Applying Mobile Code to Distributed Systems

David Alan Halls

Computer Laboratory
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

June 1997

Abstract

Use of mobile code can make distributed systems and the abstractions they provide more flexible to build and use.

Richer functionality can be given to the interaction between processes by allowing code to be sent between them. More convenient, application-level operations can be made over a network. By making higher order language features transmissible, distributed components can be tightly bound together when they communicate. At the same time, familiar distributed systems can be built using mobile code.

Mobile code can make distributed systems adaptable to application needs. Rather than fixing the interface to a resource and the pattern of interaction with it, a minimal interface can be defined and code implementing higher-level interfaces placed alongside it as and when required. These higher-level interfaces can be application-specific, allowing for interaction patterns that were unknown at the time the resource was made available. Sending code close to a resource can also reduce network usage because the point of interaction with it moves.

The combination of document markup supporting hypertext and a language supporting state-saving allows for stateful client-server sessions with stateless servers and lightweight clients. Putting dormant mobile code in documents provides an alternative to holding knowledge of application functionality on a server machine or running arbitrary code on a client machine.

Mobile code helps to support user mobility. Personalised environments that support state-saving can follow a user between computers. Heterogeneous state-saving allows a user's programs to be relocated between computers. By using a mobile code system with language support for state-saving, applications can direct arbitrary component migration without priming program servers with specific support.

In summary, this dissertation supports the thesis that mobile code can be used to enhance distributed systems.

To Mum, Dad, Jackie and Nan

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

No part of this dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed sixty thousand words, including diagrams, footnotes and bibliography.

Acknowledgements

I would like to thank my supervisor, Jean Bacon, for her invaluable support, encouragement and constructive criticism. I am also grateful to other members of the Computer Laboratory, in particular John Bates, Tim Mills and Sean Rooney, for their valuable advice and helpful discussions.

I would like to thank my parents for their moral and financial support while I have been studying.

This work was supported by a studentship from the Engineering and Physical Sciences Research Council.

Publications

Some of the work described in this dissertation has been published [**Bates96**, **Halls96**, **Halls98**, **Bacon97**].

Contents

Chapter 1

Introduction

1.1 Distributed Systems

A distributed system consists of a number of computers that can send data to each other via a network. This dissertation assumes that the hardware required for such an arrangement is in place and that operating system and communications software which provides reliable transport of data between applications on different machines is present.

Software at a higher level provides convenient abstractions to applications for communicating with each other. Examples are remote procedure call and distributed object systems. This dissertation discusses flexible provision of such interaction by exchange of messages that are programs. This is known as mobile code and can help dynamically to change levels of abstraction and to change where interaction takes place.

1.2 Mobile Code

Mobile code is data that can be executed as a program. The code can be pre-compiled for immediate execution on the recipient's processor, compiled upon receipt for subsequent execution or interpreted.

A simple classification for mobile code systems is:

Simple distribution Code moves once to its destination and runs there until it finishes executing.

Explicit dispatch Code creates new program text for sending elsewhere and possibly inserts data values from itself into it.

Migration Executing code is frozen, sent elsewhere and restarted.

Higher-order Programs send arbitrary closures and continuations[1], together with their closing environments, elsewhere for execution.

This dissertation describes the design, implementation and use of a higher-order mobile code system.

1.3 Research Motivation

Distributed systems are increasingly making compliance with standards and inter-operability their main features. This dissertation argues that they should also provide facilities for dynamically changing where applications execute and the way they communicate over a network. It should be possible to define a minimal communication method that supports this and at the same time allows both for inter-operability and for different abstractions to be defined on top of it. There is no single means of abstracting application-level communication. Existing systems provide inter-operability at too high a level of abstraction.

There is a growing body of research into mobile code. The results from this should be applied to building remote procedure call and distributed object systems. There are novel ways in which existing distributed systems can benefit from the use of mobile code too.

1.4 Research Statement

The research described in this dissertation has produced a mobile code system that supports the transmission of higher-order functions and continuations over computer networks.

The mobile code system has been used to build a distributed processing environment that is flexible in the communication abstractions it provides to applications. Varying degrees of abstraction are provided, from basic message passing to remote method invocation and from tight integration with application state to object-based data hiding. A distributed object abstraction has been written which allows an application to define, offer and adapt at run-time the means that clients should use to communicate with it.

The mobile code system has been used to enhance existing distributed applications. It has been applied to ATM network control, to the management of state in client-server interaction and to event-driven user mobility. Details of these experiments are given in this dissertation.

[1] A closure is a function together with its defining scope. A continuation is a closure that represents the state of execution.

1.5 Boundaries of Research

This research does not address the following issues:

Distributed lexical scope If a mobile code system supports this, then when a piece of code moves, access to location-dependent data is transparently mapped back to the originating site. This can cause network access to occur that is not explicit in the program text.

Distributed garbage collection Extending automatic memory management from a single site to multiple sites enables data not being used on any site to be freed up automatically. Algorithms for doing this are often complex and add to network usage. The research described here does not rely on the availability of distributed garbage collection.

Security Only very basic security, to execute mobile code in a restricted environment, is discussed in this dissertation. This is not adequate for a production system. Wider consideration should be given to security amongst services in distributed systems. The following issues are especially important for mobile code:

- The recipient of a piece of mobile code will want to deny it access to sensitive resources. There should be operating system support for this [**Zakinthios97**] to help avoid naive implementation of security in user space [**McGraw96**, **Felten97**] and to provide a flexible system that is secure from the lowest level upwards. Current practice involves users having to trust software vendors that their code is safe; this applies as much to applications installed from CD-ROM as it does to mobile code. Moving towards lower-level support for restricted environments of execution would allow more software to be run more securely [**Adl-Tabatabai96**].
- A piece of mobile code might be sensitive to third-party intervention. That is, one might trust the recipient but not the network. This applies in general to data exchanged between hosts, not just to mobile code. Encryption can help the recipient to check that the integrity and security of the data has been maintained. A public key system can be used for authentication purposes.
- The sender of a piece of mobile code might not trust its recipient. This is a very difficult problem that has only just begun to be addressed [**Minsky96**]. It is important for code that carries negotiation strategies around with it, for example. A related problem is ensuring that services perform the tasks for which they were contracted. This is important when the service is an

execution platform for mobile code — it might not execute completely or provide bogus results, for instance — although auditing of services is important for distributed systems in general.

This dissertation does not address security for mobile code. There is considerable research to be done in this area. Mobile code contracts and offers external services and so must not implement security in isolation. Some recent work [**Farmer96a**, **Farmer96b**] has addressed the notion of trust in mobile code systems. Other work [**DARPA97**, **Cardelli97**] has started to examine fundamentals of mobile code security. [**Hayton96**] discusses more general aspects of high-level access control amongst services in distributed systems.

Language support for communication This dissertation describes applications that require a program to manage only the connection to its destination when it moves. The method for doing this is different for each application and is made available through separate communication libraries. The applications are also restricted in that coordination is only required between two processes at any one time. For these reasons, use is not made of channel-based languages (for example the distributed π calculus [**Sewell97**], the distributed join-calculus [**Fournet96**] and (modified) Facile [**Knabe95**]) which provide a formal model of mobile, concurrent and distributed processes that carry connections with them as they move. Further research is needed into criteria which determine when a channel-based language is required.

1.6 Outline of Dissertation

Related work is reviewed in each chapter.

Chapter 2 talks about the implementation of a mobile code system. It is used to prototype ideas discussed in later chapters. Each of Chapters 3-7 contains a description of experiments carried out with this mobile code system. Chapter 2 gives some detail to the techniques used in building it.

Chapter 3 presents a method for making closures and continuations transmissible over a network. It supports heterogeneity and is independent of its host language implementation. This chapter shows how an explicit-dispatch mobile code system, such as the one described in Chapter 2, can be made into a higher-order one.

Chapter 4 shows how mobile code can be used to build flexible distributed object systems that can vary dynamically in the abstractions and functionality they provide. A method for allowing richer conversation than that provided by simple remote method invocation is presented. This is extended to show how a service can allow clients to bind tightly to its internal state when object-based abstractions are not required.

Chapter 5 looks at using mobile code to change where interaction between components in a distributed system takes place. By moving computation close to resources, network access can be reduced and otherwise infeasible application-specific access patterns deployed. These advantages are discussed in the context of work that uses mobile code in network control. The mobile code system described in Chapters 2 and 3 is integrated with an ATM control architecture in order to carry out these experiments.

Chapter 6 shows how mobile code can be used to make servers stateless. This is achieved without moving computation into clients. Instead, arbitrary client session state is put into the responses that a server returns to a client. Servers in this arrangement are used only to provide general execution facilities; they hold no application-specific code. An implementation is described that uses a networked hypertext system to provide a context for client-server interaction.

Chapter 7 talks about supporting user mobility. That is, the ability of a user to move location and take a personalised environment with him. Mobile code can help to migrate his state. An architecture for supporting user mobility is discussed. Details of two implementations of this architecture are then given. This chapter demonstrates that a layer is required which bridges support for application mobility and network support for mobile computing.

Chapter 8 concludes by showing how the research goals have been met, discusses the general role of mobile code in distributed systems and states the contribution of this dissertation to research.

Chapter 2

Implementation 1
The Tube: a Mobile Code System

Enhancements to a Scheme system are described that permit language expressions to be transmitted over a network. Expressions may be tagged as executable so that the recipient site runs them as programs. Such mobile code can be easily generated in Scheme because programs and data share the same representation. A number of facilities are provided to support the implementation, including access to operating system functionality.

2.1 Introduction

2.1.1 Motivation

In order to experiment with using mobile code, a system is required that supports sending programs over a network. It should provide a basis for the transparent state-saving technique described in the next chapter by allowing remote execution of programs. Building a mobile code system allows it to be evolved according to the requirements of its applications. Facilities should be provided to enable it to interact with other services in a distributed system. It should provide a foundation for the experiments described in Chapters 4-7.

2.1.2 Functionality

The mobile code system described in this chapter is called the *Tube*. It is implemented as a native code executable, which is run on every machine to which code might be sent. Each instance is called a *Tube site* (see Figure 2.1). Programs are written in the Scheme language [**Clinger91**] and transmitted over a network as byte sequences.

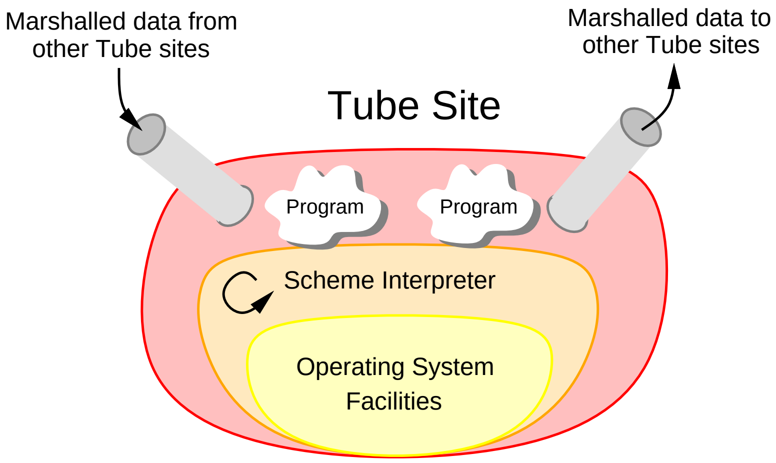

Figure 2.1: Tube site

Tube sites provide a function that turns a Scheme expression into a sequence of bytes (held as strings), which can then be sent over a network or saved to persistent store. This process is known as marshalling. The opposite process, which converts strings (read from the network) into expressions, is known as unmarshalling and is also provided as a function by Tube sites.

Tube sites contain a main control loop which continually reads (marshalled) expressions sent over the network. The function that does this reading recognises specially-tagged expressions as being programs and passes them to an interpreter for execution. Programs sent to a Tube site have access to this function, and to a function that sends expressions over a network. They can also tag expressions they send as executable. They can therefore send other programs elsewhere and read new ones over the network.

Scheme is used because Scheme programs have the same representation as Scheme data. Programs can create other programs through simple assembly of data structures. Other languages, that do not have this duality of code and data, have to resort to techniques such as printing program text and their variable values to strings. Further, Scheme's quoting facilities allow programs easily to insert values into the new programs they create. An alternative approach, to make higher-order language features transmissible over networks, is discussed in Chapter 3.

To summarize, Scheme expressions can be transferred between Tube sites. Expressions can be tagged as executable, in which case they are interpreted as programs on receipt. These programs can themselves send and receive further expressions and programs.

Tube sites are multi-threaded so that more than one program can be run at once. In order to support exchange of data between Tube sites, access is provided to operating system and network operations. Programs running on Tube sites can access World-Wide Web servers and can be launched from and embedded inside Web pages. They can also create, manipulate and move graphical user interfaces.

The rest of this chapter gives an overview of related work and describes the facilities provided by Tube sites in more detail.

2.2 Related Work

This section discusses work that falls into the first two categories of the classification for mobile code systems given in Section 1.2.

A mobile code system that supports simple distribution allows programs to be moved once, to their destinations. When a program delivered by such a system reaches its destination, it runs there until it finishes executing. Programs running under an explicit dispatch mobile code system can themselves create other programs to be delivered and executed elsewhere.

Programs running under a simple distribution mobile code system move once only. When they reach their destination, they do not move again. Programs running under an explicit dispatch mobile code system can move more than once. After moving, they can transfer new or duplicate program texts elsewhere.

2.2.1 Mobile code systems supporting simple distribution

Any software that is installed from transportable media or over a network can be considered mobile. It is distributed through these channels and installed at the invitation of an end user or system administrator. Some mobile code systems have been built to enhance the simple distribution of software, with facilities such as secure operation, resource control at execution time, heterogeneity and typed dynamic linking.

PostScript [**Taft85**], the page description language, is an example of remote programming. Documents are programs that are sent over a network to a printer which then executes them in order to render the desired output. They are remotely executed on the printer. The alternative would be to interpret documents using a PostScript interpreter on a computer and send the resulting bitmap image to the printer. However, this would place a heavier load on the network, since the images produced by a PostScript program are in most cases larger than the program itself. It would also be impractical in a multi-user system where many users might be running a PostScript interpreter at once. Thus it is accepted practice to have documents delivered to printers as programs, in order to reduce network and machine load.

SQL [**ANSI92**] is a database query language. When a database server is accessible over a network, SQL expressions are sent to it. SQL is a limited form of programming language, restricted to querying of data. Like PostScript, SQL is an example of a simple mobile code system already in common use. When the amount of data held by a database server is large, sending queries to the database instead of duplicating the data to each client machine reduces network usage.

Protocols used for communication between applications (e.g. Network News Transfer Protocol, Simple Mail Transfer Protocol, Simple Network Management Protocol) are limited forms of programming language. Requests are sent over a network and interpreted as instructions by their recipients, which perform the actions required. Protocols are very simple programming languages.

As the protocol becomes more complex (the X Windows protocol, for example), it approaches what one normally terms a programming language. X events are "interpreted" by a client and dispatched appropriately; a server does likewise with requests. NeWS (Network Extensible Window System [**Sun92**]) replaces the X Windows protocol with a PostScript-based one, giving true remote programming facilities — window management and widget realisation are carried out by small programs sent between client and server. The SunDew windowing system [**Gosling86**] was a forerunner to NeWS.

The Java virtual machine [**Gosling95**] is used for simple distribution of code. It supports heterogeneity because interpreters for it exist on a number of platforms and can exchange data encoded in a single format. Java applications are only mobile once; their compiled images are loaded into Java interpreters and cannot move after they start.

Java interpreters attempt to execute their programs in a secure environment but some security breaches have been found [**Felten97**]. Java bytecode is linked with an interpreter at run-time, based on the type assigned to it by its programmer. The Java virtual machine is defined so that programs can be validated as being safe to execute before they are started. Just-in-time compilers such as Kaffe [**Wilkinson97**] translate a program's byte code representation into native code after validating it. This allows long-running Java programs to be executed faster than under an interpreter, whilst at the same time ensuring that they do not access unauthorized areas of memory.

The main application of Java has been to add dynamic and interactive displays to World-Wide Web pages. Other systems that allow Web browsers to download programs embedded inside Web pages include Juice [**Franz97**], CandleWeb, Grail [**CNRI97**] and ActiveX [**Microsoft97**]. Juice is designed to execute faster than Java. CandleWeb is aimed at producing interactive animations. Grail uses a restricted version of Python [**Watters96**]. All three deal with programs that are architecturally neutral and which move once in their lifetimes (into Web browsers from a server). ActiveX incorporates a facility for downloading any program, usually compiled to native code, into Web browsers in order to provide executable content in Web pages. It relies on authentication techniques to ensure that unsafe code is not executed.

Omniware [**Adl-Tabatabai96**] delivers programs over a network in a simple byte code. It compiles them to native code and executes them until they terminate. A technique called *software fault isolation* is used to prevent programs from making attempts to access unauthorized areas of memory while they are running. This allows the virtual machine to be much simpler than Java's, for example, because program validation is based on the semantics of the underlying processor rather than the source language.

Finally, the Inferno [**Lucent97**] operating system and its Limbo programming language support delivery of applications over a network by compiling them into a machine-independent byte code.

2.2.2 Explicit dispatch mobile code systems

The Unix command `rsh` allows shell scripts to be executed on a remote computer. However, the environment provided is not safe because the scripts can execute any external program. If the commands available in the host user account were restricted then some level of security might be provided. Shell scripts are limited to the syntax provided by the shell and are plain-text only. However, they are able to dispatch other scripts to other computers.

The Network Command Language (NCL) [**Falcone87**] provides transparent access to distributed servers in a heterogeneous environment from different clients. This is achieved by expressing calls in a universally-understood Lisp-like language; clients send procedures in this language to servers for them to execute. Those procedures may themselves send other procedures for execution elsewhere.

Remote Evaluation (REV) [**Stamos86**, **Stamos90a**, **Stamos90b**] is similar to NCL in that it allows code to be transferred between computers over a network. One implementation of REV [**Clamen90**] allows Lisp expressions to be transmitted over a network. The Tube mobile code system described in this chapter provides the same basic functionality using the Scheme dialect of Lisp.

Late-binding RPC [**Partridge92**] provides transmissible functions in a single universal intermediate language. Like NCL and REV, it is aimed at replacing RPC systems with function shipping to make more efficient use of networks with fast computers but slower (and non-improving) communication latency. An experimental late-binding RPC system was written which uses Lisp as an intermediate language to carry procedure calls across the network.

The TACOMA project [**Johansen95**] addresses operating system support for mobile code and the use of programs that move about in the Internet. It supports Tcl, Perl, Python and Scheme mobile programs. However, it does not support transparent migration. Rather, programs must themselves construct scripts to be transferred for remote execution and provide some initialisation data for them.

Neither the Java virtual machine nor the Java language [**Arnold96**] provide support for migrating programs. Java interpreters cannot save the state of a running program. They simply load programs compiled in a common bytecode format and execute them from the beginning.

The Aglets Workbench [**Chang96**] allows Java programs to be written that move between virtual machines. However, it does not save the execution state of a program. A program that moves from one machine to another is restarted on each by calling the same, top-level function. Each program has to set itself up in the state it was in before it moved. The programmer must explicitly define the state to be used for each move, making a program's data more difficult to manage than if it was declared using the natural constructs of the language and transferred transparently.

The Mole project [**Straßer96**] implements mobile programs in Java using a similar technique to Aglets.

Using the technique described in the next chapter allows transparent migration of programs to be made available on unmodified Java virtual machines. Programmers are able to move programs running on the Java virtual machine in a single call and have them restart at any point. Section 3.6 briefly describes a minimal port to the Java virtual machine.

This chapter describes the Tube (explicit dispatch) mobile code system. Section 2.1 above discussed its general functionality. The Tube is similar to NCL and REV in that it sends Lisp-based code over a network. However, it has been converted into a higher-order mobile code system that supports transparent migration (see Chapter 3) and used in applications involving existing distributed systems (see Chapters 4-7). The rest of this chapter details the facilities it provides.

2.3 Interface to Operating System Facilities

Some functions provided by the underlying operating system are exposed to mobile Scheme programs running on Tube sites.

2.3.1 Threads

Programs can specify that an expression be interpreted in a separate thread of execution. Each Tube thread maps to one operating system thread. The following example creates two threads and waits for them to finish. One thread counts from 1 to 5, the other from 6 to 10. The interleaving of their output is non-deterministic.

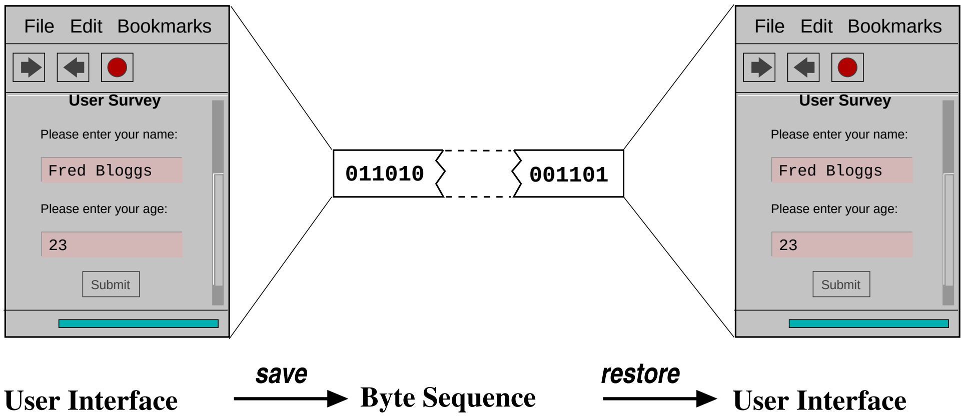

Figure 2.2: Saving the state of a user interface

```
(let ((thread1 (thread-create
          (let loop ((i 1))
             (if (<= i 5) (begin (print i) (loop (+ i 1)))))))
      (thread2 (thread-create
          (let loop ((i 6))
             (if (<= i 10) (begin (print i) (loop (+ i 1))))))))
   (thread-join thread1)
   (thread-join thread2))
```

The `thread-create` syntax behaves like `begin` except that its expressions are evaluated in a new thread of control. The thread's exit status is the value of the final expression. The function `thread-join` waits for the given thread to finish and returns its exit status.

In order to support thread programming, access is also provided to locking facilities and condition variables.

2.3.2 TCP/IP networking

A range of operations is made available to programs so that they can make and accept connections and send and receive data over a network. TCP/IP socket functions are exposed for this purpose. Sockets appear to programs as special types of Scheme port, on which Scheme's standard stream-based input and output operations can be performed. Failures signalled by the socket functions are raised as exceptions using the host Scheme's exception mechanism; network-aware programs must be able to handle them.

2.3.3 Dynamic loading

Pre-compiled, native code libraries can be loaded by a Tube site and functions contained in them made available to Scheme programs being inter-

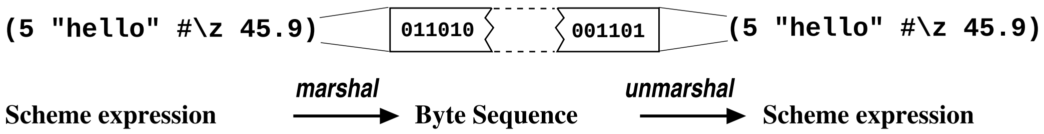

Figure 2.3: Marshalling and unmarshalling Scheme expressions

preted there. This capability has been used to interface the Tube with existing software, including a remote procedure call (RPC) system. Client stubs generated from a service interface are compiled into a library, which is then loaded into a Tube site. Scheme programs are then free to make RPC calls to legacy services by calling into the library.

2.4 Graphical User Interface

The XForms user interface toolkit [**Zhao97**] is fully exposed to Scheme programs running on Tube sites. This means they can create, manipulate and destroy their own user interfaces. Scheme wrapper functions for the XForms library are automatically generated from its C header file.

The Tube extends the XForms toolkit with the ability to return the state of any user interface as a series of bytes (see Figure 2.2). A corollary function takes a saved user interface and recreates it in a visible form. This allows a mobile program to create user interfaces on one Tube site and retain them as embedded state when it moves. When used in combination with the mechanism for saving program state described in the next chapter, a program and any user interfaces it creates can be migrated between Tube sites in a single call, without requiring special support to be written into the program itself (see Section 3.5.6).

2.5 Communicating with Tube Sites

2.5.1 Sending and receiving data

Tube sites provide a function, `marshal`, that converts a Scheme expression into a string (sequence of bytes). The `unmarshal` function converts a string returned by `marshal` back into the expression that produced it (see Figure 2.3). `marshal` can convert any of the standard Scheme data types (numbers, lists, strings, characters, symbols and vectors) except for closures. Chapter 3 discusses converting closures into byte sequences. The following comparison is guaranteed to be true for any value of `exp` (except for closures):

```
(equal? exp (unmarshal (marshal exp)))
```

That is, an expression and the result of marshalling and then unmarshalling it are structurally equivalent — recursive lists are respected. The string format generated by marshalling expressions is the same for Tube sites running on all the platforms listed in Section 2.7, and can be unmarshalled on each. The functions described below allow Tube sites to exchange marshalled expressions, thus supporting interaction over networks of heterogeneous computers.

Functions are provided which use `marshal` and `unmarshal` to send and receive Scheme expressions to and from Scheme ports. `send-sexp` uses `marshal` to convert an expression into a string and then writes the string to a port. `recv-sexp` reads a string from a port and then uses `unmarshal` to convert it into an expression.

For example, consider a Scheme port on one Tube site connected using TCP/IP functions (see Section 2.3.2) to a port on another Tube site. Given that both ports are bound by `p`, the first site can send a list of three integers to the second with the following expression:

```
(send-sexp p (list 1 2 3))
```

The second site receives the list with:

```
(recv-sexp p)
```

To send a program between Tube sites, it must be tagged as executable using `make-rep` (REP stands for Remotely Executable Program). For instance, the following expression sends down port `p` a program that prints a greeting:

```
(send-sexp p (make-rep '(print "hello world")))
```

`recv-sexp` recognises expressions returned by `unmarshal` that are tagged as executable and passes them to a Scheme interpreter (described in Chapter 3) for evaluation. Symbols that are unbound in these expressions are resolved in the global environment of the receiving Tube site. A single set of global bindings is imposed across all Tube sites. `recv-sexp` returns to its caller the result of the evaluation.

For the above example, `recv-sexp` would receive the expression `(print "hello world")`, notice that is it tagged as executable and pass it to the interpreter. The string `hello world` would be printed to the receiving Tube site's standard output port. Since the return value from the call to `print` is the string itself, `recv-sexp` also returns to its caller the string `hello world`.

The main control loop of a Tube site repeatedly calls `recv-sexp` on ports made from sockets created by listening on a well-known TCP/IP port. Other Tube sites connect to the port and can send programs for execution there by using `send-sexp` and `make-rep`.

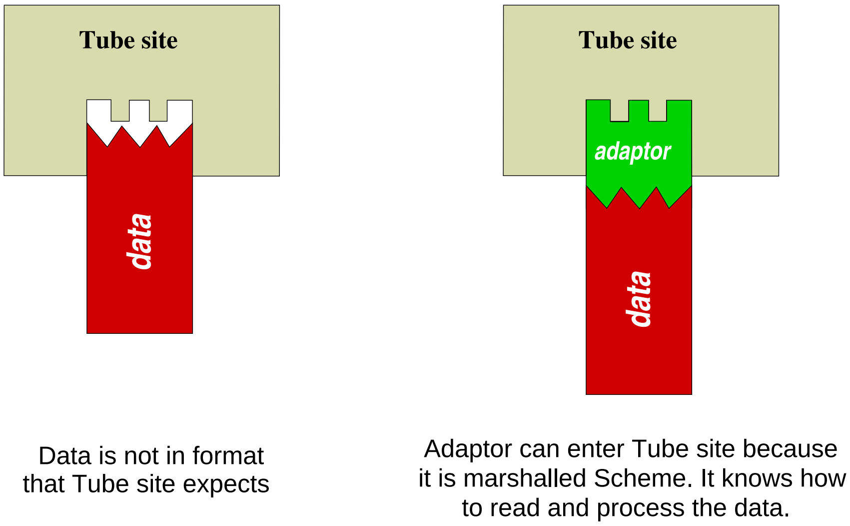

Figure 2.4: Using adaptors to handle different data formats

2.5.2 Adapting connections to new data types

The previous section discussed how Scheme expressions are sent between Tube sites and how a Tube site uses the function `recv-sexp` to read data sent to it. However, `recv-sexp` can only handle marshalled Scheme data, which means that an application connecting to a Tube site cannot send data to it in a different format. For example, one application may generate a stream of statistical data that it wishes to graph using a Tube site's user interface facilities. Another may wish to send programs typed in by a user as plain text to a Tube site for execution.

One way of allowing a Tube site to handle different types of data is to modify its implementation so that each can be read. However, this means that every time a new data format is invented, Tube sites have to be stopped, modified and started again. This is impractical for a Tube site in constant use by many client applications. Also, regular introduction of new data formats would place a significant maintenance burden on running a Tube site.

A better way is to send a program that can read the new data format to the Tube site. The program can either be sent separately from the data it handles or over the same connection. In the former case, the program is sent once, installs itself on the Tube site and reads data sent to it by applications over subsequent connections.

In the latter case, the program is sent over a connection before data in the new format. It is called an *adaptor* because it adapts the connection to the new data format.

An adaptor is a (marshalled) program sent down a connection to a Tube site in order to handle data that is sent after it (see Figure 2.4). The data that follows an adaptor is not marshalled Scheme so it cannot be read by `recv-sexp`. The adaptor, which knows about the data's format, precedes it on the connection and runs on the Tube site. It processes the data and sends back results in the other direction. It adapts the Tube site to receive a new type of data. Connecting parties can use adaptors to specify how their data is to be read and processed at a Tube site.

An example use for adaptors is to provide interactive access to Tube sites. A separate application (`ucon`, for universal connector) takes input from the user and forwards it onto a Tube site. The user's input is sent as a series of (plain text) ASCII characters. An adaptor is used to adapt connections made by `ucon` to the plain text data format.

The adaptor for plain text is sent by `ucon` when it connects to a Tube site. It runs on the Tube site and reads ASCII characters sent on afterwards. The adaptor parses them into Scheme expressions, which are evaluated. Output from the evaluation is sent back down the connection by the adaptor, which `ucon` then displays to the user's terminal. In this way, a user can interact with a Tube site.

2.5.3 Network addresses

The Tube supports different types of network connection. A table is maintained that specifies how to handle operations on different types of connection. By default, the table contains entries for three operations on two different types of connection. Connections between Tube sites are known as Tube connections and consist of two TCP/IP sockets, one for transferring data in each direction. Single-socket connections are also available and use the same socket to transfer data in both directions.

Tube connections use two sockets because Scheme ports are uni-directional. Single-socket connections can be used to communicate with other TCP/IP-based services, such as Web servers and the COBRA information retrieval system [**Mills97b**] (see Section 6.6).

Programs can open a connection, close a connection or retrieve the input and output ports associated with a connection. When opening a connection, programs supply a network address. The address specifies the type of connection to be made and sufficient information to enable the connection to be established. For instance, an address for a Tube connection is made with the function `make-tube-address`; the following expression forms the address of a Tube site that is listening for connections on TCP/IP port 1234 of the host `foo`:

```
(make-tube-address "foo" 1234)
```

To connect to the Tube site, a program would pass the address to `connect`:

```
(connect (make-tube-address "foo" 1234))
```

`connect` calls the function that opens Tube connections to establish a connection to TCP/IP port 1234 on host `foo`. It returns a handle on the connection that consists of two Scheme ports, one for reading from and one for writing to. The caller uses the ports to exchange data with the other Tube site.

Socket connections are opened by passing a socket address to `connect`:

```
(connect (make-socket-address "bar" 4321))
```

Socket addresses return Scheme ports using the same socket descriptor for reading from and writing to.

Programs can also use the handle returned by `connect` to close a connection.

The table that contains operation handlers can be extended at runtime. The distributed object system described in Chapter 4 specifies higher-level addresses and operations for resolving them. Chapter 5 introduces a new type of address for sending Scheme expressions over ATM networks.

2.5.4 Remote operation

A remote operation facility is provided that uses `send-sexp` and `make-rep` to allow programs to send expressions for evaluation at another Tube site and receive the results. The `on` function is used for this purpose. For example, to add two numbers, one bound to the symbol `num`, on a remote Tube site (host `foo`, port 1234), a program would use the following expression:

```
(on (make-tube-address "foo" 1234) `(+ ,num 149))
```

An application of `+` is formed, using quoting to insert the value of `num`. The function `on` uses `connect` and `send-sexp` to send it to the remote site and `make-rep` to embed it in a program that runs remotely. The program captures the application's return value and sends it back down the connection established by `connect`.

2.5.5 Asynchronous communication

The TCP/IP networking operations provided to Scheme programs act synchronously. Connecting to a Tube site or writing data to a port that is connected to one blocks the caller until the operation completes.

Programs that require asynchronous communication can use threads so that they do not block on these operations. A function, `dispatch`, is also provided to deliver an expression asynchronously to a given address. It uses a thread farm (see Section 2.6.2) to do this.

A caller to `dispatch` provides an address and an expression. One of the farm's threads connects to the address and delivers the expression (using `send-sexp`).

2.5.6 Netscape plug-in

The Netscape World-Wide Web browser can be extended by writing a *plug-in* to handle a specific data type. When the browser encounters data of that type embedded in a Web page, it passes the data to the plug-in. The plug-in processes and renders data it receives from the browser.

The Tube Netscape plug-in handles marshalled Scheme expressions (see Figure 2.5). It forwards them onto a Tube site running as a separate process. It also passes details of the screen window that is available for each so that they can display user interfaces embedded inside Web pages.

Keeping the browser and mobile code system separate from one another minimises integration effort. Also, using a different browser to embed Tube programs requires only the plug-in to be changed.

2.6 Other Facilities

2.6.1 Access control

Basic facilities for maintaining access control lists are provided. They can be used in conjunction with the interpreter described in the next chapter to restrict the functions and data that a program can access (see Section 3.5.2). If the programs were authenticated by some security mechanism, then a different restriction could be applied to each.

2.6.2 Thread farms

Tube sites provide concurrent computation using threads. Internal Tube services can use *thread farms* to limit the number of threads they create. A thread farm consists of a limited number of threads (see Figure 2.6). Clients of the service place work to be done on a queue. If any of the farm's threads is idle, the work is removed from the queue and given to a thread for processing. Otherwise, the work remains queued until a thread is available.

Services that use thread farms are the main control loop of a Tube site and asynchronous sending of data with `dispatch`. A site's main loop places data read from the network on a farm's queue, which limits the number of programs evaluated at once. `dispatch` places the data a program wishes to send on a farm's queue, which limits the number of threads communicating at once.

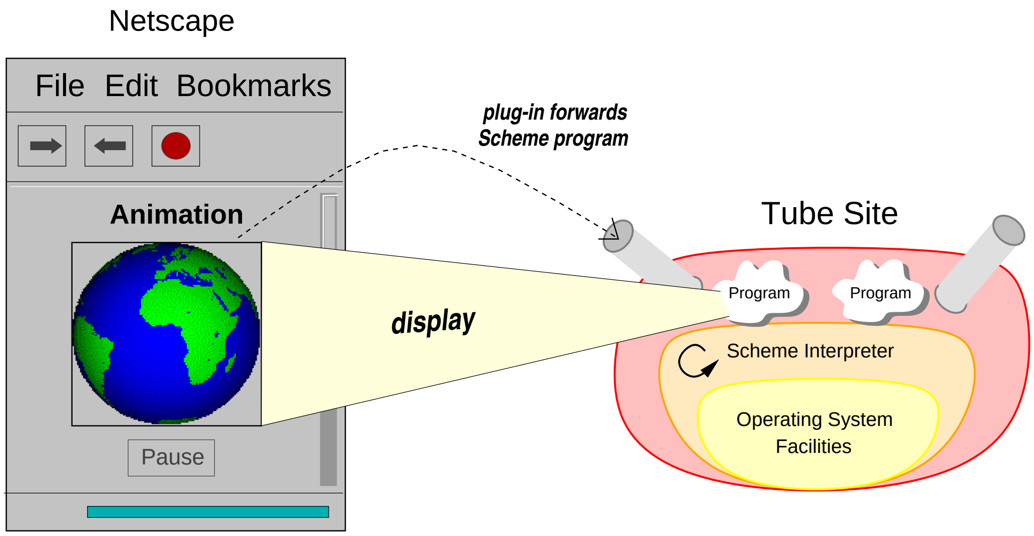

Figure 2.5: Tube Netscape plug-in

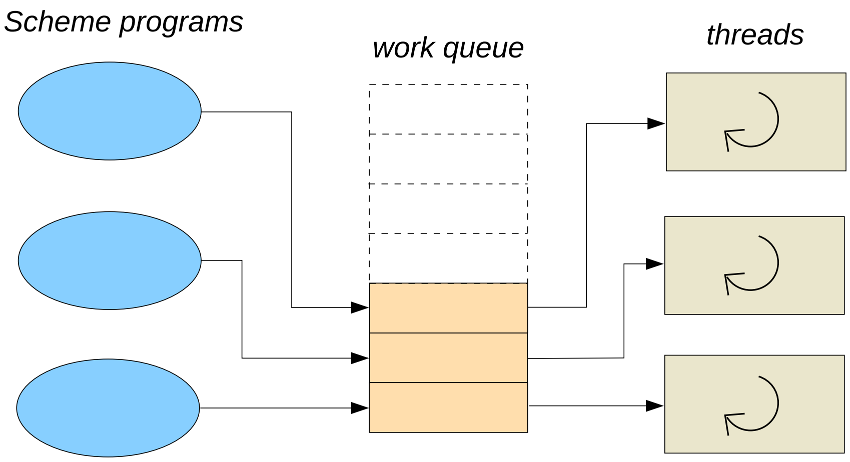

Figure 2.6: Thread farm

2.6.3 Noticeboard

A noticeboard is provided, which a program can post messages to or read messages from. Data values can thus be left by a mobile program for others that may arrive later to use. The noticeboard can be divided into different areas. An access control list is associated with each area, which can be used by programs to limit access to posted data. This would be useful in conjunction with a security mechanism that provided authentication of programs received over the network.

2.6.4 World-Wide Web access

The Tube provides functions for programs to retrieve pages from Web servers.

2.7 Implementation Platform

The current implementation of the Tube is Unix-based. It uses the Bigloo [**Serrano95**] Scheme compiler/interpreter, with patches to make it thread-safe, and the XForms user interface toolkit. Code to provide access to operating system (POSIX) functions is written in C.

Versions of the Tube are running on DEC Alphas under Digital Unix, SUN SPARCs under Solaris, Intel Pentiums under Linux and HP9000s under HP-UX. The size of a Tube site in memory when it starts up is approximately 8 Megabytes, which includes 4 Megabytes of free heap space.

