A model for the coordination of mobile processes

Neil Berrington

October 1998

Abstract

Coordination is an important part of any computer program. As well as performing computation, applications need to interact with their environment (for example, reading a file from a disk, displaying information to a user). In distributed applications, processes that together make up the overall program also need to coordinate amongst themselves. Such coordination includes synchronisation and exchange of data.

Allowing processes to be mobile, where they are able to migrate around a network of nodes, allows novel solutions to be programmed where alternatives with static networks of processes would be inefficient or impossible to implement. For example, one method of finding "interesting" data on a network would be for a client machine to read the data off a variety of servers, and perform the filtering locally. A mobile solution could be to migrate processes to the servers, where they would query the data locally, migrating back to the client with the results. If the filter processes are small, and the databases large, this would result in reduced network usage and decreased search time.

Coordination becomes problematic in the presence of mobility. If two processes are allowed to migrate around a network, it becomes difficult to maintain connectivity between them.

This thesis explores how a set of interacting mobile processes may coordinate their activities. By abstracting away from physical networks to a model based on channels, distributed applications designers need not be aware of where processes are executing; two processes will always be able to communicate if they have access to a common channel.

The use of higher order channels presents a unified model of coordination — channels are used for migration (processes are transmitted to their destination node over a channel) and reconfiguration (channels may be transmitted over channels).

This model presents a powerful toolbox to the distributed applications designer. A distributed implementation, which binds the model with a symbolic computational language is provided, with sample applications used to demonstrate the systems capabilities.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Mobility in a distributed system

1.1.1 Distributed architectures

There are a number of ways of describing a distributed system. One method is to classify the architecture which a distributed application uses. These fall into three camps:

Shared memory — A number of processes access a common shared memory. The computer architecture ensures that all processors see the same view of this memory through the use of cache coherency. Special processor instructions allow primitives to be designed to control which processor has access to the memory, to prevent corruption of shared data structures.
These systems allow multithreaded programs originally designed for uniprocessor machines to run without modification on parallel machines. The scalability of this architecture is poor, as all processors need to frequently access the same memory, resulting in the shared memory becoming a bottleneck;

Closely coupled — These distributed systems use a multiprocessor machine with a very high bandwidth and highly interconnected communications network. Each processor in the machine has its own memory that cannot be directly accessed by the other processors. Applications which target these architectures typically do so in order to gain a speedup over their sequential equivalents;

Loosely coupled — Distributed systems utilise a network of workstations interconnected using a Local Area Network (LAN) and/or a Wide Area Network (WAN). Applications utilise these networks as a necessity in order to perform their applications. Examples include groupware applications, which need to communicate with the user of their workstation, as well as with other nodes in the groupware application, and client server architectures, where the server performs useful processing on behalf of a number of clients.

The loosely coupled network of workstations on a LAN has traditionally not been a good target for distributed applications which are designed for speedup over their sequential equivalents.

LANs are relatively slow at transferring data, compared with other media such as memory and disks. The bandwidth of a LAN is shared between all the nodes on a segment, thus in a 10Mbps LAN with 10 stations on, if all stations were to transmit at the same time, the effective bandwidth would reduce to 1Mbps. In the case of Ethernet this is further reduced by the CSMA/CD method of accessing the network. When a packet is transmitted, if another node on the segment is also transmitting, a collision is detected and the two stations back off before trying to retransmit the frame. The number of collisions will increase as the network load increases, further reducing available bandwidth to nodes on heavily loaded networks. It also means that the transmission time of a packet cannot be guaranteed.

Another problem is with guaranteed delivery of packets. LANs implement a "best effort" network; when a packet is sent from a node, routers and switches in its path do their best to forward the packet towards its destination. In a congested network packets will be lost, and there is no method for selecting what packets a router/switch should drop.

New networking standards are overcoming these problems.

Speed — Networks are becoming faster. The physical network layer is now capable of transmitting frames at an increased rate. 100Mpbs Ethernet is now widely available and 1Gbps Ethernet is being standardised.

In addition, switching technologies are increasing throughput of packets between different LAN segments. The relatively low price of switch ports means that the number of stations on a LAN segment can be reduced, thus increasing the amount of bandwidth available to individual nodes, and in the case of Ethernet, reducing the likelihood of collisions. Indeed in an ATM (Asynchronous Transfer Mode) network there can only be one node per switch port, so the node has all the available bandwidth between itself and the switch;

Delivery — New protocols allow streams between two endpoints in a network to be designated high priority. RSVP ((Braden *et al.* 1997)) is a protocol by which two endpoints signal to all the routers and switches in a route that the data stream needs to reserve a given bandwidth. If available bandwidth is available on all network segments between the endpoints, then the reserved bandwidth data path may be established. Routers and switches in the route of the data path will treat the stream as high priority and forward packets in a timely manner.

ATM has a number of quality of service levels which an endpoint may signal to the switch fabric when setting up a data channel.

1. UBR — Unrestricted bit rate. This is much like the best effort strategy of traditional networks. The switch will forward as many cells (a fixed size frame) on a UBR channel as is possible;
2. CBR — Constant bit rate. This allows an endpoint to signal that it intends to send a stream of data at a constant bit rate (for example voice).

The stream will only be set up if there is enough bandwidth between the endpoints;

3. VBR — Variable bit rate. The rate at which data is transmitted down the stream changes (for example compressed video) but will remain within the limits supplied when the channel was established;
4. ABR — Available bit rate. The switch periodically signals the endpoint to inform it how much bandwidth is available to it. This could be used, for example, to degrade a video stream if available bandwidth was reduced, improving it only when the ABR channel was signalled to say that available bandwidth had increased.

With the increase in speed of today's workstations and networking technologies, distributed applications designed specifically for speed up can be targeted at loosely coupled networks.

1.1.2 Distributed data flow

Another method of describing a distributed system is not on the underlying architecture, but rather by how processes act on data within the application. There are three main camps:

Data->process — Data is sent to the node which requires the data. For example, a network file system can be thought of as sending data back to the requesting process;

Process->data — For large data sets, it could be more efficient to move the process to the data rather than the data to the process. A process could also move to the data in order to reduce latency. For example a Java applet sent to a web browser allows a user to input data and get results faster than if the input data had to be sent to a web server, and then wait for the web server to process the data and send back the results;

Combination — In many cases a client machine requires a subset of data held by the server (for example a database application). In this case a process can be sent to the server which selects what data to retrieve (for example a SQL ((American National Standards Institute 1992)) statement), and the subset of data is sent back to the client for further processing. Another scenario is that both the data and process move to some third party node, perhaps because the third party node has some special processing capability (for example a vector processor).

1.2 The mobile process

The previous section has shown that moving a process to a node where data resides can be more efficient than moving the data towards the process. This is especially true when dealing with large sets of data. Processes could also benefit by moving to another node to access other fixed resources of that node (for example, a user or a vector processor).

This section addresses a number of issues which need to be taken into account when designing a system with support for mobile processes.

1.2.1 *Data representation*

A loosely coupled network is likely to contain machines of differing architectures. There could be differences between CPU type, operating system or both.

This can effect the movement of data between machines as different architectures can have different representations for data. Differences include:

Endianness — The order in which high and low order bytes of a machine word are stored. Some architectures store high end first, others low end first. Therefore if a machine word were to be transferred bitwise between two machines with different endianness, the receiving machine would see it as a different value to the sending machine ((Batey & Padget 1993));

Word size — Different CPUs have different machine word sizes. 32 bits is prevalent today but 64 bit processors are becoming more common. Thus in the language **C**, `sizeof(int)` on one machine may not equal `sizeof(int)` on another;

Alignment — In some architectures data can only be read from memory if it is aligned on a machine word boundary. This could mean that packed structures (where data items are not aligned on word boundaries) would be unable to be read. A packed structure which is valid on one machine would have to have each element machine word aligned in order to be read on a machine which enforces alignment.

Thus in order to transmit data between heterogeneous machines reliably, a common data representation needs to be adopted when transferring data between machines. Transmitting processes can convert their machine specific data representation to a common representation for sending over a network, and receiving processes can convert the incoming common data representation back to their internal representation. XDR ((Sun Microsystems 1987)) is one method which performs these transforms. An optimisation is to query the architecture of the receiving machine and only convert to common data representation for transmission if the architectures of the sender and receiver differ. If the architectures are the same, the data may be transmitted in internal form with no need for conversion at either end.

The process of converting an internal data representation to a common data format for transmission is called *marshalling*; the corresponding process of converting a common format back into a machine specific implementation is *unmarshalling*.

1.2.2 *Process representation*

Different CPUs in workstations in a network affect how a process may be represented. A binary executable which works on one machine in the network is by no means guaranteed to work on other machines in the same network. Different CPUs may have a different machine code, and thus will fail to execute a binary

which contains machine code for a different processor. Even where the CPUs are the same, if the operating system differs between two machines on a network this can also make their binary executables incompatible. This is because the binary contains a loader that will contain operating specific code, as well as code to access shared libraries in an operating specific manner.

There are a number of ways in which a process can be made portable so as to be able to run on a variety of nodes within a network:

Interpreter — The source code of a process can be transmitted to any machine in the network. Each machine in the distributed environment contains an interpreter which is capable of executing the source code representation of the process;

Byte code — Similar to interpretation of source code, the source code of the process is compiled into a common byte code form. Each machine in the distributed environment contains a byte code interpreter to execute processes;

Multiple binaries — If the source code of the process is compiled into binary form for each architecture in the distributed environment, then sending a process to a node involves querying the node to find its architecture, and selecting the correct binary for transmission. Another method is to encode multiple object formats for different platforms in one single binary, called a fat binary;

Portable machine code — Also called thin binaries, a common object format is translated into equivalent machine specific binary before being run.

Both the source code and byte code interpreter solutions allow easy creation of portable processes. The interpreter needs to be written and ported to each machine in the target distributed environment. A significant disadvantage of interpretation over binary executables is execution speed. Interpretation takes significantly longer than direct execution. Byte code interpretation is more efficient than source interpretation, and byte code processes are likely to be more compact than their source code equivalents. Performance of a byte code process can be further enhanced by the use of just in time compiling, where the byte code process is further compiled into CPU specific machine code.

The multiple binaries solution requires more effort on the part of the process designer, as rather than creating one binary representation of the process (byte code), multiple binaries need to be created. On a network with a large variety of architectures, a considerable number of binaries could be needed. If a new architecture is introduced to the network, all processes in the distributed application will have to be recompiled to target the new architecture in order for it to participate in the distributed environment. In the case of the interpreter solution, a new interpreter for the architecture will need to be written, but once done will run all existing processes in the distributed environment.

1.2.3 Mobile environment

For a process to be truly mobile, the movement of code and data need to be combined. Although a process might move to another node in order to access a data

set residing on the node, it will also need to take local data with it. The local data, for example, could be used to instruct the process what portion of the remote data set to interrogate.

One method would be to represent a mobile process as a block of code (which could be source, byte code or binary) with an entry point, and a parameter block which contains all the mobile data the process needs in order to operate on the target node. When starting the mobile process, the remote node would supply the parameter block to the process's entry point.

This method could be used for many of today's languages. For higher level languages, more elegant solutions can be proposed.

The closure allows both environment and code to be packaged together as one high level object. For example, in Lisp the following function produces a closure which takes a number and adds a given number to it:

```
(define (make-add n)
  (lambda (x)
    (+ x n)))

(define add-one (make-add 1))

(add-one 2) => 3
```

Note that the closure returned can be thought of as a function of one argument. However it also packages up the binding of `n` in its environment, thus allowing `n` to be referenced in the body of the closure.

In this way a mobile process can be represented as a closure, with the closure holding in its environment bindings which need to be transported in order to execute the code.

1.3 Process migration

Previous sections have described the conditions necessary for the migration of processes and data around a network of heterogeneous workstations.

This section introduces other factors which must be taken into consideration when designing a distributed environment for mobile processes.

It should be pointed out that in a distributed environment for mobility, it is intended that mobility support should be present in each node of the system. That is, all processes in a node should be able to migrate to other nodes, not just processes which originated on that node. For example, this definition removes an SQL server from being considered a distributed environment for mobility. Although there is process migration (an SQL program sent from the client to the server), once at the server it does not move again. Likewise, a Java applet sent from a web server to a client browser is not considered a candidate for a distributed environment for mobility.

Being able to move between nodes allows the implementation of flexible distributed applications. For example, a process could follow its user as they move between workstations, or travel between nodes in a network collecting data required by a user.

1.3.1 *Placement*

Where distribution is controlled by the application and not the underlying operating system, the distributed application designer must be able to direct processes to the nodes on which they are required to execute.

Possible methods include:

Automatic — The node chooses to migrate processes to other nodes. For example, this could be done to achieve load balancing between all nodes in the environment;

Explicit — The process asks to be migrated to a specific node (perhaps addressed by host name, for example);

By requirements — The process could query the distributed environment to return a handle to a node which meets its requirements (for example has a certain data set, or has a vector processor). Once a suitable node has been discovered, the process can migrate to it.

The automatic movement of processes to achieve load balancing is performed by a number of distributed systems, for example, Sting ((Jagannathan & Philbin 1992)). This thesis does not concentrate on this aspect of mobile computing. Rather it concerns itself with the explicit migration of processes between nodes, with a mechanism for discovering nodes which meet the requirements of a process.

1.3.2 *Environmental awareness*

Once a process has migrated to a foreign node, it needs to be able to request services available on that node. Thus each node needs to provide foreign processes with the ability to discover and bind to its resources.

A process may also need to be able to send results back to its home node. A common need for this feature might be for a data collecting process which migrates between sites looking at data sets and sending any "interesting" data back to its home site.

1.3.3 *Efficiency*

Given that a mobile process could migrate between many nodes in its lifetime, it is possible that a process will not execute all the code in its execution on any single node. Thus a possible efficiency saving could be made by only partially migrating a process when it moves between nodes.

Code that is likely to execute would be migrated, with the remaining code being held back at the originating node, and only sent to the remote node if the migrated process needs it.

The problem comes in guessing what code is most likely to be executed. If the algorithm is over cautious, then too much code will be migrated, under cautious

and too little code will be migrated initially, resulting in code fetches at a later time. Both would result in more bandwidth being used.

One possible solution is to look at the stack frames of the migrating process. Each stack frame represents a computation going back in time, with frames at the top of the stack being the most recent. As frames from the top of the stack will be popped first, it follows that these will point to code which will be executed before frames at the bottom of the stack. Thus partially migrating a process involves sending only part of the stack and the code it references.

In an environment which includes garbage collection, partial migration can lead to the need for more sophisticated distributed garbage collectors. This is because process state is distributed between multiple nodes, with data on one node possibly referencing data on another node.

The closure representation of processes discussed in section 1.2.3 makes the guesswork more difficult as the closure can be considered atomic, and the choice of where it could be split for partial migration is non obvious.

1.4 Lisp as a mobile language

Lisp was developed in the 1960's by a team led by John McCarthy. He published *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1* ((McCarthy 1960)) giving the first description of Lisp.

Since then Lisp has evolved steadily into a mature Language ((McCarthy 1981), (Steele Jr & Gabriel 1993)), with many dialects and implementations available for varying platforms (including specially designed Lisp machines).

Major dialects of Lisp include:

Common Lisp ((Steele Jr 1984), (Steele Jr 1982)) This has adopted by ANSI[1] as their standard Lisp ((ANSI X3.226-1994 1994)). It is a large language designed for commercial strength applications. Several companies market environments and compilers for the language, with public domain implementations also available;

Scheme ((Clinger. & Rees 1991)) A much smaller dialect than Common Lisp, Scheme was designed from the outset to be a clean, simple language. A denotational semantics is provided with its definition. Its lack of a standard object system makes object oriented programming convoluted ((Abelson *et al.* 1985) chapter 3, (Adams & Rees 1988)). Proprietary implementations of object systems for Scheme do exist ((Queinnec 1993));

IsLisp ((ISO/IEC 13816:1997 1997)) This has been adopted as the ISO[2] standard Lisp. It is a subset of Common Lisp made up from a blend of the German DIN Kernel Lisp ((Brand *et al.* 1992)) and the Japanese Lisp((Yuasa *et al.* 1992));

[1] American National Standards Institution
[2] International Standards Organisation

EuLisp**((Padget *et al.* 1993))** This is being developed as a European standard Lisp. It aims to be more complete than Scheme by including an object system, primitives for concurrency, and modules supporting programming in the large, and simpler than Common Lisp by using a layered structure.

Lisp provides a number of features which make it suitable for representing mobile processes, and also a suitable choice as an implementation language for modelling mobile processes.

1.4.1 Symbols

Lisp is a symbolic language, with the symbol as one of its major data types. Another major data type is the *cons* cell, which can be used to create lists of data.

Lists made up of symbols and other lists (for example the list `(if (= x y) 'yes 'no)`) are known as *s-expressions*, with Lisp programs made up of such s-expressions. The primitive `read` provides the Lisp programmer with a mechanism for reading *s-expressions* from input streams. Interpreters for Lisp can be written within Lisp, with `read` being used to parse the *s-expressions* and `case` statements matching the prefix commands to interpret them. Thus code may be treated as data. A Lisp interpreter that can interpret itself is called a metacircular evaluator.

1.4.2 Object systems

A number of Lisps support object oriented programs through built in object systems. The degree of integration with the rest of the Lisp environment varies. The EuLisp object system, Telos((Bretthauer *et al.* 1992), (Bretthauer *et al.* 1993), (Broadbery & Burdorf 1993)), has been designed at the same time as the rest of the languages, with the result that objects "go all the way down", in that all Lisp data are objects which are instances of some class.

Some Lisps extend the object model with a metaobject protocol ((Kiczales *et al.* 1991)), where the programmer can extend the object system with classes which manage objects, known as procedural reflection. This allows systems to export in a controlled fashion details of their implementation, allowing other programmers to tailor the system for their requirements without affecting the original code ((Kiczales 1992)).

1.4.3 Continuations

Many dialects of Lisp provide primitives or special forms which allow the current continuation to be reified into a data structure that can be applied. Continuations represent the remainder of a computation. In machine terms, a continuation represents the stack of a computation and all its registers at a given point. Once captured, continuations can be used in a number of situations ((Allison 1988), (Allison 1990)), for example to implement non local exits, model threads, and provide a mechanism for the clean handling of errors (where functions are given success and failure continuations and invoke the required continuation upon completion or failure).

Interpreters need not have access to continuations in order for an embedded language to use them ((Bonzon 1990)). Programs can be written in continuation passing style, in which each function takes an additional argument, its continuation (modelled as a closure) which it invokes, passing it the result of the function. Continuation passing evaluators take standard Lisp code and evaluate it using continuation passing style. By providing primitives to reify this continuation, the features described above can be implemented in an embedded interpreter even if the underlying Lisp does not have direct support for continuations.

1.5 Coordination

In previous sections the properties needed for a process to be able to migrate between nodes has been discussed. In summary these are:

- Support for running a process on differing architectures;
- Support for transmission of code as well as data over a network link;
- Support for transmission of a process's environment to a remote node;
- Ability of a process to interact with the environment of a remote node.

These features allow processes to run on any node in the distributed environment, and to move between nodes when needed.

We have previously centred on the example of one process migrating between nodes and interacting with each node's environment (for example, a data gathering process which migrates between nodes querying each one's local data set).

Now consider properties a distributed environment requires to support a set of mobile processes. If each mobile process acts in isolation (multiple data gathering processes, all carrying out different tasks, for example) then the answer is none. The properties described above provide a sufficient environment to run N instances of independent data gathering processes.

However if a set of mobile processes need to interact to perform a common task, then additional properties need to be provided by the distributed environment. Processes mainly interact in one of two ways, both of which we will refer to as coordination.

Synchronisation —A process halts its execution until a condition is met which allows it to proceed. Examples of synchronisation include mutual exclusion (for example to stop two processes simultaneously updating the same data set), and event handling, where a process waits for a signal (generated by another process) before continuing;

Communication — Processes share (partial) results amongst themselves. This enables a task to be broken down into many subtasks which are executed by different processes, which can lead to an increase in parallelism. For example, if items from two separate data sets are required in order to make a decision, two processes may be created to query each data set in parallel. Once both results have been received the decision making process may continue.

Coordination in a distributed environment poses problems; protocols need to be devised by which some operations (for example mutual exclusion) appear to be atomic to the distributed application but are actually achieved using a set of operations or messages between nodes.

Additional complexity is introduced when processes become mobile. Sets of interacting mobile processes need not all migrate to the same node, but still need to communicate. Thus the distributed environment needs to be able to track the movement of processes so it may route messages for a process to the node on which it is currently executing.

1.6 Research statement

This work describes the design and development of a distributed environment to support interacting mobile processes and investigates the applicability of the environment to a number of applications.

In addition this research presents a study of traditional distributed environments, coordination languages and existing environments which support mobile processes.

The prototype environment produced is not of production quality and does not integrate solutions in a number of key fields that a production level environment would require (for example, authentication). It is intended to illustrate the applicability of the communications model and programming language to mobile applications.

The originality of the work presented within this thesis lies in the combination of a higher order communications model with a dynamic programming language to produce an environment which meets the needs of applications requiring interacting mobile processes.

1.7 Overview

A study of existing distributed environments is presented in chapter 2. This highlights how dynamic languages have traditionally coped with distribution and how suited these environments are for the execution of mobile processes.

Chapter 3 provides a literature review on coordination languages. It describes each language and how it combines with computational languages to provide an environment for distributed applications.

The applicability of the π-calculus as a language for coordination is presented in chapter 4. It is shown that, together with a suitable computational language, an environment suitable for mobile processes can be envisaged.

Chapters 5 and 6 describe the design and implementation of the language, and demonstrate through the use of examples how distributed applications which use interacting mobile processes can be executed.

Solutions to further examples are presented in chapter 7. These show how mobility can be used in programming solutions to problems. In addition it is shown

that the language is a suitable base for implementing other distributed environments such as the document flow model ((Berrington, DeRoure, Greenwood & Henderson 1993)).

Finally chapter 8 goes on to discuss future enhancements which could be made to the environment to bring it closer to a production level system and sums up with conclusions.

1.8 Summary

This chapter has shown how the increases in the performance of today's workstations and local area networks allow distributed applications to be built which would previously have been targeted at specialised hardware.

In addition environments which traditional specialised distributed computing machines are unable to support may be envisaged. These applications can take advantage of the fact that they run on workstations on a local area network, for example to interact with users or resources on the network.

The network of workstations is able to support a mobile computing environment. Three main requirements have been shown to be needed in order to provide a practical environment in which mobile processes may execute:

1. Computational language. This should be able to be executed on multiple types of heterogeneous workstations. It should also be able to be represented in a form which can be used to transport mobile processes between workstations;
2. Ability to interact with the distributed environment. This includes discovering potential nodes which meet the migrating process's requirements, and the ability of a migrated process to interact with its host node's local environment;
3. Coordination between sets of processes which have potentially migrated to disparate nodes. This allows a programming task to be split into a number of separate processes, resulting in a increase in parallelism and reduced process complexity.

The next chapter goes on to discuss current distributed environments.

Chapter 2

Distributing computation

2.1 Introduction

This chapter looks in greater depth at current technologies available to the distributed application writer. As outlined in the introduction the technologies are organised into three main categories:

1. Shared memory. Multiple processors in a system share a single memory through a common bus;
2. Closely coupled. A set of processors with separate memory address spaces which communicate over a high performance network;
3. Loosely coupled. A set of workstations communicating over a local area network and/or a wide area network.

Flynn's taxonomy ((Flynn 1972)) classifies four categories of hardware used in parallel machines:

1. **SISD** — Single Instruction Single Data;
2. **SIMD** — Single Instruction Multiple Data;
3. **MISD** — Multiple Instruction Single Data;
4. **MIMD** — Multiple Instruction Multiple Data.

SISD is used on conventional single processor workstations, where each instruction acts upon a single piece of data. SIMD extends this to perform parallel data operations where instructions operate on multiple data (for example, a vector processor). The MISD classification is an unusual case where many instructions operate in parallel on a single datum. Finally MIMD architectures use multiple streams of instructions, each acting independently on data.

This chapter concentrates on architectures and languages corresponding to the MIMD classification, as distributed systems usually fall within this classification.

2.2 Shared memory architectures

Many of today's workstation operating systems have multiprocessor support built in, and multiprocessor systems are becoming popular in the server market.

This is because of the low cost involved in obtaining a performance increase out of existing applications by simply running them on multiprocessor hardware.

A multithreaded application can achieve speedup if the multiprocessor operating system chooses to run the application's threads on different processors simultaneously. Even for a single threaded application speedup can be achieved as other applications' threads running on the machine can be executed on other processors in the system, increasing the availability of processor resource to the single threaded application.

However, care must be taken when writing multithreaded applications if it is likely that the application will run on multiprocessor systems. If a thread locks a resource for a long time when it is not actually accessing it, it may not result in any performance decrease on a uniprocessor. This is because as long as the uniprocessor has a thread it can execute, it does not matter if another thread is blocked waiting for a resource. On multiprocessor systems, long term locking of unused resources can reduce the scope for parallelism within the application, thus reducing overall speedup. Excessive locking of resources for short periods of time, on the other hand, can lead to degradation on multiprocessor hardware as threads running on different processors constantly compete for shared resources.

2.2.1 *Operating system threads*

A common method of achieving distribution of tasks amongst the processors of a shared memory machine is through the use of operating system threads. Many operating systems provide support for multithreaded applications(for example Mach threads ((Cooper & Draves 1988)), Sun lightweight processes ((Sun 1988))), comprising of a number of services.

1. Scheduler. There are two main types of scheduler. Pre-emptive schedulers allow each thread to run for a specified period of time, known as a *timeslice*. The thread is able to run for this set of time, unless it blocks for some other reason, or a higher priority thread becomes ready to run (for example, after having serviced an interrupt). Once a thread's timeslice is exhausted, another thread is given the chance to execute. With cooperative schedulers a thread must explicitly allow another thread to be scheduled by either blocking or yielding control of the processor. An example of a cooperative scheduler is the **WIN16** scheduler for Windows 3.1. When an event is dispatched to an application, the application takes control of the processor for as long as is required to process that event. If the application fails (for example it goes into an infinite loop), this can prevent the system dispatching events to other applications, thus freezing the whole system. A pre-emptive scheduler on the other hand, would allow other applications to continue operating, even in the presence of a failed process[1];

[1] memory protection is also used to reduce scope for system wide failures

2. Threads interface. This API is provided to user applications to allow management of threads. Services provided include thread creation, destruction and retrieval of the thread's status;
3. Mutual exclusion. Another API provided by the operating system, this allows the application programmer to protect resources from pre-emption. This allows programs which are decomposed into separate threads to cooperate on shared data structures, for example a server thread to add work items to a queue which is accessed by worker threads retrieving work items. It is important to ensure the server and worker threads do not try to access the queue simultaneously as this could result in the queue's data structure becoming corrupted;
4. Signals. Threads often cannot perform useful work until another thread has performed some task. Signals, another API provided by operating systems, provide a mechanism for threads to wait until the application or operating system is ready for them to proceed. This is an efficient alternative to busy waiting, where a thread continually loops, checking to see if the system is ready for it to continue, using up processor resource even when there is nothing to be done. Using signals, a thread blocks until it is signalled by another thread to continue, which is more efficient than busy waiting, especially if the thread blocks for a relatively long period of time. Threads can also block waiting for external signals from other applications or the operating system, for example signalling that an IO operation has finished, or that a thread has terminated.

This set of services is capable of running an application with multiple threads either on a single processor machine, or a shared memory multiprocessor. This is because the thread is a unit of execution, and given a common memory and resource pool, can execute on any processor which has access to this pool. Hence the thread is a good basic abstraction to achieve parallelism on shared memory multiprocessor hardware.

Other abstractions which make use of shared memory multiprocessor hardware are often mapped down on to underlying operating system threads ((Mohr 1991)).

At any given time an operating system has a set of threads which are able to execute (because they are not blocked waiting for some condition, have not been terminated or suspended, and are not finished). The operating system can then choose which threads to execute on its available processors (one thread per processor). The choice of which thread to execute from the available pool is often based on the thread priority and previous execution time of threads. The threads will then run until their allocated timeslice has been exhausted (for pre-emptive schedulers) or until they block (pre-emptive and cooperative schedulers). At this time other threads will be scheduled to execute and the scheduler loop continues.

As each time a thread is scheduled it can be executed on a different processor, it could be argued that the threads migrate between processors (although processor

caches may make it more efficient for threads to stick to one processor). However, threads cannot explicitly migrate between processors and in a symmetric multiprocessor system there would be no advantage gained in being able to do so, as each processor has access to the same resources as all other processors in the system.

Many operating systems have direct support built into their kernels to support multiprocessing systems. Examples of multithreaded operating systems with support for running on multiprocessor hardware include Windows NT (with Win32 threads), Novell NetWare SMP, and Solaris SMP.

2.2.2 *Ada*

Tasks are the unit of concurrency available to Ada ((Barnes 1989)) applications. They can either be defined statically (for example if defined inside a procedure the task will be started each time the procedure is called) or dynamically by defining parameterised task types. Synchronisation and communication are provided by the rendezvous mechanism. The `select` statement allows non deterministic choice of a rendezvous.

2.2.3 *Modula 3*

Modula 3 ((Birrell 1991)) provides support for parallelism through the use of a threads package. An interface to allow the creation, initialisation and starting of threads is provided. Communication between threads is achieved through the use of shared variables, with semaphores being used to guard access. Threads may synchronise through the use of `wait` and `signal`. These primitives allow a thread to block, waiting for a condition which another thread may signal, allowing an event driven programming style.

2.2.4 *MultiLisp*

MultiLisp ((Halstead 1985)) is a dialect of Scheme ((Clinger. & Rees 1991)) which uses constructs to introduce explicit parallelism into applications running on shared memory multiprocessor machines.

Two constructs provide support for explicitly creating concurrency, *pcall* and *future*.

Pcalls (parallel calls) evaluate all their arguments in parallel and then perform an application of the evaluated arguments. Thus `(pcall + (f 1) (f 2))` is equivalent to `(apply + (list (f 1) (f 2)))` except that all the arguments to `pcall` are evaluated in parallel.

Futures allow programmers to indicate where useful parallelism can be gained in their programs and spawn threads to evaluate expressions whose results are not needed immediately. It is the programmer's responsibility to ensure that the parallelism created is safe, for example does not change the state of shared variables used by the main thread of control. If code executed in a future needs to access a resource which may be accessed by other futures or the main thread of control (for example, a shared data structure), then locking primitives can to be used in order to serialise access to the resource.

The form `(future` *expression*`)` returns a placeholder — a handle on the future. The implementation may choose to create a thread to evaluate the future expression concurrently with the main thread of control, or inline the expression.

In MultiLisp, once a future has been evaluated the placeholder is replaced with the value of the expression. If the value of the future is required before the future has been determined, then the requesting thread is blocked and waits for the future to finish evaluating. The placeholder can be passed as arguments to functions without the need to be evaluated.

For example, the following code calculates the number at sequence position n in the Fibonacci sequence:

```
(define (fib n)
    (if (< n 2)
      1
      (+ (future (fib (- n 1))) (future (fib (- n 2))))))
```

In this case for each iteration of the loop where $n > 1$, two futures are created which compute the Fibonacci values further up the sequence. However we can see here that in fact the **pcall** construct would have been more appropriate for no sooner than both futures have been created, their values are needed by the + operator. This causes the placeholders to be read, so the thread blocks until both futures have been evaluated.

Furthermore, the above example shows another problem with using futures, that it is relatively easy to create too much parallelism, where the system spends a large proportion of its time managing threads rather than executing them. Several solutions have been proposed. With dynamic load-based partitioning, the decision to inline or spawn a task to evaluate a future is based on system load at the time the future is reached.

Mohr ((Mohr 1991)) shows this policy can make inefficient use of resources and even lead to deadlock under certain conditions. These conditions can be difficult to detect statically by both compiler and programmer. He goes on to present an alternative method using lazy task creation. When a future is reached it is automatically *inlined*, so no partitioning occurs. However the parent of the future (the future's continuation) is stored in a lazy task queue. An idle processor can steal this continuation and create a task to evaluate the parent, thus creating parallelism. This is known as task stealing. Tasks will only be created when resources are available to execute them.

This could be implemented on top of multiprocessor operating systems by spawning N threads at system start up, where N is the number of processors. Each thread could then loop, waiting to steal a continuation, executing it and looping back ready to steal the next available continuation. Pscheme ((Yao 1994)) also makes use of the **pcall** mechanism but extends this with the notion of *ports*, allowing the parallel computations of a pcall to be captured in a similar manner to the

capture of continuations. Values can then be "thrown" down these ports, allowing the pcall operation to be completed multiple times, with potentially differing results.

2.2.5 *Sting*

Sting ((Jagannathan & Philbin 1992), (Philbin 1992)) is another dialect of Scheme, with support for the efficient execution of multiple threads. The system uses a set of virtual processors, each responsible for the scheduling of the threads executing on them. A virtual processor may "steal" threads from other virtual processors when it has no work to do, achieving load balancing between virtual processors. When threads are created, storage needs to be allocated for their heap and stack, which can be costly, especially if an application creates many threads throughout its lifetime. Sting allows thread storage space to be reused by delaying the creation of threads. A terminated thread's stack and storage space is reused by a newly started thread, with new storage only being allocated if no space can be reclaimed from terminated threads.

2.2.6 *QLisp*

Tasks created by Qlisp ((Gabriel & McCarthy 1984), (Goldman & Gabriel 1988*b*), (Goldman & Gabriel 1988*a*)) are placed in a queue until resources become available to execute them. Each construct capable of creating tasks takes a propositional argument. If the value of the proposition is true then a task is spawned to evaluate the construct, otherwise it is inlined. This allows the user to control the amount of parallelism created (for example by only spawning new tasks for the first few levels of a tree search). Primitives to examine the task queue enable the user to implement a form of load balancing.

It is likely that implementations of QLisp running on various platforms will require different propositional expressions, depending upon the number of processors and execution speed. This would mean programs would have to be fine-tuned for each platform on which they are designed to run. The use of the propositional argument could also detract from the intuitiveness of a program as load balancing code and process code are mixed.

The construct `(spawn prop form)` creates a task to evaluate `form` if `prop` evaluates to true, otherwise the form is evaluated inline. Spawn returns a future placeholder if a task is spawned or the value of the form if it is inlined.

The most common way of introducing parallelism in Qlisp is with the **qlet** construct. This is similar to a Lisp `let` construct, in that it introduces new bindings into the lexical scope of an expression, except tasks are spawned to evaluate each new binding (if the propositional argument to `qlet` evaluate to true). By default the body of the `qlet` expression is not evaluated until all tasks spawned by the `qlet` have finished and all the variables of the `qlet` have been bound. Eager evaluation of a `qlet` may be selected through the use of a keyword. In this case the variables of the `qlet` are bound to future placeholders, which will be replaced with the value of their expression when the task evaluating it has completed.

The **qlambda** construct allows the user to implement mutual exclusion zones in the code. A closure is returned, which when invoked will spawn a task to evaluate the closure (if the proposition evaluates to true). Any other process invoking this closure will be blocked and wait until the closure becomes free for execution.

2.2.7 *Spur Lisp*

Spur Lisp ((Zorn *et al.* 1989)) provides support for multiple processes with a very simple threads interface. The form `(make-process` E`)` creates a new process which immediately starts executing the expression E. The language also has support for futures as in MultiLisp. Communication between processes is via *mailboxes*. Primitives are provided to send and receive data to/from a mailbox. Mailboxes may contain arbitrary amounts of data so the send operation is guaranteed not to block. The receive operation will only block if the mailbox contains no data. A process may choose to receive mail from a number of mailboxes, in which case mail will be returned from the first non empty mailbox. This is similar to CSP's ((Hoare 1985)) **alt** construct, except mailboxes are scanned in order and posting to a mailbox is not synchronised with a process retrieving data from a mailbox.

No direct support is given for semaphores or locks that allow synchronisation between processes, although such data structures can be modelled using mailboxes.

2.2.8 *EuLisp*

The EuLisp definition allows the concurrent execution of expressions through the use of threads, and atomic communication and mutual exclusion via the use of semaphores ((Berrington 1990), (Berrington, Broadbery, De Roure & Padget 1993), (Berrington, Deroure & Padget 1993)). A thread is an abstract data type representing the flow of control in a program, and interacts with other threads via shared memory.

When a thread is created, an initial function is supplied; arguments are provided when the thread is started, and the thread executes the application of the initial function to these arguments. When the function returns, the value of the thread is set to the return value of the function, and the thread completes normally.

The following interface is provided for the creation and control of threads:

`<thread>` Class object representing the class *thread*. This object is pre-installed in the Telos[2] ((Bretthauer *et al.* 1992), (Bretthauer *et al.* 1993),(Broadbery & Burdorf 1993)) hierarchy;

`(make <thread> 'init-function` *function*`)` Allocates, initialises and returns a thread with *function* as its initial function. The thread object is a first class object as is any other object in the Telos hierarchy;

`(thread-start` *thread* . *args*`)` Starts a thread with the specified arguments;

`(thread-reschedule)` Indicates the current thread can concede control to another thread;

[2]The EuLisp Object System

`(thread-value` *thread*`)` Blocks the current thread until *thread* has completed, and returns its value.

The definition does not force a conforming EuLisp implementation to adopt any particular scheduling policy, allowing implementers to choose a suitable one for the hardware and operating system platform they are targeting.

A portable EuLisp program must therefore not assume a particular scheduling policy itself; for example, assuming a time-slicing scheduler and therefore never explicitly calling `thread-reschedule` may cause the program to fail on a scheduler which lacks pre-emptive tasking, where threads must concede control voluntarily.

A fairness guarantee also needs to be stated. Due to the possibility of a host operating system handling the execution of threads, it is impossible to make a strong guarantee.

However a sufficient, although weak, guarantee is that if a thread reschedules infinitely often, then every other ready thread will also be scheduled infinitely often. Semaphores are provided for synchronisation between threads and mutual exclusion. The following primitives are provided:

`<lock>` A class object representing the class *lock*. This object is pre-installed in the Telos hierarchy;

`(make <lock>)` Method for the generic function `make` which returns a newly allocated open lock;

`(lock` *lock*`)` Perform a **P** ((Dijkstra 1965)) operation on *lock*. This blocks the current thread until such time as no other thread has ownership of the lock. At this time the thread takes ownership of the lock;

`(unlock` *lock*`)` Perform a **V** operation on *lock*. This removes ownership of the lock and allows the ownership to be transferred to a waiting thread.

An implementation is free to choose its strategy for the locking operation. For example, by blocking a thread on a lock until it becomes free or by busy waiting by rescheduling a thread until the lock becomes free.

2.2.9 *Summary*

One significant advantage of shared memory multiprocessor machines over other distributed environments is that a large existing code base can be run on them. This is due to a number of common operating systems having built in support for these machines, thus allowing applications targeted at these operating systems to run unmodified.

As all processors share the same physical memory, performance can be reduced as processors compete for the memory resource. Local caching of memory on each processor in the machine can improve performance at the cost of having to implement a cache coherency protocol. However even with local caching the number of processors which can be supported by such a system remains small.

These systems are not good candidates for mobile processes, as processes running on such systems are only distributed amongst processors within the workstation. However, they could be used as high performance nodes within a distributed environment. In commercial environments, shared memory multiprocessors are often used as high performance application and file servers, supporting a number of client machines over a local area network.

2.3 Closely coupled architectures

Shared memory multiprocessors do not scale well due to the bottleneck introduced by processors having to access the same memory pool. By giving each processor its own separate memory space this bottleneck can be removed. This effectively results in N separate computers, where N is the number of processors in the multiprocessor machine.

As application threads running on different processors do not have access to the same shared address space, new methods of communication between threads need to be found. Possible solutions include:

- Separate address space/shared memory hybrid. Threads have access to a local memory space for storing information not needed by other threads (e.g. code, local variables — although the use of pointers could make this problematic), and use the shared address space for storing data structures that are shared with other threads in the application. This decreases the complexity of writing applications, but reintroduces the bottleneck of shared memory. Indeed such architectures may have little performance increase over traditional shared memory architectures, depending on the efficiency of local caching;
- Message passing. If each processor in the machine is able to transmit and receive messages from every other processor in the machine then communication can be achieved by copying data between address spaces. As no shared memory is involved, bottlenecks can be reduced (assuming that the communications mechanism between processors is efficient). Existing threads based applications will need to be redesigned to use message passing styles, resulting in a non trivial porting effort being required;
- Simulated shared memory. This makes use of the underlying message passing architecture of the multiprocessor machine, but presents the programmer with a familiar shared memory programming interface. Optimisations performed by the compiler and runtime system are used to increase efficiency of the application towards that of an equivalent application running on truly shared memory architectures.

There are a number of technologies which have been designed for closely coupled architectures using a high bandwidth network, with communication in mind.

2.3.1 Occam

The CSP model of concurrency (see section 4.2.1) has been adopted by the language Occam ((Occ 1988), (Pountain & May 1988)), the main language for the Transputer.

This maps CSP constructs, together with common programming language constructs onto an Algol like syntax. Channels are used to synchronise and communicate data between processes.

This presents a message passing paradigm to the programmer. Messages are transmitted over channels, with the sending and receiving processes synchronising during this operation (a transmitting process cannot proceed until a receiving process is ready to receive the data).

2.3.2 *Message Passing Interface*

In a distributed environment where processors do not share memory, an effective and efficient method needs to be used to communicate between parts of the system. A widespread technique used is that of message passing. The necessary information is packaged up by a process into a message and passed to some form of communication layer. This layer then transfers the packet to its destination, where it is passed to the receiving process. As well as point to point, other communication models such as multicast are supported.

The Message Passing Interface (MPI) ((MPI Forum 1994)) library is the product of a committee of the research and industrial communities. The mission of the committee was to develop a standard for message passing, thus hopefully reducing the number of vendor specific and proprietary message passing libraries with incompatible interfaces. It focused on providing an interface for high performance applications.

MPI-1, the first version of the specification, focuses mainly on point to point communication routines. In addition, the library provides support for collective communication amongst groups of processes. The topologies of communication networks between processes can be specified, enabling the implementation to optimise the underlying communication network and mapping of processes to processors.

It is not designed to be used in a heterogeneous environment, and different vendors' implementations of the standard are not required to inter-operate. Hence MPI implementations may make use of architecture specific encoding and be fully optimised for their target environment.

2.3.3 *Distributed shared memory*

Alewife ((Kranz *et al.* 1992)) is a combination of hardware and software which provides an environment for parallel processes through a combination of distributed shared memory and message passing.

This allows the applications programmer to choose the most appropriate technology for an individual task, with applications being able to use both shared memory and message passing concurrently.

2.3.4 *Summary*

The closely coupled architecture is suited to writing high performance parallel applications. Its separate address spaces and high bandwidth internal communications network between processors removes the bottlenecks associated with shared memory multiprocessors.

However the separate address space for each node in the multiprocessor means that porting existing non message passing applications is far more involved than porting them to run on shared memory machines (often no porting effort is required for these machines).

Mobile processes could be deployed on such architectures to help with load balancing amongst nodes within the machine, or if different resources need to be accessed on different nodes.

2.4 Loosely coupled architectures

A set of workstations connected by a local or wide area network has features in common with the closely coupled architecture discussed in the previous section. Each node in the network has its own address space, in addition to other resources only accessible through the node (for example, a database). Communications between nodes can only be achieved using message passing over the network as there is no shared memory between nodes (although some nodes in the network may be shared memory multiprocessors).

However the network of workstations also has some differences over the closely coupled architecture.

- Multiuser. Workstations tend to be located on users' desks, with users logging on to their own workstations. This allows a "user" resource to be associated with a workstation, whereas with the closely coupled architecture it is more difficult to associate a user with an individual node;
- Heterogeneous. Although it would be possible to build a heterogeneous closely coupled machine, with different nodes having different CPUs, it is most common for all the nodes in the computer to be of the same type. This is less so with a network of workstations. As discussed in the introduction, there are a number of ways in which workstations in a network can differ, including CPU type and operating system;
- Reliability. Local and wide area networks around a building/campus/world tend to be less reliable than a network between nodes in a single machine. This is because there are more parts in the network, including wires, hubs, routers and switches, and therefore more things to go wrong.

There are a multitude of technologies and languages which have been designed to support loosely coupled networks. A review of these has been split into three main sections:

1. Languages and technologies which have no direct support for mobility. These systems have the potential to support mobile systems, but such features have to be added by the applications programmer;
2. Languages and technologies with support for limited mobility. These systems allow a process to migrate to a node where it is executed. Once it has migrated, it cannot migrate again;
3. Support for full mobility. Processes may migrate between nodes many times in their lifetimes.

2.4.1 *No mobility*

This section reviews toolkits and distributed systems which have no direct support for mobile processes.

Sockets

A low level interface, sockets ((W.R.Stevens 1990)) provides an abstraction to the underlying network protocols. The main protocols abstracted by sockets are UDP, an unreliable datagram service based on top of IP, and TCP, a reliable byte stream service also based on top of IP.

The sockets abstraction allows connections to be set up to remote computers and data to be sent and received across these connections. Sockets impose no rules on the format of data which can be sent over a connection, therefore it is up to the applications programmer to ensure that data sent over a connection can be read and understood by the receiving end.

The application can achieve this by tackling a number of areas. In order for two applications communicating over a network to communicate, they must agree on the protocol to be used. This includes the types of messages, when they can be sent, and what they should contain.

In addition a common data format needs to be used when sending messages. When communicating between heterogeneous workstations, it is important that the receiving workstation receives messages with the same contents as the sending workstation. This can be upset if both processors have a different endianness, for example, this would cause an integer sent from one node to be interpreted as having a different value by the receiving node. Alignment and word size can also affect message interpretation if they are different on the sending and receiving workstations.

Resolution of this problem can be achieved, for example, by converting integers which are sent over the link to network byte order by the sending node, and back to host byte order by the receiving application. This will allow integers to be correctly transmitted between machines, even if each of the machines has different byte order for data representation. XDR ((Sun Microsystems 1987)) provides a standard for the transmission of data in a common format over a network of heterogeneous workstations.

An optimisation of this solution is to negotiate on the link (using network byte order messages) whether any conversion is needed when the connection is first

established, and only subsequently to convert messages to network byte order if sending a message to a workstation with different characteristics.

This interface is very low level and provides no built in support for the discovery of nodes on the network, fault tolerance and many other features desirable in distributed applications.

Parallel Virtual Machine

Parallel Virtual Machine (PVM) ((Geist *et al.* 1994), (Sunderman 1990)) provides a package for programming a network of heterogeneous machines, consisting of a library for C and Fortran programmers, and a set of system programs which support the virtual machine.

In PVM a virtual machine consists of a number of machines linked together by support software. The user is able to specify a list of physical machines which should make up the virtual machine. This configuration need not be static, as machines can be added and removed whilst the virtual machine is operational. Individual users may configure their own virtual machines, with a single physical machine capable of being part of a number of virtual machines. A speed rating may be associated with machines, allowing PVM to bias the placing of processes towards faster machines (although it does not take account of machine load and network performance).

The PVM library consists of a set of message passing interfaces and process control routines. Processes are started by specifying the program name, and, optionally the machine or architecture on which it should be started. PVM will then select an appropriate machine and instruct the PVM daemon running on that machine to start the process.

Messages in PVM consist of a message tag and message body. A set of routines is supplied to marshall and unmarshall data to/from a message body. A message can be sent to a specific process using its **tid**(process id assigned by PVM). Messages sent between two machines are guaranteed to arrive and are delivered in order. They are buffered at the receiving end, where the process may either choose to consume them serially, or specify a message tag to receive a message by type.

Broadcasts may be accomplished by sending a message to a set of **tids** or by specifying a named group. Processes can join named groups, allowing operations to be performed on the group as a whole, including broadcasts and barrier synchronisations.

