Technical Report

UCAM-CL-TR-415
ISSN 1476-2986

Number 415

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Application support for mobile computing

Steven Leslie Pope

February 1997

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

© 1997 Steven Leslie Pope

This technical report is based on a dissertation submitted October 1996 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Jesus College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-415*

Abstract

In recent years small, completely portable computers have become available on the marketplace. There is demand for such computers, termed *walkstations*, to access network services while retaining their mobility, and to operate effectively in a wide range of conditions. Future office environments are expected to support wireless networks with bandwidths which are several orders of magnitude greater than are available outdoors. In such environments, there will be powerful compute servers available for a walkstation's use.

This dissertation describes a novel architecture called Notus and its support for applications operating in a mobile environment. The concept of the traded handoff is introduced, where applications are able to participate in the handoff process, rebuilding connections to the most appropriate service. This is expected to benefit walkstations which roam over large distances, where connections to servers would otherwise be strained, and also between heterogeneous networks where cooperation between the networks in performing a handoff might be problematic. It is also proposed in this dissertation that applications could benefit from the ability to migrate onto compute servers as a walkstation moves into the office environment. This enables both the walkstation to conserve its own resources, and applications to improve the service provided to the end user. Finally, by interleaving a traded handoff with the migration process, it is possible for a migrating application to easily rebuild its connections as it moves to a new host.

The Notus architecture has been implemented, including a traded handoff service, and a new application migration service. The new application migration service was designed since existing application migration services are unsuited to mobile environments and it enables applications to migrate between heterogeneous hosts with little disruption. Applications which use the service are written in a standard, compiled language, and normal running applications suffer little overhead. A number of existing applications which are representative of a walkstation's interactive desk-top environment have been adapted to use the Notus architecture, and are evaluated.

In summary, this work describes how mobility awareness and the support from appropriate tools, can enable walkstation applications to better adapt to a changing mobile environment, particularly when the walkstation is carried between different network types or over great distances.

Preface

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 my 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 tables, footnotes, and bibliography.

Publications

Some of the work described in this dissertation has been published [Pope96].

Copyright ©1996 Steven Pope. All rights reserved.

Trademarks

Ethernet is a trademark of the Xerox Corporation.
Mips is a trademark of MIPS Technologies Inc.
Ultrix is a trademark of Digital Equipment Corporation.
Unix is a trademark of AT&T.
X-Windows is a trademark of the Massachusetts Institute of Technology.

Contents

List of Figures

List of Tables

Glossary

AMO	At Most Once
ATM	Asynchronous Transfer Mode
DPE	Distributed Programming Environment
FSP	Fixed Switching Point
HDI	Handoff Interface
IDL	Interface Definition Language
IP	Internet Protocol
MRN	Mobile Radio Network
MSBR	Mobile Support Border Router
MSS	Mobile Support Station
NAM	Notus Application Manager
NFS	Sun Network File System
NLE	Notus Language Extensions
NSM	Notus Support Module
NWM	Notus Walkstation Manager
PVM	Parallel Virtual Machine
QoS	Quality of Service
RPC	Remote Procedure Call
SAP	Service Access Point
TCP	Transmission Control Protocol
VCI	Virtual Circuit Identifier
WLAN	Wireless Local Area Network
WWW	World Wide Web

Chapter 1

Introduction

In recent years, computers which are small and light enough to be completely portable have become available. Such computers, termed *walkstations* [Imielinski92] have the attraction of allowing users to access their computing environments from wherever they happen to be. Since computing environments are becoming inherently distributed, and walkstations are expected to make great use of network provided services, wireless connectivity between walkstations and the static network via *base stations* has become important.

As a walkstation roams in a wireless network, it is often not able to communicate with its current base station and must use another. These circumstances require the rebuilding of a walkstation's network connections, a process which is termed a *mobile handoff.*

1.1 Contribution

This dissertation describes a novel architecture called Notus and its support for applications operating in a mobile environment. The architecture supports the new concept of a *traded handoff*, where applications are able to participate in the handoff process, rebuilding their connections to the most appropriate service. This is expected to benefit walkstations which roam over large distances, where connections to servers would otherwise be strained, and also between heterogeneous networks where cooperation between the networks in performing a handoff might be problematic.

It is also proposed in this dissertation that applications could benefit from the ability to migrate onto compute servers as a walkstation moves from an out door low-bandwidth environment, to a high-bandwidth office environment.

This would enable both the walkstation to conserve its own resources, and applications to improve the service provided to the end user. However, existing application migration services are unsuited to mobile environments.

An implementation of the Notus architecture has been made, and is described and evaluated in later chapters. The implementation provides applications with flexibility in their use of the traded handoff, from a default which requires no application involvement, to the complete involvement of an application in the handoff process.

The implementation also demonstrates a new application migration service which is suitable for interactive applications operating in a mobile environment. The service enables the migration of applications between heterogeneous computers and with potentially little disruption to applications during the migration process. Consideration is made for the migration of multi-threaded applications by providing a number of options which ensure consistency between threads during migration. Also, by interleaving a traded handoff with the migration process, it is possible for a migrating application to rebuild its connections as it moves between the compute server and walkstation.

In summary, this work describes how mobility awareness and the support from appropriate tools, can enable walkstation applications to better adapt to a changing mobile environment, particularly when the walkstation is carried between different network types or over great distances.

This chapter will proceed with an introduction to current mobile network architectures, and explains both the traded handoff concept and application migration in a mobile environment.

1.2 Mobile Network Architectures

The means by which a walkstation interacts with its distributed computing environment is first considered. In the simplest case, a walkstation might be periodically connected to a network by plugging it into a *docking station.* Many vendors currently offer docking stations which provide capabilities such as charging a walkstation's batteries, a network connection, and other peripherals such as multimedia devices. This configuration has become quite popular, but has problems when the walkstation is disconnected from its docking station for long periods of time. During these periods the services available depend solely upon the walkstation's own resources.

In order to provide access to the network without requiring a walkstation to be physically connected to a docking station, it is necessary to offer a wireless network interface. In the local area, this is termed a Wireless LAN (WLAN), and implementations have used radio [Kuo74, Porter94, Trotter95] or infra-red [Adams93, Harter93, Condon95] as a communications medium. Current research into hardware integration is progressing towards the availability of a single device which provides both radio and infra-red connectivity [Hager93].

Currently, radio-based implementations offer bandwidths in the region of 10Mbit/s and bandwidths are expected to double in the near future; for example, HIPERLAN defines a standard for 24Mbit/s, operating in the 5.2GHz frequency band [ETS95]. Infra-red network bandwidth is lower, in the region of 2Mbit/s and since infra-red does not penetrate walls, the size of the communicating cell is smaller. This small cell size has been used in applications where the location of a person or piece of equipment in a building is important [Want92b, Harter94]. In the wide area, connectivity is available using the existing Mobile Radio Network (MRN) infra-structure [Rahnema93, Shearer95]. Bandwidths available are much lower than for a WLAN, in the region of 20Kbit/s.

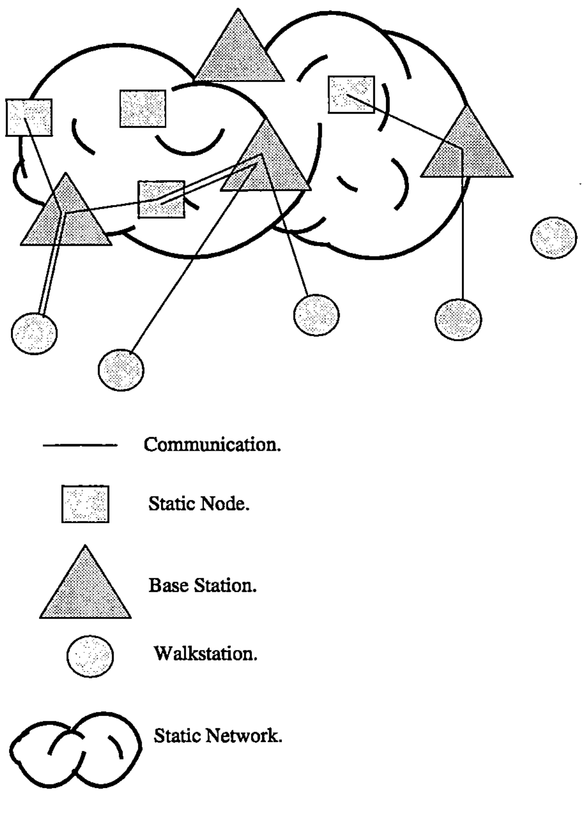


Figure 1.1: A Mobile Network

When considering a network containing walkstations (Figure 1.1), it is usual to make a distinction between *static hosts* and walkstations. Static hosts are connected to the network through a fixed link and tend to move infrequently.

Walkstations are able to access the network using a static host, termed a *base station* as a gateway. Most walkstation architectures [Ioannidis91, Imielinski92, Teraoka93, Bakre94, Comer94] have been designed around this concept.

It is generally accepted that wireless network connections suffer error rates which are much higher than those experienced over wired connections[1]. This can be caused by a number of factors such as:

- signal attenuation through material such as walls,
- front-end-overload, caused by a powerful transmitter of another frequency, such as a microwave oven, overwhelming the filters of the receiver,
- narrow-band interference, caused by another transmitter overlapping the frequency band used by the walkstation,
- multi-path interference, caused by reflections and diffractions of a signal, resulting in destructive interference at the receiver, and
- background noise, for example an infra-red receiver in direct sunlight.

Conversely, errors on the static network are generally caused by dropped packets due to congestion within the network. This has implications for the treatment of errors by wireless network protocols and is discussed further in Section 3.1.

In all walkstation architectures there arises at some point the need for a mobile handoff. This occurs when a walkstation, due to its own movement, is no longer able to communicate with its current base station and has to use another base station. A *mobile handoff* is the process of rebuilding wireless network communications when a walkstation moves from the range of one base station to another. The criteria for defining the appropriate moment to handoff and the choice of the new base station include the changing Quality of Service (QoS) which is provided by the wireless medium, the static network, and the walkstation's servers, together with the predicted QoS and future intentions of movement by the user. The user's intentions of movement can be supplied directly by the user, from diary applications, or guessed using hints from a user-location mechanism such as the Active Badge [Want92b].

For network connections, QoS has generally been described in terms of bandwidth, delay, and jitter, but might also include terms such as cost and expected bit error rates. The end-to-end QoS available between a walkstation and its

[1]It has been reported that in some circumstances an indoor WLAN can achieve comparable error rates to that of a static network [Eckhardt96].

servers depends upon factors such as load on the server and congestion in the static network. Perhaps most significant is the QoS available between the walkstation and its base station. This is likely to be highly variable, depending on: the shielding effects from the surroundings, the varying tariffs charged for access, the distance from the base station, and the load upon the base station.

The salient features of computing in a mobile environment are that walkstations are expected to move and operate over large distances, traversing many different network types or providers. In doing so, walkstation applications are required to adapt to a wide range of conditions, such as communication bandwidths which fluctuate over several orders of magnitude, intermittent connectivity, and various levels of support from static compute resources.

1.3 Traded Handoffs

Existing walkstation architectures typically provide a transparent mobile handoff mechanism at the network level. This essentially redirects connections through a new base station to the walkstation.

At the network level, a connection is described using terms which define the network level Service Access Point (SAP), rather than the properties of the service to which the connection is made. Handoffs made at the network level can only redirect connections back to their original end-points, with the disadvantages being that the network cannot consider the services to which an application is communicating and no use can be made of replicated services which are available locally. For this reason, handoffs of network level connections are termed *flat handoffs* in this dissertation.

Figure 1.2 illustrates a case where a flat handoff results in communication with server S_1, which requires the traversal of data over the backbone network. This has ignored the existence of the locally replicated server S_2, which could have been used instead of the original.

As well as the loss of any potential use of replicated servers, the implementation of a flat handoff mechanism over a wide area is potentially problematic. As a walkstation moves between different network types and administration boundaries, a uniform protocol for handoff requests must be supported and with minimal disruption to the walkstation. Applications for which timeliness is important, such as those dealing with multimedia streams, might well discard data which is "late". If this is due to the time spent by the network in performing the handoff, then the effort expended by the network has simply been wasted. Such circumstances would constitute a violation of the *end-to-end ar-*

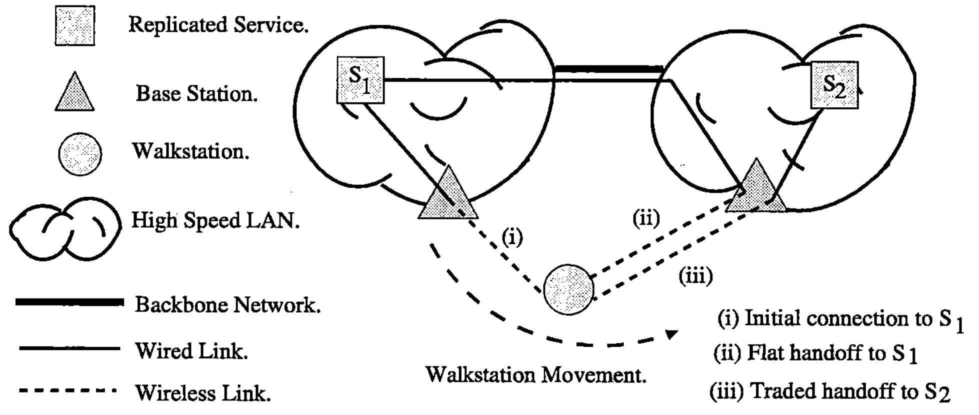


Figure 1.2: A Comparison of Flat and Traded Handoffs

gument, which suggests that functions placed at low levels of a system may be redundant or of little value compared with the cost of providing them at that low level [Saltzer84].

A *traded handoff* is a handoff where connections are rebuilt to the most appropriate service. This requires that connections to services are described at a higher level than the network level SAP. For example, in the case of a video server this abstract description might indicate the service name, the title of the film which is being transmitted, and the encoding format of the video stream.

Use of a traded handoff gives walkstation applications the added benefit of greater flexibility than is possible using flat handoffs. Instead of reconnecting the application to the original endpoint, it becomes possible to connect to locally available, replicated or compatible services. For applications which always require reconnection to their original servers, such as mail retrieval applications, the traded handoff can be reduced to the flat handoff by suitable default behaviours.

1.4 Application Migration

If a walkstation is in a WLAN environment where bandwidth is suddenly reduced[2] or moves to an MRN environment, adaptations which could usefully be made by the walkstation include: the pre-caching of "useful" information and ensuring that information is presented in a bandwidth optimal manner; for example, presenting textual dispatches of news instead of video clips.