2.8 Lightweight Implementation

A lightweight version of the Tube, called *Tub*, has been written for embedding inside other programs. Tub allows mobile code experiments to be extended to applications that require the deployment of many small processes. It can also be used to add network connectivity to applications, for instance by sending Scheme expressions as messages, by remote execution (see Section 2.5.4) or through the distributed object system described in Chapter 4.

Tub is written in C++ as a single object class. Instances of the class are interpreters that can receive Scheme expressions (and programs tagged as executable) over TCP/IP connections in the same way as the Tube. Tub shares the same marshalling format as the Tube, so Tube sites can easily exchange data with Tub-enabled applications.

C++ code can call Scheme functions defined within Tub interpreters and Scheme programs can call C++ functions provided by the container application. Tub can be specialised for use in applications by defining a new object class that inherits from Tub's and supplies extra functions for Scheme programs to call. Tub is embedded in the multimedia objects described in

Chapter 7 and is specialised to allow Scheme programs to establish and close audio and video connections over the network. It is also embedded in and specialised for multimedia object factories; Scheme programs can create new camera, television, microphone and speaker objects.

Tub uses a simple mark-and-sweep garbage collector to reclaim unused storage from Scheme programs. The size of an instance of the Tub interpreter class is approximately 80 kilobytes, which includes space for 10000 data storage items.

2.9 Summary

The Tube system facilitates experimenting with mobile code by allowing Scheme programs to be sent over a network and evaluated remotely. This chapter has described the way in which Scheme expressions can be communicated as byte sequences and evaluated at interconnected Tube sites. It was argued that the code-data duality of Scheme makes it particularly suitable for writing mobile programs. The facilities available to mobile programs were also discussed. Details of the Tube's implementation were given and a lightweight version described that is suitable for embedding in applications.

Chapter 3

Implementation 2
Higher-Order State Saving

An interpreter is described which allows programs to migrate transparently between computers. A program's execution state can be saved for sending over a network or a program can itself select functions for saving. The implementation is portable and engineered to interface with the mobile code system described in the previous chapter. It is used to support experiments discussed later in this dissertation.

3.1 Introduction

3.1.1 Motivation for transparent state saving

Higher-order state saving provides a way to move programs between computers. A program can be halted, its state obtained as a sequence of bytes, sent over a network link and then restarted on a different computer. Making the mechanism transparent to programs eases the implementation of their migratory behaviour. This is important for prototyping the mobile code experiments described in later chapters:

Distributed objects in Chapter 4. Distributed objects are implemented using anonymous functions. They have to be sent as advertisements to a trader and as communication proxies between applications.

Network control in Chapter 5. Programs have to traverse computers that are close to ATM switches. Allowing them to migrate using transparent state saving makes them easier to write than having to construct the program text and insert data values at each stage.

Stateless servers in Chapter 6. Programs must be able to save their complete state into a document. Supporting this in the interpreter avoids having to write special support for each program.

User mobility in Chapter 7. Programs migrate to follow a user as he moves around a network of computers.

Providing a common mechanism for obtaining a program's execution state reduces the amount of work that needs to be done by a programmer in order to make his programs mobile.

3.1.2 Mobile code in the Tube

The Tube mobile code system described in the previous chapter provides a foundation for writing mobile code. It allows Scheme expressions to be marshalled and unmarshalled for saving to persistent store and sending over a network. It also provides access to operating system and external facilities. However, the Tube is an explicit dispatch according to the classification given in Section 1.2; it provides no support for sending functions or execution state.

In Scheme, functions remember their defining scopes and are known as *closures*. A closure that represents the current state of execution is known as a *continuation*. Calling a continuation results in the computation resuming from where the continuation was captured. Without the ability to save the state of closures and continuations, programs running on a Tube site that wish to send (parts of) themselves elsewhere must form an expression to be executed at the destination. They must explicitly quote it, inserting values from the current execution that they wish to remember after migration.

3.1.3 Implementation requirements

A way of providing transparent state saving is required that can make use of existing programming environments and which is efficient enough to support the experiments described in later chapters. This minimises the amount of supporting code that needs to be written. It should be portable between different programming environments so that the experiments can be carried over to other systems in the future. In the first instance, it should integrate easily with the Tube.

A method for saving closures, continuations and program state is required that has minimal impact on the underlying Scheme system. This avoids re-engineering the (modified) Bigloo Scheme compiler/interpreter used for the Tube and eases porting of the state saving technology to other Scheme implementations. A state saving interpreter written in Scheme provides a prototype implementation that can support the mobile code experiments described later in this dissertation. Bigloo can compile the interpreter to native code, leaving only a single layer of interpretation.

The rest of this chapter first gives an overview of related work. It then describes a Scheme interpreter that allows program state to be saved to persistent store and sent over a network. Section 3.3 presents an example transformation produced by the interpreter in order to make a program's state available throughout its execution. Section 3.4 shows how the interpreter is modified to enable a program's state to be saved as a sequence of bytes. Finally, implementations of the interpreter using the Tube and the Java virtual machine are discussed. The combination of the Tube and the state-saving interpreter produces a higher-order mobile code system.

3.2 Related Work

This section discusses work that falls into the last two categories of the classification for mobile code systems given in Section 1.2.

Mobile code systems that support migration allow programs to be frozen, sent elsewhere and restarted. Higher-order mobile code systems allow programs to send arbitrary functions, together with their closing environments, elsewhere for execution. Higher-order mobile code systems that support transmission of continuations also support program migration.

Program migration is supported by other systems with varying degrees of transparency. Some achieve transparent saving of program state. The technique described later in this chapter is portable — it can use different Scheme systems as execution platforms and provide transparent state-saving facilities to each. For example, a version has been built with Kawa [**Bothner97**] to provide migration facilities on the Java virtual machine [**Gosling95**] (see Section 3.6).

3.2.1 Mobile code systems supporting migration

The Tycoon project [**Matthes95**] provides a persistent programming environment similar to Napier88 (which is described in the next section). However, unlike Napier88 it allows a thread's execution state to be saved to persistent store [**Matthes94**] and moved from one virtual machine to another [**Mathiske96**]. Recent work on Tycoon has used a continuation-passing program transformation [**Gawecki96**], which is the same technique as the one described in this chapter. However, Tycoon implements state-saving using intermediate byte-codes; the technique described in this chapter does it at a higher level and does not require modification of the underlying system. The Tycoon project has invested considerable effort in saving the state of execution threads to support persistent programming and large database applications [**Mathiske95**, **Gawecki96**]. The state-saving described in this chapter only needs to be sufficient for building prototypes of the experiments described in later chapters.

Agent Tcl [**Gray96**] is used to support applications that manage distributed information in networks. The implementation supports transparent migration, allowing Tcl programs to move from one site to another with a single call. The Tcl interpreter is modified so that it can freeze a running script and obtain its state. The state can then be sent to another interpreter and the program can continue to run there. Transparent migration removes the burden of maintaining a program's state from its programmer.

Telescript [**White94**] is a language based around mobile code for network programming. It provides a similar facility to Agent Tcl — a program running on a Telescript interpreter is moved by freezing it, sending its state to the destination site and restarting it there. Objects owned by the program are moved to the destination when they are accessed.

Sumatra [**Acharya97**] is a extension to Java that supports mobile programs. A Java byte-code interpreter has been modified in order to allow programs running on it transparently to migrate themselves over a network. This is done by copying the interpreter's stack and registers. The method for saving execution state described in this chapter allows programs compiled to Java byte-code to be migrated transparently without needing to modify the byte-code interpreter (see Section 3.6). The major benefit of choosing to target the Java byte-code is that interpreters for it are already ubiquitous; requiring them to be replaced with modified versions negates this.

Transportable PostScript (TPS) [**Heimbigner96**] is PostScript with all the drawing functionality removed and modified to allow programs to be moved around between separate interpreters at will. They can be stopped mid-execution and restarted from exactly the same position in another interpreter. The stack frame and program counter are shipped together with program source between sites.

Ara [**Peine97**] is a platform for the execution of mobile code written in various languages. It aims to apply program mobility to "weak-connection/high-volume systems such as wirelessly or intermittently connected computers, or globally distributed large data bases". Currently, Ara supports Tcl and C/C++; migration is achieved through modification of the Tcl interpreter and precompilation to an interpretable byte code respectively. Like this dissertation, Ara aims to apply mobile code to existing systems. While Ara is concentrating at first on adapting a number of languages for mobility, this dissertation describes some real applications of a single-language mobile code system.

There are many systems that have been developed to support *process migration*. They allow operating system processes to be migrated across the network and are typically used for load balancing purposes. Process migration systems allow the state of a program's execution to be saved and sent across a network. However, some assume a network of homogenous computers, some require programs to be altered so that they can be checkpointed for heterogeneous migration and some require that uninitialised copies of every program be held at each possible destination in native binary format. [**Pope96**] reviews systems that provide low-level support for application migration and describes a mechanism for migrating them that copes with heterogeneity but requires annotation to program source.

3.2.2 Higher-order mobile code systems

Obliq [**Cardelli94**] is a lexically-scoped, dynamically typed language designed for distributed object-oriented computations. It supports transmission of procedures. When a procedure is sent to another site, references to data it uses are translated into cross-network references. Any access that the procedure then makes to the data is transparently mapped onto network calls back to the data on the originating site. This maintains the data's consistency amongst the distributed processes that share it and hides its location from them. However, programs not made aware of network access to their data cannot take relevant performance and failure considerations into account [**Waldo94**]. Mapping access to a data object back to its site of creation can incur more network usage than an approach which copies data and relaxes consistency requirements. This will be the case for small objects which are accessed frequently.

Obliq does not support transparent migration of a program's execution state. Instead, a program must specify a procedure to be run at the destination site and itself maintain data pertinent to its execution. The notion of a *suitcase* is introduced, which mobile programs carry around with them by copying and updating using side effects. Suitcases remember the state that a program needs as it moves from site to site. This is not transparent migration because a program does not restart from the same point it was at before it moved and so has to maintain an indication of its progress itself.

April [**McCabe95**] is a symbolic language for building distributed applications that work over the Internet. It allows functions to be transmitted across the network. However, rather than map data access into callbacks across the network, April copies a function's closing environment and fixes the values it contains so that they cannot be modified. April cannot capture and transmit a program's execution state.

Facile [**Giacalone89**] is a development of ML that supports synchronous communication between processes using channels. It has been modified to allow functions to be transmitted over the network and used for mobile service agents (MSA) [**Knabe95**, **Knabe94**]. The MSA framework allows applications to retrieve at run-time code that provides access to a service. Interaction between application and service can adapt to network conditions and available resources. Chapter 4 of this dissertation describes a similar technique applied to distributed objects. The modified Facile allows a function represented using continuations to be sent over a network but, unlike the interpreter described in this chapter, does not use them for transparent capture and migration of a program's execution state.

Napier88 [**Morrison93**] is a persistent programming language. It allows data objects, including procedures, to be transparently maintained on a persistent store so that programs can continue to use them if they stop and later restart. Other programs can also use the saved data objects. Napier88 does not support the saving of a program's execution state to persistent store. However, a recent paper [**da Silva97**] has proposed using a persistent programming system with mobile programs to help in reducing the amount of data transferred when a program moves. When a program moves, it can continue to access its data from the persistent store.

Kali Scheme [**Cejtin95**] and Dreme [**Fuchs95**] are both dialects of Scheme that support distributed programming by permitting transmission of higher-order objects such as closures and continuations (programs' execution states) over a network. Their state-saving techniques are not independent of the interpreters they have built and so are not trivially portable to other Scheme implementations.

The rest of this chapter presents a method for saving program state that does not require writing a system from scratch and is portable between different Scheme implementations. Programs can migrate their execution states or individual functions that they define. They can capture states with continuations and then send them at any time later. The mechanism is made transparent to the programmer and supports heterogeneity because an interpreter is used to execute programs.

3.3 A Continuation Passing Interpreter

State saving is implemented using an interpreter that converts a program into a number of closures that each perform a small piece of its computation. At each stage in the program's execution, its state and the rest of the computation is defined by one of the closures. This section and the next one describe the steps taken in writing the interpreter:

- Section 3.3.1 gives an example of a program converted into a number of closures that implement its execution using the interpreter. It also describes in detail what happens when the program is run.
- Section 3.3.2 gives the definition of part of the interpreter in order to show how the example can be generalised.
- Section 3.4 shows how the interpreter can be transformed so that a program's execution state can be saved at any time as a sequence of bytes. This is achieved by converting closures that represent program state into expressions that contain no closures at all.

  It returns to the example and shows how the state at a particular stage of its execution is turned into an closureless expression that can be marshalled into a byte-stream by a system such as the Tube.

The functions that the interpreter generates for a program are written in a *continuation-passing* style [**Appel92**]. This means that they pass as arguments to each other closures that encapsulate what is to be done next for each stage of an execution.

It is outside the scope of this dissertation to describe the theory and principles behind continuation-passing interpreters. [**Queinnec96**] discusses the subject of continuation-passing Scheme interpreters.

The interpreter described in this section interprets Scheme programs and is itself written in Scheme. It takes a program written in plain Scheme and converts it into a function that implements its computation in a continuation-passing style. When called, this function performs a small piece of the program's computation and then returns enough information for the caller to be able to restart it at some time in the future. When restarted, the same happens — another small piece of the computation is performed, the program stops and some restart information returned. This information is the program's state; the continuation-passing style makes the state of a program available at each stage of its execution.

3.3.1 Example

This section gives an example of the working of the continuation-passing interpreter used as the basis for obtaining and saving the state of programs. Consider the following expression which returns the string `even` if the (integer) variable `x` is even and the string `odd` if it is odd:

```
(if (even? x) "even" "odd")
```

Figure 3.1 shows the function, f, that implements continuation-passing interpretation of the expression. It really consists of a number of small closures implementing the different stages of execution.

There are two types of closure involved, one taking two arguments (`k` and `env`) and one taking a single argument (`v` or `w`). The former break up the computation into smaller pieces. The `k` argument is the continuation to which the result of the small piece of computation is sent. The `env` argument is an environment in which to evaluate the computation; it holds the values of the program's variables. The latter type of closure are continuations. They take one argument, which is the value to pass to the rest of the computation.

The rest of this section goes through how the computation proceeds for this example. Suppose f is passed the function `result` as the continuation to send its result to and an environment which maps the symbol `x` to the value 23. `result` is a pre-defined continuation which is used to denote the end of a computation. It takes one argument and returns a specially-tagged version of it, indicating that the computation is complete.

Working through, the closure on line 2 is called first, with the continuation defined on line 21 and `env` passed as arguments. This in turn calls the closure on line 3, with the continuation defined on line 18 and `env` (in this program, the same environment is used throughout) passed as arguments.

```
(lambda (k env)
   ((lambda (k env)
       ((lambda (k env)
           ((lambda (k env) (lookup k 'even? env))
            (lambda (v)
               ((lambda (k env)
                   ((lambda (k env) (lookup k 'x env))
                    (lambda (v)
                       ((lambda (k env) (send k '()))
                        (lambda (w)
                           (send k (cons v w)))
                        env))
                    env))
                (lambda (w)
                   (send k (cons v w)))
                env))
            env))
        (lambda (v)
           (do-apply k env (car v) (cdr v)))
        env))
    (lambda (v)
       (if v
           ((lambda (k env) (send k "even")) k env)
           ((lambda (k env) (send k "odd")) k env)))
    env))
```

Figure 3.1: An expression converted into a function f that implements its interpretation in continuation-passing style

Next, the closure on line 4 is called, with the continuation defined on line 5 and `env` passed as arguments.

This closure calls `lookup`, which looks up the symbol `even?` in the environment `env` and passes the result to the continuation `k`. In this case, `env` does not specify a value for `even?`. However, if Bigloo defines the symbol then its value will be used (see Section 3.5.2). Since `even?` is a standard Scheme function, it is defined in Bigloo's interpreter and `k` (which is actually the continuation defined on line 5) receives its value.

Instead of simply calling `k` with the `even?`'s value, `lookup` is defined to pass both to the `send` function. This is a pre-defined function which takes two arguments, a continuation and a value to be passed to it, and returns them `cons`ed together as a pair. So, the call to f now returns, with this pair as its result. This is the current state of the program; it has got as far as looking up the value of the `even?` symbol. The program can be resumed at any time by applying the pair's `cdr` to its `car`.

Suppose that this is now done and that the program resumes. The continuation on line 5 receives the `even?` function and calls the closure on line 6 with the continuation on line 14 and `env` as arguments. The closure on line 7 is then called, with the continuation on line 8 and `env` as arguments. `lookup` is called, which looks up the value of `x` in `env` as `23` and returns `k` (the continuation on line 8) `cons`ed with it. This is the state of the next stage in the program, which has stopped again.

Restarting the program by applying the returned pair's `cdr` to its `car` results in the continuation on line 8 being passed the number `23` as an argument. The closure on line 9 is called with the continuation on line 10 and `env` as an argument. A new program state is returned consisting of the continuation on line 10 and the empty list in a pair. Restarting the program in the normal way brings execution to line 11, where `v` is bound to the number `23`, `w` is bound to the empty list and `k` is bound to the continuation on line 14. The next program state is returned, consisting of this continuation `cons`ed with a list containing the number `23`.

Restarting the program using this state moves execution to line 15, where `v` is bound to Bigloo's `even?` function, `w` is bound to the list `(23)` and `k` is bound to the continuation on line 18. The next program state is returned, consisting of this continuation `cons`ed with a list containing the `even?` function and the number `23`, in that order. Restarting this state brings execution to line 19, where `v` is bound to the list and `k` is bound to the continuation on line 21. The `do-apply` function is called and passed `k`, `env`, the `even?` function and a list containing the number `23` as arguments. `do-apply` applies some arguments (in this case `23`) to a function (`even?`) and `send`s the result (`#f`) to a continuation (`k`). This returns the penultimate program state.

Restarting this state executes the continuation at line 21, with `v` bound to `#f` and `k` bound to the `result` continuation. The closure on line 24 is called, and `send`s the string `odd` to the `result` continuation, returning the final program state. Restarting this state returns the string `odd` with a special tag indicating that the program has finished its computation.

As is evident from this example, converting a program to use a continuation-passing style yields its state at every stage of execution. The program does some computation, then stops and returns its state. Running it from start to finish involves repeatedly restarting it from the successive states returned until it gives up its final value. Without conversion to continuation-passing style, the program runs from start to finish and none of its states are visible.

A program cannot be converted to use continuation-passing once it starts running. If its state is required at any stage of its execution, the whole program must be converted before it starts.

3.3.2 The `scan` function

The heart of the continuation-passing interpreter is the `scan` function. It converts a Scheme program into a function implementing its interpretation in continuation-passing style, as illustrated with the example in the previous section.

The `scan` function uses the `lambda` of the host Scheme (Bigloo) in the conversion to continuation-passing style rather than introducing a new form that is managed by the interpreter itself so that:

- continuation-based programs can inter-operate with code written directly in the host Scheme
- any optimization (e.g. conversion to bytecode or native code) implemented by the host Scheme is capitalised upon

Figure 3.2 shows the parts of the `scan` function relevant to the example discussed in Section 3.3.1. A brief description is given here, to point out the code which generated the continuation-based interpretation shown in Figure 3.1.

Lines 2–3 define the closure returned for symbols. It simply calls `lookup` to pass the value of the symbol in the environment `env` to the expression's continuation, `k` (the rest of the computation). Lines 6–18 define the closure returned for `if` statements. Line 7 converts the statement's test expression into a continuation-based function. This is passed the continuation on line 14, which will be called later when the test function has finished. Line 16 is evaluated if the test returned true, line 17 if it returned false. The statement's consequent or alternative expression (which are converted to continuation-passing style on lines 8 and 9) is accordingly passed the continuation of the `if` statement, `k`, that encapsulates the rest of the computation.

Lines 20–26 are evaluated for a function application. The `sclis` function described below converts the list containing the application into a

```
(define (scan exp)
   (cond ((symbol? exp)
          (lambda (k env) (lookup k exp env)))
         ((pair? exp)
          (cond ...
                ((eq? (car exp) 'if)
                 (let ((scan-test (scan (cadr exp)))
                       (scan-consequent (scan (caddr exp)))
                       (scan-alt (scan (if (pair? (cdddr exp))
                                           (cadddr exp)
                                           '()))))
                    (lambda (k env)
                       (scan-test
                        (lambda (v)
                           (if v
                               (scan-consequent k env)
                               (scan-alt k env)))
                        env))))
                ...
                (else
                 (let ((scanned (sclis exp)))
                    (lambda (k env)
                       (scanned
                        (lambda (v)
                           (do-apply k env (car v) (cdr v)))
                        env))))))
         (else
          (lambda (k env) (send k exp)))))

(define (sclis exp)
   (if (null? exp)
       (lambda (k env) (send k '()))
       (let ((first (scan (car exp)))
             (rest (sclis (cdr exp))))
          (lambda (k env)
             (first
              (lambda (v)
                 (rest
                  (lambda (w) (send k (cons v w)))
                  env))
              env)))))
```

Figure 3.2: Parts of the `scan` function and the auxiliary `sclis` function which convert an expression into continuation-passing style

continuation-passing closure which implements the evaluation of each of its members. The continuation defined on line 24 is passed to the result. It calls `do-apply`, which actually applies the function (the first element in the resulting list) to its arguments (the rest of the elements in the list) and sends the result to the rest of the computation held in the continuation `k`. Lines 27–28 handle any other type of expression simply by sending it to the rest of the computation.

The recursive `sclis` function defined in lines 30–41 converts a list of expressions that need to be `scan`ned into a continuation-based function that evaluates them one-by-one and makes a list of the results. Line 32 handles the terminating case when the end of the list is reached. Lines 33–41 handle a list of at least one element by creating a function which will evaluate the first element (lines 33 and 36) and pass to the evaluation a continuation (lines 37–40) that evaluates the rest. Line 34 implements the recursion that ensures all elements of the list are `scan`ned and evaluated. Line 39 ensures that the elements are `cons`ed together to form a list, which is passed to the rest of the computation.

Instead of immediately sending a value to a continuation, `send`, `lookup` and `do-apply` return them as a pair. This stops the computation but ensures that a program's state, defined by pairs of continuations and values, is available at each stage of its execution. Applying the value in a pair to its associated continuation is enough to restart the program in the same state it was in when the pair was made.

3.4 State Saving

The previous section described the `scan` function, which implements a continuation-passing interpreter for programs that allows their states to be obtained at each stage in their execution. The states are continuations of the execution, represented using Bigloo closures. *If they could be saved to and restored from a byte-stream, a program could be migrated by sending the stream over a network or stored for later resumption.*

Futhermore, `scan` allows programs to capture their own execution state, using the special language form `let/cc` (see Section 3.5.1). Programs would be able to migrate themselves to other computers or persistent store if the state they capture could be saved into a byte-stream. Functions defined by programs running under continuation-passing interpretation would also be transmissible.

`scan` generates Bigloo closures. Like other Scheme interpreters not designed for code mobility, Bigloo cannot save closures generated with its `lambda` expressions in a form suitable for placement in a persistent store or transmission over a network. *The modification to* `scan` *described in this section allows closures it generates to be saved by converting them into ex-*

pressions that contain no closures at all. No modification is required to Bigloo and the technique is portable to other Scheme systems. It does not require building a new environment for executing Scheme programs.

The rest of this section describes how the modification to `scan` is made. Section 3.4.1 presents a technique for remembering the definition of closures. Sections 3.4.2 and 3.4.3 show by means of an example how this technique is applied to the `scan` function. This allows the definitions of the continuations it generates for a program's execution to be obtained at any time, as is illustrated in Section 3.4.4.

3.4.1 Memoizing closure definitions

The technique used for saving the state of closures generated by the continuation-passing interpreter is to remember their definitions when they are created. The `scan` function described in Section 3.3.2 explicitly defines a set number of `lambda` forms, each encapsulating a part of the computation or a state of execution. Therefore, the definition of each is known when it is created with a `lambda` expression.

The `memoize-lambda` function is used to associate a closure with its definition. It does this by making a new closure which holds both the original one and its definition:

```
(define (memoize-lambda lambda-fn lambda-defn)
   (lambda args
      (if (get-replacing?)
          lambda-defn
          (apply lambda-fn args))))
```

`memoize-lambda` might be used as follows:

```
(memoize-lambda
   (lambda (x) (* x 2))
   '(lambda (x) (* x 2)))
```

This gives a function which doubles its argument and its (quoted) definition. The closure returned by `memoize-lambda` in this case will be equivalent to:

```
(let ((f (lambda (x) (* x 2))))
   (lambda args
      (if (get-replacing?)
          '(lambda (x) (* x 2))
          (apply f args))))
```

The `get-replacing?` function normally returns `#f`, so that closures returned by `memoize-lambda` behave in the same way as the ones given to it as arguments. So the closure returned by

```
(memoize-lambda
   (lambda (x) (* x 2))
   '(lambda (x) (* x 2)))
```

can be used in exactly the same way as

```
(lambda (x) (* x 2))
```

The `get-procedure-defn` function allows a closure's definition to be obtained by telling `get-replacing?` to return `#t`. So the following expression:

```
(let ((f (memoize-lambda
            (lambda (x) (* x 2))
            '(lambda (x) (* x 2)))))
   (list (f 5) (get-procedure-defn f)))
```

returns the following list:

```
(10 (lambda (x) (* x 2)))
```

This demonstrates that programs can both use and obtain the definition of closures returned by `memoize-lambda`. A closure's definition can be passed through Bigloo's `eval` function to obtain the closure again. There is a slight problem in the above example — the definition does not itself contain a call to `memoize-lambda` so the closure obtained from `eval` will be unable to yield its definition to `get-procedure-defn`. To solve this, the `memoize-lambda-to-self` macro allows closure definitions to be passed through `eval` as many times as necessary:

```
(define-macro (memoize-lambda-to-self . args)
   `(memoize-lambda
       (lambda ,@args)
       '(memoize-lambda-to-self ,@args)))
```

The doubling function can be rewritten using `memoize-lambda-to-self` as follows:

```
(memoize-lambda-to-self (x) (* x 2))
```

Passing this through `get-procedure-defn` returns its definition:

```
(memoize-lambda-to-self (x) (* x 2))
```

This definition can be passed through `eval` to obtain the closure. The definition yielded by passing that closure through `get-procedure-defn` will be the same.

3.4.2 Modifying the `scan` function

Using `memoize-lambda` and `memoize-lambda-to-self` in the `scan` function allows the closures used in continuation-passing interpretation to be turned into expressions that contain no closures. They can then be marshalled into byte-streams using the Tube's `marshal` function, for example (see Section 2.5.1).

The first step is to replace the `lambda` forms used in the `scan` function with calls to `memoize-lambda`. Figures 3.3 and 3.4 illustrate the fragment of `scan` shown in Figure 3.2 converted to use `memoize-lambda`. The definitions given to `memoize-lambda` and `memoize-lambda-to-self` use Scheme's quasiquoting and unquoting to insert values from the interpretation.

3.4.3 Example

Section 3.3.1 ran through an example of continuation-based interpretation. This section shows the definition of the state at a particular point in the example's execution. The next section discusses how a program state's definition can be extracted by recursive application of the `get-procedure-defn` function to the closures generated by `scan` for interpretation.

Figure 3.1 showed an expression converted into continuation-passing style. The result of the conversion was a closure implementing interpretation of the expression. Since the expression was an `if` statement, passing the closure to `get-procedure-defn` simply returns the following:

```
(scan '(if (even? x) "even" "odd"))
```

This is clearly correct since it was the same expression that generated the closure.

Suppose that the interpretation has reached line 9 of Figure 3.1. A program state is returned by `send`, consisting of the continuation on line 10 and the empty list in a pair. The continuation's definition can be obtained by hand through recursively replacing `k` with its definition and substituting the values of bound variables:

```
(lambda (w)
   (send (lambda (w)
            (send (lambda (v)
                     (do-apply (lambda (v)
                                  (if v
                                      ((lambda (k env) (send k "even"))
                                       result env)
                                      ((lambda (k env) (send k "odd"))
                                       result env)))
                               env (car v) (cdr v)))
                  (cons even? w)))
         (cons 23 w)))
```

```
(define (scan exp)
   (cond ((symbol? exp)
          (memoize-lambda
             (lambda (k env) (lookup k exp env))
             `(scan ',exp)))
         ((pair? exp)
          (cond ...
                ((eq? (car exp) 'if)
                 (let ((scan-test (scan (cadr exp)))
                       (scan-consequent (scan (caddr exp)))
                       (scan-alt (scan (if (pair? (cdddr exp))
                                           (cadddr exp)
                                           '()))))
                    (memoize-lambda
                       (lambda (k env)
                          (scan-test
                           (memoize-lambda
                              (lambda (v)
                                 (if v
                                     (scan-consequent k env)
                                     (scan-alt k env)))
                              `(memoize-lambda-to-self (v)
                                  (if v
                                      (,scan-consequent ,k ',env)
                                      (,scan-alt ,k ',env))))
                           env)))
                    `(scan ',exp)))
                ...
                (else
                 (let ((scanned (sclis exp)))
                    (memoize-lambda
                       (lambda (k env)
                          (scanned
                           (memoize-lambda
                              (lambda (v)
                                 (do-apply k env (car v) (cdr v)))
                              `(memoize-lambda-to-self (v)
                                  (do-apply ,k ',env (car v) (cdr v))))
                           env))
                       `(scan ',exp))))))
         (else
          (memoize-lambda
             (lambda (k env) (send k exp))
             `(scan ',exp)))))
```

Figure 3.3: Parts of the `scan` function converted to use `memoize-lambda`

```
(define (sclis exp)
   (if (null? exp)
       (memoize-lambda-to-self (k env) (send k '()))
       (let ((first (scan (car exp)))
             (rest (sclis (cdr exp))))
          (memoize-lambda
             (lambda (k env)
                (first
                 (memoize-lambda
                    (lambda (v)
                       (rest
                        (memoize-lambda
                           (lambda (w)
                              (send k (cons v w)))
                           `(memoize-lambda-to-self (w)
                               (send ,k (cons ',v w))))
                        env))
                    `(memoize-lambda-to-self (v)
                        (,rest
                         (memoize-lambda
                            (lambda (w)
                               (send ,k (cons v w)))
                            `(memoize-lambda-to-self (w)
                                (send ,,k (cons ',v w))))
                         ',env)))
                 env))
             `(scan-sclis ',exp)))))
```

Figure 3.4: The auxiliary `sclis` function converted to use `memoize-lambda`

The version of `scan` that uses `memoize-lambda` and `memoize-lambda-to-self` to remember closure definitions would represent this continuation with the following function:

```
(memoize-lambda-to-self (w)
   (send (memoize-lambda-to-self (w)
         (send (memoize-lambda-to-self (v)
                  (do-apply (memoize-lambda-to-self (v)
                                (if v
                                    ((scan "even") result env)
                                    ((scan "odd") result env)))
                            env (car v) (cdr v)))
               (cons even? w)))
         (cons 23 w)))
```

This can be obtained by using the `unmemoize` function described in the next section.

3.4.4 Obtaining and restoring program state

The first step towards converting the closures generated by the `scan` function into expressions containing no closures at all was to use `memoize-lambda` in the `scan` function. The second step is recursively to replace closures with their `memoize`d definitions. This can be done for program state, which consists of a continuation and a value to pass to it, and functions defined by continuation-based programs.

A function called `unmemoize-expression` performs this recursive replacement. Closures are not updated in-place. Rather, new expressions are generated containing their definitions, allowing closures to continue to be used afterwards. For the continuation shown above, at the end of the previous section, it would generate the following expression:

```
(MEMOIZE-REPLACED (memoize-lambda-to-self (w)
   (send (MEMOIZE-REPLACED (memoize-lambda-to-self (w)
         (send (MEMOIZE-REPLACED (memoize-lambda-to-self (v)
                  (do-apply (MEMOIZE-REPLACED (memoize-lambda-to-self (v)
                                (if v
                                    ((MEMOIZE-REPLACED (scan "even"))
                                       result env)
                                    ((MEMOIZE-REPLACED (scan "odd"))
                                       result env))))
                            env (car v) (cdr v))))
               (cons even? w))))
         (cons 23 w))))
```

This is not a function definition, but rather a list of elements; it cannot be evaluated. `unmemoize-expression` returns only basic data types (numbers,

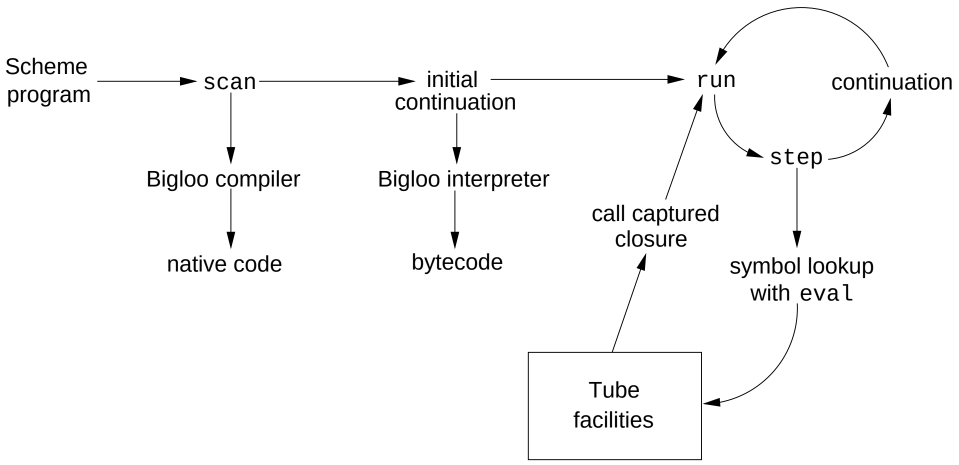

Figure 3.5: Interfacing the continuation-based interpreter with the Tube

symbols, strings, characters, vectors and lists), not closures. Lists beginning with the `MEMOIZE-REPLACED` symbol contain a closure's definition as their second element.

The function `replace-expression` takes an expression returned by the function `unmemoize-expression` and recursively replaces lists beginning with `MEMOIZE-REPLACED` with the closures that result from passing their second elements to `eval`. A closure with exactly the same behaviour as the original given to `unmemoize-expression` is thus produced.

`unmemoize-expression` generates a representation of a closure that contains no closures itself. `replace-expression` takes such a representation and regenerates the closure. Section 3.5.4 shows how the Tube uses these functions to enable program state to be marshalled into a byte-stream and transferred over a network.

3.5 Interfacing with the Tube

This section describes how the continuation-based interpreter discussed in Section 3.3 inter-operates with the Tube mobile code system discussed in the previous chapter. Figure 3.5 shows the process of converting a Scheme program into continuation-passing style using `scan` and then executing it using `run` and `step` (see Section 3.5.1). The program can access Tube facilities and make functions it defines available for other programs running on a Tube site to call (see Section 3.5.2).

Section 3.5.3 discusses run-time exceptions in the interpreter. Section 3.5.4 shows how the the technique for obtaining marshallable state described in Section 3.4 is used by the Tube to move programs. Section 3.5.5 discusses the efficiency of the interpreter and the state-saving technique. Section 3.5.6 discusses saving user interface state.

3.5.1 Calling the continuation-passing interpreter

An expression to be interpreted is passed through Bigloo's macro expansion phase before being given to the `scan` function. The function returned by `scan`, implementing a continuation-passing interpretation for the expression, is then passed `result` for the terminating continuation and a default environment.

The interpretation makes available its state at each stage of execution. To carry the computation through to termination, each state must be restarted. A driver loop function called `run` keeps restarting the computation until it detects a value tagged by `result`:

```
(define (run state)
   (if (result? state)
       (result-val state)
       (run (step state))))
```

The `result?` predicate returns true if its argument has been tagged by `result`. `result-val` retrieves the final value of the interpretation. The `step` function restarts an interpretation state.

Program state returned by continuation-passing interpretation is actually implemented using Bigloo closures, not using pairs as described in Section 3.3.1. Closures are used so that a program's state can be called as if it was an ordinary function from code not using continuation-passing. When called without arguments, the closure containing the program's state calls `run` to execute the rest of the program until it completes. Passing it to the `step` function (as `run` does, above) passes a value to the closure telling it to perform only a single stage of execution and return the next state.

Since continuation-passing interpretation splits up a program's execution into smaller stages, the processing resources it consumes can be controlled. For example, the amount of time a program spends executing can be restricted by limiting the number of times that the `run` function recurses. The speed at which the program runs can be controlled by inserting a delay before calling the `step` function.

A different driver loop function checks a message queue each time it `step`s the interpretation state. If a new thread is created to interpret a program using this function, other threads can at any time contact it through the queue and obtain the program's state of execution. They can also at the same time request that the interpretation pause or stop. *This allows programs to be migrated that do not themselves contain state-saving directives.*

Closures defined by expressions passed through `scan` can be called from code not using continuation-passing in the same way as those encapsulating program state. This allows programs to be written which contain a mixture of code that does not use the continuation-passing interpreter and code that does.

Programs can obtain their current state by using the `(let/cc k ...)` form. This binds `k` to the continuation of the program at that point, with a scope equal to the body of the form. Unlike `let/cc` in EuLisp [**Padget92**], `k` can be called outside the form's dynamic extent, at any time during or after the program's execution. `k` can be captured for later use with a side-effect or appear in the value returned by the body of the `let/cc` form. Calling `k` requires one argument, which is used as the value to substitute for the `let/cc` expression in evaluation of the rest of the program from the point at which the form occurred.

Closures and continuations defined and obtained in continuation-passing interpretation can be made available by assignment to data obtained from Tube functions (see below) or as the final return value of the evaluation.

3.5.2 Making Tube data available to the interpreter

If a symbol's value is not found in an environment then Bigloo's global environment is searched by passing the symbol to Bigloo's `eval` function. *This allows functions provided by the Tube outside the continuation-passing interpreter to be called from inside it.* Function definitions are cached in a hash table to minimise the number of times that `eval` has to be called.

By filtering access to the Tube's global environment through access control lists, the range of functions made available for (mobile) programs can be restricted.

3.5.3 Exceptions

Exception handling in continuation-passing interpretation is fully inter-operable with Bigloo's exception handling. Programs passed through `scan` and executed by `run` and `step` can catch exceptions thrown by Tube functions. They can also throw exceptions that can be caught by Tube functions.

3.5.4 Using marshallable program state

Section 3.4 demonstrated that closures encapsulating program state can be converted into representations containing no closures at all. It also showed that such representations can be converted back into the original closures. This is required because Bigloo does not provide support for marshalling closures. Converting them into expressions that can be marshalled into a byte-stream allows programs executing under continuation-passing interpretation to be migrated over the network or to persistent store.