RPC

Remote Procedure Calls ((Stamos & Gifford 1990)) provide an abstraction on top of sockets which allow distributed applications to be built in a procedural style. The application can be split in two. The server receives and executes requests, and the clients send their requests to servers. Clients and servers are separate applications, and can be executed on separate workstations, communicating using message passing over a network.

An Interface Definition Language(IDL) is used to specify the requests which the client may make to servers. These requests take the form of procedure calls, with the interface file compiling into client and server libraries. The client library packages up the procedure arguments into a message and sends it to the server to be processed. It then waits for the result from the server before returning to the caller. This gives the illusion to users of the client library that they are just making standard procedure calls, simplifying the distributed programming interface. The server library receives requests from clients, unpacks the parameters from the received message and makes the procedure calls, returning any results (the procedures return code, *OUT* parameters) to the client in another message. The Angel operating system ((Wilkinson *et al.* 1992)) provides support for an optimised RPC mechanism called Lightweight RPC.

Nexus

Nexus ((Foster *et al.* 1994)) is primarily designed as a runtime system for task-parallel languages. It couples global pointers with RPC like requests in a multi-threaded environment. A Remote Service Request (RSR) specifies a global pointer, a procedure to be invoked and the arguments for the procedure. These arguments are transferred to the location pointed at by the global pointer, and the requests procedure is invoked.

CORBA

The Common Object Broker Architecture ((Siegel 1996)) is similar to RPC in that it is an abstraction above message passing libraries such as sockets. Like RPC an Interface Definition Language is used to specify the interface between client and server. Unlike RPC, CORBA is an object oriented environment. Interfaces represent a contract between a client and a remote object. Like many objected oriented systems, CORBA has support for inheritance and polymorphism.

All requests made between clients and servers travel via an ORB (object request broker). This process uses a proprietary protocol for its connections between client and server. Clients do not have to know the exact location of objects they wish to communicate with. Rather they have handles to these objects, and the ORB resolves this handle to an object and its location in order to forward the message to the correct server for handling. Figure 2.1 shows a client sending a message to an object managed by the local ORB. A standard inter-ORB protocol allows messages to be routed to objects via remote ORBs when the object is not being managed by the clients local ORB. This feature removes the need for clients to have knowledge of their network environment. Sending a message to an object uses the same interface (through the IDL), be the object local or remote.

Objects may be discovered in a number of ways (for example, by creating a new object, using the dynamic invocation interface, or via the trader service). **CORBAServices** provide objects with a standard set of tools which they may use in order to perform their tasks, reducing the burden on the applications programmer.

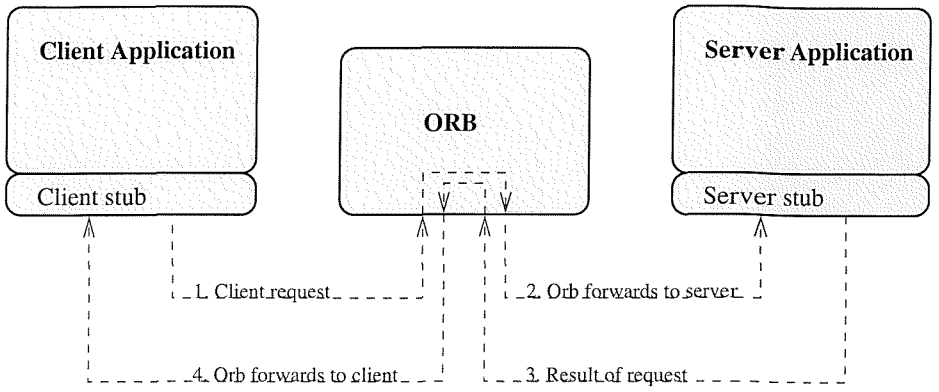

Figure 2.1: A request brokered by an ORB

This is extended with **CORBAFacilities**, a set of interfaces which provide high level tools for building common elements of applications (for example, user interfaces). The facilities are further split into horizontal facilities (which are common to many types of application) and vertical facilities (which are common to a certain type of application, for example, financial applications).

Distributed Computing Environment

The Distributed Computing Environment (DCE) ((Rosenberry *et al.* 1992)) is an initiative funded by the Open Software Foundation(OSF).

It splits distributed environments into administrative domains calls *cells*. An authentication mechanism is provided to support distributed applications running over many cells. In addition a number of technologies which should be present in a conforming DCE are specified.

- RPC. Used as the basic process for request services from a remote machine;
- Threads. Allowing concurrency to be introduced into single applications;
- Directory service. This allows applications to discover where resources they need reside in the network. Applications can also register resources they provide with the directory service;
- Time service. This allows a distributed application to have a consistent view of the current time by ensuring that all clocks in the DCE are kept synchronised with each other;
- Distributed file service. This allows applications to have access to files, regardless of which node in a cell they are executing on.

Distributed shared memory

Midway ((Bershad *et al.* 1993)) is a distributed shared memory environment. It uses a threads interface to provide the application with a means of introducing concurrency into an application. A software implementation of a distributed cache is used to maintain the consistency of shared data. Different consistency models are available to the applications writer, with each model having a trade-off between efficiency and ease of use.

A *weakly* ordered system maintains causality within the system whilst reducing the number of messages which need to flow. The DASH operating system ((John & Ahamad 1993)) uses locking primitives that provide hints about when a process expects to see a consistent view of an area of memory, allowing the operating system to ensure that the local copy of the shared memory is up to date before the critical section is entered.

MUSE ((Yokote *et al.* 1989)) is another example of an operating system which provides direct support for distributed programming (through a set of metaobjects).

2.4.2 *Limited mobility*

SQL

SQL ((American National Standards Institute 1992)) is a language used to query databases. For a large database residing on a server in a network, it is more efficient to send a query for the database server to execute against the database, rather than have the client read sections of the database from the server across the network so that it may execute the query locally. SQL is used to express queries that may be sent across the network for the database server to execute.

Three tier database systems are becoming more popular in networked environments. In the traditional two tier approach, clients send queries to the database server in order to access and update the database. Therefore the client needs to possess knowledge of the structure of the database, and the rules that are used to access and update data, so called business logic.

With the three tier approach, clients send higher level requests to servers, for example by using the CORBA or RPC interfaces. The servers then make the relevant requests to the database servers, implementing the business logic. The clients, who no longer have to implement business logic, are known as thin clients. Three tier implementations can improve security, as clients do not have direct access to the database, and can only contact it via an exposed API.

They can also improve the utilisation of database connections, as servers fielding requests from multiple clients can query the database on behalf of these clients over the same database connection. At first glance it would appear that three tier applications would place a heavier load on the network, as now not only is the database being queried, but clients and servers are communicating using another protocol. As well as producing additional network traffic, it could also increase latency. These problems can be reduced however. For example, a separate, private network could connect the servers and database. This would remove load from the rest of the network and increase security, as clients would not be able to connect directly to the database server.

The latency question depends upon how long servers take to execute the business logic of each request. For a lightly loaded high performance server, latency could actually be reduced, as the server could take less time to execute the request than a standard user workstation (which is likely to be less powerful than the

server). Management tasks can also be reduced, as changing business logic would involve updating the servers, rather than all clients which access the database.

Postscript

In the same way that SQL has been developed to reduce the amount of database traffic on a network, Postscript ((Taft & Walden 1995)) is a page description language which reduces printer traffic on a network. Rather than sending a large bitmap to a printer, a Postscript printer receives programs which generate the bitmaps locally, and these are then printed. Since the program is smaller than the generated bitmap, network traffic is reduced. It also reduces the load on the client that is printing, as it does not have to generate the bitmaps itself.

Protocols

Protocols which are used by applications to communicate with servers can be thought of as simple programming languages. For example, SMTP[3] ((Postel 1982)) is used by a client to deliver email. Commands sent to the SMTP server include `MAIL TO: <name>`, which instructs the server where the mail is to be delivered, and `RCTP FROM: <name>`, which instructs the server who the email is from. Such commands are combined in a single session to achieve the client's aim — to deliver email.

More complex protocols such as the X Windows protocol look more like programming languages. X events are interpreted by clients, and X servers respond to requests issued by clients (similar to RPC).

Java

The Java virtual machine ((Gosling 1995)) allows Java programs, compiled down into a byte code, to be executed on a variety of platforms. As the byte code is portable, any platform that has the virtual machine ported to it can execute any Java program[4].

Just in time compilers can increase the performance of Java applications (applets) by further compiling the Java byte code into native machine code. This retains the portability of Java applets, as byte code is used to distribute them, whilst improving performance by reducing the overhead of byte code interpretation. Another technique is to use threaded code ((Bell 1973)) which preserves the space advantage of byte code whilst improving performance.

Currently the main use of Java is to distribute applets from web servers to execute on local clients, with web browsers used to display the applet's output. As applets can be downloaded from servers where little is known of their contents, they can be sandboxed so as to reduce the chances of them accessing or disrupting data on client workstations.

[3] Simple Mail Transfer Protocol

[4] Although incompatibilities between VMs do exist

However, technologies such as RMI[5] ((RMI 1996)) and Jini ((Waldo 1998)) allow more complex Java applications to be written which can interact with a network of other Java nodes.

ActiveX

Another technology for distributing applications on the Web is Microsoft's ActiveX. It allows any program, often compiled into native code, to be downloaded from a web server and executed on a client, again with results sent to the web browser. As native code is harder to sandbox, authentication techniques are used to ensure that unsafe code is not executed.

Omniware

Omniware ((Adl-Tabatabai *et al.* 1996)) uses a byte code for distribution of programs over a network. These byte code programs are then further compiled into native machine code before being executed. A method called *software fault isolation* prevents programs from accessing memory outside their application boundaries. This simplifies the validation of processes by the virtual machine, as processor rather than source semantics can be used for program verification.

2.4.3 *Full mobility*

Rsh

The Unix **rsh**[6] command allows shell scripts to be written which can transplant themselves to other computers on the network. For example, the following script finds the path of a user's home directory on two machines, and displays the results:

```
#!/bin/sh
HOME1='rsh $1 echo ~'
HOME2='rsh $2 echo ~'

echo home directory on $1 is $HOME1
echo home directory on $2 is $HOME2
```

A script may be sent via **rsh** to execute on any host where the user has permission to perform **rsh** operations. Once a script has been transferred to a host, it has the shell's full range of features available to it. These range from starting programs to manipulating files. Anything the shell can do, given the user's permissions, a remote shell script can also do.

This openness is very powerful when writing scripts that perform well tested tasks in a known and limited environment. But it also means that it is very insecure.

For example the following script would attempt to perform an evil task on a remote machine:

```
!#/bin/sh
# Do not run me!
```

[5] Remote Method Invocation
[6] Remote shell

```
rsh $1 'rm -rf /'
```

For this reason care needs to be taken when writing remote shell scripts, and most networks will not allow them to be started from hosts outside their domain.

This does not make `rsh` a good candidate for writing mobile code for large scale applications.

Network command language

NCL ((Falcone 1987)) uses a universal Lisp-like language to transparently send commands between clients and servers over a heterogeneous network of workstations. Clients send procedures for the server to execute. These procedures may send other procedures for remote evaluation elsewhere, allowing a procedure to migrate more than once between machines in the network.

Kali Scheme

Kali Scheme ((Ceftin *et al.* 1995)) is a distributed Scheme programming environment built on top of Scheme-48 ((Kelsey & Rees 1995)), a byte code interpreter designed at MIT.

Unlike many other distributed Lisps, it does not implement a shared memory, but instead uses separate address spaces with a set of message passing primitives provided to the user to allow communication between nodes.

Address spaces are treated as first class objects. The `(make-address-space <node>)` primitive spawns a new Kali Scheme image on the specified node (where node is a network address or a hostname which can be resolved to a network address). The object returned by the primitive is a pointer into the address space on the remote node, and can be used to send messages to the node.

As well as supporting the transmission of basic Scheme data types (lists, symbols, constants, etc), the Kali Scheme message passing primitives can be used to send and receive higher order data types such as functions, closures and continuations. Continuations are sent in a lazy fashion; only the top few frames are sent at first, with subsequent frames being requested by the receiving node if they are needed for evaluation. This allows cheap migration of processes, and can cut down communication costs as often the process will not unwind to the top of its stack.

Continuations are used to represent threads in a Kali Scheme program. User level schedulers may be installed to handle context switching between threads. This could be used to define an application specific load balancing system by spawning a threads continuation on a remote address space.

As each Kali Scheme node has a separate address space, with data being copied between address spaces (messages), garbage collection may be performed independently. The one exception to this is a new data type introduced by Kali Scheme called the **proxy**.

A proxy may be thought of as a remote pointer. The `(make-proxy)` primitive returns a pointer to an address on the current node. This address may then be referenced using `(proxy-value <proxy>)` and set using `(set-proxy value! <proxy> <value>)`. However unlike a remote pointer, these operations are only

valid on the node which created the proxy. Remote nodes can find out the address space in which the proxy resides using `(proxy-address <proxy>)` which returns an address space object.

As the proxy is just another Scheme object, it can be passed between nodes using the message passing primitives. However as it points back into another node's address space, it is unsafe to reclaim the proxy on one node if other nodes reference the proxy. For this data type alone a distributed garbage collector is used.

To ease the programming burden, primitives have been built on top of the message passing layer. As the data does not reside in shared memory, these primitives allow computations to be migrated towards the data.

`(remote-apply <address> <expr>)` applies the expression in the specified address space, and synchronously waits for a response. This may be thought of as a remote procedure call in traditional languages;

`(remote-exec <address> <thunk>)` takes a thunk (a function of no arguments) and spawns a new process in the specified address space to execute the thunk. The requesting address space does not wait for the thunk to finish executing. This allows new processes to be introduced into a remote node.

For example, to move the current process to a remote node x the following expression would suffice:

```
(call/cc (lambda (k)
               (remote-exec x (lambda () (k #t))) #f))
```

By testing the return code from the `call/cc` the application can determine if it is the migrated process (return code is true), or is the original process (in which case the return code is false).

2.4.4 *Summary*

There is a multitude of technologies available for the distribution of applications through a loosely coupled network. These range from methods for distributing a stand alone application to clients' machines (web clients pulling applications from servers), through low level protocols for message passing such as sockets, to higher level distributed application oriented technologies like CORBA.

These technologies provide varying support to mobile processes. Languages which make use of a common code representation are better suited to process mobility as they can operate on a number of nodes in a heterogeneous network. Kali Scheme has support for the distribution of processes through the ability to capture and transmit continuations and closures. Java provides support for the transmission of code for a class through the RMI interface.

2.5 Chapter summary

This chapter has presented a survey of technologies and languages which may be used to implement parallel and distributed systems.

Shared memory systems allow the provision of parallelism to existing applications through the use of operating system threads. By executing threads on separate processors parallelism can be achieved without the need for rewriting applications that other techniques such as message passing systems require.

It has been shown that distributed environments can be categorised by their support for the mobility of processes they provide. Message passing layers such as sockets and PVM provide no direct support for mobility of processes, although Kali Scheme shows that such systems together with a portable process representation can be used to achieve mobility.

All these systems provide some support for the coordination of activities between threads and tasks (such as the semaphore or event). However, these capabilities are either deeply integrated into the computational language or are a "bolt on" library.

The next chapter introduces languages which have been specifically designed with coordination in mind, and can be separated from their computational language(s).

Chapter 3

Coordination of distributed applications

3.1 Introduction

The previous chapter discussed different architectures which can be built to support parallel and distributed applications, and computer languages that provide abstractions of these architectures.

All these languages have one thing in common — at some level they provide support for multiple processes or threads running within a single application. Even in languages which attempt to hide parallelism, the runtime system will make use of threads or multiple processes.

Two mechanisms are needed to support this concurrency:

Communication — This is needed to pass data between threads and processes so that they may cooperate in solving a single task. This communication can be explicit (message passing) or implicit (shared variable in a (distributed) shared memory);

Synchronisation — Used to ensure that processes and threads only communicate and access common areas of memory when it is safe to do so, preventing simultaneous update of a resource.

Together these two mechanisms are called coordination. This chapter investigates coordination and how these mechanisms can be separated from other aspects of a computer language.

3.2 Coordination languages

A coordination language does not specify how different elements involved in a computation should proceed, but rather how the elements interact to achieve the desired result.

A computation can be thought of as a number of elements. These elements could be as small as a thread of control within a process, or as large as a set of

elements (recursively). In (Carriero & Gelernter 1992), Gelernter and Carriero call a collection of such elements an *ensemble*.

An ensemble need not be composed of just machine elements. A user could also be thought of as an element as they are involved somehow in the computation, even if that involvement is just collecting the results the program generates. Many of today's programs coordinate more frequently with a user via graphical user interfaces.

Thus every program consists of a number of elements which need to be coordinated. Even a single threaded program needs to coordinate its results somehow, be it to a user, a file, or another program using a form of inter-process communication such as pipes.

3.3 Relationship with computation languages

It can be argued that a coordination language is orthogonal to a computational language. In order to operate correctly, an application will need to perform both coordination and computation, with computation being used to perform processing of data, and coordination managing the processing of this data.

As can be seen from above, even the most trivial programs have a combination of computation and coordination.

Traditional languages provide coordination through access to a library of routines, often provided by the host operating system. For example the *stdio* library in Unix systems provides access to basic file management routines. As these library routines are not built in language constructs, they may be thought of as interpreted commands. There is no scope to optimise these operations based on knowledge of the implementation as a whole — the same primitives are used by whichever program invokes them.

Managing coordination of tasks through a coordination language permits optimisation on a per application basis. As computational language compilers can optimise a routine based on knowledge of the computation as a whole, so the compiler of a coordination language can optimise communication and synchronisation based on knowledge of overall program structure.

For example, if it can be determined that two processes communicate with each other frequently then it can arrange for these two processes to run on the same node of a distributed system, helping to reduce communication costs and increasing efficiency.

3.4 Features of a coordination language

Coordination languages are conceptually simple as they need to express very few constructs. A good coordination language should be able to express coordination for both tightly coupled and loosely coupled architectures, thus freeing implementers of systems from knowing about the underlying architecture they are targeting.

As discussed in the introduction, a coordination language needs to be able to represent and manipulate the basic elements of coordination which occur within a running application.

1. Create new processing elements. These could be new threads of control within an existing process or a new process entirely. New processes can be targeted at a uniprocessor, multiprocessor or distributed architectures;
2. Communicate between processing elements. This communication could be achieved in a number of ways. Shared variables allow different threads within a process to communicate values between themselves. Shared memory between processes can be used on closely coupled systems. For loosely coupled systems, message passing will need to be used to communicate between elements. The coordination language can hide aspects of this in order to present a consistent interface across architectures;
3. Synchronise processing elements. This can include suspending a thread of control until another thread signals it to continue (for example a worker thread suspends itself until a master thread gives it work to do). Another form of synchronisation is controlled access to a shared resource (for example critical sections).

The following sections give example implementations of languages which allow coordination of computational languages. An overview of each language is given including how it integrates with a computational language. The benefits and drawbacks of each approach are also discussed.

3.5 Linda

3.5.1 Overview

Linda ((Carriero & Gelernter 1989)) allows application programmers to write architecture neutral applications by providing an abstract model of a computing system. In this model all processes communicate via a global address space, known as "tuple space". It is the responsibility of the underlying Linda runtime to provide a concrete implementation of this abstraction. Such implementations exist for both closely and loosely coupled architectures.

Objects which reside in the tuple space use a universal data representation to allow sharing of data between processes written in different languages. This allows a single distributed application to be built from many processes, with processes implemented in a language that suits their function.

3.5.2 Integration with computational languages

Linda extends each computational language with four primitives which provide applications with the ability to communicate via the tuple space and to manage process creation during the lifetime of the application as a whole.

Communication is achieved by creating a data object *tuple* and adding it to the tuple space. Any other process can then read this tuple, and thus inter process communication has occurred. Note that this communication is asynchronous —

the process that writes the *tuple* will not wait until the *tuple* is consumed before proceeding. The receiving process will block until a suitable tuple is present.

A *live tuple* can be added to tuple space. This in effect creates a new process (with the Linda runtime system deciding where this process should execute). Once finished the task becomes a *data tuple*, the data being the result of the process's computation.

The four primitives defined by Linda are:

`eval` — Create a live object to evaluate an expression and place it in tuple space;
`out` — Create a data object tuple and place it in tuple space;
`in` — Remove a tuple from tuple space;
`rd` — Read a tuple in tuple space, but do not remove it.

A process selects a tuple to read/remove by specifying a template the tuple must match. The Linda runtime will then attempt to match the template to a tuple in tuple space and read/remove the tuple. For example, the operation `in("test",?x)` will remove a tuple whose first element is the string `"test"` and bind the second element in the tuple to the variable x. If no match is found the process blocks until a suitable tuple is added.

3.5.3 *Linda in operation*

The tuple space abstraction greatly aids the programmer when writing portable, distributed applications. Linda has been integrated with several computational languages, including C, Fortran, Scheme, and Modula-2. Linda runs on several types of parallel machines, and sets of workstations connected by a local area network.

In (Carriero & Gelernter 1998), Carriero and Gelernter show the applicability of Linda to a range of applications. In (Carriero & Gelernter 1989), they compare Linda with other concurrent programming systems.

3.6 Actors

3.6.1 *Overview*

From a general perspective, the Actors model resembles other models which consist of processes that cooperate using inter-process communication via message passing, as message passing in the Actor model uses unidirectional, asynchronous communication with unbounded buffers.

However, it does possess features which distinguish it from traditional message passing models, with support for dynamic process creation and the use of *arrival* ordering of messages.

Hewitt and Baker defined a set of laws ((Hewitt & Baker 1977)) (see table 3.1), which can be derived from first principles. These, together with Hewitt's work on control structures ((Hewitt 1979)) as patterns of message passing, form the basis of the development of the Actor model. Agha's thesis ((Agha 1986)) provides a comprehensive description of this model.

Every object in the Actor model is an actor, for example the number `one`, or the function `nfib`. The system progresses by actors asynchronously sending messages to each other (where messages are also actors).

There are three kinds of actor:

Primitive actors — These correspond to basic data types and primitive functions of the computational language, such as the number `5` and the multiplication operation;

Serialised actors — Contain mutable local state, modifiable by the actor. Such actors are commonly used as data repositories, such as the *register* or *stack*;

Unserialised actors — contain immutable state. For example the *factorial* function may be implemented as an unserialised actor in terms of other primitive or unserialised actors such as *true*, *one*, *multiply*, and recursively, *factorial*. Any number of instances of the same unserialised actor may be executed, as referential transparency shows that this is safe given no mutations of state.

Actors communicate using messages, which are themselves actors. Each actor in a system responds to a fixed set of messages. Concurrency is introduced by an actor sending a number of messages in response to one input message, and is correspondingly reduced by an actor which receives a message not sending any new messages.

Messages are composed and sent asynchronously to target actors. A weak fairness rule guarantees which all messages are delivered sometime in the future, and will eventually be processed by the receiving actor. As messages are themselves actors they may be created by sending a message to the primitive *create-unserialised-actor*.

One important class of actors is the *continuation* actor. This actor captures the state required to complete evaluation. Typically an actor will send a continuation as part of a message to a target actor, which will send its reply to the received continuation. This effectively blocks the client actor until its request has been processed, allowing for synchronisation within an asynchronous message passing system.

Every actor may be described by a script(*behaviour*) and acquaintances(*environment*). Scripts are applied to incoming messages, where templates match the message and specify the behaviour of the actor.

The acceptance of a message by an actor is termed an "event". The exclusion of iterative constructs in the language other than recursive message passing ensures that an actor script executes in a finite time on receipt of a message, allowing an underlying *run to completion* (or *cooperative*) scheduler to implement the weak fairness guarantee. This also ensures that an actor script cannot flood the system with an infinite number of messages.

1. If an event E_1 precedes an event E_2, then only a finite number of events occurred between them;
2. No event can immediately cause more than a finite number of events;
3. Each event is generated by the sending of at most a single message;
4. The event ordering is well-founded (one can always take a finite number of "steps" back to the initial event);
5. Only a finite number of actors can be created as an immediate result of a single event;
6. An actor has a finite number of acquaintances at any time. Acquaintances are the actors to which it can send messages. An actor's acquaintances when processing a message are the union of its own acquaintances and the acquaintances of the message.

Table 3.1: Laws for actor systems

3.6.2 *Integration with computational languages*

The pervasiveness of the message, together with the *continuation* actor makes binding the actor model into existing computational languages problematic. Most computational languages have no direct support for continuations, and efficiently implementing message passing primitives with fine grain parallelism is difficult.

Several new languages have been defined which make use of the Actor model, whilst providing support for computation. SAL, which has an Algol-like syntax is one example, and Act3 ((Hewitt *et al.* 1984)) is built on a simpler actor language, Act. Act has a Lisp like syntax, with complex pattern matching being used to bind identifiers.

3.7 Opus

3.7.1 *Overview*

Opus ((Haines *et al.* 1996)) extends the Fortran programming language to provide a coordination model which allows an application to be distributed over a number of nodes. These nodes may either use a shared memory, or have disjoint address spaces with message passing providing the illusion of a shared memory. The actual address model is hidden from the programmer, and supported by the Opus runtime.

Opus introduces the SDA (ShareD Abstraction) to Fortran. This may be thought of as a class like structure in C++, although it has no support for inheritance. An SDA consists of a set of data elements and methods which act on these elements.

Access to an SDA is serialised — i.e. at any one time only one method of an SDA may be executing. External access to an SDA's data elements is also serialised. This frees the programmer from having to use critical sections and locks at the expense of possible loss of some parallelism. For example it might be safe for two methods in the same SDA to act concurrently if one does not modify data in the SDA that the other needs to access, but Opus will always serialise these methods as they belong to the same SDA.

Serialising access to an SDA also reduces the possibility of deadlock, but does not remove it. Although methods within an SDA cannot deadlock each other as

they never run concurrently, if a method needs to call a method in another SDA then this could cause deadlock. For example SDA A method x calls SDA B method y, but method y is blocked because method z is currently executing. If method z were to call another method in SDA A then deadlock would occur.

Preconditions may be attached to methods (as **when** clauses) which specify what state an SDA has to be in for a method to execute. For example in a stack SDA the pop method may have a precondition that there is at least one element on the stack.

Calls to SDA methods may be synchronous or asynchronous. In the asynchronous method any **OUT** parameters are invalid until the SDA method completes. The SDA method returns an event that may be tested (asynchronous) or waited on (synchronous).

A set of constructs is provided to manage SDAs. These include SDA creation, deletion, finding existing SDAs and serialising and unserialising SDAs from/to persistent storage.

3.7.2 Integration with computational languages

As stated in the previous section, Opus is closely tied to Fortran to provide the computational part of a distributed Opus application. Opus is similar in concept to concurrently executing objects, and other objected oriented languages have similar support for concurrency (for example concurrent C++ and Corba).

3.8 Manifold

3.8.1 Overview

In (Arbab 1995), Arbab introduces two models for coordination of massively concurrent activities.

Targeted send/receive

This models traditional message passing systems such as sockets. Processes send messages to a specific process by providing some unique address that the underlying message system can resolve into a physical destination (processor and process pair). Receiving processes typically will not specify a source from which they wish to receive, although security considerations will limit this set of processes.

Idealised worker/idealised manager

The Idealised worker/idealised manager (IWIM) model abstracts coordination away from the underlying communications network. A worker process has a number of input and output *ports* through which it receives and sends data.

A manager process is responsible for creating worker processes and configuring the (abstract) communications network by creating *channels* and attaching them between input and output ports of worker processes.

The manager process can dynamically change this network throughout the lifetime of an application.

Managers can also be managed, acting as a worker to a higher level manager. This capability allows a distributed application to be decomposed into a set of concurrent modules, with each module having its own manager.

The Manifold coordination language

Arbab goes on to discuss Manifold, a coordination language which implements the IWIM model. Worker processes are written in a high level computational language, interfacing with a library which provides the implementation of *ports* and support for linking an application into a Manifold application. Channels can perform either synchronous or asynchronous communication between ports.

Manager processes are written in Manifold, which is implemented as a state transition language. This allows networks to be created out of worker processes, without explicit knowledge of the worker's implementation language. Workers may raise events which cause state changes in their manager process, allowing managers to dynamically reconfigure their workers.

3.8.2 *Integration with computational languages*

Manifold uses a **C** runtime library to provide *ports* and *events* to worker processes. Thus the only constraint on a suitable language for the implementation of worker processes is that it must be able to link with **C** libraries.

C and **C++** are obvious choices, but other languages which have foreign function interfaces can also be used. Data that flows through ports are of standard **C** types, thus any foreign language must be able to transform data from its internal representation to a **C** representation.

3.9 MeldC

3.9.1 *Overview*

MeldC presents a distributed object oriented environment to the programmer. Unlike other systems which class themselves as a coordination language, MeldC includes a computational language as well as mechanisms for the coordination of computations.

In MeldC, most entities within the system are represented as objects. Messages can be sent to these objects, with the MeldC kernel routeing messages to the correct destination. Data types which cannot be represented as objects are modelled as built in types in the MeldC kernel. For example the message cannot be represented as an object, as a message would have to be sent to the message class in order to create a message, resulting in an infinite regression. Other primitives in the MeldC kernel which cannot be modelled as objects include threads and synchronisation routines.

Threads are used to handle requests made to objects by other MeldC objects. A transformation process is used to turn a message into a thread which dispatches a routine to process the message. On return the the thread is transformed back into a reply message and sent back to the requesting object.

Other network resources can be modelled in MeldC by creating gateway objects. These provide a MeldC object interface to other objects running in the MeldC application, and translate requests into the appropriate behaviour to operate the foreign resource.

The MeldC object system allows object behaviour to be extended in two ways, through structural and computational reflection. The structural reflection employs a meta object system ((Kiczales *et al.* 1991)) to allow the application writers to change behaviour of classes. All classes are first class objects in MeldC, with classes generally being instances of the class `Metaclass`, with `Metaclass` being an instance of itself. `Metaclass` defines the behaviour of creating and destroying objects. By deriving a class from `Metaclass`, new types can be defined with different behaviours from instances of `Metaclass` (for example, persistence).

MeldC also provides a mechanism for objects to dynamically change their behaviour through object composition and computational reflection. Secondary behaviour is added to an object by attaching a shadow object. This shadow object is a regular object, which receives requests sent to the primary object, allowing it to change the behaviour of the object to which it is attached without the need for the system to be restarted. For example, an audit object can shadow an accounting object, which in turn may be shadowed by a monitoring object. The meta object system is used to provide this functionality.

3.9.2 Integration with computational languages

MeldC provides its own computational language in addition to message passing primitives that provide support for the coordination of objects.

Foreign resources may be incorporated in MeldC through the use of gateway objects.

3.10 Logic based coordination

3.10.1 Overview

In (Diaz *et al.* 1996), it is shown how the use of logic variables can be used to coordinate multiple processes.

Two new data types are introduced to the computational language.

The logical channel — This data structure is used to transmit and receive tuples of data between processes. The primitives `Put(x)` and `Get(x)` are used to implement non blocking sends and blocking receives respectively. The channel network can be defined with one producer and many consumers to implement a form of multicast;

The logical variable — This variable can represent any standard data type within the language, and can be transmitted as a higher order object over logical channels. The variable may only be instantiated once, with consumers blocking until the variable is instantiated.

The ability to pass logical variables over channels provides a method for implementing synchronous requests. The logical variable is passed along with other

data in the request tuple, and once completed, the server can instantiate the logical variable with the request result. The client blocks, waiting for the logical variable to become ready, and once read the request is complete.

Note that as channels are not higher order objects themselves, they may not be passed as data in tuples over other channels to provide a return path to the client.

3.10.2 *Integration with computational languages*

Any language which can support the addition of the logical channel and variable constructs (be it through macros or an additional preprocessor step) is suitable for operation with the logic channel coordination model.

A suitable common data representation needs to be chosen for the transmission of tuples over channels and the setting of logical variables so that processes written in dissimilar languages or operating in a heterogeneous environment can correctly communicate.

3.11 Structured dagger

3.11.1 *Overview*

Structured dagger ((Kale & Bhandarkar 1996)) has been designed to provide support for coordination amongst processes written in Charm ((Kale *et al.* 1994)), a distributed message passing language.

Charm objects are augmented by entry methods. These blocks specify computations and the dependencies of these computations to messages, including the arrival order of messages and dependencies on other computations.

For example, a constraint could be placed on a stack object such that it will only process *pop* messages when there is data on the stack. In base Charm, this case would have to be explicitly handled by consuming the *pop* message and buffering it until a *push* message was processed. This is achieved by placing a `when` block in the charm object, allowing a guard to be specified that must hold before a received message can be processed.

The `seq` and `overlap` constructs inform the compiler about dependencies between blocks of code. The `seq` construct means that all operations must be performed in sequence, whereas the `overlap` constructs allows concurrent execution on receipt of multiple messages.

3.11.2 *Integration with computational languages*

Structured Dagger is closely linked to Charm. This is a machine independent parallel programming language. The Charm language is similar to **C** with a few syntactic extensions.

Programs consist of medium grain objects (*chares*). These *chares* interact by sending messages between themselves and through the use of special information sharing modes.

3.12 Agora

3.12.1 Overview

Agora ((Bisiani & Forin 1998)) uses a (virtual) shared memory for the communication of data between processes. Each process running in the Agora environment has an event queue which is used for event driven synchronisation between processes.

Objects stored in the shared memory are statically declared and must declare their type. Data types allowed in the shared memory include basic types, records and hash tables, but cannot include recursive types or pointers.

Objects residing in shared memory are write once, with subsequent writes producing new copies of the object. Each process has a map of object to shared memory address. These maps are lazily updated as object contents change. A mark and sweep garbage collector is used to manage the shared memory.

In loosely coupled systems the shared memory is simulated with the use of Agora servers, with a master slave replication model used to maintain state. Write requests are always sent to the master server. Reads are cached by processes, with the master server broadcasting (either using a network broadcast, multicast or multiple point to point messages) object changes to all processes which have cached copies of the object.

Shared objects have names, with a flat name space held on the master Agora server. Processes initially read objects in the shared address space by binding to the object providing its name as the key. As most of these lookups occur during the initialisation phase of the application, system performance is generally not impacted by having to send messages to the server to perform object lookups.

3.12.2 Integration with computational languages

Agora has bindings with CMU Common Lisp, C, and C++. The authors show how the shared memory system can achieve comparable results with message passing systems. In simulated shared memory implementations running on loosely coupled systems, fewer messages are used over traditional message passing systems, especially where the network supports broadcast or multicast to update caches with changed objects.

3.13 Summary

Coordination languages provide the ability for processes to communicate in a structured fashion. This coordination can be orthogonal to the computation an application carries out.

Languages such as Linda and Manifold provide constructs for coordination which are independent of the computational language. This allows the applications programmer to choose a computational language which is suitable for implementing the chosen solution, whilst writing to a common coordination model.

Other languages such as Structured Dagger and Opus are closely tied to a computational language (Charm and Fortran respectively). Providing implementations for other languages might be possible, but assumptions about available data types and computational forms might make parts of the coordination model difficult — especially if the coordination model needs to work with processes written in multiple languages.

Although coordination and computation may be considered orthogonal, there will always be overlap between the computation and coordination modules in an application. For example, data will in some way need to be shared between processes, thus enforcing a set of data types which a coordination system can handle. If the coordination system is designed to work in a heterogeneous environment with processes written in different languages interacting, this set of data types becomes an intersection of the data types of all the supported computational languages.

This can limit the expressiveness of a distributed application. For example, Lisp has support for higher order objects such as *continuations* and *closures*, whereas other languages such as C have no direct support for such types. Thus if an application were to make use of C and Lisp it would be unlikely that these higher order expressions could be used as part of the common data types.

MeldC shows that by combining coordination with a specific computation language, an arguably more powerful distributed environment can be created over a system using a separate computation and coordination language. However, the disadvantage of this closed system is the requirement to create gateway objects to access services not directly written in MeldC (although it can also be argued that gateway processes need to be written for any service not implemented using a chosen coordination language).

Chapter 4

Coordination in a mobile environment

4.1 Introduction

This chapter discusses a method for writing distributed applications which are made up of mobile processes. This presents challenges when designing the primitives which are used to provide coordination and communication within the application.

For a process to be mobile, a method of locating the process needs to be defined so other processes in the network may communicate with it. In static networks this can be determined at compile time, or system initialisation time where process placement is defined by a separate configuration file. However mobile networks require a more dynamic solution to maintain an effective communications network.

The following section discusses current models of distribution, and how these models may be used to provide a framework for writing systems with mobile processes.

Section 4.4 goes on to describe how a particular model using channels can be used as a coordination model for dynamic systems, and chooses a suitable computation language to bind with the coordination system in section 4.6.

Finally, section 4.8 describes possible implementations of the distributed system.

4.2 Models of distribution

As with other large engineering projects, modelling can play a part in the building of a distributed computing system. Various aspects of the system can be modelled, from individual components which make up the system to the structure as a whole.

Modelling the interactions between different processes in the system allows the distributed application designer to investigate which designs produce the most efficient solution to a problem. The efficiency of a solution can be measured in a

number of ways, for example by calculating the computation time of each process and finding the slowest path of execution in the application. Another method is to examine the number of cross process interactions which are likely to occur in the final system. Each interaction is likely to involve at least one message being passed between the processes, which are possibly executing on different nodes. Message passing is likely to be the slowest part of the system, as it relies on networking technology that runs at a fraction of the speed of processors, causing the network to become the bottleneck of a distributed system. By reducing the number of interactions between processes which form the critical path of execution in a distributed application, improvements in the overall system performance can be made.

Models which are based on a formal semantics can also be passed through theorem provers ((FDR 1983)) to investigate whether desired properties of the system hold true (for example, that interactions between processes never cause deadlock).

At the lowest level of a distributed system the unit of interaction between two processes operating on different nodes is the message. A number of models have been designed which abstract the message passing layer using channels.

In essence a channel may be thought of as a pipe, with two ends. One end of the pipe may be used to send messages, with the other receiving these messages. In the case of a bi-directional channel, either end may send and receive messages over the channel. Implementations of message passing systems are able to be modelled using channels. For example a socket could be expressed as a bi-directional channel, able to send and receive messages between two fixed points on a network. Thus a formal model based on channels may be able to be directly implemented using sockets as the physical representation of the channel.

Other models which have the channel as the basic element of communication make the mapping of a model to an underlying message passing implementation more complex as they extend the notion and capabilities of a channel.

The following sections present a number of languages which use channels for inter-process communication.

4.2.1 CSP

Communicating **S**equential **P**rocesses ((Hoare 1985)) provides a notation for defining a static network of processes. Each process executes a sequence of events (for example $a \rightarrow b$ represents the event a followed by the event b), whose names constitute the *alphabet* of the process. Recursion within processes is the only permitted looping construct (for example $P = a \rightarrow P$). Two processes running in parallel synchronise on events common to alphabets of both the processes — these events must execute in lock step fashion. The choice operator allows a process to proceed in a number of ways depending on which event is chosen, allowing non determinism to be introduced into a system.

The model has been extended to allow communication as well as synchronisation between events. This is achieved by one process outputting a value and

another process inputting the value. The communication occurs when processes synchronise on an event.

An example of a CSP process is a static network to act as logic gates. Two gates are shown below, the unary NOT gate and the binary AND gate. Each gate synchronises on its inputs and outputs the result of the logical operation. The binary operation introduces non determinism by allowing either of its input gates to become ready first.

$$\begin{aligned} NOT(a,b) &= a?l \rightarrow b!fnNot(l) \rightarrow NOT(a,b) \\ AND(a,b,c) &= (a?l_1 \rightarrow b?1_2 | b?l_1 \rightarrow a?1_2) \rightarrow c!fnAnd(l_1,l_2) \rightarrow AND(a,b,c) \end{aligned}$$

CSP has no support for the mobility of processes and reconfiguration of communications networks. The *alphabet* of a process is fixed, and is not a first class object (i.e. it cannot be used as an object communicated over an event). The process is also not treated as a first class object, and hence may not migrate around the network.

The most well known implementation of CSP is Occam(see section 2.3.1), but others do exist. For example an integration of CSP into Lisp is described in (C.J.Fidge 1998).

4.2.2 *The π-calculus*

> The π-calculus is a way of describing and analysing systems consisting of agents which interact among each other, and whose configuration or neighbourhood is continually changing ((Milner 1991)).

In the monadic π-calculus((Milner 1991),(Milner 1993)) there are two entities. The most primitive is the *name* (also referred to as a link, or channel). Processes are the only other entity in the system, with names used to specify interfaces between processes. If two processes have access to a common name (using normal lexical scooping rules) then those two processes can communicate using that name.

Communication between processes is synchronous. Values may be transmitted one way during a communication. In the case of the monadic π-calculus, only names may be transmitted during a communication. This implies that names are dynamic, allowing networks of processes to reconfigure themselves during execution. For example, the following expression reduces, allowing two initially unconnected procedures to communicate.

$$\begin{array}{lllll} \bar{a}b.P & | & a(c).\bar{c}.Q & | & b.R \Rightarrow \\ P & | & \bar{b}.Q & | & b.R \Rightarrow \\ P & | & Q & | & R \end{array}$$

action terms A ::=	$\bar{x}y.P$	send y over x
	$x(y).P$	receive any y over x
terms P ::=	$A_1 + \ldots + A_n$	alternative action
	$A_1 \mid \ldots \mid A_n$	concurrent action
	$\nu y P$	restriction (new y in P)
	$!P$	replication ($!P = P \| !P$)

basic rule of computation :

$$x(y).P_1[y] | \bar{x}z \rightarrow P_2 \Rightarrow P_1[z] | P_2$$

Table 4.1: π-calculus constructs

Constructs for the specification and creation of processes allow systems to be defined. The dynamic nature of the π-calculus extends to processes, with constructs provided to create new processes during execution, again dynamically changing the structure of the network. Table 4.1 shows the constructs available to the monadic π-calculus.

The polyadic π-calculus extends the monadic π-calculus by allowing multiple names to be transmitted over a name in a single operation. Milner shows that this is syntactic sugar and may be expressed using the monadic π-calculus by the addition of temporary names to pass the multiple values. In the case of output the temporary name is first transmitted, and then the values are sent over this temporary name.

$$\bar{x}(y_1 \ldots y_n) \Rightarrow (\nu w)(\bar{x}w.\bar{w}y_1.\ldots.\bar{w}y_n)$$

For input the reverse happens. The temporary name is received and then the multiple values are received from this temporary name.

$$x(y_1 \ldots y_n) \Rightarrow (\nu w)(x(w).w(y_1).\ldots.w(y_n))$$

As can be seen from the above examples, the polyadic representation is easier to understand than the monadic case. By treating it as an atomic action, deadlock can be avoided. For example in the above cases n values are transferred over channel x. In the monadic case, the sender and receiver must agree on this value of n or one process will block. In the case of the polyadic π-calculus a rule can be introduced that in order to communicate, both processes must have the same arity in their input/output operations.

Processes are also promoted to first class objects in the polyadic π-calculus. In the monadic π-calculus the only first class object is the name. The process, the only other type in a π-calculus system, cannot be directly manipulated. Instead names which represent access to the process are manipulated. For example it might be desirable to introduce a new process into a system by transmitting it over a name.

$$\bar{x}P.Q | x(R).R | S \Rightarrow Q | P | S$$

This is not possible in the monadic π-calculus. Instead a name (r) which can communicate with R is used instead.

$$\begin{array}{lll} \bar{x}r.Q & x(a).a.S & \bar{r}.P \Rightarrow \\ Q & r.S & \bar{r}.P \Rightarrow \\ Q & S & P \end{array}$$

As it has been shown that the passing of processes can be modelled by a suitable encoding in the monadic π-calculus, it is convenient syntactic sugar to be able to treat processes as first class objects like names. This allows writing π-calculus which expresses mobile code. However there is no encoding to say where a process is placed, which is a useful element to model when designing distributed applications.

If a notion of placement were to be introduced in the π-calculus then the above expressions may no longer be equivalent. In the first case the processes may have to move between two different nodes, whereas in the monadic case, the channel representing the process moves between sites and the process itself does not move.

Much work has been done to extend the concepts of the π-calculus to make it more relevant to modelling distributed applications. The spi-calculus ((Abadi & Gordon 1997)) extends the π-calculus with cryptographic primitives. The join-calculus ((Fournet *et al.* 1996)) reformulates the π-calculus with a more explicit notion of places of interaction. This aids the building of distributed applications using channels, as the π-calculus itself has no notion of process placement. The join-calculus adds the concept of named locations, and a notion of distributed failure. Locations form a tree, and subtrees can migrate from one part of the tree to another.

Mobile ambients ((Cardelli & Gordon 1998)) describes a calculus which allows movement of processes and devices through domains of administration. A study of the correspondence of the λ-calculus and the π-calculus leads to the definition of a new calculus, the blue calculus ((Boudol 1997)). It is shown that a continuation passing style can be used to transform the blue calculus into the π-calculus, with the π-calculus representing a sort of assembly language.

4.2.3 *Higher Order Communications*

The language HOC ((Henderson 1993)) is based on the π-calculus. It provides constructs which allow concurrent processes to be defined that communicate via synchronous channels. Like the π-calculus, HOC allows channels to be passed as messages along channels, giving it the ability to dynamically change the configuration of processes. Processes may also be passed over channels, allowing for the migration of processes between nodes. In addition the replication operator of the π-calculus is dropped in favour of a recursive mechanism to provide looping.

Table 4.2 shows the language constructs available in HOC.

action terms A ::=	$x!y \rightarrow P$	send y over x
	$x?y \rightarrow P$	receive any y over x
terms P ::=	$A_1 \mid \ldots \mid A_n$	alternative action
::=	$A_1 \parallel \ldots \parallel A_n$	concurrent action
::=	$P_1; P_2$	composition
::=	$skip$	terminate successfully

basic rule of computation :

$$x?y \rightarrow P_1[y] \parallel x!z \rightarrow P_2 \Rightarrow P_1[z] \parallel P_2$$

Table 4.2: HOC constructs

4.3 Other models of distribution

In addition to models focused on the channel as the basis of communication between processes, a number of other models have been developed which describe a set of interacting processes. Some have direct implementations, others are strictly theoretical.

4.3.1 Unity

The Unity model ((Chandy & Misra 1988)) allows concurrent execution of guarded expressions. At any time guards for expressions may be evaluated, and if true, the expression executed. Expressions whose guards hold may be executed in parallel, so care must be taken when expressions cause side effects. A construct for performing parallel assignments is provided.

An implementation using EULISP to map unity programs onto BSP-Occam is described in (DeRoure 1991).

4.3.2 The Paralation model

The Paralation model ((Sabot 1988)) is designed to be independent of computer language and architecture. Several implementations for various languages exist, including Lisp and C. In contrast with CSP which places emphasis on the process as the main source of parallelism, the Paralation model is a data parallel model.

The basic data type is the *field*, which contains data residing on differing sites (where sites are mapped onto available processors). A paralation defines a collection of such fields. A small number of primitives are provided to perform data parallel computation on fields (where the same procedure is applied to each site in the field), and map fields into new paralations.

For example the code to create a new field in a paralation with its elements incremented by n is as follows:

```
(defun incfield (f n)
  (elwise (f) (+ f n)))
```

This data centric view of parallel processing maps well onto SIMD multiprocessor hardware such as vector processors.

4.3.3 *Petri nets*

Petri nets ((Filman & Friedman 1984), (Peterson 1977)) are directed graphs with *places* and *transitions* as nodes in the graph. A place can contain a number of tokens, and a transition can have many inputs and outputs. All inputs must be an edge from a place to the transition, with outputs being edges from the transition to a place. A transition may "fire" if all the places connected to its inputs contain tokens. On firing, all input places have one token removed and all output places a token added. Extensions to the model include coloured tokens and firing when a specified number of tokens is present in the input places.