[2]Bandwidth may be reduced, for example by interference or by moving into a cell where bandwidth has been previously committed to other walkstations.

Conversely, when moving into a WLAN environment, the walkstation should adapt to the higher bandwidth and the proximity of compute resources. Since it is usually in the interests of a walkstation to conserve its own resources, the WLAN environment provides the opportunity for the walkstation to move its applications to static compute servers. An example application which might benefit from migration as the walkstation moves is the video player shown in Figure 1.3.

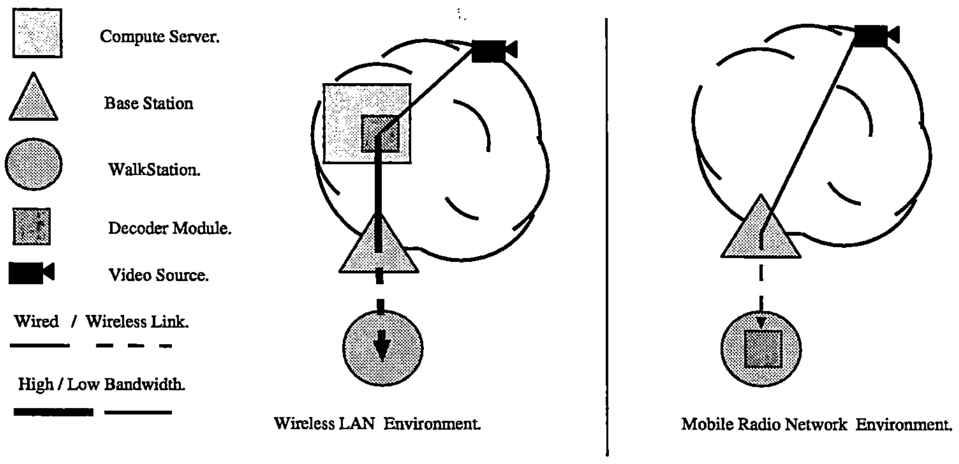


Figure 1.3: A Migrating Application

The part of the application which is decoding a compressed video stream is a computationally intensive task. By moving it to a compute server on the static network, the load on the walkstation is reduced, the trade-off being the requirement to then transmit high-bandwidth frame updates to the walkstation. If post-processing of the video stream were required, this computation would also be a candidate for migration to the compute server. There are many other applications which could be moved to static compute servers, such as news or World Wide Web (WWW) browsers, and it could be argued that much of the walkstation's desk-top environment could usefully be moved onto the static network.

An architecture which enables applications in a mobile environment to be migrated in a uniform manner must support the following requirements:

- a **low-latency** implementation. It is especially important that migration should seem instantaneous to the end user when interactive applications are migrated,
- an implementation which allows migration between **heterogeneous** hosts. There is no reason to suppose that a walkstation would be of the same processor or operating system type as a compute server. It would also be a requirement to migrate applications which have different source code representations for different architectures,
- an implementation which imposes **little overhead** when applications are running normally. Many applications, particularly those which are interactive, would not wish to compromise performance during normal execution in order to support migration.

Assuming that a mechanism is available for application migration which achieves the requirements specified above, there exists a further important requirement. This is to ensure that an application's network connections are rebuilt after the application has migrated. For example, after migration, the video decoder application must continue to receive encoded video from an appropriate source, although not necessarily the original source. As far as the application is concerned, migration to another host is similar to it running on a walkstation which subsequently moves. It is thus desirable to use the same handoff mechanism to support application migration as is used to support walkstation mobility.

Existing application migration schemes [Rashid81, Theimer85, Zayas87, Douglis91a, Litzkow92, Milojicic93] have not fully addressed the issues of performance, heterogeneity, portability, and dependencies on the original host. Implementations usually migrate large amounts of code and data, and require the source and target machines to be of the same processor type and operating system architecture.

1.5 Summary

This chapter has presented an overview of: current mobile networks, the new traded handoff concept, and the usefulness of application migration in a mobile environment. The remainder of this dissertation is as follows.

Chapter 2 describes the distributed computing and operating systems context in which this dissertation should be placed. Previous work in the area of mobile computing is discussed in Chapter 3 and further motivates the research described in this dissertation.

Chapter 4 describes the components of the Notus architecture and their interworking to support traded handoffs and application migration. Chapter 5 describes an implementation of the Notus architecture.

The Notus implementation is evaluated in Chapter 6, using a number of interactive applications which are representative of those expected to operate in a walkstation environment. The evaluation pays particular attention to the disruption experienced by the applications and exercises all the consistency options provided by the implementation for a multi-threaded application.

Related work is presented in Chapter 7. Finally, Chapter 8 describes possible further work and future directions for the Notus architecture.

Chapter 2

Research Context

2.1 Introduction

This chapter provides some historical Distributed Programming Environment (DPE) background, including a discussion on trading and binding, which is relevant to the design of the Notus architecture. The remainder of the chapter then provides an overview of two recent operating systems, Spring and Nemesis. Some functionality from both was used in the implementation of the Notus architecture.

2.2 Distributed Programming

The Cambridge Distributed Computing System is an early example of a DPE, where the distributed services included print servers, authorisation servers, and compute servers, together with a name server to map service names onto their locations [Needham82]. In this environment of *clients* and *servers*, the means of interaction between the clients and servers becomes important. Typically a client *requests* some service from a server, which *responds* with a result. Examples of requests and responses might be a request for a particular disc block or to authorise a user to use a particular service. The responses might be respectively: the disc block and an encrypted session key.

One consideration is whether or not the client should wait for the response or continue to execute in the meantime. It has been shown that given an environment containing asynchronous threads of execution, the two models are equivalent [Lauer78, Liskov85] although the synchronous case has gained greater acceptance as it is viewed as being easier to program correctly. Another consideration is that of the coding used for the information transmitted between the client and server. There must be a common standard for the process of converting application level data structures into a flat byte representation. This function is called *marshalling.*

One synchronous distributed programming paradigm which has gained almost universal acceptance is the Remote Procedure Call (RPC) [Birrell84]. This hides the complexity of marshalling and the concurrency issues of communication through the language level procedure call mechanism. A *remote procedure* is a procedure which exists outside the linkage scope of the client program, but which is invoked by the client in a similar manner to a *local procedure.* A client's view of an invocation of a remote procedure is semantically the same as invoking a local procedure. It passes arguments and execution to the procedure, which later returns with results. However, on an invocation of a remote procedure, execution is passed to the RPC system. This uses *stubs* which are generated from a description of the procedure to marshal the procedure call arguments into a machine independent flat byte representation. At the server-side, the arguments are unmarshalled by the RPC system and the remote procedure invoked. Results and exceptions are marshalled and returned to the client as if from a local procedure call. Standard representations for marshalled datatypes such as Sun XDR [Sun87] and ASN.1 [ISO95a] enable clients and servers to execute on heterogeneous machines and even be implemented in different programming languages.

To some extent the RPC paradigm provides transparency between the local and remote cases, although ultimately the failure properties of locally and remotely executed procedures must differ. For this reason, some RPC implementations, such as [Hamilton84] make the differences between a local and remote procedure explicit by the use of different calling conventions.

Any DPE must tackle the issue of naming and locating the services required by the client. Using the RPC paradigm as an example, a remote service consists of a group of related remote procedures. When the remote server which supports the service initialises, it registers itself using a name service, providing sufficient information for a client to initiate communication with the server. Early examples of these name servers simply identified each service with a unique name [Needham82, Birrell82, Sun88]. Later it became possible for a client to specify its service requirements in terms of the *properties* it required using a *constraint language.* This enabled the name server to match from a number of offers of service, returning a set of suitable candidates for the client to choose from. Such a name service has been termed a trader and is described further in Section 2.3.

One non-transparency in many RPC implementations is the requirement that a client supply an RPC binding identifier as an argument to each remote procedure invocation, enabling the RPC service to identify the server with which to communicate. (See Section 2.4 for a discussion of binding.) With the application of the RPC paradigm to object oriented languages [Black88, ANSA92, Evers93] it became possible to hide the binding behind a *proxy* object. Such an object has a suite of methods of the same type as the remote object which it represents, but on an invocation calls into the RPC library. The state contained within a proxy is roughly the equivalent to that of a traditional RPC binding.

This technology has matured into standards, such as the Open Software Foundation's Distributed Computing Environment (OSF-DCE) [OSF91], the International Standards Organisation's Reference Model for Open Distributed Processing (RM-ODP) [ISO95b], or the Object Management Group's Common Object Request Broker (OMG-CORBA) [OMG95].

These standards define services such as an RPC subsystem, a trader with constraint language and a stub compiler which is used to generated stubs from descriptions of services. A DPE today allows an application to dynamically lookup offers of service based upon a specified criteria and provides all the mechanisms required to connect to, and use these services. The environment is able to inform the application of communication failures and may allow the application to negotiate Quality of Service (QoS) for its bindings [Friday96].

2.3 Trading

The representation of the properties of a service and the communication of these properties to potential clients, can be achieved through the adoption of a trading model.

An *interface* is an instance of an abstract data type. It describes the procedure signatures, data types and exceptions which a server exporting the interface implements. Interfaces are often described using an Interface Definition Language (IDL) [ANSA92, OMG95] and are useful for ensuring a modular structure and providing type safety for distributed systems, programming languages [Cardelli88] and recently operating systems [Hamilton93a, Roscoe95b]. An *interface reference* is a piece of information which enables a client to locate and establish communication with its corresponding interface. Within a single address space, this information contains a pointer; in a distributed environment, it contains the transport level address of an RPC server. Other information which might be contained within an interface reference includes the type of the interface, QoS information, and the marshalling format required by the server.

The *properties* of a particular instance of an interface are described using expressions in a grammar which should be defined together with the IDL. Hence an *offer* consists of a property list and an interface reference, placed in a particular *context*. A client should, using a *constraint language*, be able to pose a query concerning the properties of a given interface.

There is a requirement for a forum where offers can be published by servers and where clients can issue requests for offers. The result of a successful request should be that the client is able to select an interface reference from a list of matched offers and use it to contact the interface instance (Figure 2.1). It is common to call such a forum a *trader* [ISO94]. As well as supporting a namespace for *offers*, together with a means of matching requests with offers, it is usual for a trader to support the notion of the *federation* of a portion of another trader's namespace. After federation, the federated part of the namespace is accessible by both traders. This is the means by which offers from one machine become visible over an entire network.

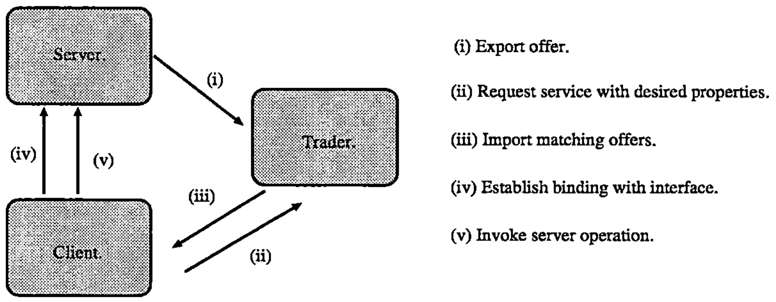

Server.
(i)
Trader.
(iv)
(v)
(iii)
(ii)
Client.
(i) Export offer.
(ii) Request service with desired properties.
(iii) Import matching offers.
(iv) Establish binding with interface.
(v) Invoke server operation.

Figure 2.1: Trading for Services

When a client application has selected an offer from the set returned by the trader, it must use the information contained within the interface reference to establish communication with the server interface. This requires the creation of an *invocation reference*, which is an object containing all the mechanisms necessary for communication with the interface. An invocation reference appears to the client as a proxy to a local instance of the interface.

2.4 Binding

Binding [Saltzer79] is the action of associating a name with a value. In the context of distributed programming, binding refers to the association of an interface with an invocation reference. The *establishment* of a binding occurs when all state required for communication with the interface has been fully created.

There are tradeoffs in making this establishment transparent to the client. If binding establishment is transparent, also known as *implicit binding*, then all information concerning the binding process is hidden from the client, with the illusion that the binding is always available for use. As well as freeing the client from considering establishment, implicit binding allows for dynamic resource allocation. For example, bindings can be established lazily or multiplexed over a single channel. Implicit binding has been implemented in many distributed systems [Birrell93, Hamilton93a, OMG95]. It does however suffer from the disadvantage that a client has no control over the establishment process. This is required in cases such as multimedia applications, where explicit QoS negotiation is required, or for mobile aware applications which require explicit control over their bindings as their environment changes. Conversely, non-transparent establishment, also known as *explicit binding*, requires the client to establish communication whenever necessary. This allows the client to remain in full control of the binding process, but requires additional implementation effort and requires that the distributed nature of the application is visible to the programmer. Explicit binding is supported by some existing distributed programming environments [Roscoe95b, Otway95] and is assumed in later discussion.

2.5 The Spring Operating System

The Spring operating system [Hamilton93a] is a recent example of a microkernel operating system, where system services are placed in separate protection domains, communicating with each other and applications via a small kernel. Other examples include Mach [Accetta86] and Chorus [Rozier89]. The advantages of this system structure are that the system as a whole becomes more tolerant to the failure of individual services, and is easier to extend and reconfigure than traditional monolithic kernels, such as UNIX [Bach86]. However, the cost of communication between clients and system servers becomes a major issue.

Spring addresses the cost of communication between protection domains through the use of an efficient cross domain, object invocation mechanism termed *doors*. A door is a piece of protected kernel state which describes a particular entry point, typically corresponding to a server object in another domain. As used in the Spring inter-domain RPC protocol, a client in one domain issues a door invocation for a server object in another domain. This causes the kernel to allocate and transfer control to a thread in the server domain, passing information associated with the door invocation and argument data. On return, the kernel deactivates the server thread and reactivates the client with either return data or exceptions raised.

While Spring and other work such as [Bershad89, Yarvin93, Liedtke93] have successfully addressed the communication costs between clients and servers in a micro-kernel operating system, all micro-kernel architectures depend heavily on the use of shared system servers executing tasks on behalf of many clients. Operating system services are multiplexed at a high level and this can lead to applications suffering from an effect which has been termed *QoS crosstalk*, where a system has difficulty in providing QoS guarantees and accounting for the resources consumed by a particular application [Roscoe95b].

The Spring RPC service contains an interesting feature which enables new behaviours to be added to RPC bindings, termed a *subcontract* [Hamilton93b]. In Spring, a subcontract is a module (some code) which is given control of the mechanism for object invocation after marshalling has taken place. This enables different invocation behaviours to be implemented. For example, a subcontract might encrypt marshalled invocations between a client and server. Other subcontracts have implemented behaviours including caching and replication. New subcontracts can be introduced to the system without modification to the base RPC service, and if a server receives an invocation from a client using an unexpected subcontract, it is able to consult a registry to find an appropriate subcontract.