The Tube provides the ability to marshal Bigloo expressions that do not contain closures into byte-streams (see Section 2.5.1). Hence, *once converted*

```
1    (let loop ((i 1))
       (if (<= i 10)
           (begin (print i)
                  (if (= i 6)
5                     (if (let/cc k
                             (dispatch
                               (make-tube-address "HostB" 54321)
                               (make-rep (savefn (lambda () (k #t)))))
                             #f)
10                        (loop (+ i 1)))
                      (loop (+ i 1))))))
```

Figure 3.6: A migrating Scheme program

into closureless expressions by `unmemoize-expression`, *a program's state can be marshalled into a byte-stream.* The byte-stream can then be transmitted over a network or saved to persistent store. A Tube site receiving it from the network or reading it from store can unmarshal the data and obtain the closureless expression holding the program's state. `replace-expression` can then be used to make a closure implementing the program's interpretation. This closure can then be invoked to restart the program — having been moved across a network or suspended and kept in a persistent store.

Since Tube marshalling can cope with machine-specifics such as byte-ordering, program state can be captured, transferred and restarted between different types of computer.

For instance, suppose there are Tube sites running on two computers, `HostA` and `HostB`, and that they can send data to each other over a network. Both listen for new connections on port 54321. Figure 3.6 shows a Scheme program that counts from 1 to 10, migrating itself between sites when it reaches the number 6. If it is executed under continuation-passing interpretation on `HostA`, then the numbers 1 to 6 will be printed on `HostA`, before the program sends its state to `HostB` and prints the numbers 7 to 10 there.

Lines 5–9 capture and migrate the program's continuation when it has counted up to and printed the number 6. The `let/cc` form binds the continuation to `k`, which is then sent to `HostB` and `#f` is returned to terminate execution. `dispatch`, `make-tube-address` and `make-rep` were introduced in Chapter 2. `dispatch` marshals and sends an expression over a network asynchronously, `make-tube-address` specifies the address of a Tube site and `make-rep` tags an expression as executable.

It should be pointed out that some method is usually provided in Lisp-like languages for capturing a program's continuation (`let/cc` here or `call/cc` in standard Scheme). Programs must use it to capture their states even if they do not migrate elsewhere. Lines 6–8 of the example in Figure 3.6 are particular to its migration.

`savefn` converts `k` into a form suitable for transmission over a network. It takes one argument, a closure that takes no arguments, and returns a marshallable expression generated with `unmemoize-expression`. When this expression is passed to `eval`, it uses `replace-expression` to recreate the closure, which is then called. In Figure 3.6, the expression returned by `savefn` contains in marshallable form the continuation held by `k` because `unmemoize-expression` operates recursively. It is tagged as executable with `make-rep` so that when it reaches `HostB`, after being sent over the network with `dispatch`, the closure is recreated and called. In turn, it calls the continuation that was transferred from `k`, passing it `#t` so that the program continues on `HostB` from line 10 and prints the numbers from 7 to 10.

The size of the state transferred by `dispatch` in Figure 3.6 is 9.2 kilobytes uncompressed, 1.4 kilobytes compressed. The transfer has been timed five times between two Sun UltraSPARCs running at 143Mhz and connected by TCP/IP over a local area ethernet. The average time that elapsed between the program calling `dispatch` on line 6 and restarting in the closure on line 8 was 199 milliseconds.

The `savefn` function is defined as follows:

```
(define (savefn fn)
  (let ((ustate (unmemoize-expression `(,fn))))
    `(eval (replace-expression ',ustate))))
```

`ustate` holds a marshallable call to the closure `fn`. `savefn` returns an expression that recreates the closure and passes the call to `eval`, which evaluates it.

Tube functions used by a continuation-based program (see Section 3.5.2) are passed through `memoize-lambda` so that `unmemoize-expression` can obtain their definitions. The caches used to store these functions are stored in the environments used to remember the values of variables. Exception handlers are stored in environments too. Doing this means they are saved by `unmemoize-expression`, since `scan` puts environments into closure definitions. If necessary, programs can migrate while they are handling exceptions.

3.5.5 Efficiency

This chapter has described a method for allowing the state of Tube programs to be transparently marshalled into byte sequences without modification to the underlying interpreter. It exists as a layer on top of the underlying Scheme system.

The implementation for the Tube uses Bigloo as the underlying Scheme system. Since Bigloo can compile Scheme to native code, the state-saving layer does not add an extra layer of interpretation. The continuations generated by interpretation are represented using closures of the host system; Bigloo compiles them to bytecode. Although interpretation generates many closures, it has proved efficient enough for the experiments described in the rest of this dissertation.

`memoize-lambda` inserts a call to `get-replacing?` in the closures it returns. Since it is used to remember the definitions of the closures generated by `scan`, many calls to `get-replacing?` will be made in the course of an interpretation. `get-replacing?` checks the value of a thread-specific variable that `get-procedure-defn` sets to return a closure's definition. An alternative method that does not place such an overhead on interpretation would hold closure definitions separately. However, support from the garbage collector would then be required so that definitions are deleted when closures are collected.

The implementation saves the whole of a program's state. If a program migrates from one computer to another over a network, sending all of its state at once might be unnecessary. For slow networks, an alternative would be to transfer part of a program's state only when it is needed for execution. However, this would make the program dependent on the network throughout the lifetime of its execution; if the network failed, the program would not be able to proceed. It would also result in the state being held in two places, placing a residual burden on the computer that the program moved from.

One optimization that can be made is to the modified `scan` function described in Section 3.4.2. The function allows saving of program state by remembering closure definitions in full. Bigloo's `eval` function is used to turn them back into closures when restoring program state (see Section 3.4.4). This invokes Bigloo's byte code compiler to generate new code for each restored closure.

Instead, each form of closure that `scan` generates can be pre-compiled and given an index. Each closure instance that `scan` makes can then simply be memoized with the index of the form that defines it, together with any values needed to parameterize it (i.e. the values that are inserted into closure definitions by using quasituoting and unquoting in Figure 3.3). The closure forms are compiled once, into native code, and instances are made at run-time when restoring program state by closing them in new environments containing the memoized values.

For example, the closure generated on lines 3–5 of Figure 3.3 can be changed to the following expression:

```
(make-form 0 exp)
```

The function `make-form` is defined as follows:

```
(define (make-form index . args)
   (memoize-lambda (apply (vector-ref forms index) args)
                   (cons index args)))
```

`make-form` instantiates a parameterized closure instance (the first line of the definition) and memoizes it with a pair consisting of the form's index and the values used to parameterize the instance (the second line of the definition). The vector `forms` holds, in order of form index, functions that create parameterized closure instances:

```
(define (forms (vector
   (lambda (exp) (lambda (k env) (send k (lookup exp env))))
   ...)))
```

This defines the form with index 0, which corresponds to the closure generated on lines 3–5 of Figure 3.3.

When `replace-expression` (see Section 3.4.4) comes across a memoized form, `f`, it can recreate the closure instance that `f` represents by using the following expression:

```
(apply make-form (cons (car f) (cdr f)))
```

Implementing state saving in this way is more efficient than using `eval` to compile closure definitions into byte code because it uses parameterized instances of closures that are compiled to native code. Avoiding use of `eval`, which allows access to the full range of Bigloo functions, makes implementing a secure interpreter a little easier too.

All of the experiments described in this dissertation have been made over a local area network and no program's state has been bigger than 64 kiloybtes uncompressed. They usually compress to one fifth in size. Therefore, when a program is migrated, its state is transferred in its entirety before the program is restarted. Some measurements of the size of mobile programs and the time they take to move over the network can be found in Sections 3.5.4, 4.2.1, 6.6 and 7.5.3.

3.5.6 Program and user interface state

Provision is made for callbacks associated with Tube user interfaces to be migrated. The technique described above using `unmemoize-expression` and `replace-expression` is combined with the state-saving of user interfaces described in Chapter 2. A function called `saveUIs-fn` is defined which behaves like `savefn` but takes as extra arguments an arbitrary number of graphical user interfaces. It returns an expression that recreates all the given interfaces and the callbacks associated with them when it is evaluated. It also recreates and calls the closure passed to it, like `savefn`. `saveUIs-fn` allows a program containing one or more user interfaces to take them along when it migrates. The user interface state-saving facility of the Tube ensures that they are recreated in the same state they were saved in.

Chapter 2 also described a facility which allows Tube programs to display their user interfaces embedded inside Web pages, using a Netscape plug-in.

It showed how such programs communicate with the plug-in, which informs them when the user visits and leaves pages. It relied on the transparent state-saving facilities described in this chapter to allow programs to give their state to the plug-in when the user leaves a page and restore themselves from that state if the user visits the same page again.

3.6 State Saving on the Java Virtual Machine

The continuation-passing interpreter and the state-saving technique devised for it, described in Sections 3.3 and 3.4 respectively, are portable between Scheme systems. This is because the interpreter is written in standard Scheme. The implementation described interfaces with the Tube and uses Bigloo to compile the interpreter into native code.

A second, mimimal, version has been implemented with a different Scheme system to demonstrate the technique's portability. The state-saving interpreter is compiled with Kawa [**Bothner97**] into Java bytecodes. Compiling bytecodes using a Just-In-Time (JIT) compiler, such as Kaffe [**Wilkinson97**], leaves only a single layer of interpretation.

This version of the interpreter allows programs transparently to move between unmodified Java virtual machines (JVMs). Unlike Aglets (which are discussed in Section 3.2), programmers are not burdened with the responsibility of saving a program's execution state. The interpreter allows programs running on a JVM to be migrated without having to specify state to transfer explicitly and reconstruct execution context by hand.

3.7 Summary

Allowing a program's state transparently to be saved to a byte-stream eases the writing of migratory applications. This chapter motivated the need for migrating program state in order to support experiments described in subsequent chapters. It introduced a portable interpreter that makes available program state at each stage of execution. An example execution under the interpreter was then described. A portable modification to the interpreter was presented which allows a program's state to be saved to persistent store or sent over a network. A simple example of a program that migrates itself from one computer to another over a network was given. Finally, this chapter discussed the integration of the state-saving interpreter with the Tube mobile code system and a minimal port to the Java virtual machine.

Chapter 4

Mobile Code for Distributed Objects

This chapter shows how a distributed object system can be simply and cleanly built using mobile code. Techniques for enhancing the interaction between processes connected over a network are discussed. A simple object system is described that can make use of the state saving technique described in the previous chapter. It is then extended to allow communication with objects over a network. Mobile code is used to support object mobility and interaction.

4.1 Introduction

4.1.1 Motivation

This chapter demonstrates the following:

- mobile code can be used to enhance the way distributed processes interact
- a distributed object abstraction can be built on top of mobile code

in terms of the Tube higher-order mobile code system described in the previous two chapters. A distributed object system is required for the user mobility framework discussed in Chapter 7.

4.1.2 Benefits

Using mobile code supports the following statements made in the abstract on page iii and in Section 1.3:

- *Richer functionality can be given to the interaction between processes ... distributed components can be tightly bound together*

Section 4.2.2 shows how a client can send over a network code that embeds itself in a server. Section 4.2.3 shows how a client and server can share the same connection for calling each other.

- *convenient, application-level operations can be made over a network.*

  Section 4.2.1 shows how clients can send code to a server to interact with it there in arbitrary ways.

- *familiar distributed systems can be built with mobile code.*

  Section 4.4 shows a distributed object system based on mobile code, with object interfaces, remote method invocation, proxies, typing and traded offers.

- *[distributed systems] should also provide facilities for dynamically changing the way they [applications] communicate over a network.*

  Section 4.4.6 shows how object proxies can be updated at run-time.

- *It should be possible to define a minimal communication method that ... allows both for inter-operability and for different abstractions to be defined on top of it.*

  The Tube mobile code system uses the same communication method and data format to support messaging, remote procedure call, remote operation, distributed objects and mobile programs.

- *There is no single means of abstracting application-level communication. Existing systems provide inter-operability at too high a level of abstraction.*

  The Tube supports the abstractions listed above within a single implementation. An application that has no need for mobile programs or distributed objects does not pay for them because they are layered on top of simple messaging.

The rest of this chapter describes how mobile code can enhance the interaction between distributed processes and support a distributed object abstraction. A review of related work can be found at the end of the chapter. The work described in the rest of this chapter and in subsequent chapters can make use of support for heterogeneity provided by the underlying mobile code system (see Section 2.5.1).

4.2 Enhancing Interaction

4.2.1 Application-level remote operation

Consider a server process that provides information about share prices. In the first version, each client connects to the server and sends the name of

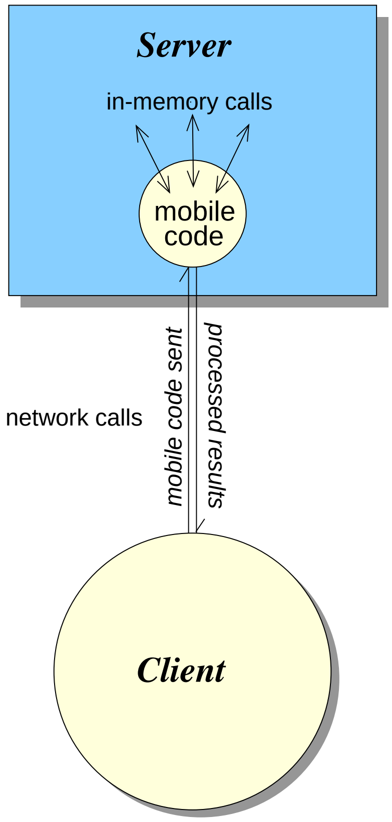

Figure 4.1: Application-level remote operation using mobile code

the company whose share price it is interested in. This works well for clients that only need to retrieve one share price at a time.

However, clients that need to retrieve more than one share price must make a separate network connection for each one. To reduce the number of connections that have to be made, a second version of the server is written that can return more than one share price at once. Clients send lists of company names and receive share prices for each.

Now suppose that a client wishes to calculate the average of a number of share prices. It would retrieve a list of them all from the server and then calculate the average. However, this means that all of the prices are transferred across the network, even though the client is interested only in their average. In order to reduce network usage, the calculation could instead be carried out on the server and only the result (a single number) transferred over the network.

This could be done by writing yet another version of the server, which is able to receive requests for share price averages. However, what if another client is written at some time which wants to know the highest from a set of share prices, and another the lowest? For each client with a new requirement, the server must be modified accordingly.

When both a server and the clients that use it are controlled by the same person, he can easily modify the the server when a client with a new requirement is written. However, when a server is used by large numbers of clients, written and run by many different people, incorporating their requirements into the server incurs significant administrative overhead. When a new client is written that wants to use the data provided by the server in a way not already supported, its author must contact the server's author and apply for the required changes to be made.

Imposing this administrative overhead in systems where a server's role is to provide access to valuable data restricts innovation in its use. It is impossible to predict the uses that other authors will want to make of a service when it is first written. An unmodified service cannot be optimised to place a minimal burden on the network for each new use. Requiring modification of a server that is used by growing numbers of applications under constant development for long periods of time is impractical.

Using mobile code allows a server to provide for the changing requirements of its many client applications in a way that is optimised in network usage for each and which does not require modification to the server. Instead of sending pre-determined requests to a server, clients send arbitrary code to execute there. This code calls the functions on the server but is able to calculate application-specific results for sending back over the network (see Figure 4.1).

The server is written with a minimal set of functions and application-level behaviour is uploaded to it as code. Since the code is sent by applications, they can change it as they undergo development, without requiring the server to be changed. Neither does the server have to be modified when a new application is written that wants to use the data it provides in a different way.

Doing this is useful only in reducing network usage. Functionally, it is the same as sending a server's data to the client and performing application-specific processing there. Using mobile code to move application-level processing into the server reduces network usage if the processing produces less data than it consumes from the server. The cost of transferring the code from client to server must also be taken into account.

For example, consider the share price server, implemented in Scheme using the Tube. The version that does not use mobile code takes a list of company names and returns a list with the share price of each in it. Suppose a function `get-share-prices-from-server` is written for clients to do this. A client would then calculate the average of a number of share prices with the following expression:

```
(let ((prices (get-share-prices-from-server
                     "Bass" "BT" "Railtrack" "PowerGen")))
   (/ (foldl + 0 prices) (length prices)))
```

The definition of `foldl` is provided at all Tube sites:

```
(define (foldl f a lst)
   (if (null? lst)
       a
       (foldl f (f (car lst) a) (cdr lst))))
```

Calculating the average in this way retrieves all of the share prices over the network. Using mobile code allows the client to move the calculation of average share price onto the server, without requiring modification to it, so that only a single value is transferred over the network. If the server is running on port 1234 of host `foo` and provides the function `share-prices` that returns company share prices to its caller, then a client can calculate an average with the following expression:

```
(on (make-tube-address "foo" 1234)
    '(let ((prices (share-prices "Bass" "BT" "Railtrack" "PowerGen")))
       (/ (foldl + 0 prices) (length prices))))
```

Section 2.5.4 describes the `on` function.

Notice that the amount of data transferred in the opposite direction, from client to server, is greater since the code must be sent along with the list of companies. However, this overhead is constant and for more than a few companies is offset by the reduction in network traffic from server to client. The client is always free to decide whether to perform the calculation on the server or retrieve the share prices and perform it locally.

A simple measurement was made to demonstrate that sending mobile code to a server can reduce network usage. The server generated an array of one hundred thousand identical numbers, which clients could process either by retrieving them over the network or sending a mobile program to the server for accessing them there. The first client calculated the sum of all the numbers by transferring the array from the server and processing them itself. A second client was written which dispatched to the server a program that calculated the sum there and returned it over the network. All programs were written using the Tube.

The measurements were conducted five times using two Tube sites (one for the clients and one for the server) running on 143MHz Sun UltraSPARCs and connected by TCP/IP over a local area ethernet. The first client took an average of 5.470 seconds and the second took an average of 365 milliseconds to calculate the sum. This demonstrates that the ability to process data at the server before it is sent to clients can significantly reduce network usage.

For comparison, rewriting the server and first client in C and using Sun RPC on the same machines to transfer the data took an average of 450 milliseconds to calculate the sum. This demonstrates that interpreted mobile code can increase performance, compared with compiled and static distributed code, if it reduces network usage by co-locating processing and data. It also shows that the Tube's marshalling code would need to be improved for use in a commercial system (this is noted in Section 6.6 as well).

The reduction in network traffic from server to client could also have been achieved by writing a new version of the server. However, this would have to be repeated for every client that comes up with a new requirement for processing the data.

4.2.2 Code injection

Chapter 3 presented an interpreter that uses continuation-passing to save the state of a running program. The same technique enables code read from the network to become part of a running program. The new code is embedded as if it was written as part of the program text. It is given access to the variables in lexical scope at the point where the embedding occurs.

The interpreter converts a Scheme expression into a nested collection of functions that implement its evaluation (see Section 3.3). To start the program, the top-level function is called and passed a continuation (to which the result of the evaluation must be sent) and an environment (which holds the values of any variables available to the program).

A program that wishes to give an expression read from the network access to its execution state simply has to provide its current environment. The `current-environment` function is provided by Tube sites and returns the environment that is active when it is called.

The `recv-sexp` function described in Section 2.5.1 for reading and executing Scheme expressions from the network is extended to take an optional extra argument. Callers can provide a function to be called for evaluating an expression after it is read.

For example, the following program reads an expression from some port `p` and gives it access to the local variable `bar`:

```
(let ((bar "an arbitrary value"))
   (recv-sexp p (let ((env (current-environment)))
                   (lambda (exp) (run ((scan exp) result env))))))
```

The `run`, `scan` and `result` functions are described in Section 3.4. The `eval-here` function is provided as syntactic sugar for the closure in the above expression:

```
(let ((bar "an arbitrary value"))
   (recv-sexp p eval-here))
```

A client could then retrieve the value of `bar` from the server with the following expression:

```
(on (make-tube-address "some-host" 5678) 'bar)
```

This dynamic loading of code into programs allows a closer integration than other methods such as dynamic linking on Unix operating systems or Java's class loader. The code being loaded has access to all of the program's execution state. Without this, a program has to pass values it wants to the loaded code explicitly.

The approach described above is called *code injection*. It makes no distinction between the running program and the code it loads; the code is integrated into the program's execution as if it had been part of its source text. Unfettered access is provided to the program's variables. When a program does not want to hide its data from embedded code, this technique allows them to be tightly bound together.

4.2.3 Sharing a connection

The `on` function allows a program to perform an operation on a remote Tube site and receive the result. The previous section showed how the operation can access the execution state of another program running remotely.

Another function, `remop`, is provided by Tube sites. It does the same as `on` but instead of making a new connection, it re-uses an existing one. That is, the address argument to `on` is replaced by two Scheme ports — one for writing the operation onto and one for reading the result from.

This allows a server program that has been called (using `on`) by a client program to make a call back into the client. The server's call uses `remop` and takes place across the same connection that the client made. A client's call to `on` only returns when the server sends back an expression that is not tagged as an executable program (i.e. not a server-initiated call back). This expression is the result of the remote operation.

Sharing a connection in this way has the following advantages over making a separate connection from the server to the client's Tube site:

- Fewer network connections have to be established. Indeed, both client and server can make further calls over the same connection until the original call (from client to server) is completed.

- When the server makes a call back into the client, it is processed by the same thread of control in the client that made the first call to the server. That is, calls made between client and server over a connection are processed by a single thread in each. There is no need to create a new thread for each call and so single-threaded clients and servers can be supported.

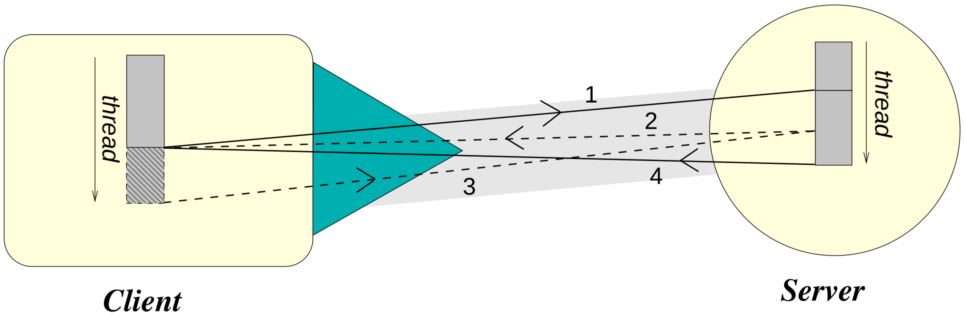

Figure 4.2: Sharing a connection

- The server's call back into the client can use the code injection technique described in the previous section to access the client's execution state. The server can then tailor its response to the client's original remote operation accordingly.

Figure 4.2 shows an example interaction between two threads over a single network connection. The client thread initiates a connection to a server thread. The diagram shows the following steps:

1. the client calls `on`, which makes a connection to the server and sends an expression to it for evaluation

2. the server (or the client's evaluation in the server) calls `remop` to send an expression over the same connection for evaluation in the client

3. the server's evaluation in the client finishes, leaving the client to continue waiting for its call to `on` to complete and allowing the client's evaluation in the server to proceed

4. the client's evaluation in the server finishes and sends its result back for `on` to return to the client

Having received the result of its evaluation in the server, the client can proceed with its execution.

4.3 A Scheme Object System

This section briefly describes the notation of a simple object system written in Scheme. It is fully portable between Scheme implementations and runs on Tube sites in order to support the distributed object system described in Section 4.4.

4.3.1 Defining objects

Objects are defined with the `make-object` syntax. For instance, the following expression creates a counter object with two methods, `inc` and `dec`:

```
(make-object () ((count 0))
   (inc: () (set! count (+ count 1)) count)
   (dec: () (set! count (- count 1)) count))
```

The `count` variable is private to the object; it is used in the object's methods but is not available outside their definitions. Both methods take no arguments, increment or decrement `count` by 1 through a side-effect and return the new value of `count`.

Different methods can be defined that share the same name but take different numbers of arguments. For instance, variants of `inc` and `dec` that increment and decrement the counter by given amounts can be defined:

```
(make-object () ((count 0))
   (inc: ()      (set! count (+ count 1)) count)
   (dec: ()      (set! count (- count 1)) count)
   (inc: (delta) (set! count (+ count delta)) count)
   (dec: (delta) (set! count (- count delta)) count))
```

An object's methods are invoked by using the object in an application and passing the method's name as a symbol. For example, the following expression returns a list containing the numbers 4 and 3:

```
(let* ((obj (make-object () ((count 0))
               (inc: () (set! count (+ count 1)) count)
               (dec: () (set! count (- count 1)) count)))
       (val1 (obj 'inc 4))
       (val2 (obj 'dec)))
   (list val1 val2))
```

4.3.2 Type annotation

The counter object defined in the previous section does not check the type of the argument `delta` before calculating the increment or decrement. The return values from the methods are not checked either. If the underlying Scheme implementation supports exceptions then the `+` and `-` functions generate a run-time exception when `delta` is not a number.

However, object definitions can also be annotated to check that a method's arguments and return value are always of a given type. The types of a method's arguments and return value are specified in its definition by enclosing them in angled brackets. The type of a return value is preceded by the symbol `->`. Types that can be specified include Scheme's built-in types, such as strings and characters, and any objects defined with `make-object`. For example, the counter can be constrained as follows:

```
(make-object () ((count 0))
   (inc: () -> <integer>
      (set! count (+ count 1)) count)
   (dec: () -> <integer>
      (set! count (- count 1)) count)
   (inc: ((delta <integer>)) -> <integer>
      (set! count (+ count delta)) count)
   (dec: ((delta <integer>)) -> <integer>
      (set! count (- count delta)) count))
```

Different methods can be defined that share the same name and arity but have different types of argument. For instance, variants of `inc` and `dec` that increment and decrement the counter by amounts given as floating point numbers can be defined:

```
(make-object () ((count 0.0))
   (inc: () -> <float>
      (set! count (+ count 1)) count)
   (dec: () -> <float>
      (set! count (- count 1)) count)
   (inc: ((delta <integer>)) -> <float>
      (set! count (+ count delta)) count)
   (dec: ((delta <integer>)) -> <float>
      (set! count (- count delta)) count)
   (inc: ((delta <float>)) -> <float>
      (set! count (+ count delta)) count)
   (dec: ((delta <float>)) -> <float>
      (set! count (- count delta)) count))
```

4.3.3 Inheritance

An object can inherit the behaviour of other objects, so that the methods they define can be invoked. For instance, assuming the above definition of the counter object is bound by the symbol `counter`, then the following expression creates an object which augments its functionality with a method that adds 10 to the counter's value:

```
(make-object counter ()
   (add10: () -> <float>
      (this 'inc 10)))
```

The symbol `this` is bound to the object itself. A new copy of `counter` is made, from its original definition, for each object that inherits from it. That is, copies of `counter` made for inheriting are always set to zero regardless of the state of `counter` itself. Inheriting an object links to its behaviour, not its state. In this case, any method not recognised by the new object is passed to its copy of `counter`.

New instances of an initial counter object can be made by inheriting:

```
(make-object counter ())
```

This behaves exactly the same as `counter` and starts counting from zero.

4.4 A Distributed Scheme Object System

The previous section described the syntax of an object system written in Scheme. The object system provides inheritance and method overloading but avoids creating registers of object types. The implementation uses closures to represent objects. Chapter 3 showed how closures can be saved into a byte-stream and migrated between Tube sites. Hence, objects can be sent between Tube sites.

This section shows how a mobile code system supporting closure (and hence object) transmission can provide the functionality of a distributed object system. The distributed Scheme object system described is called Drool (for Distributed Remote Object-Oriented Layer). A Drool object can advertise itself in a trader so that programs running on other Tube sites can invoke its methods. Advertisements can contain code implementing proxies which define how to contact and interact with the object. Client programs load these proxies at run-time and use them for communicating with the object.

Since the object system itself does not keep note of object names, the coordination of equivalent definitions on multiple Tube sites is not addressed here.

4.4.1 Advertising objects

Objects create advertisements written in SGML [**ISO86**]. They are free to use whatever tags they choose. No specific naming scheme is used for experimentation but for use with many different types of object one would have to be imposed.

Consider for example a very simple type of object, with a single method that prints the string "Hello World". Many instances of these objects can exist, each with a different name. The advertisement for an instance named `Hailer` might be:

```
<ObjectType>Hello</ObjectType>
<ObjectName>Hailer</ObjectName>
```

The tags `ObjectType` and `ObjectName` are invented for describing the object.

Programs must be able to retrieve advertisements in order to find out about objects. They use a simple query language to specify the advertisements they want to examine. The simplest query consists of a single word.

Any advertisement containing the word is matched. Tags in the advertisement can be matched with the #FIELD syntax. For example, the following query matches any advertisement for a Hello object:

```
#FIELD(ObjectType=Hello)
```

The #BAND and #BOR forms specify logical-or and logical-and respectively. For instance, the following query matches the advertisement for the Hello object named Hailer:

```
#BAND(#FIELD(ObjectType=Hello) #FIELD(ObjectName=Hailer))
```

#BAND can also be used to match tags containing more than one word. For example, the advertisement

```
<ObjectName>Head Hailer</ObjectName>
```

can be fully matched with the following query:

```
#BAND(#FIELD(ObjectName=Head) #FIELD(ObjectName=Hailer))
```

In fact, this query would also match the following offer since tags with the same name are given equal consideration:

```
<ObjectName>Head</ObjectName>
<ObjectName>Hailer</ObjectName>
```

Using SGML to define advertisements means that Document Type Definitions can (optionally) be written defining their structure. SGML tools can then be used to validate them and a program can be sure that the advertisements it retrieves are in a common format.

Adverts written in SGML are easy to turn into HTML for viewing. The `<html>` tag is reserved for allowing them to insert arbitrary HTML text. See Section 7.4.4 for an example.

4.4.2 Publishing advertisements

The previous section discussed the format of advertisements used by objects and the syntax of queries used by programs to retrieve them. Objects use advertisements to make their existence and capabilities known to programs. This section discusses how advertisements are made available to programs.

A trader [**ISO94**] is used to mediate between objects and programs by storing advertisements. Objects publish their advertisements in the trader. Programs send queries to the trader, which returns any matching advertisements.

Two implementations of a trader for SGML-based advertisements have been used. The first is a simple Python script that holds advertisements as plain text and converts queries into regular expressions for searching through them. This implementation is adequate for small numbers of advertisements.

The second trader implementation uses the COBRA information retrieval system [**Mills97b**]. It is optimised for indexing and searching large (SGML) document collections and supports Drool's query language.

Both implementations are centralised; a single trader process is created at a well-known location and accepts advertisements and queries sent to it over the network. This limits the scalability of the current Drool implementation.

Programs and objects communicate with the trader using a simple protocol. Tube sites and applications that contain embedded interpreters (see Section 2.8) download the Scheme implementation of the protocol from the trader when they first connect to it. They can also download new versions if the server's functionality changes.

4.4.3 Object addresses

Section 2.5.3 described Tube network addresses. Using advertising and a trader, Tube sites can implement high-level addresses that map arbitrary text onto network addresses.

A new address type, *content*, is added to Tube sites' address operation handler tables. Content addresses specify an advertisement and a query. The advertisement contains arbitrary SGML and is used as the address's description. The query is expressed in the query language described in Section 4.4.1 and is used for resolving the address.

For example, suppose a Tube site is started that listens for connections on port 1234 of the host `foo`. Without content addresses, users must either pass the network (Tube) address to programs by hand or the address must be hard-coded into them. With content addresses, a name can be given to the Tube site so that programs never have to know its Tube address. The advertisement to do this contains two addresses — the content address and the Tube address — and specifies that the former resolves into the latter.

The `resolves` function is used to do this. For example, the following expression gives the name "Default" to the Tube site:

```
(resolves (make-content-address "<TubeSite>Default</TubeSite>"
                                "#FIELD(TubeSite=Default)")
          (make-tube-address "foo" 1234))
```

Programs are then free to use the content address in place of the Tube address. For example, to print "Hello World" on the Tube site, a program can use the following expression:

```
(dispatch (make-content-address "<TubeSite>Default</TubeSite>"
                                "#FIELD(TubeSite=Default)")
          (make-rep '(print "Hello World")))
```

The connection handler installed for content addresses takes care of contacting the trader to find a matching advertisement. In this case, it will receive an advertisement specifying that the content address resolves into a Tube address, on port 1234 of host `foo`. The Tube address is constructed and used by `dispatch` to send the program. Section 7.4.4 shows and describes an advertisement that resolves the name of a video camera object into its network address.

Since `dispatch` only queries the trader, the first argument to `make-content-address` is not actually required. It is only used when publishing a content address, for example with the `resolve` function. The above expression can thus be rewritten as follows:

```
(dispatch (make-content-address 'unspecified
                                "#FIELD(TubeSite=Default)")
          (make-rep '(print "Hello World")))
```

It is possible to chain together any number of content addresses, each of which resolves into the next. This is why content addresses must always contain a query — if a content address resolves into another content address then a query for the latter is required for continuing the process.

The lightweight interpreter described in Section 2.8 also supports content addresses. This allows applications it is embedded in to advertise their services in the trader and to use other services advertised there.

4.4.4 Proxies

Advertisements can be used by programs when they want to contact objects. The simplest way of doing this is to publish an advertisement containing the network address at which the object is listening for connections. Objects can use the `resolves` function described in the previous section to do this. Programs can then use a content address to contact the object by name.

Consider an object that has a method, `add`, which takes two integers as arguments and returns the result of adding them together. An initial implementation of this object that allows programs anywhere to invoke the `add` method is given in Figure 4.3.

If an object defines a `start` method then it is invoked by `make-object` after the object is created. In this case, the `start` method uses `resolves` to create an advertisement that names the object `first-adder`.

The object's network address, `obj-addr`, is found from the private variable `l`, which is bound to an arbitrary Tube address by calling the function `make-tube-listener` with zero as an argument. The Tube address to which `l` is bound is found with the function `tube-listener-address`. The `tube-listen` function uses `l` to listen out for a connection to the address. It reads an expression from the connection and then evaluates it. In the `first-adder` object, the `start` method tells `tube-listen` to use

```
(make-object ()
         ((l (make-tube-listener 0))
          (obj-addr (tube-listener-address l))
          (finish? #f))
   (add: ((x <integer>) (y <integer>)) -> <integer>
      (+ x y))
   (start: ()
      (resolves (make-content-address
                    "<ObjectType>Adder</ObjectType>
                     <ObjectName>first-adder</ObjectName>"
                    "#BAND(#FIELD(ObjectType=Adder)
                           #FIELD(ObjectName=first-adder))")
                obj-addr)
      (while (not finish?)
         (tube-listen l eval-here))
      (destroy-tube-listener! l))
   (destroy: ()
      (set! finish? #t)))))
```

Figure 4.3: Using an advertisement to provide remote access to an object

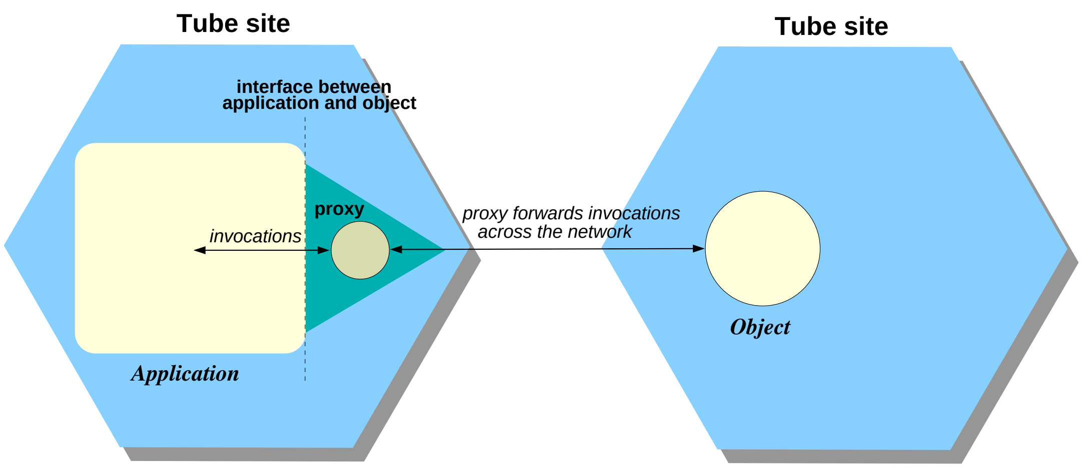

Figure 4.4: Using a proxy

`eval-here` to evaluate the expression and loops until the `destroy` method is called. `eval-here` allows programs to access symbols in scope when it is called (see Section 4.2.2).

A program wishing to use the object to add two numbers together would use the following expression:

```
(on (make-content-address
        'unspecified
        "#BAND(#FIELD(ObjectType=Adder)
               #FIELD(ObjectName=first-adder))")
    '(this 'add 519 302))
```

An object-based interface can be given to this operation. A *proxy* object is written that has the same methods as the real object but implements none of their functionality. Instead, the proxy's methods forward invocations across the network to the real object (see Figure 4.4). Programs can then be written that invoke the proxy as if they are invoking the object itself. If the proxy's methods contain the same type annotations as the real object then any invalid arguments can be detected before they are sent over the network. A proxy for the `first-adder` object is defined by the following expression:

```
(let ((obj-addr (make-content-address
                    'unspecified
                    "#BAND(#FIELD(ObjectType=Adder)
                           #FIELD(ObjectName=first-adder))")))
   (make-object () ()
      (add: ((x <integer>) (y <integer>)) -> <integer>
         (on obj-addr `(this 'add ,x ,y)))
      (destroy: ()
         (on obj-addr '(this 'destroy)))))
```

If this proxy is bound to the symbol `proxy` then the addition can be performed with the following expression:

```
(proxy 'add 519 302)
```

Proxies present an object-based interface for programs to call into. Using a proxy hides the exact means of communicating with an object from programs. The proxy is free to use whatever method it chooses for communicating with the object over the network. Section 7.4.5 for example discusses how proxies have been used to hide non-Tube communication from Scheme programs.

The Tube mobile code system can marshal objects into byte sequences. This allows them to be transmitted over a network or held in persistent store. Making a proxy transmissible allows programs to load it dynamically from the network when they need to communicate with its object. They

```
(make-object ()
         ((l (make-tube-listener 0))
          (obj-addr (tube-listener-address l))
          (finish? #f))
   (advert: ()
      (make-object () ()
          (add: ((x <integer>) (y <integer>)) -> <integer>
             (on obj-addr `(this 'add ,x ,y)))
          (destroy: ()
             (on obj-addr '(this 'destroy)))))
   (content: () -> <string>
      "<ObjectType>Adder</ObjectType>
       <ObjectName>second-adder</ObjectName>")
   (query: () -> <string>
      "#BAND(#FIELD(ObjectType=Adder)
             #FIELD(ObjectName=second-adder))")
   (add: ((x <integer>) (y <integer>)) -> <integer>
      (+ x y))
   (start: ()
      (advertise this)
      (while (not finish?)
         (tube-listen l eval-here))
      (destroy-tube-listener! l))
   (destroy: ()
      (set! finish? #t)))
```

Figure 4.5: A publishable proxy

do not have to be linked with the proxy's implementation before they start executing. Storing proxies in advertisements allows programs to receive them automatically when they query the trader.

For example, Figure 4.5 shows the definition of a second version of the Adder object that publishes a proxy instead of an address. The object's address is embedded inside the proxy; when the proxy is loaded into a program, it knows the location of the object without having to go back to the trader.

