	→	server Lookup	:	lookup(replyRouter , 129.48.32.3)
	server Lookup	→	server Lookup	:	129.48.32.3(replyRouter)
	server Lookup	→	server Router	:	replyRouter(routerFtpCli)
	server Router	→	client Router	:	routerFtpCli(129.48.32.3 , 168.32.16.5 , 3022 , 21 , 150)
	client Router	→	client Lookup	:	lookup(replyRouter , 129.48.32.3 , 168.32.16.5)
	client Lookup	→	client Lookup	:	129.48.32.3(same , swap)
	client Lookup	→	client Lookup	:	same(ftpCliHost , 255.0.0.1)
	client Lookup	→	client Router	:	replyRouter(ftpCliHost , 255.0.0.1 , 168.32.16.5)
	client Router	→	client Transport	:	ftpCliHost(255.0.0.1 , 168.32.16.5 , 3022 , 21 , 150)
D	client Transport	→	client Transport	:	3022(255.0.0.1 , 168.32.16.5 , 3022 , 21 , 150)

The client listens for an acknowledgement:

♠	client App	→	client Transport	:	receive(ftpCliSocket , replyCli)
♠	client Transport	→	client Transport	:	ftpCliSocket(n)
♠	client Transport	→	client Transport	:	n(168.32.16.5 , 255.0.0.1 , 21 , 3022)
D	client Transport	→	client Transport	:	3022(255.0.0.1 , 168.32.16.5 , 3022 , 21 , 150)
	client Transport	→	client App	:	replyCli(150)

♠ : may happen at any time between ♠♠ and when it is given. The two traces above synchronise on D. On receipt of this acknowledgement the FTP client opens the data socket and sends a packet to initiated it:

	client App	→	client Transport	:	connect(168.32.16.5 , 3021 , 3021 , socketReply)
	client Transport	→	client App	:	socketReply(dataSocket)
	client App	→	client Transport	:	send(datasocket)
	client Transport	→	client Transport	:	ftpCliSocket(n)
	client Transport	→	client Transport	:	n(168.32.16.5 , 255.0.0.1 , 3021 , 3020)
	client Transport	→	client Router	:	routerFtpCli(168.32.16.5 , 21 , 255.0.0.1 , 3021 , 3020)
	client Router	→	client Lookup	:	lookup(replyRouter , 168.32.16.5 , 255.0.0.1)
	client Lookup	→	client Lookup	:	168.32.16.5(same , swap)
	client Lookup	→	client Lookup	:	swap(routerFtpSer , 168.32.16.5)
	client Lookup	→	client Lookup	:	255.0.0.1(same)
	client Lookup	→	client Lookup	:	same(_ , 129.48.32.3)
	client Lookup	→	client Router	:	replyRouter(routerFtpSer , 168.32.16.5 , 129.48.32.3)
	client Router	→	server Router	:	routerFtpSer(168.32.16.5 , 129.48.32.3 , 3021 , 3020)
	server Router	→	server Lookup	:	lookup(replyRouter , 168.32.16.5)
	server Lookup	→	server Lookup	:	168.32.16.5(replyRouter)
	server Lookup	→	server Router	:	replyRouter(ftpSerHost)
	server Router	→	server Transport	:	ftpSerHost(168.32.16.5 , 129.48.32.3 , 3021 , 3020)
A	server Transport	→	server Transport	:	21(168.32.16.5 , 129.48.32.3 , 3021 , 3020)

The server listens for the data connection:

◊	server App	→	server Transport	:	listen(3021 , replySocket)
5	server Transport	→	server Transport	:	3021(168.32.16.5 , 129.48.32.3 , 3021 , 3020)
	server Transport	→	server App	:	replySocket(ftpSerDataSocket)

The server then discards the initiation packet and sends the file:

	server App	→	server Transport	:	receive(ftpSerSocket , replySer)
G	server Transport	→	server Transport	:	ftpSerSocket(n)
G	server Transport	→	server Transport	:	n(129.48.32.3 , 168.32.16.5 , 3022 , 21)
G	server Transport	→	server Transport	:	3021(168.32.16.5 , 129.48.32.3 , 3021 , 3020)
	server App	→	server Transport	:	send(ftpSerDataSocket , dataFile)
	server Transport	→	server Transport	:	ftpSerDataSocket(n)
	server Transport	→	server Transport	:	n(129.48.32.3 , 168.32.16.5 , 3020 , 3021)
	server Transport	→	server Router	:	routerFtpSer(129.48.32.3 , 168.32.16.5 ,
				:	3020 , 3021 , dataFile)
	server Router	→	server Lookup	:	lookup(replyRouter , 129.48.32.3)
	server Lookup	→	server Lookup	:	129.48.32.3(replyRouter)
	server Lookup	→	server Router	:	replyRouter(routerFtpCli)
	server Router	→	client Router	:	routerFtpCli(129.48.32.3 , 168.32.16.5 ,
				:	3020 , 3021 , dataFile)
	client Router	→	client Lookup	:	lookup(replyRouter , 129.48.32.3 , 168.32.16.5)
	client Lookup	→	client Lookup	:	129.48.32.3(same , swap)
	client Lookup	→	client Lookup	:	same(ftpCliHost , 255.0.0.1)
	client Lookup	→	client Router	:	replyRouter(ftpCliHost , 255.0.0.1 , 168.32.16.5)
	client Router	→	client Transport	:	ftpCliHost(255.0.0.1 , 168.32.16.5 ,
				:	3020 , 21 , dataFile)
E	client Transport	→	client Transport	:	3020(255.0.0.1 , 168.32.16.5 , 3020 , 3021 , dataFile)

The lines marks G discard the packet used to open the connection and so may happen later than given. The client listens for the data file:

	client App	→	client Transport	:	receive(ftpCliDataSocket , filerep)
	client Transport	→	client Transport	:	ftpCliDataSocket(n)
	client Transport	→	client Transport	:	n(168.32.16.5 , 129.48.32.3 , 3020 , 3021)
E	client Transport	→	client Transport	:	3020(129.48.32.3 , 168.32.16.5 , 3020 , 3021 , dataFile)
	client Transport	→	outside processes	:	fileReply(dataFile)

Output and links:

- 1 links ftpCliSocket to (168.32.16.5 , 255.0.0.1 , 21 , 3022)
- 2 links ftpSerSocket to (129.48.32.3 , 168.32.16.5 , 3022 , 21)
- 3 outputs 255.0.0.1 on storeIP , 3021 on storePort and sets the active flag.
- 4 links ftpCliDataSocket to (168.32.16.5 , 129.48.32.3 , 3021 , 3020)
- 5 links ftpSerDataSocket to (129.48.32.3 , 168.32.16.5 , 3020 , 3021)