Client and server-side subcontracts are not required to be identical so long as they are compatible. Spring defines subcontract A to be *compatible* with subcontract B if the marshalling code for subcontract B can cope with receiving an object from subcontract A. Although subcontracts are typed in the Spring object oriented type system, the compatibility relation between subcontracts cannot be deduced from their type. This results in ad-hoc type comparisons within the subcontract at the time invocations are made.

The Spring work on subcontracts has influenced the design of CORBA Object Adaptors [OMG95]. These differ from the Spring subcontracts in that there is no scope for application writers to create new object adaptors or for the dynamic selection of object adaptors during binding establishment.

2.6 The Nemesis Operating System

The Pegasus project [Mullender92] is a joint effort between the Computer Laboratory and the University of Twente, and has resulted in the design of the Nemesis operating system which supports multimedia applications, providing QoS guarantees and resource accountability.

This is achieved through the multiplexing of resources at the lowest levels of the system, eliminating shared servers as far as possible, the adoption of a single address space, and the use of a scheduler which provides QoS guarantees.

Applications in the Nemesis environment consist of a number of modules, which are units of loadable code, containing no unresolved references and no mutable data [Evers94]. To use the code within a module, an application must first locate (possibly dynamically) an interface for the module. As with the Spring operating system discussed in the previous section, the structure of modules with strongly defined interfaces is maintained at all levels of the system. Interface types are defined using Middl IDL [Roscoe94] which was developed from ANSA IDL [ANSA92] with the notable addition of constructs for local machine and low level operating system interfaces. The Middl stub compiler is used to translate the module type definition into a programming language template for a concrete implementation and to generate marshalling code for the data types defined, arguments to operations, and exceptions.

In Nemesis, a module is instantiated through binding state to a *closure*, where a closure consists of a pair of pointers: one to a module's methods, and the other to per-instance state. When instantiating a module, an application either binds state which is known from the module's IDL or requests that the module bind its own state by invoking an initialisation method of the module. In the former case, the state of a module is *explicit*, its type being defined in the module's interface. In the latter, the type of this *implicit* state is visible only within the module.

At the start of this investigation, the Nemesis implementation was incomplete, however the type system, linkage structure, namespace, IDL, and stub compiler were all well developed [Hyden94, Black95, Roscoe95b]. The type system provides a primitive form of dynamic typing, with operations such as *IsType*, which determines whether a given type conforms to a particular type, and *Narrow*, which converts one type to another specified type, so long as the conversion is permitted.

These features, together with the uniform namespace and linkage model, were sufficiently powerful to support the *Clanger* interpreted programming language [Roscoe95a]. Subject to authorisation, a Clanger script is able to bind to an interface exported by any system module and invoke operations. This should be compared to a similar interpreter for the Spring operating system, which is forced by the lack of a dynamic type system to parse IDL descriptions on the fly in order to determine the type of objects.

Chapter 3

Background

This chapter presents a survey of previous work in the area of wireless network architectures and the applications which are expected to use them. The discussion aims to justify the claims of Chapter 1. First, that there are difficulties in providing transparency in a mobile network – not least in that new mobile aware applications do not require this transparency; second, that previous research has not produced a solution which satisfies all the requirements for application migration in a mobile environment.

3.1 Transparent Mobile Networks

This section describes mobility considerations for a number of different Wireless LAN (WLAN) classes, followed by a discussion of Mobile Radio Network (MRN) developments. The different approaches to wireless connectivity have advanced independently, and interoperability is likely to become a major issue in the future.

One classification of network architectures differentiates between *connectionless* and *connection oriented* networks, with the former perhaps most commonly associated with the Internet Protocol (IP) [Postel81a], and the latter currently receiving great attention with Asynchronous Transfer Mode (ATM) networking.

Development of ATM has continued over many years [Hopper78, Fraser93] and is now accepted as a solution in environments where networks are expected to carry audio and video, along with other forms of data. The ATM compromise enables the network to provide Quality of Service (QoS) guarantees for connections where timeliness is important, whilst at the same time retaining the statistical multiplexing characteristics of packet switched networks. Large numbers of IP based networks currently exist and their number is rapidly increasing as the commercialisation of the Internet continues apace.

WLAN implementations have consequently tended to interwork with either IP or ATM. Those based around IP have the advantage that IP processing on reception is able to accommodate the out of order packets which might result from mobile handoffs. This reduces the handoff problem to one of re-routeing. ATM based implementations require the additional synchronisation between the old and new connections during a handoff, and consideration to the problem of preventing lost or out of order cells.

For the following discussion, it is necessary to distinguish between the *home*, which is the area where a walkstation is normally connected to the network, and the *local* area, which is the proximity to the walkstation, wherever it happens to be.

3.1.1 Mobile IP

The Columbia **Mobile-IP** scheme [Ioannidis91] is designed around two fundamental assumptions: that protocols at or above the transport layer should continue to operate as if they were on ordinary static hosts, and that the addition of mobility should require no changes to the software of non-participating hosts or gateways.

The protocol associates a single IP address with each walkstation, regardless of where it is on the network. Ancillary compute servers known as Mobile Support Stations (MSS) are responsible for ensuring that packets are correctly routed. The scheme defines a *campus* as a small number of cooperating MSS and might also be considered a single administrative domain. Within the same wireless cell (using the same MSS), walkstations are able to communicate directly using a modified address resolution protocol, and the MSS acts as a gateway when routeing packets between the walkstation and hosts on the static network.

A host wishing to contact a walkstation will send packets to the walkstation's home MSS. This will then determine the MSS which is local to the walkstation. Packets are then tunnelled to the walkstation via this MSS. Here, tunnelling refers to the encapsulation of a packet addressed to a walkstation within another IP packet addressed to the walkstation's local MSS. On receipt by this MSS, the encapsulating packet is stripped and the original packet forwarded to the walkstation.

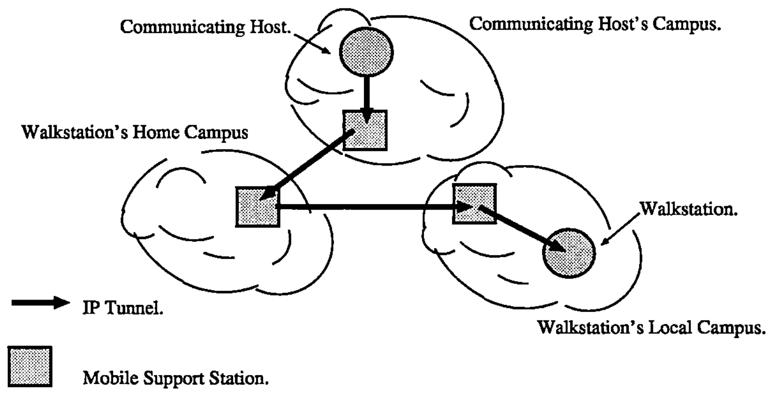

Communicating Host.
Communicating Host's Campus.
Walkstation's Home Campus
Walkstation.
IP Tunnel.
Walkstation's Local Campus.
Mobile Support Station.

Figure 3.1: Mobile-IP Tunnelling

Problems with this scheme occur when a walkstation moves out of its home campus. A walkstation entering a new campus receives a transient IP address for communication with its local MSS and the walkstation must inform its home MSS of its new location. Packets are forwarded from the home MSS, to the local MSS, and on to the walkstation. This results in communication often following sub-optimal routes. For example, Figure 3.1 shows a walkstation which has been carried over a large distance. All packets are sent to the walkstation via its home MSS instead of a more direct route.

The Columbia scheme is compatible with standard IP, relying on encapsulation to transfer data between support stations, and the Internet Draft [Perkins96] is set to be adopted as an Internet standard. Other IP based schemes, such as [Teraoka93] rely on unusual IP options, with the associated danger that these might not be dealt with correctly by all routers in the network.

The efficient provision of inter-campus mobility has been addressed in [Aziz94]. If a static host wishes to initiate an inter-campus communication with a walkstation, two IP tunnels are set up, one leading to a Mobile Support Border Router (MSBR) at the edge of the walkstation's local campus, the other from the edge MSBR to the walkstation's local MSS, and so to the walkstation over a wireless link. When the walkstation changes location within the same campus, re-routeing is only required for the second tunnel.

The case that the connection originates from somewhere other than the walkstation's home campus is illustrated in Figure 3.2. The first packet (i) reaches the walkstation's local MSBR after being tunnelled via the walkstation's home MSS. The local MSBR sends a *redirect* message (ii) to the originator's MSBR, causing a tunnel to be established directly between the originator and the local MSBR (iii) thus removing the home MSBR from the data path.

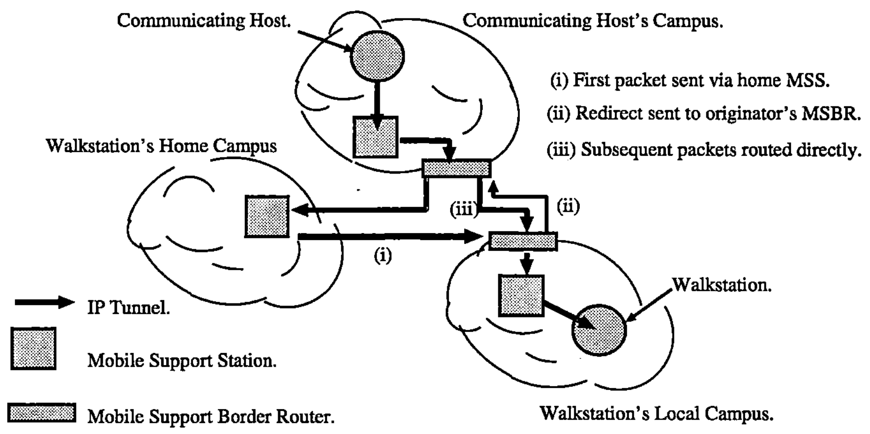

Communicating Host.
Communicating Host's Campus.
(i) First packet sent via home MSS.
(ii) Redirect sent to originator's MSBR.
(iii) Subsequent packets routed directly.
Walkstation's Home Campus
(iii)
(ii)
(i)
Walkstation.
IP Tunnel.
Mobile Support Station.
Mobile Support Border Router.
Walkstation's Local Campus.

Figure 3.2: Inter-Campus Routeing

By redirecting these tunnels in a hierarchical manner during a handoff, the involvement of routers within the network is minimised, and the optimal route to the walkstation is maintained. A similar scheme has been described [Johnson94] where the new location of a walkstation is propagated between routers along the data path from the walkstation. The drawback with this addition to the Mobile-IP architecture is the requirement for the modification of intermediate routers.

Future versions of the Internet Protocol (IPv6) [Deering95] will update the current addressing scheme[1]. As well as increasing the size of the address space, a generalisation of the existing IP *loose source route* is to be adopted which will enable an address to specify a cluster of hosts. This increased flexibility in addressing can be used to optimise routes to a walkstation. If the address of the cluster of hosts which comprises the local campus is embedded in all outgoing packets from the walkstation, then hosts which are sending to the walkstation can route directly, once a packet has been received from the walkstation. However, initial contact with the walkstation must still be made using an indirection through the walkstation's home campus.

There is one final limitation with the described Mobile-IP schemes. Packets sent *to* the walkstation are encapsulated and tunnelled via the home MSS, while packets sent *from* the walkstation are simply launched using the walkstation's fixed IP address as the source address. It is likely that such packets are regarded as invalid by intermediate security conscious routers, and dropped. This problem has previously been addressed, with the home MSS used to forward encapsulated packets both to and from the walkstation [Baker96, Cheshire96]. However the scheme might still fail in circumstances where the connection is

[1]A general overview of the IPv6 addressing scheme is described in [Francis94].

required to pass through an intermediate security firewall which is not prepared to accept encapsulated packets.

The authors of [Cheshire96] advocate the use in different environments of a number of different options for the encapsulation and indirection of both incoming and outgoing packets. They also stress the importance of a walkstation having the option of no Mobile-IP support, that is sending and receiving unencapsulated packets using a temporarily allocated address. This option was thought useful for applications which make use of short lived connections or possess a higher level recovery mechanism.

3.1.2 Mobile TCP

One of the main applications over Mobile-IP is the Transmission Control Protocol (TCP) [Postel81b]. If unmodified TCP is used in a wireless environment there are a number of performance problems. The most serious is that TCP treats packet loss as an indication of network congestion. Once packet loss is detected, a TCP sender will retransmit using an exponential back-off and also reduces subsequent throughput, typically through initiation of the *slow start* algorithm [Jacobson88]. Over a wireless link there is a relatively high error rate compared with the wired network, and packets lost here, should be retransmitted without back-off or reduction in the sender's rate of transmission.

With this problem in mind, **I-TCP** [Bakre94] interworks a wireless transport protocol between a walkstation and base station, with unmodified TCP on the static network. The protocol is designed so that packet loss on the wireless link does not invoke the end-to-end congestion control mechanisms of TCP. Instead, retransmissions are made from the base station, using a retry strategy which may be tuned to the wireless link. The authors have reported improved throughput over unmodified TCP. Other schemes for improving transport layer throughput include [Balakrishnan95a] which is similar to I-TCP, except that a single end-to-end TCP connection is maintained.

Once the transport layer becomes aware of the mobile nature of the network and particularly in the case of I-TCP, where the connection is split and buffered at the base station, a potentially large amount of connection state can be held at the base station. A transparent handoff would require that this state be transferred between base stations. In the I-TCP implementation, handoffs require cooperation between two MSS, transferring the state of I-TCP connections from one to the other. The amount of state transferred depends upon the amount of data buffered; for idle connections the implementation [Bakre95a] requires 265ms to handoff. Most of this time is spent to synchronise the walkstation and base stations, but about 50ms is required to transfer state for each idle connection. This time dramatically increases as more data is buffered at the base station, from about 300ms when 4Kbytes are buffered, to over one second when 32Kbytes are buffered.

One other option which avoids modifications to TCP is to hide non-congestion related errors from the TCP sender, in effect making the lossy wireless link appear as a higher quality link with a lower bandwidth. Implementations [Ayanoglu95, Balakrishnan95b, Balakrishnan96] typically cache packets and perform local retransmissions over the wireless link. Use is made of TCP acknowledgements, and a retransmission timeout is used which is shorter than that used by TCP. It is possible to consider the data cached by these reliable link layers as a hint and not copied over to the new base station during a handoff. Although this enables the reported handoff time to be significantly reduced, the TCP sender is still required to retransmit any data which was buffered on the path to the old base station, causing considerable disruption to the end application.