Tube sites provide two functions, `advertise` and `retrieve`, to manage advertising of proxies.

The first function, `advertise`, takes a single object as an argument and advertises a proxy for it in the trader. It expects the object to define the following three methods:

- `content`, which returns SGML markup describing the object
- `query`, which returns a query corresponding to the content
- `advert`, which returns a proxy for the object

`content` and `query` are used by `advertise` to make a content address for the object. `advertise` constructs an advertisement for the object that consists of the content address and the proxy returned by `advert`, marshalled into a byte sequence. In this example, only one proxy is returned for advertising by the object. The `content`, `query` and `advert` methods can also return lists of values so that proxies implementing different communication behaviours can be advertised. Different proxy implementations can then be written to address different network conditions or to distribute their loads between client programs and the object differently, for example.

The object shown in Figure 4.5 advertises itself, in the `start` method. This puts an advertisement in the trader that names the object `second-adder` and contains a proxy allowing programs to communicate with it.

The second function, `retrieve`, takes a query as an argument, which it submits to the trader. If a matching advertisement is returned, it is scanned for a proxy. If a proxy is found in the advertisement, it is unmarshalled and returned to the caller.

For example, a program running on a Tube site or using an embedded interpreter can then use the following expression to obtain a proxy for communicating with it:

```
(retrieve "#BAND(#FIELD(ObjectType=Adder)
                 #FIELD(ObjectName=second-adder))")
```

4.4.5 Factories

A factory is an object that creates instances of other objects. Figure 4.6 shows the definition of a factory for Adder objects. It has a single method, `create`, for creating named instances of the Adder object shown in Figure 4.5.

The `create` method takes one argument, the object's name. This is used in the advertisement for the created object.

The factory defines its own `content`, `query` and `advert` methods and is advertised with a call to `advertise` after it is defined. Rather than advertise a proxy for the factory, the whole factory is advertised by returning `this` from the `advert` method. Programs retrieving the factory advertisement therefore get a complete copy of it and can proceed to instantiate Adder objects themselves, in the same address space.

The sequence of events is as follows (see Figure 4.7):

1. a factory is defined on and advertised from a Tube site
2. a program wishing to create an Adder object retrieves a copy of the factory from the trader
3. the program instantiates an Adder object using its copy of the factory

```
(let ((hello-factory
          (make-object () ()
             (advert: ()
                this)
             (content: () -> <string>
                "<Factory>Hello</Factory>")
             (query: () -> <string>
                "#FIELD(Factory=Hello)")
             (create: ((name <string>))
                (make-object ()
                         ((l (make-tube-listener 0))
                          (obj-addr (tube-listener-address l))
                          (finish? #f))
                   (advert: ()
                      (make-object () ()
                          (add: ((x <integer>) (y <integer>)) -> <integer>
                             (on obj-addr `(this 'add ,x ,y)))
                          (destroy: ()
                             (on obj-addr '(this 'destroy)))))
                   (content: () -> <string>
                      (sprintf "<ObjectType>Adder</ObjectType>
                                <ObjectName>%s</ObjectName>" name))
                   (query: () -> <string>
                      (sprintf "#BAND(#FIELD(ObjectType=Adder)
                                      #FIELD(ObjectName=%s))" name))
                   (add: ((x <integer>) (y <integer>)) -> <integer>
                      (+ x y))
                   (start: ()
                      (advertise this)
                      (thread-create-detached
                         (while (not finish?)
                            (tube-listen l eval-here))
                         (destroy-tube-listener! l)))
                   (destroy: ()
                      (set! finish? #t)))))))
   (advertise hello-factory))
```

Figure 4.6: A publishable object factory

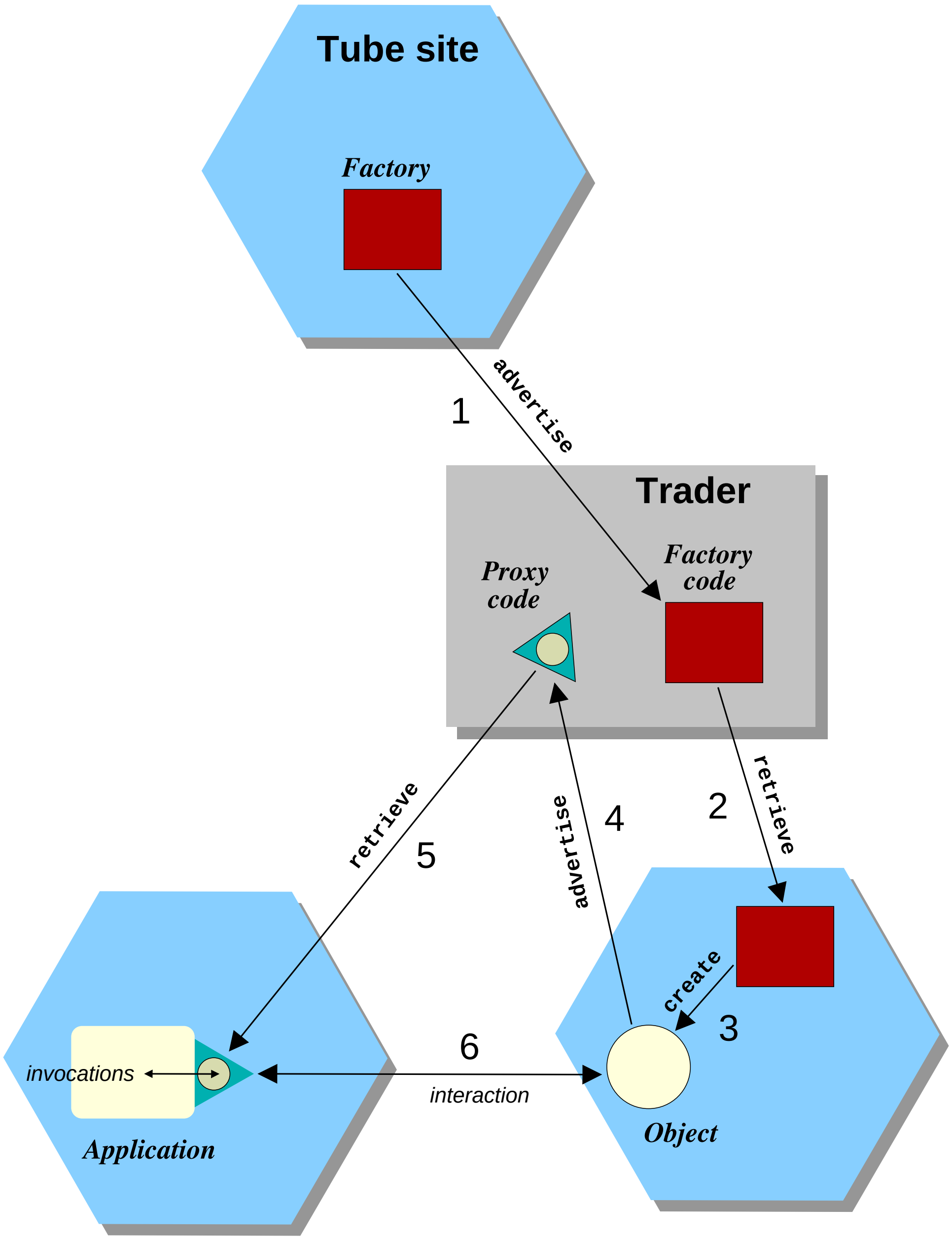

Figure 4.7: Using a factory to create and communicate with an object

4. the Adder object advertises a proxy so that other programs can communicate with it
5. applications retrieve copies of the proxy
6. they use the proxy to communicate with the object (see Section 4.4.4)

Note that the `start` method defined for objects created by the factory uses `thread-create-detached` to run the loop that listens for connections in a separate thread. This allows the object's `start` method and hence the factory's `create` method to return as soon as an object is created. The factory can thus be used to create many objects that are active concurrently. `thread-create-detached` behaves like `thread-create` (see Section 2.3.1) except that the thread's exit status is discarded.

4.4.6 Dynamic proxies

Using mobile proxies for communicating between a client program and a remote object moves the interface between the two from the network into the Tube site on which the client is running (see Figure 4.4). Through proxies, an object's programmer can use knowledge he has about the object's functionality to manage how client programs communicate with it.

A further step would be to change proxy implementations at run-time (new code would be sent by the server) in order to take account of changes in server functionality and network conditions, for instance.

The object system described in Section 4.3 allows objects to change their inheritance hierarchy. The `add-super!` method adds an object to the list of objects that an object inherits from. The `remove-super!` method removes one. For example, the following expression returns a list containing the numbers 1 and 2:

```
(let* ((obj1 (make-object () () (doit: () 1)))
       (obj2 (make-object () () (doit: () 2)))
       (obj3 (make-object obj1 ()))
       (val (obj3 'doit)))
   (obj3 'remove-super! obj1) (obj3 'add-super! obj2)
   (list val (obj3 'doit)))
```

This mechanism can be used for changing a proxy's implementation dynamically. A *surrogate proxy*, which inherits from the real proxy, is advertised by an object in the trader. To programs, it appears to function like the real proxy. The surrogate proxy listens for connections from the object, on which new proxy implementations are sent. When a new proxy is received, the surrogate uses `remove-super!` and then `add-super!` to replace the old one (see Figure 4.8). For example, the `advert` method of Adder objects might return a surrogate proxy defined by the following expression:

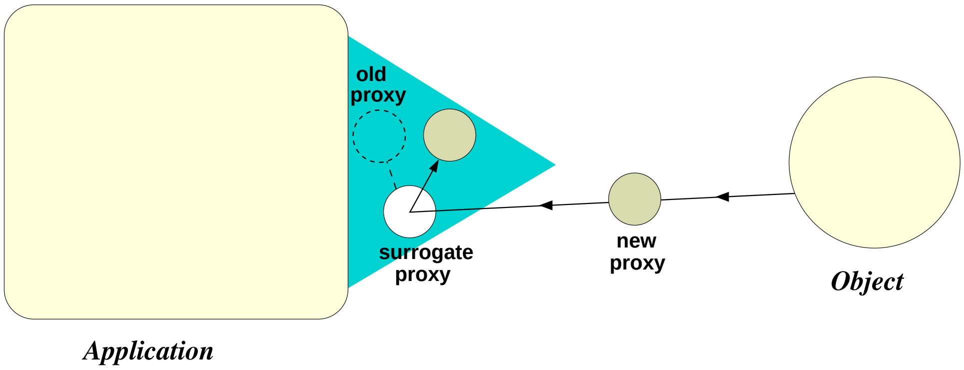

Figure 4.8: Dynamic proxies

```
(let ((proxy (make-object () ()
                 (add: ((x <integer>) (y <integer>)) -> <integer>
                    (on obj-addr `(this 'add ,x ,y)))
                 (destroy: ()
                    (on obj-addr '(this 'destroy))))))
   (make-object proxy ((l (make-tube-listener 0))
                       (this-addr (tube-listener-address l)))
       (start: ()
          (thread-create-detached
             (on obj-addr `(this 'register-proxy ,this-addr))
             (while proxy
                (tube-listen l (lambda (exp)
                                 (this 'remove-super! proxy)
                                 (this 'add-super! exp)
                                 (set! proxy exp))))))))
```

The symbol `obj-addr` is assumed to be bound to the address of the object that generated the proxy. Before listening for connections, the object is contacted and told the address to which new proxy implementations should be sent (it is assumed to have a method called `register-proxy` for doing this). The surrogate proxy's thread can be terminated by sending it `#f`.

Like the other objects defined in this chapter, the proxy and the surrogate proxy do not guard against concurrent access to them from multiple threads. When used by more than one thread at a time, objects should arrange mutual exclusion from sections of code that are not safe for concurrent access.

The object can send new proxies for the client program to use by connecting to the surrogate proxy. For example, assuming that the surrogate's address is bound to the symbol `surrogate-addr`, the object can push part of its functionality (adding two numbers together) onto the client-side of the network with the following expression:

```
(dispatch surrogate-addr
          (make-object () ()
              (add: ((x <integer>) (y <integer>)) -> <integer>
                 (+ x y))
              (destroy: ()
                 (on obj-addr '(this 'destroy)))))
```

When the client program invokes the `add` method of this proxy, no network access is instigated. The `add` method is easily distributed because it is stateless and does not need to access resources at any particular location.

Object functionality might be moved into client programs according to network conditions or machine load, for example. Doing more in the proxy can reduce network usage because operations do not have to be carried out remotely from a program. However, if an operation modifies state held by the object (and replicated in proxies), then maintaining consistency between all the components is made more complicated. An object may also update proxy implementations to reflect changes in its functionality or extensions to the interface it wishes to present to programs. An updated proxy used by a number of programs should maintain support for old behaviour if required.

4.5 Related Work

4.5.1 Enhanced interaction

SQL [**ANSI92**], PostScript [**Taft85**], NCL [**Falcone87**], REV [**Stamos86**, **Stamos90a**, **Stamos90b**] and late-binding RPC [**Partridge92**] allow application level processing to be moved from client to server, in order to reduce network usage. They are discussed briefly in Sections 2.2.1 and 2.2.2.

Most Scheme implementations allow a program to obtain its execution environment and to specify the evaluation environment for expressions. Code injection combines this with the ability of the Tube to send expressions between programs over a network.

ANSAware [**ANSA93**] and most CORBA implementations (e.g. DIMMA [**Li94**] and OmniBroker [**OOC97**]) can multiplex invocations made between two objects onto a single connection in a manner that is transparent to the application programmer. The method for sharing connections described in Section 4.2.3 exposes this facility to applications so that they are always aware of the network connections being used.

4.5.2 Scheme object systems

Meroon/Meroonet [**Queinnec96**] is an example of an object system written in Scheme. Unlike Meroon, the object system described in Section 4.3 does not have the notion of object classes, which treat behaviour of object types separately from their instances. Rather, it has a single form, `make-object`, which defines both object instances and behaviour that other objects can reuse by inheriting.

This is done to make the object system as simple as possible for use as the basis of the distributed object system described in Section 4.4. In particular, the object system does not have to maintain a directory of class names, so that naming issues are deferred to advertisements (see Sections 4.4.1 and 4.4.2). However, this means that object definitions cannot be cross-referenced between Tube sites.

Also, an instance of an object has to be made before inheriting its behaviour. In cases where initialising the object's private data has some side-effect, this may not be desirable. This can be worked around by writing constructor functions that act like object classes in that they instantiate objects of a certain type. When creating a new object that inherits from another, the second can be made at that point in time, and does not have to exist beforehand. This makes the object system behave like Yasos [**Adams88**].

4.5.3 Dynamic distributed objects

Java Remote Method Invocation (RMI) allows code providing access to an object to be downloaded into client programs. It generates minimal client stubs from the Java object's source text but programmers can augment their behaviour by writing object classes that use them. These classes can be used by client programs in a similar way to proxies (see Section 4.4.4).

Java-based CORBA implementations (e.g. Java IDL [**Sun97a**] and OrbixWeb [**IONA97**]) also allow communication stubs to be delivered to a client program at run-time. OrbixWeb also allows programmers to write their own stubs (called *smart proxies*), which provide similar functionality to the proxies described in Section 4.4.4.

Unlike dynamic proxies, neither object classes that use RMI stubs nor smart proxies provide direct support for updating their implementations during execution.

4.6 Summary

The Tube mobile code system supports different abstractions for communicating between processes. This chapter showed how moving application-level processing can reduce network usage and allow servers to provide for the changing requirements of many clients. It introduced code injection, which allows mobile code to bind closely to programs that use it. It showed that the Tube's remote processing facilities support bi-directional client-server interaction over a single connection. A distributed object system was described that uses mobile code to implement dynamic communication proxies which can be delivered and updated at run-time.

Chapter 5

Mobile Code in Network Control

The use of mobile code in an ATM network control architecture is described. It is used to decentralise control and reduce network usage. Details are given of the integration of a mobile code system and an ATM control architecture. This is used to implement a facility which helps maintain coherence between the control architecture and the switches it manages.

This chapter addresses issues that have arisen from research described elsewhere [**Rooney97c**]. It discusses an implementation for the work proposed there.

5.1 Introduction

The ability to send executable code amongst computers allows the point of interaction between resources, such as software services, applications and hardware devices, to be moved.

Any resource has a defined way in which it can be accessed. If a resource is networked then it can be accessed by software running on computers other than its host. There will be a defined way in which this can occur. Rather than communicate across the network, code might be sent to talk with it on its host machine (see Figure 5.1). That is, if an application repeatedly communicates with a resource across the network, the number of network invocations made can be reduced by moving part of it to run on the same computer as the resource. Invocations instead take place on the machine, without network access until results are returned to the main part of the application.

Further, moving access closer to a resource can reduce the amount of data that needs to be transferred over the network. If the application performs any kind of filtering of data provided by the resource, for instance searching

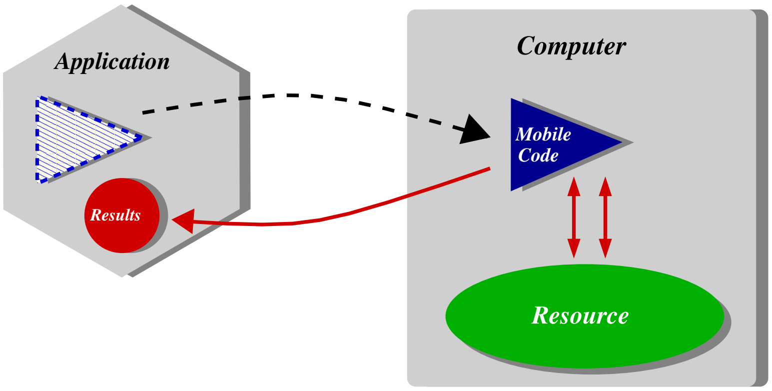

Figure 5.1: Moving computation to resources

or numeric processing, then moving (part of) it to the resource instead of moving the resource's data to the application may reduce network usage. This will be the case if the filtering reduces the amount of data (because less will be returned over the network) and if the saving more than offsets the cost of moving application code.

Operations involving analysis of large amounts of state obtained from a resource become more tenable when the analysing code can move to the state rather than the other way around.

The rest of this chapter discusses moving code close to resources in the context of network control. In the described application, the resources are switches and the code is sent by advanced ATM control architectures. We wish to minimise the amount of data transferred over the network by control architectures because network control is a common, low-level facility that manages the flow of data between all communicating applications.

This chapter does not address the nature of communication between applications once data flow is established. In particular, they may choose to make liberal use of the network resources assigned to them by a control architecture (the stateless servers described in Chapter 6 are an example of this).

5.2 Network Control

The term *network control* refers to functionality that helps to manage data flow between computers in a network. It includes the setup and termination of data transfer, routeing of data between sender and receiver and making sure that the states of devices connected to the network are consistent with each other. Examples of network control are management of virtual channels in ATM networks and routeing tables in IP networks.

Allowing network control software to send code close to the resources it manages promotes a decentralised and efficient approach. Certain control functionality can be moved close to network devices so that their states do not have to be transmitted to some central entity for analysis.

To test this proposition, the Tempest framework [**Merwe97**, **Merwe98**, **Rooney97b**] has been used as a context in which mobile code and network control can easily be combined. The Tempest enables diverse control architectures to be run over an ATM network simultaneously. The control architectures it provides are "open" because they define high-level interfaces which applications can use to access network control functionality, such as connection setup.

Briefly, the Tempest consists of:

- an interface to switches called Ariel
- a partitioner of the resources of a switch between control architectures called Prospero
- a network builder which dynamically creates virtual networks for control architectures
- a set of standard libraries to allow developers to create control architectures

The experiments described in this chapter use the Hollowman control architecture [**Rooney97b**], which implements ATM control in a distributed processing environment.

5.3 Mobile Code in the Tempest

[**Rooney97c**] has identified how mobile code can be applied to the Tempest. This section motivates the implementation described later by discussing the advantages of using mobile code in network control architectures.

Ariel provides a minimal interface to switches. Control architectures which make many invocations on them or transfer large amounts of switch state over the network would benefit by sending code to execute near to an Ariel interface.

Each Tempest control architecture maintains a view of the part of the network allocated to it, e.g. routeing information for switches. This view can never be fully synchronised with the actual state of the network because it is held at a higher level, in some control software running on a workstation. Control architectures see out-of-date state because it has to be transferred to them from switches over the network. They see the state of switches as they were when the state was sent, not when they receive and process it. The faster the network used to transfer the state, the more consistent a control architecture's view is with the actual state of a network. If the network fails, then the two become completely de-synchronised.

For effective management, a control architecture's view of the state of its network must be kept as coherent as possible with the actual state. This means that the network state must be transferred to the control architecture as quickly as possible. Using mobile code, control architecture functionality that needs to be tightly synchronised with network state can be moved close to network resources such as switches, reducing the distances their states have to travel. Control architectures can increase their synchronisation with the network by sending code that analyses switch state to run close to switches.

Allocation and release of network resources is complex and error-prone. This is because control architectures manage distributed resources — computers and network devices such as switches. At any time during a connection's lifetime, a computer or switch may not be able to meet its resource guarantees. All resources allocated to the connection across the network must then be freed.

The Tempest can provide automatic resource freeing (ARF'ing) in the form of a piece of mobile code which periodically goes to each of the switch interfaces in the control architecture's network and checks for inconsistent switch state. Inconsistencies can be verified against the control architecture's view of network activity and either cleaned up automatically as they are detected or flagged to a system administrator for further attention. Using mobile code to do this has the following advantages over some central entity periodically retrieving switch state:

- Network usage is greatly reduced; the code which checks for incoherent switch state moves between switch interfaces and makes invocations on them locally rather than having to transfer all switch state to some central location.

- The mobile code executing on the same host as the switch interface will be as closely synchronised as possible with the switch's state.

Using mobile code in conjunction with open network control architectures allows some control functionality to be implemented in an adaptive and efficient manner *that is consistent with* a wider view of the overall state of the network. Without the Tempest, ARF'ing code would not be able to check the inconsistencies it finds with known activity across the network as a whole.

The following section reviews related work on mobile code in network management. The implementation of automatic resource freeing is described in Section 5.7, using the integration of a mobile code system and a control architecture that is described in Section 5.5. Testing of this integration is discussed in Section 5.6.

5.4 Related Work

The WAVE project [**Sapaty94**] propagates "language strings" onto the network and interprets them at each node. Emphasis is placed on the lack of centralised supervision of these and the dynamics of such a system. One application that WAVE has been used for is intelligent management of open computer networks:

> These structures may provide self-organisation and self-recovery from complex failures as well as forming the basis for integration of other (distributed and heterogeneous) systems. [**Sapaty94**]

This is relevant to the work discussed in this chapter because they have distributed interpreters throughout their networks. However, whereas our mobile code is part of, and launched from, an open ATM control architecture, they have "integrated with a usual management system by establishing two channels through Unix for transferring activation code and data between the systems in both directions" [**Götz96**]. We argue that mobile code must have access to the internal interfaces of the control architecture, and that use of mobile code in network management is best suited to non-critical maintenance procedures and not for essential implementation and control. Finally, our work is based on ATM and mobile code has direct access to switch state; WAVE is implemented over TCP/IP and interfaces with a traditional network management system.

The MAGNA project [**Magedanz96**] has identified the deficiencies of a classical client-server based distributed processing environment as being inflexible and difficult to adapt and extend. They have proposed using mobile code in order to add autonomy, mobility and adaptability to the TINA-C proposal [**Barr93**]. MAGNA discusses dynamically extending the control architecture with mobile code both to perform client-specific functionality and for management delegation. The Tempest uses connection closures [**Rooney97a**] to do the former; however, this chapter concentrates on an implementation of the latter — mobile code that is configured and initiated from the control architecture itself and which communicates with the Ariel switch interface directly.

Another approach is to make switches programmable by putting interpreters on them. Programs are sent along the data path and tagged for interpretation. Switches detect these programs and execute them as they pass by. This allows for intelligence to be downloaded onto the switches themselves, achieving locality of access and the ability to change the functionality of services that a switch can offer. Netscript [**Yemini96**] uses this for "dynamic deployment of software at all nodes"; the Active Networks project [**Wetherall96**] and Switchware [**Smith96**] use this approach to allow data to carry their own networking algorithms with them.

The Tempest has defined in the Ariel interface a low-level API for communicating with switches and developed switchlets for allowing multiple control architectures to be imposed on an ATM network. The approach taken in this chapter sits on top of Ariel and can achieve both locality of access, by moving code close to switches, and management flexibility in combination with the Tempest. We believe the advantages of putting interpreters onto switches can be achieved whilst keeping network intelligence at a high level — allowing switches to be simple and inexpensive.

Goldszmidt's Management by Delegation (MbD) project [**Goldszmidt95**] uses a similar approach to ours. It uses *elastic processes*, which allow computation to be moved close to network resources. An elastic process is a process that can move (part of) itself. The term is language-independent, and describes an approach rather than an implementation. MbD is proposed for network management in order to provide locality of access, to overcome network latency and to add flexibility to the "rigid servers that provide data access capabilities to platform clients" [**Goldszmidt95**].

Our approach is similar to Goldszmidt's in the sense that we use mobile code (elastic processes) to overcome network latency and make different resource access patterns available. However, we rule out moving new computation onto network elements themselves. Rather, our code sits on top of the Ariel switch interface, which provides uniform access to switch functionality and the ability to participate in multiple control architectures. Our code also sits below Tempest control architectures, which allows it to see and contribute to a more complete view of the network as a whole.

BT have investigated the use of mobile code in making networks robust and adaptable [**Appleby94**]. They have devised mobile programs that alter routeing tables in order to manage load across a network and run them under simulation. We believe mobile code is suitable for carrying out such "background processing" in the network; indeed, our Automatic Resource Freeing mobile program (see Section 5.7) does this in a real network.

Finally, [**Baldi97**] proposes the use of mobile code in network management. Like this chapter, it sees the benefits as being adding flexibility to the way resources are accessed and reducing network usage. However, it does not identify a specific problem or describe an implementation. We have used a mobile code system and an innovative ATM control architecture to implement a specific network control function that benefits from code mobility.

5.5 The Tube and The Hollowman

The Tube mobile code system described in Chapters 2 and 3 has been interfaced to the Hollowman control architecture. The Hollowman is an innovative Tempest control architecture which devolves control from ATM switches into an application-level distributed processing environment. Because func-

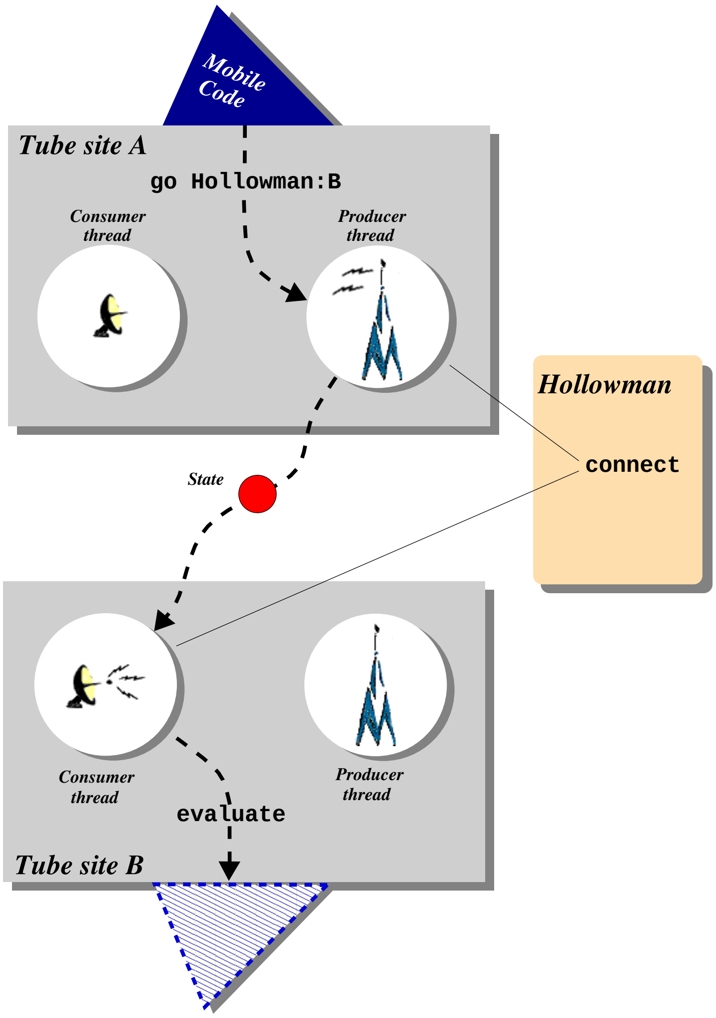

Figure 5.2: Tube/Hollowman integration

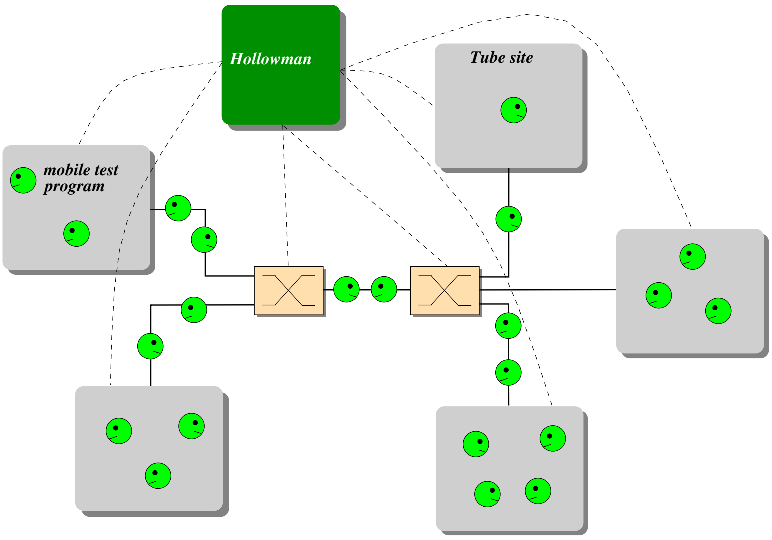

Figure 5.3: Testing Hollowman using mobile code

tionality is available at a high level, convenient encapsulation can be given to Tube programs.

The Tube has been linked with the Hollowman software library, using automatically generated stubs to glue the two together. These allow code running under the Tube to access the full functionality of the control architecture. Hollowman uses a CORBA [**Siegel96**] distributed processing environment for its internal communication; the Tube is hidden from this however — it just calls functions provided by the Hollowman stubs.

The Hollowman services used by the Tube are:

- importing offers (for data producers and consumers) from a trader
- invoking third-party connection over an ATM network between a producer and a consumer
- closing down ATM connections

These services are used in the Tube to send data over ATM connections between computers (see Figure 5.2). Each Tube execution site registers through Hollowman both data producer and consumer threads. A program running on a Tube site which wishes to send some data to another site imports the current site's producer offer and the destination site's consumer offer and then invokes third-party connection between them. The specifics of how this is achieved and the trading facilities used are internal to the Hollowman.

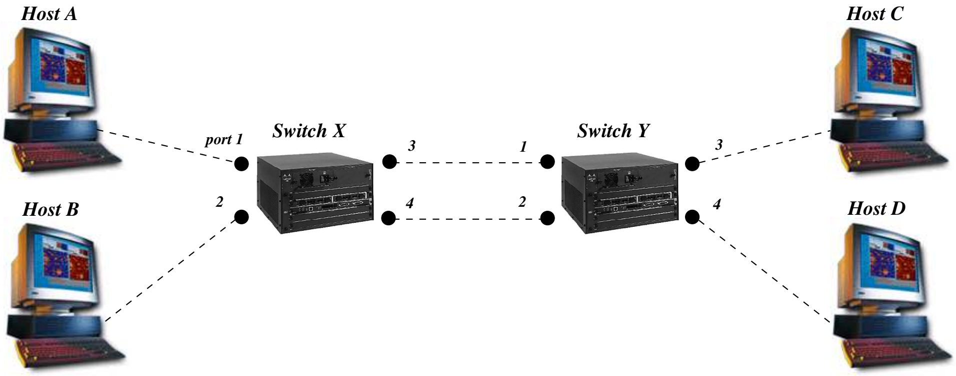

Figure 5.4: An ATM network

At this point, an ATM connection is in place. The data is then given (via shared memory) to the producer. At the destination site, the consumer receives the data from the ATM connection and gives it (again via shared memory) to its Tube site for evaluation. The connection is then destroyed by the consumer. Any type of data made transmissible by the Tube can be sent in this manner, including executable code.

This procedure for sending Tube data over ATM networks using the Hollowman is made easier to use by hiding it behind a special type of address. In the Tube, network addresses may be tagged with their type (see Section 2.5.3). Handlers (e.g. for name resolution or connection) are registered for each type. When data is sent to an address, the appropriate connection handler is invoked. A special Hollowman address type performs the connection procedure described above; in this way, programs only have to specify that they want to send data to a Hollowman address, give the name of the destination's consumer offer and the connection handler does the rest.

Making Hollowman functionality available in the Tube allows it to be called at run-time from an interpreter. This is useful for creating and manipulating connections by hand at a high level. The ability to send Tube data over an ATM network using the Hollowman allows the control architecture to be tested and resource freeing to be implemented using mobile code.

5.6 Testing the Hollowman

Mobile code has been used to test the stability of the Hollowman when many concurrent invocations are made on it from different locations in a network (see Figure 5.3).

In the test, a Hollowman-enabled Tube execution site is placed on each computer in an ATM network. A program has been written which runs on

Tube sites and can move itself between computers via the network by using the Tube/Hollowman interface. It repeatedly moves around the network from host to host. At each stage, the choice of which host to move to is made randomly and the time before it moves on has a random element too.

By varying the number of these moving programs in the network, the Hollowman control architecture is tested at varying levels of concurrency. The more there are active in the network, the greater the number of connection requests handled by the control architecture at the same time.

This test was easily turned into a simple demonstration by giving each program a graphical display which it carries around from host to host. Spectators type comments into the display, which are remembered by the program as it moves around.

5.7 Automatic Resource Freeing

As discussed in Section 5.3, control architectures should be provided with a means of ensuring that the network resources allocated to them remain in a clean state. Using mobile code to do this was shown to reduce network usage and provide close synchronisation between the control architecture's view of the network's state and the actual state of the switches.

An ATM network consists of a number of computers interconnected via switches (see Figure 5.4). The principle task of a switch is to help route data from its sender (producer) to its recipient (consumer). In order to do this, a switch maintains a mapping which determines where data it receives should be sent. More than one switch may be involved in this routeing and data may be sent to more than one consumer (i.e. it can be multicast).

Data is sent in small cells from one switch to another until it reaches its destination. Cells are delivered to the consumer in the same order as they left the producer. Since cells are small, the time they take to reach the network is short. This makes ATM suitable for traffic with real-time requirements, such as audio and video. Cells may be dropped by an ATM network; the consumers of Tube data described in Section 5.5 notice when they are.

The physical links between switches are plugged into *ports*. Each switch has a number of ports, which are sockets into which cable from another switch or a computer is plugged. The switches' mappings allow data to be transferred between two hosts. The end-to-end connection is called a *virtual circuit.* Quality of service parameters can be specified for a virtual circuit, such as the bandwidth reserved for its data. Virtual circuits are only set up if the resources they require are available in the network. The resources reserved for a virtual circuit are guaranteed after it is set up.

For instance, in Figure 5.5, if Host A wishes to send data to Host D then Switch X's mapping might be:

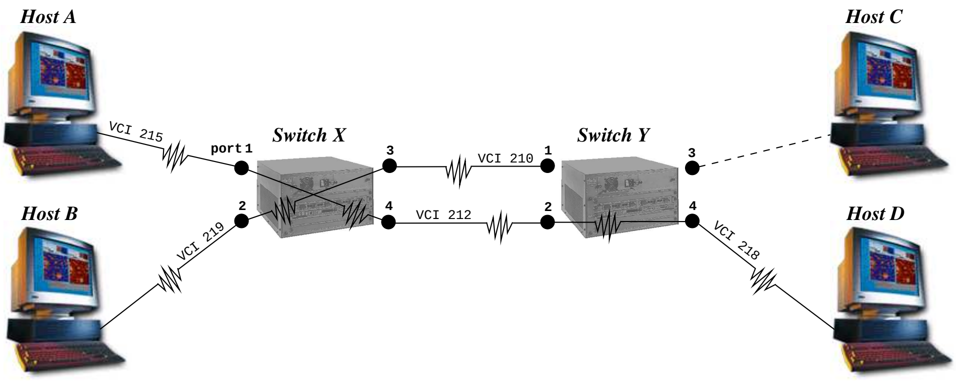

Figure 5.5: Connections in an ATM network

$$\text{Port } 1 \rightarrow \text{Port } 4$$

and Switch Y's mapping would be:

$$\text{Port } 2 \rightarrow \text{Port } 4$$

In fact, things are a little more complex. Physical links between components are multiplexed to allow them to carry more than one virtual circuit. Links between network devices are partitioned into virtual channels and virtual paths. A virtual channel carries a virtual circuit between two devices; a virtual path may carry a group of virtual channels. Each virtual path is assigned a unique Virtual Path Identifier (VPI) and each virtual channel is assigned a unique Virtual Channel Identifier (VCI). A virtual circuit consists of a number of virtual channels that carry the connection from producer to consumer via switches. Between switches and between switch and computer, VPIs and VCIs must match; within a switch, mappings may change a virtual circuit's VPI and VCI. The Hollowman's current implementation uses VPI 0 to carry virtual circuits between each switch in their paths. In the following discussion, VPI 0 is assumed for all VCIs.

For instance, in the virtual circuit between Host A and Host D, Host A might send its data on VCI 215. Switch X receives it on Port 1, VCI 215. It might then map this to be output on Port 4, VCI 212. Switch Y then receives the data on Port 2, VCI 212. If this is mapped to Port 4, VCI 218 on Switch Y, then Host D receives the data on VCI 218.

In the Tempest, control architectures such as the Hollowman set up virtual circuits between data producer and consumer processes running on computers connected to an ATM network.

A control architecture has a view on the virtual circuits in place over its networks. Being held in software running on a workstation, this view is not closely synchronised with the state of the switches' mappings. When the

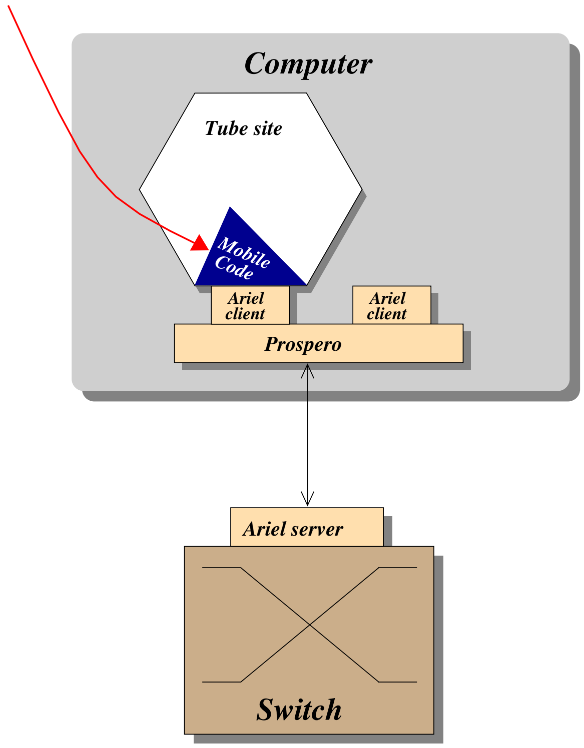

Figure 5.6: The Tube and Ariel

switches' state becomes different to the control architecture's view, it must be cleaned up so that the two are re-synchronised. Host-to-host connections (virtual circuits) may exist that the control architecture no longer knows about.