4.3.4 *Document flow model*

Actons are the only processes in the document flow model ((Berrington, DeRoure, Greenwood & Henderson 1993), (Berrington *et al.* 1994)). An acton maintains a collection of documents in its "in tray", and has a set of rules that may act on these documents. For example, an acton could combine two documents into a single document (such as a report document and a changes document being combined into a finished report document). Rules can also be used to send documents to another acton in the network, allowing for the workflow applications to be modelled.

An implementation of the document flow model is presented in section 7.4, together with sample DFM applications.

4.3.5 *Time Warp*

The Time Warp model ((Jefferson 1985), (Tinker & Katz 1988)) stems from database transaction systems ((Kung & Robinson 1981)), allowing for optimistic concurrency. Processes proceed without synchronisation and can later roll back if actions they performed are found to be incorrect. Causality is maintained though the use of distributed clocks ((Lamport 1978)).

The system needs to capture the state of a process before each action is taken so that it may be rolled back if it is found that the action should not have taken place. Rollback is achieved by sending *anti-messages* for all events following the event to be rolled back to. This in turn can cause other processes to roll back their state, resulting in more *anti-messages* being sent. There is a danger that the system could spend more time rolling back than rolling forward, resulting in wasted resource and slow progress in a system. However Gupta ((Gupta *et al.* 1991)) has shown that implementations of the model do show degrees of speed up.

4.3.6 *Chemical abstract machine*

The chemical abstract machine ((Berry & Boudol 1992)) or *cham* provides a novel approach to the modelling of parallel applications. Rather than focusing on a process and message model like CSP or the π-calculus, it abstracts away from these elements and instead presents a framework in which non determinism plays a large part in the progress of a system.

A *solution* contains many *molecules*, which are an abstract representation of some data type (for example, one could think of a solution of integers). This solution is stirred by some method (Brownian motion in traditional chemistry, and some arbitrary mechanism in this model), allowing molecules to come into contact with other molecules.

A set of *reaction rules* specify how molecules that meet may interact, allowing the system to proceed. Further rules allow the combination of molecules (*cooling rules*) and the splitting of molecules into a set of simpler molecules (*heating rules*).

For example, returning to the solution of integers, a rule could be introduced such that when two molecules meet, the larger integer is destroyed if it is a multiple of the smaller integer (other than *one*). Once the reaction had finished, the solution would contain only prime numbers, as all the primes multiples would have been removed.

It has been shown that the chemical abstract machine is general enough to be able to model the π-calculus and a concurrent λ-calculus.

4.4 HOC as a coordination language

By combining HOC with a higher order computational language, a language for programming dynamically changing networks of interacting processes can be envisaged.

This section examines the features that HOC possesses which make it suitable for programming networks of mobile processes. Changes to the model are introduced in order to model the placement of processes.

4.4.1 Dynamic configuration

One feature of HOC is the ability to reconfigure dynamically the communications infrastructure of a set of processes by transmitting channels over other channels. In this way, a process receiving a channel can then use this new channel to send and receive further messages, resulting in the network between sending and receiving processes changing over time.

This is useful with the presence of mobile processes, as a constant method of communicating with the process can be used (via a channel which the process "takes with it" when it migrates).

If this were not the case, a separate mechanism for locating processes would be required and connections would have to be re-established before communication could continue.

4.4.2 Mobility

The ability to send processes over channels as higher order objects allows the modelling of mobility in a distributed environment.

As demonstrated in section 4.2.2, in a distributed environment, two processes communicating using a common channel need not be executing on the same node.

Therefore sending a process over a channel can cause that process to move between nodes. In the presence of distribution, the following reductions become ambiguous:

$$(x!P \rightarrow A)||(x?Q \rightarrow Q||B) \quad \Rightarrow A||P||B$$
$$(x!() \rightarrow A||P)||(x?() \rightarrow B) \quad \Rightarrow A||P||B$$

These two expressions reduce to the same set of three parallel processes. However, in the first expression the process P is sent over channel x and the receiving process then causes the process to be executed, whereas in the second expression, no data is passed when the two processes initially synchronise using channel x, and the left hand side process causes P to be executed.

From the above expressions it is not clear on which node each process executes. If a rule is introduced that the $||$ (parallel execution) operator causes processes to be executed on the same node as the parent processes (unless explicitly specified), then the node on which processes execute can be made explicit by parameterising them with their node identifier.

For example, suppose two nodes are given the names a and b. The above expressions could be rewritten as follows:

$$(x!P \rightarrow A)_a||(x?Q \rightarrow Q||B)_b \quad \Rightarrow A_a||P_b||B_b$$
$$(x!() \rightarrow A||P)_a||(x?() \rightarrow B)_b \quad \Rightarrow A_a||P_a||B_b$$

These expressions now explicitly show the node on which each process executes. The first expression shows that although the process executing on node a has access to process P, it does not execute it. Instead, it sends it over channel x, where the receiving process on node b causes it to be executed. Therefore this expression shows the process P has migrated between node a to node b. The second expression shows that no migration has taken place, and that process P is executed on node a, and that the process executing on node b has no knowledge of P.

An example of process mobility is the *Entrance* process. This process accepts all incoming processes being sent over a channel and executes them locally on the node on which it is executing.

$$Entrance(x) = x?P \rightarrow Entrance(x)||P$$

Such a process could be used to allow processes to migrate between well known nodes, if an initial network is set up with entrance processes operating on nodes parameterised with global channels. For example the following network uses two

global channels $goto_a$ and $goto_b$ to allow a travelling process to migrate between nodes.

$$\begin{aligned} Entrance(goto_a)_a || Entrance(goto_b)_b || (goto_a!(goto_b!P))_c \Rightarrow \\ Entrance(goto_a)_a || Entrance(goto_b)_b || (goto_b!P)_a \Rightarrow \\ Entrance(goto_a)_a || Entrance(goto_b)_b || P_b \end{aligned}$$

This example shows a process initially executing on node c migrating a process to node a, which in turn migrates process P to node b.

4.4.3 *Conclusion*

Using Higher Order Channels as a communications model provides the ability for mobile processes to communicate and synchronise over a dynamically changing configuration.

By extending the model to include the placement of processes, the system can model a distributed set of processes and the migration of processes over channels to other nodes in the network.

4.5 HOC as a computation language

The π-calculus and HOC have only the process and channel as their basic data types. Although these languages can be used to describe the interactions between processes in a complex system, actually representing the computation that processes perform in terms of π-calculus and HOC constructs is more problematic.

4.5.1 *Data types in the π-calculus*

The π-calculus and HOC can themselves directly model other data types. Milner shows in (Milner 1991) that the π-calculus can be used to represent λ calculus expressions. It can also be used to model a number system directly using just processes and channels.

Two channels are used to represent a number process, a channel representing a unit and a channel representing zero. For example, a process which represents the number three can be directly encoded as follows in HOC.

$$three(u, z) = u!() \rightarrow u!() \rightarrow u!() \rightarrow z!()$$

Processes can then be designed which perform arithmetic operations on numbers. To add two numbers for example, two sets of unit and zero channels need to be supplied as inputs, and a new set of unit and zero channels provide the sum.

$$\begin{aligned} &add(u_1, z_1, u_2, z_2, u_{sum}, z_{sum}) = \\ &\quad u_1?() \rightarrow u_{sum}!() \rightarrow add(u_1, z_1, u_2, z_2, u_{sum}, z_{sum})| \end{aligned}$$

$$z_1?() \rightarrow copy(u_2, z_2, u_{sum}, z_{sum})$$

$$copy(u, z, u_{copy}, z_{copy}) =$$

$$u?() \rightarrow u_{copy}!() \rightarrow copy(u, z, u_2, u_{copy}, z_{copy})|$$

$$z?() \rightarrow z_{copy}!()$$

The process synchronises on the unit channel of the first number until the first number sends on its zero channel. For each unit received, a corresponding transmit is performed on the unit channel, representing the sum of the two numbers. When the zero channel synchronises, the second number in the sum is copied (including the zero) to the sums channels. When the zero channel of the second number synchronises, the addition has finished and the zero channel of the sum is used to signal this.

The following process can be used to represent the number six, by building a network of processes that add the number three twice in order to produce the desired result:

$$six(u, z) = \nu(u_1, z_1, u_2, z_2)(three(u_1, z_1)||three(u_2, z_2)||add(u_1, z_1, u_2, z_2, u, z))$$

Multiplication of numbers is more problematic as number processes are only one shot, in that once a number process has output its value, it terminates. One solution is to spawn multiple copy processes to provide enough copies of the number to satisfy multiplication by repeated addition. Another solution is to redesign the number process to be a server, spawning processes to deliver its value when requested. The numbers server process has one channel, on which it outputs unique unit and zero channels. Once a unit and zero channel have been sent, a process is spawned to deliver the number over these channels as before. The number three process now becomes:

$$\begin{aligned} three(s) &= \nu uz(s!(u, z) \rightarrow three(s)||threeClient(u, z)) \\ threeClient(u, z) &= u!() \rightarrow u!() \rightarrow u!() \rightarrow z!() \end{aligned}$$

Each mathematical operator must also now be expressed as a server process, allowing networks of arithmetical expressions to be built. The multiplication server and client process is given below.

$$multiply(a, b, product) =$$

$$(\nu u_1 z_1 u_{product} z_{product})(product!(u_{product}, z_{product}) \rightarrow a?(u_1, z_1) \rightarrow$$

$$multiplyClient(u_1, z_1, b, u_{product}, z_{product})||multiply(a, b, product))$$

$$multiplyClient(u_1, z_1, b, u_{product}, z_{product}) =$$

$$u_1?() \rightarrow (\nu u_2 z_2)(b?(u_1, z_2) \rightarrow copyUnits(u_2, z_2, u_{product})) \rightarrow$$

$$multiplyClient(u_1, z_1, b, u_{product}, z_{product})|z_1?() \rightarrow z_{product}!()$$

$$copyUnits(u, z, u_{dst}) =$$
$$u?() \rightarrow u_{dst}!() \rightarrow copyUnits(u, z, u_{dst}) | z?() \rightarrow skip$$

Similar techniques can been used to implement addition, subtraction, division and copying of numbers using the number server protocol. This technique is not pure π-calculus as it relies on the dynamic creation of processes and on recursion, which are not fundamental properties of the π-calculus.

4.5.2 Conclusion

The previous section highlights a number of problems in representing a number system using the π-calculus:

- Efficiency. Computers have a very efficient means of representing numbers. It is unlikely that the channels and process representation of numbers used by the π-calculuswill be anywhere near as efficient as using a computer's native representation however well optimised the π-calculus runtime system is;
- Complexity. In order to perform a mathematical operation in the π-calculus, a complex network of channels and processes must be constructed. Contrast this with the ease with which mathematical operations can be achieved in traditional computational languages.

It is clear that in order to write processes which perform computations and represent data types in an efficient manner then HOC alone is insufficient. Instead a computational language needs to be available, with HOC constructs being used to perform coordination and communication between the processes.

4.6 Choice of computation language

A number of considerations need to be taken into account when choosing a computational language. The following sections introduce the factors that need to be considered when choosing one which has to be able to support a dynamic set of mobile processes.

Lisp is treated as a strong candidate due to its support for dynamic programming.

4.6.1 Binding the computation and coordination language

Any computational language is likely to have more special forms than the coordination language with which it is being integrated (in this case HOC). It therefore makes sense to merge HOC into the computational language, rather than the other way around. This is further supported by the fact that HOC syntax can not be directly encoded by today's keyboards and so will need to be changed to make it easier to program.

There are a number of ways in which extensions to the computational language to support coordination can be achieved.

```
;; Definition of a box
(define (make-box) (cons 'box ()))
(define set-box-value! Set-cdr!)
(define box-value box cdr)

;; Receive two numbers over a channel and return their sum
(define (channel-add c)
  (let ((b1 (make-box)) (b2 (make-box)))
    (channel-input c b1 b2)
    (+ (box-value b1) (box-value b2))))
```

Figure 4.1: Receiving values over a channel using boxes

Library

A library of functions with a well defined application programming interface could be provided to supply the coordination capabilities to the computational language. The applications programmer could then use the standard features of the language to perform relevant computations, calling on functions in the coordination library to perform synchronisation and message passing. For example, the following C function prototype could represent the API for sending data over a channel:

```
void ChannelOutput(Channel* c, Data* d1, Data* d2);
```

This function would output the supplied data variables `d1` and `d2` over channel `c`, and return when a synchronisation with a process which received data from `c` was completed.

The design of a function to input data over a channel is less obvious. In HOC, data received over a channel is bound to the receiving process's arguments supplied to the receive operation. Thus after the following reduction the name y actually refers to z in Q.

$$x!z \rightarrow P || x?y \rightarrow Q \Rightarrow P || Q$$

The binding of data to variables is problematic in function calls. One solution in C would be to pass in pointers to the addresses that should be set to point at received data values. This mechanism would produce the following specification for receiving data over a channel:

```
void ChannelInput(Channel* c, Data** d1, Data** d2);
```

When the function returns, `*d1` and `*d2` will point at the items of data received in the synchronisation with the sending process.

Other languages such as Lisp have no direct support for pointers. As a solution to this a box object could be passed as a parameter to the receive function. The receive function could then use a mutation to set the contents of the box to the value of the received data.

Figure 4.1 shows the definition of a box object, and a function which calls `ChannelInput` to receive two numbers and return their sum.

Another method in Lisp would be to provide a closure to the input function which should be called once the input has synchronised with an output process and the data items have been received. The input function should return the value which the closure computed. This would remove the need to pass box objects as parameters to the input function. The example of a function receiving two values on a channel and returning their sum shown in figure 4.1 could then be expressed as follows:

```
;; Receive two numbers over a channel and return their sum
(define (channel-add c)
  (channel-input c (lambda (a b) (+ a b))))
```

By passing in a closure to the input function, the scope of the variables bound to the received data items is explicitly defined to be that of the scope of the closure itself.

New special forms

Lisp, like most computational languages, evaluates functional arguments before applying the function to its arguments. The expression `(f (+ 1 2))` would cause the application `(f 3)` to be evaluated.

Macros, on the other hand, leave their arguments unevaluated. Thus if `f` were defined to be a macro then it would receive as its argument the expression `'(+ 1 2)`. The expression returned by the macro is then evaluated.

Thus macros can be thought of as source code translators in that they take unevaluated source as their arguments and return a transformed source code expression to be evaluated in their place.

The translation performed by the macros need not occur at runtime. A compiler could invoke macros when it is parsing a source file and compile the expressions returned by the macro for execution at runtime. This allows new special forms to be introduced to the language with no runtime overhead.

For example, the function to input data over a channel could be transformed into a special form which removes the need for the programmer to pass in a closure for evaluation. Instead, the special form would take as arguments the variables which are to receive the data values over the channel, and the expression which should be applied once synchronisation is complete. The example of a function receiving two values over a channel and returning their sum could then be written as follows:

```
;; Receive two numbers over a channel and return their sum
(define (channel-add c)
  (channel-input c (a b) (+ a b)))
```

After macro translation the function would look similar to the previous input function, taking a closure as the argument to process received data.

```
;; After macro translation
(define (channel-add c)
 (internal-channel-input-function c (lambda (a b) (+ a b))))
```

Complete parsing

Parsing, like macros, allows transformation of source code before execution.

Not all languages have support for macros, or have only a limited support. In order to introduce special forms to these languages a new pre-processor could be written which parses the source code and, in a similar fashion to macros, translates new special forms into source code of the base language.

Introducing new special forms using parsing rather than macros has a number of advantages:

- Universal. This technique can be applied to any language, without any special support being needed in the base language;
- Language independent. The parser need not be written in the target language which it is to output. For example **lex** and **yacc** could be used in conjunction with the C language to produce a parser which outputs Lisp;
- Syntax independent. The parser can define a complete language syntax, which may be very different from that of the base language it is targeting.

One disadvantage of the use of a parser over macros is that a parser must be able to parse the complete source code, even if it only translates a small number of new special forms. With macros, these new special forms can be defined using macros, with the main parser used to parse the standard components of the language. This significantly reduces the effort required to extend the computational language.

Interpreter

The least efficient of all the options, interpreting is also the most general.

Like macros and parsing, interpreting allows the introduction of new special forms to a language. However as it is the interpreter which is responsible for executing the program, it can specify how this execution is to be achieved, rather than leave it up to the compiler of the base language. This makes it possible for an interpreter which executes a particular dialect of Lisp to be written in a different dialect of Lisp (for example a Lisp interpreter which provides dynamic scoping may be written in a dialect of Lisp which has support only for lexical scope rules).

Primitives from the host Lisp can be exposed to the interpreted Lisp if they provide the desired behaviour, and can be used to increase the efficiency of common operations (for example list operations).

Due to its simple syntax, it is relatively easy to write an interpreter in Lisp. Figure 4.2 shows the main function used to evaluate a simple Lisp like language.

```
(define (eval e a)
  (if (atom? e)
    (if (number? e)
      e
      (eval-lookup-symbol e a))
    (case (car e)
      ((if) (if (eval (cadr e) a) (eval (caddr e) a) (eval (cad-
ddr e) a)))
      (else (eval-apply e a)))))
```

Figure 4.2: Outline of a simple interpreter

It takes two arguments as its parameters, the expression to evaluate and the environment which represents the context within which the expression should be evaluated.

In this example, the source expression passed to the interpreter is either an atom or a list. These have been pre-parsed by the host Lisp's reader function. For example, a read–eval–print loop which inputs expressions to be evaluated and outputs their results would make use of the host Lisp's reader to input expressions as follows:

```
(define (read-eval-print)
  (print "?")
  (print (eval (read) *top-level-environment*))
  (read-eval-print))
```

Interpreters which execute different inputs (for example byte codes) can be built in a similar fashion.

The system can be extended to continuation passing style, as shown in figure 4.3. The continuation is represented as a closure of one argument, the value to be passed to the continuation. The evaluator then either invokes the continuation (for example providing it with the value of an atom), or extends it (for example, when evaluating an `if` expression, the predicate needs to be evaluated first, with the continuation extended to evaluate the relevant expression depending on the result of the predicate).

This supports multitasking; at any point where the evaluator is about to invoke a continuation, it can store it away (together with the value that should be passed to it), and invoke another process's continuation. Thus the continuation models a process containing both its code and current state.

4.6.2 Multitasking

Although the target environment is a network of workstations with multiple processors available to execute processes of the distributed application, it is possible that the number of processes needed to execute the application will outstrip the number of processors available to execute them. With the ability of processes to

```
(define (evalk k e a)
  (if (atom? e)
    (if (number? e)
      (k e)
      (k (eval-lookup-symbol e a)))
    (case (car e)
      ((if) (evalk (lambda (v) (evalk k (if v (caddr e) (cad-
ddr e)) a))
                   (cadr e) a))
       (else (evalx-apply k e a)))))
```

Figure 4.3: Outline of a simple continuation passing interpreter

specify which processor they should execute on, the problem of multiple processes per node can arise even if other nodes in the network are idle.

Thus the system needs each node to be able to multitask the set of processes it is currently managing.

Most operating systems already have support for multitasking, both inter-process and intra-process. Inter-process tasking is achieved by allowing processes to spawn new processes, typically with separate address spaces. Intra-process multitasking may be achieved using a threads package, with a process having multiple execution threads running within the same address space.

Some dialects of Lisp provide support for multitasking by provision of a threads library ((Padget *et al.* 1993), (Kelsey & Rees 1995)). Where there is no direct support for multitasking, continuations may be used to provide the illusion of cooperative multitasking. A blocked thread may be modelled as a captured continuation. Unblocking the thread is achieved by simply invoking the continuation.

Figure 4.4 shows an implementation of two basic primitives, `create-thread` which is used to spawn new threads and `yield` which is called by a running thread to allow other threads a chance to execute. Finally `start-thread-scheduler` is used to set things going.

4.6.3 Support for mobility

Processes written in the computational language can be migrated between nodes by outputting them over channels. This implies that it must be possible to get some sort of handle representing the current process in order to be able to migrate it, and that the underlying system has to be able to support this migration.

The following sections describe various methods for migrating code around a network.

Source code representation

Languages with a simple syntax, such as Lisp, allow programs to be described in terms of data types of the language. For example the Lisp expression `(+ 1 2)` consists of a list which contains three atoms — two numbers and a symbol. This simple and uniform syntax makes it easy to build Lisp programs which generate Lisp code.

```
;;The list of available threads
(define *ready-threads* '())

;;The schedulers continuation. Called when no more threads to schedule
(define *thread-finished* '())

;;Create a new thread. The argument (a thunk) will be invoked the
;;first time the new thread is scheduled
(define (create-thread thunk)
  (add-ready-thread
    (lambda (v) (thunk) (schedule-next))))

;;Yield the current thread
(define (yield)
   (call/cc (lambda (k)
              (add-ready-thread k)
              (schedule-next))))

;;Add a thread to the thread pool
(define (add-ready-thread thread)
  (set! *ready-threads* (append *ready-threads* (list thread))))

;;Schedule a new thread to be invoked
(define (schedule-next)
   (if (null? *ready-threads*)
      (*thread-finished* '())
      (let ((next (car *ready-threads*)))
        (set! *ready-threads* (cdr *ready-threads*))
        (next '()))))

;;Start the threads scheduler
(define (start-thread-scheduler)
   (call/cc (lambda (k)
              (set! *thread-finished* k)
              (schedule-next))))
```

Figure 4.4: A simple threads package using continuations

The above expression could be built by evaluating the Lisp expression `(list '+ 1 2)`. This could be abstracted into a procedure that builds an expression to add two numbers.

```
(define (generate-add a b) (list '+ a b))
```

Using this same technique more complex expressions can be built, generating source that when evaluated would compute the desired result.

As these expressions are generated in source form it is possible for them to be evaluated on any evaluator which is running the distributed application. Processes would be able to migrate by generating source code to run on the remote node, and sending the source code to that node (via channel communication) for evaluation.

Each evaluator needs to be able to accept a source code expression and execute it. Many dialects of Lisp have the **eval** function which is used for this purpose. It takes as its only argument an expression in source form, and returns the result after having evaluated it.

Some dialects of Lisp choose not to provide **eval** as it muddies the semantics of the language. By using the interpreting option outlined in the previous section, eval can be implemented even if the base language does not support it.

Being able to represent a process in source code form makes it easily transportable between nodes (as it is just a basic Lisp data type). An early implementation of the Tube mobile code system ((Halls 1997)) passes around processes in source code form.

The major disadvantage of this approach is that, whilst it is possible to write Lisp programs which generate other Lisp programs, and because of the simple syntax it is easier than in many other languages, it is still takes more effort than is desirable. In effect the programmer is writing two programs, one to generate the process code, and the actual process code itself.

No transmission of code

If migration of processes in their source code form is discounted, two possible paths remain open for the migration of processes between nodes:

1. RPC interface for migration;
2. Portable representation of code.

For an RPC like interface the code of a process is not migrated between nodes. Instead, all nodes in a distributed application must have access to the code of any process which could migrate to the node during the lifetime of the application.

In addition, migration points must be set at application design time as they have to be explicitly encoded into the process.

Given a system with RPC like semantics, migration may then be performed by issuing a remote procedure call to the node where the process wishes to migrate, passing as arguments any data which the process needs in order to continue functioning on the remote node. The RPC server can then serve the call by spawning a new thread to continue the process, passing the received parameters as start arguments to the thread. The client can then terminate its thread, as the process has now migrated to the remote node. Figure 4.5 shows an example of a process migrating between nodes. Although this example refers to RPC, it is equally applicable to a system using channels as its means of communication. For example the migrating process could output its argument over channel x, and the receiving process, on a remote node, spawns a new thread on receiving the new process's arguments on channel x.

This approach to migration has the advantage that the cost of migrating a process is small, as no code is passed between nodes. The fact that code is not transmitted between nodes means that no common code representation is required, and so

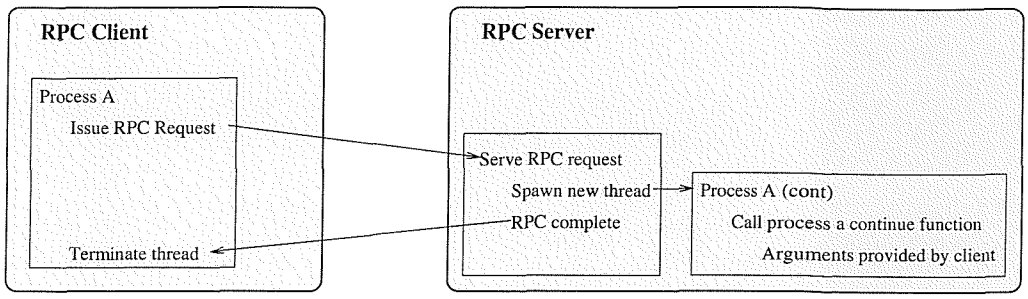

Figure 4.5: Process migration using RPC and static code

code executing on nodes can be fully compiled into architecture specific machine code.

There are also some disadvantages. The system is not easily extensible. In order to introduce a new type of mobile process, the system needs to be stopped, nodes need to be recompiled to include the code for the new process, and the system restarted. If the number of nodes is large, this could cause significant down time. The migration points have to be fixed and explicitly coded for, which does not allow the underlying system to move processes between nodes. Migrating continuations (a representation of a stack frame which describes how a process has progressed during its lifetime) is also problematic. It is difficult to pass stack frames over a communications link, even if the recipient node contains all the relevant code referred to by the continuation. This is because pointers will need to be translated to point at the relevant code on the receiving node, which is difficult to determine at runtime. Thus this solution is only suitable for migration where the continuation is not sent between nodes. Instead the process can be thought of as being stopped on the leaving node, and restarted with a new continuation on the receiving node.

Portable representation of code

The other approach to migration is to migrate both code and data of a process to the remote node. Lisp has a number of ways in which both code and data may be transported to remote nodes:

1. Function pointers. The start address plus arguments used to invoke the process can be sent as distinct data elements;
2. Closures. Data and code are packaged up into one higher level object which may then be transmitted to the remote node;
3. Continuations. The remainder of computation the process has to perform (akin to a call stack in C) may be captured as a higher level object (a function of one argument) by some dialects of lisp. Such objects are called continuations. If this object is then invoked on a remote node, the process continues to execute on that node.

The above list deliberately looks only at the transportation of processes from the user's point of view. Clearly some means of transporting code as well as data

HOC Construct	Lisp construct	Description
$x!a$	`(output` x a`)`	send a over x
$x?a \rightarrow expr$	`(input` x `(`a`)` $expr$`)`	receive on x, bind to a and execute $expr$
$A\|B$	`(alt` A B`)`	Choose either A or B
$A\|\|B$	`(spawn` A`)` `(spawn` B`)`	Execute A and B concurrently
$A;B$	`(`A`)` `(`B`)`	Composition of A and B
$skip$	`(exit-process val)`	Terminate process (returning `val`)

Table 4.3: HOC syntax within Lisp

between nodes is needed. The code must be reified into a data structure suitable for transmission, and the receiving node must be able to turn the data structure back into code for execution. A common approach is to adopt a common code format to represent processes. Nodes in the distributed system then interpret this common code, or compile it further to native code on receipt of the process.

The approach of transmitting code as first class objects allows the programmer to easily package code for transmission over the network.

4.6.4 Scheme: A higher order dynamic programming language

Scheme ((Clinger. & Rees 1991)) is a dialect of Lisp with a relatively small set of primitives and a clean syntax. It has several features which lend it to being a computational language in a distributed environment.

- Ability to represent code as higher order data objects (continuations and closures). This provides a mechanism for capturing processes for migration over a network;
- Simple syntax. The Lisp syntax makes for easy parsing, and the simple introduction of new special forms;
- Dynamic. Scheme functions are not statically typed, rather it is the data that is typed. This allows the creation of generic operations, for example the mapping of a function over a list, or the sending of any type of data over a channel;
- Small. Unlike many other Lisps, Scheme is a relatively small language, making it easier to write interpreters for it.

4.7 HOC Scheme: A language for mobile computing

To aid integration into Scheme, the HOC constructs of table 4.2 need to be given a Lisp-like syntax. Table 4.3 shows the proposed new syntax and their HOC equivalents.

Standard Scheme constructs are used to define processes. The `spawn` primitive takes a thunk (a closure with no arguments) as the representation of a process which it is to start.

4.7.1 Mobile phones

To illustrate using this language an example is taken from Milner's π-calculus tutorial ((Milner 1991)). This example shows many of the constructs shown in table 4.3. It is also a good example of the ability of higher order channels to reconfigure

networks of processes dynamically. It models a simple mobile phone system, with a car containing a mobile phone which changes the base station with which it communicates. The car process communicates with one of a number of bases, with a centre process controlling which base is communicating with the car.

The `car` process communicates with a base through two channels. The `talk` channel allows communication to pass between the car and base, and the `switch` channel allows the base to instruct the car to switch to a new pair of communication channels. The car process described in lisp syntax is given below.

```
(define (car-process talk switch)
   (alt ((input talk) (car-process talk switch))
        ((input switch (talka switcha) (car-process talka switcha)))))
```

A base has four channels. The `talk` and `switch` channels are used to communicate with the car, and the `give` and `alert` channels are used to communicate with the centre. A base can be in one of two modes — either active, in which case it is communicating with the car (and may be instructed by the centre to perform a switch operation), or idle, in which case it awaits a signal from the controller to become active once a switch has been made.

```
(define (active-base talk switch give alert)
  (alt ((output talk) (active-base talk switch give alert))
       ((input give (talka switcha)
          (output switch talka switcha)
          (idle-base talk switch give alert)))))

(define (idle-base talk switch give alert)
  (input alert)
  (active-base talk switch give alert))
```

The `centre` process communicates with bases, controlling which one communicates with the car. A `centre` process using two bases is given below.

```
(define (centre1)
   (output give1 talk2 switch2)
   (output alert2)
   (centre2))

(define (centre2)
   (output give2 talk1 switch1)
   (output alert1)
   (centre1))
```

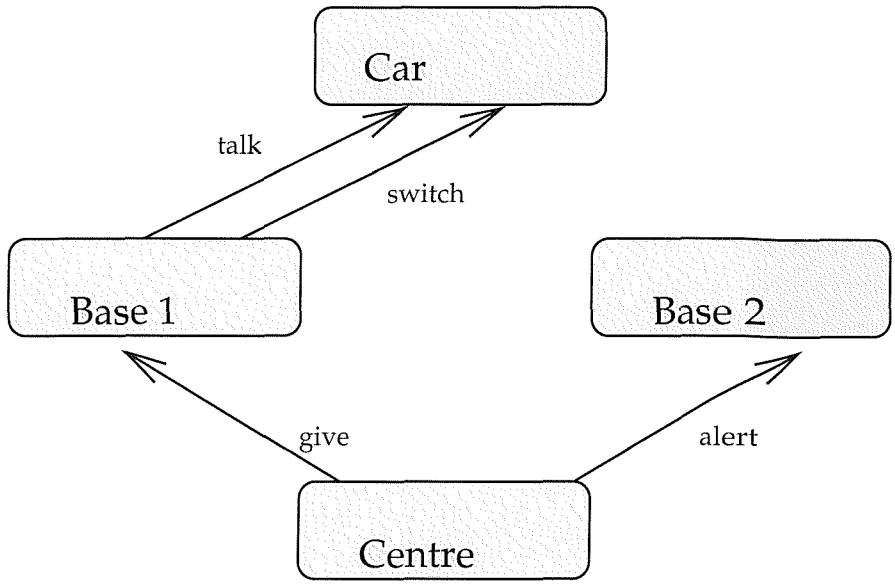

Figure 4.6: Mobile phone network configuration

To start the system, the car process, two bases and a centre need to be running in parallel.

```
(define talk1 (make-channel))
;;and repeat for switch1, give1, alert1, talk2, switch2, give2, alert2

;;Spawn processes
;;car
(spawn (lambda () (car-process talk1 switch1)))
;;base 1 (active)
(spawn (lambda () (active-base talk1 switch1 give1 alert1)))
;;base 2 (idle)
(spawn (lambda () (idle-base   talk2 switch2 give2 alert2)))
;;centre
(spawn centre1)
```

Figure 4.6 shows the configuration of the network when the car is communicating with base one.

4.8 Implementing HOC Scheme

In order to demonstrate the ability of the combination of HOC and Lisp to form a natural language for creating distributed applications, experience must be gained in using the language to ascertain its advantages and disadvantages.

Sample applications need to be written in the new language and compared and contrasted with other implementations for distributed hardware. In addition, the behaviour of such a system needs to be studied in order to discover properties of the language which do not show up in theoretical models. This includes how

efficient the language can be made, and whether any of the constructs described for the language cause implementation difficulties. It is also important to test the ease with which distributed applications can be written, as the purpose of the language is to abstract the distributed system to a level where the developer has a powerful set of tools for creating interacting processes.

In order to perform the experiments it is useful to be able to execute sample applications, preferably over a distributed set of nodes. There are a number of ways of executing such applications.

4.8.1 Stepper

Processes in HOC synchronise on input and output over a common channel. A stepper allows the user to view available synchronisations between all processes running in the system and direct which synchronisations should proceed.

This can be a useful tool in understanding the behaviour of the processes and tracing possible sequences of events that a running system could take.

As the time the user takes to select synchronisations is likely to be longer than the time taken by the system to compute the sets of possible synchronisations, the user becomes the bottleneck in the system. Therefore speed is not the overriding objective when designing and implementing a stepper.

Because of this it is not necessary to provide a distributed stepper. Instead one process providing the illusion of executing multiple HOC Scheme processes will be sufficient to provide the user with an adequate tool.

4.8.2 Simple interpreter

An interpreter capable of directly executing the expressions of HOC Scheme in single address space (which implies a single machine) is not a good candidate for executing a finished application, as it provides none of the benefits that HOC is capable of expressing (multiplatform, distributed, migration). However such a system is a good candidate for testing applications before deployment on a distributed network. Advantages of this approach include:

- Simple debugging. All the application's processes execute in the same address space, which allows the user to inspect the entire state of the system (if the interpreter allows this). Contrast this with a distributed system, where if one process breaks, others can continue to execute, resulting in difficulties when debugging;
- Simple configuration. As no network of distributed nodes is involved, then there is no network configuration to be carried out;
- Simple loading. Again as there is only a single node, loading applications becomes trivial.

4.8.3 Distributed system

When the application has been tested and debugged, the final distributed implementation of HOC Scheme can be used. As well as executing the application, the final system should also be used for a final test of the application, to show up bugs

not found by the stepper or single node (for example bugs that arise from processes being distributed, possible race conditions and bugs in the HOC Scheme implementation).

4.9 Summary

This chapter has presented a number of channel based abstractions which may be used in the design of a distributed system.

Abstractions that treat channels as higher order objects (such as the π-calculus) allow dynamic reconfiguration of networks during the lifetime of a distributed application. Milner shows that a formal model can be built up for such an abstraction.

Extending the channel model to be able to pass other data types over channels allows even greater flexibility in designing distributed applications. The ability to pass processes over channels in HOC allows dynamic reconfiguration of process graphs, with the explicit placement of processes shown in section 4.4.2 demonstrating this.

The ability of processes to be passed over channels has an obvious application in the design of distributed applications, that of producing mobile code. Such systems allow processes to execute on the most "appropriate" node, where appropriate is defined by the application.

By integrating a higher order channels model into a computational language, a powerful system can be envisaged that provides a means for programmers to write computational applications which can take advantage of mobile code, and dynamic reconfiguration of networks.

The following chapters discuss the implementation of a set of tools that the HOC Scheme programmer can utilise when designing, testing, and deploying distributed applications.

Chapter 5

Implementing a stepper

5.1 Introduction

The stepper allows applications writers to design the basic processes of a distributed HOC Scheme application and model the interactions between them. It is not intended to be able to execute the final application; rather it implements a subset of HOC Scheme.

5.2 The language

The first task when designing the stepper is to specify the source language it must execute. This is a modified version of HOC Scheme. Major differences include:

1. Channels are named at creation. This allows the user interface to present a consistent name for the channel, regardless of which variables the channel is bound to;
2. Processes are named at creation. Again this aids the user interface, presenting named processes which may communicate over a channel;
3. Input and Output expressions are named. Once again, this information can be used to aid the user in understanding what point processes that make up the distributed system have reached;
4. Limited environment. As the stepper is not intended to execute full scale applications, its initial top level environment provided to the application programmer need not be as complete as final distributed systems.

The following sections discuss the requirements of the stepper in more detail and present the design of the system, together with possible alternative routes in the design, and reasons for the paths taken.

5.3 User interface

Once the language itself has been specified, the interaction of the user with the stepper needs to be defined. This is the most important aspect of the system, as a poor user interface will not encourage users to go to the effort of writing the application for the stepper in order to investigate properties of their proposed system.

Efficiency, although important, takes second place in priority over the user interface. The main importance of efficiency is to provide the user with a sufficiently quick response to requests they make of the system, and so can be thought of as part of the user interface experience itself.

When designing a user interface, the first step is to consider whether this should be windows or character based.

Windows based interfaces are regarded as being superior to character based ones in that they are more user friendly, and can have a consistent look and feel with other windows based applications running on the system.

Character based interfaces also have advantages. From the user perspective, it is possible to see (and record) a trace of all commands and responses issued to/received from the stepper. With a sufficiently good input tool, command histories can be searched, edited and reissued. This can make the character based interface appropriate for more advanced users, who are the most likely to be working with the stepper. Character based interfaces are also simpler and less time consuming to the implementer than the windows based alternative.

Thus the character based interface was chosen for the user interface of the stepper. This interface has two modes of operation:

1. Read–eval–print loop. This main outer loop allows the user to define channels and processes. It also allows processes to be started;
2. Stepper. This displays a list of all the possible interactions between processes that are currently possible and allows the user to choose which one proceeds.

The stepper mode of operation could either start automatically, as soon as there are two running processes which are able to communicate with each other, or be started manually by a command issued by the user when in the read–eval–print mode. The manual mode has the advantage that the user may create the complete set of processes that are to be run, before starting the stepper mode.

In a similar fashion, the stepper could be stopped midway through the execution of a set of interacting processes, and control returned to the read–eval–print loop. This could be used for modelling external events occurring within the system (for example an additional node coming online and introducing new processes to the set of interacting processes, or an interrupt causing an external event or modifying a global variable). Once again issuing a command in the read–eval–print loop could manually restart the stepper–print loop, allowing it to continue where it left off, possibly with additional possible interactions.

The stepper presents a list of possible interactions between processes to the user. This list needs to be expressive enough for the user to understand each option. This is achieved through the naming of process, channels and synchronisation operations, producing an output for each possible synchronisation of:

- Channel name;
- Output process name;
- Output synchronisation name;

- Input process name;
- Input synchronisation name;
- Data items to be transferred from the output process to the input process.

Once the list of possible interactions between all processes has been displayed, the stepper prompts the user for their choice. A number of possible options are available to the user:

1. Allow one of the interactions to proceed. The user inputs one number corresponding to the interaction to proceed;
2. Step many. By typing in a list of numbers, the user can direct which current interaction is to proceed, and which subsequent ones should proceed;
3. Return control to the read–eval–print loop. This option leaves all executing processes in the blocked state;
4. Quit the stepper and return control to the read–eval–print loop. Rather than leave the processes, this option kills all running processes and returns control to the read–eval–print loop, so that the user may start another set of processes or retry the current set;
5. Dump a trace. This prints a list of all the interactions which have been performed since the stepper started to direct the processes to their current states.

The step many option could be used to return to a known position in subsequent runs (by taking a trace of the steps which have been taken to reach this position). In order for a subsequent run to reach the same position, the stepper must be consistent across runs. The order in which it presents possible interactions must be the same for each run.

5.4 Implementation

The stepper is implemented using the continuation passing evaluator features discussed in section 4.6.1. As has been shown, the continuation can be used to represent process state, as it is an object which represents the remainder of a computation. By scheduling different continuations to run, and blocking running continuations at appropriate times, a multitasking environment of multiple processes can be modelled.

This model maps nicely onto the HOC Scheme process, with one continuation representing each process. There are a number of points at which these processes could be blocked in order to provide the illusion of multiprocessing.

1. When a process is created. If another process is currently running then the newly created process can be placed on the ready queue. This means that new processes are created in a blocked state;
2. When the process needs to synchronise with another using a channel. The process must be blocked until another process is willing to synchronise and the user selects the interaction to proceed;
3. When a process ends. A process waiting on the ready queue may be scheduled to run;

4. For smoother multitasking, processes can be blocked midway through computations (whenever control passes back to the evaluator, for example to invoke or extend a process's continuation). This would result in more frequent rescheduling of processes. As long as the evaluator reschedules processes in a consistent manner, this will not effect the order in which possible interactions are presented to users in multiple runs.

It was decided that the scheduler should only block processes when necessary, when new processes are created or a process is waiting to synchronise using an `input` or `output` clause, a so called "run to completion scheduler". This simplifies the implementation of the scheduler, and is sufficient for executing processes in the stepper, provided that all processes eventually block on an `input` or `output` clause, or terminate.

The example continuation passing evaluator of section 4.6.1 modelled the process as simply a continuation. Although the continuation is needed to model a HOC Scheme process in the stepper's evaluator, it is not sufficient, as additional information needs to be stored along with the process's continuation.

One piece of information is the process name, used by the stepper to communicate to the user which processes are involved in possible interactions. This needs to be associated with the blocked process.

A solution is to transform the evaluator from a continuation passing evaluator into a process passing evaluator. The process object that the evaluator passes around contains the process's continuation, as well as its name and any other state needed to be recorded per process by the stepper. Another useful piece of state would be a process identifier (PID), so that each process running in the stepper could be uniquely identified.

The choice operator (`alt`) results in the one continuation per process model being broken. This is because each branch of the `alt` statement represents a different possible route for the process, with each route having a unique continuation. In order to evaluate an `alt` statement, each branch needs to be partially evaluated to discover the input or output clause each will block on. Note this means that we assume each branch of an `alt` clause does eventually block. Furthermore no branches should alter the state (environment) of the process before blocking, as this would mean that a branch would affect the process even if it was not chosen to proceed, and another branch was taken. Each branch in turn is evaluated until a blocking statement is met, at which point the next branch is evaluated. When all branches have been evaluated the process is blocked and another process can be scheduled to execute. This results in the same process being blocked multiple times, with each block having a different continuation, only one of which may proceed. In order to enforce this, the stepper should remove any alternate continuations when one branch has been selected to proceed by the user.

The stepper consists of number of modules, each of which have data structures associated with them.

5.4.1 *Evaluator*

The evaluator module is responsible for actually evaluating processes and expressions passed to it from the read–eval–print loop. There are five main data structures associated with this module:

1. Processes;
2. Continuations;
3. The top level environment;
4. Process specific environments;
5. Expressions.

In addition, the module provides a public interface, exposing two functions:

- `(eval <process> <expression> <env>)` This evaluates the provided expression in the context of the process and its environment. It is used by the read–eval–print loop to evaluate expressions the user has input, and by special primitives which need fine control of the evaluator;
- `(start-scheduler)` This function invokes each process in turn in the evaluator's ready list. The function returns to the caller when there are no more processes to schedule (all processes have either blocked waiting to communicate over a channel, or have terminated).

The following sections discuss in more detail the data structures used by the evaluator.

5.4.2 *Processes*

The process is a data structure consisting of three main elements:

1. Process identifier;
2. Process name;
3. Process continuation.

The process object is an abstract data type, with a number of functions provided for manipulating process objects.

- `(process-allocate-id)` generates a new unique process id, using the global variable `*global-process-allocator*` to store the next process id to be allocated;
- `(process-make <id> <name> <continuation>)` creates an object to represent the process with a unique combination of id, name and continuation;
- `(process-id <process>)` returns the process id of a process object;
- `(process-name <process)` returns the process name of a process;
- `(process-k <process>)` returns the process continuation of a process.

5.4.3 *Continuations*

The continuation is modelled as a function which takes one argument, the value of the computation that has currently been computed. By applying this value to the continuation, the process is able to proceed.

```
(define (sample-tail-recursion c n)
  (output c n)
  (sample-tail-recursion c (+ n 1)))
```

Figure 5.1: A tail recursive procedure

One way of thinking of a continuation is as of a collection of stack frames. The continuation grows when a process calls on a (non tail recursive) procedure, and shrinks when a procedure returns to the caller, at which time the current continuation is invoked, with its argument being the value computed by the procedure. For tail recursion, it is important not to grow the continuation, as this method is used for infinite recursion. For example the procedure of figure 5.1 is a tail recursive procedure. It calls itself but does not need the value computed by the procedure. In this case, the function call can be replaced with a *goto*-like statement.

Tail recursion can be spotted by the evaluator by noticing that the return value of the recursive call is the return value of the procedure that is currently being executed, i.e. when the procedure is the last expression of the current procedure. In this case the current continuation can be passed to the evaluator to evaluate the recursive call.

Although it would be possible to model the continuation as a list of frames, with each frame being a data structure representing how the computation should proceed when the frame is invoked, the closure is a more natural choice. As discussed in the introduction, a closure encapsulates code with an environment in which this code should be evaluated when the closure is invoked.

By representing each frame in a continuation as a closure, with the most recent frame encapsulating the closures of the previous frame, the complete continuation of a process can be passed around as a single closure.

This method has been chosen to represent continuations of the stepper's process. It has the advantage of being an efficient means of representing the continuation, as it makes use of the host Lisp's closure representation, and therefore uses the host Lisp's invocation mechanism. An object based representation of a continuation using frames would have to have a new invocation mechanism written for it, although it would provide the advantage of allowing stack dumps of processes which could aid debugging.

5.4.4 Top level environment

The top level environment is shared between all processes running in the stepper. It contains procedures and variables which are global to the whole system. New items can be added to the top level environment through the `define` construct available in the read–eval–print loop. Once added to the top level environment, the values become immutable. If mutation were to be allowed, this would give an alternative means of communication, via shared variables, rather than the channels based communication that is intended to be the only means of communication between processes.

The top level environment is similar to local environments of processes, in that its job is to bind names with values. Thus the data structure used to represent the top level environment can be reused to help represent local environments.

The environment is modelled as a set of pairs, containing the variable name and the value of that variable. The value can be any Lisp data type, including closures, channels, lists etc. Similar to the process, an API provides access to the environment.

- `(env-make)` This function creates and returns a new environment object. This function is only called once when the stepper initialises to create the top level environment;
- `(env-add <env> <name> <box>)` This function adds the binding of `<name>` and `<box>` to the supplied environment (see below for the definition of a box). The function returns a Boolean value. 'True' represents success, whereas 'false' means that the name already exists in the environment;
- `(env-find <env> <name>` Called to lookup values in the top level environment, this function returns a box (see below) which contains the value associated with `<name>`, or null if the name does not exist in the environment.

In order to distinguish between a failure to lookup a name, and the value associated with a name being null, a wrapper needs to be placed around the value so that null will only ever be returned from a lookup if the name does not exist in the environment. This is called a box. It has other features that may be used by other areas of the stepper; for example it could be used to pass around an area of mutable memory. The box data structure also has an API:

- `(box-make <value>)` Creates a box containing `<value>`;
- `(box-value <box>)` Returns the value contained within the box;
- `(box-set! <box> <value>)` Mutates the value contained in the box to a new value.

5.4.5 *Process specific environments*

Environments that are local to processes reuse the environment and box data structures provided for the top level environment. The local environment extends the top level environment by providing a frame based lookup. This is used to extend environments when the lexical scope of a function is extended (for example through the use of a `let` clause). The API for the local environment consists of three main functions:

1. `(local-env-make <top-level-env>)` Creates and returns a local environment object which contains the top level environment as its first frame;
2. `(local-env-add <local-env> <env>)` Returns a new local environment with the environment `<env>` added to it;
3. `(local-env-find <local-env> <name>)` Searches all frames in the local environment for a binding to `<name>`, returning the box associated with

`name` if it exists. The environments are searched in order, from the most recently added environment to the top level environment which was added when the local environment was created.