3.1.3 Mobile ATM

ATM networks are connection oriented, with communication taking place over *virtual circuits*. Data is transmitted in small cells[2], each with a header containing amongst other information, a Virtual Circuit Identifier (VCI). During connection establishment, a virtual circuit is established from one host to another using a number of intermediate switches. Each hop on the data path in both directions is allocated a VCI. During data transmission, when a switch on the transmission path receives a cell for forwarding, it must insert the correct VCI into the cell's header for the next hop and transmit the cell on the correct output port.

Figure 3.3 illustrates a mobile handoff in an ATM network. The handoff protocol must first establish a route from the walkstation's base station to a switch which intersects the old route (i). This switch is termed the Fixed Switching Point (FSP). A new connection is made from the FSP to the walkstation via the new base station (ii). Once the new connection is established, the handoff is performed. This requires a change in the entry of the VCI table of the FSP, so that cells will subsequently be forwarded on the new data path. The connection from the FSP to the old base station is then closed down (iii).

The above mechanism is at the heart of most recent ATM based mobile handoff implementations [Porter94, Rajagopalan95, Condon95]. The difficulties

[2]The current ATM standard defines a 53 octet cell, containing 48 octets of data and 4 octets of header information.

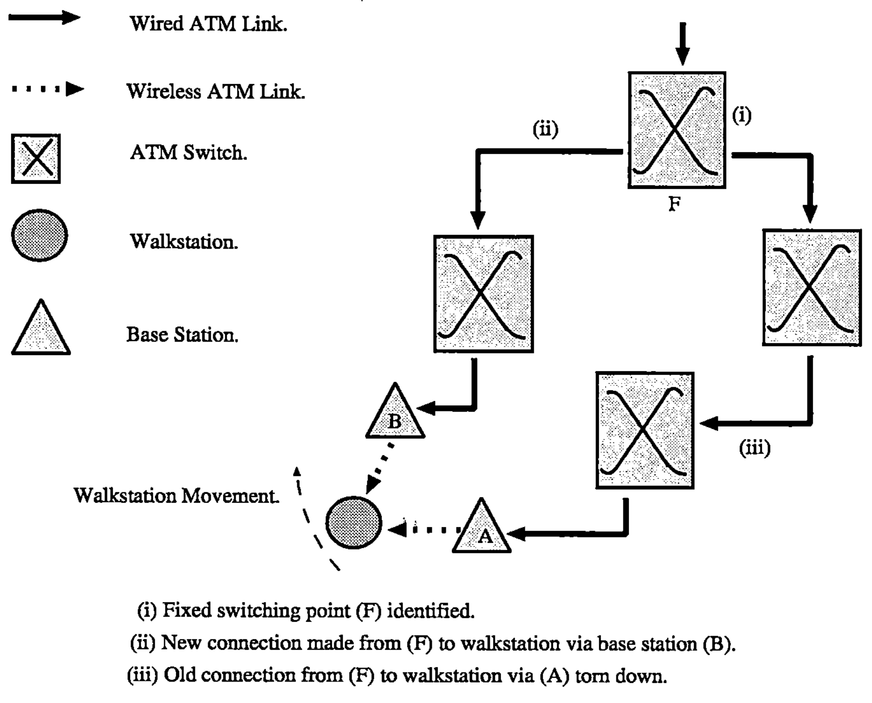

Wired ATM Link.
Wireless ATM Link.
ATM Switch.
Walkstation.
Base Station.
(i)
(ii)
(iii)
F
B
A
Walkstation Movement.
(i) Fixed switching point (F) identified.
(ii) New connection made from (F) to walkstation via base station (B).
(iii) Old connection from (F) to walkstation via (A) torn down.

Figure 3.3: A Mobile ATM Handoff

in performing the mobile handoff are in:

- synchronising between the walkstation and the two data paths from the FSP, so that cells are not lost or received out of order,
- ensuring that routeing updates resulting from mobile handoffs scale in the network, and
- locating a switch which is common to the two data paths and which is prepared to perform the mobile handoff.

A slightly different approach to the problem of synchronisation in an environment which does not require continuous connectivity has been considered [Condon95]. Here, any possibility of cell misorder is removed by completely destroying the old connection from the FSP to the walkstation before establishing a new connection. The large disruption resulting from this method is acceptable only because the environment does not require continuous connectivity.

The disruption to applications during an ATM handoff from experimental evaluation is not currently available. However, [Keeton93] describes an analytical model of a number of handoff algorithms, reporting a best disruption to the walkstation of about 100ms. Also, a simulation of ATM handoffs has been made [Toh95] with handoffs performed by VCI remapping within the Fairisle switch [Leslie94]. The author reports disruptions to the walkstation of about 7ms and also compares this method with the earlier work of [Biswas94] where handovers and cell forwarding are performed by a *redirection module* at the transport layer. Cell forwarding at a high level in the protocol stack not only affects performance in terms of bandwidth, but also changes the temporal characteristics of the network traffic.

3.1.4 Mobile Radio Networks

The existing MRN infra-structure has primarily evolved for the purpose of Mobile Telephony. Most network implementations, such as those described in [Lambley84, Goodman91] solve the problem of a limited bandwidth for a large number of users in the wide area by splitting the area of coverage into small cells, so that the frequencies used in adjoining cells do not overlap. This enables frequencies to be reused in distant cells and also reduces the transmission power requirements. For these reasons, the cellular network concept has also been adopted by indoor WLAN implementations.

Each cell is generally allocated a fixed bandwidth, giving a fixed maximum number of simultaneous connections at each base station. Hence, great care is taken to ensure that cells are sufficiently small to cope with the expected demand. The positioning of cells takes into account radio propagation effects caused by the natural topology and structures such as buildings.

The first systems to evolve tended to be incompatible with each other, but moves have been taken to reach an international standard, notably with the introduction of **GSM** [Rahnema93]. Recent digital systems have started to offer data channels to subscribers. GSM, for example offers data channels of various bandwidths up to 9.6Kbit/s.

Using GSM as a typical example, the location of each mobile user is tracked by the base stations periodically broadcasting interrogation messages. A time division multiplexing scheme is used to arbitrate the wireless accesses by different mobile units in the same cell. During the period when a mobile unit is not transmitting or receiving data, it is able to examine the signal strengths from other base stations using a broadcast channel specially allocated for this purpose. The mobile unit initiates a handoff by transmitting the average signal strengths for each of the surrounding base stations to its current base station. This determines whether a more appropriate base station is available to accept the call. The handoff itself can be executed anywhere in the network hierarchy. For example, locally between base stations which share a base station controller, between regional mobile switching centres, or between national network centres. The actual handoff might take up to 320ms, although much of the work is performed in advance and the disruption to the end user is considerably reduced.

A walkstation with both MRN and WLAN interfaces might, when operating in an MRN environment, wish to choose between different network operators in order to minimise tariffs or to ensure good coverage. Applications running on the walkstation might also wish to take advantage of additional services provided by the different network operators. For example, some operators already provide mail facilities. These are currently much cheaper to use than the cost of a call to an Internet service provider or some other gateway computer. As the user moves into a WLAN environment, applications which were configured to use these additional services would be expected to adapt to use services provided in the new environment.

3.2 Mobile Applications

The main advantage of a network which provides transparency is that it enables a walkstation to run all the applications which were previously used in a static environment without modification. This section describes a number of applications and application frameworks have been implemented, which adapt in different ways to a changing mobile environment. All require information to be available which concerns their changing environment.

The types of applications which are expected to be used on walkstations are next considered:

- **Database** queries over the static network for information such as weather, or traffic conditions, and performing share transactions, or home shopping.
- **Client–server** applications, such as World Wide Web (WWW) browsing, electronic mail, Usenet news, and remote sessions on static computers.
- **Multimedia** applications, such as a video phone, television broadcasts, video mail, and video on demand (the first two applications imply "live" sources of data).
- **Collaborative working**, requiring a group protocol for distributed transactions and floor control.

While some of the simpler applications, such as electronic mail might operate effectively in a transparent environment, the more ambitious applications would be expected to adapt to such changes as network bandwidth, connectivity, and the proximity of useful resources.

The **MOST** project [Davies94b, Friday96] has centred on a cooperative application for field engineers in the electricity supply industry. This application requires the use of mobile, peer-to-peer communication paradigms and emphasises operator safety. The project has made extensions to the ANSA/REX communications protocol [ANSA92] to introduce QoS managed bindings whereby call-backs are made to applications if QoS constraints are not met. The communications protocol which was implemented (QEX) has also been used to investigate back-off and fragmentation strategies pertinent to mobile computing. The MOST environment makes use of replicated services and different network types, but requires the selection of services to be made by the user of the application. The additional flexibility of the traded handoff might have been beneficial in automating this process.

The **Bayou Architecture** [Demers94] consists of variable consistency, replicated databases in a mobile environment, and is intended for collaborative applications which require read and write access to shared data. Walkstations are able to interact with any available database server, taking into account locality and network performance. For Bayou clients, this offers similar advantages to applications using a traded handoff.

IP Fast Fail [Montenegro95] recognises that applications should be informed of disconnections from the network. This enables applications to continue to function in a diminished capacity, instead of blocking as network operations fail. The implementation takes the form of a daemon which monitors for disconnections of the host from the network. On a disconnection, the daemon configures the system so that the network driver returns an error whenever transmission is made on a non-loopback interface. A number of unmodified applications are demonstrated which immediately fail rather than hang when the system is disconnected. Other applications are introduced (a mail server and a file system) which are able to continue using their own caching strategy on disconnection.

The **Odyssey Architecture** [Noble95] allows applications to register an interest in available resources. Call-backs are made to the application when these resources change and are used by the application to change its *data fidelity*. For example, a video server may support the *movie* abstraction, which consists of a number of different copies of the same video at different levels of fidelity. The video client requests the highest fidelity stream which can be played out given the available bandwidth. On receipt of call-backs indicating a reduced available bandwidth, the client requests that the video stream changes to a lower fidelity.

A mechanism for walkstation access to file systems has been implemented which allows a walkstation to **Hot Replace** its read-only file systems [Zadok93]. It is argued that this is desirable even if the underlying network provides transparent connectivity, since both the latency and reliability of operations worsen as the walkstation moves away from the file system server. The work is based around an extension to the Sun Network File System (NFS) [Sandberg85] and because of its stateless server model, the implementation is able to switch over open (read-only) files to a new server without any client state transfer. A method has been described of supporting a read/write, replicated file service [Tait92]. Walkstations are assigned a *primary* file server, which propagates updates to a quorum of *secondary* file servers. The walkstation caches its requests until it has received an acknowledgement that the request has been safely propagated to the secondary servers. This caching makes it possible for the walkstation to switch to a new *primary* without the transfer of any state. A special module called the *matchmaker* is used to locate a suitable primary as a walkstation is moved.

It has been noted that migration of a compute desk-top is an aid to cooperative working. In the X-Windows **Teleporting** environment [Richardson93] the windows which comprise a user's desk-top can be *Teleported* from one workstation to another. This is achieved through a proxy window server which acts as a client to the real window server on the remote workstation. Using this arrangement, window updates from Teleported applications are passed via the proxy to the remote workstation. The Teleporting environment has the attraction that applications and servers do not require modification and are unaware of mobility. However, high-bandwidth applications suffer from the indirection through the proxy server. Such applications would benefit from being restarted on the same host as the real window server and are candidates for a low-latency migration mechanism.

Finally, work in the area of **Context Aware** applications has investigated how the behaviour of applications should adapt to changes in a context which represents their location and nearby resources. For example, an alarm application might be configured to respond in a different manner when the user is in a meeting, compared with when the user is alone or with co-workers. Other examples of context aware computing include an investigation into the control of peripherals depending upon the individual's location [Want92a, Want92b] and the use of a walkstation with knowledge of its physical location to attach virtual reminder notes to physical objects and locations [Brown95]. Another project, [Schilit93b] presents a dynamic Remote Procedure Call (RPC) service for the Parc-Tab walkstation [Schilit93a] which allows environmental changes to be propagated to applications. Clients subscribe to the RPC service and receive call-backs when their environment changes. Cited uses for the service include: determining the closest printer or finding a suitable display onto which to migrate an application's user interface.

3.3 Application Migration

In Chapter 1, a motivation was provided for application migration in a mobile environment. It outlined the primary requirements for a migration service as being heterogeneity, a low-latency, and the imposition of a low overhead on running programs. Other requirements include:

- the removal of all dependencies on the original host after migration. An application with dependencies, such as a requirement to redirect system calls, requires communication with the original host and becomes more vulnerable to failure as it migrates.
- the implementation of the migration service should not be dependent upon a particular operating system or compiler. The use of a special compiler or operating system would reduce the availability of the migration service over a large number of platforms.

This section describes the significant amount of work which has been carried out in this area and shows that no existing scheme matches all the requirements for application migration in a mobile environment. Much of this work has taken place for reasons of load balancing and is described in its historical context.

3.3.1 Load Balancing

Application migration, also known as *task* or *process* migration, should be distinguished from both *static* and *dynamic* load balancing. These techniques aim to assign computational tasks to computational resources such that the time to process all tasks is minimised. Once assigned to a host, a task runs to completion and may not be migrated to another host. This is based on the assumption that the cost of moving an already started task will outweigh any benefits of migration and that all tasks have similar resource requirements [Eager88].

Static load balancing algorithms [Tantawi85, Agrawal88] attempt to precompute the placement of tasks according to their known resource requirements. This is useful where a system is running a well understood distributed application, but cannot handle a system where new applications may be introduced or where the behaviour of applications is unpredictable. Dynamic load balancing algorithms [Barak85, Eager86, Zhou88, Douglis91b, Zhou93] react to changes in system state when placing tasks on suitable hosts. This requires an exchange of state information. A host with a new task to execute, probes other hosts to determine the most suitable location for the task. These schemes offered improved performances, even when modest heuristics for *transfer* and *location* policies were used [Eager86, Benmohammed94]. The transfer policy determines whether to execute the task locally or remotely, and the location policy determines the host to which a task selected for transfer should be sent.

3.3.2 Process Migration

The motivation for application migration emerged as the use of distributed environments containing workstations used by individuals became widespread. It was noticed that at any one time, a large number of workstations in the environment were idle and could be utilised for other long running or computationally intensive applications, such as simulations. In this environment, the interactive response time of a workstation is an important issue and results in a requirement to evict applications which have been placed on an idle workstation when its interactive user returns. This new requirement of achieving load balancing while respecting the interactive workstation users, promoted application migration as a useful tool. However, the main issue is the speed by which resources are returned to the interactive user, not the latency of the migration process or the disruption to the migrating application.