Broken connections may exist too. A connection is broken if one switch has a mapping which sends data to a port connected to another but the second switch does not have a mapping to receive the data. For instance, if Switch X in Figure 5.5 has the following mappings:

Port 1 VCI 215 $\rightarrow$ Port 4 VCI 212
Port 2 VCI 219 $\rightarrow$ Port 3 VCI 210

and Switch Y has the following mapping:

Port 2 VCI 212 $\rightarrow$ Port 4 VCI 218

then the connection between Host A and Host D is still in place but there is also a broken connection originating from Host B. Port 1, VCI 210 is not in use on Switch Y. The mapping Port 2 VCI 219 $\rightarrow$ Port 3 VCI 210 on Switch X is a candidate for removal.

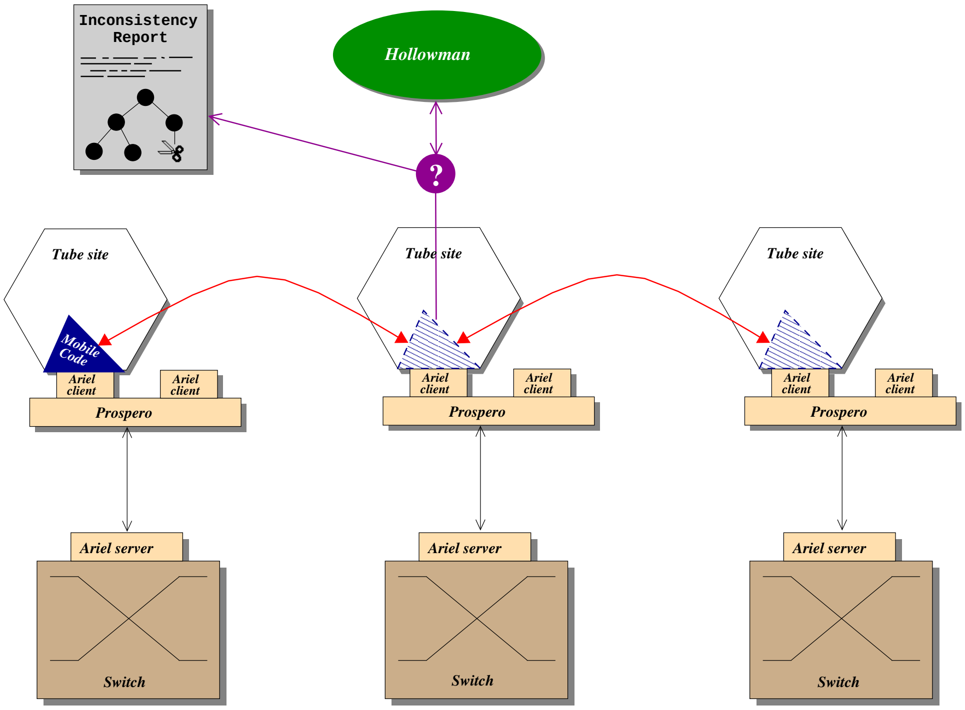

Figure 5.7: Automatic Resource Freeing

Before mappings are removed from switches, the control architecture must be consulted because it knows about activity across the network as a whole. For example, although a mapping in a switch may not be in use for transferring data, the control architecture might want to keep it in place for re-use when setting up a new connection. Any activity that cleans up switch state must be coordinated with the appropriate control architectures, in order to check which connections they know about and to check that broken connections are not being cached for later use.

Mobile code is used to obtain and check the consistency of switch states because the states do not have to be transferred to the Hollowman's location, reducing network usage and providing close synchronisation.

The low-level Ariel switch interface is used to communicate with switches. Rather than defining static interfaces to higher-level functionality, using mobile code allows for a more flexible approach in which application-specific access is moved to the location of the switch's Ariel interface.

Automatic Resource Freeing (ARF'ing), using mobile code to check the consistency of switch state, has been implemented using the Tube and the Hollowman. For each switch in the network, a Tube execution site is created on the workstation closest to it (i.e. that has the shortest path to it or is directly connected to it, see Figure 5.6). These Tube sites make an Ariel function available to programs which returns the state of a switch via shared memory.

Mobile code to implement ARF'ing has been written. Programs are sent out by the Hollowman control architecture and move around the ATM network, using the Tube/Hollowman integration to visit each switch's associated Tube site (see Figure 5.7). At each site, the switch's state is retrieved. Because the Tube provides state-saving facilities, each mobile ARF'ing program can carry about with it a view on the state of the network. When retrieving a switch's state, it can check for inconsistencies against the state of the previous switch it visited. Inconsistencies must be checked with the control architecture before further action is taken because it has a wider view.

The following parameters may be varied:

- the number of ARF'ing programs active at any one time
- how quickly they move
- whether they clean up switch state themselves or simply notify a system administrator of inconsistencies
- how often they check with the control architecture for its view of network activity
- the frequency at which they execute; as more inconsistencies are found, this can be increased, allowing the process to be adaptive

ARF'ing has been implemented which uses both mobile programs that clean up inconsistent switch state and that display notification of broken connections (see Figure 5.8). In the figure, a multicast connection from the host *bailey* is good for data transfer to the host *scilly* but broken in two other places (Port 3, VCI 214 and Port 8, VCI 216 on the switch *aynho*).

The sizes of the ARF'ing programs are around 1 kilobyte. This overhead is offset because only one set of mappings has to be transferred between Ariel interfaces, whereas the states of all switches have to be sent to one location in a centralised solution. Processing is distributed and, given knowledge of network topology, the distances ARF'ing programs travel can be minimised. ARF'ing programs in regular use may be cached at Ariel interfaces so that they only have to be sent to them once. The cached programs may then exchange switch state amongst themselves.

Control architectures are free to implement their own methods for ensuring coherence between their views of network activity and the state of the switches. They can send to Ariel interfaces arbitrary resource freeing functionality that is tightly synchronised with switch state because it executes close to the switch. However, if they do not wish to do so, they may rely on any problems eventually being cleaned up by the mobile ARF'ing programs that the Tempest provides.

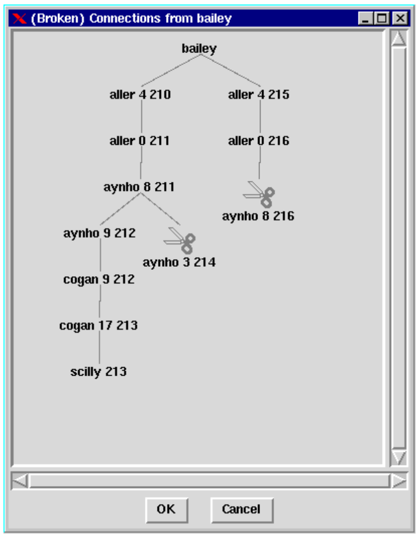

Figure 5.8: Broken multicast connection

5.8 Summary

Mobile code can help to provide decentralised, adaptive and efficient network control. This chapter has described the integration of a mobile code system and an ATM network control architecture. It demonstrated the use of this implementation to send mobile code over ATM networks, for testing the control architecture and cleaning up unused resources. The implementation runs mobile programs within the control architecture and above a common switch interface that is responsible for managing access to hardware. They are not transmitted in the data path or loaded onto switches. Mobile code is applied to network control in a way that makes use of control and management functionality, such as access control, connection setup and resource partitioning, but does not replace it.

Chapter 6

Stateless Servers

A way of arranging client-server interaction is described which moves all persistent knowledge about applications from clients and servers into the documents they exchange. This can be used to support ubiquitous computing with many lightweight clients and a small number of heavyweight servers. Both clients and servers are stateless, needing to maintain no application-specific software. Implications for application mobility, logging of user activity and handling user interface events as part of program flow are discussed in relation to an implementation made over the World-Wide Web.

6.1 Introduction

6.1.1 Motivation

Ubiquitous computing systems involve many machines. For deployment on a large scale, each should be as simple and inexpensive as possible. End-user machines should ideally do nothing more than display information, rather than run applications loaded from the network. Obtaining an application (from a storage server) should be a simple matter of selecting from a catalogue. It is difficult to store state or maintain execution threads on servers permanently for each client since their number is potentially unbounded, as is the duration of any client-server session. Strategic use should therefore be made of powerful processing servers that provide execution facilities only.

Commercial development of the *network computer* [**Oracle96**] has achieved simpler workstations, for example the JavaStation [**Sun96**] and the Acorn Network Computer [**Acorn97a**], but at the expense of making servers more complex. All permanent storage is associated with servers and network computer clients download applications from them.

As computers become faster, it becomes feasible to build inexpensive machines from commodity hardware which act as general processing servers [**Becker95**, **Anderson95**]. Their only task is to read code sent to them by clients, execute it and return the results. Knowledge about application functionality can be held elsewhere and sent to them. Permanent storage is provided as a separate facility; they simply provide processors and memory on which anyone can rent time. This chapter shows how applications can be written to execute on such machines but not to reside there continuously during their lifetimes.

Interactive applications without a real-time requirement do not need a long-term presence on client machines. They only require to interact at certain times with the user through an interface that is perhaps specified in a document markup language. Client machines for these applications can be simpler than network computers because they do not have to run new software; the applications are run on servers and user interfaces are sent to the client when appropriate.

It would be inappropriate to run interactive applications on general processing servers because they would require a permanent presence there while waiting for the users' input. The number of such users could be large and too many applications permanently residing on a server might result in it being overloaded or charging clients for excessive use. In the case that a client does not return with the user's input, its state would be left residing on the server indefinitely.

Network bandwidth is becoming more plentiful. Video-on-demand is planned for the home [**Acorn97b**], which requires data transfer rates in the order of 3 megabits per second. With high-speed networks, it is feasible to hold application state in the messages exchanged between client and server.

6.1.2 Exchanging application state

Servers which support state-saving of programs can save applications as dormant parts of the user interfaces they generate. This chapter looks at applications which move to the server when they need to be executed and move to the client as dormant state when they want to interact with the user. Servers are stateless because they hold no enduring knowledge of applications. User interfaces (documents) are stateful because they are used to store application state.

Any application running on a computer has a state. For an application that runs to completion on one machine, its state at any one time is held in the memory and processor. A client-server based application involves interaction between a client process and a server process over a network. Its state can be held in a combination of the following three locations:

The client A process on the client side might be running.

The network The client may have dispatched a request or the server may have returned some pertinent results. Communication also passes through the operating system.

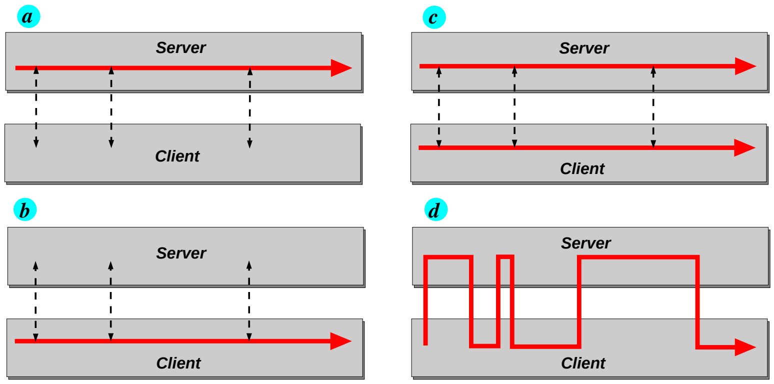

Figure 6.1: Application residence
(a) On a server
(b) On a client
(c) On both client and server
(d) Alternating between client and server

The server A process on the server side might be running.

Where application functionality is held varies, according to resources available on the client side, server side and in the network. The complexity of the application and of the data sent across the network are influential too. In current systems, knowledge of application functionality persists in the client, in the server or in a combination of both (see Figure 6.1a-c).

The term *session* is used to denote an instance of a client-server application. A session can involve many interactions (and connections) between the client and server components. However, the state of a session must be remembered in the client, in the server or in the data transferred between the two.

There is currently no way of supporting the use of computers for clients that can only run pre-installed code whilst at the same time using servers which hold no application-specific code and do not remember session information.

This chapter discusses a technique for remembering an application session's state in information exchanged between the computers used to provide client side and server side computation. Persistent knowledge of an application is held neither in the client nor in the server (see Figure 6.1d). Exchanging application state between client and server places a greater burden on the network than holding it permanently in client or server. Distributed systems that have very lightweight client machines, fast networks and servers that handle many requests and have plenty of processing power and memory are particularly suited to this technique.

Commercial implementations of this technique should assure the security of application state that is transferred over the network. However, alternative forms of ubiquitous environment must protect sensitive data too. They must provide encryption or authentication facilities, which can be used to protect application state. The extra concern with placing application state alongside sensitive data is that untrusted parties that do manage to breach security obtain not only the data itself but also code that can be used to interpret it. However, if someone is prepared to expend considerable effort to obtain your sensitive data then presumably he has the wherewithal to make use of it. The prototype implementation described in Section 6.4.2 does not address security issues.

Section 6.2 shows how a mobile code system can be used to capture and save application state. Section 6.3 discusses related work. Section 6.4 describes an implementation made using the World-Wide Web. Section 6.5 discusses the relationship between an application and the user interfaces into which its state is embedded. Section 6.6 presents an application written using the Web-based implementation that embeds its state in Web pages and performs only transient computation on servers. Section 6.7 shows how the ability to save application state can provide a powerful logging mechanism. Section 6.8 discusses uses of being able to move application state around once it is saved into user interface documents. Finally, Section 6.9 summarises the chapter.

6.2 Exchanging Application State with Mobile Code

6.2.1 Saving state using higher-order mobile code

A mobile code system is ideal for placing application state in data sent between a client and a server. Higher-order mobile code systems, which support state-saving of closures and continuations, allow (parts of) applications to be transparently saved into byte-streams. Applications can be restored and restarted from saved byte-streams.

The Tube mobile code system described in Chapters 2 and 3 provides a transparent and portable method for saving applications written in Scheme.

6.2.2 Running applications on stateless clients and servers

Using higher-order mobile code, applications can be marshalled for saving to persistent store or transmitting over networks. This can be used to embed client-server applications in a user interface displayed at the client when input is required. The following progression is proposed for such an application so that it does not impose a permanent burden on the server and the client does not have to run any new code (see Figure 6.2):

1. The user simply clicks on the name of an application in a document

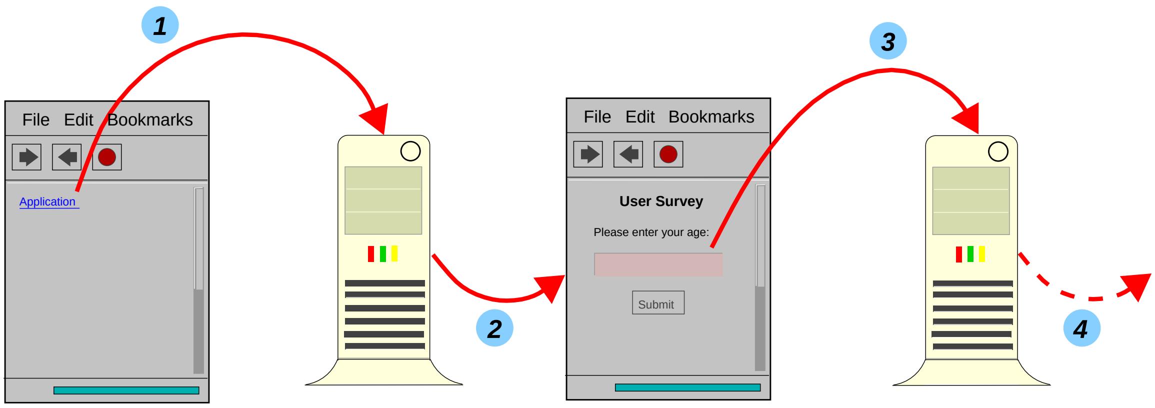

Figure 6.2: Application state in documents

browser. The browser extracts the application from the document and sends it to a generic mobile code server for execution there.

The application starts up on the server.

2. If execution can proceed until termination without further user interaction, results are sent back to the client (browser) and the application terminates.

   If on the other hand user interaction is required, the application embeds its current state in a document sent back to the client asking for some input. The application then terminates itself at the server. It has become dormant and moves back into the client.

3. The client renders the document it receives, allowing the user to enter the required input. When the input has been entered, the user simply clicks on a button to submit it. The client extracts the application's state from the document and sends it, along with the user's input, to the mobile code server. The mobile code server restarts the application in its previous state and gives the user's input to it.

4. The application sends itself and some results to the client and the cycle repeats.

There are two points to note here. Firstly, the application's state always accompanies its point of control. That is, when the user is providing input, the application is there with the document in the client; when the user has provided input and is waiting for the server to respond, the application has moved there to do the processing.

Secondly, the application can store different states in the document sent to the client containing the results. Each one might represent the application

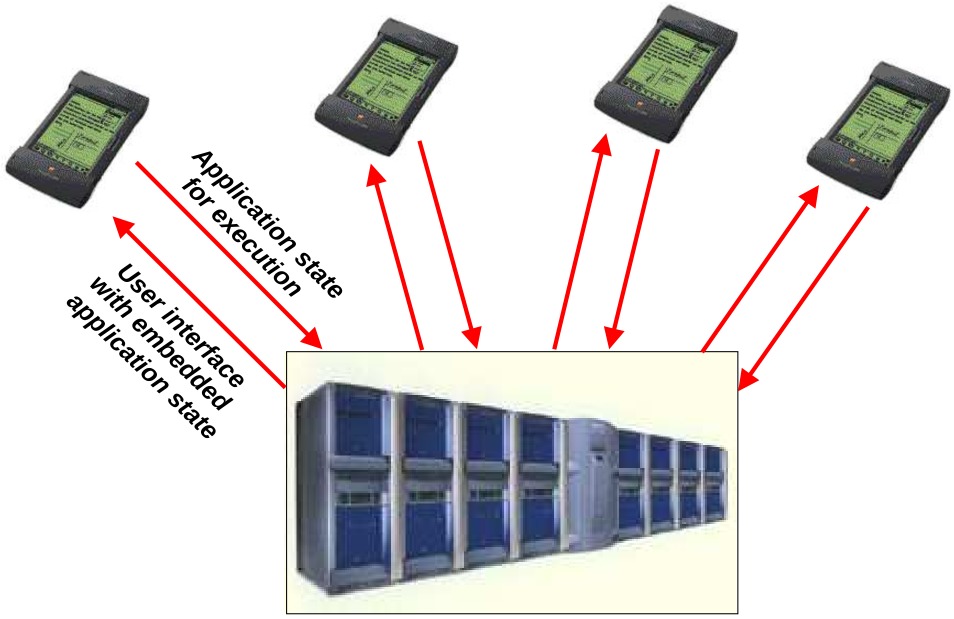

Figure 6.3: Stateful documents, stateless servers and lightweight clients

following a different path of execution. The user controls which one is sent back to the server for restoration through the input he provides.

A client-server application using this technique has the following characteristics:

- Clients are lightweight and stateless. They only have to render results and submit embedded data to servers. Therefore, they will not be suitable for user interfaces with real-time requirements (e.g. three-dimensional graphics viewers, multimedia presentations and games).

- Servers are completely stateless.

- Documents exchanged between clients and servers are heavyweight. They are *stateful documents*. This is not necessarily a problem if network resources are plentiful.

- A document sent to the client replaces the previous one. If the document's markup language does not support splitting it up into areas that can be updated individually, the user will see the whole display being refreshed each time. This might be a problem for applications that require finer grain control of a user interface.

Using stateful documents is not proposed as a way to do all client-server computing. Rather, it matches specific requirements — non-real-time applications that have to operate in distributed systems with very lightweight clients (e.g. display tiles), servers with large processing capabilities but without dedicated storage to maintain many persistent application states and a high-performance network (see Figure 6.3). Servers might not want to maintain application (session) state both because of storage limitations and due to the nature of the applications likely to be run on them — they might be expecting a large number of clients to connect for example.

To support stateful documents, a server has to provide facilities for executing applications and saving, marshalling and unmarshalling their states to and from the network. This places an extra load on them.

The rest of this chapter discusses the use of stateful documents to make servers stateless in more detail. An implementation made over the World-Wide Web and some experiments made with it are presented. Related work is discussed first to provide a context.

6.3 Related Work

The design tradeoffs explored in the 1980s for LAN-based distributed systems for the workplace are being revisited for the ubiquitous environments of the 1990s. Diskless workstations were used in the V system at Stanford [**Cheriton83**] and the idea has reappeared in the network computer proposals. The Cambridge Distributed Computing System (CDCS) was the first to investigate the potential of a "processor bank" [**Bacon90**]. The CDCS processor bank comprised heterogeneous hardware, to allow for system evolution, and support for loading heterogeneous software into processor bank machines. Our approach has this functionality but employs a different mechanism.

Simple mobile code can be used to install software on client and server machines as it is needed. This is just a convenience, comparable with installing compiled binaries from traditional transportable media. The following issues remain:

- Application-specific code has to persist on clients and/or servers
- Clients and/or servers have to maintain session state (whether in the form of data or program threads)

Python [**CNRI97**], ML [**Rouaix96**], Java [**Arnold96**] and many other languages can be sent to run on client machines through the World-Wide Web. The TACOMA project has allowed its mobile processes to be uploaded from clients to servers via Web documents, where they are launched and carry out their tasks [**Johansen96**]. Jeeves, the Java Web Server [**Sun97b**], similarly allows a server dynamically to load functionality. Neither considers subsequent relocation to the client of processes at the server.

Some World-Wide Web services (for example indexers like AltaVista [**Seltzer96**]) have been able to eliminate the need to run application-specific code on clients and the need to maintain state on the server by returning results in HTML and embedding session state inside those results. The Web client re-submits the state when the user clicks to see more. This is essentially the method described above in Section 6.2.2 except that by using higher-order mobile code

- arbitrary applications can be saved into documents by marshalling continuations — no special code needs to be written; and
- no prior or persistent knowledge is required on the server of what the user wants to run

Web* [**Almasi95**] allows scripts to be embedded inside World-Wide Web pages and executed by a server. When a page containing embedded scripts is requested from the server, a new page is formed for returning to the client by substituting each script with the output that results from executing it. This allows page elements, for example the current date, to be dynamically generated. Web* is similar to the Common Gateway Interface (CGI) [**W3C97**], which allows scripted generation only of whole Web pages.

Web* also provides support for returning a script's state in Web pages. The state is then given to scripts embedded in pages subsequently retrieved by the user. However, Web* differs from the work described in this chapter in the following ways:

- Web* can only save variable values in Web pages. This means that a script's author must explicitly name global variables to be put into a Web page. The implementation described below in Section 6.4 provides a transparent mechanism to authors for saving state. It can save a program's execution state in a single call.
- Web* is page-oriented whereas the implementation described below is program-oriented. Web* applications have to be split into parts that are separately defined as scripts in Web pages. The implementation described in Section 6.4 allows programs to be written normally and provides a facility for saving their state simply in pages that they generate. User interaction via Web pages is implemented in a way that is integrated with program execution (see Section 6.5). Using Web*, programs have to be adapted to fit the Web's model of interaction; the implementation described below adapts the Web to become part of program flow.
- Web* assumes that pages containing scripts already exist and continue to exist at the server. This chapter describes a technique that places no

such burden on the server — no permanent knowledge of an application is held there.

The Network Computer [**Oracle96**] with its *thin clients* and application servers is a vision of the future particularly relevant to the work described here. However, network computers have to be able to execute arbitrary code downloaded to them. This makes them more expensive than they would otherwise be and is not always necessary.

The rest of this chapter discusses sending applications to the server when they need to execute and putting them dormant into the documents sent to the client when user input is required.

6.4 Application to the World-Wide Web

6.4.1 The Web as a testbed

The Web is suited for stateless servers and stateful documents because:

- A Web server potentially deals with requests from large numbers of diverse, widely distributed clients and maintaining their states in one place for long periods of time is burdensome. Their needs cannot be generalised and their physical connections may be unstable.

- The Hypertext Transfer Protocol (HTTP) [**Fielding97**] is stateless so servers do not by default maintain information about clients.

- Browsers are lightweight clients which simply display documents written in the HTML markup language.

In the system proposed here and detailed in the next section, servers provide general processing to browsers. They are written to treat requests for pages as programs to execute and send back to browsers the output of executing those programs. They forget about a program once it terminates. Browsers send applications to a server and then display their results as they are produced. If applications save their states in the results, they can be restarted at the server when the user provides input in the browser.

This allows for an application's state to be maintained whilst keeping both browsers and servers stateless. A server remembers nothing about the applications it executes when they finish running there and are sent back to the client. Browsers simply display the pages that servers return — they just need to render markup. Application state is contained in the links and forms contained in the pages. When the user provides input, the state is sent to the server and the application starts running again. It can then repeat the process by returning a new Web page containing new state.

Putting application state in Web pages increases the size of data sent between browser and server (see Section 6.6 for an indication of how much). Putting application state in documents is suited to fast networks. When a computation has to reside on the user's machine (e.g. for real-time graphical displays or polling user interfaces), then some application code must execute on the client-side. Putting application state in documents is not proposed as a replacement for client-side execution. However, it does provide an alternative that places computation on the server side, does not require servers to maintain session state or a thread of execution for each client and works with very lightweight clients.

6.4.2 Implementation

An implementation has been made using the World-Wide Web as a context for client-server interaction. It provides lightweight clients in Web browsers (text or graphic) and a simple document markup language in HTML (that can be used for embedding application state).

Web browsers expect to communicate with servers that understand HTTP. An HTTP server has been written with the Tube higher-order mobile code system. It is installed simply by sending it to a Tube site.

Once installed, Web browsers can connect to this server. It treats any requests for pages it receives as (mobile) code to execute. The code is executed and any output produced sent back to the browser. This allows the Web to be used to provide convenient access to the Tube's compute servers.

The programs sent by browsers as page requests act as transitory Web servers; they run on the Tube Web server and disappear after producing their tasks' output as Web pages. They can be complete Web applications that implement an active and stateful user interface to generic services provided on the server side. They are novel because they are able to embed (part of) themselves into the documents they return to their browsers. They can do this because they are written as higher-order mobile code and can marshal arbitrary closures and continuations as plain text. The Tube mobile code system provides this facility.

The states are made part of links within HTML pages returned to browsers. Link addresses are set to point to the Tube HTTP server. After an application finishes sending its HTML document back to the browser, it terminates. The states it embedded in the document represent potential to execute again.

When the user clicks on one of the links in the browser, the embedded state becomes a request to the Tube HTTP server and the application starts running there again. It can then repeat the process by returning a new Web page with new states embedded in it. To summarise, an application runs on the HTTP server until interaction with the user through the client browser

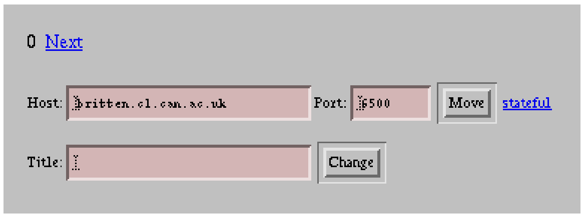

Figure 6.4: Counter in a stateful document

is required. At this point, it is removed from the server and its dormant state passed back with the interface presented to the user. After the user enters his input, it is sent back to the server with the application, which is restarted.

In the current implementation, a whole new HTML document is returned to the browser so that the user sees the whole page being updated. One might be able to achieve finer grain control over update by using Netscape's non-standard frame or layer enhancements to HTML.

A simple example is shown in Figure 6.4. It is a monotonically increasing counter. When started, it displays its initial value, zero. Every time the user clicks on the *Next* link, a new page is returned displaying the next value. This is because the next stage of the computation (that increments the counter and returns a new page) is stored as a marshalled continuation in the link. The size of the counter's marshalled state is around 10 kilobytes. However, the marshalling format is particularly amenable to compression — when compressed, the counter's state is just over 2 kilobytes in size. Section 6.6 contains further discussion of application state and the time it takes to compress and marshal it.

6.5 User Interface and Program Structure

The implementation benefits from the clean separation between user interface and application functionality that generating document markup promotes. That is, an application can proceed with its computation until it requires user input. It then generates from its data structures a fresh document to present to the user and waits for input. After the user provides input, the application can resume exactly in the same state as it was before waiting. The input can be processed and computation proceed accordingly.

This method of handling user input means that computation is driven by the application rather than by the user interface. Traditional callback-based methods use the user interface to drive applications. The problem with using callbacks is that a side-effect must be used to note that a particular input has occurred. Subsequent callbacks can then tell what has happened

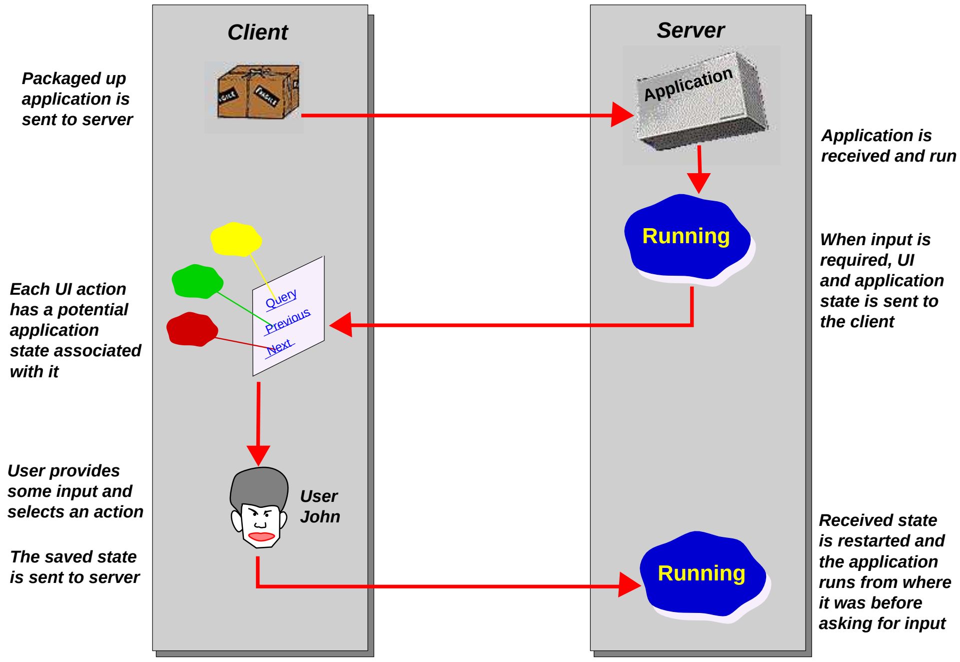

Figure 6.5: Separation of user interface and application functionality

before. The programmer must maintain some global state that is side-effected with the application's state each time something happens. Callback-driven applications can be converted into application-driven ones by using a continuation-passing style [**Fuchs96**].

For client-server applications, a stateful server that maintains application threads can support application-driven user interaction by blocking after sending a user interface to the client until receiving input back from it. Stateless servers that are enabled by stateful documents achieve this too because while the user is providing input, the application is dormant and stored as part of the interface for later resumption (see Figure 6.5). Program flow is controlled not only by matching on the input when the application resumes on the server but also by placing different continuations in different links; the application is implicitly told which one the user selected by the state it is resumed in. The actual user interface is separated from application functionality but the events it provides are handled more naturally as part of program flow.

6.6 An Information Retrieval Query Interface

An existing application written in Java [**Mills97a**] has been rewritten to use HTML and stateful documents. The Java version uses client-side presence to remember session state. The stateful document version remembers all application state in the HTML document. It does not require Java support from the Web browser — the text-based Lynx browser can be used, for example. Neither the browser nor the Tube HTTP server that it connects to maintains persistent knowledge of the application.

The interface is shown in Figure 6.6 displayed by the Netscape Web browser and in Figure 6.7 by the text-based Lynx browser. It allows the user to search a collection of historical material covering the lives and events of the English village Earls Colne between 1400 and 1750. The COBRA information retrieval system [**Mills97b**] is used to carry out indexing and searching. The interface has to remember more state than a simple one such as AltaVista [**Seltzer96**] because it supports relevance feedback. Relevance feedback allows the user to mark results that are relevant and present them as hints to the retrieval system for use in query refinement. A review of work on relevance feedback can be found in [**Frakes92**]. A user interface supporting relevance feedback must remember the complete context of a query, i.e. the user's search terms, search terms suggested by the system and any results marked as relevant to the query.

This state and the functionality of the interface must be held somewhere. The Java version holds them in the client. They could be held in the server if scripts were pre-installed, implementing the application's functionality, and space made available for persistent storage of state. The stateful document version remembers both the application's functionality and query context in the HTML documents displayed in the browser. At any one time, the state of the application is contained either in its execution at the server (in between user interaction), in a document sent to the browser for display or in a request sent from the browser to the server.

The average size over ten queries of the Earls Colne application's state when saved in a Web page is 42 kilobytes, which includes ten sets of results. This compresses to just over 7 kilobytes in size. On a lightly-loaded DEC Alpha running Digital Unix at 166MHz, it takes about 0.3 seconds to compress the state and 0.2 seconds to decompress it. Whether it is worth spending the extra time compressing and decompressing the state depends on network conditions. For slow connections, it might take more than 0.5 seconds to transfer the extra 35 kilobytes both ways.

The most expensive operations in the current implementation are marshalling to and unmarshalling from plain-text representations of program state. The 42 kilobyte state takes 0.6 seconds to marshal into a document and 0.7 seconds to unmarshal from one on the same DEC Alpha. Commercial implementations would have to optimise these operations. One possibility would be to unmarshal program state on demand; that is, a part of the state would only be unmarshalled when required for execution. This would be a simple matter of using delayed evaluation in the Tube's Scheme implementation.

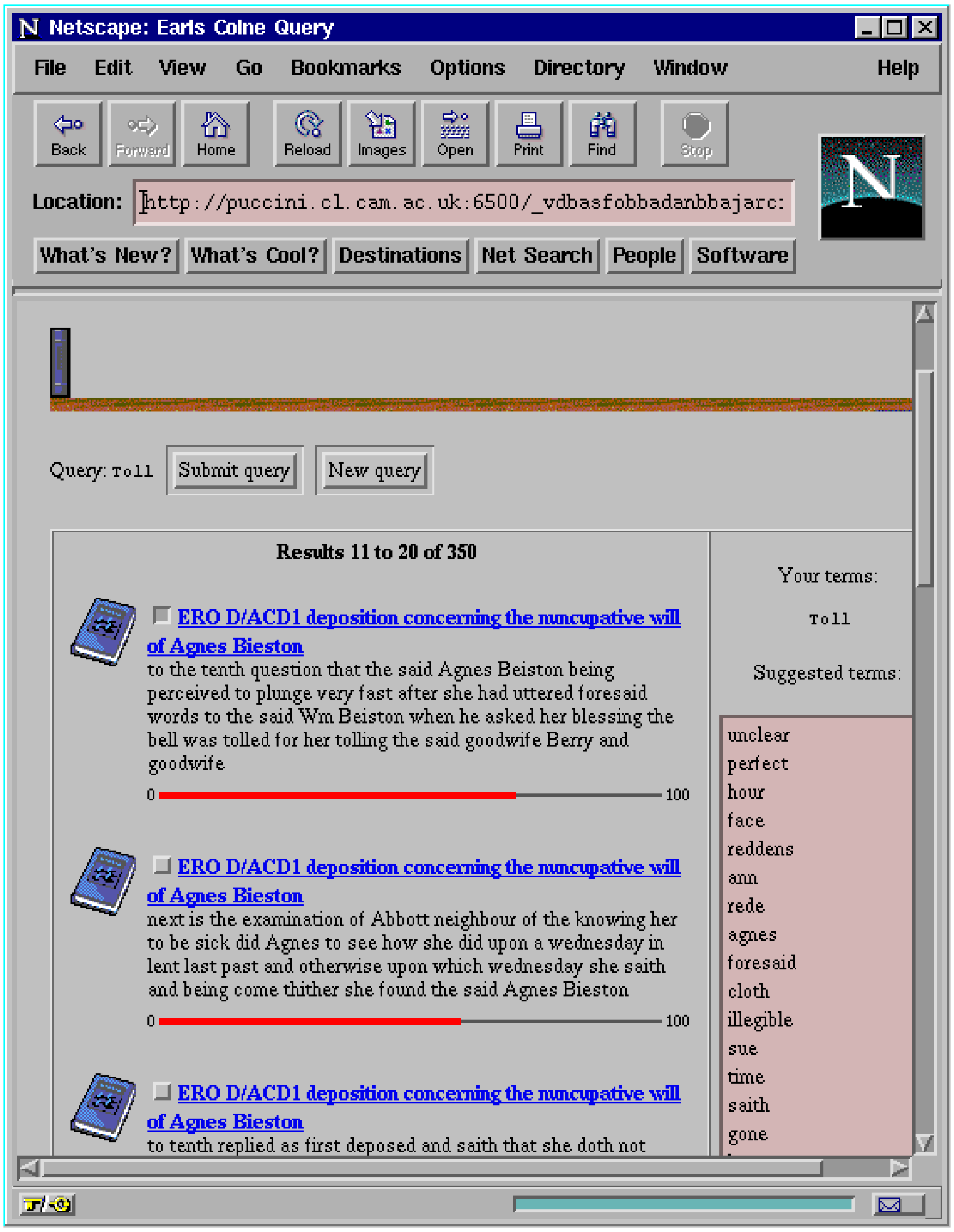

Figure 6.6: The stateful information retrieval interface displayed by Netscape

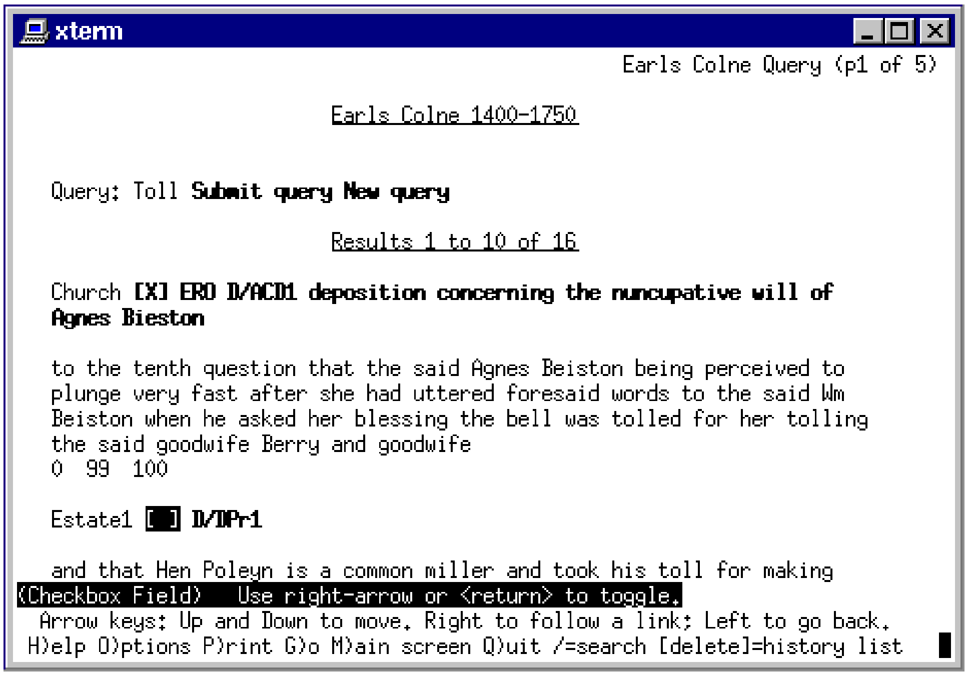

Figure 6.7: The stateful information retrieval interface displayed by Lynx

More detail on how the Earls Colne interface is used can be found in the description of the Java version that requires support for client-side execution [**Mills97a**]. The user can enter queries, see matching documents and terms suggested relevant by the retrieval engine and mark some results as relevant.