5.4.6 *Expressions*

Finally, in order to execute processes the evaluator needs to be given expressions that represent thems. The stepper represents these expressions as basic Lisp structures containing lists, symbols and atoms. The native `read` function takes input from the console and parses it into a standard list. For example the input:

```
(define (add1 x) (+ x 1))
```

would be parsed into:

```
(list 'define (list 'add1 'x) (list '+ 'x 1))
```

5.4.7 *Other structures*

The other main structure used by the evaluator is the list of processes that are ready to execute. This is contained in one global variable `*ready-list*`. Each entry in the list is an invocation — a pair containing the process which is ready to execute and the value that should be passed to the process's continuation when it is rescheduled. Thus the ready queue has two APIs, one for manipulating the ready queue and the other for managing invocations. The invocation API is described below.

- `(make-invocation <process> <value>)` Returns a new invocation object containing the process and value;
- `(invocation-process <invocation>)` Returns the process contained in the invocation;
- `(invocation-value <invocation>)` Returns the value contained in the invocation;
- `(invocation-invoke <invocation>` Invokes the process contained within the invocation with its value.

The ready queue API makes use of the global variable `*ready-list*` without exposing it to the user.

- `(add-ready-process! <invocation>)` Adds an invocation to the tail of the ready list;
- `(get-ready-process)` Removes and returns an invocation from the head of the ready list. If the ready list is empty then null is returned.

Using this API it is trivial to write a scheduler, shown in Figure 5.2.

```
(define (start-scheduler)
  (let ((inv (get-ready-process)))
    (when (inv)
      (invocation-invoke inv)
      (start-scheduler))))
```

Figure 5.2: A round robin scheduler for the stepper

5.4.8 Stepper

The stepper module also has a number of data structures it uses to maintain state between invocations. The main data structure on which the stepper acts is the channel. This has a number of elements:

1. Name. The name given to the channel when it was created;
2. Processes blocked on output;
3. Processes blocked on input.

Once again, an API is used to manipulate these channel objects.

- `(make-channel <name>)` Creates a new channel. This function is exposed to processes running on the stepper by adding it to the top level environment when the stepper initialises;
- `(channel-name <channel>)` returns the name of the channel;
- `(channel-inputs <channel>)` returns a list of processes blocked waiting to input data on the channel;
- `(channel-set-inputs! <channel> <list>)` sets the list of processes blocked on input to the provided list;
- `(channel-outputs <channel>)` returns a list of processes blocked waiting to output data on the channel;
- `(channel-set-outputs! <channel> <list>)` sets the list of processes blocked on output to the provided list.

As well as channels, the stepper module also manipulates a number of other data structures. These include:

- List of possible interactions between processes. The building of this list is discussed below;
- List of pending steps. This list is used when the user inputs a number of steps to be taken. A number is taken off the head of the list each time the stepper is invoked;
- List of completed steps taken since the stepper started. This is used to provide the user with a trace of the steps taken to reach the current state.

The list of possible interactions is built up dynamically as processes block on `input` and `output` clauses. These functions are provided in the top level environment as special types of primitives which take the current process as an argument and do not continue it. Instead they place the process on the relevant blocked

queue of the channel, and return. Control is then passed back to the scheduler and another process on the ready list is rescheduled.

There are two main ways of computing the list of invocations which should be presented to the user when all processes have blocked. One method is to determine all possible invocations when all processes have blocked. The other method dynamically updates the list when evaluating `input` and `output` clauses. There are advantages and disadvantages to each method. The advantage of the static approach is that the same simple algorithm is followed each time the stepper is invoked.

```
clear the stepper interaction list
for each process in the blocked process list
  if there is another process blocked on the same channel and
    if that process is prepared to synchronise with the first then
      add them to the steppers interaction list
```

However this method is less efficient than a more dynamic method as the whole process set has to be searched multiple times. In complexity, this algorithm is order N^2.

The dynamic approach on the other hand, updates the stepper's interaction list as and when processes become blocked. This results in fewer steps being taken to compute the interaction list. However, this approach involves more complex data structures, as the stepper interaction list is never cleared. Instead, when an interaction is selected to be stepped, it must be removed from the list. In addition, any potential interactions on the list involving the processes that have just interacted also need to be removed from the list. This can occur when one branch of an `alt` clause of a process has been selected, since the remaining `alt` clauses become invalid as their paths will not be taken, and so any interactions that they could potentially become involved with also become invalid.

5.5 Future extensions

The features described above allow the user to write applications in a simplified form in order to test properties of their system (mainly interactions between processes) before implementing and deploying the system on a distributed network.

This section describes possible extensions which could be added to the stepper to enhance the number of services available to the user. Any additional functionality made available to the user will enforce the benefits of writing the application prototype to run on the stepper, before going on to the full implementation.

5.5.1 Deadlock detection

Complete deadlock

Complete deadlock occurs within a system when every process is blocked, waiting for an event from another process. For example, the following system defined below will deadlock immediately.

$$x!a \rightarrow y?b || y!c \rightarrow x?d$$

The first process needs to transmit over x before it can receive a value on y, whereas the second process must transmit on y before receiving on x. In a synchronous system like the π-calculus, deadlock occurs because both processes need to send before receiving.

To resolve the deadlock in this particular case, the events of one process should be swapped (it need not matter which process, but it should not be more than one).

Complete deadlock of a system can be easily detected and flagged to the user by the stepper. The stepper needs to check that the following conditions are met before flagging there is deadlock.

1. There are no possible interactions between processes. If there are events which can be stepped, then the system can proceed and has not deadlocked. This condition could be met when the system terminates successfully, and so additional checks need to be made;
2. There are no processes on the scheduler's ready queue. Again, the system may proceed if there are processes that are able to execute;
3. Every process in the system is blocked, waiting to input or output data over a channel.

The stepper could perform this check every time control was returned to it from the scheduler. By implication the second case, that there are not processes on the scheduler's ready queue, must hold, otherwise the scheduler would not have returned control. Determining that there are no possible interactions between processes is trivial, as the stepper has to determine all possible interactions to present a list to the user.

The third check is more problematic. However this can be transformed into an easier check. As processes can only block waiting to input or output on channels, it holds that if a process exists and is not on the ready queue and is not executing then it must be blocked on a channel. Thus when the control is passed from the scheduler back to the stepper, all active processes are blocked, waiting to input or output over a channel. The only check needed is to see if there are any blocked processes. If there are, and no communications between these blocked processes is possible then the system is deadlocked.

Partial deadlock

Partial deadlock occurs when a subset of all active processes within the system deadlock amongst each other. Other processes in the system continue to run normally. This could occur in a system which is carrying out multiple tasks in parallel, with each task subdivided into multiple processes. One of these tasks could deadlock, but would not necessarily affect the progress of the other tasks being worked on. The following system contains partial deadlock:

$$P(x,y) = x!a \rightarrow y?b \rightarrow P(x,y)$$
$$Q(x,y) = x?a \rightarrow y!b \rightarrow Q(x,y)$$
$$P(a,b)||P(b,a)||P(c,d)||Q(c,d)$$

The first two processes deadlock, as neither process is able to transmit on a or b. However the system as a whole may continue as the third and fourth processes can always interact with each other, first communicating over channel c and then channel d.

Because the complete system does not come to a halt, partial deadlock is not as easy to spot as a system that has completely deadlocked.

Systems do exist that can examine chains of interactions between processes and detect deadlock within a single chain ((K.M.Chandy & J.Misra 1981)).

Deadlock detection in distributed databases

Hilditch and Thomson describe a solution for distributed detection of deadlock in (Hilditch & Thomson 1993). Each transaction in a system has an associated deadlock detection manager process, which is informed each time its transaction is dependant on another transaction (for example, if transaction A is waiting for a lock currently possessed by transaction B, then A is said to depend on B).

Transactions deadlock detection processes communicate with each other to maintain a distributed dependency graph. Two algorithms are presented to maintain the dependency graph. In the first example deadlock detection managers may receive three messages:

`add-graph` T The deadlock manager's transaction depends on another transaction T, which should be added to its dependency graph. The deadlock manager also probes T;

`sub-graph` T The deadlock manager's transaction no longer depends on transaction T, and so it should be removed from the dependency graph;

`probe` T P A probe informs a deadlock manager that a transaction T depends on it. The current transaction selected for abortion if deadlock is found is also sent (P). If the deadlock manager finds T in its dependency graph then a cycle has been found and P is aborted. Otherwise the probe is propagated to all deadlock managers whose transactions are dependent on the probed deadlock manager, allowing cyclic chains to be found.

Figure 5.3 shows this system modelled using HOC Scheme. With suitable conversion the algorithm can be tested in the stepper. The deadlock detection manager for a transaction takes as its parameter a transaction object. This object contains channels to all the elements of the transaction (for example its `abort` channel), as well as channels to all the deadlock manager's services.

A simple lock manager and transaction process has also been implemented. The complete system allows the user to obtain or release a lock for a transaction. A

```
;;Either accept dependency graph change operation or probe
(define (start-DDM transaction)
  (DDM transaction '()))

(define (DDM transaction conflicts)
  (alt ((add-graph transaction conflicts))
       ((sub-graph transaction conflicts))
       ((probe transaction conflicts))))

;;Probe conflicting transactions DDM, and add to list of conflicting
;;transactions
(define (add-graph transaction conflicts)
  (input (transaction-add-graph transaction)  (conf)
    (output (transaction-probe conf) transaction)
    (DDM transaction (cons conf conflicts))))

;;Remove from list of conflicting transactions
(define (sub-graph transaction conflicts)
  (input (transaction-sub-graph transaction) (conf)
    (DDM transaction (remove conf conflicts))))

;;Perform probe
(define (probe transaction conflicts)
  (input (transaction-probe transaction) (init current-victim)

    ;;Reset victim to DDM's transaction if it is younger
    (if (younger? transaction current-victim)
      (set! current-victim transaction))

    ;;If in our conflict list then deadlock so abort victim
    ;;Otherwise propagate probe
    (if (member init conflicts)
      (output (transaction-abort current-victim))
        (map (lambda (t)
               (output (transaction-probe t) init current-
victim)) conflicts))
    (DDM transaction conflicts)))
```

Figure 5.3: Modelling a deadlock detection manager in HOC Scheme

deadlock manager will abort a transaction process when cyclic dependencies are found. Aborting a transaction results in the transaction immediately releasing all locks it possesses and releasing all locks it is waiting for as they become available. Only when it has no locks does the transaction process terminate.

The paper suggests that it is essential that asynchronous message passing should be used as the communication primitive. Though this would be possible to model in the π-calculus (by defining processes to behave as communication buffers) it was found to be unnecessary. The only case where asynchronous communication is needed is where a deadlock manager needs to communicate with itself. This can only happen if a transaction tries to acquire a lock it already possesses, in which case a deadlock manager would attempt to probe itself. This would also break the

two phase locking protocol that transactions should adhere to, and so should not occur amongst conforming transactions.

The second algorithm records all probes to a deadlock manager and introduces an `anti-probe` message, allowing deadlock managers to retract probes when dependencies have been resolved.

By recording which transactions have probed a deadlock manager, cycles in the dependency graph can be found when processing `add-graph` messages. The algorithm also uses an algorithm by Obermark to reduce the number of messages by only sending probes and anti-probes to transactions which are younger than the sending transaction. Figure 5.4 shows an implementation in HOC Scheme of the improved deadlock detection algorithm.

Again a set of transaction processes and a simple lock manager run in parallel with the deadlock detection processes to simulate a transaction system as in the first algorithm.

5.5.2 *Dead processes*

Another form of deadlock, dead processes occur when a process is blocked, waiting to input/output over a channel, and no other process will synchronise with it. This will result in the process never being unblocked and therefore never proceeding.

One method to spot dead processes is to examine the lexical environments of all processes to see if they have bindings to the channel that the potential blocked process is waiting on. If no other process has access to the channel in its environment, then it follows that no other process will be able to perform the corresponding input/output on the channel, and so the waiting process will never unblock. The process can then be flagged as dead to the user.

However, this method will not spot all dead processes. Just because a process has access to the dead processes channel in its environment, it does not necessarily follow that the process will communicate using that channel. Further lexical analysis could be performed on the process to try to determine if it will use the channel in the future. Again this analysis may be incomplete, as it could find that the communication will occur if the process takes a particular branch, but at runtime the process chooses an alternative path of execution.

5.5.3 *Automatic runs*

Rather than a formal verification and trace of all possible sequences of interactions between processes, automatic runs could be initiated by the user to perform random steps and thus produce a random trace through the system. A limit on the number of steps to be made per run could be set, and the system could carry out repeated runs, with each producing a different random trace. By combining the automatic run with the features described above, the system could be used to discover paths which lead to deadlock and dead processes.

5.5.4 *Profiling*

By gathering statistics about what interactions occur most frequently between processes, the application designer can optimise the system for running in a distributed environment.

For example, the statistics could show that two processes frequently communicate with each other through a common channel. In this case it would be most efficient to co-locate the processes on the same node, as this would cut down on the number of messages flowing between nodes. If co-locating the two processes is inappropriate, for example each process is servicing a separate user and needs to be located on their user's workstation, then consideration should be given to changing the algorithm so the two processes do not communicate as frequently, which would again reduce the number of messages flowing between nodes.

5.6 Summary

The stepper as implemented is a useful tool available to application developers using HOC-Scheme to prototype their applications in order to investigate the communications and interactions between processes.

Further enhancements could be made to the stepper to provide a better set of services to the application developer.

```
(define (start-DDM transaction)
  (DDM transaction '() '()))

(define (DDM transaction conflicts probes)
  (alt ((add-graph  transaction conflicts probes))
       ((sub-graph  transaction conflicts probes))
       ((probe      transaction conflicts probes))
       ((anti-probe transaction conflicts probes))))

(define (add-graph transaction conflicts probes)
  (input (transaction-add-graph transaction)  (conf)
    (if (member conf probes)
      ;;If probed then abort
      (output (transaction-abort conf))
      ;;Probe conflict if we are younger
      ;;Probe conflict for all our probes that are younger
     (progn
       (if (younger transaction conf)
         (output (transaction-probe conf) transaction))
         (map (lambda (p) (if (younger p conf)
                (output (transaction-probe conf) p) '()))
           probes)))
  (DDM transaction (cons conf conflicts) probes)))

(define (sub-graph transaction conflicts probes)
  (input (transaction-sub-graph transaction) (conf)
    ;;send anti-probes
    (if (younger transaction conf)
      (output (transaction-antiprobe conf) transaction))
    (DDM transaction (remove conf conflicts) probes)))

(define (probe transaction conflicts probes)
  (input (transaction-probe transaction) (init)
    (if (member init conflicts)
      (output (transaction-abort init))
        (map (lambda (conf)
               (if (younger init conf)
                 (output (transaction-probe conf) init)))
          conflicts))
    (DDM transaction conflicts (cons init probes))))

(define (anti-probe transaction conflicts probes)
  (input (transaction-antiprobe transaction) (init)
    (map (lambda (conf)
           (if (younger init conf)
             (output (transaction-antiprobe conf) init)))
      conflicts)
    (DDM transaction conflicts (remove init probes))))
```

Figure 5.4: An improved deadlock detection algorithm

Chapter 6

A distributed implementation of HOC Scheme

6.1 Introduction

A distributed implementation of HOC Scheme allows the users and programmers of applications to gain experience of how applications behave in a truly distributed environment.

This chapter introduces an implementation of HOC Scheme which runs on a network of distributed workstations. Each workstation in the HOC Scheme system executes an instance of the HOC Scheme runtime system. In turn, each runtime is responsible for executing the HOC Scheme processes that go to make up the distributed application. The chapter is split into a number of sections:

1. Design of the language. This section describes the language the HOC Scheme nodes execute, extending the basic language structure defined in chapter 4;
2. The user interface. The design of the user interface can affect the design of the underlying system and so needs to be considered at the outset of the implementation work;
3. Achieving distribution. This section discusses additional features that HOC Scheme needs in order to provide support for the distribution of processes. In particular, the bootstrap case needs to be considered;
4. Implementation. Finally, when all other issues have been settled, the implementation is described;
5. Future work. This section discusses possible extensions which could be made to improve the HOC Scheme environment.

6.2 Aside — the single node

The single node system is intended to allow application developers to run a complete HOC Scheme implementation on just one node. This should allow the majority of application debugging to be achieved within a simple non-distributed environment.

Once debugged, the application can be tested on a distributed system using multiple nodes.

The single node need not possess every feature required by a node that operates in a distributed system (for example, network connectivity is not required by a single node system). However, the converse is not the case, and it is true to say that a node running as part of a distributed system should possess all the features of a stand alone node. Namely:

- A scheduler with the ability to support multiple processes;
- The ability to allocate channels;
- The ability for processes running on the same node to communicate with each other.

By designing the distributed node with the ability to operate as a single stand alone node, implementation effort can be greatly reduced, as only one system need be written. Given that the distributed node is a superset of the stand alone node, it does not place greater requirements and implementation effort on the distributed system designer.

For this reason, a distributed HOC Scheme node may operate as a single stand alone node, and given suitable configuration of the node, execute HOC Scheme applications which have been designed to run over a network of nodes.

6.3 Language

The HOC Scheme nodes execute the language described in section 4.7. Thus it supports the computational features of Scheme bound to the coordination features of HOC.

The top level environment found in all nodes of the HOC Scheme network also contains additional primitives not defined in section 4.7. These new primitives fall into two main camps:

1. Support for distribution. These primitives are provided to aid the application during boot strap and also allow the system to modify its state. Section 6.5 discusses these primitives in detail;
2. Support for the user interface. This provides the application user with a limited toolbox for creating basic user interfaces. Section 6.4 defines these primitives.

6.4 User interface

Similar to the stepper, the user interface could either be character based or windows based. The presence of a number of nodes in the distributed system increases the problems of designing the user interface. Not only does the decision need be made about the type of user interface, but on which node to display the output.

The choice of node on which to output information depends upon the type of application that is being run. For example, if the application is designed to solve a mathematical problem, then it is likely that only one node will act as the user

interface — the workstation at which the operator is sitting, with other nodes in the system used as workers to help achieve speedup. A workflow application on the other hand might require the input from several users in order to proceed. In this case each user's workstation would be running an instance of HOC Scheme, and would therefore be a node in the HOC Scheme network. Each of these nodes would have to be capable of interacting with their user, and each would therefore require some form of user interface.

This implies that the system has to be able to support multiple user interfaces. This leads to a further issue. When a process requires data to be input or output, how does it specify which node in the system should be used to handle this request?

One solution would be to add an additional parameter to each user interface function that specified which node should be used to perform this input or output request. There are a number of disadvantages to this approach:

1. Naming of nodes could be problematic as the node names would have to be known at programming time;
2. A communications protocol between the nodes for the handling of user interface input and output requests would have to be designed. This would be in addition to the protocol design needed to implement the channels based communications between processes running in the HOC Scheme environment.

Another approach is to perform the input or output on the node on which the process making the request is executing. This would mean that no additional parameters would be needed by the input and output primitives. However this method also presents problems.

1. What if the process which wants to perform the input/output routine is not executing on the correct workstation where the operation should be performed?
2. What if the process does not explicitly know (by name) which node should handle the input/output operation?

These two problems can be solved using the coordination features provided in HOC Scheme. If a process is not executing on the correct node then it has two choices:

1. Migrate to the correct node and then perform the input/output operation;
2. Using channels, communicate with a process executing on the required node and request that it perform the input/output operation on its behalf.

Going back to the workflow example described above, a good solution to the input/output problem would be for a user interface process to run on each user's workstation, using channels to interface with other worker processes that need to input requests from/output results to these users. This has the added advantage that a user could change workstation, causing their user interface process to migrate to their new workstation. Because the channels are node independent the

worker processes in the system would not have to be reconfigured to perform input/output operations on the new node. Instead they would carry on communicating with the user's user interface process using the same channels as when the user was located on his/her previous workstation.

In the case of a single user running an application on a HOC Scheme network, the model above also holds. For example, worker processes could be dispatched from the user's workstation to other workstations within the system. When these worker processes have completed their tasks, they could migrate back to the user's workstation to report their results.

Thus using the coordination primitives of HOC Scheme combined with a simple user interface input/output model is sufficient to handle both the single user and multiple user applications.

We can now go back to the choice of a windows or character based interface. The stepper chose the character based interface because it was sufficient for its needs and reduced the complexity of the implementation. For the distributed implementation, the arguments for a windows based interface become stronger. Advantages include:

- Non-blocking. The Lisp reader provided by most implementations blocks the main thread of control whilst waiting for the user to input a complete expression. Windows based interfaces are generally event driven, allowing the system to continue doing useful work whilst the user is typing in data, for example, executing other HOC Scheme processes;
- More intuitive. Whereas the stepper was designed to be used by application designers, it is envisaged that the distributed system could be used by a less proficient class of user. By providing an interface closer to the ones which they are used to working with, acceptance of new applications could be made easier.

If a windows based interface is to be provided, then the completeness of this implementation has to be considered. A complete interface would provide the applications programmer with access to the full range of windows and controls that the underlying windows system has to offer. As most windows systems do not come with a Lisp interface, this would be a significant piece of work, both in designing the API and the actual implementation. As this research is not centred around the provision of user interfaces, this complete approach was discounted.

Another approach is to provide the user with access to a limited toolkit of user interface components, sufficient to build limited user interfaces. This has the advantage of being a more manageably sized project, whilst providing the application programmer with a non-blocking input/output interface.

This approach has been adopted to provide the windows based interface for HOC Scheme. This user interface consists of a number of components.

1. Expression evaluator. This is similar to the character based read–eval–print loop provided by the stepper. It consists of three panes, an input pane where

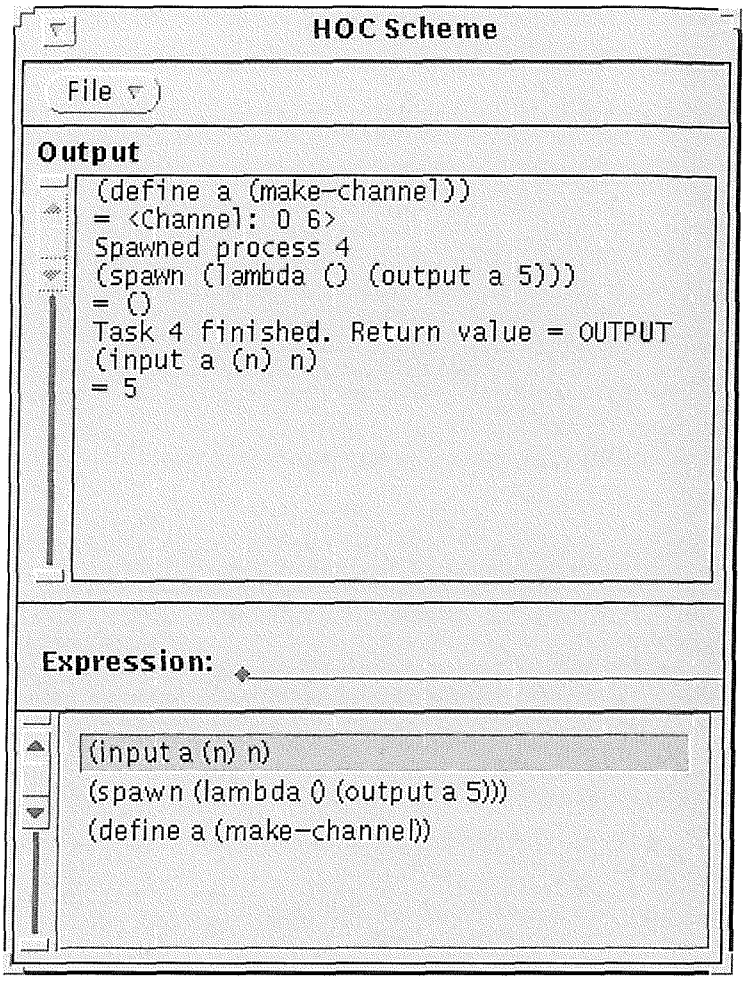

Figure 6.1: Example of the expression evaluator window

users may type expressions to be evaluated, an output pane which displays the history of input expressions and their results, and a list window that allows previously entered expressions to be recalled and edited. Figure 6.1 shows the expression evaluator;

2. List window. This displays the contents of a list, where each element is a text string. The user may then select one of these list items, and this item is the return value of the list pane function. Figure 6.2 shows an example list window;
3. Input/Output window. This allows the application programmer to provide the user with a window, separate from the expression evaluator, in which they can view output and input data for the system. Figure 6.3 shows an example input/output window.

For completeness, a character based interface is also provided by the HOC scheme system. This includes a read–eval–print loop as well as the ability for processes to output and input data from the console. Figure 6.4 shows the output of the mobile phones example running on a single node with character output.

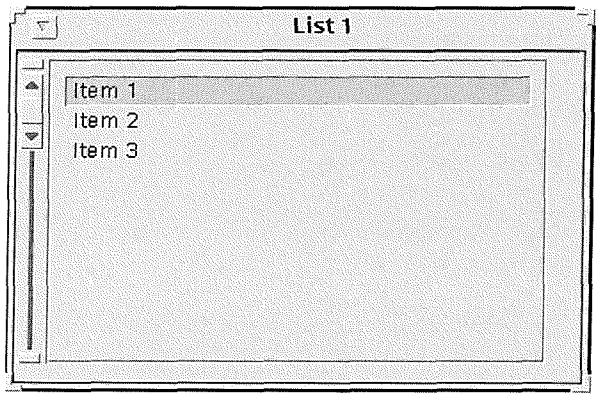

Figure 6.2: Example of the list window

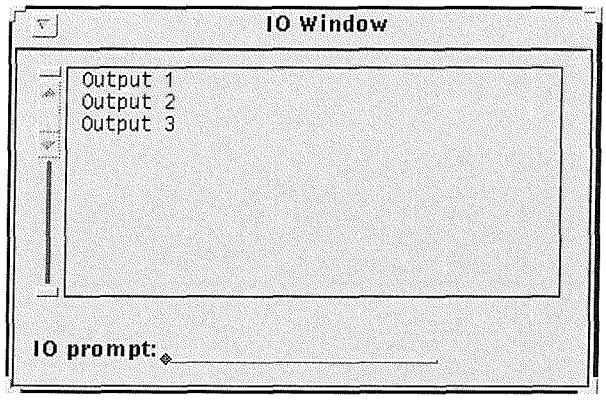

Figure 6.3: Example of the input/output window

Thus the applications programmer has the choice over the type of interface to provide to the user. The applications programmer could even mix interfaces, perhaps with workstations used by general users providing a windows based interface, and nodes used by system operators using a character based one.

6.5 Distribution

6.5.1 Starting HOC Scheme nodes

When a HOC Scheme node is started it may not be alone. Other HOC Scheme nodes could be executing on the network. An issue that needs to be addressed is how new nodes join the existing network when they are started.

One solution is for the new node to broadcast a message to all workstations on the network asking whether they are executing HOC Scheme nodes. Workstations that are executing HOC Scheme will listen for such messages and send a confirmation message back to the sender. The new node will thus find out about existing nodes and can then use a *join network* protocol to bind itself in with the other nodes. If no response message is received within a given timeout period, the node can assume that it is alone and start acting as a single station.

```
HOCScheme? (load "mobile.scm")
= OK
HOCScheme? (start)
Spawned process 3
Spawned process 4
Spawned process 5
Type s to switch to base 2
s
Give
Alert
Switch
Type s to switch to base 1
s
Give
Alert
Switch
Type s to switch to base 2
```

Figure 6.4: Mobile phones running under HOC Scheme

This has advantages and disadvantages. The main advantage is simplicity for the user. All the user has to do is start the node on their workstation, and the node will do its best to join an existing network.

One disadvantage arises if the HOC Scheme network needs to be run over the wide area. Broadcasts typically do not leave the site on which the broadcast was made. This protects the wide area network from congestion due to broadcast traffic (as wide area networks are usually much slower than local area ones, they are less suited to carrying the large amount of broadcast traffic that a local area network typically carries). Therefore a different mechanism will be needed for connecting HOC Scheme nodes running on different sites through a wide area network.

Another problem with the broadcast model comes when multiple networks of HOC Scheme nodes need to be built, with no interaction between these networks. As an example, this could be required for operating a live network, on which users perform real work, in parallel with a test network, on which new applications are developed.

This separation could be achieved by using separate physical networks, or separate subnets through which broadcast traffic does not pass. However, requiring a physical change in the network infrastructure to solve what is essentially a software problem is impractical.

An alternative solution is to name an existing HOC Scheme node in a network when starting a new node which should join that network. This solves the problems that the broadcast model introduces. HOC Scheme networks that span different sites using a wide area network can be easily built, by naming nodes which reside in different site over a WAN connection.

Multiple networks of HOC Scheme nodes can also be created over one physical network. Two seed nodes could be started without naming any other node in the network. This would cause both nodes to create their own HOC Scheme networks.

Additional nodes could then be started, and by naming either seed node, the new node could be directed to join the intended network. New networks could be created at any time without affecting the separate HOC Scheme networks that are already running.

This does pose problems however. It is not as trivial for a user to join a new node to a HOC Scheme network as the broadcast solution, because the user is required to have knowledge of an existing node in the network. In addition, an ordering problem arises, in that nodes that are to be named as existing nodes in the network must be started before the nodes that name them. This problem could be exacerbated when creating a network which runs over a wide area. One solution would be to designate one node that should be named by all other nodes joining the network, and to ensure that this one node is operational for most of the time.

The naming scheme for introducing new nodes to the network was chosen over the broadcast mechanism as it is independent of the underlying network.

6.5.2 Stopping HOC Scheme nodes

By allowing HOC Scheme nodes to be stopped as well as started, the network of nodes becomes dynamic, in addition to the dynamic network of processes executing on these nodes. This allows dynamic reconfiguration of the network to bring online additional resources when required, and to reduce the size of the network when it is prudent to do so.

However, the stopping of nodes does present problems. A node can be stopped in two ways:

1. Planned. The user instructs the node to stop and the node is detached from the network in a graceful manner;
2. Unplanned. The node is removed from the network due to a failure.

Handling the failed node correctly is vital to provide a reliable environment ((Birman 1993)). Different methods of handling each type of shutdown need to be planned. For the optimal case, the following conditions should be met:

1. The remaining nodes in the network do not fail due to any node in that network shutting down;
2. Management is taken over by another node of the channels that the closing node contained;
3. Processes that were executing on the closing node are moved to another node in the network;
4. Any communications involving the closing node are retried once the network has reconfigured.

In the case where the node is shut down cleanly, all the above conditions could be met. The node will have time to offload the processes it was executing and channels it was managing. Whilst in this mode, it could fail any messages that were sent to it, causing the remaining nodes to retry once the network was reconfigured.

Failure of a node can occur for many reasons. For example, the HOC Scheme process could be terminated without letting it execute its shutdown routine, or the workstation on which the HOC Scheme node was executing could crash.

In such cases the shutdown routine of the node will not run, and for all the conditions specified in the above list to be met, a robust fault tolerant infrastructure would be needed. This would involve the mirroring of processes, so that when a process changed state, its mirror process would reflect this. Thus if a node crashed, the mirror node would be able to take over, continuing from the same state as the crashed node had reached. Such a scheme is costly, both in terms of communications and implementation effort. Solutions could be achieved by using existing fault tolerant tools. For example, much work has been done in the area of fault tolerant databases. If the state of each process were to be written to such a database, then if the node failed, a backup node could query the database to resurrect the processes. This would also allow processes to be persistent, giving the system the ability to completely shutdown, and be brought back online in the same state at a later time. This was beyond the scope of this thesis however, and a simpler solution was sought.

In order to allow the network to continue in the presence of a failed node, some of the above conditions still need to be met.

1. The remaining nodes in the network do not fail due to any node in that network failing;
2. Management is taken over by another node of the channels that the failing node contained;
3. Any communications involving the failed node are retried once the network has reconfigured.

If these conditions are met then the network should remain fully operational. However, applications running in the HOC Scheme network could fail. This is because processes running on the failed node are not preserved. Thus any process which attempts to communicate with a process on a failed node will also fail. In this case the communicating process will block as there will be nothing to receive the communication.

Moving the management of channels from the failed node can also cause problems. This is because it involves state being maintained for these channels. Again a backup node is needed in order to mirror this state, so that the backup node may take over management if the primary node fails.

One possible solution that does not require a backup node would be to remove the management of channels from these nodes. Instead, a core set of centralised nodes could handle the management of channels. Thus if a node on a user's workstation failed, no state other than the processes running on that node would be lost. However, this just moves the problem into the centre of the network, away from user workstations. Although it could be argued that centralised server workstations are less likely to crash than user workstations, they can still do so, and thus

some mirroring scheme would still be required. In addition, this loads the communications mechanism between user nodes and the centralised channel management nodes. By allowing each user node to manage channels, the communications load becomes more diffuse across all nodes in the HOC Scheme network.

Another type of failure is partial failure. This occurs when a node(s) in the HOC Scheme network becomes temporarily unavailable. This could occur, for example, when the physical network becomes partitioned due to a break in a wire. Nodes on one side of the break will not be able to communicate with nodes on the other side until the break has been fixed. For individual nodes in the network, it is difficult to distinguish between a partial and total failure of a node. In both cases, the node becomes unavailable and stops responding to messages. For this reason it is safer to assume that the node has completely failed. In the case of a network partition, nodes on both sides of the partition will assume that nodes on the other side have failed and continue to operate independently, resulting in the HOC Scheme network becoming partitioned. This presents a problem when the fault is fixed as the networks should now merge back to form the original structure.

As can be seen, failure situations, both complete and partial, pose many problems when designing a distributed system which is intended to have a degree of fault tolerance. The implementation of HOC Scheme that this thesis introduces does not fully solve any of these problems. It does, however, present an architecture that is extensible and could be extended in such a way as to provide a fault tolerant network.

6.5.3 Starting a HOC Scheme application

Many HOC Scheme nodes may be used to execute a single application. Thought needs to be given to how such applications are started.

One method is to start each node by loading the application file (HOC Scheme source code) and a boot file specific to each node which starts the application's processes applicable to that node.

An alternative is to load the application on one node, and start all processes on that node and quickly migrate these processes to the destination node where they are intended to execute.

By explicitly loading the application and boot file on each node, more work is placed on the user. However, this method means that all nodes running the application will have the applications code preloaded and ready to run any processes which migrate to them. Using only one node to load an application reduces the effort required by the user to start HOC Scheme applications running, but places more load on the HOC Scheme runtime as initially only one node in the system will contain the applications code. This means that code of a migrating process must be sent across the network in addition to its environment.

Both methods leave one unresolved issue, that of the linking up of nodes. HOC Scheme enforces the rule that the only method of communication between processes is by sending and receiving data over channels. This implies that in order

```
;;Accept processes over a channel and spawn
(define (entrance c)
  (input c (p)
    (spawn p))
  (entrance c))
```

Figure 6.5: A function to spawn processes received on a channel

to communicate, both processes must have access to the same channel. How is this achieved if processes start on different nodes, in the case of applications being preloaded on each node? In the case where the application is loaded on only one node, how do processes discover and migrate to other nodes in the network to create a distributed system? Some alternative method of distribution needs to be defined to solve this chicken and egg situation.

In the case where the application is loaded on a single node and processes migrate to their destination nodes, a distribution mechanism needs to be provided for all nodes in the system. Such a system could be provided by an entrance process which listens on a channel and receives closures that it subsequently spawns to execute on its node. Figure 6.5 shows the implementation of such a process, implemented in HOC Scheme.

For this system to operate correctly, the channel that each node's entrance process uses needs to be available to the node that has loaded the application, in order that it may communicate with these entrance processes to start the distributed application.

6.5.4 *Top level environment*

The top level environment can be used to help solve some of the problems involved with starting applications across a distributed network.

The top level environment contains the bindings which are common to all procedures and therefore processes within the application. If each node were to maintain its own separate top level environment, it follows that the application would have to be loaded into each environment on every node on which the application was to execute.

By allowing the top level environment to be shared between all nodes in the HOC Scheme network, only one node would be required to load the application. In order to achieve this distributed environment, a distribution mechanism needs to be defined.

This mechanism cannot easily be built on top of the channel communication protocols as it is orthogonal to them. The top level environment must be capable of distributing channels as well as any other bound object.

It is important to enforce the rule that channels are the only means by which processes are able to communicate with each other. For this reason, the top level environment has been made immutable. If bindings in the top level environment

were to be made mutable, this would give processes an alternative means of communication, via the shared variable. Without the provision of mutual exclusion primitives to protect variables being simultaneously written to by multiple processes, and event primitives to signal to a process that a shared variable is ready to be read, then the use of shared variables for communication is risky.

A mutable variable also poses problems concerning distribution, especially if two processes simultaneously change the value of a variable on different nodes, as it is important that all nodes in the network finally settle on one of the two values. If some nodes were to settle on the first value, and others to settle on the second, then the top level environment would become inconsistent.

The same problem arises in an immutable environment. When two nodes simultaneously define the same variable, but with different values, the whole system must agree which definition should be accepted, with the other definition reporting that the name is already contained within the top level environment and is immutable.

6.5.5 *Discovering nodes in HOC Scheme*

A distributed top level environment can be used to distribute channel bindings which allow nodes to migrate processes to other nodes. For example, the first node that starts the network could create a shared channel which enables processes to migrate to a random node. Other nodes would then receive this channel in their copy of the distributed top level environment when they boot up, and could spawn an `entrance` process to listen on this channel. When a process wants to migrate to a random node, it could then transmit itself over this well known channel. It is then up to the underlying channel management system to determine which `entrance` process should rendezvous with the sender, thus determining to which node the process will migrate.

Naming of channels that communicate with `entrance` processes on specific nodes is more complex. Each name in the top level environment must be unique. In addition, if names are not to be hard coded into the application then a mechanism needs to be designed to allow processes to discover nodes.

The resource locator process has been designed to solve this. Again a well known channel is used to access this process to allow other processes to discover nodes that contain a certain set of resources. This resource set is modelled as an unstructured list of symbols, with each symbol representing a resource. There is no restriction on the type of resource a symbol may represent, from the physical, for example `printer` might represent this node has a printer, to the more abstract — `fred` representing the user "fred" is logged onto this machine. Figure 6.6 shows the implementation of a resource locator. For brevity, the mechanics of matching resources is not shown.

An API for accessing the resource locator which hides the interface between processes making the requests and the resource locator server process is included.

```
;;The resource locator process. Can either take input to register new resources
;;or perform a resource location
(define (resource-locator query-channel register-channel database)
   (alt ((input query-channel (required-resources reply-channel)
            (output reply-channel (query-database
                                    database required-resources))
            (resource-locator query-channel register-
channel database)))
        ((input register-channel (resources node)
            (resource-locator query-channel register-channel
              (extend-database database resources node))))))

;;Start one resource locator for the whole network
(define *resource-locator-query-channel* (make-channel))
(define *resource-locator-register-channel* (make-channel))
(spawn (lambda () (resource-locator *resource-locator-query-
channel*
                                    *resource-locator-register-
channel*
                                    (make-empty-database))))

;;API for accessing the resource locator

;;Find a node
(define (find-node resource-list)
  (let ((reply (make-channel)))
    (output *resource-locator-query-channel* resource-list reply)
    (input reply (process-spawner-channel) process-spawner-
channel)))

;;Register a node with a set of resources
(define (register-node resource-list process-spawner-channel)
  (output *resource-locator-register-channel*
    resource-list process-spawner-channel))
```

Figure 6.6: The resource locator process

The communications protocol consists of two operations, querying the database and registering new resources.

The query operation is split into two separate communications. The first is sent from the requesting process to the resource locator server. A list of resources which should be matched is passed through the communication in addition to a channel on which the result should be sent. The request locator server can then perform the lookup and use a second communication to send the result back to the client. The split of a two way communication allows different methods of handling requests. The implementation of the resource locator server shows the query request being handled synchronously. However this need not be the case. For example, the server could spawn a new process to handle the request whilst the server process immediately recurses, ready to receive the next request. If lookups take a long time, this method would help increase the throughput and responsiveness of the resource locator server.

Although the client API does not support it, the two stage communication protocol also allows clients to lookup nodes in an asynchronous fashion. The client can first output the lookup request to the resource locator server, along with its reply channel. Whereas the `find-node` client API then immediately blocks waiting to input the result from the server, clients could actually perform other tasks, only later collecting the result.

Care should be taken when writing asynchronous clients that the system does not deadlock. For example, if the client tried to output two lookup requests to the server and then wait for both results, the synchronous server of figure 6.6 would deadlock. If asynchronous clients were to be combined with asynchronous servers that spawned processes to handle requests, then no deadlock would occur.

There is only one resource locator server in a HOC Scheme network. This process resides on the first node that starts, with other nodes being able to access this process through its well known channel, which is part of the distributed top level environment.

As this server and its API are written purely in HOC Scheme, it is not considered part of the underlying HOC Scheme runtime. Instead it executes as a user process on top of the runtime system, as would any HOC Scheme process an application might spawn. The same is true of the `entrance` process.

The source code for these processes is precompiled into the HOC Scheme runtime. The resource locator is only loaded into the top level environment and the server process spawned if it is the first node to join the network, whereas `entrance` processes are started on each node of the network.

Finally, the issue arises of how to register resources for a node. Nodes may be able to auto discover resources (for example, query the operating system to see if a printer is attached to the workstation). Another solution could be to configure the resources a workstation contains in an initialisation file, for example, `.hocschemerc`. Finally, resources could be specified on the command line when booting a HOC scheme node. This method could be used to specify resources that are dynamic (for example, which user is logged onto the network).

6.6 Implementation

Each HOC Scheme node that makes up a distributed network of nodes consists of the same set of modules. For nodes with the same architecture and operating system, the same binary image is used. Figure 6.7 shows the structure of a HOC Scheme node.

Each module is responsible for the operation of part of the HOC Scheme node:

- Compiler. This module is used to compile HOC Scheme source code into a form which can be executed by the runtime interpreter;
- Interpreter. This module is responsible for executing HOC Scheme processes, both application and system. It maintains the environment for each process and the top level environment shared between all processes;

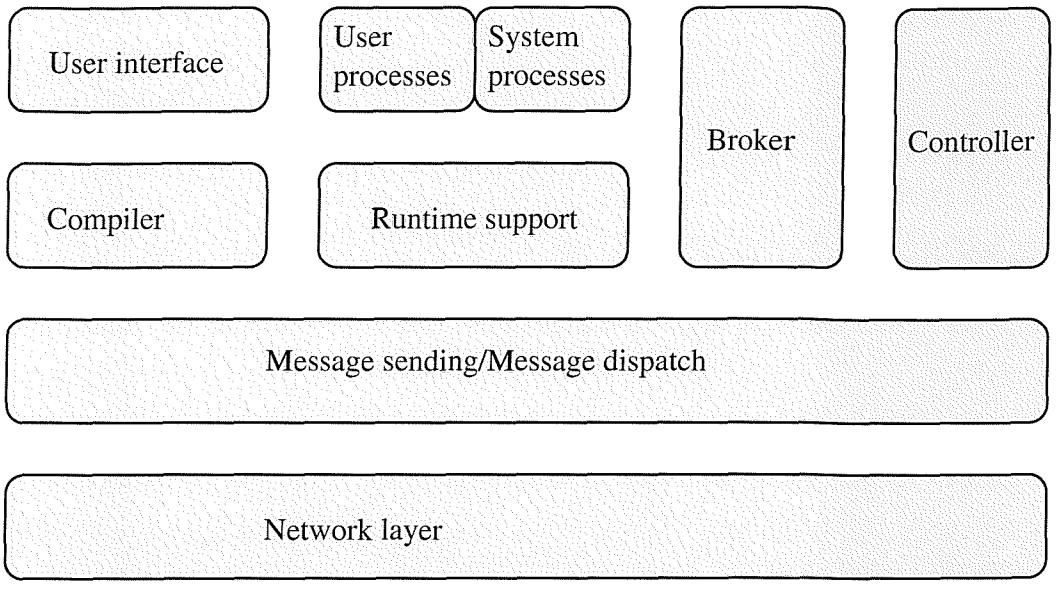

Figure 6.7: Modules contained in a HOC Scheme node

- Scheduler. This is used to maintain the illusion of multitasking of HOC Scheme processes. It is responsible for maintaining a ready queue of processes, and supporting the blocking and resumption of processes;
- Channel manager. This module is responsible for maintaining the state machine used in the protocol for handling channels created on the node. Channels created on other nodes in the network are managed by their nodes channel managers;
- Communications system. This module handles all the inter node message transfers, as well as the delivery of incoming messages to their destination modules;
- Boot module. Responsible for initialisation of the node.

The following sections discuss in greater detail the modules which make up a HOC Scheme node.

6.6.1 Implementation language

Section 6.6.1 discusses the requirements of a runtime system that has to be able to support mobile code. This mobility is achieved through the use of a byte code interpreter, which is capable of executing on a variety of architectures without recompilation.

This section discusses the choice of byte code, the implementation of the interpreter and the underlying platform on which the interpreter and other HOC Scheme modules are implemented.

Several choices could be made when deciding how to implement the system. The first is the choice of language. Although any computer language should be capable of writing the compiler and interpreter, Lisp was quickly settled on as it has a number of advantages over other potential languages.

1. A built in parser. This provides a trivial way of reading in Lisp expressions from an input stream and parsing them into Lisp data structures, which can later be further parsed by the compiler;
2. Built in Lisp data types. These data types can be provided "as is" by the byte code interpreter to the processes that it is executing;
3. Garbage collection. Again, not only can this be used to garbage collect objects discarded by the compiler and interpreter, it can clean up objects that runtime processes no longer require.

Alternatives do exist. For example, the Tube system ((Halls 1997)) uses a compiler producing Java byte code from a Lisp like language. This byte code can then be executed on any standard Java virtual machine, of which there are many implementations.

Once it was decided to use Lisp to implement the compiler and runtime system, an implementation had to be chosen. Because the system interprets HOC Scheme processes, compiled into byte codes, it is desirable that the byte code interpreter should not be interpreted itself. This is because interpretation is slower than the direct execution of binary code by a workstation's CPU. Having the HOC Scheme byte code interpreter itself interpreted by an underlying Lisp interpreter would result in HOC Scheme processes being executed through two levels of interpretation, greatly reducing the execution speed of these processes. For this reason, a Lisp compiler producing native code that could directly execute on workstations was sought.

Compilers which generate architecture specific code have to be ported to new architectures as and when they arise. It is therefore important, if HOC Scheme is to execute on a variety of nodes in a heterogeneous network, that the compiler be able to target many platforms.

Some compilers produce **C** code, which is then further compiled to native code by a workstation's **C** compiler. Such a compiler would appear to be a good candidate for the HOC Scheme system, as the system could be compiled on any platform that contained a C compiler. However, provision of the runtime support needed by applications is often architecture specific, the garbage collector, for example.

The following sections briefly describe compilers suitable for compiling a HOC Scheme node written in Lisp into a set of binary executables that can target multiple platforms.

Scheme->C

Scheme->C ((Bartlett 1989)) is a scheme compiler which generates C as its target language. This makes the compiler portable across platforms. A separate runtime library is linked against the object code to produce the final application.

The system has a number of extensions over standard Scheme which aid the compilation process.

1. Modules. This allows a single application to be split into several files, which aids programmers in producing structured code, and gives them the ability to quickly find the code they require;
2. Foreign language interface. This allows Scheme programs to call functions implemented in another language, that have been compiled into object code which is linked into the final application;
3. Constants. By declaring a name as mapping to a constant value, the compiler need not place this name in the top level environment, and can instead perform substitution of constant names with their values at compile time. This increases runtime performance, as constants will not need to be looked up every time they are referenced;
4. Interpreter. A runtime system can be built which boots into the Scheme->C interpreter, a standard read–eval–print loop. However, in addition to the standard Scheme functions, the top level environment contains all definitions from modules that are linked into the executable. This allows easy unit testing of the distributed system, as individual modules can be tested by calling their top level functions directly from the interpreter;
5. Debugger. The interpreter can also be used to aid debugging of modules. From the interpreter, users can set breakpoints, including conditional breakpoints that execute an expression to determine if they should break. Once in a breakpoint, the user can examine the call stack and state of local variables.