For this reason, application migration has usually been implemented by migration of the underlying operating system *process* abstraction, enabling unmodified applications to be used. Although process migration has been implemented at various levels in different classes of operating system, none of these implementations have fully addressed the issues of performance, heterogeneity, dependencies on the original host, and portability. As a result, these schemes do not fulfill all the requirements for a mobile environment.

Condor [Litzkow92] was motivated by the need to provide a migration mechanism for long running applications such as simulators. The implementation provides a user-level process migration facility for UNIX. Migration was carried out at a coarse grain (decisions every 10 minutes) and the whole address space was copied. This took a long time (2min over 10Mbit/s Ethernet). A facility called *remote system calls* was provided, where system calls from a migrated process were passed back to the originating machine where objects such as file handles could be resolved correctly.

Spice [Zayas87] implemented process migration within the Accent kernel [Rashid81] using a method of demand paging from the source, as and when pages belonging to the migrated process were later referenced. This method was found to greatly reduce the amount of data which needed to be transferred since many pages were never referenced. However use of the method leaves a dependency on the source for the servicing of page requests, leading to problems of performance and reliability. Chains of these requests were possible for processes which migrated more than once.

Sprite [Douglis91a] implemented transparent process migration in a monolithic kernel using a similar method of demand paging, but with dirty pages from the source machine first copied and later serviced from a network server. This method has the advantage that the source is not required to satisfy page requests, although in a mobile environment, these requests would still be required to be serviced over the wireless network.

V [Theimer85] implemented a process migration scheme within a message based distributed operating system, called *preemptable remote execution.* The implementation used a pre-copying scheme, where most of the pages of the executing process were copied to the target before the migrating application was suspended. This was intended to minimise the disruption to the application, though at the expense of copying some pages more than once. Essentially this scheme reduces the disruption experienced by the application during migration by increasing the amount of time spent in preparation. However, in a mobile environment a walkstation might have little notice that it is about to loose a communication link, hence it is important that its applications are able to migrate with little preparation.

An implementation of process migration in the Mach [Accetta86] operating system has been described [Milojicic93]. This implementation has the potential of also providing migration support for operating systems emulated by Mach. The mechanism used depends heavily on distributed shared memory between hosts and the transparent message passing facilities of Mach, leaving a large dependency on the original host after migration. The implementation also leaves an emulated process abstraction on the original machine and much communication is directed back to the original host.

Finally, a recent effort has been described which improves the disruption experienced by applications during migration [Rouche95]. When migrating, the application's working set of pages is first transmitted to the target, together with enough state to restart the application. Other pages and state then follow. The aim is to quickly restart the application, while preventing it from immediately stalling on page faults. However this scheme still requires large amounts of state to eventually be transferred and as with all the process migration schemes, does not allow process migration between heterogeneous platforms.

3.3.3 Object Migration

Heterogeneous object migration has appeared in some object management systems such as [Olsen92, Davies94a]. Here, the aim is to support the dynamic reconfiguration of the placement of objects for load balancing or in response to changes in the distributed environment. Before migration, these systems require objects to respond to a *passivation* request by suspending all further activity. In [Davies94a] objects are then issued with a *checkpoint* request requiring them to generate and return their state for transfer. This step is automated in [Olsen92] provided that the application programmer has declared the object's persistent state in an Interface Definition Language (IDL). Both these schemes address the problem of heterogeneity by the use of distributed programming techniques for marshalling state in a manner similar to Notus (described in Chapter 4). However, neither support the migration of thread state for active objects.

3.3.4 Parallel Programming

The existence of very large computationally intensive applications, provides a motivation for application partitioning and parallel execution. Both have been achieved over specialised parallel hardware and also in the context of clusters of heterogeneous workstations. In the later case, parallel programming is now supported through de-facto standards such as Parallel Virtual Machine (PVM) [Geist93] which provides message passing facilities and other support for parallel execution on heterogeneous computers. Two programming models which have become accepted are task and data parallelism.

Data-parallel applications execute within the single instruction multiple data paradigm and can be thought of as a single program replicated over a number of compute servers, with all instances executing in lock-step. This enables operations which can be parallelised to be evaluated with different portions of data at the same time. In the context of ensuring reliability for large data-parallel programs, the **Dome** [Beguelin94] heterogeneous migration scheme was implemented over PVM. Applications make use of pre-defined objects which implement data-parallel operations. Additionally, each of these objects contains a method which marshalls its internal state. No transfer of execution state is required because of the model of synchronised execution. The implementation, however, provides no support for applications which use any objects other than those in the pre-defined library.

Task-parallel applications are characterised by their readiness to partition along functional lines and a typical application might be a discrete event simulation of a distributed system. Migration of the parallel tasks from one computer to another has been motivated by load balancing and fault tolerance, and has been achieved by the migration of the underlying operating system abstraction [Casas95]. Unsurprisingly, this form of migration has similar characteristics and shortcomings in a mobile environment to the process migration schemes described earlier.

There is another method for parallel task migration in circumstances where all compute servers execute copies of the same program, with each program copy supporting different active tasks. To achieve migration, the underlying facilities of PVM are used to activate one instance of a task and deactivate another instance. Before activation of the new task, it is necessary to marshall and transfer all the task's state. Such a mechanism has been implemented [Prouty94] and extended [Shum96].

Although techniques for parallel task migration have been refined over many years, parallel programming models and the heavyweight mechanisms for parallelism in a cluster of workstations do not fit the programming model or resource requirements for the interactive applications which are expected to be common on walkstations.

3.4 Summary

This chapter has described a trend towards exposing mobility at higher layers in the network protocol stack. At the network layer, efficient routeing solutions have required the cooperation of intermediate switches (or routers). This leaves unresolved problems of switch conformance to the handoff protocol in existing networks, which have been deployed without mobility considerations.

Even with mobility implemented at the network layer, there is a significant disruption experienced by the walkstation during a handoff, and in circumstances where the walkstation roams over large distances, it has already been described how applications suffer from the increased latency and poor error characteristics of their connections to remote servers.

Mobility was exposed at the transport layer in order to improve throughput. This places the additional burden of transport level state being transferred from one base station to another during a handoff. Since this state might include a large amount of buffered data, transport level handoffs impose a significant disruption on the application. Without the application's knowledge of the semantics of the data being transferred over the network, there can be little optimisation at this level. For many applications, such as those processing multimedia streams, this disruption might result in the data copied during a handoff being regarded by the application as being late and so discarded.

At the same time, applications and application frameworks have appeared which make good use of information concerning their environment. Such applications are aware of mobility and their existence weakens the main argument for transparency, namely that applications should continue to run in a mobile environment without any modification. Where the environment does not provide transparency, legacy code can be supported through libraries which implement a default behaviour.

This motivates an investigation of the traded handoff concept, which was introduced in Chapter 1. By involving applications in the handoff process, good use is made, as the walkstation roams over large distances and administrative boundaries, of replicated servers and different network types or providers, offering different QoS, such as tariffs or bandwidths.

This chapter has also examined a number of existing application migration schemes, with the conclusion that none are entirely suitable for use in a mobile environment. In order that the two main requirements of heterogeneity and a low-latency implementation be met, it was felt that an implementation should concentrate on minimising the state required to be transferred during migration, and ensuring that state is transferred in a heterogeneous manner.

Chapter 4

The Notus Architecture

4.1 Introduction

The Notus architecture was designed to provide support to application developers, enabling the construction of applications which take an active role in both the traded handoff process and migration.

Section 4.2 will outline the various components of the architecture. The approach taken to naming and location, and the assumptions on which the architecture is based are discussed in Sections 4.3 and 4.4.

The chapter then describes, in Sections 4.5 and 4.6, how the components interact with each other and a walkstation's applications, to perform traded handoffs and application migration.

Finally in Sections 4.7 to 4.9, the issues of consistency, garbage collection, and security which are raised by the architecture are addressed.

4.2 Notus Overview

In the Notus architecture, an application consists of a number of objects, termed *modules*. Each module carries out a well-defined task, and may contain asynchronous threads of execution (emphasised using the term *active module*). The type of a module is defined using an Interface Definition Language (IDL), and consists of the signatures of operations and the data types implemented by an instance of the module. For implementation purposes, there exists a mapping from the module type description onto a language level template. The Notus architecture assumes that all applications execute within a context, termed a *domain*, where invocations between modules incur no additional overheads other than the function call conventions.

4.2.1 Components

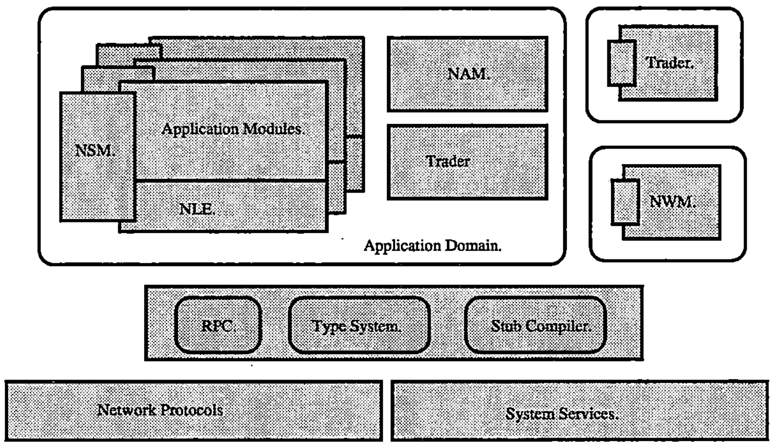


Figure 4.1: The Notus Architecture

The components of the Notus architecture are shown in Figure 4.1. Services are grouped by the degree of shared state and synchronisation which they require, with the aim of reducing interaction with shared servers. Each of these service classes is supported by a different component of the architecture.

- The walkstation runs an instance of the Notus Walkstation Manager (NWM). This supports services which require the orchestration of all the walkstation's applications or mediation with the network, such as the federation of traders through new base stations.

- Every application contains an instance of the Notus Application Manager (NAM). This supports services which require synchronisation within an application, such as the assembly of a consistent checkpoint from a number of independent application modules.

- Each of the application's modules is *associated* with an instance of the Notus Support Module (NSM). The form of this *association* depends upon the implementation environment, for example class inheritance or argument passing. The NSM supports services which are relevant to a single application module, such as the assembly of the module's checkpoint, or the synchronisation of the module's threads during a traded handoff.

- The implementation language environment has its functionality extended through Notus Language Extensions (NLE). For example, to provide facilities for the back-tracing of a module's execution stack when building its checkpoint. It is possible to implement the NLE using compiler extensions or a preprocessing step.

- In order to provide transparency for legacy code or where simple defaults are sufficient to specify an application's behaviour on a handoff (Section 4.5.4), enhanced functionality from the Middleware is required. Similarly, the standard trading functions require enhancements to support the additional consideration of Quality of Service (QoS) over different access routes (Section 4.5.1). It is intended that Notus should use, as far as possible, standard trading and Middleware services, and that any additional functionality in these areas be kept to a minimum.

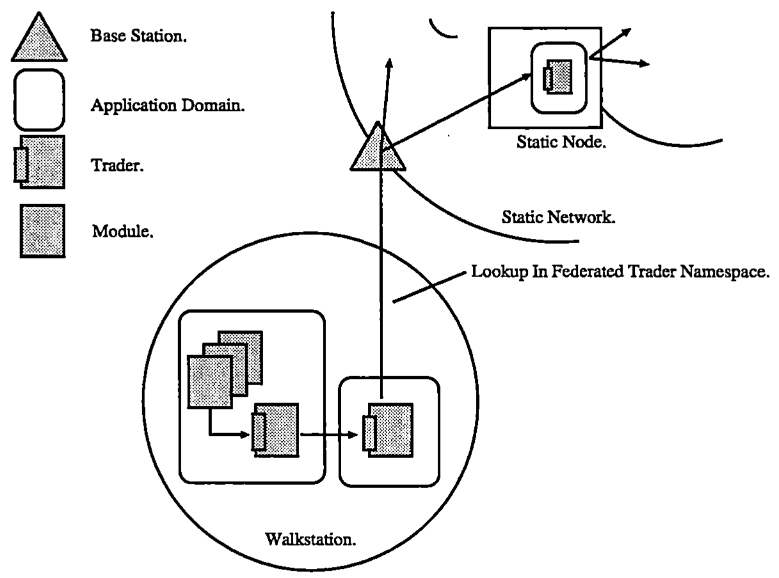


Figure 4.2: Trader Federation

As shown in Figure 4.2, each application has a trader module within its own domain, federated with a trader running on the walkstation. This in turn is federated over the wireless network with (potentially) a number of traders on the static network. A trader within an application's domain allows queries which can be resolved within the scope of the application to be handled without the overhead of inter-domain communication.

4.3 Naming and Location

This section briefly outlines the implications of the adoption of a trading environment on the naming and location of walkstation exported services.

The wireless network implementations described in Chapter 3 allocate a uniquely identifiable *machine-oriented* name to the walkstation. This name is used to route to the *home* location of the walkstation, where an indirection determines a route to the *current* location of the walkstation. The reason for this indirection is that a single name cannot provide hints to the changing location of a walkstation.

The Notus architecture supports walkstation mobility between heterogeneous network types, using the trading environment as the point of indirection between service offers and machine oriented names. If communication is required with a walkstation, and this indirection maps onto a globally unique name, the network must perform a further indirection in order to route to the walkstation. This provides motivation for an argument which favours the network allocating only temporary names to the walkstation at each base station, thus removing the requirement for an additional network level indirection. This approach has been recently adopted [Baker96], and is described in Section 3.1.1.

In the trading environment, services are located using names which are composed of properties and a context. An indirection through the walkstation's home location is required to locate the services exported by its applications. This is achieved in the Notus architecture by the federation of the walkstation's trader with a fixed trader at the home location, thereby enabling a third party to import offers from the walkstation through the home trader.

Hence, a suitable name for locating the services exported by a walkstation is an interface reference to its home trader. A global directory service would only be required to perform a mapping from the user's subscribed name onto this interface reference. There have been similar proposals for global location in a mobile environment, such as [Findlay96] which uses a scheme based on the X.500 directory service [ISO88] to perform a mapping from subscriber directory names onto mobile services. However, the problem is not considered further in this dissertation.

4.4 Assumptions

The Notus architecture makes a number of assumptions concerning the relationship between itself and the underlying wireless network environment. First, there is an assumption that information, such as the status of all of a walkstation's network interfaces, will be provided to the walkstation's NWM as the network environment changes. The means by which this information is communicated are beyond the scope of this dissertation, though call-backs or the probing of interfaces would be suitable.