6.7 Logging

The ability to put all of an application's state inside a document is very useful for the Earls Colne query interface. By logging the documents returned to the browser, a history of the user's activity can be formed. One can then jump backwards in the log to a particular query and its context. For instance, a user might find that he has degraded results over time through questionable relevance feedback choices and wants to backtrack to a previous context. All he has to do is reload a previous document from the log. A simple annotation facility is provided in case he wants to remember particular instances of the interface by name.

It should be emphasised that no special support is required here for logging. The application does not have to undo changes itself in order to backtrack to a previous state since each log entry is a complete and automatically-generated instance. When an entry is restored, it is the complete application that is restored, with the same context as when it was logged. It is not just the application's appearance that is restored, because potential new execution states are embedded as continuations within the user interface's hypertext links. Using stateful documents makes logging easy; using code resident in clients or on server machines requires bespoke solutions.

Logging can be carried out in the client (by the Web browser) or in the server (by the Tube HTTP server). If done on the server side, the states can be sent anywhere for storage — the client may tell the server to send them back to its local domain or may have contracted storage facilities from a third party. In the current implementation, the server saves states to disk and then makes them available for users to navigate via a management interface.

Web browsers can be made to log applications either by saving documents to disk or by bookmarking their addresses. The latter works because application state is contained in links within documents. These links also contain the address of the server on which the state should be restored. When one clicks on the link, the browser "goes to" that location, delivering the state to the server. The application is then resumed and produces a document. The browser's current location thus contains the application's continuation, i.e. how the document was produced, and can be bookmarked.

This logging facility might be useful in analysing how a Web application is used since one can recreate the complete history of its execution. There is an option in the Tube HTTP server's management interface that prevents logged states from being logged themselves when they are replayed so that one can view a user's activities without generating copious amounts of new information. Storage and analysis of logging information is outside the scope of this research.

Logging can help to cope with failure. Servers can send application states they receive to a persistent storage service before unmarshalling and executing them. If a server crashes then it can retrieve from storage the states of the applications it was running before the crash. A client (browser) can send application states to a persistent storage service as well as to a server, so that it can resubmit them if it crashes.

6.8 Moving State

Keeping all state in documents allows the user easily to move applications between servers, simply by changing the address of a link. Since servers keep no application-specific code, any one can be used, wherever it is located. This is useful if the user has to rent time on servers and a cheaper alternative is found, or if he moves and wants to use a server local to his new location. In this case, he can save the document to a file, change the link's address

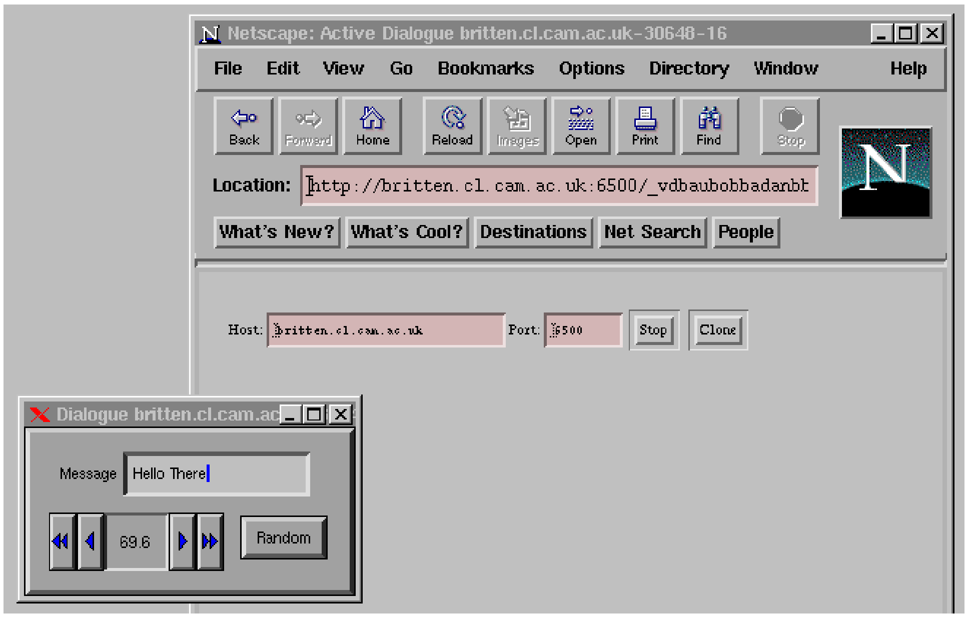

Figure 6.8: Embedding external user interfaces in a stateful document

to the new server's location and simply load it into a browser at his new location. Alternatively, a location-aware program can do this for him (see Section 7.5.4).

Another use is for publishing programs. One could list stateful document Web applications in a Web page. Then either:

- The user saves that page, editing links to fill in his local server's address. This might be automated with support from the browser and involve a trader lookup. Or:

- The author might allow his server to be used for priming applications for use on a user's local server. The user would type in his server's address and a document containing links ready for launching the application in his domain would be returned.

Alternatively, the link containing the application's state might be e-mailed to the user, who would just fill in his local server's address and point his Web browser to it.

Since browsers download the initial states of applications from a central site, updating applications with new versions is simple. The new version of an application is installed at the download site and browsers use it for sending to a server for execution. This results in the decreased administration costs often cited for Network Computers [**Oracle96**] — users cannot corrupt the installation of an application.

Finally, stateful documents have been extended to enable applications employing traditional user interface toolkits, such as Motif and XForms, to be launched and suspended from a Web browser. An example using the Tube's user interface state-saving facility is shown in Figure 6.8. It is launched by clicking on a link in which its code is embedded. The dialogue box is then displayed, which the user can manipulate by typing in a message or setting the counter.

At the same time, a page is returned to the Web browser which can be used to stop the dialogue box and place its state into another page returned to the browser. This page can then be used to restart the box in exactly the same state, with all of its components working. One can carry on saving and restoring its state indefinitely.

Just like stateless server Web applications, the dialogue box in this example can be moved between servers and saved to disk once it is captured inside a Web page. Its state as it is suspended and restarted can also be logged. One can also clone its state instead — as many copies of the dialogue box can then be made as required from the same page.

6.9 Summary

Using higher-order mobile code, application state can easily be embedded in documents. This chapter has examined how making documents stateful can remove the need to keep persistent knowledge of a client-server application from both client and server. It was argued that this is particularly suited to distributed systems with lightweight clients, heavyweight servers and high-speed networks. An implementation was discussed that uses a higher-order mobile code system for executing applications at the server and a networked hypertext system for moving their states into user interfaces at the client when input is required. The use of this to re-implement an existing information retrieval interface was described. Finally, this chapter looked at using stateful documents for logging, restoring and moving applications.

Chapter 7

Location-Oriented Multimedia

This chapter presents a framework to support mobile multimedia applications and describes the role mobile code can play. The framework can be mapped onto different implementations. One which uses mobile code to support user mobility is described. Particular attention is paid to high-level control, construction of multimedia objects, communication between components and user mobility. A simple and flexible trading mechanism is used to assist the work.

7.1 Introduction

7.1.1 Motivation

As computing equipment is more ubiquitously deployed, it becomes desirable to support users who move between machines. Rather than carry the same portable computer around with him, a mobile professional who requires regular access to facilities can instead find a computer to use at each location. He can use his applications when he moves to a new machine, and they resume execution in the same state they were in before he moved.

Ubiquitous computing resources can be used like telecommunications resources are now. While some people have mobile telephones, most make use of telephones installed ubiquitously at the places they visit. Mobile telephony services are charged at a premium rate, so it is cost-effective for their owners to use land-line telephones instead when they are available.

Not all users will want to own and carry a portable computer in order to be able to use applications wherever they go. Even owners of mobile computers will only want to take them on a journey if they want to work while travelling. If computing facilities are available at his destination, a travelling professional can use them when he gets there. His applications

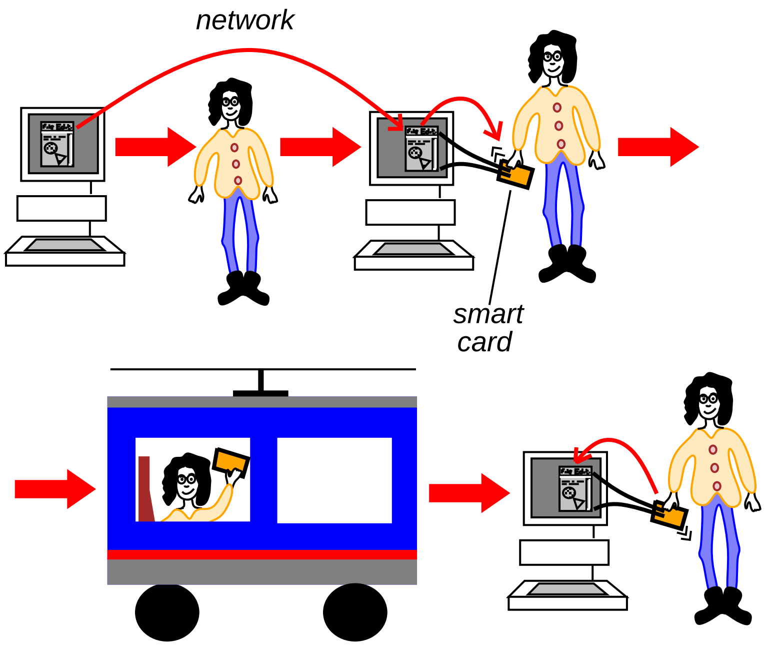

Figure 7.1: An application following a user wherever she goes, via a network and on a transportable smart card

can be moved so that he still has access to them.

This chapter describes a framework in which application components follow users. Such user-targeted mobility allows personalised application state to accompany a mobile user, either over the network or carried on transportable media such as smart cards (see Figure 7.1). Third-party storage servers can also be used to store application states when they are not in use. Services which provide identification and location information about users are also involved.

The framework can be applied to local area networks, particularly in the context of multi-user applications, or for wide area networks consisting of large numbers of public or private access terminals with access to local computation servers.

7.1.2 Using mobile code

Each location to which a user moves has its own set of services. When an application moves to follow a user, it rebinds to the services at his new location. Applications provide high-level control functionality, coordinating the use of lower-level services, and can move between computers.

In order to relocate an application to a user's new location, its state must be transferred there. Using a mobile code system, such as the one described in Chapters 2 and 3, has the following benefits:

- Applications can be transparently migrated between computers. No special state-saving code has to be written into each one.
- Applications do not have to be installed at each site to which a user might move. Mobile code can be sent for execution at a new location without having to install anything there beforehand.
- Network usage can be reduced by co-locating code with the data it processes rather than sending the data in its entirety to applications for processing elsewhere.

7.1.3 Mobile multimedia

Mobile users can benefit from multimedia applications which are able to move around a network, remapping user interfaces and stream-based endpoints like cameras, microphones and speakers to the user's current location. Some examples of such applications are:

- *Sending running programs to other users.* David can prepare and send a multimedia presentation about his latest project to John's current location, with an enclosed questionnaire. John can examine it, annotate it, fill in the questionnaire and send it back.
- *Location-triggered media presentations.* An application can be configured to monitor for when Jean and Ken are in the same room, go to that location and play a video clip from John.
- *Mobile multimedia.* Distributed media endpoints, such as software abstractions of cameras and microphones, as well as user interface components such as video display windows, can be part of a mobile application. A simple example is tracking a user with video. When he moves, the image switches to the camera nearest to him. Another example is music following people around the home through the nearest set of speakers.
- *Mobile cooperation.* A natural progression of mobile multimedia is to support mobility for a cooperating set of users. In a multimedia conference, if a user moves then the camera, microphone, speaker and video window endpoints must be moved to his new location. Connections to the other users involved must be re-established.

This chapter describes a framework to support user mobility that can map onto different implementations. The discussion is oriented towards multimedia applications consisting of audio, video and user interface driven software and concentrates on the role of mobile code to realise a particular implementation. Some examples of implemented applications are given.

7.2 Related Work

This section discusses three projects which have targeted mobility support for application users. ORL's *Teleporting* project redirects the user interface of a mobile user to his current location. The *Total Mobility* environment provides support for users who move between machines and work over low bandwidth or disconnected network links. The work on *Migratory Applications* using Obliq employs a mobile code system which supports a mobile user interface.

7.2.1 Teleporting

Teleporting [**Richardson94**] is a technique for supporting user mobility in which the user interface of an application follows a mobile user. In order to *teleport*, a user clicks a button on his active badge (a type of electronic tag — see Section 7.3.1) when he is near to a workstation. His current session then pops up on the workstation's display.

The teleporting system is based on the technology of the X Windows system. A proxy X server is able to forward X protocol requests to another server, thus providing a level of indirection. The display of X applications can be moved to follow a mobile user by making sure the indirection always points to the display closest to him.

Teleporting is a powerful technique, the main advantage being that it makes any existing X application mobile. However, it also has some disadvantages:

- It is tied to the X Windows system. Only the display connections follow a user; there is no dynamic reconfiguration at the application level. Any medium that does not go through X, for example audio, cannot be supported by the teleporting system.

- It is potentially inefficient since data has to pass through an extra party.

- Applications do not themselves move, are unaware of their user interfaces moving and are not built around the premise of mobile users. This means that factors explicitly related to mobility cannot be capitalised on. In particular, the distribution of an application's components cannot be changed according to its location.

The user mobility framework described in this chapter allows applications and their user interfaces actually to migrate between computers, thus avoiding these disadvantages.

7.2.2 Total Mobility environment

The Total Mobility environment (TM) [**Wachowicz96**] views computers as "replaceable tools, which can be hired, used and given up once no longer needed". Like the work described in this chapter, it aims to support the use of locally available computing resources whenever a user moves, to eliminate the need to carry equipment when it is not useful. It addresses the problem of providing users with access to their data even when using mobile devices that are not connected to a network.

TM provides location information about users and computer equipment and a data management system based on a disconnected file system architecture. Users register and de-register for use of a mobile computer in disconnected operation. Their files are replicated to the mobile devices when they register. A logging process is used to identify modifications that users make while they are on the move and disconnected. When the user deregisters with the mobile device, or network connectivity is available again, the main copies of the files are updated with the modifications. The logging process also helps to cope with conflicting modifications made by multiple users sharing the same set of files.

The user mobility framework described in Section 7.3 differs from TM in that it concentrates on support for making (multimedia) applications mobile over broadly uniform networks rather than providing generic access to user files over low-bandwidth links. TM does not address migration of applications to follow users. However, combining the two would provide migratory, follow-me applications with support for working in low-bandwidth and disconnected environments.

7.2.3 Migratory Applications

The Migratory Applications work [**Bharat95**] has linked user and application mobility. It employs the Obliq mobile code system [**Cardelli94**] so that applications can move from one user to another or can be configured to follow users from machine to machine. Support for state saving in Visual Obliq allows applications to take their user interfaces with them when they move.

The Migratory Applications work is similar to that described in this chapter in that it employs mobile code and knowledge of user location. However, it produces monolithic applications for single users in which the mobile code component contains all control functionality and the user interface. The rest of this chapter describes a framework for user mobility which subsumes this functionality but also has the advantage of being able to support mobility for distributed multimedia applications involving more than one user.

7.3 Framework

A flexible framework for supporting mobility of users is described here (see Figure 7.2). It comprises a number of components:

- event-driven location awareness
- migratory programs
- stream-based multimedia objects
- simple trading system
- application-level communication
- mobile user interfaces

Implementations of this framework allow multi-user multimedia applications to be written which follow a user as he moves around a network of computers. This section describes the framework in more detail. Section 7.4 gives an implementation which includes use of mobile code. Section 7.5 describes some experiments made with it.

7.3.1 Event-driven location awareness

Applications must be notified when and where a user moves in order to follow him. This is *event-driven mobility.* A location service collects location information and notifies its clients of any events which match their interests. Client applications register interest in pertinent location events.

The location service uses a method of event-based programming which allows applications to be written in terms of actions that are triggered by external activity. The model is applied to distributed systems by allowing event occurrences in one place to be sent across the network to trigger actions in applications running elsewhere. Further details about the implementation of an event-based programming system can be found in [**Bacon95**]. It is very briefly described here but this dissertation does not contribute towards its development.

The event system is implemented as an extension to an ODP-compliant distributed programming platform [**ANSA93**]. The interface definition language has been extended to allow specification of event types. An event language has been devised which allows applications to register interest in complex event occurrences.

The location service uses this event system to make mobility information available to clients. When user movement is detected, it is matched against event expressions registered by applications. Invocations are made to trigger actions in any that match. By remembering movement events, the service can provide information about where a user was last seen.

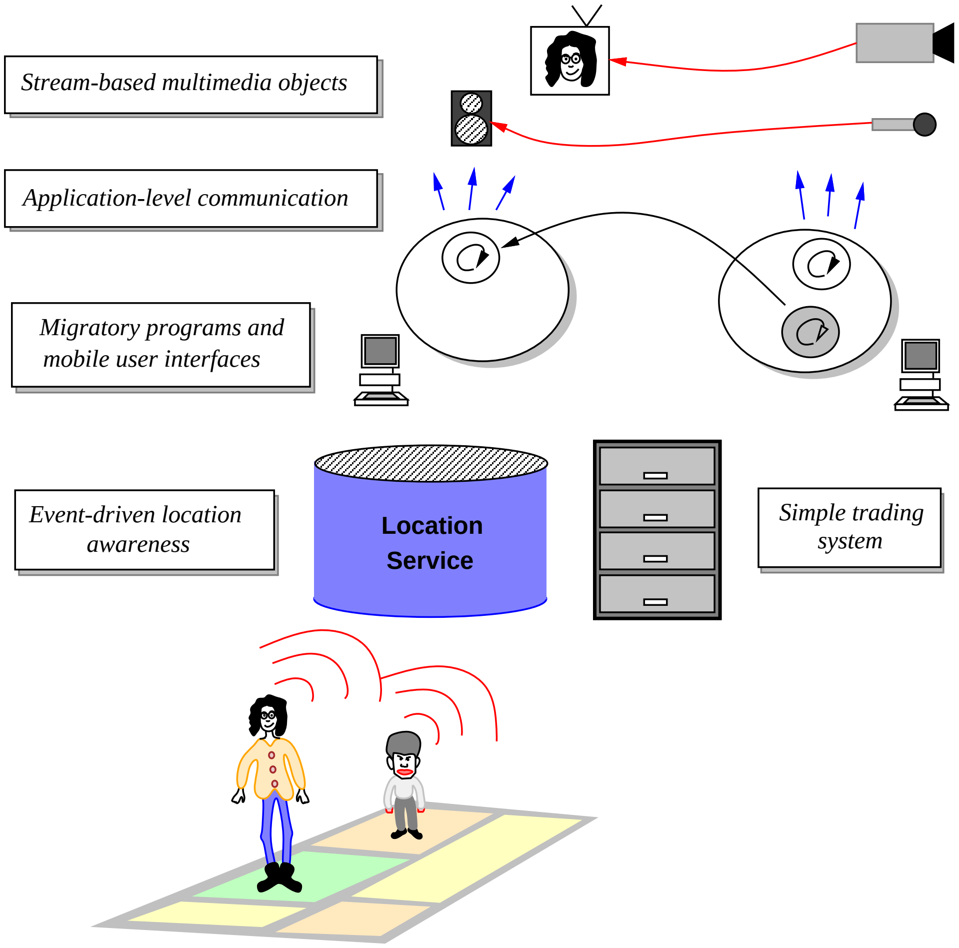

Figure 7.2: Framework for supporting user mobility

There are a number of technologies which the location service can use to obtain location information, for example monitoring of user logins or electronic tags that users wear. The implementation used to support the work described in this chapter employs active badge location technology [**Harter94**]. Each user wears a badge which periodically transmits a unique infra-red signal. A network of detectors allows the location of the user to be pinpointed. Event expressions are evaluated when a user's location changes.

User mobility events are augmented with information about room names, building geography, equipment stored in each room and the names and capabilities of each piece of equipment. This is useful for obtaining proximity information, both between users and between users and equipment. It can also help to build adaptive location-aware multimedia applications [**Schilit94**]. For example, if an application is following a user or is dispatched to a new location, it can check that the required facilities are available there. If it involves audio and video but these are not supported in a particular location, text-only conferencing might be used instead.

7.3.2 Migratory programs

When a user moves, his applications must also move to follow him. This can be achieved in a number of ways. Firstly, the application's user interface can be moved to display on the machine nearest to him, while keeping the main body of the application in the same place (see Figure 7.3(i)). This is adequate for applications which do not need to have their computation relocated or that run over fast networks. However, with high-latency networks, interaction will be cumbersome. Some applications will need to relocate, such as those needing access to local multimedia devices or those that adapt to use local facilities when a user moves.

Secondly, application code can be held at each site to which the user can move (see Figure 7.3(ii)). Factories are distributed services that create objects of defined classes on request. An application (or part of one) can be moved by asking the factory at the destination site which knows about its implementation to create a new instance. The instance at the source site is then terminated. Protocols understood at both the source and destination are used to control the process of object creation. This approach to application mobility requires that all sites be primed with an application's implementation before it can move. This imposes an administration overhead whenever a new application is written.

Thirdly, mobile code can be used to implement application mobility (see Figure 7.3(iii)). At each site, generic execution facilities are provided that can receive new application code from the network and execute it. Using mobile code allows application components which wish to relocate to do so without having to install them beforehand. Using a higher-order mobile code system facilitates applications with transparent migration running on

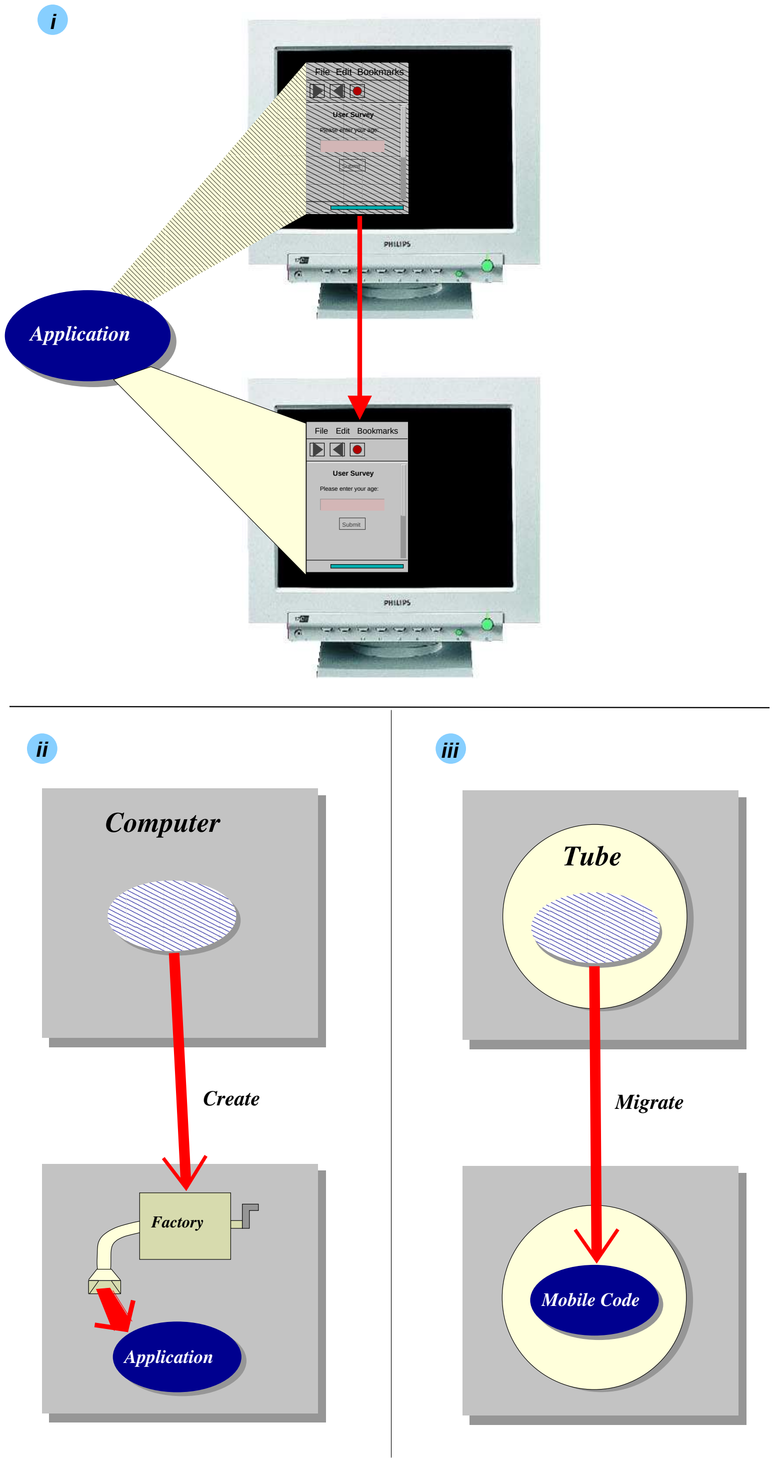

Figure 7.3: Moving an application to follow a user
(i) Re-targeting the user interface
(ii) Holding application code at each site
(iii) Using mobile code

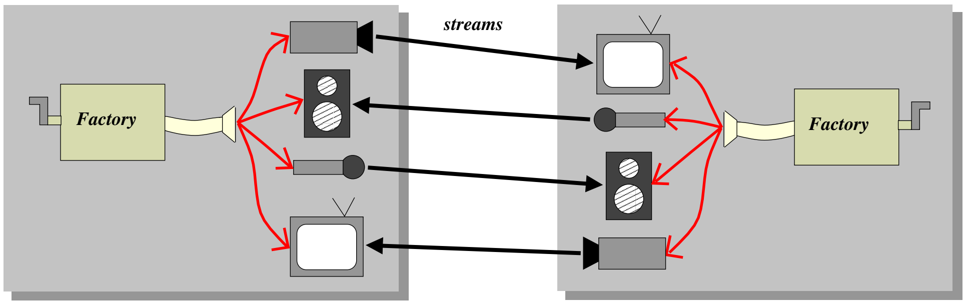

Figure 7.4: Stream-based multimedia objects

networks of heterogeneous computers.

Mobile code is suited to high-level control of distributed applications because it usually sacrifices efficiency for portability. Writing entire applications with a mobile code system is not proposed for this framework. Rather, it is used to provide flexible scripted control of services that already exist. New applications are written as migratory programs that can adapt and bind to services as they move. As applications are written at a higher level, so the ability to tie together extant components becomes more attractive. Components are written according to their particular requirements. For instance, those needing to process multimedia data will be written in an efficient low-level language; those needing access to third-party software might be linked with compiled libraries. The top-level control loop of an application, perhaps tied to a user interface, can be written at a high enough level of abstraction to use an interpreted, higher-order mobile code system. Components implementing specific functionality operate at a lower level and use bespoke techniques for yielding their states.

7.3.3 Stream-based multimedia objects

The framework includes simple low-level stream-based audio and video software objects (see Figure 7.4). They control real microphones, speakers, cameras and video windows using whichever means are appropriate for the hardware. Control is abstracted to a distributed object layer so that applications can easily create, connect and destroy multimedia sources and sinks. Object factories residing on machines with appropriate devices are used to instantiate objects. For instance, on every workstation with live video capture, there will be a factory capable of creating software camera objects. Objects to be connected together must use compatible raw data formats.

Multimedia objects are described in terms of streams. Sources (microphones and cameras) have output streams and sinks (speakers and video windows) have input streams. Notionally, applications connect a source object to a sink object by plugging the source's output stream into the sink's input stream. Sources should be able to multicast their data using streams; whether the stream implementation takes advantage of network support for multicasting is not specified here.

Mobile users of multimedia are supported by allowing them to have personal audio and video objects which follow them around. As a user moves, his microphone and camera objects move too, so that anyone listening or watching him can continue to do so. Any speaker and video window objects he owns will also move, so that he can continue to listen to and watch others.

Just as multimedia objects can map onto different hardware, so they can be arranged in different ways to build multi-user mobile multimedia applications. The hardware may dictate where knowledge of each connection is held. If an input or output device is shareable by more than one process, then a different object can be created for each stream connection. However, if only one process per machine is able to access the device, then one object only must be created for it and all connections made from or to it. Knowledge about who owns which connection must be held elsewhere, perhaps in virtual objects that forward requests onto the real one, in a per-user connection manager or in a centralised connection manager for all users.

Besides hardware limitations, it is not desirable to specify an exact arrangement for applications. For various reasons, including efficiency, available resources and use of third-party software, applications will have different requirements. Some may wish to create a number of loosely-coupled, autonomous components; others might prefer a centralised solution. In the context of mobile users of multimedia, where connections are managed can vary.

If separate objects are used to manage connections between users then the objects reconfigure their connections as they move (see Figure 7.5(i)). Creating separate objects for each user places a heavier burden on the host computer because there are more processes running at the same time. Sharing one object between users on each machine lessens the burden but forces connection management to be done at the application level (to remember which connections are owned by which users). This might be done by creating separate application objects which themselves move and rebind to real multimedia objects at their destinations. This approach increases again the burden on the host computer but does mean that the multimedia objects can be kept simple.

Alternatively, applications might have some central entity manage connections (see Figure 7.5(ii)). The nature of a central connection manager can vary. There might be one per user, so that it knows which objects to talk to and which connections to manipulate when its user moves. It would move to follow a user as he moves between domains. For applications with a single instance, which perhaps cater for a set number of users, such as video conferencing, connection management might be centralised even fur-

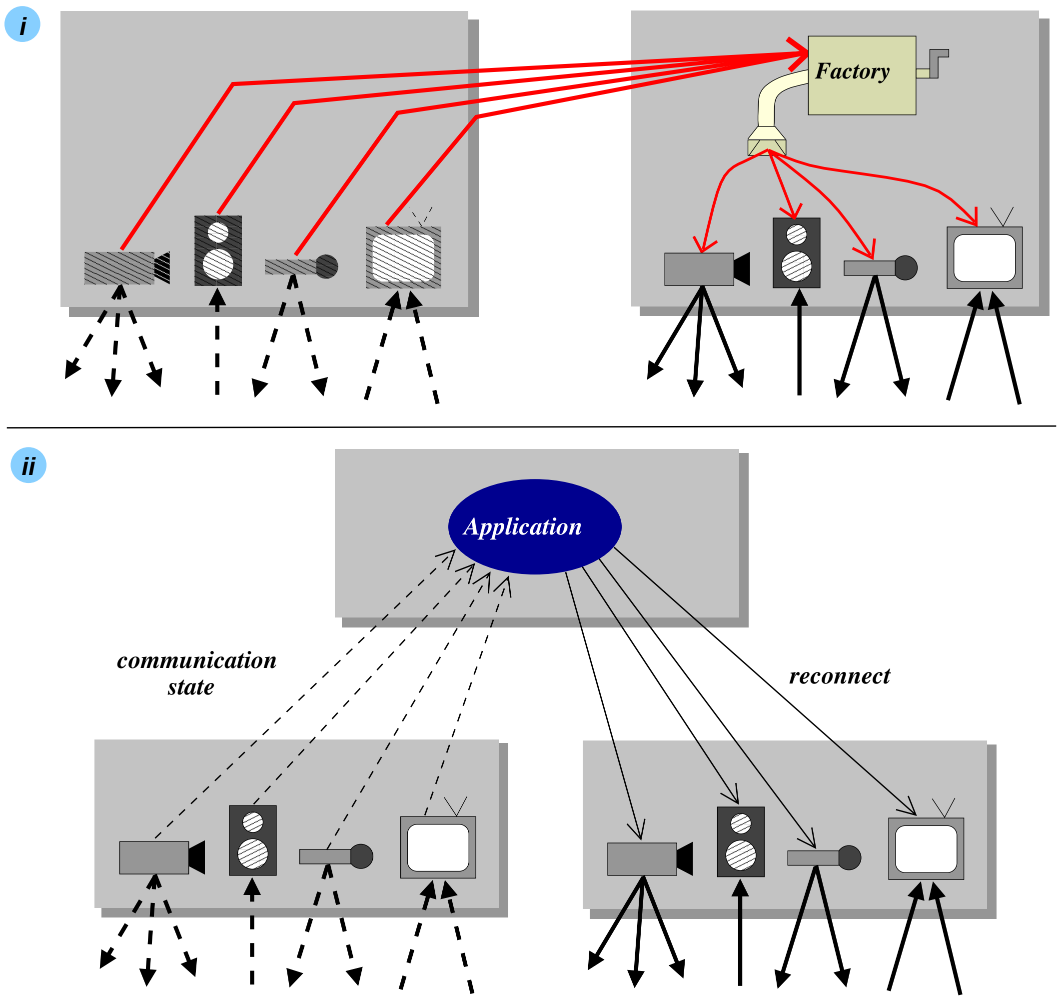

Figure 7.5: Managing connections
(i) Using separate objects
(ii) Centrally

ther. Knowledge of every connection would be held in one place; when one of its users moves, the application would reconfigure his connections. A video conferencing manager, for instance, might move to follow the conference's chairman.

The extent to which a mobile application's activity (connection management in the case of video conferencing) is centralised can be affected by consideration of:

efficiency the processing and network resources available to an application should affect how much control is devolved into distributed components

authoring the extent to which an application is naturally distributed and how much state components are likely to share should be taken into account

mobility management coordination of an application when it moves becomes more difficult because it is decentralised

Mobile applications are free to coordinate their own components' movements using appropriate methods. Distributed components might return their states to one controlling entity or move independently. Centralised applications coordinate activity in one place anyway.

However, the framework must provide general support for mobility. Stream-based multimedia objects must cope with re-routeing of their data (see Figure 7.6). Applications which either move distributed components separately or communicate with other applications must ensure that they provide notification of movement and react to notifications received. Components' communication state must at some level be transferred.

Re-routeing of streams is known as *handoff*. This is commonly provided by the network layer (see for example [**Rajagopalan95**] or [**Porter94**]), though application support for handoff has been proposed [**Pope96**].

How handoff should be achieved is not specified here. Applications or components that use stream-based communication should use the same handoff methods to ensure data is re-routed correctly. Some applications may not even need handoff. For instance, multimedia objects which can tolerate data loss might simply use a specialised user-space implementation that closes down their connections before moving and re-establishes them afterwards. Centralised applications which reconfigure connections, but in which little moves, can coordinate control over what happens because there is little autonomy; they can direct activity to ensure no data is lost.

In general, applications will wish to support handoff to varying degrees. Forcing it to be transparent to applications or always carried out by applications is unnecessary and undesirable [**Agrawal96**]. Applications might wish to provide assistance to the network or perform some housekeeping when a

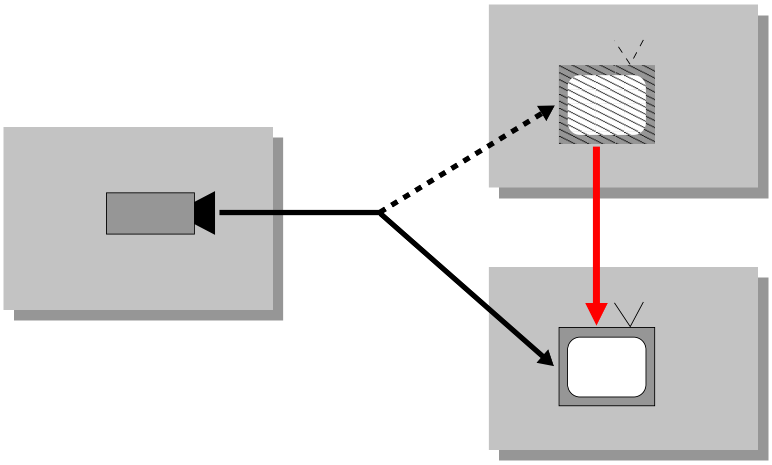

Figure 7.6: Re-routeing of streams (handoff)

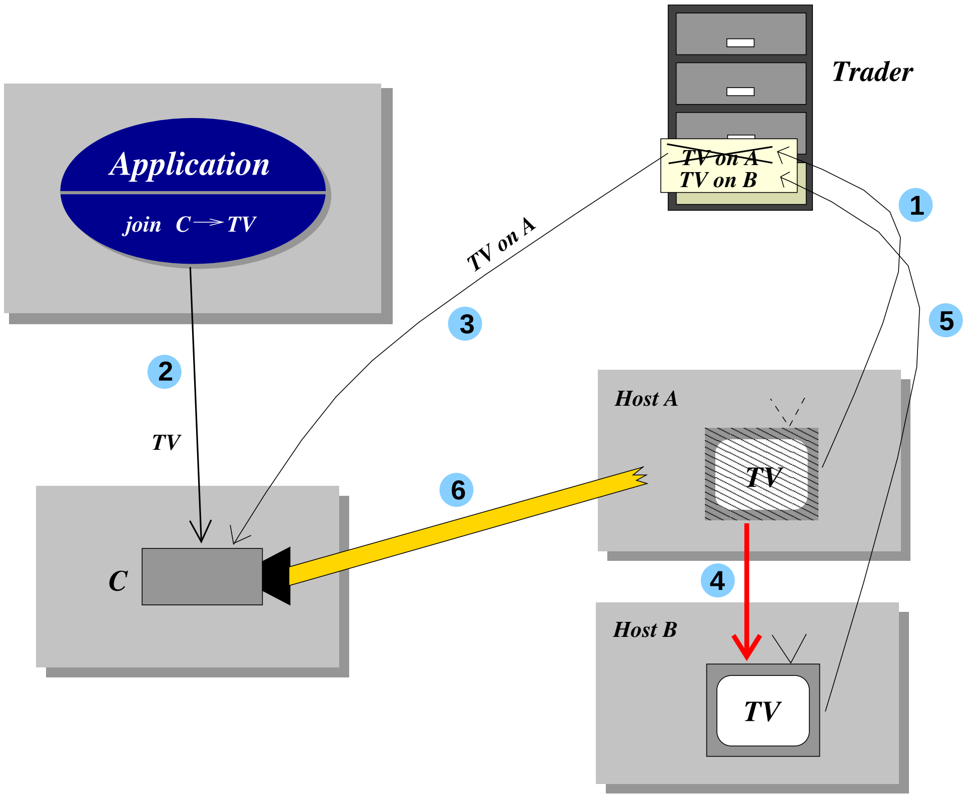

Figure 7.7: Trading mobile applications

stream is re-routed. Future work might capitalise on research being carried out into allowing application-specific policy for mobility to be closely bound at runtime with network control functionality [**Rooney97c**].