Figure 6.8 shows a sample module and how it is compiled into an executable, and finally that executable running. Note that this example makes extensive use of continuations, showing that the compiler can cope with the most complex Scheme data type. It is left as an exercise for the reader to validate that the output produced is correct!

Bigloo

Similar in function to Scheme->C, Bigloo is a Scheme compiler which generates C source code, which is then further compiled to object code and linked against a runtime library to produce an executable.

One advantage of the Bigloo system over Scheme->C is that it comes with a built in object system, Meroon ((Queinnec 1993)). This extends the Scheme programming language with an object system that allows generic functions to be created (these are functions with multiple bodies, with the actual body chosen based on the classes of arguments passed to the function), and a class based type system with support for single inheritance. For an excellent grounding in Lisp object systems see (Kiczales *et al.* 1991).

Scheme48

Scheme48 does not compile into native code, instead it uses a byte code representation with byte code interpreters able to execute on multiple platforms.

However it does possess many of the features that are required in order to run a HOC Scheme system.

```
nb88r% more mondo.sc

(module mondo (main call-mondo))

(define (call-mondo command-line-args) (mondo-bizarro) (newline))

(define (mondo-bizarro)
  (let ((k (call/cc (lambda (c) c))))
    (display 1)
    (call/cc (lambda (c) (k c)))
    (display 2)
    (call/cc (lambda (c) (k c)))
    (display 3)))

nb88r% scc -o mondo mondo.sc
mondo.sc:
nb88r% mondo
11213
nb88r%
```

Figure 6.8: Mondo bizarro by Eugene Kohlbecker compiled using Scheme->C

1. Functions are able to be reified into a data structure that contains the byte code of the function. This data structure can then be transmitted over a network and reconstructed on a remote node. This would allow the creation of mobile code;
2. The threads interface allows execution of multiple tasks by each individual node;
3. A foreign language interface allows the system to create primitives that map onto functions defined in other languages.

Given this underlying system, primitives could be provided to manage channels and the communication between nodes. The Scheme48 byte code interpreter would then directly execute HOC Scheme processes, calling on a library of functions to provide the additional HOC Scheme functionality that Scheme48 does not natively provide. This solution was discussed in section 4.6.1.

The disadvantage of this approach is the difficulty in providing a distributed top level environment, although other approaches to the distribution of well known channels could be envisaged, for example by communicating with a well known process on another machine.

Commercial Lisps

Commercially available Lisp compilers are more powerful than their public domain equivalents. These compilers typically produce native code directly, choosing not to go via some intermediate language such as **C**. This allows the compiler to have full control over the optimisation process.

These Lisps typically come with powerful debugging and profiling tools, and are a good candidate for producing commercial quality applications.

The choice

Scheme->C was chosen as the compiler used to produce the HOC Scheme node binaries. This implies that the implementation language for writing the HOC Scheme node is Scheme.

This does not mean future versions of HOC Scheme nodes need to be compiled using Scheme->C. Bigloo is also a good candidate for compiling HOC Scheme nodes. Some porting effort would be required, from the extensions to Scheme provided by Scheme->C to equivalent extensions provided by Bigloo.

As long as the communication protocols between nodes are not altered, a different implementation language could be chosen, with these nodes being able to interoperate with HOC Scheme nodes written in Scheme.

Scheme->C was chosen over the other candidates due to a set of modules which HOC Scheme node makes use of being available. These modules are introduced in subsequent sections.

6.6.2 *Byte code interpreter*

Within the HOC Scheme node lies the interpreter module. This module is responsible for executing the HOC Scheme processes. These processes are either standard HOC Scheme processes provided by the HOC Scheme system, (the `entrance` process, for example), or processes spawned by the distributed application the HOC Scheme node is executing. It is also responsible for executing expressions provided by the read–eval–print module, and the windows interface module.

The interpreter does not execute expressions in the form parsed by the Lisp reader receiving expressions to be evaluated. Instead it executes a form of byte code which is compiled from the read expressions.

This byte code has been adapted from a previous project which used Scheme->C as its compiler, the ICSLAS project, run at INRIA, Paris ((Queinnec 1990)).

Part of this project provides a two pass compiler and runtime interpreter. The first pass of the compiler parses the expression, itself parsed into Lisp *s-expression* form, into an intermediate syntax tree structure known as *s-code*. This *s-code* is then further compiled into a form which can be directly executed by the interpreter. This form is known as *i-code*. Both types of code make use of the Meroon object system. For example, all instructions that the HOC Scheme runtime interpreter executes are derived from the base class `Icode`. The instruction that represents "evaluates to an atom" would be represented by an instance of the following class:

```
(define-class RT-Quotation Icode
  ( value )
)
```

This defines a subclass of `Icode` called `RT-Quotation`. Instances of this class contain one field (similar to a non static C++ member variable), called `value`.

The Meroon object system automatically creates a constructor for the class, `make-RT-Quotation` that takes one argument. This creates a new instance of `RT-Quotation` with the `value` field initialised to the value of the argument, and returns this new instance. It also creates functions which manipulate the field. `(RT-Quotation-value <instance>)` returns the value field of the instance, and `(set-RT-Quotation-value! <instance> <new-value>)` assigns `<new-value>` to the value field of the supplied instance.

Once the instruction and the data it encapsulates have been defined, code can be added to the interpreter so that the instruction can be executed. All instructions(e) are executed in the presence of a continuation(e), a lexical environment(r) and a top level global environment(g).

A generic function is defined with these arguments, indicating the expression e is the argument that is specialised by methods added to this generic function. The default operation of the function is to raise an exception stating that a non valid i-code instruction was provided.

```
(define-generic (evaluate (e) r k g)
  (raise-exception (make-Non-Evaluatable-Exception e) #f k g) )
```

A method can then be added to this generic function to evaluate instances of `RT-Quotation`. This instruction is evaluated by passing the value field of the object to the continuation of the expression.

```
(define-method (evaluate (o RT-Quotation) r k g)
  (resume k (RT-Quotation-value o) g) )
```

In addition to classes which represent code, runtime objects are also represented as instances of classes in the Meroon class hierarchy.

A set of classes is used to represent continuations. The continuation hierarchy consists of three classes, an abstract base class and two subclasses. These represent a linked list of frames, with the bottom continuation being a primitive Scheme function that is used to clean up the evaluation, for example to print the value for a read–eval–print loop, or tidy up a completed process.

```
 (define-class Continuation Object
  ( )
  :virtual :immutable )

; ; ; Bottom continuations end a continuation. When such a continuation
; ; ; receives a value, it invokes its finalizer on it. It is only used
; ; ; to build the initial continuation k.init.
(define-class Bottom-Continuation Continuation
  ( finalizer                              ; a function (of the underlying Scheme)
    )
```

```
  :immutable )

;;; Regular continuations hold a frame, the others are linked through others.
(define-class Full-Continuation Continuation
  ( others                                  ;a Continuation
    frame                                   ;a Frame
    )
  :immutable )
```

Frames are used to represent the various ways in which a continuation can be extended. Like continuations, all types of frames are derived from the abstract base class `Frame`. For example when a conditional expression is evaluated, the current continuation is extended with a frame that when invoked, either evaluates the consequent code or alternative code, depending on the value passed to the continuation. The interpreter then goes on to execute the conditional statement. The value computed by executing the conditional statement is the value passed to the extended continuation.

```
(define-class Frame Object
  ( )
  :virtual :immutable )

;;; The continuation of an alternative: (if ? then else)
(define-class If-Frame Frame
  ( consequent                              ;a Icode form
    alternant                               ;a Icode form
    r                                       ;a lexical Environment
    )
  :immutable )
```

The continuation is invoked by the `return` generic function, which uses the top frame of a continuation to choose which method to execute. In addition, the function takes the remainder of the continuation(**k**), the value passed to the continuation(**v**) and the top level environment as arguments(**g**).

```
(define-method (return (fr If-Frame) k v g)
  (evaluate (if v (If-Frame-consequent fr) (If-Frame-alternant fr))
            (If-Frame-r fr)
            k
            g ) )
```

This approach to evaluation differs from that used when writing the HOC Scheme stepper. In the stepper, parsing of *s-expressions* and execution of these expressions are carried out at the same time. The distributed HOC Scheme system

splits it into three distinct sections, parsing *s-expressions* to *s-code*, compiling the *s-code* into *i-code*, and finally executing the *i-code*. This has a number of advantages over direct interpretation used by the stepper.

The *i-code* compiler can perform optimisations statically before the code is executed. For example, rather than searching the environment for a variable contained in the top level or lexical environment, the compiler can statically determine the exact position these variables lie in the environment, resulting in faster environment access times.

It can also produce inlined code, for example when a function which is bound to an immutable top level environment variable is called, it is safe for the compiler to inline this function.

The separation of parsing from execution also allows different types of binaries to be built. Nodes which accept expressions from the user, either via a character based read–eval–print loop or a windows based interface, need to be able to produce *i-code* so that these expressions may be evaluated. However, nodes whose only function is to evaluate code, and who do not read users' expressions, need only have the *i-code* interpreter as part of their binaries. This can result in two different types of nodes — listeners which interact with the system on behalf of a user and feed instructions to the second type of node, evaluators.

The runtime environment also differs from that used by the stepper. The most important difference, in terms of portability, lies in the way continuations are represented. In the stepper, continuations are modelled using the host Lisp's representation of closures. The continuation passed around in the evaluator is a closure which takes one argument, the value passed to the continuation. In fact, the closure may encompass many closures, with outer closures wrapping existing continuations when the continuation needs to be extended.

For example, as demonstrated above, the conditional construct `if` is shown to extend the continuation. In the stepper, a function which evaluates an `if` expression(**expr**) in the presence of a continuation(**k**) and environment(**a**) could be expressed as follows.

```
;;Evaluate an if expression
(define (eval-if k expr a)
  (eval (lambda (v)
          (eval k (if v (if-consequent expr) (if-alternative expr)) a))
        (if-condition expr)
        a))
```

This calls back the evaluator in order to compute the conditional clause of the `if` expression. It passes back to the evaluator an extended continuation which again calls the evaluator to compute the consequent or alternate clause depending

on the result of computing the conditional clause. When the conditional or alternate clause is evaluated, the original continuation is passed, equivalent to popping the `if` frame off the top to the stack.

Although this is an elegant use of a higher order data structure provided by the host Lisp system, it does have drawbacks. As the continuation is always represented as one closure it may be considered an atom. In fact, this one closure may wrap many other closures. Representing the continuation as one closure makes it impossible to inspect all the closures contained within. This can be useful when debugging an application, as the set of frames within a continuation represents the call stack.

By representing continuations as a Meroon object which in turn contains other Meroon objects, the distributed HOC Scheme system allows the inspection of continuations. The inspection process is enhanced by the ability to place *i-code* instructions in the continuation. This provides the back trace not only with the frames of the continuation used to hold runtime data, but also the instructions which caused the continuation to be extended.

A simple method can be added to the `return` generic function to ensure that *i-code* frames within the continuation do not raise exceptions.

```
(define-method (return (fr Icode) k v g)
  (resume k v g) )
```

This method simply resumes the rest of the continuation, which in turn pops off the next frame in the continuation and calls return using the frame as the discriminator. In this way, *i-code* frames act as an identity, they do not affect the runtime execution of a program.

Whether *i-code* frames are added to the continuation depends on a global variable `*trace-computation*`. If this value is set to non nil, *i-code* frames will be included in the continuations. This runtime flag could be replaced with a constant, resulting in debug or non debug binaries being built, one with the flag set, the other with the flag cleared. This would give the Scheme->C compiler greater chances to optimise the runtime interpreter, as runtime checks to see if *i-code* frames should be added to the continuation would not be required. Further enhancements could be made by including within the *i-code* instruction the *s-expression* that caused the instruction to be generated. This could be used to aid the programmer in identifying which lines in the source code are associated with the trace of the continuation.

The stepper's representation of a continuation also makes for difficulties when designing a system for mobile processes. This is because in order to migrate the process to another node, the continuation of the process needs to be transmitted to that node. If the continuation is represented using a host Lisp's closure, then in order to transmit this to another node, the host Lisp must provide primitives to reify the closure into a Lisp data structure which is capable of being transmitted. The

receiving node will require a primitive to transform this received data structure back into a closure. Most host Lisps do not provide this functionality.

By representing continuations using objects within the Meroon class hierarchy, the contents of the objects can be inspected. If the same representation of an object is used on all nodes of a system, objects can be decomposed into separate fields with each field being transmitted separately to the remote node, and the instance being reconstructed by the receiving node. Objects contained within objects are further decomposed into their fields by a recursive process. More information on the structure of message is given in section 6.6.3.

Similarly, Meroon objects represent closures in the distributed HOC Scheme system, again allowing the transmission of higher order data types between nodes running the *i-code* interpreter. The interpreter used by the stepper represents closures in a similar manner to continuations, by using the host Lisp's representation of closures, making it difficult for this interpreter to be used in systems where code must be mobile.

As has been shown, the *i-code* interpreter makes extensive use of the Meroon object system. The class hierarchy is used to represent the syntax tree, byte code and runtime objects such as continuations and closures. Generic functions are used to manipulate these objects. This object-oriented design makes the system easily extensible. For example the introduction of a new special form to the language is achieved with the following steps.

1. Defining an additional class in the syntax tree. This should be subclassed from `Scode`;
2. Defining an additional class in the *i-code* hierarchy. This should be subclassed from `Icode`;
3. Changing the *s-expression* to *s-code* parser to recognise the new special form and generate the corresponding *s-code* objects;
4. Defining a method for `compile2Icode` which takes the new *s-code* objects and generates the correct *i-code*;
5. Defining a method for the `evaluate` generic function, which takes the new instruction as its discriminator. This method can then be used to produce the desired behaviour in the system for this special form.

Existing code is not modified apart from in the *s-expression* parser, where the data structure being parsed is not in Meroon object form. In all other cases, additional code is added to the system without affecting existing code.

This technique has been used to extend the interpreter with new special forms introduced for distributed applications ((Queinnec & DeRoure 1992)).

6.6.3 Connectivity

This section describes how HOC Scheme nodes interact at the lowest level — the messaging layer. Other sections describe the protocols which run over the message layer to provide the connectivity that HOC Scheme applications use (the distributed top level environment, and the implementation of channels).

```
;;Public interface of network module

;;Represents a node identifier.
(define-class Node Object
  (  network-address  )
)

;;Initialise the network module
(define (network-initialise network-address) ...)

;;Returns a Node object that represents this node
(define (network-get-local-node) ...)


;;The base class from which all messages should be derived
(define-class Message Object
  ()
)

;;Send a message to a node
(define (network-send node message) ...)

;;A generic function which is called when a message is received from a node
;;Modules should add methods to this function to handle their appropriate
;;message types
(define-generic (network-receive node (message)))

;;Allow the network to perform house keeping
;;There is no separate thread that receives messages. Therefore call this
;;periodically to receive and dispatch messages
(define (network-schedule) ...)

;;Wait for a message to arrive and then dispatch. Call this when the node
;;has no useful work to do and so blocking the thread is not a problem
(define (network-wait) ...)
```

Figure 6.9: API exposed from the network module

The purpose of the messaging layer is to provide other modules with an interface to a reliable method of sending and receiving messages between nodes in the HOC Scheme network.

Network module interface

Figure 6.9 shows the interface that is exposed to other modules which make up the HOC Scheme node.

As can be seen from this interface, the underlying network transport is hidden from the modules which make use of the message system. Instead, an abstract interface is presented, where only the `Node` object (that represents how to access a remote node) is required in order to send a message. By making the `network-receive` function generic, modules can easily extend the message system to process their module specific messages, without the need to explicitly register this with the network module.

Network module transport

The abstract interface now needs to be turned into a concrete implementation. The first choice that needs to be made is which underlying transport will be used to send and receive messages. The choice of transport is not binding. Given the abstract interface, another implementation can be written if necessary. For example, if HOC Scheme were to be targeted on a set of workstations connected by an ATM network, an implementation of the network module which maps directly onto an ATM transport using virtual circuits might be more efficient than using a transport based on LAN emulation.

The transport chosen for this implementation of HOC Scheme was TCP/IP sockets. This provides the following features:

- Connection oriented interface. Before sending a message to a remote node, a connection to that remote node needs to be initiated. The remote node must accept the incoming connect before the message can be sent;
- Persistent connections. Once a connection is set up between nodes, multiple messages may be sent over the connection. The connection must be explicitly closed in order to tear it down;
- Reliable transport. The TCP/IP protocol ((Wright & Stevens 1995)) guarantees messages are received in the order that they were sent, and that no messages are lost or corrupted. It achieves this using a sliding windows based protocol with checksums used to ensure message integrity.

Other higher level message abstractions could have been chosen (for example PVM). However, although at a low level, socket's implementations are widely available and provide a suitable set of tools for creating the concrete implementation of the network module.

Given the transport, a class can be created that represents a network address contained within a Node object. This contains the IP address and the port number which should be used to connect to the node.

```
(define-class IPNetworkAddress NetworkAddress
  ( ipaddr
    port)
)
```

The only time that the other HOC Scheme modules directly create an `IPNetworkAddress` object is during initialisation. An instance of this class is passed to the `network-initialise` function. This provides the network module with information such as on which interface it should accept connections, and on which port to listen for incoming connections. Both the `ipaddr` and `port` field can be set to 0, indicating respectively that any physical network interface may be used and any available port may be used to listen for incoming connections.

Number nodes	Number connections	
	star network	connected network
1	0	0
2	1	1
3	2	3
4	3	6
8	7	24
16	15	120
n	$n-1$	$\frac{1}{2}n(n-1)$

Table 6.1: TCP/IP connections used in HOC Scheme networks

Connections between nodes

Although the underlying transport mechanism has been chosen, there are a number of methods which could be used to establish and maintain links between nodes.

1. Routed network. Connections are established between nodes that represent the architecture of the underlying physical network. For example if there are two groups of nodes separated by a wide area network, then there will be only one connection over the wide area, between two nodes, one on each side of the WAN. Messages from nodes not directly connected are routed through these gateway nodes;
2. Star network. A type of routed network where all nodes are connected to a central node, which routes messages between nodes not directly connected;
3. Fully connected network. Each node has direct connections to every other node in the HOC Scheme network;
4. One shot connections. Connections are established between nodes when a message needs to be sent. When the message has been sent, the link is closed;
5. Dynamic connections. Connections are established between nodes when a message needs to be sent. These connections can then persist so that subsequent messages sent to the same nodes do not need to re-establish connections.

The routed network option uses less resources in the workstation's TCP/IP stack than the fully connected network option, as the resource usage of a TCP/IP stack directly corresponds to the number of connections being made through the stack. As the number of nodes in the HOC Scheme network increases, so does the difference in the resource usage between the routed network and the fully connected network. For example table 6.1 shows the number of connections made by a fully connected network and routed star network for varying number of nodes. Clearly the fully connected network is not practical if the system is to be able to support a large number of nodes. However, the routed network does place additional overheads on the network module, in that it must have knowledge of the connections in the HOC Scheme Networks, and be capable of routeing messages that are en route to a remote node.

Using one shot connections reduces resource usage, as on average at any given time the number of established connections in the HOC Scheme network will be

small. However, overall network load will increase, as the TCP/IP protocol needs to send several packets over the network in order to establish a connection. This will also result in increased latency when sending messages. The mechanism could be used if UDP were chosen as the underlying network transport. This transport does not establish connections between nodes, and therefore there is no overhead to sending a message. However, this protocol does not guarantee the reliable delivery of messages, and so a mechanism for detecting failures in packet transmission would be required.

The dynamic connection approach is a compromise between the fully connected network and the partially connected routed network. It has the advantage that nodes which need to communicate are directly connected, removing the routeing overhead from the network module. It also is more efficient than the one shot connection case, as once established, links between nodes can persist and more messages can be sent over them. As links can be re-established, the network module, is free to choose when to close dynamic links. This could be when a threshold of a maximum number of links has been reached, or the link has remained idle for a period of time.

The network module for HOC Scheme uses the dynamic connection method as a compromise between the complexities of including a routeing algorithm in the network module, which the underlying network layer is more suited to handling, and reducing the number of persistent connections in order to preserve resources.

Latency vs. efficiency

When implementing the network module consideration needs to be given to the type of network traffic it is likely to produce. In HOC Scheme networks many of the messages flowing between nodes are likely to be small (in the order of a 100 bytes or so). However, occasionally larger messages will need to be transmitted (for example when processes migrate between nodes).

It is common to prepend messages with their size when transmitting them over the network. This allows the receiving node to determine easily how many bytes need to be received in order to decode the message. This fixed format clashes with the default TCP/IP implementation. As the TCP/IP layer does not know the format of the data that is being sent over a connection, it tries to optimise the length of packets it transmits over the network. It does this using the Nagle algorithm. After data has been given to the TCP/IP stack to be sent over a connection, it is queued ready to be sent. However, a packet is not sent when the amount of data is small (where small is determined by the TCP/IP stack). Rather, it waits for a small amount of time to see if the application will send more data over the connection. If this occurs, then both sets of data can be combined into one packet sent over the network.

The Nagle algorithm helps reduce network traffic for streams based applications, where data is constantly being given to the TCP/IP stack for transmission. However this algorithm can adversely affect performance of message based applications. This is because the application knows that a message is complete, passes

it to the TCP/IP stack for transmission, but transmission is delayed whilst waiting to see if more data can be sent over the connection. For message based protocols which require acknowledgements, the latency is doubled as the holding off transmission occurs serially, once at the initiating node, and then at the replying node.

Luckily, the Nagle algorithm can be disabled on a per connection basis. This allows connections that are to be used to send messages between nodes to avoid this bottleneck. If the Nagle algorithm is disabled, the TCP/IP stack sends a packet each time it is given data to send, rather than delaying the sends. In order to prevent too much packet fragmentation, it is important to pass *complete* packets to the TCP/IP stack.

By disabling the Nagle algorithm, latency is reduced at the expense of a possible increase in network load. Sending a message takes a finite amount of time. A number of steps need to be taken.

1. The message needs to be converted into a form suitable for transmission over a byte stream connection;
2. The converted message needs to be given to the TCP/IP stack;
3. The TCP/IP stack needs to convert the data into a packet suitable for transmission over the network. This includes calculating a checksum of the data, creation of a TCP and IP header and creation of a MAC header (header for the underlying network, for example Ethernet or Token Ring);
4. The TCP/IP stack gives the packet to the network driver;
5. The network driver programs the network card to send the packet;
6. The network card sends the packet.

There is a fixed overhead in sending a packet across the network that is introduced by these separate stages, regardless of the size of the packet that is to be transmitted. Given this, it is more efficient to send larger packets than smaller ones. This does not mean that small packets should be padded with zeros to make them larger — this will not increase efficiency as null data is being sent over the network. However, if two small messages are to be sent over the network to the same receiving node, then sending these messages in one packet would be more efficient than sending them in two packets.

This implies that perhaps the Nagle algorithm does have a part to play in message based connections. Turning on the Nagle algorithm would certainly combine two small messages into one packet. But it would also continue to introduce latency when only one message needed to be sent over a link.

The solution is to implement a similar concept to the Nagle algorithm in the network module, and not use the underlying Nagle algorithm provided by the TCP/IP stack. This provides the network module with greater control over when exactly messages are to be sent.

The `network-schedule` function is called periodically to process received messages that have been queued by the TCP/IP stack. The flushing of outbound messages to the TCP/IP stack could also be performed out of this function. This would produce the following algorithm.

- The `network-send` function does not pass messages directly to the TCP/IP stack. Instead it queues them. This queue could either be a single queue of all messages waiting to be sent, or a per node queue;
- When the `network-schedule` function is called, messages to be sent to a node are combined and given to the TCP/IP stack as a single block of data. As the TCP/IP stack is not using the Nagle algorithm, these messages are then immediately sent over the network. This is repeated for each node to which messages need to be sent.

This algorithm introduces some latency to the sending of messages, as they are not transmitted out of the `network-send` function. As long as the `network-schedule` function is called regularly then this latency can be controlled. Care must be taken not to call the `network-schedule` function too often, as this will reduce the chance that multiple messages are queued to be sent to a node, resulting in fewer opportunities to combine messages into a single packet.

The algorithm can be further improved by flushing outbound message queues when the length of the queued messages exceeds a threshold. This is because most physical networks impose a limit on the length of a packet that may be transmitted over the medium. For example, the maximum size of packet able to be transmitted on an Ethernet network is approximately 1.5k, and on Token Ring is configurable up to 16K, with nodes typically set to transmit packets at the maximum size of 4K. Thus if data of a length that exceeds this maximum packet size is given to the TCP/IP stack for transmission, it is split into multiple packets. If it could be detected that enough messages had been queued to fill a packet, these could be given to the TCP/IP stack directly out the `network-send` function, rather than waiting for `network-schedule` to be called.

Managing connections

Sockets represent active connections between nodes. The TCP/IP stack provides an interface where data can be sent and received over these sockets.

A table of active connections which maps `Node` objects to sockets that may be used to communicate with the node needs to be maintained. If there is no match then a new connection to that node needs to be established.

The data contained within the `IPNetworkAddress` is all that is required to establish a new connection. The address and port uniquely identify a remote node. The TCP/IP stack's `connect` primitive can be used to set up the connection, and returns a new socket that may be used to communicate with the remote node. This new connection can then be added to the node to socket mapping table.

Once a new connection is established, the first message sent over the connection contains the `IPNetworkAddress` of the local node. This allows the remote node to enter the new socket into the its remote node socket mapping table. It also allows the remote node to report the address from which messages have been received.

The remote node does not need to send back its `IPNetworkAddress`. This is already known by the local node as it must have used this address object to establish the connection.

Each node has a special socket which listens for connections on the network address specified during initialisation (see section 6.6.3). This is used to receive incoming connections.

When checking to see which sockets are ready for processing, the following algorithm is used:

1. Ask the TCP/IP interface which sockets (both the special listening sockets and all active data sockets contained in the node to socket mapping table) have data ready waiting for processing (using the `select` primitive);
2. For each ready socket, perform the appropriate action on that socket.
   - For listening sockets, accept the connection and receive the `IPNetworkAddress` from the new data socket, and add it to the node to socket table. Then perform the receive data operation on the newly created socket;
   - For data sockets, receive and dispatch messages on the socket until no more data is available on it.

Sockets may be closed by the network module by calling the `closesocket` function in the TCP/IP stack. This can be used to manage the number of active connections. For example, the node to socket table could be ordered with the most frequently used sockets at the top of the table. If the number of connections grows too large, the network module can begin closing connections from the bottom of the table (the least frequently used sockets).

The remote endpoint discovers that a socket is closed by finding that directly after the socket has been flagged as ready, there is no data to receive on the socket. The remote node can then close its socket representing the connection, and remove it from the node to socket table.

There are two ways of managing messages that are to be sent to the local node (i.e. the sending and receiving nodes are one and the same).

1. Use a loopback TCP/IP connection. The initialisation code could create a connection to its node's `IPNetworkAddress`. The loopback case could then be managed in the same way as sending and receiving messages to/from remote nodes;
2. As part of the implementation of `network-send`, notice that the address specified corresponds to the local node, and place the messages in a special loopback queue. This queue could then be processed in the `network-schedule` function, by calling `network-receive` on each message in the queue.

Whilst the use of the TCP/IP connection is elegant, as once initialisation has been performed, loopback messages do not present a special case, it is inefficient.

```
; Sample server code using sockets API
(define s (make-socket))  ; create a new TCP/IP socket
(socket-bind s 1234)      ; bind the socket to port 1234
(socket-listen s 5)       ; listen for connections (max. 5 waiting)
(define new-sock (socket-accept s))  ; accept a new connection

; Sample client code using sockets API
(define s (make-socket))  ; create a socket
(socket-connect s "hostname" 1234)  ; connect to remote host
```

Figure 6.10: Sample code using sockets API

This is because computation is carried out that could have been avoided. The message needs to pass through the node's TCP/IP stack, which involves extra overhead. But more costly, the message needs to be converted into a byte stream for transmission through the TCP/IP stack, and then parsed back into a Lisp data structure upon receipt from the stack.

By using the second method and keeping messages that need to be looped back in a separate queue, the translation of the message to and from a byte stream can be avoided, greatly reducing the cost of sending loopback messages.

Errors on connections are currently not handled by this module. Extensions need to be made in order to signal to interested modules when connections are closed abnormally. The current implementation will notice that the connection is closed, and will then try to reopen it to send subsequent packets. This is acceptable if the remote node has not crashed, although messages may have been lost when the previous link broke.

Interface to sockets library

Many operating systems provide a standard interface to the TCP/IP stack using a **C** API (sockets) and a library to link into applications.

In order for the network module to make use of this API, a foreign language module which exposes the socket's functionality to Scheme->C programs was created.

A `C` program file was used to map from a function style suitable for Scheme programs to the equivalent calls in the sockets library. A Scheme wrapper file that defined the foreign language interface was also created. This provided a simple Scheme API for manipulating sockets. For example, figure 6.10 shows sample code of how a server might wait for connections and a client may connect.

The Scheme socket API allows two basic types of data to be sent and received — integers and strings (**C** strings represented as an array of characters with a null terminator). All other Lisp data types which need to be sent over the network need to be converted into strings and integers.

The function `(send-sexpression <socket> <object>)` converts any type of Scheme `<object>` into a mixture of strings and integers and sends the result over `<socket>`. Conversely, `(recv-sexpression <socket>)` receives a

Expression	Encoding
boolean	0 – false, 1 – true
null	2
number	3 followed by value
character	4 followed by ascii value
string	5 followed by length of string followed by characters of string
symbol	7 followed by symbols string
pair	8 followed by encoded `car`, followed by encoded `cdr`
vector	9 followed by number of elements followed by encoded elements
item in cache	10 followed by cache code

Table 6.2: Encoding of *s-expressions*

```
; ; On the sending side
(define cell (cons 1 2))
(define xfer (cons cell cell))
(eq? (car xfer) (cdr xfer))      ; returns true
(send-sexpression s xfer)

; ; On the receiving side
(define received (recv-sexpression s))
(eq? (car received) (cdr received)) ; returns true
```

Figure 6.11: Preserving eq-ness over sockets

mixtures of integers and strings over `<socket>` and reconstructs the original *s-expression*, which it returns. Table 6.2 show how basic Lisp types are encoded for transmission. Meroon objects are represented as vectors, and so can be transmitted over the network using `send-sexpression`

In order to preserve data structures correctly when they are transmitted between nodes, the *s-expression* senders and receivers preserve the **eq-ness** of the objects they send. This maintains the links between any shared data structures within the *s-expression*. Figure 6.11 shows an example of a structure being sent over a network with its eq-ness preserved.

A hash table of encoded objects is built on the sending node. If an object is found to be in the cache, a *lookup cache* instruction is encoded instead of the object.

The receiving end also builds up a hash table of objects it has received. As it uses the same data as the sending end, these hash tables will be equivalent. If a *lookup cache* instruction is received, the object is looked up in the hash table and a reference to the same object is returned, thus preserving shared structures.

Hash tables are reset after every transmission or receipt of an *s-expression*. Therefore eq-ness is only preserved across individual sends. If the structure in figure 6.11 were transmitted twice, the two received *s-expressions* would not be `eq?`. This stops the hash table from growing too large, and prevents it from referencing objects

which would otherwise be garbage collected, whilst allowing the programmer to transmit shared data structures when required.

6.6.4 Top level environment

Section 6.5.4 introduces some of the requirements that the implementation of the top level environment must satisfy. The choosing of a compiler, interpreter and network module place additional constraints on the top level environment.

The top level environment is accessed by all nodes, be they nodes that are purely used for evaluation of processes, or nodes that also accept and compile expressions from the user. This implies that all nodes must be given access to this top level environment. As processes may access the environment at runtime, this access should be made as efficient as possible, as any delays in accessing the top level environment will result in a slow down in the execution speed of processes.

Given these conditions, it becomes clear that the most efficient means by which nodes can be given access to the top level environment is by each node having a copy of it. Having one copy stored centrally, with other nodes sending and receiving messages to query the environment would have drastic effects on the runtime performance of processes.

However, when updating the top level environment, efficiency is not as crucial. This is because the top level environment is immutable, so processes cannot change it at runtime. The only time it can be changed is when distributed applications are being loaded into HOC Scheme. This is likely to occur far less frequently than access of the environment during execution of processes.

The environment should possess the following features:

- Extensible. It is desirable that additional bindings can be added to the top level environment at runtime. This allows the loading and deployment of new processes and applications in a running system;
- Protected against duplication. Names in an environment should be unique. Thus if the same application is loaded simultaneously by different nodes, this should not result in multiple copies of the same binding occurring in the environment. More importantly, it should not result in different bindings to the same name occurring.

Replicated master model

The design of the top level environment for this implementation of HOC Scheme uses a centralised master copy of the environment which is distributed in its complete form to all nodes in the HOC Scheme network. Additional updates may be sent to the nodes when new bindings are introduced to the environment as additional applications are loaded.

All reads of the top level environment are carried out on a node's local copy. This results in a fast lookup of bindings when processes are executing, where the top level environment is in the critical path.

```
;;Abstract base class for all top level environment
;;related messages
(define-class TLE-Message Message
  ()
  :virtual)

;;Register to receive changes in the top level environment
(define-class (TLE-Request-Receive-Changes TLE-Message)
  ())

;;Fetch a complete copy of the top level environment
(define-class (TLE-Request-Env TLE-Message)
  ())

(define-class (TLE-Response-Env TLE-Message)
  ( bindings ))   ;;list of bindings that make up the top level environment

;;Request to add a set of bindings to the top level environment
(define-class (TLE-Request-Add-Bindings TLE-Message)
  ( bindings ))   ;;list of names to be added to the env

;;Response sent to requesting node
(define-class (TLE-Response-Add-Bindings TLE-Message)
  ( bindings      ;;list of names that should be added to env
    position ))   ;;position of first new binding in vectorised env

;;Indication sent to all other registered nodes
;;when top level environment is extended
(define-class (TLE-Indication-Add-Bindings TLE-Message)
( bindings        ;;list of names that should be added to env
  position ))     ;;position of first new binding in vectorised env

;;Initialise binding
(define-class (TLE-Request-Initialise-Binding TLE-Message)
  ( position      ;;position of the variable in the top level env vector
    contents ))   ;;value of the initialised variable

(define-class (TLE-Response-Initialise-Binding TLE-Message)
  ( position      ;;position of the variable in the top level env vector
    result ))     ;;True if ok to initialise variable, FALSE if already
                  ;;initialised

(define-class (TLE-Indication-Binding-Initialised TLE-Message)
  ( position      ;;position of the variable in the top level env vector
    contents ))   ;;value of the initialised variable)
```

Figure 6.12: Messages that make up the top level environment protocol

Writes are performed by sending *update* messages to the node containing the master copy of the top level environment. The master environment then distributes the updates to all other nodes in the HOC Scheme network.

Nodes communicate with the master top level environment node using a set of messages that together form the top level environment protocol. Figure 6.12 shows the format of these messages.

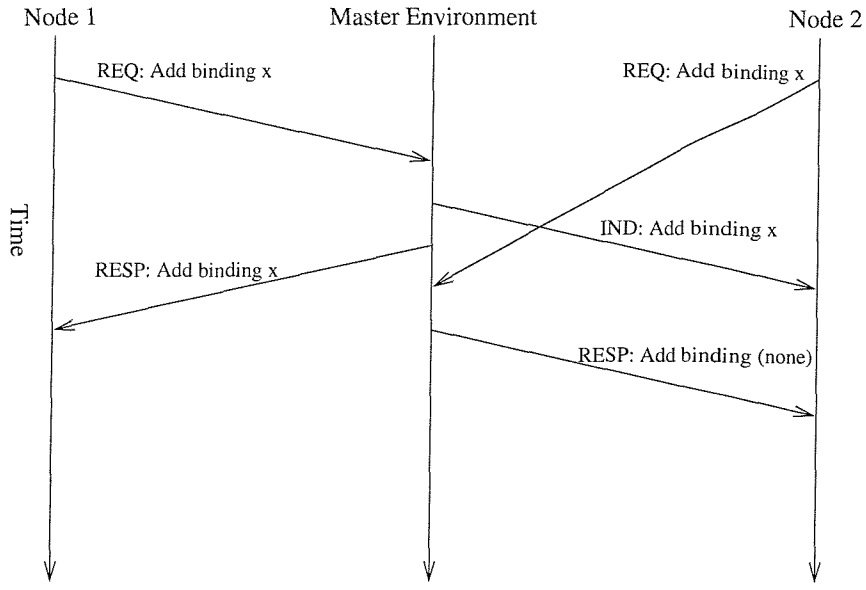

Figure 6.13: Simultaneous update of the top level environment

Implementing the protocol

The top level environment is closely tied to the compiler and runtime interpreter. The parser that translates *s-expressions* to *s-code* takes a global environment handler as an argument. This handler allows the parser to extend dynamically the global environment when it comes across new top level bindings. Once the expression has been parsed, the handler can be queried to find out what new top level bindings are present in the parsed expression.

If the parser discovers that the expression references new top level bindings, a `TLE-Request-Add-Bindings` message is sent to the node holding the master copy of the environment.

Upon receipt of this message, the master node extends its environment with the new bindings. It then sends back a response message to the requesting node, and sends an update indication to all other registered nodes.

By always extending the global environment by going via the master copy, updates are serialised, allowing simultaneous updates to be handled correctly. This ensures that no duplicate names occur in the top level environment, and all nodes agree on the correct position of each binding. Figure 6.13 shows the messages that flow between two nodes and the node holding the master environment when both nodes attempt to extend the environment with the same name simultaneously.

The node that sent the `TLE-Request-Add-Bindings` message waits for the response message from the master node before beginning the compilation stage of the process. At this stage the nodes top level environment has been extended with the new bindings. The *s-code* to *i-code* compiler can then be run with the generated *i-code* containing direct references to the new top level environment bindings.

Finally, the compiled code can be executed. This execution may initialise the new top level bindings. This results in the `TLE-Request-Initialise-Binding` message being sent to the master node. Again, the master node serialises initialisation, failing initialisation if another node initialises it first.

With a cursory glance, it could be claimed that the protocol can be improved by combining the add bindings and initialise binding messages into a single combined stage. However, to compile Lisp expressions correctly, this cannot be the case. Consider the following code.

```
(define (foo) (+ 1 bar))
(define bar 3)
```

The function `foo` is compiled into code before the definition of `bar`. However, `foo` refers to `bar` in its code. Thus in order to correctly compile `foo`, an uninitialised binding to `bar` must exist in the top level environment. This enforces the separation of extending the top level environment from initialising variables.

Initialisation

All nodes in a HOC Scheme network start with the same top level environment. This contains the primitives of HOC Scheme (for example, `cons`, `car`, `cdr`).

Additional changes to the top level environment need to be fetched from the node that holds the master copy. This presents the first issue with initialising the top level environment module. One node in the network has to be flagged as holding the master copy of the environment. Other nodes need to obtain the network address of the master node.

For this implementation, command line parameters are used to specify whether a node is to be master, and if it is not, the address of the master node.

Non master nodes then need to register with the master environment and obtain any changes that have been made to the environment since the system as a whole was started. This is done using two messages. The `TLE-Request-Receive-Changes` message informs the master node that indications should be sent to this node when updates to the environment occur. The `TLE-Request-Env` results in current changes to the environment being sent to the node. It is important to send these messages in the correct order. If the messages were sent the other way around, any changes made to the environment by other nodes between the fetching of current changes and registering for new changes would be lost.

These two messages could be combined into a `register` and `receive changes` message. They are kept separate to allow nodes to register with a new master node, if in future the protocol is extended to allow the master node to change during the lifetime of the system.

Primitives that refer to compiled Scheme->C code are not directly referenced by the environment, as this leads to difficulties in distribution. As with continuations and closures represented using the host Lisp's abstractions, primitives cannot

be converted into a byte stream for transmission by the network module to other nodes.

Instead, each node contains a separate primitive table, which is the same on every node. Instead of including a direct reference to the primitive function in bindings, they refer to indexes into the primitive table. Resolving a binding which represents a primitive therefore takes an additional lookup, slightly decreasing performance. However it does mean that bindings which refer to primitives can be transmitted between nodes, as the offset stored into the binding is easily convertible to a byte stream representation.

6.6.5 Scheduler

The scheduler implemented for distributed HOC Scheme is similar in concept to the scheduler used by the HOC Scheme stepper. This module does not interact directly with other nodes in the HOC Scheme network, and therefore has no message protocol associated with it.

A process is represented by an instance of the class `process` that contains four elements.

1. The process identifier. This is an instance of class `PID` and uniquely identifies a process across all nodes. It contains the `Node` on which the process resides, together with an integer which uniquely identifies the process on its node;
2. The process state. This is either null if the process is ready to run, or a blocking object if the object is blocked on some event;
3. Message id. This is used by the channel protocol used to perform input and output operations;
4. Code. An object that represents the *i-code* which the process will run when it is next scheduled.

The scheduler module contains primitives to create, find, and schedule these process objects. The scheduler itself is a simple loop that runs the process at the head of the ready queue and then repeats. In between scheduling processes to run, it calls a housekeeping function which is used to perform tasks that keep the HOC Scheme node in an up to date state, for example by calling `network-schedule` to process message queues.

When no processes are able to be scheduled, the scheduler module enters a sleep state by calling `network-wait`, and will only restart once a message has been received from the network that could have unblocked a process.

The scheduler also defines the `spawn` primitive which is added to the top level environment at initialisation time. This is shown in figure 6.14.

As can be seen, the spawn primitive takes one argument, the thunk (a function of no arguments) that is to be used as the start function for the new process. It then creates a process structure and sets the code to be an invocation object, which when evaluated will execute the thunk, sending its result to the supplied continuation. This continuation is another object, one which prints the result of the evaluation and tidies up the process object.

```
;;Bottom continuation prints process result and frees process.
(define task.k.init
  (make-Bottom-Continuation
   (make-Primitive-Reference
    (lambda (v g)
      (formatx #t "Task ~a finished. Return value = ~a~%"
      (process-get-id *current-process*) v)
      (process-free *current-process*)))))

;;Spawn schedules the thunk to run as a new process
(define (spawn thunk)
  (let ((process (schedule-create-process)))
    (process-set-code! process
      (make-Invokation thunk '() task.k.init g.init))
    (schedule-process process)
    (formatx #t "Spawned process ~a~%" (process-get-id process))
    'OK))

(defprimitive spawn spawn 1)
```

Figure 6.14: The spawn primitive

A quieter version of `spawn`, called `quiet-spawn` is also included. This performs the same task as `spawn` but does not output any information to the user.

6.6.6 *Channels*

Channels are the only means by which processes running in a HOC Scheme network can communicate. HOC Scheme channels have a number of features which aid the distributed application programmer, as they are higher order, but require effort in order to correctly implement in a distributed system.

What is a channel?

There are differing views of a channel. To the applications programmer, the HOC Scheme channel is a basic data type of the language. The `make-channel` function will construct and return one of these data types.

The programmer may then use this object to achieve communication between two processes. How that communication is achieved does not concern the applications programmer (although they should be aware that using a channel is more costly in terms of time than other HOC Scheme operations).

The programmer can treat the channel as they would any other HOC Scheme object. The channel can be bound to names, either in the local environment of a process, or in the top level environment common to all nodes in the system. The channel can be transmitted over another (or even the same) channel to another process, possibly residing on another node.

To the systems programmer, designing and implementing the HOC Scheme node, a channel is a combination of data structures and messaging protocols which

satisfy the applications programmer's view of a channel. The atomic channel element that the application programmer sees hides the implementation of the channel abstract data type.

In this way, different implementations of channels may be created for differing architectures without the applications programmer needing to change the implementation of their systems.

Channel state

In order to implement channels, some state needs to be maintained for each channel active in the HOC Scheme network.

For example, two processes do not have to perform channel operations at the same time. One channel may perform an input operation and have to wait until another process performs a corresponding output operation. Data pertaining to the input operation will need to be stored until the synchronisation can be completed.

This section discusses the state which is maintained for the implementation of this HOC Scheme node. As important as what state needs to be maintained is the question of where the state should be maintained. There are four main possibilities:

1. State should be maintained on the node where the sending process is executing;
2. State should be maintained on the node where the receiving process is executing;
3. State should be maintained on some other node;
4. Some/All of the above.

One problem with implementing the HOC Scheme channel stems from its dynamic nature. Any process which has access to a channel (as it is bound in the lexical environment of the process, or is in the top level environment) may perform input/output operations using that channel. Thus, when a process performs an operation on a channel, it cannot be statically determined which process will synchronise to complete the operation. Any process that has a binding to the channel could perform the synchronisation. If the channel is bound in the top level environment, this generalises to all processes being able to synchronise with the channel (although it would be possible to reduce this statically by determining which processes accessed the top level binding).

When a process performs an output operation, where does the process send the data? This can only be determined when another process performs an input operation on the same channel.

It is clear that state needs to be maintained on the nodes involved in the communications. At the very least, it should be flagged that the processes have blocked, waiting to synchronise on a channel. The sending process might need to store the data it is to transmit until a receiving process has been found.

Matching communicating processes

Given that two processes wish to communicate over a common channel, the question then arises of how do the two processes "discover" each other.

One solution could be to broadcast to all HOC Scheme nodes information about which channels a node can currently communicate over. Receiving nodes could then check if any matches could be made with their blocked processes, and communications initiated when matches are found.

This solution could work without a large overhead on a local area network that used shared media. A single multicast message could be sent, with all HOC Scheme nodes on that network segment receiving the message.

A network of HOC Scheme workstations which does not use a single shared segment of a LAN will increase traffic. For example, a separate message would be sent to a node that resided on the other side of a WAN. In the worst case, a purely switched network (for example ATM or switched Ethernet), N messages would need to be sent in order to broadcast a single message to all N nodes in the HOC Scheme network.

The broker model

Another solution is to use a third party to match processes that wish to perform an output operation with processes that wish to perform an input operation on the same channel. This broker process resides on a (possibly) different node, fixed for the lifetime of the channel it is managing. Both the input and output processes send a message to the broker indicating that they are prepared to communicate over a channel. By storing this information, the broker process can build up knowledge of what processes are waiting to communicate. When the broker finds a pair of processes that match, the communication protocol used to achieve synchronisation between the processes can be started.

The broker model has been adopted for this implementation. It has the advantage that the model works with the same efficiency regardless of the underlying network architecture.

The number of brokers in the HOC Scheme network directly affects the protocol needed to achieve communications.

If there is only one broker, it holds the complete state about which processes are able to communicate. This allows it to match processes blocked on input with processes blocked on output, and know that the synchronisation will succeed.

Consider the case where a process has several channels on which it may synchronise (as the result of evaluating an `alt` statement). In the case where there is one broker, a branch can be chosen to proceed and the broker can remove all other branches from its data structure. Thus the first chosen branch will proceed and no other branches for that `alt` will be given the opportunity to proceed.

If there are multiple brokers, then a single broker may only have a subset of knowledge of the branches the process may proceed with, with other brokers having knowledge of branches involving channels not managed by the first broker.

Two brokers could simultaneously choose to proceed two separate branches of the same `alt`. The process blocked in the `alt` state would then have to proceed with one of the synchronisations and fail the other. Thus the concept of failing synchronisations needs to be introduced if multiple brokers are present in the HOC Scheme network, increasing the complexity of the synchronisation protocol.

Multiple brokers do however help increase scalability and parallelism in the distributed application. If all channels based communication were to happen through a single broker, then all synchronisations between processes would be serialised. This could become a bottleneck when many processes are communicating over channels at the same time. By having multiple brokers each managing a subset of channels, the communications load is spread more evenly, with parallel synchronisations occurring between channels managed by different brokers. Because of this potential increase in parallelism, the use of multiple brokers has been adopted for HOC Scheme.