Second, as a walkstation moves into the area of coverage of a new base station, it is necessary for a *greeting* to take place, so as to register the walkstation with the base station and to ensure that communication is possible. It is assumed that the mechanics of a greeting are handled entirely by the corresponding wireless network protocols. Once a greeting has been made, the network should ensure that the walkstation is able to set up and later tear down communications using the base station. If a walkstation creates a communications endpoint, this endpoint must be named and in such a manner that the name can be used by a third party to initiate communication.

Finally, it is anticipated that some wireless networks will perform handoffs without providing any information to the walkstation. In such cases, a traded handoff can only be initiated based upon the QoS experienced by the walkstation's applications. A working example of such a network has been provided in the context of mobile telephony, where the motivations for transparent handoffs are that the end receivers are intended to be simple devices and because the main application is a telephone call between two people, where communication is always required to be re-established between the original end-points. There are other circumstances where handoffs should be handled transparently, such as when handoffs are expected to occur frequently. This situation might occur in a large gathering of mobile users where bandwidth is shared between a number of base stations. These frequent handoffs are handled efficiently by existing Wireless LAN (WLAN) implementations [Ioannidis91, Johnson94, Porter94] since the scope of re-routeing remains within the local area and a single network instance.

4.5 Traded Handoffs

The Notus architecture supports a number of different traded handoff behaviours suited to mobile client and server applications, which may or may not be aware of the walkstation's mobility, and which are designed to use either stateless or stateful servers.

This section first describes the general scenarios which would cause a traded handoff to be initiated, and the means by which the trader environment is used to make visible the changing properties of services as the walkstation moves. The traded handoff protocol is then described in Section 4.5.3 for a client application which is aware of its mobility and wishes to be involved in the traded handoff. The architectural support for applications which are unaware of mobility and require a default traded handoff mechanism is then described in Section 4.5.4. Finally, in Section 4.5.5, consideration is made for server applications which are running on a walkstation. These are required to redirect their clients as the walkstation moves.

4.5.1 Traded Handoff Initiation

There are two general scenarios which cause a traded handoff to be initiated: the use of a new network type, or the realisation that the movement of the walkstation has resulted in strained connections to servers through the existing network.

New Network Type: Communication is required through a new network type, because either service is (or is expected to be) unavailable through an existing network, or where a new network becomes available which offers better service than the first; for example, when moving into a building supporting a WLAN.

The motivation for performing the handoff in this case results primarily from the wireless network providing information to the NWM. When the NWM decides that communication ought to take place through the new network, it instructs the walkstation's trader to federate over the new wireless network with a default trader. Once the new trader has federated, all offers available through the new network are visible to the walkstation's applications.

Existing Network: A traded handoff can be initiated even where there is no change of network type, but a number of applications are dissatisfied with their services. Thus, an application which has noticed that it is unable to negotiate an acceptable level of QoS with a server, will independently perform a traded handoff, reconnecting to more appropriate services. This activity can be noticed by the NWM, which would then initiate a traded handoff for all the walkstation's applications.

4.5.2 Updating The Trader Namespace

For an application to find more appropriate services during a traded handoff, it must have a means of determining whether such services are available. This corresponds to a requirement for maintaining the trading environment, to reflect the available services as the walkstation moves.

The namespace seen by an application potentially contains the offers from a number of federated traders through different base stations, each corresponding to different network types, providers, or gateways. When trading for offers in this environment, a client may wish to place additional *constraints* on the offer based upon the QoS obtainable; for example: minimising the cost of communication or maximising the available bandwidth.

These constraints do not relate to properties of the service exported by the server, but instead to properties of the service which would be received by an application given the current and expected behaviour of the walkstation. It would therefore be impossible for an offer to contain these properties when exported. Hence QoS constraints are applied by the walkstation's trader at the time a client requests that offers are imported. The walkstation's trader is aware of the context of the request and ensures that these additional properties are eventually present in the offers received by the client.

Consider as an example, a client which issues the constraint that a particular service type must be available with a low-latency and that the cost must be within a given bound. The walkstation's trader would make requests only to other federated traders which are available over links within the given bounds. When the offers are returned to the client for consideration, the walkstation's trader inserts *additional* properties which reflect the cost and latency of each offer, as far as it is able to discern.

What the trader cannot do is negotiate precise QoS parameters for a particular service. These will vary according to the load on the server and the availability of different wireless network types. At the same base station QoS is dependent upon factors such as: congestion on the static network, signal quality, and the number of walkstations simultaneously using a particular base station. An application must negotiate its own QoS at the time a binding is established and re-negotiate as conditions change.

4.5.3 Traded Handoffs for Mobile Clients

Using the pure Remote Procedure Call (RPC) paradigm, the function of the RPC mechanism is to execute a procedure which is remotely exported by a particular server. The extent to which these servers hold state corresponding to a particular client is a tradeoff for application designers. For example, the Sun Network File System (NFS) [Sandberg85] was originally built around a stateless model, with the advantage of a fast recovery from server crashes, but with the disadvantage that client caches must make more coherency calls than would have been the case had the server held enough client state to make cache invalidations. In this case, client state held at the server would potentially have improved the performance of the implementation. Other examples of client state transfers to servers occur during authorisation and authentication activities, and there has also been a trend towards a model of distributed objects placing an emphasis on server objects with state [ANSA92, OMG95].

Stateful server applications can be built using a standard RPC service. After a binding is established with a server, the client then creates a *session*, possibly transferring state to the server in the process. In order that subsequent client invocations are identified with the session, and so with the state, it then becomes necessary to prefix all invocations with an application defined session identifier. This has led some RPC implementations, such as [Hayton96] to maintain an abstraction of a session which might last longer than the lifetime of an established binding. The traded handoff is intended to support applications which use stateful servers, whilst not penalising those that do not.

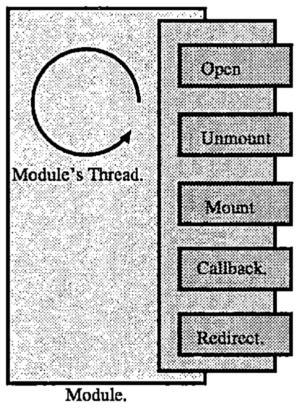


Figure 4.3: The Handoff Interface

A module which is to use the traded handoff mechanism must provide an implementation of the Handoff Interface (HDI), shown in Figure 4.3. The application invokes these methods during startup or on a communication failure, and the NAM invokes the methods to coordinate a mobile handoff. The semantics of an implementation of the HDI are first described, followed by an example which illustrates the traded handoff protocol.

The *Open* method requires the module to create all its bindings to servers. The *Unmount* method requires a module to close down its bindings, and optionally the module should block its threads at one of a set of well-defined *synchronisation points*. This is necessary in order to prevent the module from attempting to use a binding during the handoff process. *Mount* requires the module to initiate a new session with the server, transferring any appropriate client state. At this point, the module's threads (if blocked) continue. *Callback* and *Redirect* are required when a module is acting as a server (see Section 4.5.5).

The separation of *Open* and *Mount* follows from the observation that a binding may be established with a server before a session is formed. It is possible to create a binding in advance of any disruption to the application, and to store the binding as a hint, temporarily in a local repository termed a *stash*. A stash differs from a cache in that coherency of data is not maintained. This was originally termed *quasi-caching* [Alonso90] though later *stashing* by the same authors. The stashing of bindings is especially useful during a mobile handoff, where it is advantageous to perform as much work as possible in advance, minimising the disruption to the application.

The following example shown in Figure 4.4, illustrates how the traded handoff is performed, assuming a module which already has a number of open sessions. The NAM first invokes *Open*, resulting in bindings to new servers being placed in the stash, then *Unmount*, which causes the module to close its existing sessions and block.

If the module's threads do not reach synchronisation points within a short time, for example because one thread is blocked at another point, the initial invocation of *Unmount* made by the NAM should time-out. The NAM may then request that the module's bindings be destroyed by a subsequent invocation of the *Unmount* operation, which specifies no synchronisation with the module. This would typically cause an exception later within the module if the application attempts to use the destroyed bindings.

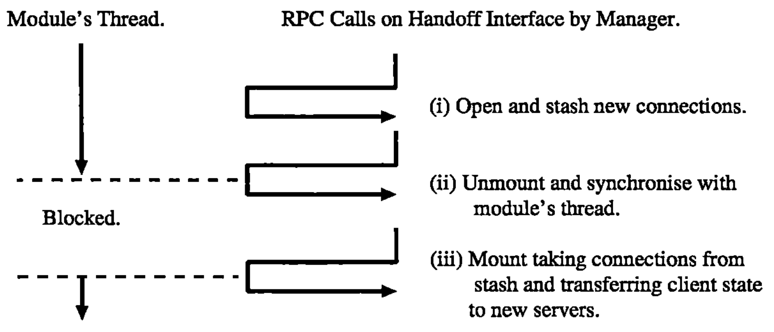

Module's Thread.
RPC Calls on Handoff Interface by Manager.
(i) Open and stash new connections.
(ii) Unmount and synchronise with module's thread.
Blocked.
(iii) Mount taking connections from stash and transferring client state to new servers.

Figure 4.4: Blocking during a Traded Handoff

Once the module has synchronised, the client is instructed by the NAM to *Mount* its new servers. The stashed bindings are used by the client to establish sessions with the new servers[1]. At this point the traded handoff is completed, the client module's threads are unblocked and the application continues normal execution. It should be noticed that during the handoff process, it is only necessary to disrupt the module for the time taken to *Unmount* and *Mount* the remote server.

[1]A stateless server would not require any session establishment.

In cases where the *Unmount* and *Mount* operations require the transfer of much state, then the application should take into account the relative advantages of possible communication with a more local server, over the resumption of communication with the old server. If the latter is chosen, the client application can either use its old *interface reference* to the old server or else specify the old server's unique properties in a trader query. In this case, after binding is re-established, it is important that the server is able to associate the client with its old state.

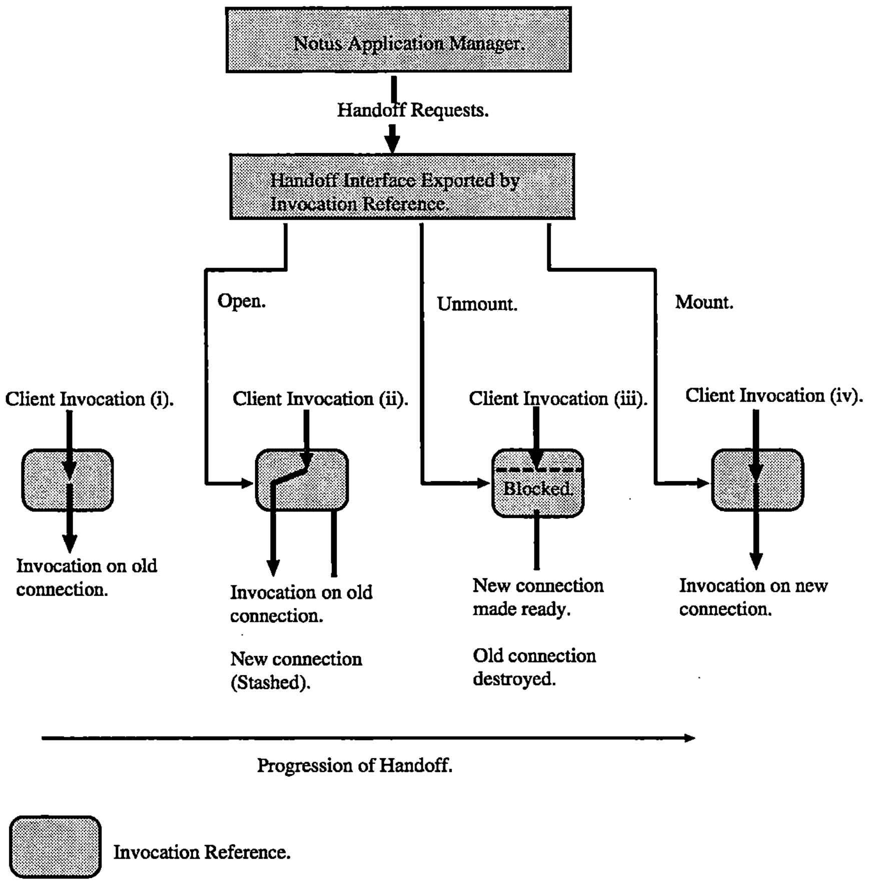
4.5.4 Default Traded Handoff Behaviour

In the above discussion, it has been assumed that an implementation of the HDI be provided where necessary by an application's modules. If the application wishes to use a default behaviour or where the application is unaware of traded handoffs, the invocation references[2] for the application's bindings should provide an implementation of the HDI and directly respond to requests from the NAM.

Applications which are aware of the traded handoff specify a default behaviour of either rebuilding bindings to the original server interface or to another server which matches a given constraint. For applications which are unaware of mobility, the invocation reference should respond by rebuilding bindings only to the original interface. This would enable the use of legacy code in an environment which relies on traded handoffs. In the general case, it would be expected that even within a single application, some bindings would require a default handoff behaviour, while others corresponding to modules which provide their own implementation of the HDI would not.

The default traded handoff is illustrated in Figure 4.5, showing a single invocation reference, together with four separate client invocations made during the handoff. A client invocation (i) made before the handoff takes place, results in a communication over the existing connection to the server. When the invocation reference receives the *Open* request, a new connection is made to either the original or a new server (depending on the specified behaviour), and stashed. At this point, client invocations (ii) are still made on the existing connection to the server. On receipt of the *Unmount* request, outstanding client invocations are awaited for a short period of time, before the old connection to the server is destroyed. New client invocations (iii) made after the *Unmount* request are blocked until the invocation reference receives the *Mount* request. Further client invocations (iv) are then made over the new connection to the server. In the case that a client invocation is made before the *Unmount*,

[2]Appearing to the client as a local instance of the server interface.



Notus Application Manager.
Handoff Requests.
Handoff Interface Exported by Invocation Reference.
Open.
Unmount.
Mount.
Client Invocation (i).
Client Invocation (ii).
Client Invocation (iii).
Client Invocation (iv).
Blocked.
Invocation on old connection.
Invocation on old connection.
New connection (Stashed).
New connection made ready.
Old connection destroyed.
Invocation on new connection.
Progression of Handoff.
Invocation Reference.

Figure 4.5: Default Traded Handoffs

but does not return before the server connection is destroyed, the invocation is retried on the new connection.