7.3.4 Simple trading system

The trading mechanism provided for applications to locate each other must be able to handle mobility in conjunction with the methods used for communication between components. Figure 7.7 shows how an application can obtain and use out-of-date information:

1. Television object TV registers its initial location on *Host A* with a trader

2. Application initiates a connection between camera C and television TV

3. C imports TV's location from the trader as *Host A*

4. TV decides to migrate, in response to user movement

5. TV registers its new location on *Host B* with the trader

6. C tries to connect to TV at its old location on *Host A* and fails because TV has moved to *Host B*

When moving, an application should ensure that any trader entries containing its location are updated to point to its destination; further communication with it should be directed there. Communication with an application while it is moving should not fail. Rather, it should either be blocked until the move is complete or inform the application of the situation.

How this should be achieved is not specified here. However, one possibility is to post an indication that movement is in progress to a trader (possibly replicated or federated [**ISO94**]). Applications finding such an indication would either ask to be told when the move is completed or simply poll the trader until the new location is registered. A home trader might be associated with each application, which always holds its current location. This would help to distribute trading activity if applications had different home traders.

Further, a mobile application may be able to register with only the trader in its current domain. Local applications in the same domain do not then have to go outside it to look up its address. Traders in other domains are told which domain the application is in, through a system of federation. Applications in other domains are then able to perform a lookup that involves only their local domain and the moving application's current domain.

Alternatively, mobile objects might leave *tombstones* behind when moving that hold forwarding information or leave behind a minimal process which handles requests while movement is in progress. Both introduce a problem of garbage collection.

[**Pope96**] provides a discussion of trading (and handoff) in mobile computing which this framework subsumes for user mobility amongst ubiquitous computing facilities. Not all applications will need special support for mobility from the trading mechanism. Examples are those which are centralised or do not involve movement of components that are in contact with other applications.

A simple syntax is proposed for traded advertisements. They consist of arbitrary numbers of field→value mappings presented in an SGML-like syntax. Fields are denoted by start and end tags; values are enclosed by tags. This is the system used in the building of distributed object systems with mobile code described in Section 4.4.1.

Using such a syntax for advertisements provides great flexibility because arbitrary fields can be traded. Lookup of advertisements is specified by presenting a query in a similar way to searching information retrieval engines. One specifies a logical combination of text occurrence within fields.

A global naming scheme has to be agreed for use by implementors. The technology used to implement trading is not specified here. One might map the syntax onto an existing trader implementation, use an information retrieval engine or match queries with regular expressions, for example.

7.3.5 Application-level communication

Applications can use any method to communicate with each other. In cases where mobile code is used to bind to services in a new domain, it must obviously know how to talk to them. If the means to do so are not available on the execution platform there, then it must carry appropriate code. One possibility is to use mobile code itself to build a distributed object system (see Section 4.4) and hide the exact communication method behind a common object model. Examples of other distributed object systems are DCOM [**Brown96**], CORBA [**Siegel96**] and ANSA ODP [**ANSA93**].

7.3.6 Mobile user interfaces

An application should be able to take its user interface with it when it moves. This might be achieved simply by redirecting rendering of the interface (while the application itself remains stationary) [**Richardson94**] or by saving the actual user interface components and associated callbacks as part of the application's state when it moves (see Chapters 2 and 3). Use of advanced implementation techniques which promote a clean separation of user interface layout and application functionality should also be supported (see Chapter 6). These lend themselves to mobility.

The rest of this chapter looks at how the mobile code system described in Chapters 2 and 3 and the work discussed in Chapters 4 and 6 have been used to implement some of the elements in the user mobility framework described above. Two different implementations of mobile-aware video conferencing are given, together with that of a mobile World Wide Web-based application.

7.4 Realisation

The previous section described a flexible framework to support mobile users of multimedia applications. This section gives an implementation using a mixture of mobile code and more traditional methods.

7.4.1 Event-driven location awareness

The Tube mobile code system described in Chapters 2 and 3 has been interfaced with the event-based programming system described in [**Bacon95**]. Location information is made available to mobile applications; they are able to register interest in user movement events. This allows programs to be sent to users rather than to a named host. By using location information, the host nearest to a user's current location can be determined. Thus, Tube-based applications are able to *follow* users by triggering movement in response to location events.

7.4.2 Migratory programs

The Tube provides transparent migration amongst heterogeneous computers so migratory applications are simple to realise. High-level application functionality is written to use the Tube. The application binds to resident event, multimedia, trading, communication and user interface rendering facilities as it moves. Knowledge about an application does not have to exist at a site before an instance is sent there for execution.

7.4.3 Stream-based multimedia

Provision of networked audio and video is made through simple objects written in C++. On each site with suitable hardware, a multimedia object factory waits for requests to create the following:

Microphones capture audio and send it to speaker objects over the network

Speakers read audio from the network and play it through loudspeakers

Cameras capture video and send it to television objects over the network

Televisions read video from the network and show it in a window on the screen

Each type of object is linked with a stream library which provides the ability to send and receive data over the network. Microphones and cameras are source objects and have output streams; speakers and televisions are sink objects and have input streams. There is one stream for each data format supported by the object. Streams abstract connections between objects.

Two implementations have been written. Both use IP-based sockets to connect over the network and provide multicasting by creating a separate connection for each destination. Both provide functions for source objects to initiate connections to sinks and write data onto them and for sink objects (transparently) to accept connections from sources and read data from them. A connection made by an output stream for a source object has a unique identifier, which can be used at a later date to close it down.

The use of IP in the current stream implementations is not exposed to applications. Future implementations might use native ATM to transport data or IP multicasting, for example. Provision is made for this by typing the addresses that output streams connect to. An application wishing to send data using an ATM control architecture such as the Hollowman [**Rooney97b**] would specify the destination to its output stream as a Hollowman-type address. Quality of service constraints can be specified on connection setup, although this is specific to the transport method used and is not supported by the current IP-based implementations.

The two implementations of the stream library differ in that one performs handshaking when making a connection and remembers the connection information obtained from doing so (see Figure 7.8(i)), while the other remembers no information about each connection (see Figure 7.8(ii)). The former implementation allows communication state to be kept in the low-level multimedia objects themselves; using the latter lifts this responsibility into application-level code that uses them. An object built with the first implementation can yield its communication state to applications; this can be used in applications involving mobile multimedia.

The multimedia objects all use the same multimedia subsystem (DEC Alpha Multimedia Extensions for Digital Unix) and fixed rate, uncompressed audio and video. Video is sent using UDP/IP and audio using TCP/IP over a local area ATM network. Computers using other capture and playback devices could be used as well, as long as their multimedia objects read and write data in the same format. Microphone, speaker and television objects can share their hardware capture and playback devices. However, the hardware dictates that only one process can capture video on a computer at any one time.

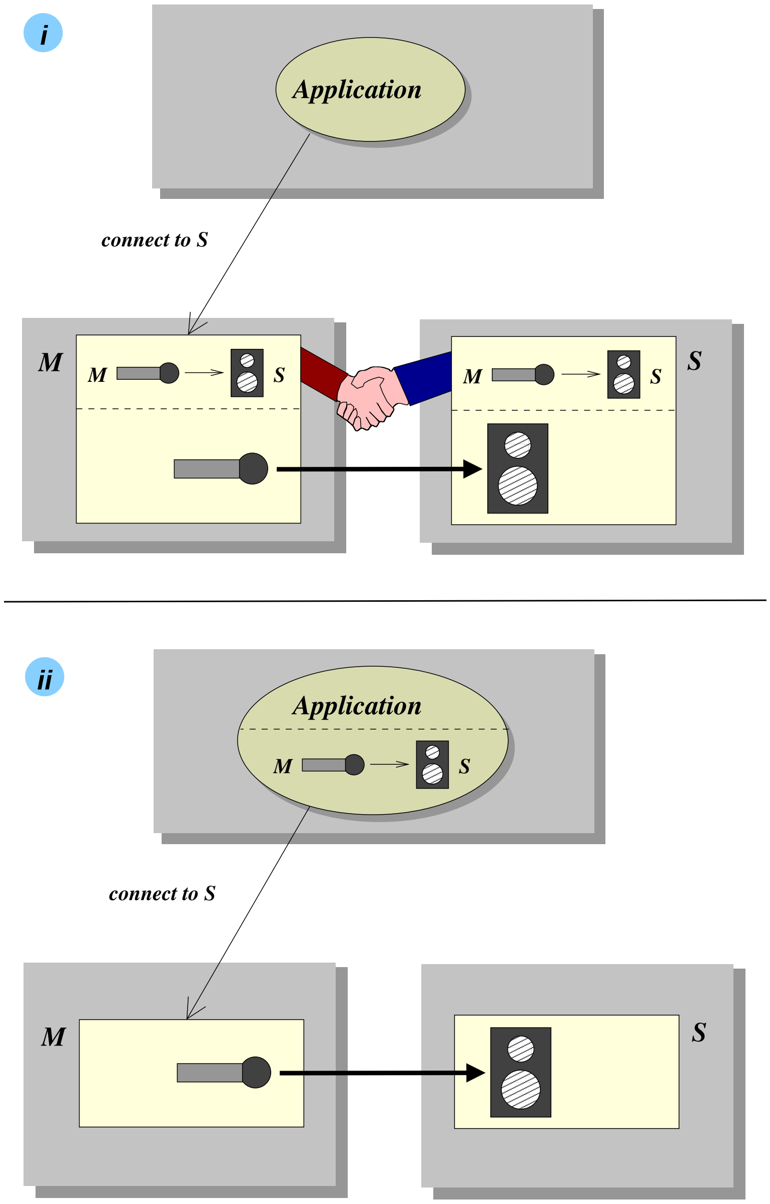

Figure 7.8: Stream library implementations
(i) Remembers connection information in objects
(ii) Maintains no communication state

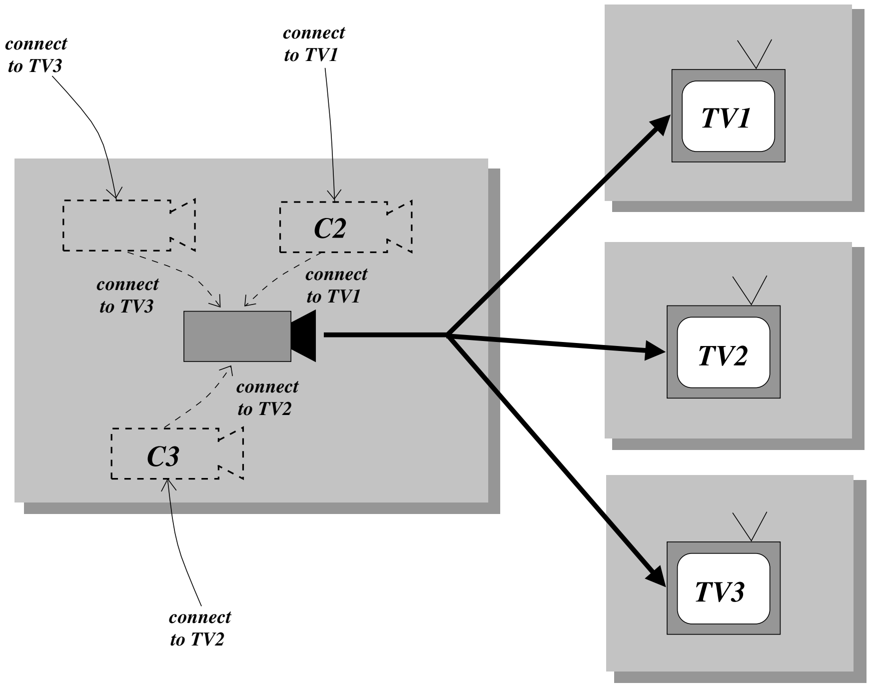

Figure 7.9: Sharing one real video camera using virtual camera objects

When using the stream implementation which remembers connection state, one real camera object is created which distributes video pictures and then virtual camera objects are created subsequently which forward connection requests to the real one (see Figure 7.9). In this way, connection information for applications can continue to be held in separate low-level camera objects. When using the stream implementation which does not remember connection state, this is not necessary since communication state is remembered at a higher level anyway and there is no need to have separate low-level objects.

Both stream implementations require handoff to be handled at a higher level. Communication state, in the form of a list of connections, is either given to the application by the multimedia objects or held there anyway. The implemented multimedia objects can tolerate data loss, so in practice when an application moves, old connections are closed down before it moves and then recreated at its destination. A future implementation might contact some network control entity to have connections re-routed without data loss.

7.4.4 Simple trading system

A simple method has been used for handling mobility in the trading mechanism. When an application moves, it replaces any trader entries pointing

```
<Agency>Tube</Agency>
<Naming>address</Naming>
<addressType>content</addressType>
<addressContent>
<objectType>Tub VideoCamera</objectType>
<html><br>
<img src=http://www.cl.cam.ac.uk/users/dah28/images/camera.gif>
</html>
<objectName>John's camera</objectName>
</addressContent>
<addressQuery>
#BAND(#FIELD(objectType=Tub) #FIELD(objectType=VideoCamera)
      #BAND(#FIELD(objectName=John's) #FIELD(objectName=camera)))
</addressQuery>
<resolvesType>Tube</resolvesType>
<resolvesHostname>britten.cl.cam.ac.uk</resolvesHostname>
<resolvesPortnum>1270</resolvesPortnum>
```

Figure 7.10: Advertisement for a camera object

to its location with a marker indicating that movement is in progress. In the current implementation, applications should periodically poll the trader when discovering that movement is in progress, until the new location is available. In more advanced trader implementations, they might be told by the trader when the new location is posted by the moving application. Applications are free to handle movement of others in an appropriate manner. They might choose to poll the trader for a period of time, perform some other task in the meantime or even inform the user. This choice might be hidden by the communication system used for applications to talk to each other.

This method as it stands is not scalable, but has proved sufficient for experiments in a local area network. Other possibilities are discussed in Section 7.3.4.

The SGML-like syntax used for application advertisements has been mapped onto implementations which use the COBRA information retrieval system [**Mills97b**] and regular expression matching. See Section 4.4.1 for more details. Applications wishing to share trading information must use the same trader implementation.

All multimedia objects and Tube sites can import and export advertisements. As an example, Figure 7.10 shows the advertisement for a camera object called *John's camera*.

The `Agency` tag specifies that this is an ordinary advertisement within the Tube system. The `Naming` tag indicates that the advertisement contains a mapping between two addresses, typically from an abstract spec-

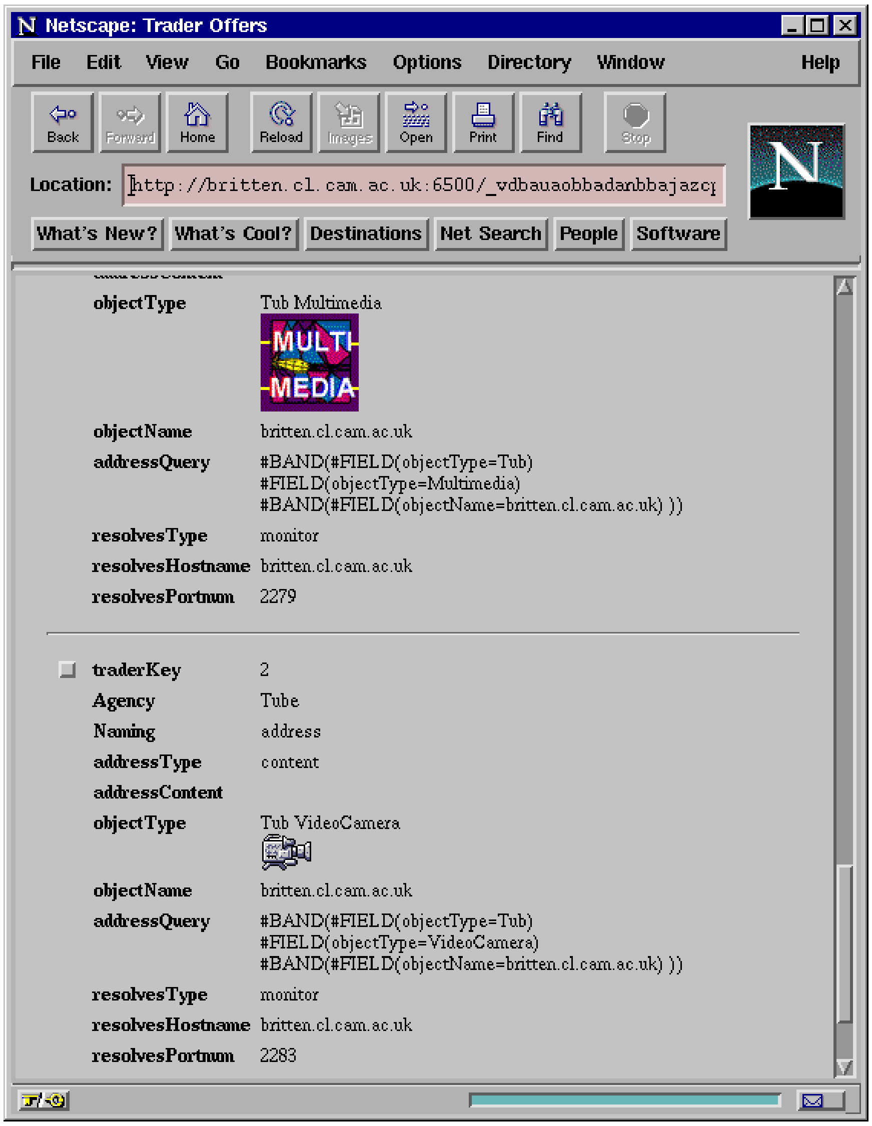

Figure 7.11: Trader advertisements in a Web browser: a Multimedia object factory and a video camera

ification to some low-level location information. The first is specified by tags beginning with `address`; the second is specified by tags beginning with `resolves`. In this example, the abstract address is a content address (specified by the `addressType` tag). A content address contains arbitrary tags. Here, the type of the object is defined by the `objectType` tag and its name by the `objectName` tag. The `html` tag allows arbitrary HTML markup to be inserted when Web browsers display the advertisements in a trader (an example is shown in Figure 7.11).

The location of the camera is on the host `britten.cl.cam.ac.uk`, port 1270. The `resolvesType` tag indicates that the object can be controlled using the Tube mobile code system. Applications send the following query to the trader in order to look up the advertisement:

```
#BAND(#FIELD(objectType=Tub) #FIELD(objectType=VideoCamera)
      #BAND(#FIELD(objectName=John's) #FIELD(objectName=camera)))
```

See Section 4.4.1 for details of the syntax. The matching advertisement can be processed to extract the host name and the port number. Resolution is recursive, so that if a query returns an address that itself maps onto another, the lowest-level location is returned. In this way, indirection can be used, for instance to map from a person's name to his location to the camera at that location.

7.4.5 Application-level communication

Two systems have been used for inter-application communication. In initial experiments, MSRPC [**Roscoe94**] enabled mobile applications to talk to multimedia objects. A stub generator that outputs Scheme from object interface definitions is used to allow programs running on Tube sites to do this.

The Drool distributed object system described in Section 4.4 was used in subsequent experiments. Transition was smoothed at first by hiding MSRPC communication behind Drool proxies. This allowed Tube-based applications to be written using Drool and the original multimedia objects with MSRPC interfaces. The multimedia objects were then rewritten to use Drool for communicating with the Tube mobile code system. A lightweight version of the Tube (see Section 2.8) is embedded inside them. It is able to receive simple Scheme expressions, execute them and return the results.

Multimedia object factories make the following functions available:

- `camera` creates a video camera object
- `tv` creates a video window (television) object
- `mic` creates a microphone object

```
(let* ((stream-type
         (make-object () ()
            (connect: (addr (host <string>) (port <integer>)) -> <integer>
               (on addr `(connect ,host ,port)))
            (connect: (addr (name <string>)) -> <integer>
               (on addr `(connect ,name)))
            (close: (addr (chan <integer>))
               (on addr `(close ,chan)))))
      (source-cloxy
         (make-object () ((stream (make-object stream-type ())))
            (advert: ()
               (save this))
            (content: () -> <string>
               "<Cloxy>Source</Cloxy>")
            (query: () -> <string>
               "#FIELD(Cloxy=Source)")
            (stream: () -> <stream-type>
               stream)
            (exit: (addr)
               (on addr '(exit))))))
   ...)
```

Figure 7.12: Source Class Proxy

- `speaker` creates a speaker object

All take as a parameter the name of the object to create.

Source objects (microphones and cameras) make the following functions available:

- `connect` takes as a parameter either the name of a sink object or its location and connects the audio or video stream to it; it returns a key uniquely identifying the connection

- `close` takes as a parameter the key of a connection, which it closes

If the name of a sink object is given to `connect`, its location is resolved by the source object through trader lookup. `connect` can be called more than once because the stream libraries support multicast.

The multimedia objects are given a Drool abstraction in Scheme so that mobile applications can control them easily. They do not have to dispatch expressions to objects themselves. An example is shown in Figure 7.12. See Chapter 4 for details of Drool syntax. It shows a class proxy, or *cloxy*, for source objects. A cloxy encapsulates the behaviour of a type of object, in this case source objects (cameras and microphones).

First, the type of a source stream is defined. The `connect` and `close` methods use the `on` function (see Section 2.5.4) to forward the appropriate

Scheme expression to the actual object, whose address is passed in the `addr` parameter. Neither the stream nor the cloxy hold state. The cloxy is then defined. A stream instance is declared when a cloxy is made. The `advert`, `content` and `query` methods enable cloxies to be published in traders. The `stream` method simply returns the stream object; there is one stream object for each instance of the cloxy. The stream object's methods could be collapsed into the cloxy; they are separated to make it clear that streams are joined together, not objects — future camera implementations might have another stream for compressed video, for instance. Finally, an `exit` method is provided to terminate the object.

An application might join John's camera to Kerry's window with the following expression:

```
((source-cloxy 'stream)
     'connect
     (make-content-address
               'unspecified
               "#BAND(#FIELD(objectType=VideoCamera)
                      #FIELD(objectName=John's)
                      #FIELD(objectName=camera))")
     "Kerry's window")
```

The Drool-based multimedia objects use the stream library that does not remember connection state. Using cloxies to provide abstraction pushes the responsibility to remember connection state up into the application.

However, suppose instead that each *instance* of a multimedia object creates on its local Tube site a Scheme object which talks directly to it. This *shadow* object can be used to remember connection state. Applications contact the shadow when making or destroying connections, which captures as a side effect information about the connections in place. Instead of a one-to-one mapping between multimedia objects and their shadows, many shadow objects might map to the same real one. This is useful when hardware devices cannot be shared by more than one process. If shadows are used to remember connection state, those shadowing sink objects must be told about connection activity too.

Decentralised mobility can be implemented by making mappings between shadow and real multimedia objects temporary and based on a user's location. Shadow objects move to follow a user and bind to the real ones at his location (see Figure 7.13). Since they remember communication state, they can re-establish connections.

Drool uses proxies that are delivered at run-time to give clients the wherewithal to communicate with a service. Mobile-aware proxies might be sent to clients by services that support mobility. These would implement server-specific mobility policy. For example, a server could tell its proxies that it is moving so that clients calling into them find this out without going

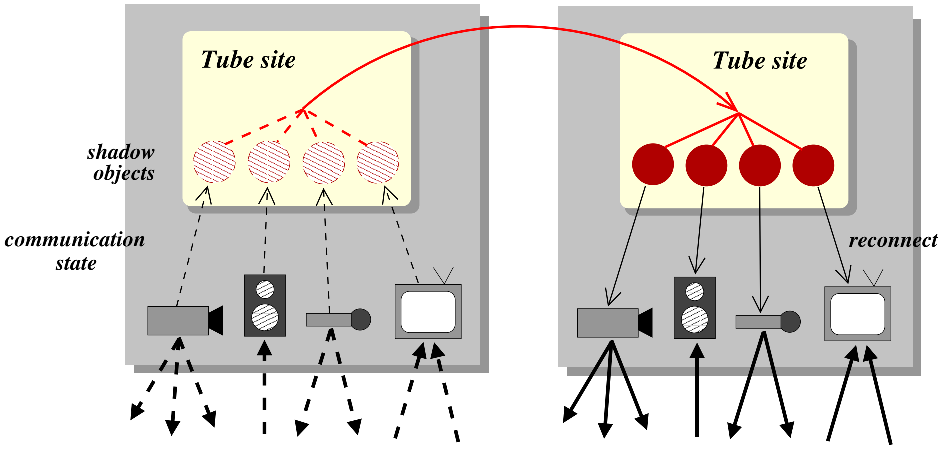

Figure 7.13: Supporting user mobility using mobile Scheme objects

to the trader. A server might wish to provide its own retry algorithm to clients which wish to cope with mobility, or even provide some code which mimics its own behaviour whilst it is moving. Clients might be isolated from knowing about mobility by sending to them proxies which themselves perform retry of trader lookup whilst appearing to the client to block. The ability to send and dynamically change the code used by clients to access Drool server objects allows for flexible mobility policy to be implemented.

Making basic multimedia facilities simple and shifting responsibility for managing mobility to a higher level enables different implementations to co-exist. Implementing high-level mobile applications using mobile code allows them transparently to migrate along with the components they manage.

7.4.6 Mobile user interfaces

The Tube can save the state of user interfaces created by programs, including the callbacks associated with them. This is used to allow applications to carry their user interfaces with them when they move. In conjunction with event-driven migration, user interface mobility allows a user to have his applications appear and execute on computers in his vicinity as he moves.

The work on stateless servers described in Chapter 6 lends itself to mobility because applications are written only to execute temporarily on compute servers before embedding themselves in documents. When it is not running, knowledge about a stateless server application is held only in a document displayed to the user. Applications can thus be moved simply by transmission of a document (see Figure 7.14).

Mobile stateless server applications are supported by two functions. The first retrieves the URL being displayed by a Web browser. The second sends

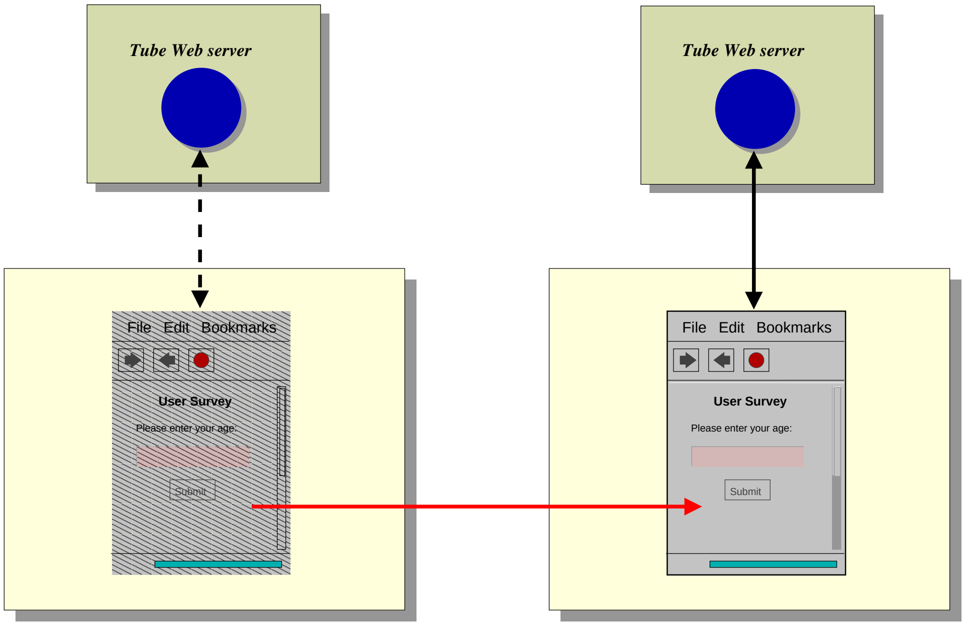

Figure 7.14: Mobile stateless server applications

a browser to a given URL. Since the implementation of stateless server applications embeds their states inside HTML document links, moving them simply requires capturing their URLs (current states) and then sending browsers running at the destination to visit those URLs after movement.

7.5 Experiments

All of our experiments have employed mobile code in provision of high level application coordination. Applications can be prototyped rapidly as mobile Scheme programs which can create and configure distributed multimedia objects. They can also register interest in and respond to user movement.

This section describes briefly two implementations of mobile video conferencing and one of a mobile Web-based application.

Follow-me audio and video is used in mobile video conferencing. Wherever a user moves, video windows and software speaker objects follow so that he can continue to watch and listen to other people. His camera and microphone objects follow him too, so that others can continue to see and hear him.

Centralised and decentralised implementations are given. The decentralised implementation is more suited to wide-scale mobility but is also more complicated. The centralised implementation is simpler and suited for supporting mobility in a local area network.

7.5.1 Decentralised mobile video conferencing

The first implementation of follow-me audio and video conferencing distributes information about a user's connections with other people as much as possible (see Figure 7.15).

On each machine, a multimedia object factory and a camera object are permanently resident. There can only be one camera per machine because of hardware limitations. When a user moves to a machine, the following objects are created there for him with the factory:

- a virtual camera that remembers all outgoing video connections, forwarding them to the resident real camera object
- a video window (television) for each incoming video connection
- a microphone which serves all his outgoing audio connections
- a speaker for each incoming audio connection

These are all built with the version of the stream library that performs handshaking between connecting parties and remembers connection information. Each object returns its communication state (list of connections) when it is terminated. For each connection, the names of the users at each end are given.

Response to the user's movement is coordinated by a management object written in Scheme and running on a local Tube site. It registers an interest in his movement events with the active badge location service. The user generates movement events by clicking the button on his badge.

Conferencing must be set up through management objects so that each can record which objects and connections it creates for its user. This information is required when a user moves. If a manager is told that its user has clicked his badge's button in a new room, it does the following things (see Figure 7.17):

1. Terminates all of the user's multimedia objects at his old location, receiving a list of connections for each in doing so.
2. Looks up in a location database which computer is nearest to the user's new position.
3. Decides whether it needs to move to another Tube site. Rather than there being a Tube site running on each machine, there might only be one per domain. If it needs to move between domains, it simply dispatches itself there.
4. Recreates on the computer at the user's new location all of the objects it terminated, using the factory there.

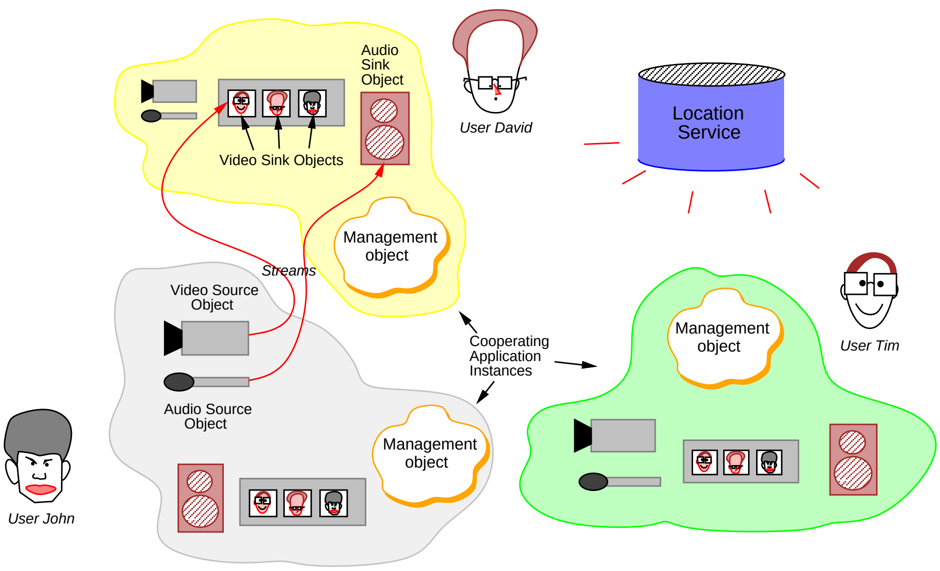

Figure 7.15: Decentralised mobile video conferencing

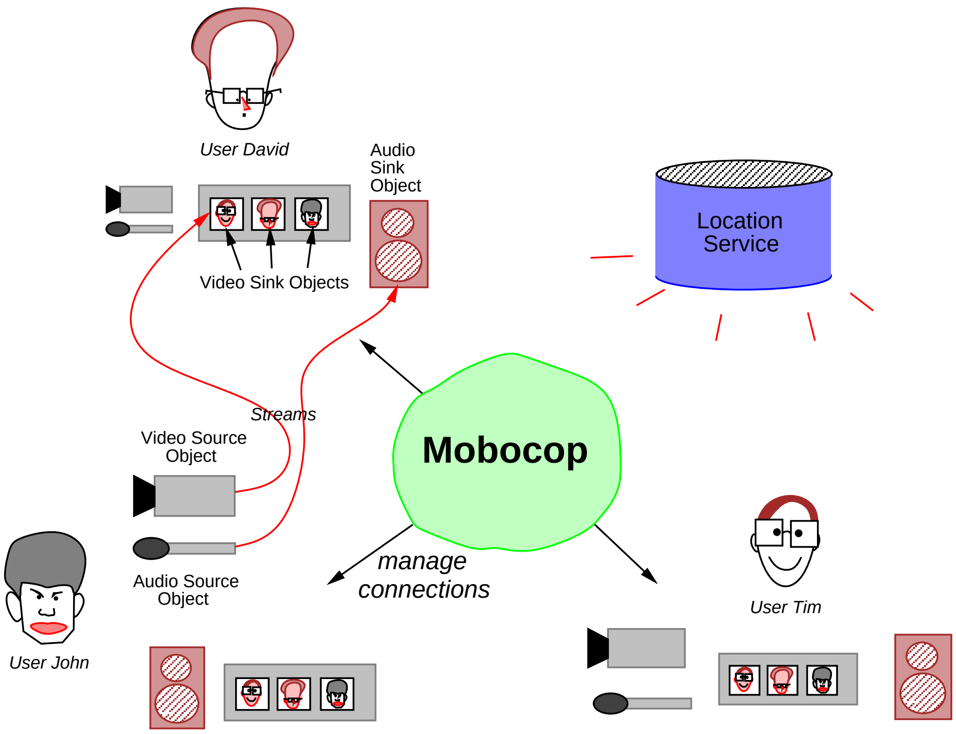

Figure 7.16: Centralised mobile video conferencing

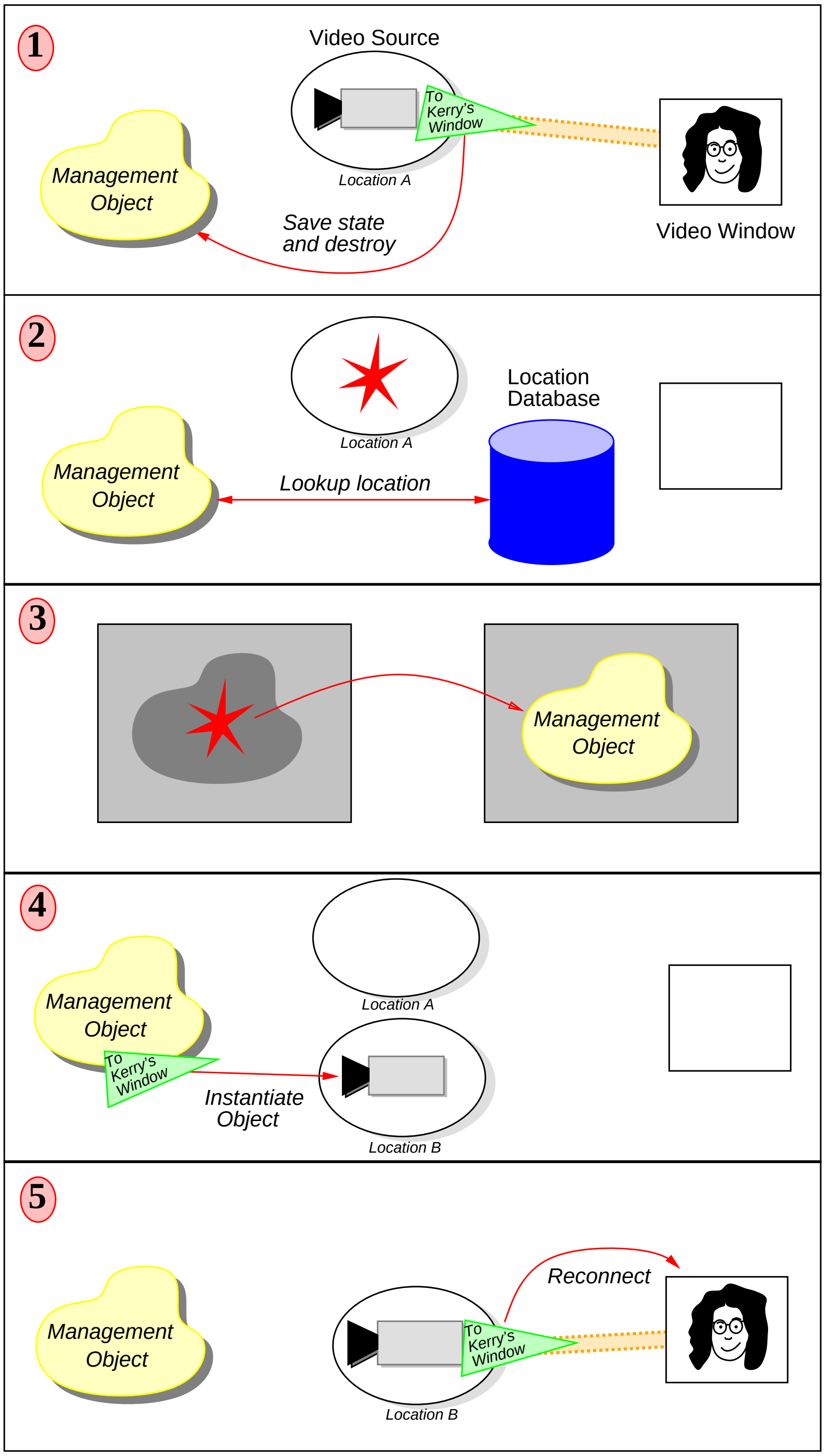

Figure 7.17: Stages in moving a media object

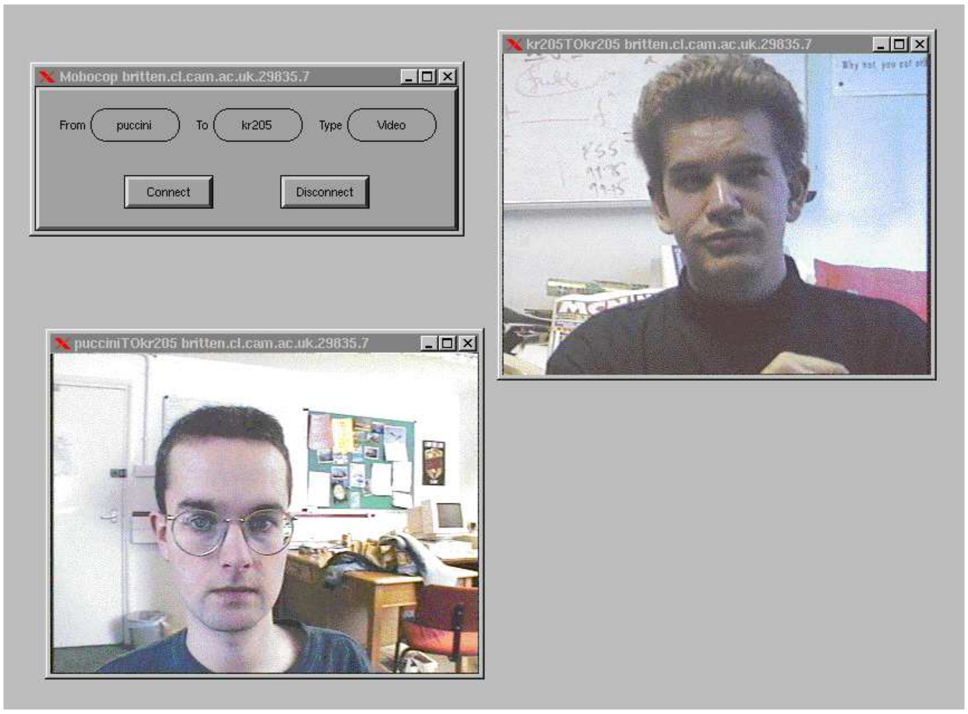

Figure 7.18: The Mobocop video conference manager

5. Re-establishes all of the connections.

This video conferencing implementation consists of a number of mobile per-user connection managers which create low-level objects to process data and remember connection information. As the manager moves to follow its user, objects and connections are created and destroyed appropriately.