The broker model is similar in concept to Mobile IP ((Perkins 1997)). When mobile computers are plugged in different subnets of a common internet, they have to be configured with an internet address suitable for that subnet. Therefore the mobile computers will be given a different address for each subnet in which they are plugged. Although this is fine for client pull applications, where the mobile computer uses the Internet to pull data from servers, for example by browsing the web or getting email from a POP3 mail server, it does not lend itself to server push applications.

For example, an agent could be "launched" into the Internet by a mobile computer plugged in subnet A. The mobile computer could then move to subnet B, being configured with a different internet address. The agent will have difficulty in returning to the computer as its address has changed.

One solution would be for the agent software to inform some centrally located server of its current address, allowing agents to query this database to find if its computer is online, and if it is, the address it should use to contact it.

Mobile IP generalises this to allow mobility without the need for specialised location servers for each type of application. A mobile computer is given a fixed internet address, which actually resolves to a proxy. The proxy is informed of the mobile computer's current address and forwards packets to that address. This allows the mobile computer to advertise a single address from which it may be contacted, regardless of where it is in the network. In a similar manner, channels have a single address, that of their broker, by which any process using a channel may be contacted.

Channel state revisited

The concept of a channel can now be formalised for the multiple broker's model. A channel is an object containing two items:

1. A `Node` object which represents the broker process managing this channel;

2. An `index` object. This references the specific data structure that the broker uses to represent the channel.

Thus a channel object which HOC Scheme applications pass around can be thought of as a remote pointer, pointing at the channel's data structure managed by its broker.

This representation of a channel leads to more issues which need to be resolved. Firstly the issue of memory management.

Distributed garbage collection

A channel may be thought of as a remote pointer to some state held on a broker node. This channel object may be copied between nodes (by being transmitted over channels or bound in the top level environment). Therefore, although a single node can determine whether the channel object should be garbage collected, it cannot determine if the state the channel refers to on the broker's node should be collected. This is because other nodes may still contain a copy of the channel, and thus the channel might still be used for communications. The state for the channel held by the broker can only be discarded when all processes in the HOC Scheme network no longer have access to the channel object. This can be achieved using a distributed garbage collector.

However, the need for a distributed garbage collection can be removed if an appropriate data structure is used by the broker.

Hitherto, the state maintained for a channel by the broker has just been referred to as a data structure, implying the broker always holds some persistent state about the channel. However, this need not be the case. When a channel is not being used for synchronisation, the broker need hold no data about the channel.

The broker only holds data for a channel when a process has performed an input or output operation on that channel. As soon as a match is found, the rendezvous protocol can be started and the information can be removed. If there are no more pending inputs and outputs on the channel, the whole channel structure can be removed. When the channel is subsequently used for synchronisation, the data structure can be recreated.

This means that when the broker holds information on a channel, the channel is in use for an input or output operation, and therefore is referenced by at least one process in the HOC Scheme network. When no operations are being performed on the channel, the broker holds no information about the channel. Thus if all HOC Scheme nodes independently garbage collect all references to the channel, no additional garbage collection is required on the channel's broker node. In this way the need for a distributed garbage collection can be avoided.

Choice of broker

Another issue is to determine a method for choosing which broker should manage a newly created channel. This can affect the efficiency of the synchronisations between processes.

One option is the round robin approach. Channels are shared out equally amongst the brokers, by allocating a new broker for each newly created channel. Scope for parallelism within the channel communication system is increased, as by evenly distributed channel management amongst the brokers, the probability of any two channels being managed by the same broker is reduced. This then increases the chance that any synchronisations involving two different channels will be able to execute in parallel.

Whilst in an idealised environment the round robin strategy of broker allocation may be appropriate, it pays no regard to the usage of channels. The channel communication system uses a message passing protocol, part of which involves communicating with the channel's broker. As sending a message across a network is more costly than sending a message to another subsystem operating within the same HOC Scheme node, the fewer cross node messages that are sent the better.

If the broker for a channel is situated on the same node as the input or output process, then the number of cross node messages needed to perform the synchronisation is reduced, thereby increasing the efficiency of the communication. The optimal case is where the broker, sending process and receiving process are all located on the same node, as this results in no messages leaving the node.

The round robin allocation method could increase the chances of a cross node message needing to be sent in order to communicate with a channel's broker. Another method could be to run a broker on every HOC Scheme node and use the broker on the node on which a channel was created. This increases the chance that at least one communicating process using the channel will be co-located with the channel's broker.

It should be noted that with this method, migration becomes more costly, not only in the cost of migrating the process to another node, but in the cost associated with the process communicating over channels it created on its former node, as messages now need to be sent over the network to the channel's broker. It can also be inefficient to send a channel to processes which then use it to communicate on nodes other than the broker of the channel. However, if a channel is permanently associated with a broker, then these cases are bound to arise in a network that supports mobility and higher order channels.

A larger problem comes with dealing with channels created when defining top level environment bindings. If the above algorithm is followed, top level channels will always be managed on the broker of the compiling node. However, it is unlikely that this is the node which will make use of the channels. For example, the following code defines some channels and a process that acts on these channels.

```
(define a (make-channel))
(define b (make-channel))

(define (flip-flop a b)
  (output a 'flip)
```

```
  (input b (flop))
  (flip-flop b a))

(define (flop-flip a b)
  (input a (flop))
  (output b 'flip)
  (flop-flip b a))

(output *any-evaluator* (lambda () (flip-flop a b)))
(output *any-evaluator* (lambda () (flop-flip a b)))
```

As can be seen, although the compiling evaluator defines the channels, the nodes to which the processes migrate make use of them.

A different initialisation method for channels bound to the top level environment is used. On creation, the channel points at no broker, instead it is set to an uninitialised state. Upon use of such a channel, a node will attempt to bind it to its broker module. A message is sent to every node in the HOC Scheme network informing it of the new state of this channel. Nodes receiving this message can then initialise any copies of the channel contained within their environments. If multiple nodes try to initialise the channel simultaneously, the first message received is used and all subsequent messages ignored.

The initialisation multicast is sent via the master top level environment node for two reasons:

1. The master top level environment has knowledge of every HOC Scheme node in the network as all nodes register with it. Other HOC Scheme nodes may only have partial knowledge and therefore will be unable to perform a multicast to all HOC Scheme nodes;
2. By sending messages via a single node, message order is preserved. If two separate nodes try to initialise the same channel, the master environment node will broadcast first one initialisation message then the other, with all nodes in the system guaranteed to receive the messages in the order that the master environment node sent them. If both nodes were to send an initialisation message direct to every node in the HOC Scheme network simultaneously, nodes could receive the messages in a different order to other nodes, resulting in an inconsistent view of the channel amongst nodes.

Although this initialisation mechanism could be used for all channels created in the HOC Scheme network, be they bound to the top level environment or created within the lexical environment of a process, it could lead to inefficiency.

Top level environment channels are only created at application load time, whereas channels in the lexical environment of a process can be created throughout the lifetime of an application. This can lead to the initialisation process being performed throughout the lifetime of an application rather than for a small period, generally shortly after load time, when top level channels are used. Therefore use of the

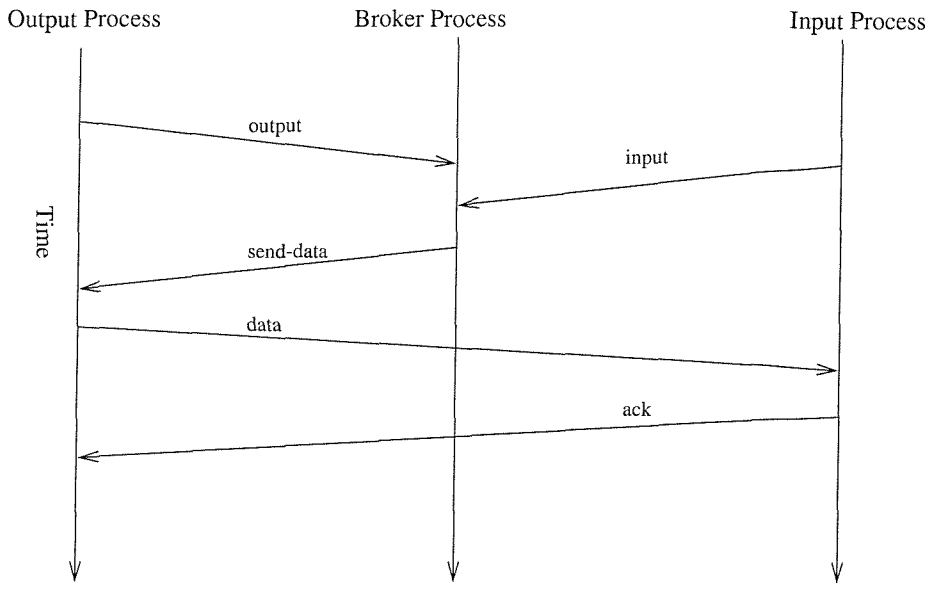

Figure 6.15: Protocol for synchronisation of two channels

initialisation protocol for channels created in the lexical environment for a process will lead to increased network traffic, and additional latency in creating channels.

In addition, channels created in the lexical environment of a process are likely to be used by the creating process, and so allocating the broker of the node on which the process is running without using the initialisation process is a suitable optimisation.

Synchronous communication

The protocol used to achieve communication and synchronisation using the broker model involves message passing between the two communicating processes and the broker.

Figure 6.15 shows the messages that are passed for a successful synchronisation between two processes.

As can be seen, five messages are used to perform the single synchronisation. At the first stage, the broker is informed about the processes wishing to communicate. Once a match has been made, the remaining protocol can be started.

A match is found when two processes wish to communicate over the same channel and the following conditions are met:

1. One process is performing an `output` operation and the other is performing an `input` operation;
2. The sending and receiving processes are communicating the same number of data items;
3. The sending and receiving operations belong to different processes (i.e. they are not different branches of an `alt` operation performed by a single process).

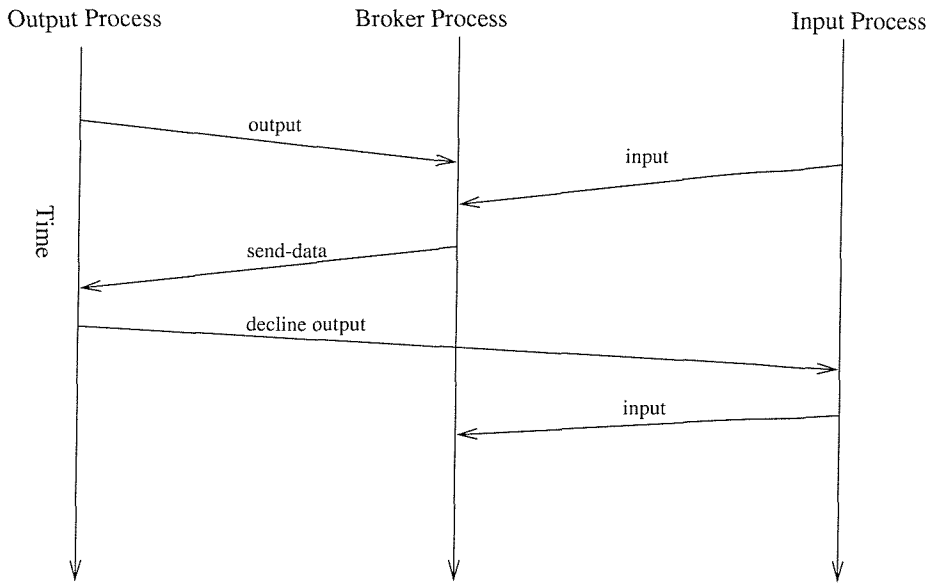

Figure 6.16: Failing an output branch

When the broker finds a suitable pairing of processes, it sends the output process a message instructing it to send its data to the input process. At this stage the output process has a chance of declining the synchronisation (if another branch of an `alt` operation has been taken). It does this by sending a *decline synchronisation* message to the input process. The input process can then re-send its `input` request to the broker to allow it to participate in a subsequent synchronisation. This is shown in figure 6.16.

Assuming the output process can indeed participate in the synchronisation, it sends a message to the input process that includes the data to be passed over the synchronisation. Again the opportunity for the input process to decline the synchronisation needs to be given. The input process declines in a similar manner to the output process, by sending the output process a *decline synchronisation* message, shown in figure 6.17. This allows the output process to re-send its `output` message back to the broker. If the input process accepts the data message sent by the output process, the input process can be unblocked and placed on the scheduler's ready queue. The input process can then reply to the output process, allowing it to be unblocked and proceed.

During the time the output process is waiting for an acknowledgement or rejection message from the input process, messages may be received from other brokers and/or input processes requesting that other branches of the `alt` operation should proceed. These messages should be stored until confirmation or rejection or the currently proceeding branch is confirmed. If confirmed, all stored messages should be rejected, and if rejected, a stored message indicating that another branch should proceed should be acted upon, with remaining messages left in the store.

Finally, when both processes have decided to proceed, any remaining branches can be failed. This involves rejecting outstanding messages and informing other

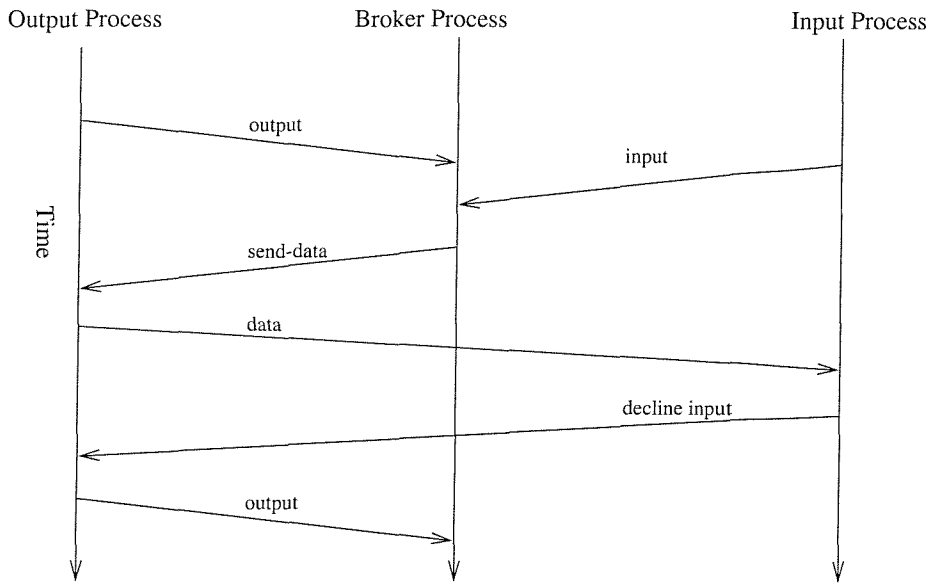

Figure 6.17: Failing an input branch

brokers holding state on the `alt` operations that the branches will no longer proceed, causing the brokers to remove the state from their data structures.

The above protocol gives both the sending and receiving processes the opportunity to fail a synchronisation. Optimisations can be made to this protocol when it is known that either party involved in a synchronisation cannot fail. This is the case where a process performs a synchronisation outside of an `alt` operation. There is only one possible outcome for the process: it synchronises with another process using the channel and then continues. It cannot fail to synchronise. The optimal case is where both processes will not fail a synchronisation. Figure 6.18 shows the protocol used between the processes and the brokers to perform this operation.

When a process performs a channel operation which cannot fail, it flags this to the broker. In the case of an output operation which cannot fail, the data to be communicated through the synchronisation is also sent and stored at the broker. This reduces overall latency of the synchronisation, but does not reduce the number of messages sent. If a large amount of data is to be communicated through a synchronisation (for example, a process) then the data is kept at the sender. On synchronisation, the broker sends a message to the send process instructing it to send its data to the receiving process and to continue to execute. Figure 6.19 shows this protocol.

In the case where one process involved in the synchronisation will not fail, the optimised protocol also uses four messages (when the synchronisation succeeds). The protocol used when the output process is an `alt` branch and the input process will not fail is the same protocol followed for large data transfers, shown in figure 6.19. If the output branch cannot send its data because another branch of the `alt`

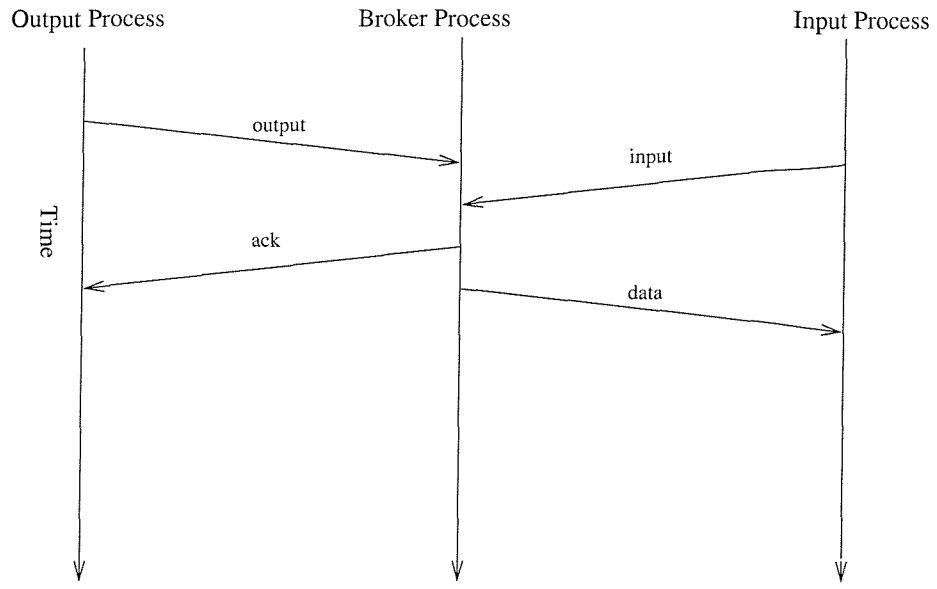

Figure 6.18: Synchronisation protocol with no failure cases

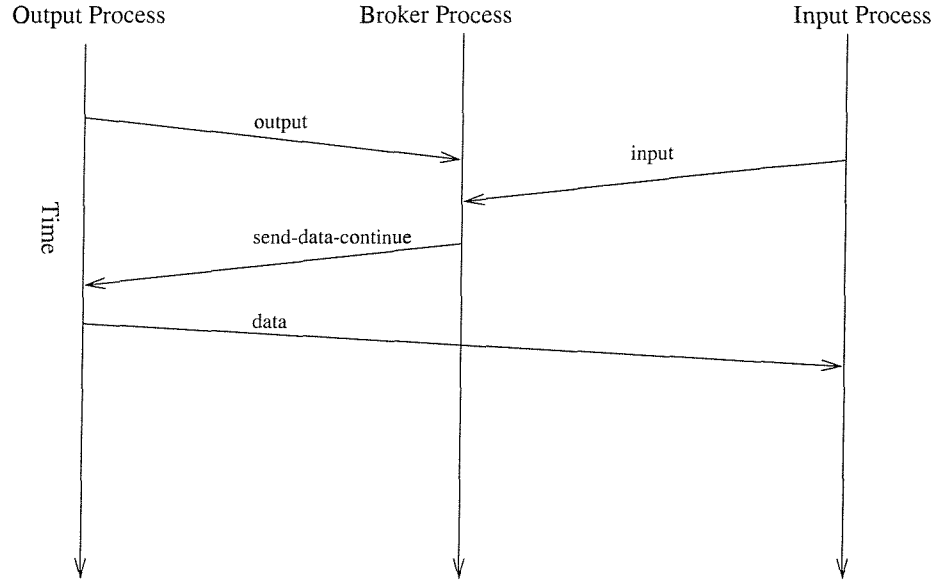

Figure 6.19: Synchronisation protocol with large data transfers

has already proceeded, then the failure case of figure 6.16 is followed, allowing the input process to re-send its `input` message to the broker.

The final case is where the output operation will not fail, but the input process could. This is shown in figure 6.20. The output branch is not automatically proceeded by the broker, rather the input process decides whether the synchronisation can proceed. If it can, it sends an `ack` message to the output process to allow it to proceed, otherwise it follows the failure case of figure 6.17 and sends a `nack` message to the output process, causing it to re-send its `output` to the broker. For large data transfers, the complete protocol of figure 6.15 is used.

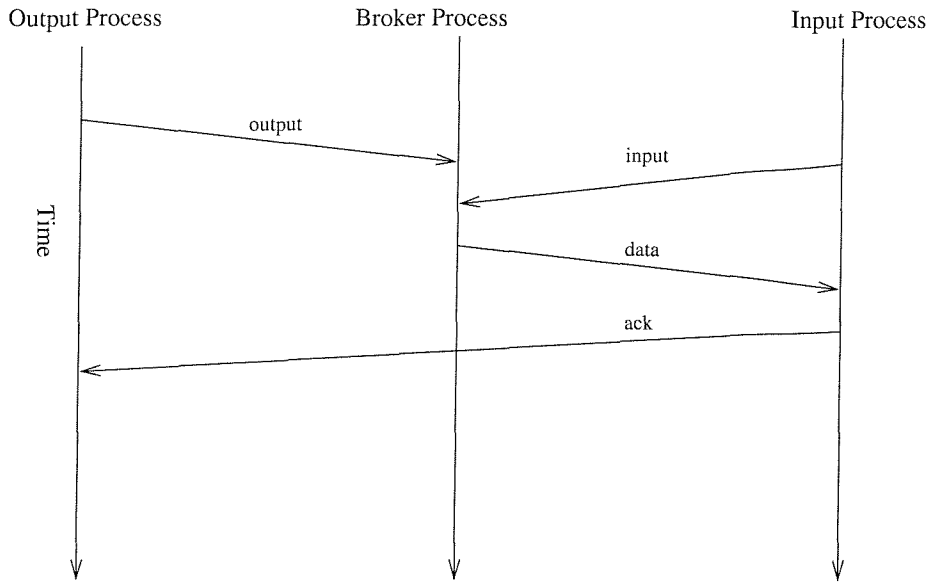

Figure 6.20: Synchronisation protocol where input process can fail

These protocol diagrams show that in order to perform a synchronisation between two processes, at least four messages need to flow between the processes and channel broker. For cases involving failure, even more messages will need to be sent. This shows the importance of the correct choice of broker in order to help reduce the amount of cross node network traffic.

Asynchronous communication

In many cases in a distributed system, the need for processes to synchronise can be separated from their need to communicate. Thus the synchronous communication protocol discussed above can cause inefficiency in a distributed system as senders needlessly wait for a suitable receiving process to become ready before being able to proceed ((Liskov *et al.* 1986)).

By allowing an asynchronous send operation on channels, communication can be achieved without the need for two processes to synchronise. Figure 6.21 shows the protocol used to perform asynchronous output.

The input operation is still synchronous, as input processes will always block until data becomes available. An asynchronous input could be designed where the process informs the channel system that it wishes to receive data on a channel, and a future is returned. The process may then continue to do work, touching the future when it needs the result of the input operation. If the operation has completed, the process will not block. If the operation has yet to complete, touching the future will cause the process to block until the data arrives, similar to the synchronous input. The message protocol used to achieve this input is the same for synchronous and asynchronous versions of `input`.

Asynchronous operations may not be performed where the operation chooses which branch of an `alt` operation will proceed. This is because in order to make

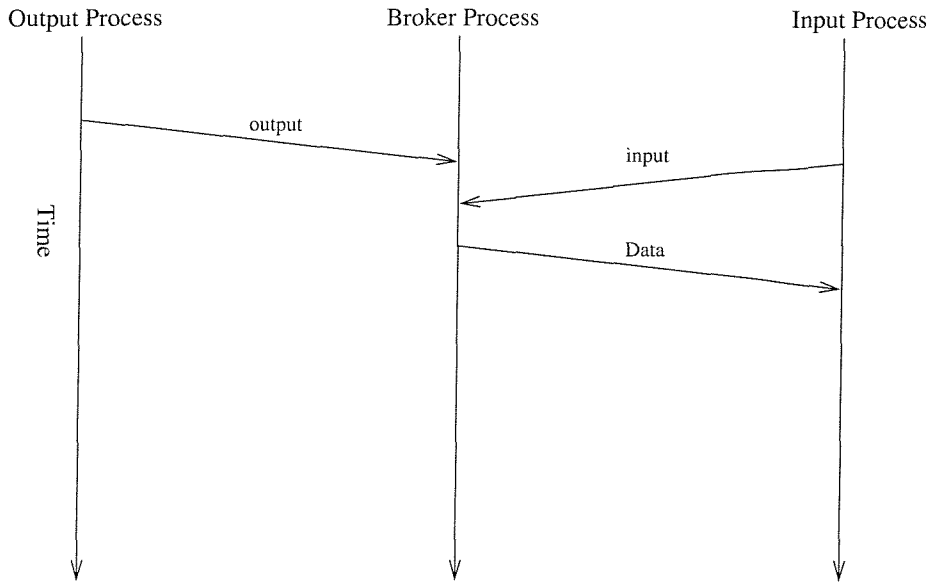

Figure 6.21: Asynchronous communication

the choice of which branch should proceed, one of the operations has to complete synchronously.

Modifications to the broker are required to implement asynchronous communications. Firstly, the asynchronous message protocol needs to be recognised. In addition, the broker has to cope with multiple input and output operations from the same processes and ensure that operations do not complete out of order. If two processes perform an output operation on a common channel simultaneously, the broker can arbitrarily choose which operation should be the first to synchronise with an available output process (although its algorithm should guarantee some form of fairness).

If the same process were to perform two asynchronous output operations on the same channel, it is desirable that an input process receiving on the channel receives its input in the same order as that which the sending process transmitted. Therefore it is desirable that the broker buffers subsequent output operations until previous operations have completed, thus ensuring the ordering of messages sent over a channel.

A problem can arise with this method if the process migrates to another node in the meantime. For example, figure 6.22 shows a process P perform an asynchronous output operation on node N_1 at time t. The resulting output message arrives at the broker at time $t + 10$. After performing its asynchronous output operation, P immediately migrates to another node (the channel protocol used to achieve this is not shown), and continues to execute on this node N_2 at time $t + 5$. Thus the migration took less time than the it took for the original output message to reach the broker. This could occur if the broker node was on the far side of a WAN link, with the process migrating to a node on the LAN, or if a connection had to be set up to the broker, but already existed between N_1 and N_2. Now

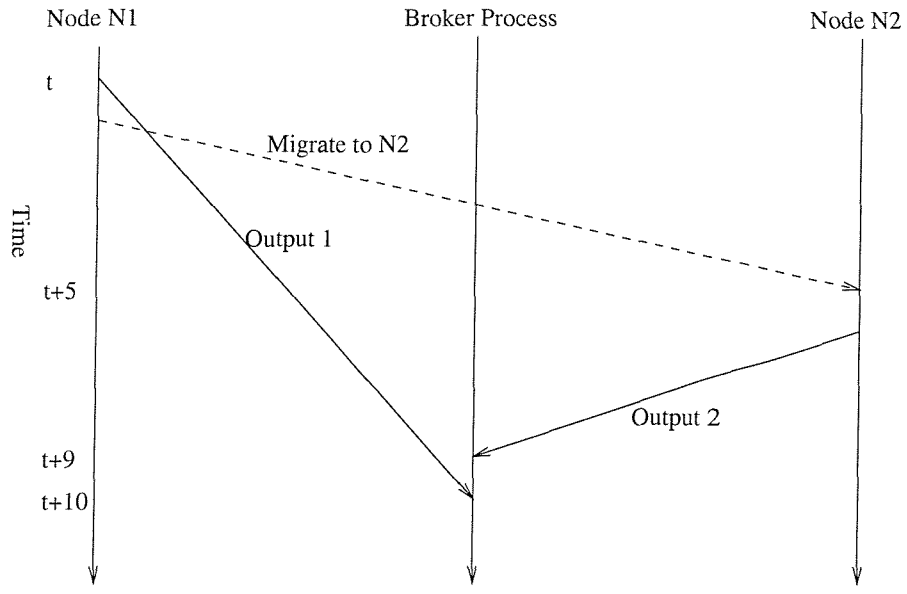

Figure 6.22: Out of order message delivery

the process performs a further asynchronous output on the same channel, with the output message arriving at the broker at time $t+9$. It can be seen that the messages arrive at the broker out of order, and thus are dispatched to a receiving process out of order.

Further problems arise when attempting to generalise the preserving of message order to all channels in a system. If a process performs an asynchronous output on channel a, and then channel b, it could be argued that the data of channel b should not be made available until a process had received the data sent over a. However channels a and b could be managed by different brokers, and therefore b's broker would have no direct knowledge of when the data on a had been consumed.

The data could be held at the sending process. By using the synchronous communication protocol of the above section, the sending node could feed the messages to the broker in the correct order. In the case above, it would first send an output message for a, and then when told to continue by a's broker, would send the queued output message for b. In this manner the processes would perceive an asynchronous interface, whilst the underlying synchronous message passing protocol would preserve event ordering within a process.

In order to preserve message order in the presence of migration, a synchronous send operation should be used to migrate a process. With the above algorithm, the channel operation used to migrate the process will not occur until all previous operations have completed. Using synchronous output, it is ensured that no future channel operations are performed until after the process has migrated. This prevents a set of operations being left on the previous node of the process, again allowing for the possibility of out of order message delivery.

Thus although the asynchronous protocol of figure 6.21 uses fewer messages than the synchronous communication protocols, by allowing for the delivery of out of order messages, it places extra burden on the distributed applications programmer.

By using an asynchronous programming interface to the underlying synchronous message protocol, with asynchronous operations queued at the client, message order is preserved, whilst allowing processes to improve performance.

Bounded queues can be used by the asynchronous channels interface to limit the amount of state held by a process. For example, a limit of n outstanding channel operations could be imposed (either per process, or for the node as a whole). If another asynchronous channel operation were performed, the channel system could transform it into a synchronous operation, to allow the buffer to completely clear. Alternatively it could block the process until the number of outstanding channel operations reached a low water mark, at which time the process could resume. This technique helps with the management of memory. For example, if an asynchronous producer was working at a faster rate than consumers were collecting the data and an infinite asynchronous buffer was used, then eventually the system would run out of memory.

6.6.7 *User interface*

Character based interface

The character based interface is a simple read–eval–print loop. This is represented as a listener process called `REP`, running on the HOC Scheme node. Thus configuring a node to have a character interface is as simple as deciding whether or not to create the `REP` process.

When the process is scheduled to run, it blocks the whole HOC Scheme node, waiting for the user to input an expression. This expression is then compiled and evaluated in the context of the listener process. The evaluation is passed an initial continuation which displays the result of the evaluation and reschedules the `REP` process.

As the `REP` process blocks the HOC Scheme node from executing whilst waiting for user input, a mechanism of turning off the `REP` is provided in order to allow unhindered execution to processes running on the node. The primitive `wait-all` blocks the current process, only allowing the process to proceed when no more processes are running.

Graphical user interface

The GUI for HOC Scheme uses Sun's **U**ser **I**nterface **T**oolkit (UIT). This is a set of C++ classes which provide a class based interface onto X View, a toolkit for X Windows.

The foreign function interface is used to provide primitives which the Scheme system may call on to perform windows operations.

In a similar fashion to the `REP` module, a `GUI` process is created when the HOC Scheme node is configured to run a GUI. When scheduled, this process calls the

GUI module to process any queued events. This is needed as the HOC Scheme node still runs as a single threaded application when the GUI is present, and so regular time needs to be given to the GUI module to handle its events.

When evaluating an expression, the GUI interface first compiles the expression. However, unlike the `REP` process, it does not execute it within the context of the `GUI` process. Instead it spawns a process to execute it. This is to prevent the expression blocking the GUI process.

Input and output primitives are extended to check the environment in which they are running. A read from a character based environment will read an expression from the console, blocking the HOC Scheme node until the expression has been input. If running under a GUI, the GUI module is informed that it should read an expression from the user. The reading process then blocks until the GUI module informs it that the expression has been read, at which point it can continue to execute. Note that only the reading process blocks; other processes are allowed to proceed due to the asynchronous behaviour of the GUI module.

Additional primitives only valid for GUI based nodes are provided as part of the HOC Scheme top level environment. These allow a wider range of GUI based input/output operations including the list choice and input/output window discussed in section 6.4.

The GUI module uses sockets as its means of interfacing with the X windows server. It needs to be informed when the socket is ready through a notification interface. Additional sockets may be given to this interface, allowing it to be informed of messages from remote HOC Scheme nodes and new incoming connections that should be processed. The network module passes responsibility for scheduling of message handling functions to the GUI module when executing in GUI mode, and is called back when the GUI module detects messages, or incoming connections need to be processed.

The scheduler never returns as there is always at least one process to execute, the `GUI` process. If the `GUI` process were to always process pending events and reschedule, this would lead to busy waiting when there were no events to dispatch, and no processes other than `GUI` to schedule. Therefore the `GUI` process checks for these conditions, and if it is the only process running, it blocks the system until one of its sockets becomes ready.

6.6.8 Starting a HOC Scheme node

Given the implementation of all modules, it is now possible to execute a HOC Scheme node. All that is required is for each module to be initialised in the required manner.

HOC Scheme nodes take a number of command line parameters at startup. This is preferable to using an initialisation file (for example `.hocschemerc`), as command line parameters allow different instances of HOC Scheme to be started on the same workstation — useful for test purposes.

The command line parameters are as follows:

- `controller` — Specifies that the node is the first node in the HOC Scheme network and should therefore maintain the master copy of the top level environment. It should also boot the resource locator process and create the channels associated with this process. If present this should be the first parameter;
- `<hostname>` — If the node is not to be a controller for a HOC Scheme network, the first parameter should be the hostname of the HOC Scheme controller node. Controllers listen on a well known port known by all HOC Scheme nodes, so it is not necessary to pass the port as a parameter;
- `listener` — Specifies that a character based read–eval–print loop should be started;
- `xlistener` — Specifies that a GUI window for the input of expressions should be started;
- `<resource>` — any other value is taken as a resource name which should be registered with the resource manager as a means of contacting the node.

Once the command line parameters have been parsed, initialisation occurs in the following order:

1. Global variables within the HOC Scheme modules are initialised;
2. The initial top level global environment is created;
3. The network module is started;
4. If the node is a controller, the master top level environment module is started, followed by the resource manager server process;
5. If node is not the controller, it connects to the controller and registers with the master top level environment;
6. If the node is a listener, the `REP` process is started; if it is an xlistener then the `GUI` process is started;
7. For all types of node, an `Entrance` process for the global *entrance* channel is started;
8. A newly created channel is registered with the resource manager for the supplied command line resources and an `Entrance` process which listens on this channel is created;
9. The scheduler is started.

The HOC Scheme node is then operational and can participate in the execution of distributed applications.

6.6.9 Stopping a HOC Scheme Node

Section 6.5.2 introduced the cases in which a HOC Scheme node could be removed from a network. These were:

1. The node fails;
2. The physical network becomes partitioned;
3. A shutdown request is received by the HOC Scheme node.

This current implementation of distributed HOC Scheme only caters for the last case, the ordered shutdown. Failure of a node is currently not handled. It will cause all processes on that node to stop interacting with the HOC Scheme network. It will also cause processes running on other nodes in the HOC Scheme to fail, most often by deadlocking if they try to perform a synchronisation with a process executing on a failed node. However, unless the failed node is the controller node, other nodes in the network will remain operational.

In the case where the controller node fails, other nodes will remain operational but will fail if they try to extend the global environment or attempt to initialise a top level channel.

The shutdown request is implemented by providing a primitive `(shutdown)` which causes the node on which it is executing to close. This can either be input directly using the read–eval–print loop (or windows interface), or by migrating a process to the node which needs to be shut down. For example, the following code closes a random node.

```
; ; Shutdown a node, any node.
(define (random-death)
  (output *any-evaluator* (lambda () (shutdown))))
```

The shutdown primitive performs the following steps:

1. Informs the controller that the node is closing.
2. Sends its broker's state to the controller, which takes over management of the channels the node's broker maintained;
3. Sends processes (including process state) to the controller, which takes over the execution of the processes;
4. Informs the controller that the migration is complete. The controller then broadcasts a message to all other nodes informing them that the controller is taking over the node's channels. Any subsequent channel operations which would have been directed at the closing node are then sent to the controller (this is handled by nodes network modules);
5. The controller sends a message to the closing node informing it that it can now exit;
6. The closing node forwards all channel messages to the controller, until it receives the shutdown confirmation message from the controller;
7. The closing node may now exit.

This method ensures that all processes and channels are moved off the closing node before it exits, allowing the HOC Scheme network as a whole to continue.

Processes that used user interface elements on the closing node will fail, as these are no longer available.

All channel and process state is transferred to the controller, as the controller is guaranteed to be available during this process. It follows that the controller cannot be shutdown, and if attempted, the `shutdown` primitive returns false.

6.7 Future work

This HOC Scheme environment is not of production quality. Further improvements could be made to allow it to run production quality applications.

For example, the windows interface could be extended, possibly by porting a complete toolkit interface to HOC Scheme. This would allow applications to construct interfaces suited to their requirements, rather than relying on the limited set of user interface components provided by the current implementation.

Further work needs to be carried out to improve the resilience of HOC Scheme, particularly in the presence of failure. This could require changes in the HOC Scheme language, for example, allowing synchronous channel operations to specify a time out period.

6.8 Summary

This chapter has detailed the design and implementation of a distributed system supporting HOC Scheme applications.

A partially connected network model is used, allowing the underlying network protocols to perform routeing operations whilst conserving network resources by not connecting nodes which do not communicate directly.

Application code and global variables are distributed though a shared immutable top level environment, with a replicated master model used for efficient lookups during runtime. All updates to the environment are handled by the master, allowing for the serialisation of update requests and the distribution of a consistent environment to all nodes in the system.

The broker model is used to implement a generalised model of communication, allowing the distribution of channels to change dynamically without the need to reconfigure the underlying system. Also, it does not rely on any particular network topology other than being able to send a message between any two points.

An existing runtime interpreter and compiler from the ICSLAS project was integrated into the HOC Scheme system to provide process compilation and evaluation support. All code and runtime data structures are Meroon objects, allowing for the distribution of platform independent code over the network. The compiler runtime was extended to allow distribution of code which references primitive functions to be transmitted over a network.

The next chapter goes on to show applications written in HOC Scheme and running on HOC Scheme networks.

Chapter 7

Applications of HOC Scheme

7.1 Introduction

Experimentation is an important part of investigating the value of a programming environment. Many programmers create a **hello world** program, which simply outputs a string to the console as a means of getting started with a programming environment, be it a completely new language, or a new compiler/interpreter for a language they already know. This allows programmers to discover many things about the new system such as basic program structure, input output libraries and operation of the compiler/interpreter through this one simple program.

Figure 7.1 shows a hello world program for HOC Scheme. It makes no use of the channels system, distribution or processes. Rather it allows the programmers to familiarise themselves with the HOC Scheme environment. The example shows one HOC Scheme node being started, with a character based read–eval–print loop, and the hello world program being loaded and executed.

The following sections discuss more complex experiments which have been carried out using HOC Scheme.

7.2 Prime numbers

Prime numbers have always been a source of fascination for mathematicians, going back to Euclid (who showed that there are infinitely many primes). Today primes are used in real world applications, for example their use in cryptography, where keys are generated using very large prime numbers.

A prime number is a member of the set of positive integers Z^+. This set can be partitioned into three sets which have empty intersections, 1, *prime* numbers $2, 3, 5, 7, \ldots$ and *composite* numbers $4, 6, 8, 9, \ldots$. For a number to be prime it can have only two divisors, one and itself. All other numbers (apart from one) are composite, in that they may be composed by the multiplication of other numbers. For example, two is a prime number, in that is can only be divided by itself and one, but all other even numbers are composite, as they can be divided by two.

```
nb88r% more hello-world.scm

;; Hello world
(define (hello-world) (format #t "hello world"))

nb88r% hocscheme controller listener
HOCScheme> (load "hello-world.scm")
OK
HOCScheme> (hello-world)
hello world
HOCScheme> (exit)

nb88r%
```

Figure 7.1: A hello world program for HOC Scheme

7.2.1 *Motivation*

The distributed HOC Scheme implementation is not designed for high performance systems such as prime number generators. The protocol needed to perform synchronous communication is too heavyweight for applications where processes must achieve a consistent high throughput of data in order to achieve a speedup over a sequential version of the same application.

However, HOC Scheme is a suitable language for writing such applications, even if the current implementation is not tailored to execute them. The channels model provides an elegant way for networks of processes to be created, and this section looks at various network configurations which can be used to generate prime numbers.

7.2.2 *Implementation*

There are several algorithms for generating and testing prime numbers on computers, including probabilistic methods for testing large primes. This section focuses on the use of a traditional method of computing prime numbers, using a sieve, with channels used to construct a data flow network in order to achieve parallelism though pipelining.

Sieve algorithms

A number of algorithms that make use of a finite number of sieves to generate prime numbers have been developed, the most famous being the *sieve of Eratosthenes*.

The sieve principle is similar to the filter primitive which acts on lists. The filter function is defined as follows:

```
(define (filter list predicate)
   (cond ((null? list) '())
         ((predicate (car list))
            (cons (car list) (filter (cdr list) predicate)))
         (else (filter (cdr list) predicate))))
```

Given a list of elements, the function returns a new list which contains only the elements of the initial list that hold when passed to the supplied predicate.

In the same fashion, a number n will only pass through a sieve containing a set of numbers S if it is divisible by no number in S, other than 1 or n. Thus if S is the set of positive integers (an infinite set of numbers) then any number which passes through the sieve of S will be prime, by definition.

Clearly it is impossible to test n against an infinite set of numbers. Two techniques allow this set to be reduced to a manageable size, whilst allowing the sieve to still guarantee that n is prime if it passes through the sieve.

1. Consider a sieve element m which is a composite number with factors k_i (i.e. $m = k_1 \times k_2 \times \ldots$). If any number in k_i divides by n, then so will m. Thus if all factors of m are in the sieve there is no need to test n against m. This implies that composite numbers whose factors are in the sieve need not be in the sieve themselves. This can be extended to factors which are themselves composite numbers, and by reduction, all composite numbers can be removed from the sieve, leaving only primes present in S. This reduces S from the set of all positive integers to the set of all primes;
2. The greatest element in S which needs to be tested to determine if n is prime is the square root of n. If a prime p was found to be a factor of n, then $n = p \times k$, where k is the other factor. If $k > \sqrt{n}$, then $p < \sqrt{n}$. Hence it is possible to determine if a number is prime by testing for factors up to $\sqrt{n}$.

These techniques allow a manageable sized set of primes to be held by the sieve. In order to test whether number n is prime, only primes less than the square root of n will need to be present in the sieve.

Through filtering a stream of integers it is possible for a sieve to grow dynamically by placing numbers it finds to be prime in the sieve, allowing the sieve to find larger prime numbers. Two guarantees must be met:

1. The current set of primes contained within a sieve must include all the primes up to $\sqrt{n}$, in order that it can be correctly determined whether n is prime;
2. A prime p must be added to the set of primes S used by a sieve before a value n where $n > p^2$ is processed by the sieve.

The first case is satisfied by passing in an ascending sequence of values which includes all primes, provided that no value is greater than the square of its predecessor. The sequence of integers starting from two (one is treated as a special case) and the sequence consisting of two followed by every odd number are suitable sequences for generating the set of prime numbers. The second guarantee places a requirement on the implementation to ensure that a prime is in the sieve before a number which requires that prime to correctly determine if it can be factored, is tested by the sieve.

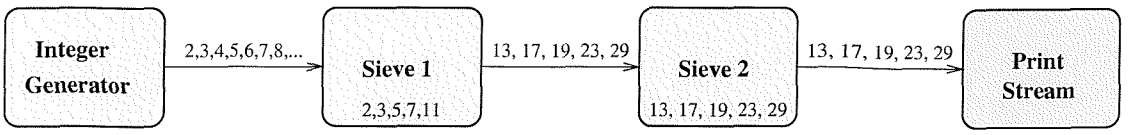

Figure 7.2: A pipeline of sieves

Using multiple sieves

A single sieve can be replaced by a number of sieves, provided that the union of sets of integers contained within the sieves meets the above requirements. This may be used to improve the performance of the prime number generator.

Figure 7.2 shows a simple pipeline of sieves. The channels are used to create a data flow network between the sieve processes. An integer generator process is placed at the source end of the network and feeds numbers into the pipeline. At the far end, prime numbers are produced and can be consumed by an application which then uses them for another application (for example to produce keys for cryptographic applications, or to simply display a list of prime numbers to a user).

It is important that these primes are incorporated into the sieves in order that they may correctly generate larger primes. There are a number of ways of doing this:

- The final sieve in the network, which determines for sure if a number is prime, incorporates all primes into its sieve. This causes an imbalance of work amongst the sieves. The final sieve will contain far more numbers than the sieves upstream. Indeed, unless upstream sieves are created with a pool of precomputed primes, they will be of no use as they cannot accumulate primes. Clearly, this scheme is inefficient;
- The primes are distributed evenly amongst the sieve processes. This is a more efficient algorithm, as each sieve has an equal amount of work to perform. If sieve processes are located on different nodes, the pipeline can perform work in parallel. It is important to ensure that a prime is added to a sieve before integers that could be factored by the prime are passed through the sieve;
- As HOC Scheme is a dynamic system, a dynamic approach could be taken. Rather than having a fixed static network of sieves, the network could be allowed to grow as more primes are found. This approach is discussed in more detail below.

A dynamic network of sieves

The dynamic network of sieves has a number of advantages over its static counterpart. No feedback mechanism is required, as a sieve is either at the end of a network generating and accumulating primes (similar to the first case discussed above), or is in the middle of a network and is using a fixed set of primes in its sieve. When an accumulating sieve is full, it can generate a new accumulating sieve process, whilst itself becoming a fixed, non accumulating sieve. Additional HOC

Scheme nodes can be added dynamically to the network, and become part of the prime number computation application when new sieve processes are spawned on them. A static network of sieves would have to be reconfigured to take account of new nodes, and would probably have to be restarted.

One disadvantage of this approach is the potential for performance drop off due to the large numbers of sieve processes that could be generated. It has been shown that channel communication uses a number of messages passed between the processes and the channels broker. If sieve processes are located on different nodes, some of these messages will have to be transmitted over the network. Thus in a pipeline of n sieves, $n+1$ channel synchronisations will occur, or $2(n+1)$ inter node messages, assuming that the broker and output process of using a channel are co-located. Therefore the more sieve processes, the longer the pipeline, and the longer the pipeline, the more time a prime will take to pass through due to the latency of message transfers. In a static network, the communications overhead is constant, as the length of the pipeline is fixed.

With the dynamic sieves implementation the size of the set of primes each sieve manages can also affect performance. Sieves in the pipeline that are "close" to the integer generator are going to process more numbers than sieves further downstream. For example, every even number other than two will be swallowed by the first sieve in the pipeline, halving the flow of numbers downstream of the sieve. This will in turn filter out further integers which it can factor, again reducing the rate of numbers flowing downstream to the next sieve.

In general, sieves containing lower number primes will filter out more integers than sieves containing larger primes. If the size of the set of primes managed by each sieve is kept at a constant, sieves downstream will perform less communication and processing than sieves further upstream. By increasing the size of the set each sieve manages as the network grows, decreased time performed doing communication can be compensated by increased time taken by the sieve to process numbers in the pipeline. This helps to keep sieves in the network doing roughly the same amount of work. Also by locating small primes in the sieve next to the integer generator, the number of communications can be drastically reduced, as numbers are more likely to be factors of small primes than larger ones contained in sieves downstream.

Implementing the dynamic network

Figure 7.3 shows the code used for creating a dynamic network of sieves. For brevity, the code which deals with the **set** abstract data type is omitted.

This provides a system that can be started with three processes — an integer generator, an initial sieve and a stream printer to display the results of numbers that pass through the pipeline (primes).