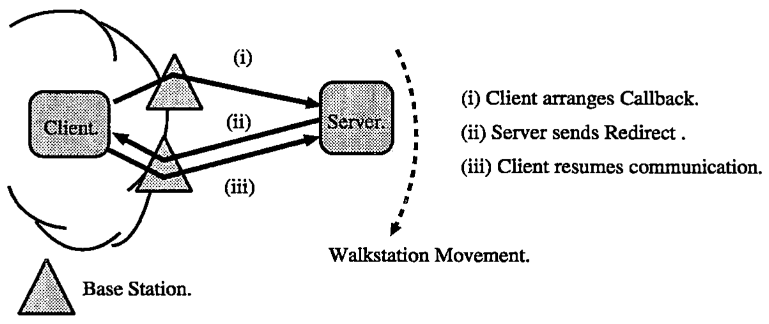
If the default is chosen which supports reconnection to the same server, a retry because of a traded handoff should be recognised by the server and the unacknowledged results retransmitted to the client. However, selecting the default of any server matching a set of properties, might result in repeated invocations being made over different server instances during a handoff. It is important that applications which use this default accept the possibility of this behaviour. This usually implies a requirement for the use of stateless servers by the application.

In both the default cases, there is no state transfer between servers, and the client perceives no obvious change to the binding. It is important to note that supporting the handoff only at this level is insufficient for applications with stateful servers, since there is no possibility for transferring client state to another server.

4.5.5 Traded Handoffs for Mobile Servers

The case of an application running on a walkstation, which has exported a service over the wireless network, is next considered. When the walkstation moves and a traded handoff occurs, it is necessary to ensure that its clients are able to rebind.

The redirection of clients could be achieved by the server closing its bindings and forcing clients to rebind through the new offer, after waiting for the new offer to propagate through the trader environment. However, this might cause considerable disruption to clients and a more direct method is desirable. The solution adopted here, requires both the client and the server to export an instance of the HDI and is illustrated in Figure 4.6.



(i)
(ii)
(iii)
Client
Server
(i) Client arranges Callback.
(ii) Server sends Redirect .
(iii) Client resumes communication.
Walkstation Movement.
Base Station.

Figure 4.6: Mobile Server Handoffs

First, assuming that a client is aware of the mobile nature of the server, the client locates the server's HDI and invokes the *Callback* method (i). The invocation passes the interface reference to the client's HDI to the server, where it is stashed for future use. As the server performs a handoff, it creates a new offer of service then, using the stashed interface references, binds to the HDI of each of its clients and invokes their *Redirect* methods (ii). This informs each client of the new server offer. When the client receives a *Redirect*, it must synchronise with relevant application threads to ensure that no invocations are made during the handoff. The client then binds to the new offer and releases any blocked threads. Subsequent invocations made by the client (iii) are transmitted using the new binding. Since the same server instance is rebound to, there is no need for a client to *Unmount* or *Mount*. However, a client which does not require a rebinding to the same server can choose another server at the time the *Redirect* is received.

Second, if a client is unaware of the mobile nature of the server or does not wish to ever take any action given a *Redirect*, other than to rebind to the original server, then it would be appropriate for the redirection to be handled transparently. This is provided, as was the case for mobile clients in Section 4.5.3, through defaults implemented by the RPC service.

In the case that the server is unable to contact the client, for example because the client is also mobile and is performing a handoff at the same time as the server, then the *Redirect* call is lost. The client will eventually find its binding broken, but should be able to rediscover its server through the trader.

4.6 Application Migration

In the Notus architecture, application migration between hosts is achieved by an application – the *source*, first producing a *checkpoint*, which contains the execution state of the application, then starting a version of the application – the *target*, on the remote host and transferring the checkpoint to the target. The target application then restores from the transferred state and continues execution. This section describes how the Notus architecture provides the support required for application checkpointing and restoration.

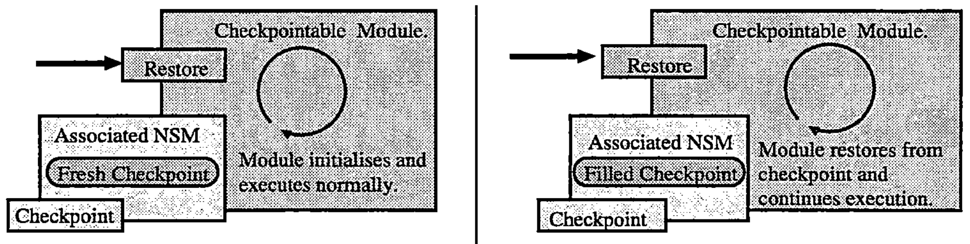
The transferred *execution state* consists of thread function call state and module state. A *checkpointable module* is a module which inherits from a base type containing one operation: *Restore*. Each NSM can be configured to contain the heterogeneous representation of the execution state of its associated checkpointable module. It also supports the operation *Checkpoint*, which returns the execution state of the checkpointable module.

It is possible for the actual format of the execution state to be module specific. For example, a module which does not contain a thread of execution is not required to supply thread call state. It is also possible for applications with existing state saving functionality to store this state in the NSM or transfer it directly to the target during migration.

4.6.1 Module Initialisation

A checkpointable module is initialised by invoking its *Restore* operation. There are two cases to consider, depending upon whether the associated NSM contains a *fresh* or a *filled* checkpoint. A fresh checkpoint has not been used to store the state of the module, whereas a filled checkpoint contains the execution state of a previously checkpointed instance of the module.

Figure 4.7 illustrates the two cases for module initialisation. In case A, the associated NSM contains a fresh checkpoint. This is detected by the module, which initialises and begins to execute normally. In case B, the associated NSM contains a filled checkpoint. The module restores its state from the filled checkpoint and restarts execution.



(A) Initialisation during application startup.

(B) Initialisation after migration.

Figure 4.7: The Initialisation of a Checkpointable Module

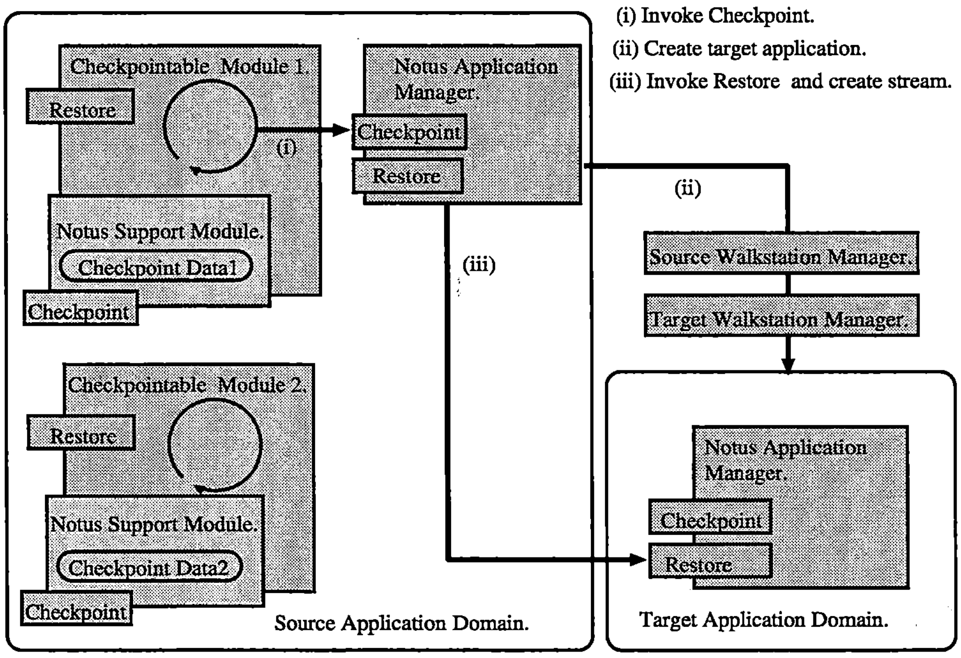
During initialisation, NLE support is required to detect a *filled* checkpoint and to enable a checkpointable module to automatically restore state from the checkpoint. If all datatypes which can be checkpointed are defined in the checkpointable module's interface, the NLE enables function call and nominated module state to be automatically marshalled using stubs generated from the interface. State which cannot be described in the module's interface is marshalled using code supplied by the application. The NLE allow this additional state to be appended to the module's automatically generated checkpoint.

4.6.2 Initiation of the Migration Process

The migration process is initiated when the application's NAM is requested to migrate a *subset* of the application's checkpointable modules to another host. This request is made either by the application itself or the NWM. Once the decision to migrate has been made, a specific target must be identified. The choice of target should be made by the user after consideration of information provided by the NWM, which performs a resource location operation. It is important that the user of the walkstation remains in control of the migration process, since there is little point in migrating unwanted applications.

The most basic resource requirements are that a target host is willing to accept the user's application and that an executable version of the application exists on the target. The first requirement is essentially a system administration issue, requiring authentication of the user at the target. The second might require the transfer of an executable to the target in advance of the migration. Other requirements of the target are specified in the form of property constraints and are queried using the trader.

With a suitable target found, the most appropriate subset of the application's modules should be specified for migration. It is possible that an application supports a number of different configurations, depending upon the computational resources of the source and target. In such cases, it would be useful to allow the user to choose between the different possible configurations.



(i) Invoke Checkpoint.
(ii) Create target application.
(iii) Invoke Restore and create stream.
Checkpointable Module 1.
Restore
(i)
Notus Application Manager.
Checkpoint
Restore
Notus Support Module.
Checkpoint Data1
Checkpoint
(iii)
(ii)
Source Walkstation Manager.
Target Walkstation Manager.
Checkpointable Module 2.
Restore
Notus Support Module.
Checkpoint Data2
Checkpoint
Source Application Domain.
Notus Application Manager.
Checkpoint
Restore
Target Application Domain.

Figure 4.8: Initiating the Migration Process

Figure 4.8 shows an active checkpointable module in the source application domain, which initiates application migration by invoking (i) the *Checkpoint* method of its NAM. The NAM requests the creation of a target application domain, via the NWM at the source and target (ii). Once the target domain has been created and a target NAM is running, the source NAM invokes the *Restore* method of the target NAM (iii) to request the creation of a stream for the transfer of execution state.

4.6.3 Completion of the Migration Process

At the target, an instance of the application is initialised with no modules instantiated other than the NAM. After the invocation by the source NAM of the *Restore* method of the target NAM, a stream is created between the source and target NAM which is used to transfer the application's state. At the source, the associated NSM for each module which is to be migrated is called to add its checkpoint of the module's execution state onto the stream to the target. Support is required to create the checkpoint in a consistent manner and is described later in Section 4.7. The target NAM then receives the stream of data and decomposes it into a number of checkpoints for individual modules.

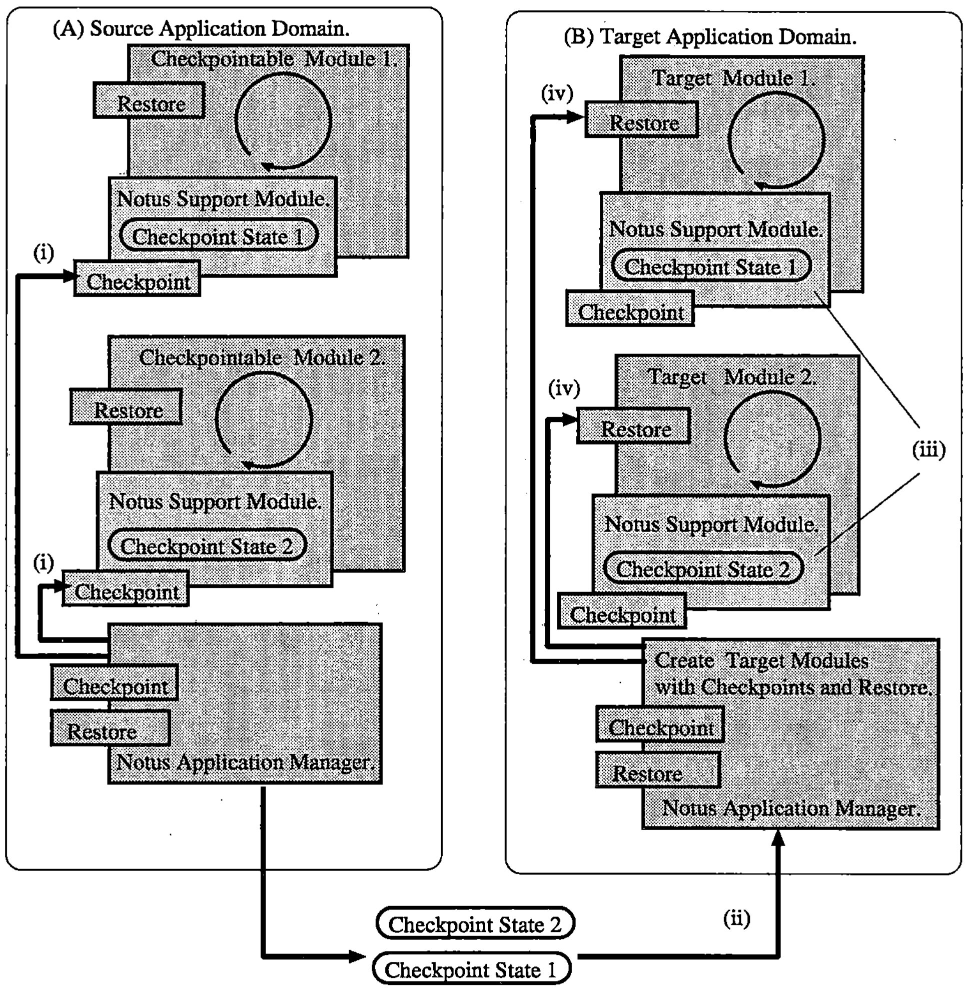


Figure 4.9: Transfer of Checkpoints and Target Restoration

Each checkpoint is typed, enabling the target NAM to instantiate a module of the correct type and invoke its *Restore* operation. With the new target module is associated a new NSM containing the received checkpoint. During initialisation, the target module notices, with the support of NLE, the filled checkpoint, unmarshals execution state, and re-starts execution at the correct point in the module. The migration process is completed when all target modules have successfully restored from their checkpoints. For migration, the modules at the source which correspond to those restored at the target are killed[3].

Figure 4.9 shows the completion of the migration process. The source NAM

[3]Application replication would result if the source modules were allowed to continue.

invokes the *Checkpoint* method (i) of the application's checkpointable modules. The modules return their checkpoint state, which is transmitted to the target application (ii) and received by the target NAM. The target NAM creates new modules and seeds each associated NSM with the appropriate state (iii) finally invoking the *Restore* method of each module (iv).

4.6.4 Interworking Migration with Handoffs

It is possible to interleave the traded handoff and application migration services, the intention being to rebuild an application's bindings as the application migrates to another host. The process of handing-off bindings during migration is shown in Figure 4.10 and starts with the source NAM issuing an *Unmount* request to the source application's modules, followed by a request for the application to migrate. When the application has restarted on the target host, the target NAM issues the *Open* and *Mount* requests.