7.5.2 Centralised mobile videoconferencing

The second implementation of follow-me audio and video conferencing holds all information about connections between users in one place (see Figure 7.16). On each machine, a camera, microphone and multimedia object factory are created.

On a Tube site running on the conference chairman's computer, a video conference application called Mobocop is executed (see Figure 7.18). The chairman uses this to make connections between users. For outgoing audio and video, Mobocop uses the resident camera and microphone running on the user's nearest computer. For each incoming audio or video connection, it creates a new speaker or video window there. Mobocop registers interest in movement of all of the users participating in the conference.

Mobocop knows about all connections, speakers and video windows associated with each user. When a user clicks on his badge's button in a new room, Mobocop is notified and terminates his speakers and video windows on the computer at his old location. They are then recreated at his new location and connections remade to them and from the cameras and speakers running there.

Mobocop can itself move to follow the conference chairman. More than one Mobocop can be run over the same network because connections are remembered at a high level, not on the resident camera and microphones or transient speakers and video windows. If each user is given a Mobocop which manages his connections, the implementation becomes more like the decentralised one described in the previous section.

The multimedia objects in this implementation use the simple version of the stream library that remembers no information about its connections. Management of communication state uses the class proxies described in Section 7.4.5 and is pushed up to the application level.

7.5.3 Measurements

Timings were taken using the centralised video conference application described in the previous section. An audio and video conference was set up between three people with access to three DEC Alpha computers, each in a different room, equipped with a video camera, microphone and speakers, connected to a local-area ATM network and running a Tube site under Digital Unix. One of the users was designated chairman and the Mobocop application was able to follow him as he moved.

Each user was able to see and hear the others as they moved between rooms. All the timings given below include a constant overhead of 2 seconds for the active badge system to report badge sightings to Tube sites.

The average time taken for a user's multimedia objects to appear at his destination when he moved between rooms was 8.9 seconds (815 milliseconds to close the old connections, 1.614 seconds to destroy the old objects, 2.934 seconds to create the new objects and 1.509 seconds to recreate the connections between the new objects). The time taken between him arriving at a new room and the other users seeing him there was 9.2 seconds on average.

These times proved adequate for conferences with only sporadic movement. They are also respectable given the heavy load placed on the computers. Each user had outgoing audio and video connections to the other users and to himself. That is, each user had six outgoing connections and there were eighteen connections in total. The video objects had to send and receive 1.33 megabytes of data over each connection per second (there were 12 frames per second, each of which was 384 by 288 pixels in 8-bit colour). The audio objects had to send and receive 22 kilobytes of data over each connection per second (it was sampled at 11kHz, using 16 bits per sample).

The DEC Alpha computers were overloaded by this amount of data, so that the playing of audio became intermittent and the display of video jerky. Handling of user mobility had to compete with the processing of this multimedia data, without help from the Digital Unix scheduler which cannot impose quality of service guarantees. Use of an operating system designed to provide such guarantees would have helped here [**Leslie96**]. Performing stream handoff at a lower level, instead of tearing down and rebuilding connections, would have reduced the amount of software involved in re-routeing data (see Section 7.3.3). Using a separate device for sending data from video cameras onto the network [**FORE97**] would have reduced the load placed on the computers.

Note that the related projects cited in Section 7.2 above give no performance measurements.

7.5.4 Mobile Web-based applications

The Earls Colne information retrieval interface described in Section 6.6 has been made mobile. This is achieved with a generic Tube-based daemon which moves around a network following a user. Whenever the user moves, it finds out which URL he is looking at in his Web browser, moves to his new location and updates the browser there so that it shows the same page.

Any stateless server application written using the implementation described in Chapter 6 can be moved in this way, simply by viewing it in a Web browser. When one moves room, the daemon follows, bringing with it the URL containing the application's current state. It is re-targeted to use the nearest Tube Web server. Because stateless server applications embed all of their state in Web pages displayed by browsers, they do not have to be changed to be made mobile.

7.6 Summary

Supporting mobile multimedia applications and detecting movement of people allows a user's working environment to follow him around. This chapter has described a framework for supporting user mobility. A location service monitors user movement and notifies applications. Migratory programs and mobile user interfaces follow users around as they move. Stream-based multimedia objects send audio and video data over networks. A simple trading service supports location of mobile objects. Finally, this chapter described an implementation of the framework which uses mobile code and some experiments carried out with it.

Chapter 8

Summary and Conclusions

8.1 Summary

This dissertation has described the implementation and application of a higher-order mobile code system. It has discussed higher-order state saving, provision of communication abstractions and some real uses of mobile code in distributed systems. The research statement made in Section 1.4 has been expanded upon by subsequent chapters in the following manner:

- *The research described in this dissertation has produced a mobile code system that supports the transmission of higher-order functions and continuations over computer networks.*

  Chapter 2 described the Tube mobile code system, which allows Scheme expressions and programs to be transmitted over a network. Chapter 3 described an interpreter that allows a program's execution state to be migrated across a network by serialising closures and continuations into a byte stream.

- *The mobile code system has been used to build a distributed processing environment that is flexible in the communication abstractions it provides to applications. Varying degrees of abstraction are provided, from basic message passing to remote method invocation and from tight integration with application state to object-based data hiding.*

  Chapter 2 showed how the Tube mobile code system supports simple messaging between distributed processes and remote evaluation of language expressions by allowing messages to contain programs. Chapter 4 showed how mobile code can be used to move application-level functionality, share connections and provide networked access to execution state. It also showed how an application's data can be given an object-based abstraction.

- *A distributed object abstraction has been written which allows an application to define, offer and adapt at run-time the means that clients should use to communicate with it.*

  The distributed object system described in Chapter 4 uses mobile code to implement dynamic communication proxies that can be delivered and updated at run-time.

- *The mobile code system has been used to enhance existing distributed applications. It has been applied to ATM network control, to the management of state in client-server interaction and to event-driven user mobility.*

  Chapter 5 described the use of mobile code to decentralise control and reduce network usage in an ATM network control architecture. Chapter 6 showed how mobile code can be used to store persistent knowledge about client-server applications in the documents exchanged between client and server. Chapter 7 presented a framework to support mobile multimedia applications and described an implementation that uses mobile code.

8.2 Conclusions

8.2.1 Value of mobile code

Mobile code is useful because it allows a networked service to be written without having to decide in advance which abstractions to provide for clients. Clients can impose their own abstractions on the service by sending code to it. Since clients know best how they wish to process the service's data, they can optimise use of the network by filtering the data before it is returned.

This is useful for the development of distributed applications by more than one person. The programmers of client programs can develop optimised use of a service without involving its programmer. Once development has finished, the mobile code might be made a permanent part of the service. However, a service which needs to support unbounded numbers of clients that make unpredictable use of its (immutable) data is effectively under constant development and can make use of mobile code a permanent part of its functionality.

Reduced network usage and development flexibility are offset by performance and security considerations. In order to use mobile code in networks of heterogeneous computers, it must either be interpreted or compiled to native code before being executed. These both impose a performance penalty compared with pre-compiled code. Therefore, mobile code is not useful for distributed applications that do not need to sacrifice performance for flexibility because they can predict how their data will be processed and transferred over the network.

However, as use of the Internet grows, it will become increasingly difficult to predict how clients of a service will want to process its data, making the case for mobile code greater. The widespread use of (interpreted) scripting languages shows that the need to pre-compile applications is diminishing and that interpretation is well-suited for high-level control [**Ousterhout97**].

Care must be taken to make restriction of access to sensitive resources as simple for mobile code interpreters as it is for protocol handlers. One should also be able to limit the amount of time that mobile code can execute for, so that it cannot impose an endless burden on a service's host machine. Whilst networked services that do not admit mobile code must also be able to cope with denial of service attacks, widespread deployment of mobile code for applications such as those described in this dissertation relies on research into making it secure (see Section 1.5).

The Tube mobile code system described in this dissertation has proved to be efficient enough for experimenting with (see Section 3.5.5). It provides simple security by restricting the functions and data that programs acquired from the network can access (see Section 3.5.2). It can also control the duration and speed of a program's execution (see Section 3.5.1).

The Tube uses continuations to implement mobile computation, which is a powerful, high-level and portable technique (see Chapter 3). However, there are now many mobile code systems available (see Sections 2.2 and 3.2), each targeted towards different application areas. The exact method used to achieve mobility is less important than the deployment of mobile code itself.

8.2.2 Deployment of mobile code

If mobile code is widely deployed, future distributed systems might consist of many services that accept and execute programs from the network. They would delegate responsibility for providing abstractions, and making optimal use of the network, to their clients.

Chapter 6 showed that some applications can benefit by placing more load on the network and demonstrated that mobile code can be used to do this. For these applications, optimal use of the network is not necessarily minimal use. Use of mobile code allows applications to determine where they execute, how they use the network and where their state is held at each stage of their execution.

Chapter 4 showed that different ways of communicating can be layered on top of mobile code. Chapters 2 and 3 showed that mobile code can itself be represented using a simple message format. Applications which interact using a common message format that supports mobile code can use it to implement their own methods of interaction, such as protocols, remote procedure call or distributed objects and share them at run-time. This is already starting to happen with Java [**Wollrath96**].

In contrast, approaches such as CORBA [**Siegel96**] standardise at a level of interaction that requires distributed object software to be used by (and usually compiled into) every client of a service. Chapter 4 showed how a distributed object abstraction can be provided as an optional extra, so that applications not wishing to use it do not pay for it. A service is free to interpret the messages it receives however it wants. Programmers of client applications can either communicate with it directly or through proxies delivered by the service.

Chapters 5 and 7 have shown that mobile code can also be applied to existing distributed systems. There is a burden of proof that the addition of mobile code can help either prototyping or to reduce network usage and maintenance costs. The case for mobile code is stronger for distributed systems that involve large numbers of programs which evolve over time than for distributed systems running over a local area network, with small numbers of programs written by one or two people.

Research and commercial interest has been growing in mobile code during the last three years (see Sections 2.2 and 3.2). New systems are announced regularly (recent examples are Voyager [**ObjSpace97**], Concordia [**Mitsubishi97**] and Odyssey [**GenMag97**][1]). The more that mobile code is used in applications, the greater acceptance it will achieve.

8.2.3 Contribution to research

The main contribution of this dissertation to research is to demonstrate some specific applications of an implemented mobile code system. It is hoped that this will strengthen the case for using mobile code in future research and development.

In order to investigate the potential applications of general support for mobile code, a system was designed and implemented that compares well with contemporary research efforts. Migration is achieved by mechanisms which are transparent to the application and portable across hardware and operating systems.

The system was used for a variety of applications, each of which was shown to derive real benefit from the ability to migrate. New approaches were explored which are applicable to emerging ubiquitous computing environments, as well as to more established distributed computing applications.

[1] these systems all implement mobile programs in Java using a similar technique to Aglets (see Section 2.2.2)

Bibliography

[**Acharya97**] Anurag Acharya, M. Ranganathan, and Joel Saltz. *Sumatra: A Language for Resource-aware Mobile Programs.* In J. Vitek and C. Tschudin, editors, *Mobile Object Systems*, Lecture Notes in Computer Science. Springer-Verlag, 1997. (p. 26)

[**Acorn97a**] Acorn Computer Group. *The Acorn Network Computer*, May 6, 1997. http://www.acorn.co.uk/acorn/products/nc/. (p. 89)

[**Acorn97b**] Acorn Online Media. *Cambridge Interactive TV Trial*, March 18, 1997. http://www.acorn.co.uk/acorn/news/releases/1997/march/trial.html. (p. 90)

[**Adams88**] Norman Adams and Jonathan Rees. *Object Oriented Programming in Scheme.* In Proceedings of the 1988 ACM Conference on LISP and Functional Programming, July 1988. (p. 72)

[**Adl-Tabatabai96**] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. *Efficient and Language-Independent Mobile Programs.* In Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI), pages 127–136, May 1996. (pp. 3, 11)

[**Agrawal96**] Prathima Agrawal, Partho P. Mishra, and Mani Srivastava. *Network Architecture for Mobile and Wireless ATM.* In Proceedings of IEEE ICDCS '96, May 1996. (p. 119)

[**Almasi95**] G. Almasi, Anca Suvaiala, Ion Muslea, Calin Cascaval, Tad Davis, and V. Jagannathan. *Web* — A Technology to Make Information Available on the Web.* In Fourth IEEE Workshop on Enabling Technology: Infrastructure for Collaborative Enterprises (WET ICE '95), 1995. (p. 96)

[Anderson95] Thomas E. Anderson, David E. Culler, and David A. Patterson. *A Case for Networks of Workstations.* IEEE Micro, February 1995. (p. 89)

[ANSA93] Architecture Projects Management Limited. *Advanced Networked Systems Architecture Testbench Implementation Manual*, 1993. (pp. 71, 112, 122)

[ANSI92] American National Standards Institute. *Database Language SQL*, 1992. ANSI Standard X3.135-1995. (pp. 10, 71)

[Appel92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992. (p. 28)

[Appleby94] S. Appleby and S. Steward. *Mobile software agents for control in Telecommunications networks.* BT Technology Journal, 12(2), April 1994. (p. 78)

[Arnold96] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996. (pp. 12, 95)

[Bacon90] J. M. Bacon, I. M. Leslie, and R. M. Needham. *Distributed Computing with a Processor Bank.* In Schroder-Preikschat and Zimmer, editors, *Proceedings of the Workshop on Distributed Computing, a European Update*, number 433 in Lecture Notes in Computer Science, pages 147–161, Berlin, Germany, 1990. Springer-Verlag. (p. 95)

[Bacon95] J. M. Bacon, J. Bates, R. J. Hayton, and K. Moody. *Using Events to Build Distributed Applications.* In Proceedings of SDNE '95, 1995. (pp. 112, 123)

[Bacon97] Jean Bacon, John Bates, and David Halls. *Location-Oriented Multimedia.* To appear in IEEE Personal Communications, 1997. (p. ix)

[Baldi97] Mario Baldi, Silvano Gai, and Gian Pietro Picco. *Exploiting Code Mobility in Decentralized and Flexible Network Management.* In Proceedings of the First International Conference on Mobile Agents (MA '97), Berlin, Bermany, April 1997. (p. 78)

[Barr93] W. J. Barr, T. Boyd, and Y. Inoue. *The TINA initiative.* IEEE Communications, March 1993. (p. 77)

[**Bates96**] John Bates, David Halls, and Jean Bacon. *A Framework to Support Mobile Users of Multimedia Applications.* ACM Mobile Networks and Nomadic Applications (NOMAD), 1(4), 1996. (p. ix)

[**Becker95**] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer. *Beowulf: A Parallel Workstation for Scientific Computation.* In International Conference on Parallel Processing, 1995. (p. 89)

[**Bharat95**] Krishna Bharat and Luca Cardelli. *Migratory Applications.* In Proceedings of the 1995 ACM Symposium on User Interface Software and Technology, Pittsburgh, USA, November 1995. (p. 111)

[**Bothner97**] Pat Bothner. *Kawa, the Java-based Scheme system*, March 31, 1997. http://www.cygnus.com/~bothner/kawa.html. (pp. 25, 48)

[**Brown96**] Nat Brown and Charlie Kindel. *Distributed Component Object Model Protocol — DCOM/1.0.* Microsoft Corporation, November 1996. http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt. (p. 122)

[**Cardelli94**] Luca Cardelli. *Obliq: A Language with Distributed Scope.* Technical Report 122, Digital Equipment Corporation, Systems Research Center, June 1994. (pp. 27, 111)

[**Cardelli97**] Luca Cardelli and Andrew Gordon. *Abstractions for Mobile Computation*, 1997. http://www.research.digital.com/SRC/personal/Luca_Cardelli/Ambit/Ambit.html. (p. 4)

[**Cejtin95**] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. *Higher-Order Distributed Objects.* ACM Transactions on Programming Languages and Systems, 17(5):704–739, September 1995. (p. 28)

[**Chang96**] Daniel T. Chang and Danny B. Lange. *Mobile Agents: A New Paradigm for Distributed Object Computing on the WWW.* In OOPSLA '96 Workshop, Toward the Integration of WWW and Distributed Object Technology, San Jose, USA, October 1996. (p. 12)

[**Cheriton83**] D. R. Cheriton and W. Zwaenepoel. *The Distributed V Kernel and its Performance for Diskless Workstations.* In Proceedings of the ACM Symposium on Operating System Principles, pages 129–140, October 1983. (p. 95)

[**Clamen90**] Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles, and Jeannette M. Wing. *Reliable Distributed Computing with Avalon/Common Lisp.* IEEE International Conference on Computer Languages, pages 169–179, March 1990. (p. 11)

[**Clinger91**] William Clinger and Jonathan Rees (editors). *Revised(4) Report on the Algorithmic Language Scheme.* ACM LISP Pointers IV, July–September 1991. (p. 7)

[**CNRI97**] Corporation for National Research Initiatives. *Grail Home Page*, March 12, 1997. http://monty.cnri.reston.va.us/grail/. (pp. 10, 95)

[**da Silva97**] M. Mira da Silva and A. Rodrigues da Silva. *Insisting on Persistent Mobile Agents Systems.* In Kurt Rothermel and Radu Popescu-Zeletin, editors, *Proceedings of the First International Workshop on Mobile Agents (MA'97)*, number 1219 in Lecture Notes in Computer Science, Berlin, Germany, April 7–8, 1997. Springer-Verlag. (p. 28)

[**DARPA97**] DARPA. *Foundations for Secure Mobile Code Workshop*, Monterey, California, USA, March 26–28, 1997. http://www.cs.nps.navy.mil/research/languages/wkshp.html. (p. 4)

[**Falcone87**] Joseph R. Falcone. *A Programmable Interface Language for Heterogeneous Distributed Systems.* ACM Transactions on Computer Systems, 5(4):330–351, November 1987. (pp. 11, 71)

[**Farmer96a**] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. *Security for Mobile Agents: Issues and Requirements.* In National Information Systems Security Conference, Baltimore, Maryland, USA, October 22–25, 1996. (p. 4)

[**Farmer96b**] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. *Security for Mobile Agents: Authentication and State Appraisal.* In Elisa Bertino, Helmut Kurth, Giancarlo Martella, and Emilio Montolivo, editors, *Fourth European Symposium on Research in Computer Security (ESORICS '96)*, number 1146 in Lecture Notes in Computer Science, pages 118–130. Springer-Verlag, September 1996. (p. 4)

[**Felten97**] Edward W. Felten. *Java Security: Frequently Asked Questions.* Princeton University Secure Internet Programming Team, April 28, 1997. http://www.cs.princeton.edu/sip/java-faq.html. (pp. 3, 10)

[**Fielding97**] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol — HTTP/1.1.* IETF Request for Comments 2068, January 1997. (p. 97)

[**FORE97**] FORE Systems, Inc. *FORE ATM Video Solutions*, 1997. http://www.nemesys.co.uk/products/video/index.html. (p. 139)

[**Fournet96**] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. *A Calculus of Mobile Agents.* In 7th International Conference on Concurrency Theory (CONCUR'96), pages 406–421, Pisa, Italy, August 26-29 1996. Springer-Verlag. LNCS 1119. (p. 4)

[**Frakes92**] William B. Frakes and Ricardo Baeza-Yates (editors). *Information Retrieval Data Structures and Algorithms.* Prentice Hall, 1992. (p. 101)

[**Franz97**] Michael Franz. *Adaptive Compression of Syntax Trees and Iterative Dynamic Code Optimization: Two Basic Technologies for Mobile-Object Systems.* Technical Report 97-04, Department of Information and Computer Science, University of California, Irvine, USA, February 1997. (p. 10)

[**Fuchs95**] Matthew Fuchs. *Dreme, for Life in the Net.* PhD dissertation, New York University Computer Science Department, September 1995. (p. 28)

[**Fuchs96**] Matthew Fuchs. *Escaping the event loop: an alternative control structure for multi-threaded GUIs.* In C. Unger and L. J. Bass, editors, *Engineering the Human Computer Interface (EHCI '95).* Chapman and Hall, 1996. Published in Engineering HCI. (p. 100)

[**Gawecki96**] A. Gawecki and F. Matthes. *Exploiting Persistent Intermediate Code Representations in Open Database Environments.* In Proceedings of the 5th Conference on Extending Database Technology, EDBT '96, Avignon, France, March 1996. Springer-Verlag. (p. 25)

[**GenMag97**] General Magic. *Odyssey*, April 1997. http://www.genmagic.com/agents/odyssey.html. (p. 144)

[**Giacalone89**] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. *Facile: a Symmetric Integration of Concurrent and Functional Programming.* International Journal of Parallel Programming, 18(2):121–160, April 1989. (p. 27)

[**Goldszmidt95**] Germán Goldszmidt and Yechiam Yemini. *Distributed Management by Delegation.* In Proceedings of the 15th International Conference on Distributed Computing Systems, June 1995. (p. 78)

[**Gosling86**] J. Gosling. *SunDew: A Distributed and Extensible Window System.* In Proceedings of the 1986 Winter Usenix Technical Conference, pages 98–103, 1986. (p. 10)

[**Gosling95**] James Gosling. *Java Intermediate Bytecodes.* ACM SIGPLAN Notices, 30(3):111–118, January 22, 1995. Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations (IR '95). (pp. 10, 25)

[**Götz96**] Hans-Thomas Götz. *The WAVE Project*, December 4, 1996. http://www.ira.uka.de/I32/wave/wave.html. (p. 77)

[**Gray96**] Robert S. Gray. *Agent Tcl: A flexible and secure mobile-agent system.* In Mark Diekhans and Mark Roseman, editors, *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL '96)*, Monterey, California, July 1996. (p. 25)

[**Halls96**] David Halls, John Bates, and Jean Bacon. *Flexible Distributed Programming using Mobile Code.* In Proceedings of the Seventh ACM SIGOPS European Workshop, Connemara, Republic of Ireland, September 1996. (p. ix)

[**Halls98**] David Halls and Sean Rooney. *Controlling The Tempest: Adaptive management in advanced ATM control architectures.* To appear in IEEE Journal on Selected Areas in Communication, 1998. (p. ix)

[**Harter94**] A. Harter and A. Hopper. *A Distributed Location System for the Active Office.* IEEE Network, 8(1), 1994. (p. 114)

[**Hayton96**] Richard Hayton. *OASIS: An Open Architecture for Secure Interworking Services.* PhD dissertation, University of Cambridge Computer Laboratory, 1996. (p. 4)

[**Heimbigner96**] Dennis Heimbigner. *CU Arcadia Project: TPS*, January 1996. http://www.cs.colorado.edu/~arcadia/Software/tps.html. (p. 26)

[**IONA97**] IONA Technologies. *OrbixWeb for Java*, 1997. http://www.iona.com/Orbix/OrbixWeb/index.html. (p. 72)

[**ISO86**] International Organisation for Standardization. *Standard Generalized Markup Language (SGML)*, 1986. ISO Standard ISO 8879:1986. (p. 59)

[**ISO94**] International Organisation for Standardization. *ODP trading function — Part 1: Specification*, 1994. ISO Standard ISO/IEC DIS 13235-1. (pp. 60, 121)

[**Johansen95**] D. Johansen, R. van Renesse, and F.B. Schneider. *An Introduction to the TACOMA Distributed System.* Technical Report, Department of Computer Science, University of Tromsø, Norway, June 1995. (p. 11)

[**Johansen96**] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. *Supporting Broad Internet Access to TACOMA.* In Proceedings of the Seventh ACM SIGOPS European Workshop, Connemara, Republic of Ireland, September 1996. (p. 95)

[**Knabe94**] Fritz Knabe, Pierre-Yves Chevalier, André Kramer, Tsung-Min Kuo, Lone Leth, Sanjiva Prasad, Jean-Pierre Talpin, and Bent Thomsen. *Mobile Service Agents.* Technical Report ECRC-94-2i, European Computer-Industry Research Centre, 1994. (p. 27)

[Knabe95] Frederick Colville Knabe. *Language Support for Mobile Agents.* PhD dissertation, School of Computer Science, Carnegie Mellon University, December 1995. (pp. 4, 27)

[Leslie96] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications.* IEEE Journal on Selected Areas in Communication, September 1996. (p. 139)

[Li94] Guanxing Li, Andrew Herbert, and Dave Otway. *An Overview of the Distributed Interactive Multimedia Architecture*, 1994. APM Limited Report APM.1295.00.05. (p. 71)

[Lucent97] Lucent Technologies. *Inferno*, 1997. http://www.lucent.com/inferno/. (p. 11)

[Magedanz96] T. Magedanz, K. Rothermel, and S. Krause. *Intelligent Agents: An Emerging Technology for Next Generation Telecommunications?* In IEEE INFOCOM 1996, San Francisco, USA, March 24–28, 1996. (p. 77)

[Mathiske95] B. Mathiske, F. Matthes, and J.W. Schmidt. *Scaling Database Languages to Higher-Order Distributed Programming.* In Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Italy, September 1995. (p. 25)

[Mathiske96] B. Mathiske, F. Matthes, and J. W. Schmidt. *On Migrating Threads.* Journal of Intelligent Information Systems, 1996. (p. 25)

[Matthes94] F. Matthes and J. W. Schmidt. *Persistent Threads.* In Proceedings of the Twentieth International Conference on Very Large Databases, VLDB, pages 403–414, Santiago, Chile, September 1994. (p. 25)

[Matthes95] F. Matthes, G. Schröder, and J.W. Schmidt. *Tycoon: A Scalable and Interoperable Persistent System Environment.* In M. P. Atkinson, editor, *Fully Integrated Data Environments.* Springer-Verlag, 1995. (p. 25)

[McCabe95] Frank G. McCabe and Keith L. Clark. *April — Agent PRocess Interaction Language.* In M. J.

Wooldridge and N. R. Jennings, editors, *Intelligent Agents*. Springer-Verlag, 1995. (p. 27)

[McGraw96] Gary McGraw and Ed Felten. *Java Security: Hostile Applets, Holes and Antidotes.* John Wiley and Sons, 1996. (p. 3)

[Merwe97] Kobus van der Merwe and Ian Leslie. *Switchlets and Dynamic Virtual ATM Networks.* In Proceedings of IM '97, San Diego, USA, May 1997. (p. 75)

[Merwe98] Kobus van der Merwe and Ian Leslie. *Service Specific Control Architectures for ATM.* To appear in IEEE Journal on Selected Areas in Communication, 1998. (p. 75)

[Microsoft97] Microsoft Corporation. *Active Platform*, April 15, 1997. http://www.microsoft.com/activeplatform/default.asp. (p. 10)

[Mills97a] Tim Mills, Ken Moody, and Kerry Rodden. *Providing World Wide Access to Historical Sources.* In Proceedings of the Sixth World-Wide Web Conference, Santa Clara, April 1997. (pp. 100, 103)

[Mills97b] Timothy J. Mills. *Content Modelling in Multimedia Information Retrieval.* PhD dissertation, University of Cambridge Computer Laboratory, 1997. In preparation. (pp. 17, 61, 101, 127)

[Minsky96] Yaron Minsky, Robbert van Renesse, Fred B. Schneider, and Scott D. Stoller. *Cryptographic Support for Fault-Tolerant Distributed Computing.* In Proceedings of the Seventh ACM SIGOPS European Workshop, Connemara, Republic of Ireland, September 1996. (p. 3)

[Mitsubishi97] Mitsubishi Electric Information Technology Center America. *Concordia Home Page*, April 13, 1997. http://www.meitca.com/HSL/Projects/Concordia/. (p. 144)

[Morrison93] R. Morrison, A. L. Brown, R. C. H. Connor, Q. I. Cutts, A. Dearle, G. N. C. Kirby, and D. S. Munro. *The Napier88 Reference Manual (Release 2.0).* Technical Report CS/93/15, University of St. Andrews, 1993. (p. 27)

[ObjSpace97] ObjectSpace, Inc. *Voyager: Agent-enhanced Distributed Computing for Java*, 1997. http://www.objectspace.com/Voyager/voyager.html. (p. 144)

[OOC97] Object Oriented Concepts, Inc. *OmniBroker Home Page*, 1997. http://www.ooc.com/ob.html. (p. 71)

[Oracle96] Oracle Corporation. *Network Computing Architecture White Paper*, September 1996. http://www.oracle.com/nca/html/nca_wp.html. (pp. 89, 97, 105)

[Ousterhout97] John K. Ousterhout. *Scripting: Higher Level Programming for the 21st Century.* Sun Microsystems Laboratories, May 10, 1997. http://www.sunlabs.com/people/john.ousterhout/scripting.html. (p. 143)

[Padget92] Julian Padget and Greg Nuyens (editors). *The* EuLisp *definition.* Technical Report, University of Bath, 1992. (p. 43)

[Partridge92] Craig Partridge. *Late-Binding RPC: A Paradigm for Distributed Computation.* PhD dissertation, Harvard University, Cambridge, Massachusetts, March 1992. (pp. 11, 71)

[Peine97] H. Peine and T. Stolpmann. *The Architecture of the Ara Platform for Mobile Agents.* In Kurt Rothermel and Radu Popescu-Zeletin, editors, *Proceedings of the First International Workshop on Mobile Agents (MA'97)*, number 1219 in Lecture Notes in Computer Science, Berlin, Germany, April 7–8, 1997. Springer-Verlag. (p. 26)

[Pope96] Steven Leslie Pope. *Application Support for Mobile Computing.* PhD dissertation, University of Cambridge Computer Laboratory, October 1996. (pp. 26, 119, 122)

[Porter94] J. Porter and A. Hopper. *An ATM Based Protocol for Wireless LANs.* Technical Report 94-2, Oracle Olivetti Research Limited, Cambridge, 1994. (p. 119)

[Queinnec96] Christian Queinnec. *Lisp in Small Pieces.* Cambridge University Press, 1996. (pp. 29, 71)

[Rajagopalan95] B. Rajagopalan. *Mobility Management in Integrated Wireless ATM Networks.* In Proceedings of MOBICOM '95, Berkeley, 1995. IEEE. (p. 119)

[Richardson94] T. Richardson, F. Bennett, G. Mapp, and A. Hopper. *Teleporting in an X Window System Environment.* IEEE Personal Communications, August 1994. (pp. 110, 122)

[Rooney97a] Sean Rooney. *Connection Closures.* Computer Communication Review, 27(2), April 1997. (p. 77)

[Rooney97b] Sean Rooney. *An Innovative ATM Control Architecture.* In Proceedings of IM '97, San Diego, May 1997. (pp. 75, 124)

[Rooney97c] Sean Rooney. *The Structure of Advanced ATM Control Architectures.* PhD dissertation, University of Cambridge Computer Laboratory, 1997. In preparation. (pp. 73, 75, 121)

[Roscoe94] T. Roscoe, S. Crosby, and R. Hayton. *MSRPC II User Manual.* University of Cambridge Computer Laboratory, 1994. (p. 129)

[Rouaix96] François Rouaix. *A Web navigator with applets in Caml.* In Proceedings of the Fifth World-Wide Web Conference, Paris, France, May 1996. INRIA. (p. 95)

[Sapaty94] P. S. Sapaty and P. M. Borst. *An overview of the WAVE language and system for distributed processing in open networks.* Technical Report, Department of Electronic and Electrical Engineering, University of Surrey, UK, June 1994. (p. 77)

[Schilit94] Bill N. Schilit, Norman I. Adams, and Roy Want. *Context-Aware Computing Applications.* In Proceedings of the 1994 Workshop on Mobile Computing Systems and Applications, pages 89–90, Santa Cruz, California, USA, December 1994. IEEE Computer Society. (p. 114)

[Seltzer96] Richard Seltzer, Eric J. Ray, and Deborah S. Ray. *The AltaVista Search Revolution: How to Find Anything on the Internet.* Osborne McGraw-Hill, 1996. (pp. 96, 101)

[Serrano95] M. Serrano and P. Weis. *Bigloo: a portable and optimizing compiler for strict functional languages.* In Second Static Analysis Symposium, Lecture Notes in Computer Science, pages 366–381, Glasgow, Scotland, September 1995. (p. 21)

[Sewell97] Peter Sewell. *Local Channel Typing for a Distributed π-calculus. Abstract.* In Kohei Honda, Martin Odersky, Benjamin Pierce, Gert Smolka, and Phil Wadler, editors, *High-Level Concurrent Languages*, number 164 20.01.–22.01.07 (97041) in Dagstuhl-Seminar-Report series, pages 16–17. IBFI Schloss Dagstuhl, January 1997. (p. 4)

[Siegel96] Jon Siegel. *CORBA — Fundamentals and Programming.* John Wiley and Sons, 1996. Object Management Group. (pp. 80, 122, 144)

[Smith96] J. M. Smith, D. J. Farber, C. A. Gunter, S. M. Nettles, D. C. Feldmeier, and W. D. Sincoskie. *SwitchWare: Accelerating Network Evolution.* Technical Report, CIS Department, University of Pennsylvania and Bell Communications Research, June 1996. White Paper. (p. 77)

[Stamos86] James W. Stamos. *Remote Evalution.* PhD dissertation, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1986. Technical report MIT/LCS/TR-354. (pp. 11, 71)

[Stamos90a] James W. Stamos and David K. Gifford. *Implementing Remote Evaluation.* IEEE Transactions on Software Engineering, 16(7):710–722, July 1990. (p. 11)

[Stamos90b] James W. Stamos and David K. Gifford. *Remote Evaluation.* ACM Transactions on Programming Languages and Systems, 12(4):537–565, October 1990. (pp. 11, 71)

[Straßer96] Markus Straßer, Joachim Baumann, and Fritz Hohl. *Mole — A Java Based Mobile Agent System.* In ECOOP '96 Workshop on Mobile Object Systems, Linz, Austria, July 1996. (p. 12)

[Sun92] SunSoft, Inc. *NeWS 3.1 Programmer's Guide*, 1992. (p. 10)

[Sun96] Sun Microsystems, Inc. *JavaStation — An Overview*, 1996. http://www.sun.com/javastation/whitepapers/javastation/javast_ch1.html. (p. 89)

[Sun97a] Sun Microsystems, Inc. *Java IDL*, 1997. http://splash.javasoft.com/JavaIDL/pages/index.html. (p. 72)

[Sun97b] Sun Microsystems, Inc. *The Java Server Product Family*, 1997. http://jeeves.javasoft.com/. (p. 95)

[Taft85] Ed Taft and Jeff Walden. *PostScript Language Reference Manual.* Addison Wesley, 1985. Adobe Systems. (pp. 9, 71)

[W3C97] World Wide Web Consortium. *CGI — Common Gateway Interface*, 1997. http://www.w3.org/pub/WWW/CGI/. (p. 96)

[Wachowicz96] M. Wachowicz and S. G. Hild. *Combining Location and Data Management in an Environment for Total Mobility.* In Proceedings of the International Workshop on Information Visualization and Mobile Computing, Rostock, Germany, February 1996. (p. 111)

[Waldo94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. *A Note on Distributed Computing.* Technical Report TR-94-29, Sun Microsystems Laboratories, Inc., November 1994. (p. 27)

[Watters96] Auron Watters, Guido van Rossum, and James Ahlstrom. *Internet Programming with Python.* MIS Press/Henry Holt publishers, October 1996. (p. 10)

[Wetherall96] Dave Wetherall and David Tennenhouse. *The ACTIVE IP Option.* In Proceedings of the ACM SIGOPS European Workshop, Connemara, Republic of Ireland, September 1996. (p. 77)

[White94] James E. White. *Telescript Technology: The Foundation for the Electronic Marketplace.* Technical Report, General Magic, Inc., 1994. White Paper. (p. 26)

[Wilkinson97] Tim Wilkinson. *Kaffe — A virtual machine to run Java code*, 1997. http://www.kaffe.org/. (pp. 10, 48)

[Wollrath96] Ann Wollrath, Roger Riggs, and Jim Waldo. *A Distributed Object Model for the Java System.* USENIX Computing Systems, 9(4), 1996. (p. 143)

[**Yemini96**] Y. Yemini and S. da Silva. *Towards Programmable Networks.* In IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, L'Aquila, Italy, October 1996. (p. 77)

[**Zakinthios97**] Aris Zakinthios and E. Stewart Lee. *A Least Privilege Mechanism for User Processes.* In R. K. Iyer, M. Morganti, W. Kent Fuchs, and V. Gligor, editors, *Dependable Computing for Critical Applications.* IEEE Computer Society, March 1997. (p. 3)

[**Zhao97**] Dr. T.C. Zhao and Mark Overmars. *XForms*, 1997. http://bragg.phys.uwm.edu/xforms. (p. 14)