The performance of the application can be modified in two ways:

```
;;Integer generator, head of stream
(define (integers c n)
  (output c n)
  (integers c (+ n 1)))

;;Printer, tail of stream
(define (print-stream c)
  (input c (n) (display n))
  (newline)
  (print-stream c))

;;Accumulating sieve
(define (accumulating-sieve chan-upstream chan-primes S max)
  (input chan-upstream (p)
    (if (factor? p S)
      (accumulating-sieve chan-upstream chan-primes S max)
      (accumulate-prime chan-upstream chan-primes S max p))))

;;Add a prime to an accumulating sieve
(define (accumulate-prime chan-upstream chan-primes S max prime-
number)
  (output chan-primes prime-number)
  (let ((extendedS (set-extend S prime-number)))
    (if (< (set-size extendedS) max)
      (accumulating-sieve chan-upstream chan-primes extendedS max)
      (generate-new-sieve chan-upstream chan-primes extend-
edS max))))

;;Spawn a new accumulating sieve on another node
(define (generate-new-sieve chan-upstream chan-primes S max)
  (let ((pipe (make-channel)))
    (output *any-node* (lambda ()
                         (accumulating-sieve pipe chan-primes
                           (set-emptyset) (generate-new-
max max))))
    (fixed-sieve chan-upstream pipe S)))

;;A fixed sieve
(define (fixed-sieve chan-upstream chan-downstream S)
  (input chan-upstream (p)
    (when (not (factor? p S))
      (output chan-downstream p)))
  (fixed-sieve chan-upstream chan-downstream S))

;;Create a prime number generator pipeline
(define (generate-primes max)
  (let ((pipe (make-channel))
        (print (make-channel)))
    (spawn (lambda () (integers pipe 2)))
    (spawn (lambda () (print-stream print)))
    (spawn (lambda () (accumulating-sieve pipe print (set-
emptyset) max)))))
```

Figure 7.3: A prime number generator using dynamic sieves

1. Changing the initial maximum set size. This affects the size of the set accumulated by the initial sieve in the network — the sieve closest to the integer generator;
2. Varying the `generate-new-max` function to change the method of computing the set size of subsequent sieves in the network.

7.2.3 *Conclusions*

As with any channels based model, HOC Scheme lends itself to the easy implementation of streams based processing solutions. Pipelines of processes can easily be created, with synchronous output ensuring that processes need not perform buffering on the streams (data is effectively pulled through the network).

HOC Scheme extends this model by allowing the network connecting each stream module (process) to be extended by dynamically introducing new stream modules and reconfiguring the data flow network.

This allows an application to adapt, based on the amount of work it has to perform or on external factors such as new processing resource becoming available.

The ability for processes to migrate to other nodes in the HOC Scheme system allows dynamic building of data flow networks between nodes. For example, the prime sieve program of figure 7.3 implements a pipeline laying process, by placing each new sieve on a new node.

7.3 Metacircular HOC Scheme

This section presents a working implementation of HOC Scheme, called metacircular HOC Scheme, which is written in HOC Scheme. This implementation can be executed by the evaluator presented in chapter 6, henceforth called base HOC Scheme. The implementation is capable of evaluating any system defined in HOC Scheme, including itself. A complete code listing is provided in appendix A.

7.3.1 *Motivation*

Base HOC Scheme is based upon a number modules that do not lend themselves to the creation of dynamic networks of mobile processes.

1. A processor specific machine code (compiled using Scheme->C);
2. A static network between processing nodes (TCP/IP);
3. No built in support for multi processing (not supported by Scheme->C runtime).

This relatively low level starting point meant that in order to implement base HOC Scheme, abstractions needed to be provided to present the illusion of a dynamic environment to the HOC Scheme programmer.

By definition, the HOC Scheme system presents an environment where programmers can write processes which are mobile and use channels as first class objects to implement dynamically changing communications networks between the processes.

Such an environment would have been an ideal starting point to implement HOC Scheme. Given that this environment now exists, it seems appropriate to

demonstrate how it can support the implementation of other distributed environments. The fact that in this section we implement HOC Scheme is just one example of a distributed environment. In the next section, we go on to show how the Document Flow Model can be implemented in HOC Scheme.

An alternative implementation is also useful in verifying the correctness of the channels model. It is also a reasonably sized application which makes use of most of the features provided by base HOC Scheme, and thus can be used to increase confidence in the base implementation.

The system is truly metacircular — that is it can load and execute itself as well as other HOC Scheme applications. This allows confidence to grow not only in the base HOC Scheme, but also in the metacircular implementation.

7.3.2 Implementation

Models

Section 4.6.1 describes a typical continuation passing metacircular evaluator. This shows how constructs of the base language can be propagated through as constructs provided by the evaluator. For example the evaluator does not modify most basic types such as symbols, numbers and pairs. Other more complex types, such as abstractions (lambda expressions), can also be exposed. In addition, primitives of the underlying language may be exposed as primitives of the evaluated language (with appropriate wrappers to handle continuations).

Within HOC Scheme a number of base types can be exposed through the metacircular evaluator. These include:

- Basic types. For example symbols, numbers, pairs;
- Abstractions (lambda expressions);
- Processes;
- Channels.

Within this list the exposure of underlying processes and channels as primitive processes and channels of the metacircular evaluator are the most contentious. The passing through of underlying data types as types of the evaluator removes the ability of the evaluator to model the underlying types. In the case of processes and channels it can be argued that this reduces the scope of the HOC Scheme metacircular evaluator to test the underlying system by modelling its features.

It was decided that the most important feature to model in the metacircular HOC Scheme was the channels mechanism. It is the contention that a cleaner model of channels can be written in HOC Scheme than in a language with an interface to a more static communications mechanism.

Representing the metacircular process

The HOC Scheme process is a less interesting feature to model. The underlying process representation is sufficient for the metacircular HOC Scheme. There is a case for modelling the scheduler in future models of HOC Scheme. This would allow different scheduling strategies to be investigated.

The process is not a first class object. The continuation of a process can be captured, but this does not capture all information about a process. For example the following function captures a continuation and sends it to a remote node (using the supplied channel) for further evaluation.

```
(define (remote node)
   (call/cc (lambda (k)
              (output node (lambda () (k 'remote)))
              'local)))
```

Although we talk about the process migrating to the remote node, this is not strictly the case. The continuation of the process is copied to the remote node and is executed in a newly spawned process (performed by the `Entrance` process). Once the synchronisation has completed there will be two processes, each executing the same continuation. In this respect the `remote` function behaves in a similar fashion to the Unix `fork` routine. The return value of the `remote` function can be used to determine what behaviour the continuation should take.

Thus in order to expose the underlying implementation of processes through the metacircular HOC Scheme, a data type representing a process object need not be implemented. Instead, the API that manipulates processes needs to be exposed. For the HOC Scheme process, this means that the `spawn` primitive needs to be made available to the metacircular HOC Scheme.

A primitive is defined in the top level environment which takes a thunk as its argument and calls the underlying HOC Scheme `spawn` to create a new process to execute the thunk. As this thunk is part of a new process, a new continuation needs to be given to it before it can be executed. The `spawn` primitive is shown below.

```
(primitive 'spawn
  (lambda (thunk)
    ;; Call underlying HOC Scheme spawn
    (spawn (lambda ()
             (applyx #f (lambda (alt v) v) thunk '())))
    'ok))
```

The metacircular evaluator

The evaluator used in metacircular HOC Scheme is a standard continuation passing evaluator with one modification, the addition of an extra parameter to control whether the process being evaluated is currently evaluating an `alt` statement. This allows the channel module to behave differently, depending on whether an `input` or `output` operation is used as part of an `alt` statement. Section 7.3.2 discusses the implementation of channels.

In order to propagate this `alt` value around, the continuation is also modified to take two arguments — the value to be passed to the continuation and whether the continuation is being executed as part of an `alt` statement. Most primitives simply propagate the `alt` parameter back to the evaluator (through calls to `apply` or `eval`).

The `input` and `alt` special forms are also handled by the evaluator. An alternative would be to define primitive functions and implement a macro system to translate the special forms into function calls, but this was considered too costly for implementing just two additional special forms. The evaluator calls on `eval-input` and `eval-alt` respectively to process these special forms.

The top level environment is created with enough primitives to be able to load and execute the metacircular HOC Scheme. The distribution of this environment is handled differently to base HOC Scheme implementation. Instead of having a replicated master model of the environment, a single copy of the environment is held at the node where the (only) read–eval–print loop executes. This reads an expression from the user and calls `eval` to execute the expression. One of the parameters to `eval` is the environment in which the expression should be executed. In the base implementation of HOC Scheme, this contains just the lexical environment of any procedures that are being evaluated. The top level environment is held as a separate data structure, with the compiler being able to determine if the lexical or top level environment should be accessed to fetch the value of a binding. In metacircular HOC Scheme the environment passed to the `eval` function includes the top level environment in addition to any lexical environment.

This approach has the advantage of implicit distribution of the top level environment when a process (in the form of continuation of abstraction) migrates to another node. Since an abstraction or continuation contains references to the environment, this environment will be transmitted between nodes, and since the environment encompasses the top level, the complete environment needed to evaluate the abstraction/continuation will be sent.

The disadvantage of this approach is that only one evaluator can reliably extend this environment, forcing there to be only one read–eval–print loop. If multiple read–eval–print loops are required, it is relatively trivial to implement a replicated master environment, the skeleton of which is shown below.

```
(define (replicated-master clients database change-channel register-
channel)
  (alt ((input change-channel (change)
          (when (database-update database change)
            (map (lambda (c) (output c change)) clients))
          (replicated-master clients database
            change-channel register-channel)))

       ((input register-channel (new-client)
          (map (lambda (change) (output new-client change))
```

```
    (database-changes database))
  (replicated-master (cons new-client clients) database
    change-channel register-channel)))))
```

This function accepts change requests for a generic database and broadcasts it to all clients if the database accepts the change. It also allows other clients to register with the master, first sending it all changes performed on the database and then incorporating the client in the system to receive subsequent updates.

Metacircular channels

As discussed in section 7.3.2 the easiest way to model channels in the metacircular HOC Scheme is to make direct use of the underlying channel model. With this method, for example, the `make-channel` function would directly map onto the underlying `make-channel` function provided by the base HOC Scheme, and likewise for other channel primitives.

However one purpose of the metacircular HOC Scheme is to provide an alternative implementation of channels in order that the protocols used to implement channels may be further investigated, and the above direct implementation approach will not gain any further understanding in implementing channels based communication.

This is not to say that channels provided by the base HOC Scheme will not be used in the implementation of metacircular HOC Scheme channels. Indeed the choice of using the base process model and the desire to allow processes to communicate across nodes requires that we use base HOC Scheme channels (as the only method of inter-process communication in HOC Scheme is through channels).

In order to model the broker model it is appropriate to create a broker process with which evaluating processes can communicate to perform channel operations. There should be at least one broker process, although as in the base HOC Scheme this number can be increased to provide some parallelism in the channel's communication protocol.

As the broker is a separate process, channels are required to communicate with it. The implementation uses three channels, which the broker process waits on, using an `alt` statement.

1. An input channel which accepts input operations to be performed on the channels the broker manages;
2. An output channel which accepts output operations to be performed on the channel the broker manages;
3. A new channel channel which outputs a new (metacircular) channel that is to be managed by the broker.

The `make-channel` primitive can be now be defined as simply inputting and returning a channel using the new channel channel. If more than one broker is to be run, these brokers can be started using the same base HOC Scheme channel to output a new channel on, causing the `make-channel` primitive to select a broker

based on which one is ready to communicate. Alternatively each broker could be started with a unique new channel channel, with the `make-channel` primitive selecting from which broker to input a new channel, perhaps based on which node the `make-channel` operation is running.

The input and output channel used by the broker should be unique to each broker. Given this information the channel object returned by the broker can now be defined. It is modelled as a vector containing three items.

1. The input channel which should be used to contact the broker for input operations;
2. The output channel which should be used to contact the broker for output operations;
3. An identifier which uniquely identifies the channel within the broker.

Given such an object, a process has the ability to make input and output requests on a metacircular channel by communicating with its broker.

First the protocol for performing basic input and output is described, and then this is extended to cope with `alt` statements.

In a similar fashion to base HOC Scheme, the metacircular HOC Scheme broker accepts input and output requests for channels which it manages, and when an input request is found to match an output request, the rendezvous protocol is started.

This protocol uses dynamically created base channels to perform additional communications between the transmitting and receiving process. When performing an output operation on a metacircular channel, a process sends three items over the *output* channel to the broker.

1. The (metacircular) channel over which it wishes to communicate;
2. The number of data items which it wishes to send over the channel;
3. A (base) channel which should be used to specify where to send the data, called the reply channel.

If the broker cannot match this request against a corresponding input request, then it is queued until a suitable input request is received.

The output process then waits on the reply channel, over which it will receive another (base) channel which should be used to send the data to the receiving process. The process can then send the information and continue.

The input process performs a similar protocol. An input request is sent to the broker process containing the same three types of data as the output operation, a metacircular channel, the number of items that are to be received, and a base channel which the broker can use to kick off the rendezvous protocol, its reply channel.

The input process then waits on its reply channel, and inputs a request from the broker to rendezvous with an output process. The input process can then send its reply channel to the output process, and again wait on the channel. This time it will receive the data items from the output process, which it can then bind to the

input statement's variables. The synchronisation is then complete and the input process may continue.

In the case where no `alt` statements are in force, the channel mechanism allows optimisation of this protocol. The output process, rather than waiting for a channel on which to output the data items, could immediately output these items on its reply channel. The input process could then receive these data items when it receives the output's reply channel from the broker. This would remove the need for the input process to send a message to the output process, informing it where to send its data.

As in the base model, the provision for `alt` statements adds complexity to the rendezvous protocol as the scope for failure needs to be taken into account. The same basic protocol is used to perform the rendezvous, but extra data items are communicated to allow recovery from failures.

As shown in the previous section, sending a message to the input process starts the rendezvous protocol. The input process fails by sending `null` to the output process instead of the reply channel on which the data should be sent. On receipt of such a null message, the output process re-sends its output request to the broker. In a similar manner, the output process can fail by sending `null` when requested by an input process to send its data, forcing the input process to re-send its input message to the broker.

The extra parameter passed around with continuations is used by the channel module to process `alt` statements. When not involved in an `alt` statement, the parameter is set to `null`. However, the `eval-alt` function whose job is to evaluate an alt statement changes the value of this parameter to be a newly created base channel. This channel is used as the reply channel in any input or output requests that the branches of the `alt` statement send to brokers.

Rather than block waiting to input a message on the reply channel, input and output statements return an object which represents how the `alt` branch is blocked. Return is passed to the `eval-alt` function which can go on to partially evaluate the next branch, with the branch again returning control when an input or output statement is encountered. When all branches have been partially evaluated, the `eval-alt` function contains a list of objects representing all branches in the `alt` statement, with the brokers having received the input and output requests of the statement.

As all requests made to brokers were sent with a common reply channel, the `alt` process can then block waiting to input a message from any broker for any branch. To identify correctly which branch a broker needs to proceed, an extra parameter must be passed, indicating which channel this request is for. The `alt` process can then match this against the list of branches to determine the correct branch to proceed. For there to be a common reply channel for all branches, the broker also needs to indicate whether it is performing an input or output operation.

Once a request is received over the reply channel, the `alt` process continues, using the standard rendezvous protocol discussed in the previous section. Assuming the rendezvous succeeds, the `alt` process may proceed by invoking the continuation of the branch that has been selected (with the `alt` parameter reset to `null`). However, in order to prevent deadlock in the system, the reply channel must remain active, as other broker processes may try to use it for further synchronisations.

A separate process is spawned to input requests on the reply channel and to respond with a failure message to indicate the rendezvous has failed (the `fail-alt` process). There are a number of methods of preventing a build up of such processes:

1. Specify a count when spawning the process. This should be equal to the number of branches minus one. Once the process has performed this number of synchronisations, it can terminate, as all branches have been failed;
2. Have a single failure process which listens on all reply channels and fails them. Again a counter could be used to ensure that the number of channels the process fails does not reach infinity;
3. Have a separate protocol to inform brokers that an input/output request is no longer valid. This could then be used to inform brokers that all remaining branches of the `alt` statement are invalid. Once this is complete, no broker will try to communicate over the reply channel and so the `fail-alt` process can terminate.

The third option was chosen as it guarantees that the `fail-alt` process will eventually terminate, whereas the other options rely on the other branches of the `alt` statement being tried, which cannot be guaranteed.

Resource locator

The resource locator was not implemented directly in metacircular HOC Scheme. This is because there was no need; the resource locator is itself written in HOC Scheme and thus can run directly without modification in metacircular HOC Scheme.

The same is true of the `Entrance` process, although channels in the base HOC Scheme are used to distribute metacircular HOC Scheme `Entrance` processes to nodes in the HOC Scheme network.

7.3.3 *Experiments*

Confidence can be gained in the correctness of the model by running applications on top of it. Three applications were chosen.

1. The prime number generator of section 7.2 shows that the model can support processes migrating between nodes in order to build a network of processes;
2. The mobile phones example shows that metacircular channels can be passed over metacircular channels to reconfigure the communications network between processes;

3. Running metacircular HOC Scheme on top of metacircular HOC Scheme. This demonstrates that the model is truly metacircular and also that the protocol to handle `alt` statements works correctly (as the broker process uses an `alt` statement).

As shown, all these applications test different features of HOC Scheme, and together make up a comprehensive test of the model.

7.3.4 Conclusions

It is possible to model the HOC Scheme language directly in HOC Scheme. Indeed some of the features of HOC Scheme aid the implementation.

The use of base channels to communicate with brokers presents a flexible framework for the implementation of brokers. The metacircular channel neither cares where the broker resides, nor even if the broker is mobile. Further research could be carried out on the movement of brokers to achieve the best possible rendezvous speeds for the set of channels it is managing.

In the optimal case where `alt` statements are not involved, the rendezvous process can be achieved with three synchronisations using base channels. This is one less than the four messages the base model requires to perform a rendezvous (although metacircular channels make use of base channels, and thus the three synchronisations it performs translate into twelve messages being sent between nodes by the base HOC Scheme).

7.4 Document flow model

The document flow model provides a paradigm for process support through a network of distributed sub-processes. These sub-processes are distributed, and are also capable of being mobile in the sense that they can change their configuration dynamically and the physical processor on which they are running.

Section 4.3.4 provides a fuller description of the document flow model (henceforth referred to as DFM).

7.4.1 Motivation

DFM was developed in order to support the research being carried out into process enactment systems. Being able to execute DFM applications directly allows models to be investigated.

As its name suggests, DFM provides support for processing flows of documents around an application. These documents can be used to model documents found in traditional business processes, for example, a travel authorisation form, or more abstract notions such as speech in a mobile phone conversation. This ability allows DFM to model processes other than the business processes that were in mind when the model was developed. An investigation into the applicability of DFM to other application areas is carried out.

Several implementations of DFM have been developed. As well as giving an insight into DFM as a modelling language, different implementations provide a

means of investigating the applicability of the base language for implementing other languages.

7.4.2 *Implementation*

Two Lisp implementations were written, both interpreting a common base language representing DFM systems.

The first implementation was written in EuLisp and presents a stepper-like interface to a single user. The system is not distributed across nodes, but does make use of the EuLisp threads system, allowing for concurrent execution of threads on multiprocessor systems.

The second implementation is written in HOC Scheme. As might be expected, this allows DFM systems to run across multiple nodes in the HOC Scheme network, with multiple users interacting with the application.

This section presents the common language used to express DFM applications and an overview of each implementation.

A Lisp syntax for DFM

There are three basic types which are definable within DFM:

1. Documents;
2. Actons;
3. Rules.

Instances of each of these types needs to be defined within a DFM application. Each document has a type, with all documents of the same type having a common interface. Documents also contain data, and this is represented as fields within a document. For example the *source file* document could contain fields such as *author*, *version* etc. The definition of a *source file* document type is given below.

```
(defdoc SourceFile
  (name author version code))
```

This definition creates functions to create instances of the document type (`make-SourceFile`), and access items within a document (`SourceFile-author`).

Actons are containers which hold sets of documents, rules that act on these documents and a binding table which holds the addresses of other actons with which they communicate. The following code defines an acton, binding it to the top level variable *repository* and calling it *Source Code Repository*.

```
(defacton repository "Source Code Repository")
```

The most complex type defined in DFM applications is the rule. This takes a list of documents that it is to process, and when the guard allows, applies the action associated with the rule. An example rule to check-in a source code document into a repository is given below.

```
(defrule checkin ((s SourceFile) (c CheckIn))
  guard: (and (equal (SourceFile-name s)
                     (SourceFile-name (CheckIn-SourceFile c)))
              (< (SourceFile-version s)
                 (SourceFile-version (CheckIn-SourceFile c))))
  message: "Check in source file"
  action: (send self (CheckIn-SourceFile c)))
```

This rule requires that there are any two documents contained within an acton, where one is of type `SourceFile` and the other of type `CheckIn`, they correspond to the same source file and the checked in source file is newer than the source file currently in the acton's repository. When two documents are found that match these criteria, the rule is able to be fired and the message is displayed to the acton's user (if it has one). When the user selects the rule to be fired, the action clause specifies that the new source file should be added to the acton's document store, replacing the old source file. It is replaced as both the source file and check-in documents are removed before applying the rule; they must be returned explicitly to the document store by the action clause in order to persist.

A method of installing rules and setting bindings within rules needs to be provided.

```
;; Install the checkin rule to the repository acton
(add-rule repository CheckIn)
;; Set the acton that logs check outs
(set-binding repository checkoutlog checkoutlog-acton)
```

These methods are in fact syntactic sugar which wrap the rule/binding into an `AddRule` or `SetBinding` document, and then send the document to the acton. On receipt of the document the acton automatically fires a built in rule which causes the rule/binding to be installed.

And finally, a means of sending documents to actons and looking up bindings in actons needs to be defined.

```
;; Send an initial version of a source file to the source code control system
(send-acton repository (make-SourceFile "/test" "nb88r" 1
                           "code goes here"))
;; Sample use of lookup-binding
(send-acton (lookup-binding repository 'checkoutlog)
            (make-log "Doc added"))
```

DFM in EuLisp

The implementation of DFM in EuLisp consists of three modules:

acton Implements the behaviour of actons. It defines the structure of documents, rules and actons;

acton-language This module provides macros which map the DFM language constructs defined in the previous section into function calls made to the **acton** module;

acton-stepper Provides a stepper like interface to a set of actons by using an interface exposed by the **acton** module. This allows actons to be queried to see the contents of an acton's document store and which rules may currently fire, and to fire acton rules.

DFM applications can be defined in their own modules, importing the above modules to access the DFM functionality.

Rather than computing the complete set of rules which may fire when the stepper module queries an acton, a cache of combinations of documents which can cause rules to fire is maintained. This cache is updated when documents are added or removed to/from the acton's document store. This allows limited searches to be performed against the document store rather than a complete search which would be needed if caching were not implemented.

The EuLisp object system(Telos) is used to implement the **acton** module. For example the class `ActiveRule` is a subclass of `Rule`, containing the cache of documents which can be fired for the rule. The **acton-language** module `defrule` macro creates an instance of `Rule`, and installing this in an acton creates an `ActiveRule`. A generic function is used to process received documents (instances of classes subclassed from `Document`). Standard rules are applied to process received documents requesting that rules be added to or removed from an acton.

EuLisp threads are used to increase concurrency between actons. This might seem unnecessary when using a stepper, where the majority of time is spent waiting for the user to select rules to fire. However, the stepper is only one method of managing actons, and other more efficient managers could be written that automatically step rules and would benefit from concurrent execution of actons.

There are a number of methods by which threads may be utilised in evaluating actons.

Each acton could have a thread associated with it, with all requests made to the acton being passed to its thread for processing. This has the advantage that no protection against concurrency of the acton's data structures is required, as only the acton's thread will directly access this data. However, forcing all requests to go through a single thread of control can reduce the scope for concurrency (for example, multiple rules in an acton cannot fire concurrently), and also increase the work required by synchronous calls. For example, in order to retrieve the rules which may currently fire in an acton, the application must signal the acton's thread to perform some work, passing it a data structure that contains the request to retrieve active rules, and then wait until the acton's thread has completed the request.

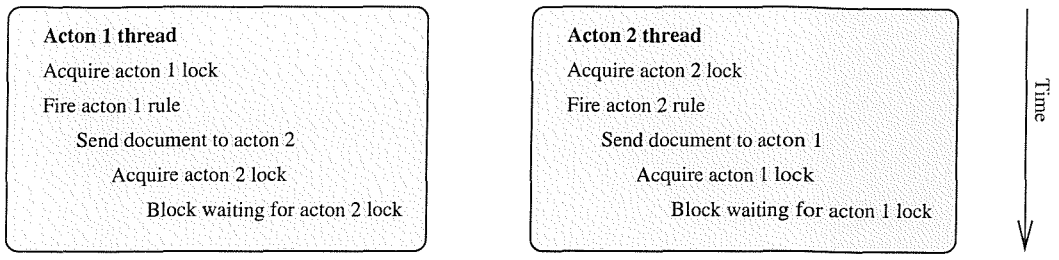

Figure 7.4: Deadlock with concurrent firing of rules

Whilst this message passing paradigm might be appropriate for distributed systems where the requester and requestee reside on different nodes, it only presents additional complexity to applications that are to run on a single node.

An alternative approach is to spawn threads to fire rules asynchronously. This would quickly free up the acton management thread, allowing it to select other rules to fire or perform other processing such as logging that it fired a rule. The thread which fired the rule would persist until the rule had completed processing. Locks or semaphores need to be used to prevent rules from accessing an acton's data structures which other threads (firing rules or the management thread) are accessing. There are two main items which can be protected:

The rule — This protects the rules cache from concurrent updates;

The acton — This protects the acton data structures (set of rules, document store, and bindings) from concurrent updates.

A coarser grain of concurrency could be gained by using just the acton's lock. By acquiring the acton's lock before modifying any rule installed in the acton, protection can be gained against concurrent access to rules. This is sufficient, as all actions go through an acton (delivery of documents, requests to obtain rules which may fire), and thus the acton can control concurrent access.

Care needs to be taken to avoid deadlock when dealing with multiple threads and locks. For example, the case where two actons simultaneously send each other documents (by each acton firing one of its rules). Figure 7.4 shows how deadlock could occur if each rule is executing in its own thread.

A trivial way to avoid this type of deadlock is to never acquire more than one lock at a time. However, it is not always easy to implement, and is sometimes impossible. With DFM, such a strategy can be employed to avoid deadlock, as when a rule fires it contains all the information needed to process the action clause of the rule. It has the documents, and knows the address of its acton (`self`). This means that a rule does not need to keep open the lock to its acton. It only needs to take out locks on actons when looking up bindings (when it obtains its own acton's lock) and sending a document to an acton. As these actions can be serialised there is no need for a rule to have ownership of more than one lock at any one time.

The one thread per firing rule implementation was adopted, due to its simpler implementation and greater scope for concurrency. A disadvantage of this

approach over the single thread per acton is that firing a number of rules could introduce too much concurrency to the system, causing the EuLisp runtime to spend too long managing its pool of threads rather than executing them.

This has been solved by actons limiting the number of rules they have concurrently executing. If a manager requests that an acton fire a rule, this request can be stored and executed at a later time if the acton finds that too many rules are currently executing. This approach can also be used to improve thread usage — the thread which is used to fire an acton's rule can be reused. Once the thread has finished firing the rule, rather than terminating, it can check the acton to see if it has any pending rules to fire. If so it can "steal" the rule from the acton and fire it, removing the need for the acton to create a new thread to fire the rule.

DFM in HOC Scheme

The HOC Scheme implementation makes use of the process and channels model to provide a multiuser distributed environment in which DFM applications can execute. This is a useful demonstration of the ability of HOC Scheme to provide powerful constructs which can be used to simplify the implementation of other distributed environments.

Like the EuLisp implementation, the implementation of an acton is kept separate from its user interface. This allows different types of user interface to be created (a Windows interface and stepper are provided in this implementation), and also allows the acton to continue operating in the absence of a user interface. This could be appropriate for actons which automatically fire rules, or when the user logs out but wishes to continue to collect documents in order to participate in the application when they come back online.

A single process model is used to implement an acton, with channels providing the external interface by which the requests may be made to the acton. Whilst this is similar to the single thread model discussed and discounted in the EuLisp implementation, the presence of distributed nodes makes the single process per acton more appropriate for the HOC Scheme implementation.

Figure 7.5 shows the basic skeleton of an acton process. Five channels are used to manage the interface to the acton process. The *receive* and *fire* channels process new documents arriving at the acton and requests to fire rules owned by the acton. On receipt of these messages the appropriate processor is called, which then performs the requested action and tail recursively calls the `acton-engine` to continue handling requests (with a possibly updated state).

The *listen* and *unlisten* channels provide user interfaces with a means of registering a channel on which they wish to receive notifications when the state of an acton has changed. Indications sent out on listener channels are sent asynchronously to prevent deadlock from occurring (due to a user interface process attempting to fire a rule whilst the acton process attempts to notify the user interface).

Finally, user interfaces may always input data from an acton's *ready* channel to receive the current state of an acton. This is useful for interfaces which do not require asynchronous notification, such as steppers.

```
(define (acton-engine act state)
  (alt ((input (acton-receive act) (doc)
    (acton-process-receive act state doc)))
       ((input (acton-fire act) (rule)
    (acton-process-fire act state rule)))
       ((input (acton-listen act) (listener)
          (acton-process-listen act state listener)))
       ((input (acton-unlisten act) (listener)
          (acton-process-unlisten act state listener)))
       ((output (acton-ready act) (acton-generate-ready-
rules state))
        (acton-engine act state))))
```

Figure 7.5: An acton process implemented in HOC Scheme

The `state` parameter of the acton contains state similar to that of the EuLisp implementation including:

- Document store. The current set of documents the acton has received;
- Bindings. References to other actons used by the rules of the acton;
- Rules. Installed rules including caches;
- Listeners. Channels to send asynchronous notifications about changes in the acton's state.

The acton is represented as an object which contains all the channels that may be used to communicate with the acton (the `act` parameter in the acton engine procedure). This object can be passed between actons safely (as fields within a document), due to the fact that channels are first class objects which may be transmitted over channels.

The stepper user interface takes a number of actons which it is to manage. It then queries each acton in turn on its *ready* channel and presents a list of rules which may fire to the user. The user then selects an option and the stepper communicates with the relevant acton over its *fire* channel. The process then repeats.

The windows interface uses the asynchronous notification interface of the acton to receive updates of an acton's state and presents this state in a list pane to the user. Again the user may select an item in the list pane, causing the windows interface to make a request to the acton that the rule be fired.

The caching and rule firing mechanism uses the same techniques as the EuLisp implementation, but is a slightly different implementation due to the lack of the Telos object system.

7.4.3 *Experiments*

DFM has been used to model a variety of applications, ranging from business process models through to protocols. This section presents some applications which have been written in DFM.

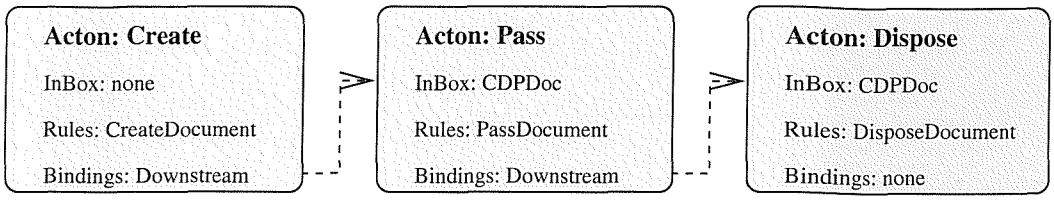

Figure 7.6: A basic acton network

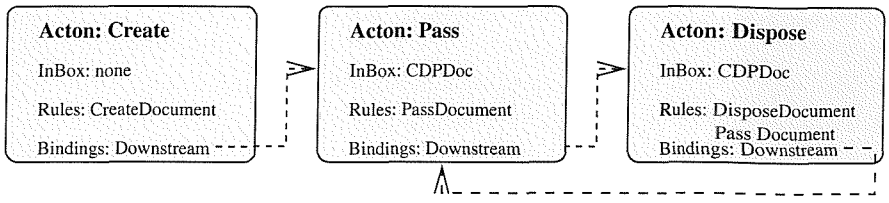

Figure 7.7: An acton network with feedback

Create, dispose, pass

A simple application, this is a similar process to that of **hello world** in that it demonstrates how to write DFM applications. In this example, three actons communicate in a pipeline, with documents being generated by the acton at the head of the pipeline, passed through by the acton in the middle and finally disposed of by the document at the end. Three rules are needed to manage these operations.

CreateDocument — Creates a document and feeds it into the pipeline;
PassDocument — Passes the document through the pipeline;
DisposeDocument — Removes a document from the pipeline.

The following code demonstrates how the **CreateDocument** rule is written.

```
(defrule CreateDocument ()     ;;no documents required to fire rule
  guard: t                     ;;can always fire
  message: "Create document"
  action: (send downstream (make-CDPDoc)))
```

Figure 7.6 shows an example network and route documents take. This network is configured by installing the relevant rules in each acton and setting bindings to ensure that documents flow in the correct direction. The network can be reconfigured easily. Figure 7.7 shows a network with the final acton having the choice of either disposing of documents or passing the document back to the head of the pipeline. This is achieved by installing the **PassDocument** rule in the last acton, in addition to the **DisposeDocument** rule. The `downstream` binding needs to be set to point to the middle acton.

Library

Consider the acton. In many ways it is like an email account. It can receive documents (akin to receiving email), send documents to other actons (send email to

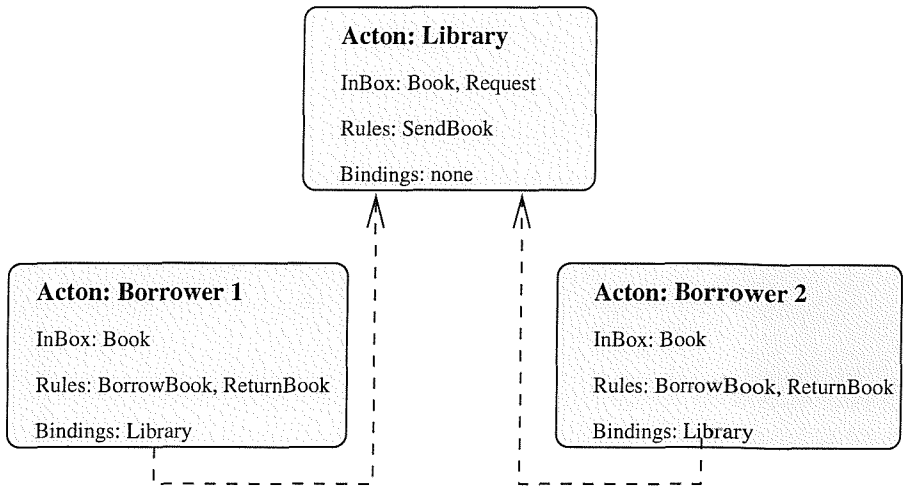

Figure 7.8: Example library network

other email accounts) and process documents (email rules). The create, dispose, pass pipeline shown above could be written using rules in standard email packages. However, DFM extends the email model by allowing the user choice, for example allowing a rule to be selected where several rules apply to the same document (as in the feedback acton of figure 7.7).

It also extends the email rules concept by allowing combinations of messages to be processed by a single rule (whereas email rules are applied to only one message, not combinations).

This example shows how multiple document types may be used in implementing a simple model of a library. There are two types of document, the **Book** and the **Request**. Figure 7.8 shows a possible configuration of a library system, containing two borrowers and a central library. Borrower actons can use the **BorrowBook** rule to send **Request** documents to the library acton. The following rule is used by the library acton to match books against borrower's requests.

```
(defrule SendBook ((request Request) (book Book))
  guard: (equal (Book-ISBN book) (Request-ISBN request))
  message: (format t "Send ~a by ~a" (Book-title book)
                                     (Book-author book))
  action: (send-acton (Request-borrower request) book))
```

This combines request and book documents contained within the libraries document store and sends the book to the borrower acton which issued the request. The borrower may return the book by applying the **ReturnBook** rule.

Mobile phones

Figure 7.9 shows the now familiar network of the mobile phones example expressed using actons. As DFM is an asynchronous model there are two options to modelling the mobile phones example.

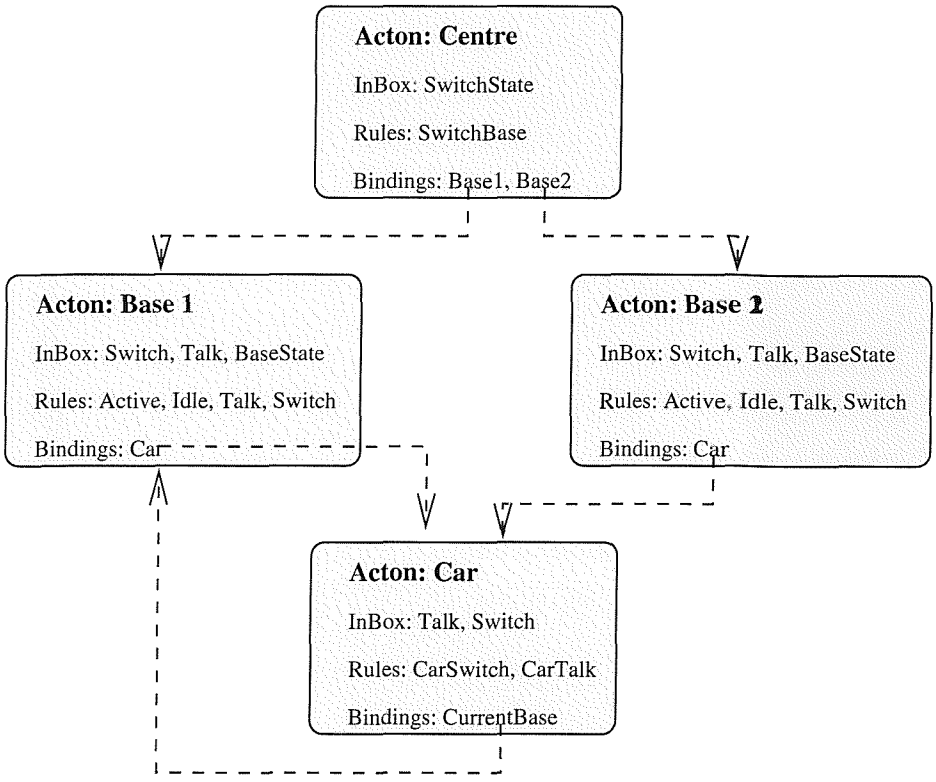

Figure 7.9: Mobile phones

1. Modify the behaviour of the system to suit the asynchronous behaviour of DFM;
2. Pass documents as reply messages to model the synchronous behaviour of channels used by the mobile phones.

The first option allows situations not supported by the channels model of mobile phones. This is because messages (expressed in terms of documents) can be received (processed by rules) out of order. For example, a **car** acton could continue to talk to **base one** even though it has received a document informing it to talk to **base two**. Only when the **car** acton has processed the **Switch** document will it send its **Talk** documents to **base two**.

By acknowledging receipt of documents, the synchronous protocol of the π-calculus mobile phones example can be modelled. However, this increases the complexity of the implementation using actons. The direct encoding in π-calculus is more suitable.

"My beautiful process"

My beautiful process is a simple example which demonstrates another feature of actons — their ability to change their behaviour over time[1].

[1] Devised by Professor Peter Henderson, University of Southampton

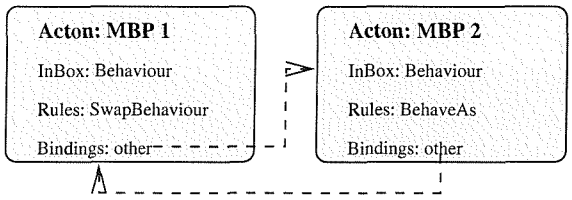

Figure 7.10: My beautiful process

Figure 7.10 shows the network which implements the example. Two actons send documents between each other which causes them to change their behaviour to be that of the other acton.

This can be expressed in terms of π-calculus as follows.

$$\begin{aligned} P &= a?Q \rightarrow Q \\ I &= a!I \rightarrow P \end{aligned}$$

As can be seen, when run in parallel, process P behaves firstly as P, and then as I, as it receives I over a. I firstly behaves as I and then as P. Thus the two processes swap behaviour, with P now behaving as I and I behaving as P. The next synchronisation then causes the two processes to swap back to their original behaviour, and so the system progresses, with the processes continually swapping behaviour.

This is encoded in DFM with two rules. **SwapBehaviour** sends a document to its companion acton, providing it a new rule to install, and **BehaveAs** consumes this document and installs the provided rule over the currently installed rule.

The two rules are shown below:

```
(defrule SwapBehaviour ()
  guard: t
  message: "Swap behaviour"
  action: (send a (make-Behaviour 'rule swap-behaviour))
          (change-behaviour self BehaveAs))

(defrule BehaveAs ((Q Behaviour))
  guard: t
  message: "Change behaviour"
  action:  (change-behaviour (self Behaviour-rule Q)))

;;Standard lisp function to remove current behaviour rule and install
;;a new one
(define (change-behaviour self new-behaviour)
  (delete-rule self (lookup-binding self 'current-behaviour))
  (add-rule self new-behaviour)
  (set-binding self current-behaviour new-behaviour))
```

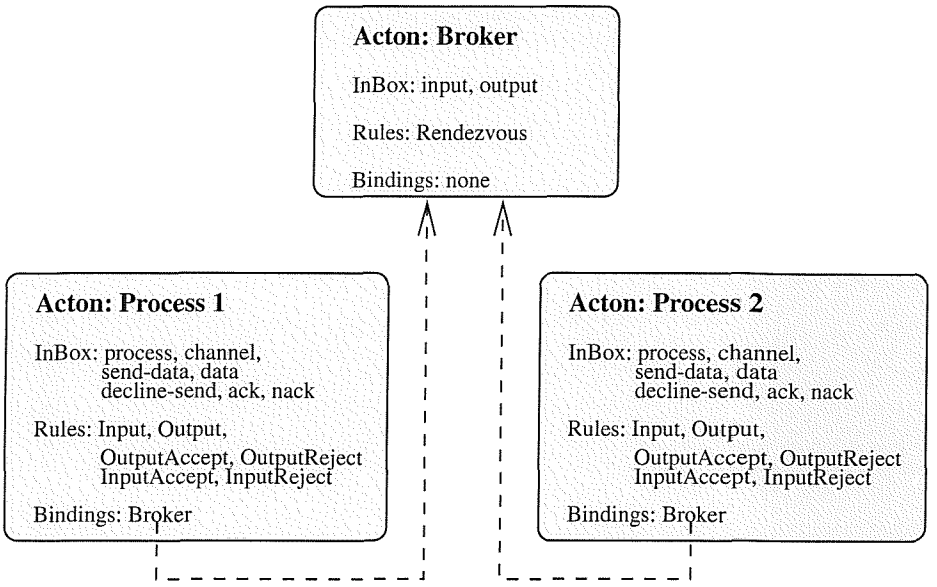

Figure 7.11: Channel protocol

The two actons of figure 7.10 can then be primed, **P** with **BehaveAs** and **I** with **SwapBehaviour**. Bindings which point to the current behaviour also need to be set. This technique demonstrates how an Acton can reflect and change its behaviour dynamically during its lifetime ((Dourish 1992)).

HOC Scheme channel protocol

The final example presented is a model of the communications protocol used by the implementation of distributed HOC Scheme. This allows transitions in the protocol and the messages passing between nodes to be viewed at a more leisurely pace than that at which the actual protocol runs.

Figure 7.11 shows the configuration of actons used to examine two processes communicating using a common broker process. Documents are used to represent the messages which implement the protocol, and a special document **Process** is used to store information about the current state of a process (for example, waiting for an **Ack** document from an input process.

Rules are used to test how messages should be handled, given the current state of a process. For example, the rule below shows how a broker kicks off the rendezvous process when it finds that an input and output document match.

```
(defrule Rendezvous ((o Output) (i Input))
  guard: (and (equal (Output-chan o) (Input-chan i))
              (not (equal (Output-sender o) (Input-receiver i))))
  message: "Perform rendezvous"
  action: (send-document (Output-sender o) (make-SendData (Output-mid o) i)))
```

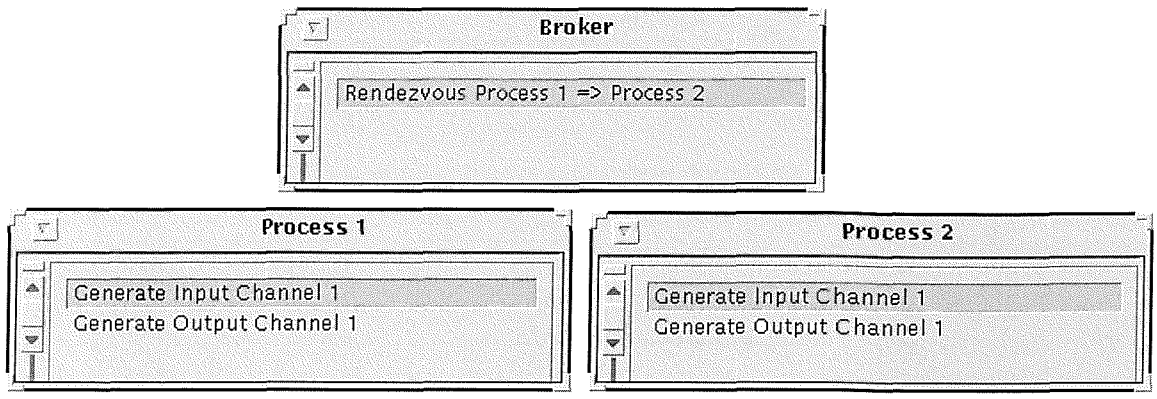

Figure 7.12: Rendezvous under windows interface

Upon receipt of the **SendData** message, an output process can apply the following rule if it is willing to accept the rendezvous:

```
(defrule AcceptOutput ((s SendData) (p Process))
  guard: (and (= (SendData-mid s) (Process-mid p))
              (eq? (Process-status p) 'generate))
  message: "Send data to input process"
  action: (let ((input (SendData-input s)))
    (send-document (InputReceiver input)
     (make-Data (Input-mid input) self))
     (set-process-status! p 'wait-ack)
     (send-document self p)))
```

Figure 7.12 shows the state where a rendezvous is about to be started when running under a windows environment.

7.4.4 Conclusions

This section has presented a method of encoding DFM applications using a Lisp like syntax, and the implementation of two environments supporting the execution of these applications.

It has been shown that EuLisp threads provide a method of increasing concurrent processing when executing DFM applications on shared memory multiprocessor machines. For distributed environments, HOC Scheme provides a suitable environment for implementing alternative distributed environments, with channels providing a suitable generic transport for creating networks of connected processes.

Further investigation could be carried out on the HOC Scheme model to investigate using other aspects of the language in optimising the execution of DFM applications. For example, acton processes could migrate towards the acton with which they perform most communication, helping to reduce communication costs. The flexible nature of channels means that user interface processes (steppers and

```
(defrule SendBook ((request Request) (books Book*))
  guard: (not (ISBN-present (Request-ISBN request) books))
  message: (format t "Decline ~a" (Request-ISBM request))
  action: (send-acton (Request-borrower request) (make-
DeclineRequest request))
          ;;Put books back into the library
          (send-acton self books))
```

Figure 7.13: A rule matching a set of documents

windows) can still contact their respective actons regardless of where they reside, even if they move between nodes during their lifetime.

The implementation of DFM presented here exposes a problem with the model. Consider the library example of section 7.4.3. The **SendBook** rule is used by the library acton to dispatch books to borrower actons when a **Request** is found to match a book in the library. No consideration was given to the case where no book in the library matches a request. Currently the only method of processing the request would be to allow failure of all requests, including ones where a book was present. It is impossible to create a rule which can be fired when a request is present that matches no book currently in the library.

The model could be extended to allow rules to match all documents of a certain type, allowing sets of documents to be searched to determine if a rule can fire. Figure 7.13 shows a rule which could be added to a library acton which checks against all books in a library to see if a request can be serviced (an asterisk is used to denote matching all documents of a certain type, with the parameter bound to a list containing all the documents).

It has been shown that DFM is a useful modelling tool, not only for business processes but also for other models such as protocols.

7.5 Summary

This chapter has shown how the features which HOC Scheme provides can be used to implement various applications.

The channels model allows data flow networks to be easily constructed, with the higher order features of channels allowing the network to be reconfigured and extended during the application's lifetime.

Lisp's ability to parse and process symbolic data structures makes it an ideal language for building interpreters for other languages. HOC Scheme extends this by allowing distributed languages to be implemented. The high level communications model presented by HOC Scheme make it an ideal starting point, as many of the features required by distributed languages are provided.

Chapter 8

Conclusions

8.1 Introduction

This chapter summarises the benefits which can be derived from distributed programming, and the role that mobility has to play in distributed environments.

HOC Scheme is compared to a number of related systems which also provide communication over a set of dynamic channels.

Finally, future directions of HOC Scheme and related technologies which could be investigated are discussed.

8.2 Mobility in distributed systems

Benefits can be gained from distributed programming without the need for mobile processes. Applications such as email and network file systems are amongst the most widely used applications on computer networks, but the processes which implement these systems run on fixed servers, with clients connecting to them using a variety of protocols.

Toolkits such as CORBA provide a powerful environment for writing client server distributed applications, which may be multi tier (as servers can act as clients to other servers), without the need to provide support for mobility.

However, mobile processes can enhance distributed systems by allowing for the movement of processes to the most "appropriate" points in the network where they may best achieve their aims.

8.2.1 Accessing large data sets

A case for the use of mobility is where the process is smaller than the set of data upon which it acts. In this case it is more efficient to move the process towards the data rather than the data towards the process.

This reduces both network utilisation (a small process travels over the network, rather than a large set of data), and latency, resulting in a more efficient system.

An example of this approach is SQL, where databases are queried by clients sending SQL programs to the database server. The server then executes this program against its database, allowing it to find and return to the client the subset of

data that it requires. This results in less use of the network (the client does not need to fetch the entire database). A similar approach is adopted by Postscript, where small programs are sent to printers and when executed produce large bitmaps.

8.2.2 *Distribution*

The ability to distribute applications through a network can be used as an aid to software maintenance. So-called "thin clients" can download applications from a central server as and when they are required (maintaining a cache of these programs locally if the client has a local disk).

This is used in the distribution of Java and ActiveX programs through web browsers. The client can check with the server for newer versions of programs, resulting in efficient distribution of updates, without the need for explicit user intervention.

8.2.3 *Agency*

Agents are small programs which perform tasks on behalf of a user ((Jennings 1995)). These tasks can range in complexity (for example a simple task could be to inform the user when their favourite web pages are updated, while a more complex example could check airline flights availability and correlate with available rooms in a hotel).

Tasks could be performed on the computer on which the agent is programmed, with it sending requests out over the network to gather information. However, like SQL, performance enhancements could be gained by providing a framework where the agent can become mobile, allowing it to visit nodes on the network to collect relevant information pertaining to its task.

This solution would allow the user to "launch" agents without having to leave their computers on. In the presence of a dynamic network, agents could return to the user's computer even if it had moved to a new point in the network (a notebook computer, for example).

8.2.4 *Dynamic distributed programming*

The ability of processes to move between nodes during the lifetime of a distributed application could also lead to changes in the way applications are programmed. For example, the prime number generator described in section 7.2 employs a process which dynamically extends a network of sieves, moving between nodes to introduce parallelism to the system.

Document processing applications could also benefit from dynamic programming. As well as distributing documents around nodes in a groupware application, processes that act on the documents could also be distributed. This would allow system administrators to introduce new types of document to the system easily, without the need to reconfigure each node in the system.

8.3 Coordination of mobility

8.3.1 Cooperating processes

Some forms of distribution require little or no cooperation between tasks to achieve their goal. For example, SQL statements act in isolation when executed by the server; they do not communicate with other SQL statements which may also be executing in parallel.

However a large set of distributed applications require processes to cooperate so that logical tasks may be partitioned over a number of processes, either to achieve speedup through parallelism or to access resources specific to a node.

Mobile processes are no exception to this. However, maintaining communication links in the presence of mobility presents problems. One method would be to inform a centralised database of the location of each process, updating the database as processes migrate. This is akin to a mobile phones network, where a phone travels between cells, and a central controller must be kept up to date so it can contact the phone when new calls arrive.

This approach can lead to race conditions; for example there could be a race between a process migrating and another process trying to locate the migrating process so that it can send a message to it.

8.3.2 Dynamic network architectures

The dynamic network of channels introduced in chapter 4 describes an elegant solution to the coordination of mobile processes. By allowing channels to be treated as standard data types, they may themselves be transmitted to other processes over channels, allowing the network to change over time.

For example, the *resource locator* process allows a channel which represents access to be returned as part of a query (which itself involves communicating with the *resource process* over channels). This channel may then be used to communicate with the resource, which may be situated on any node in the network.

The promotion of a channel to a higher order data object also removes the need for the two forms of communication shown in section 3.10, where requests are made using a fixed network of channels, and replies are implemented by the instantiation of a logic variable. With higher order channels, a channel can be passed as part of the request, with the requestor then waiting to input the result of the operation from this channel.

By allowing processes to be transmitted over channels, mobility can be introduced to the network. The dynamic nature of channels allows other processes to maintain communications with migrating processes, without the need to reconfigure themselves.

By combining these channels with a computation language it has been shown that an environment for mobile distributed computing can be created. Section 6.6.6 shows a method by which these higher order channels can be implemented.

8.4 Related work

8.4.1 Abstract machine for the π-calculus

An abstract machine for executing π-calculus processes has been developed ((Turner 1995)). This uses a reduced version of the polyadic π-calculus as its input language. Two restrictions have been introduced:

1. Replication is only allowed on input. This restriction makes it easier to spot when replication should be performed, as when the input synchronisation has occurred, a new replicated process can be spawned, allowing further inputs to be performed. With replicated output, it is more difficult to determine how many processes should be created (especially with asynchronous outputs);
2. The summation operator is disallowed. This operator is used to introduce non-determinism to a process, by allowing it to synchronise using one of a set of channels, taking different paths depending on which channel is synchronised. Turner shows that this can be modelled in a separate library and thus does not need to be part of the basic abstract machine.

The language Pict is another method of representing π-calculus processes so that they may be directly executed by computers.

The abstract machine is used to implement a Pict to C compiler. As all processes are pure π-calculus, the communications infrastructure is highly optimised. The abstract machine uses multiple threads of control, and can therefore make effective use of shared memory multiprocessors. There is currently no distributed implementation available.

8.4.2 Higher order channels

In (Muller & May 1998), another implementation of higher order channels is introduced. In this model, at any time only two processes may contain handles to a channel, one to its *input* port, and one to its *output* port (channels in this model are unidirectional).

Once a port (channel end) has been transmitted over a process, the port becomes invalid on the sending process and can no longer be used. This allows each port of a channel to know the exact location of its partner, removing the need for a third party (such as the broker) to be used in order to perform communication over a channel.

A protocol is presented that allows the reliable transmission of ports over channels. It ensures that no messages are lost whilst the port is being transferred and the network reconfigured.

Synchronous communication is used to transfer data over the channels (requiring two messages to flow over the network). The system has been extended ((May & Muller 1998)) to allow asynchronous communication (multimedia streams, for example) over channels. A hybrid synchronous/asynchronous channel is used to spot when it can safely transmit data asynchronously, and when it should return to a synchronous mode — often when channels are being moved between processes.

The system does not deal with the migration of processes over channels, and alternative methods need to be found where the sharing of a channel name between many processes would be an obvious solution (for example, a resource locator process would need to be redesigned).

8.5 Further work

Areas in which the implementation of HOC Scheme could be improved have been described in section 6.7. This section introduces additional areas in which HOC Scheme could be applied. Any changes that may be required to the implementation of HOC Scheme are highlighted.

8.5.1 Applicability of HOC Scheme to agency

HOC Scheme provides many of the features required by a framework for implementing agents. This can be classified into the following sections ((Dale 1997)):

1. Migration. For an agent to be mobile it needs to be able to migrate. It is the agent which decides where and when to migrate;
2. Communicate. Agents need to be able to communicate with other agents so that cooperative agents can be programmed. They also need to be able to communicate with fixed resources on the nodes to which they migrate;
3. Language support. Agents should use a common language. This language should be expressive enough to allow agents that have not previously met to communicate;
4. Security. Controls need to be maintained to ensure that agents do not perform operations where they do not have the required authorisation.

Both the migration and communication requirements are satisfied by the current implementation of HOC Scheme. Agents can be modelled as processes which migrate as closures or continuations over channels, and take channels with them in order that they may be contacted regardless of their location.

These channels can be used to provide a communications network between agents, and between agents and fixed resources found through the resource locator process. If agents use a common protocol to communicate over these channels, agents belonging to different applications will be able to communicate.

A method for allowing initially unconnected agents to be introduced would need to be implemented. This could be achieved using a more dynamic version of the resource locator process, with agents registering when they enter a node, and unregistering when the agent migrates. This would allow other agents to query a node to discover agents which are available to communicate.

Security is an important consideration when implementing agent systems, especially if cost is to be associated with accessing resources. HOC Scheme has no built in support for security at present. Although a security layer could be written in HOC Scheme, a built in security model is likely to be less easy to bypass.

8.5.2 *Mobile networks for multimedia applications*

At present multimedia streams are usually either between two fixed points, or use a multicast network to distribute a stream to many receiving elements. As computers become more mobile, techniques will have to be found for allowing multimedia streams to move with the user.

HOC Scheme currently supports the concept of moving streams of data. The network of channels may be reconfigured dynamically during the lifetime of an application, and the ability of processes to move between nodes is a useful concept for maintaining contact with a mobile user. As arbitrary data may be sent over channels, conceptually, streams of multimedia traffic could be sent over them.

However the implementation is currently not suited to processing multimedia streams. These streams have different properties to the types of data hitherto transmitted over HOC Scheme channels.

1. Real time. Multimedia streams are generally real time. Although a certain amount of buffering can be done at the receiving endpoint to smooth out jitter introduced by networks, the underlying transport must be capable of transferring the data in real time. If it fails to do this, the stream will break down due to buffer underruns;
2. Latency considerations. Different types of stream have differing requirements with regards to latency. For example, a broadcast video stream can have a higher latency than a video conference stream, as it doesn't matter if the observer sees a frame in a broadcast a few seconds after it has been transmitted, but is disconcerting if this frame is part of a two way video conference. In general, it is acceptable to introduce a relatively large amount of latency to a uni-directional stream. This can be introduced by the transmitting ends compression codec sending large packets, requiring it to introduce latency as it fills the data buffer, and by the receiver's decompression codec and buffering data to smooth out network jitter. In bi-directional streams with humans on each end of the link, latency should be kept as low as possible (consider the difference in quality between a transatlantic call over a land line and via a satellite);
3. Lossy. If data is lost in a multimedia stream, in general there is no time to request that it be re-sent. Whilst it may produce a glitch in the stream, the receiving codec should be able to recover and continue processing the remainder of the stream.

In the current implementation of HOC Scheme the broker model would make it very difficult to pass data between endpoints in real time over a channel. For a multimedia stream to be efficient, only one message should flow between nodes for each output over the channel. This implies that the sending process should be able to determine the destination from the channel, rather than relying on the broker to resolve a destination.

The issue of latency could be offloaded to the sending and receiving processes, by allowing them to choose the packet size for transmission, and the buffer size on receipt. Alternatively HOC Scheme could manage latency, given suitable parameters from the sending and receiving process. It could then allow the sender to stream bytes through the channel, performing internal buffering in order to transmit the correct sized packet. On the receiving end, HOC Scheme could ensure that the buffer does not grow too large, due to not having a synchronised clock.

The current use of TCP/IP as the transport for channels would also need reviewing. Whilst this is a suitable transport for non multimedia streams, it can introduce latency to a multimedia stream and might not be able to meet the streams real time requirements. UDP is a more suitable transport. The real time protocol(RTP) uses UDP for the transmission of real time traffic on an internet.

The fact that data can be lost without destroying the multimedia stream could be a boon to the HOC Scheme environment when moving multimedia endpoints between nodes. If the move fails to occur in real time, this would not cause the network to fail. Instead it may cause a glitch in the stream, but it would recover.

The different requirements of the multimedia stream suggest that there should be two classes of channel managed by HOC Scheme nodes.

Multimedia channel — A channel which can solely transmit and receive multimedia traffic between two endpoints. It uses an unreliable network transport and an optimised asynchronous message passing interface. An additional extension could be the ability of a multimedia channel to broadcast a stream to a set of nodes;

Control channel — The traditional HOC Scheme channel, this provides support for reliable communications in a dynamically configurable network. It is a higher order channel, in that processes, control channels and multimedia channels may be transmitted over it.

The two types of channel in combination provide an environment in which multimedia streams may be transmitted between two endpoints, and by using the control channel the streaming network can be reconfigured dynamically.

8.6 Summary

This thesis has presented a model for the coordination of mobile processes. By abstracting away from physical network implementations to a model using channels, the application designer is free to place and migrate processes around the network whilst maintaining connectivity.

The higher order capabilities of the model presents the programmer with a mechanism for controlling the migration of processes (by sending them over channels to another node) and reconfiguration of networks (by sending channels over channels).

Two implementations of the model have been documented, one allowing application designers to investigate process interactions, and the other showing how

the model can be implemented on top of a low level network interface, providing a distributed implementation.

Appendix A

Listing of metacircular HOC Scheme

A.1 The evaluator

```
; ; alt - channel used to communicate with exchange when evalulating alt clauses
; ; k = continuation, e = expression, a = environment
(define (evalx alt k e a)
  (if (pair? e)
      ((case (car e)
 ((if)        eval-if)
 ((case)      eval-case)
 ((cond)      eval-cond)
 ((begin)     eval-begin)
 ((define)    eval-define)
 ((lambda)    eval-lambda)
 ((let)       eval-let)
 ((set!)      eval-set!)
 ((quote)     eval-quote)
 ((and)       eval-and)
 ((or)        eval-or)
 ((load)      eval-load)
 ((input)     eval-input)
 ((alt)       eval-alt)
 (else        eval-apply)) alt k e a)
      (eval-atom alt k e a)))


;(if e c a)
(define (eval-if alt k e a)
  (evalx alt (lambda (alt v)
       (evalx alt k (if v (caddr e) (cadddr e)) a))
 (cadr e) a))
```

```
; (case e ((l1 e)))
(define (eval-case alt k e a)
  (evalx alt (lambda (alt v) (select-case alt k v (cddr e) a))
 (cadr e) a))

; ((l1 e1 e2 ..) (l2 e3 e4 ..))
(define (select-case alt k val cases a)
  (cond ((null? cases) (errorx 'no-case val))
((eq? (caar cases) 'else)
 (eval-forms alt k (cdar cases) a))
((memq val (caar cases))
 (eval-forms alt k (cdar cases) a))
(else (select-case alt k val (cdr cases) a))))


; (cond (c1 e1 e2 ..) (c2 e3 e4 ..))
(define (eval-cond alt k e a)
  (select-cond alt k (cdr e) a))

(define (select-cond alt k conds a)
  (cond ((null? conds) (errorx 'no-cond '()))
((eq? (caar conds) 'else) (eval-forms alt k (cdar conds) a))
(else (evalx alt (lambda (alt v)
   (if v
       (eval-forms alt k (cdar conds) a)
       (select-cond alt k (cdr conds) a)))
     (caar conds) a))))

; (begin e1 e2 e3)
(define (eval-begin alt k e a)
  (eval-forms alt k (cdr e) a))

(define (eval-forms alt k e a)
  (evalx alt (if (cdr e) (lambda (alt v) (eval-forms alt k (cdr e) a)) k)
 (car e) a))


; (define a b) or (define (f a b) ...)
(define (eval-define alt k e a)
  (let ((define-var (if (pair? (cadr e)) (caadr e) (cadr e)))
(define-exp (if (pair? (cadr e))
(append (list 'lambda (define-vars e))
(define-exps e))
(caddr e))))
    (evalx alt (lambda (alt v) (new-binding a define-var v)
       (sendx k alt define-var))
```

```
    define-exp a)))

(define (define-vars e) (cdr (car (cdr e))))
(define (define-exps e) (cddr e))


; (lambda (a b) e1 e2 ...)
(define (eval-lambda alt k e a)
  (sendx k alt (lambda (k alt v*)
 (eval-forms alt k (cddr e) (extend-env a (cadr e) v*)))))


; (let ((a b) (c d)) e1 e2 ...)
(define (eval-let alt k e a)
  (let ((vars (let-vars (cadr e)))
(vals (let-vals alt (cadr e) a)))
    (eval-forms alt k (cddr e) (extend-env a vars vals))))

(define (let-vars bindings)
  (if bindings
      (cons (caar bindings) (let-vars (cdr bindings)))
      '()))

(define (let-vals alt bindings a)
  (if bindings
      (evalx alt (lambda (alt v) (cons v (let-vals alt (cdr bindings) a)))
     (car (cdr (car bindings))) a)
      '()))



; (set! a b)
(define (eval-set! alt k e a)
  (evalx alt (lambda (alt v)
       (set-binding a (cadr e) v)
       (sendx k alt v))
 (caddr e) a))

; (quote a)
(define (eval-quote alt k e a)
  (sendx k alt (cadr e)))

; (and e1 e2 ...)
(define (eval-and alt k e a)
  (eval-and-list alt k (cdr e) a))

(define (eval-and-list alt k l a)
```

```
   (evalx alt (lambda (alt v)
        (if (and (cdr l) v)
    (eval-and-list alt k (cdr l) a)
    (sendx k alt v)))
  (car l) a))

; (or e1 e2 ...)
(define (eval-or alt k e a)
  (eval-or-list alt k (cdr e) a))

(define (eval-or-list alt k l a)
  (evalx alt (lambda (alt v)
       (if  (or v (null? (cdr l)))
    (sendx k alt v)
    (eval-or-list alt k (cdr l) a)))
 (car l) a))

; (load "file")
(define (eval-load alt k e a)
  (let ((h (open-file (cadr e) "r")))
    (read-forms alt k a h)))

(define (read-forms alt k a h)
  (let ((val (read h)))
    (if (eof-object? val)
(begin (close-port h) (sendx k alt 'ok))
(evalx alt (lambda (alt v) (read-forms alt k a h)) val a))))


; (input c (a b c ...) e1 e2 ...)
; Evaluate channel expression and pass to input function
(define (eval-input alt k e a)
  (evalx alt (lambda (alt v)
       (inputx k alt v (caddr e) (cdr (cddr e)) a))
 (cadr e) a))

; (alt ((e1 e2 ..) (e3 e4 ..)))
; Evaluate each branch with alt set to a channel, and call the exchange to
; handle protocol
(define (eval-alt alt k e a)
  (let ((alt-chan (make-channel)))
    (altx alt-chan (eval-alt-branches alt-chan k (cdr e) a))))

(define (eval-alt-branches alt k branches a)
  (if (null? branches)
      '()
      (cons (eval-forms alt k (car branches) a)
```

```
    (eval-alt-branches alt k (cdr branches) a))))

;Atom
(define (eval-atom alt k e a)
  (if (symbol? e)
      (sendx k alt (lookup-binding a e))
      (sendx k alt e)))


;(f a b c ...)
(define (eval-apply alt k e a)
  (eval-list alt (lambda (alt v*) (applyx alt k (car v*) (cdr v*)))
     e a))

(define (eval-list alt k e a)
  (if e
      (eval-list alt (lambda (alt1 v1)
       (evalx alt1 (lambda (alt2 v2)
      (sendx k alt2 (cons v2 v1)))
       (car e) a))
 (cdr e) a)
      (sendx k alt '())))

(define (applyx alt k f v*)
  (yield)   ;;try and stop stack from blowing
  (f k alt v*))

(define (sendx k alt v) (k alt v))


;;-------------------------------------------------
;;Environment handling
;;-------------------------------------------------

;;Environment modelled as a list of frames
;;Frame modelled as a list of bindings
;;Binding modelled as a pair (name . value)

;;New frame
(define (extend-env a vars vals)
  (cons (pairlist vars vals) a))

(define (pairlist vars vals)
  (cond ((symbol? vars) (list (cons vars vals)))
(vars (cons (cons (car vars) (car vals))
    (pairlist (cdr vars) (cdr vals))))
(else '())))
```

```
;;New binding
(define (new-binding a var val)
  (let ((binding (assq var (car a))))
    (if binding
(set-cdr! binding val)
(set-car! a (cons (cons var val) (car a))))))

;;Modify existing binding
(define (set-binding a var val)
  (let ((binding (retrieve-binding a var)))
    (if binding
(set-cdr! binding val)
(errorx 'no-binding var))))

;;Lookup value of binding
(define (lookup-binding a var)
  (let ((binding (retrieve-binding a var)))
    (if binding
(cdr binding)
(errorx 'no-binding var))))

;;Fetch binding from the environment
(define (retrieve-binding a var)
  (if a
      (let ((binding (assq var (car a))))
(if binding
    binding
    (retrieve-binding (cdr a) var)))
      '()))

(define (errorx mess val)
  (format #t "ERROR: ~a ~a~%" mess val)
  '())


;;-------------------------------------------------
;;Top level environment
;;-------------------------------------------------

;;Top level environment
(define top-frame '())

;;Add a k primitive to the top level
(define (kprimitive name f)
  (set! top-frame (cons (cons name f) top-frame))
  name)
```

```
;;Add a standard primitive to the environment
(define (primitive name f)
  (set! top-frame (cons (cons name (lambda (k alt v*)
     (sendx k alt (apply f v*))))
top-frame))
  name)

;;Add the primitives
(primitive '+ +)
(primitive '- -)
(primitive '* *)

(primitive 'format format)
(primitive 'read   read)

(primitive 'cons cons)
(primitive 'car  car)
(primitive 'cdr  cdr)
(primitive 'cadr cadr)
(primitive 'cddr cddr)
(primitive 'cdar cdar)
(primitive 'caar caar)
(primitive 'caddr caddr)
(primitive 'caadr caadr)
(primitive 'cadddr cadddr)

(primitive 'pair? pair?)
(primitive 'append append)
(primitive 'list list)
(primitive 'set-car! set-car!)
(primitive 'set-cdr! set-cdr!)
(primitive 'length length)

(primitive 'open-file open-file)
(primitive 'close-port close-port)
(primitive 'eof-object? eof-object?)
(primitive 'display  display)

(primitive 'assq assq)
(primitive 'memq memq)
(primitive 'eq?  eq?)
(primitive 'equal? equal?)
(primitive 'null? null?)
(primitive 'symbol? symbol?)
(primitive '= =)
(primitive '> >)
```

```
(primitive 'not not)
(primitive 'remainder remainder)

;;The only k primitive needed. Makes apply a binary function.
(kprimitive 'apply  (lambda (k alt v*)
      (applyx alt k (car v*) (cadr v*))))

;;Use host HOC Scheme scheduler
(primitive 'spawn  (lambda (thunk)
     (spawn (lambda () (applyx #f (lambda (alt v) v)
       thunk '())))
     'ok))
(primitive 'quiet-spawn  (lambda (thunk)
     (quiet-spawn (lambda () (applyx #f (lambda (alt v) v)
       thunk '())))
     'ok))
(primitive 'yield yield)


;;-------------------------------------------------
;;Read eval print loop
;;-------------------------------------------------
(define env '())
(define win #t)

(define (read-eval-print alt v)
  (yield)
  (display v)
  (format #t "~%REP? ")
  (evalx alt read-eval-print (read) env))

(define (start)
  (set! env (list top-frame))
  (read-eval-print #f "Hello!"))
```

A.2 The channels implementation

```
(define (exchange in out new cancel inputs outputs id)
  (alt ((input in (cin rin nargs)
  (exchange-in in out new cancel inputs outputs id (makeq cin rin nargs))))
       ((input out (cout rout nargs)
  (exchange-out in out new cancel inputs outputs id (makeq cout rout nargs))))
       ((output new (list in out cancel id))
  (exchange-new-chan in out new cancel inputs outputs id))
       ((input cancel (c r nargs)
```

```
            (exchange-cancel in out new cancel inputs outputs id (makeq c r nargs))))))


 (define (exchange-in in out new cancel inputs outputs id cin)
   (let ((cout (match outputs cin)))
     (if cout
 (begin
   (rendezvous cin cout)
   (exchange in out new cancel inputs (remove cout outputs) id))
 (exchange in out new cancel (cons cin inputs) outputs id))))

 (define (exchange-out in out new cancel inputs outputs id cout)
   (let ((cin (match inputs cout)))
     (if cin
 (begin
   (rendezvous cin cout)
   (exchange in out new cancel (remove cin inputs) outputs id))
 (exchange in out new cancel inputs (cons cout outputs) id))))

 (define (exchange-new-chan in out new cancel inputs outputs id)
   (exchange in out new cancel inputs outputs (+ id 1)))

 (define (exchange-cancel in out new cancel inputs outputs id c)
   (exchange in out new cancel
     (remove c inputs)
     (remove c outputs) id))


;;-------------------------------------------------
;;The rendezvous mechanism
;;-------------------------------------------------

(define (rendezvous qin qout)
  (quiet-spawn (lambda () (rendezvous-input qin qout (make-channel)))))

(define (rendezvous-input qin qout newc)
  (output (qreply qin) 'input? (qchan qin) (qargs qin) newc)
  (input newc  (r)
    (if r
(rendezvous-output qin qout newc)
(begin
  (output (exout (qchan qout)) (qchan qout) (qreply qout)
  (qargs qout))
  "rendezvous-failed"))))


(define (rendezvous-output qin qout newc)
```

```
   (output (qreply qout) 'output? (qchan qout) (qargs qout) newc)
   "rendezvous-complete")

;;Aux functions
(define (makeq qchan qreply qargs) (list qchan qreply qargs))
(define qchan car)
(define qreply cadr)
(define qargs  caddr)

(define exin car)
(define exout cadr)
(define excancel caddr)
(define exid cadddr)


(define (match l q)
  (cond ((null? l) #f)
((and (equal? (qchan q) (qchan (car l)))
      (= (qargs q) (qargs (car l)))
      (not (equal? (qreply q) (qreply (car l))))) (car l))
(else (match (cdr l) q))))

(define (remove a l)
  (cond ((null? l) '())
((equal? (car l) a) (cdr l))
(else (cons (car l) (remove a (cdr l))))))


;;-------------------------------------------------
;;Output protocol
;;Send exchange channel and number of args
;;Wait for reply from exchange giving input process
;;And send args to input process
;;-------------------------------------------------

(define (koutputx k alt v*)
  (let ((chan (car v*))
(args (cdr v*)))
    (if alt
(alt-output k alt chan args)
(standard-output k alt chan args))))

(define (standard-output k alt chan args)
  (let ((reply (make-channel)))
    (output (exout chan) chan reply (length args))
    (input reply (ready chan nargs sendto)
   (output sendto args))
```

```
      (sendx k alt 'output)))

 (define (alt-output k alt chan args)
   (output (exout chan) chan alt (length args))
   (make-altout chan (length args) k args))


;;-------------------------------------------------------
;; Input protocol
;; Send exchange channel and number of args
;; Wait to receive vals from output process
;; Evaluate body of input function with args bound to values
;;-------------------------------------------------------

 (define (inputx k alt chan args body a)
   (if alt
       (alt-input k alt chan args body a)
       (standard-input k alt chan args body a)))

 (define (standard-input k alt chan args body a)
   (let ((reply (make-channel)))
     (output (exin chan) chan reply (length args))
     (input reply (ready chan nargs new-reply)
       (output new-reply #t)
       (input new-reply (data)
         (if data
     (eval-forms alt k body (extend-env a args data))
     (standard-input k alt chan args body a))))))

(define (alt-input k alt chan args body a)
  (output (exin chan) chan alt (length args))
  (make-altin chan (length args) k args body a))


;;-------------------------------------------------------
;; Alt protocol
;;-------------------------------------------------------

(define (altx reply alts)
  (input reply (ready chan nargs nc)
    (case ready
      ((output?) (altx-output nc (altmatch 'output chan nargs alts)
      reply alts))
      ((input?)  (altx-input  nc (altmatch 'input  chan nargs alts)
      reply alts)))))

(define (altx-output nc out reply alts)
```

```
   (output nc (alt-args out))
   (alt-decline reply alts)
   (sendx (alt-k out) #f 'output))

 (define (altx-input nc in reply alts)
   (output nc #t)
   (input nc (data)
     (cond (data
    (alt-decline reply alts)
    (eval-forms #f (alt-k in) (altin-body in)
        (extend-env (altin-a in) (alt-args in) data)))
   (else
    (output (exin (alt-chan in)) (alt-chan in) reply (alt-nargs in))
    (altx reply alts)))))

 (define (alt-decline reply alts)
   (quiet-spawn (lambda () (_alt-decline reply alts))))

 (define (_alt-decline reply alts)
   (if alts
     (alt ((input reply (ready chan nargs nc)
  (output nc #f)
  (_alt-decline reply alts)))
  ((output (excancel (alt-chan (car alts)))
   (alt-chan (car alts))
   reply
   (alt-nargs (car alts)))
   (_alt-decline reply (cdr alts))))
     #f))


(define (altmatch direction chan nargs alts)
  (if (null? alts)
      '()
      (let ((alt (car alts)))
(if (and (eq? (alt-direction alt) direction)
 (equal? (alt-chan alt) chan)
 (= (alt-nargs alt) nargs))
    alt
    (altmatch direction chan nargs (cdr alts))))))

;;Alt data structures
(define (make-altout . outs) (cons 'output outs))
(define (make-altin . ins) (cons 'input ins))

(define alt-direction car)
(define alt-chan cadr)
```

```
(define alt-nargs caddr)
(define alt-k cadddr)
(define (alt-args a) (cadddr (cdr a)))
(define (altin-body a) (cadddr (cddr a)))
(define (altin-a a) (cadddr (cddr (cdr a))))


;;-----------------------------------------------------------
;; Integrate with evaluator
;;-----------------------------------------------------------

(define *new-chan* (make-channel))

(define (make-new-channel)
  (input *new-chan* (c) c))

(kprimitive 'output koutputx)
(primitive 'make-channel make-new-channel)

; Start an exchange process
(quiet-spawn
 (lambda () (exchange (make-channel) (make-channel) *new-chan* (make-
channel)
       '() '() 0)))
```

Bibliography

Abadi M & Gordon A (1997). A calculus for cryptographic protocols: the spi calculus, *in Proceedings of the Fourth ACM Converence on Computer and Communications Security*, pp. 36–47.

Abelson H, Sussman G J & Sussman J (1985). *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Mass.

Adams N & Rees J (1988). Object-Oriented Programming in Scheme, *in Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pp. 277–288.

Adl-Tabatabai A R, Langdale G, Lucco S & Wahbe R (1996). Efficient and language-independent mobile programs, *in Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PDLI)*, pp. 127–136.

Agha G A (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Massachusetts.

Allison L (1988). Some Applications of Continuations, *The Computer Journal* **31**(1), 9–11.

Allison L (1990). Continuations Implement Generators and Streams, *The Computer Journal* **33**(5), 460–465.

American National Standards Institute (1992). *Database Language SQL*.

ANSI X3.226-1994 (1994). *Information Technology — Programming Language — Common Lisp*. ANSI Standard.

Arbab F (1995). Coordination of massively concurrent activities, Technical Report CS-R9565, Centrum voor Wiskunde en Informatica (CWI), Department of Software Engineering, Kruislaan 413, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands.

Barnes J G P (1989). *Programming in Ada*, International Computer Science Series, Addison Wesley.

Bartlett J F (1989). SCHEME->C, A Portable Scheme-to-C Compiler, Technical Report WRL 89/1, Digital Western Research Laboratory.

Batey D J & Padget J A (1993). Towards a Virtual Multicomputer, *in Proccedings of the workshop on Heterogenous Processing*, IEEE, IEEE.

Bell J R (1973). Threaded Code, *Communications of the ACM* **16**(6).

Berrington N (1990). Building abstractions with the EuLisp parallel primitives, Technical Report CSTR 92–13, Department of Electronics and Computer Science, University of Southampton.

Berrington N, Broadbery P, De Roure D C & Padget J A (1993). EuLisp Threads: A Concurrency Toolbox, *Lisp and Symbolic Computation* **6**(1-2).

Berrington N, DeRoure D, Greenwood M & Henderson P (1993). Modelling Organisational Processes, *in Department of Electronics and Computer Science Research Journal*, University of Southampton.

Berrington N, DeRoure D, Greenwood M & Henderson P (1994). Distribution and Change:Investigating Two Challenges for Process Enactment Systems, *in* B C Warboys, ed., *Third European Workshop, EWSPT'94, Lecture Notes in Computer Science 772*, Springer-Verlag, pp. 152–162.

Berrington N, Deroure D & Padget J A (1993). Guaranteeing Unpredictability, *The computer journal* **36**(8), 723–733.

Berry G & Boudol G (1992). The chemical abstract machine, *Theoretical Computer Science* **96**, 217–248.

Bershad B N, Zekauskas M J & Sawdon W A (1993). The Midway Distributed Shared Memory System, Technical Report CMU-CS-93-119, Department of Computer Science, Carnegie Mellon University.

Birman K P (1993). The Process Group Approach to Reliable Distributed Computing, *Communications of the ACM* **36**(12), 37–53,103.

Birrell A D (1991). An Introduction to programming with threads, *in* G Nelson, ed., *Systems Programming with Modula-3*, Prentice-Hall, Englewood Cliffs, NJ, chapter 4, pp. 88–118.

Bisiani R & Forin A (1998). Multilanguage Parallel Programming of Heterogeneous Machines, *IEEE Transactions on Parallel and Distributed Systems* **37**(8), 930–945.

Bonzon P E (1990). A Metacircular Evaluator for a Logical Extension of Scheme, *Lisp and Symbolic Computation* **3**, 113–133.

Boudol G (1997). The Pi-calculus in Direct Style, *in Program Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Braden R, Zhang L, Berson S, Herzog S & Jamin S (1997). Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification, Technical Report RFC 2205, Internet Engineering Task Force.

Brand H, Dassler K, Schneider T, Cengarle M C, Mandel L & Wirsing M (1992). *An approach to the DIN Kernel Lisp Definition*. ISO input document ISO/IEC JTC 1/SC 22/WG 16 LISP N 96.

Bretthauer H, Davis H, Kopp J & Playford K (1992). Balancing the EuLisp Metaobject Protocol, *in Reflection and Metalevel Architecture*, Proc. of the International Workshop on New Models for Software architecture, pp. 113–118.

Bretthauer H, Davis H, Kopp J & Playford K (1993). Balancing the EuLisp Metaobject Protocol, *Lisp and Symbolic Computation* **6**(1-2).

Broadbery P & Burdorf C (1993). Applications of Telos, *Lisp and Symbolic Computation* **6**(1-2).

Cardelli L & Gordon A D (1998). Mobile Ambients, *in* M Nivat, ed., *Proceedings of the First International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '98), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'98), (Lisbon, Portugal, March/April 1998)*, Vol. 1378 of *lncs*, sv, pp. 140–155.

Carriero N & Gelernter D (1989). Linda in Context, *Communications of the ACM* **32**(4), 444–458.

Carriero N & Gelernter D (1992). Coordination languages and their significance, *Communications of the ACM* **35**(2), 97–107.

Carriero N & Gelernter D (1998). Applications experience with Linda, *in Proceedings of the ACM Symposium on Parallel Programming*.

Ceftin H, Jagannathan S & Kelsey R (1995). Higher Order Distributed Objects, *ACM Transactions on Programming Languages and Systems* **17**(5), 704–739.

Chandy K M & Misra J (1988). *Parallel program design: a foundation*, Addison-Wesley Publishing Company, Inc.

C.J.Fidge (1998). A LISP Implementation of the Model for Communicating Sequential Processes, *Software–Practice and Experience* **18**(10), 923–943.

Clinger. W & Rees J (1991). The Revised^4 Report on Scheme, *Lisp Pointers* **4**(3).

Cooper E C & Draves R P (1988). C Threads, Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University.

Dale J (1997), A Mobile Agent Architecture for Distributed Information Management, PhD thesis, Department of Electronics and Computer Science, University of Southampton.

DeRoure D C (1991). Parallel Implementation of UNITY, *in The PUMA and GENESIS Projects*, pp. 67–75.

Diaz M, Rubio B & Troya J M (1996). Distributed Programming with a Logic Channel Based Coordination Model, *The computer journal* **39**(10), 876–889.

Dijkstra E W (1965). Solution of a problem in concurrent programming control, *Communications of the ACM* **8**(9), 569.

Dourish P (1992). Computational Reflection and CSCW Design, Technical Report EPC-92-102, Rank Xerox EuroPARC.

Falcone J R (1987). A Programmable Interface Language for Heterogenous Distributed Systems, *ACM Transactions on Computer Systems* **5**(4), 330–351.

FDR (1983). *Failures Divergence Refinement.*

Filman R E & Friedman D P (1984). *Coordinated Computing: Tools and Techniques for Distributed Software*, McGraw-Hill Computer Science Series, McGraw-Hill.

Flynn M (1972). Some computer organisations and their effectivness, *IEEE Transactions on Computers* **21**(9), 948–960.

Foster I, Kesselman C & Tuecke S (1994). The Nexus task-parallel runtime system, *in Proceedings of the First International Workshop on Parallel Processing*.

Fournet C, Gonthier G, Levy J J, Maranget L & Remy D (1996). A calculus of mobile agents, *in Proceedings of Seventh International Conference on Concurrency Theory*.

Gabriel R P & McCarthy J (1984). Queue-based Multi-processing Lisp, *in Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 25–43.

Geist A, Beguelin A, Dongarra J, Manchek R & Sunderam V (1994). *PVM: Parallel Virtual Machine*, MIT Press.

Goldman R & Gabriel R P (1988*a*). Preliminary Results with the Initial Implementation of QLisp, *in Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pp. 143–152.

Goldman R & Gabriel R P (1988*b*). Qlisp: Experience and New Directions, *in Parallel Programming: Experiences with Applications, Languages and Systems*, ACM, pp. 111–123.

Gosling J (1995). Java Intermediate Bytecodes, *ACM SIGPLAN Notices* **30**(3), 111–118.

Gupta A, Akyildiz I F & Fujimoto R M (1991). Performance Analysis of Time Warp with Multiple Homogeneous Processors, *IEEE Transactions on Software Engineering* **17**(10), 1013–1027.

Haines M, Mehrotra P, Rosendale J V & Chapman B (1996). Opus: A Coordination Language for Multidisciplinary Applications, *Scientific Computing* .

Halls D (1997), Applying Mobile Code to Distributed Systems, PhD thesis, Computer laboratory, University of Cambridge.

Halstead R H (1985). Multilisp: A Language for Concurrent Symbolic Computations, *ACM Transactions on Programming Languages and Systems* **7**(4), 501–538.

Henderson P (1993), Reconfigurable Dining Philosophers using Higher Order Communications. unpublished, URL http://www.ecs.soton.ac.uk/.

Hewitt C E (1979). Control structures as patterns of passing messages, *in* P H Winston & R H Brown, eds, *Artificial Intelligence: An MIT Perspective (Volume 2)*, MIT Press Series in Artificial Intelligence, MIT Press, pp. 435–465.

Hewitt C E & Baker H (1977). Actors and continous functionals, *in 1977 IFIP Conference Proceedings*, North Holland, pp. 987–992.

Hewitt C, Rienhardt T, Agha G & Attardi G (1984). Linguistic support of receptionists for shared resources, *in* S.D.Brooks, A.W.Roscoe & G.Winskel, eds, *Seminar on Concurrency, Lecture Notes in Computer Science 197*, pp. 330–359.

Hilditch S & Thomson T (1993). Distributed Detection of Deadlock, *ICL Technical Journal* .

Hoare C A R (1985). *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science, Prentice-Hall, London.

ISO/IEC 13816:1997 (1997). *Information technology — Programming languages, their environments and system software interfaces – Programming language ISLISP*. ISO standard.

Jagannathan S & Philbin J (1992). A Foundation for an Efficient Multi-Threaded Scheme System, *in Proceedings of 1992 ACM Conference on Lisp and Functional Programming*, ACM Press, pp. 345–357.

Jefferson D R (1985). Virtual Time, *ACM Transactions on Programming Languages and Systems* **7**(3), 404–425.

Jennings N R (1995). Agent Software, *in Proceedings of the UNICOM Seminar on Agent Software*, London, UK, pp. 12–27.

John R & Ahamad M (1993). Causal Memory: Implementation, Programming Support and Experiences, Technical Report GIT-CC-93-10, College of Computing, Georgia Institute of Technology.

Kale L V & Bhandarkar M (1996). Structured Dagger: A Coordination Language for Message-Driven Programming, *in Proceedings of Second International Euro-Par Conference, Lecture Notes in Computer Science*, Springer-Verlag, pp. 646–653.

Kale L V, Ramkumar B & A. B. Sinha A G (1994). The Charm Parallel Programming Language and System: Part I — Description of Language Features, *IEEE Transactions on Parallel and Distributed Systems* .

Kelsey R & Rees J (1995). A Tractable Scheme Implementation, *Lisp and Symbolic Computation* **7**(4).

Kiczales G (1992). Towards a New Model of Abstraction in Software Engineering, *in Proceedings of the International Workshop on New Models for Software Architecture — Reflection and Metalevel Architecture*, pp. 1–11.

Kiczales G, des Rivieres J & Bobrow D (1991). *The Art of the Metaobject Protocol*, MIT Press, Cambridge, Massachusetts.

K.M.Chandy & J.Misra (1981). Asynchronous Distributed Simulation via a Sequence of Parallel Computations, *Communications of the ACM* **24**(11), 198–206.

Kranz D, Johnson K, Agarwal A, Kubuatowicz J & Lim B H (1992). Integrating Message-Passing and Shared-Memory: Early Experience, Technical Report 478, MIT Laboratory for Computer Science.

Kung H T & Robinson J T (1981). On Optimistic Methods for Concurrency Control, *ACM Transactions on Database Systems* **6**(2), 213–226.

Lamport L (1978). Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM* **21**(7), 558–565.

Liskov B, Herlihy M & Gilbert L (1986). Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing, *in Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 150–159.

May D & Muller H L (1998). Using Channel for Multimedia Communication, Technical Report CSTR-98-002, Department of Computer Science, University of Bristol.

McCarthy J (1960). Recursive Functions of Symbolic Expressions and their computation by machine – Part I, *Communications of the ACM* **3**(1), 184–195.

McCarthy J (1981). History of Lisp, *in ACM Conference on the History of Programming Languages (HOPL)*, pp. 173–197.

Milner R (1991). The Polyadic π-Calculus: a Tutorial, *in The Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktobberdorf. modified version of Report ECS-LFCS-91-180 from LFCS Edinburgh.

Milner R (1993). Elements of Interaction, *Communications of the ACM* **36**(1), 78–89.

Mohr E (1991), Dynamic Partitioning of Parallel Lisp Programs, PhD thesis, Yale University.

MPI Forum (1994). MPI: A Message Passing Interface Standard, *International Journal of Supercomputer Applications* **8**(3/4), 164–416.

Muller H L & May D (1998). A Simple Protocol to Communicate Channels over Channels, Technical Report CSTR-98-001, Department of Computer Science, University of Bristol.

Occ (1988). *Occcam-2 Reference Manual.*

Padget J A, Nuyens G & Bretthauer H (1993). An Overview of EuLisp, *Lisp and Symbolic Computation* **6**(1-2).

Perkins C (1997). Mobile IP, *IEEE Computer* pp. 84–89.

Peterson J L (1977). Petri Nets, *Computing Surveys* **9**(3), 223–252.

Philbin J (1992). Scheduling policy management in Sting, *in Workshop Proceedings on Parallel Symbolic Computing: Languages, Systems and Applications.*

Postel J B (1982). Simple Mail Transfer Protocol, Technical Report RFC 821, Internet Engineering Task Force.

Pountain D & May D (1988). *A tutorial introduction to Occam programming*, BSP Books, London.

Queinnec C (1990). PolyScheme : A Semantics for a Concurrent Scheme, *in Workshop on High Performance and Parallel Computing in Lisp*, EuroPal '90 – European Conference on Lisp and its Practical Applications, Twickenham (UK).

Queinnec C (1993). Designing MEROON V3, *in* C Rathke, J Kopp, H Hohl & H Bretthauer, eds, *Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP'93 Workshop*, number 788, Sankt Augustin (Germany).

Queinnec C & DeRoure D C (1992). Design of a Concurrent and Distributed Language, *in Workshop Proceedings on Parallel Symbolic Computing: Languages, Systems and Applications.*

RMI (1996). *Java Remote Method Invocation Specification.*

Rosenberry W, Kenny D & Fisher G (1992). *Understanding DCE*, O'Reilly and Associates, Inc.

Sabot G W (1988). *The Paralation Model*, MIT Press, Cambridge, Massachusetts.

Siegel J (1996). *CORBA fundementals and programming*, Wiley.

Stamos J W & Gifford D K (1990). Implementing remote evaluation, *IEEE Trans. on Software Engineering* **16**(7), 710–722.

Steele Jr G L (1982). An Overview of Common LISP, *in Conference record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 98–107.

Steele Jr G L (1984). *Common Lisp the Language*, Digital Press. Second edition, Digital Press, 1990.

Steele Jr G L & Gabriel R P (1993). The Evolution of Lisp, *SIGPLAN Notices* **28**(3), 231–270.

Sun (1988). Lightweight Processes Tutorial, *in System Services Overview (SunOS 4.0)*, Sun MicroSystems, chapter 6, pp. 71–106. Part Number: 800-1753-10.

Sun Microsystems (1987). XDR: External Data Representation Standard, Technical Report RFC 1014, Internet Engineering Task Force.

Sunderman V (1990). PVM: A framework for parallel distributed computing, *Concurrency: Practise and Experience* .

Taft E & Walden J (1995). *Postscript Langauge Reference Manual*, Addison Wesley.

Tinker P & Katz M (1988). Parallel execution of sequential Scheme with ParaTran, *in Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*.

Turner D N (1995), The Polymorphic Pi-calculus: Theory and Implementation, PhD thesis, University of Edinburgh.

Waldo J (1998). *Jini Architecture Overview*, Sun MicroSystems.

Wilkinson T, Stiemerling T, Osmon P, Saulsbury A & P.Kelly (1992). Angel: a proposed multiprocessor operating system, Technical Report TCU/CS/1992/10, Department of Computer Science, City University, UK.

Wright G R & Stevens W R (1995). *TCP/IP Illustrated, Volume 2: The Implementation*, Professional computing series, Addison-Wesley.

W.R.Stevens (1990). *UNIX Network Programming*, Prentice-Hall.

Yao C (1994), Representing Control in Parallel Applicative Programming, PhD thesis, Department of Computer Science, New York University.

Yokote Y, Teraoka F & Tokoro M (1989). A Reflective Architecture for an Object-Oriented Distributed Operating System, Technical Report SCSL-TR-89-001, Sony Computer Science Laboratory Inc.

Yuasa T, Umemura K, Hashimoto Y & Ito T (1992). *An Overview of the Kernel Language for Long-term ISLisp design*. ISO input document ISO/IEC JTC 1/SC 22/WG 16 LISP N 62.

Zorn B, Ho K, Larus J, Semenzato L & Hilfinger P (1989). Multiprocessing Extensions in Spur Lisp, *IEEE Software* .