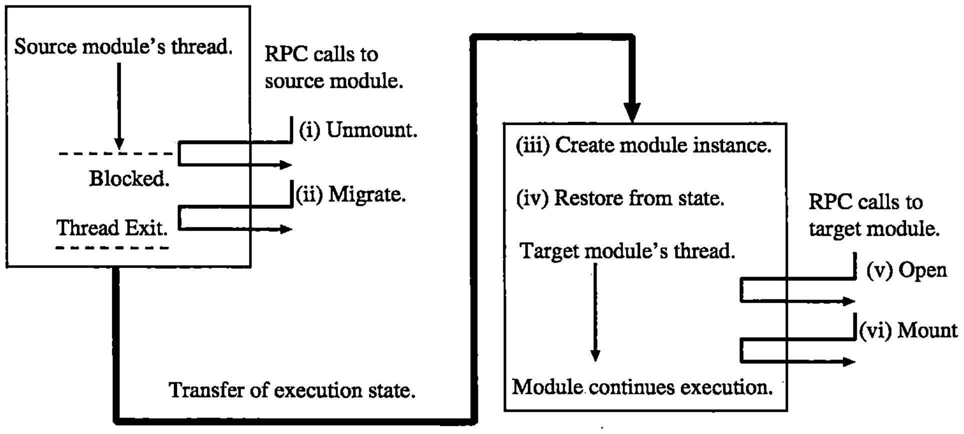

Source module's thread.
RPC calls to source module.
(i) Unmount.
Blocked.
(ii) Migrate.
Thread Exit.
Transfer of execution state.
(iii) Create module instance.
(iv) Restore from state.
Target module's thread.
Module continues execution.
RPC calls to target module.
(v) Open
(vi) Mount

Figure 4.10: A Traded Handoff during Application Migration

4.7 Checkpoint Consistency

This section considers the issues which arise when an application contains modules whose state must remain consistent with other modules within the application or the external environment. Here, a *consistent* checkpoint from a number of asynchronously executing (active) modules, is a checkpoint which captures the state of all the modules at some instant in time. The Notus architecture provides migration support for applications which are comprised of a number of active modules, and since it would be expected that some of these will communicate with each other, support is required to assist these modules in the creation of a consistent checkpoint.

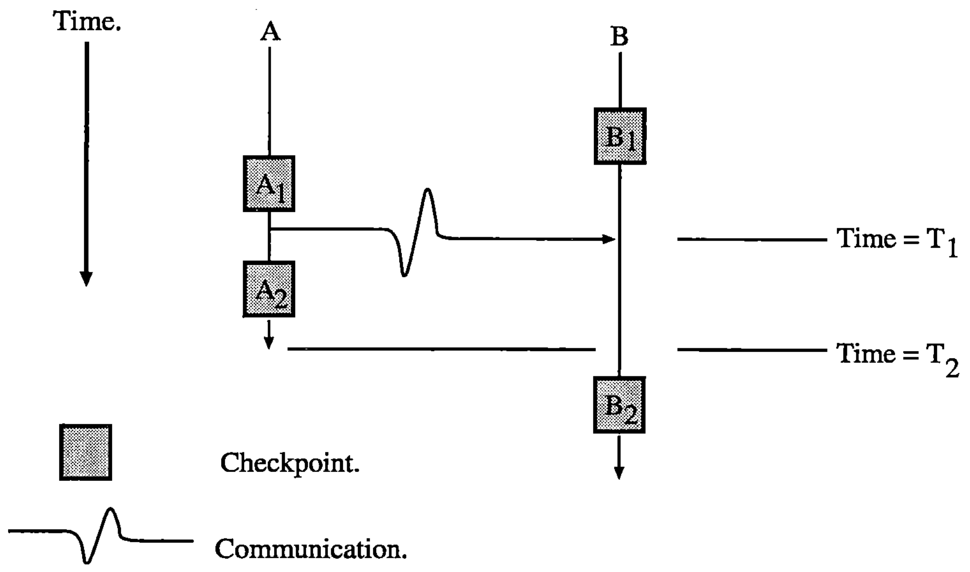


Figure 4.11: Checkpoints and Interacting Modules

For example, Figure 4.11 shows the two modules A and B, which are checkpointing without consideration of consistency. Module A updates its internal checkpoint over time, producing checkpoints A_1 and A_2. Module B similarly produces checkpoints B_1 and B_2. At time T_1 the modules synchronise and exchange state. However, if at time T_2 the most recently available checkpoint from each module is taken to form a global checkpoint of the whole application, the resulting checkpoint (A_2, B_1) is not *consistent*, since the effect of the state exchange has been reflected in A_2, but not in B_1.

The problem of ensuring a consistent global checkpoint can be solved using one of two basic strategies: *roll-backs* or *synchronisation*. The aim of the work described in this dissertation is to facilitate checkpointing of the current state of an application in order that it be restarted on another host at the same point in its computation. For this reason, the Notus architecture adopts a variety of synchronisation strategies, though for completeness, roll-back strategies are first described.

4.7.1 Roll-Back Strategies

When using a roll-back strategy, such as those described in [Randell75, Strom85] a stream of checkpoints is produced by each module, without synchronisation with any other entity. At the time that a global checkpoint is required, one of each module's checkpoints is chosen such that the set of checkpoints is consistent. In order to determine the consistent set of checkpoints, it becomes necessary for a module's checkpoint to contain a log of all messages sent and received from other modules since the last checkpoint was taken. The checkpoints of two modules are inconsistent if one module's checkpoint has recorded the receipt of a message from another module and the sender has no knowledge of the transmission. In the example shown in Figure 4.12, it is possible to create a consistent checkpoint between modules A and B, since at time T_1, although receipt of the message X is not recorded in B_1, its transmission has been recorded in A_1, and X can be replayed. However, it is not possible to create a consistent checkpoint between modules C and D, since D_1 contains no record of transmission of the message Y, and the receipt of Y, could have caused a change in state which is reflected in C_1. After the restart, circumstances might be such that D does not reach a state where Y may be correctly sent.

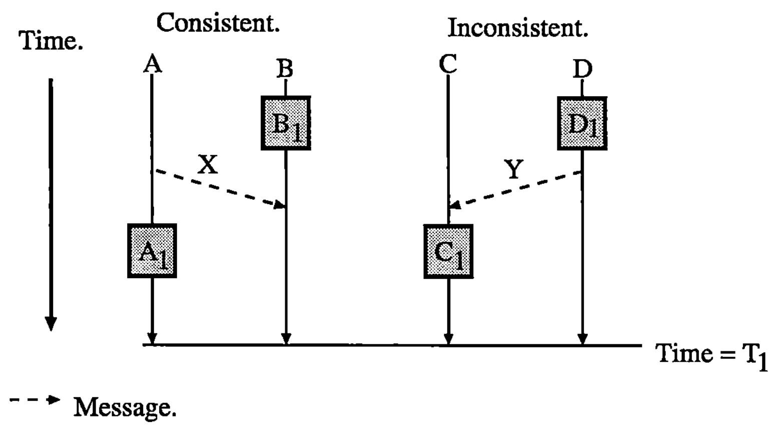


Figure 4.12: Consistency and Roll-Backs

Roll-back strategies have the attraction of causing little disruption to each module as it produces checkpoints. This is particularly useful when the motivation for checkpointing is the recovery from infrequent failures. However, at the time the application is restarted, it is sometimes necessary to roll-back through the stream of checkpoints from each module to form a consistent set. In some circumstances a large number of checkpoints must be discarded, resulting in the application potentially being forced to repeat large amounts of work. Another disadvantage is that for each module, every checkpoint made after the last which was known to be part of a consistent group must be stored until another consistent group is determined. This can be problematic, especially if checkpoints are large, and storage space, as in the case of walkstations, is limited. Finally, it should be noted that interactive applications are difficult to reconcile with a checkpointing strategy which does not save the current state of the application. It would be a requirement to store and subsequently replay during a restart, all of the user inputs to each module made after the last consistent checkpoint from that module. For these reasons, a roll-back strategy was not adopted for the Notus architecture.

4.7.2 Synchronisation Strategies

Synchronisation strategies, such as the distributed snapshot algorithm [Chandy85], require some synchronisation between modules at the time a checkpoint is made so as to ensure that each checkpoint from a module is a member of a globally consistent set of checkpoints. In the example shown in Figure 4.11, the checkpoint initiated by module A at time T_2 would not complete until checkpoint B_2 became available. The use of synchronisation strategies results in greater disruption to an application's modules during checkpointing, than would the use of a roll-back strategy, since some modules are required to wait for others to produce a checkpoint. However, synchronisation strategies guarantee that an application is never required to roll-back, and only one checkpoint per module need ever be stored.

If an application contains a number of modules, of which only some communicate with each other, then synchronisation is not required between all the modules when forming a consistent checkpoint. If a number of *checkpointing groups* are formed, each containing a set of modules which exchange information between any two checkpoints (subject to a transitive closure), then a consistent checkpoint for an application is formed from an amalgamation of the consistent checkpoints from each checkpointing group.

An implementation should allow a module to join and leave a named checkpointing group at any time, and the specification of the checkpointing group which a module wishes to join, together with the desired consistency, are expressed for each module using NLE. Since the membership of a group is dynamic and the required consistency depends upon the current members of a group, the determination of each group's membership and consistency must be made by the NAM at the time an application migrates. Also for performance reasons, an implementation such as that described in Section 5.11, should output the state from checkpointing groups as they synchronise, directly onto the stream between the source and target, rather than waiting for all groups to synchronise before commencing output.

By partitioning an application into a number of consistent groups, it is expected that the disruption to individual modules should be reduced. If one or more groups fail to reach a checkpoint, it is possible for an application to produce a *partial checkpoint* from the other groups. This might be useful during application migration, where the transfer of a partial checkpoint would correspond to an application migrating whilst leaving behind groups which were unable to reach a consistent checkpoint. Such groups might be expected to follow at a later time.

Support for three levels of consistency: *strong*, *intermediate*, and *weak* is incorporated into the Notus architecture.

Strong Consistency: A group of modules which is required to remain consistent with external state or with other asynchronously executing modules within the same group, is termed strongly consistent. In order to reach a consistent checkpoint, members of a strongly consistent checkpointing group must block after producing their own checkpoint until all members of the group have produced a checkpoint. This adds to the latency of the checkpointing operation. If one of the group does not reach a checkpoint within a useful period of time, a consistent checkpoint will not be available from that group.

Intermediate Consistency: A group of modules which must remain consistent with external state, but not with each other has the property of intermediate consistency. Since there is no requirement to synchronise with other modules, members which reach their own checkpoint can immediately add their checkpoint to the application's globally consistent set of checkpoints and continue execution. It is possible to represent an intermediate group as a number of strongly consistent groups, each of which contains a single module.

Weak Consistency: Other applications might contain groups of modules which are not required to be consistent with other modules or external events; these are termed weakly consistent. When checkpointing a weakly consistent group, it is possible to use the last checkpoint produced by each member of the group. This enables checkpoints to be prepared in advance, reducing the latency of the checkpointing operation.

Trade-offs: In choosing between weak and intermediate consistency, applications must balance the trade-offs between the overhead of producing sufficient checkpoints, so as to ensure that there is always a recent checkpoint available for each module, against the latency caused by waiting for each module to reach its next checkpoint. For this reason, weak checkpointing is an option only when the size of a module's checkpoint is small. An advantage of intermediate (and strong) checkpoint consistency is that it is not necessary to store a module's checkpoint in an intermediate buffer, since at the time the checkpoint is required, the module has blocked. This enables very large checkpoints to be sent directly onto the checkpoint stream. On the other hand, once at least one checkpoint has been made by each module, an application which uses weak checkpointing can make available a complete checkpoint at any time. This enables the application to migrate, even if one or more of its source modules are blocked at the time of the request to migrate.

The checkpoint synchronisation techniques which have been described in this section would appear similar to those used in transaction processing. For example, it is possible that a module checkpoints before starting a new transaction. On a failure or abort, the module could be rolled back to the previous checkpoint and restarted. However, the emphasis of the Notus architecture is on checkpointing the *current* state of the application and so there is no requirement to support more general circumstances such as nested transactions. Integration of the architecture with a complete transaction processing suite is beyond the scope of this dissertation.

4.8 Garbage Collection

The Notus architecture enables a subset of an application's modules to migrate to another host (see Section 4.6.2). For example, it might be useful to migrate computationally intensive modules from a walkstation to a compute server while retaining modules which manage the application's user interface. It is anticipated that applications which were formerly designed to operate in a single domain may, through the use of Notus migration facilities, become distributed over a number of domains on different hosts. Once applications become distributed in this manner, there is the requirement for some form of distributed garbage collection in order to re-use resources which are no longer required.

Where resources are collected at the level of abstract data types, such as a shared parse tree, garbage collection is at a *fine granularity*. Here, a piece of state is collected only when no module possesses a reference to it. Although not explicitly supported, an implementation of the Notus architecture does not preclude distributed garbage collection at a fine granularity.

Conversely at a *coarse granularity*, garbage collection ensures that all the resources held by all the components of a distributed application are collected when the application terminates. Since this also preserves the failure properties of a single domain, probably expected by existing applications which have been adapted to use the Notus architecture, garbage collection at a coarse granularity is directly supported by Notus.

4.8.1 Fine Granularity Garbage Collection

Garbage collection at a fine granularity is generally achieved in a single domain by tracing references to determine the unreachability of state from a set of root objects; such techniques are described in [Wilson92]. In a distributed environment, garbage collection at a fine granularity is further complicated by the requirements to operate in the face of independent failure modes, where no single component has knowledge of global state and where there are significant communication costs between collectors. There has been a considerable effort [Dijkstra78, Shapiro90, Evers93] in implementing garbage collection schemes for a distributed environment. Because of their complementary properties, these generally employ a hybrid of a reference counting and tracing schemes.

Using a reference counting scheme, a collector propagates the existence of local references to remote state (and a subsequent indication of the lack of such references) to the local collector for the state. The local collector collects the state if the reference count drops to zero. This scheme is attractive in a distributed environment, since it requires little modification to local garbage collectors or synchronisation between collectors on different hosts. However, in their simplest form, reference counting schemes are unable to collect distributed cycles of garbage. Tracing schemes follow references from root objects, collecting all garbage, but requiring greater synchronisation and cooperation between collectors.

4.8.2 Coarse Granularity Garbage Collection

In Notus, a distributed application is represented as a number of modules, some resident on a walkstation, others on the static network. The modules on the static network may execute on a number of different hosts, but on any one host an application's modules execute within a single application domain. An instance of the NAM also resides within this domain.

As shown in Figure 4.13, one particular NAM is nominated as the *Master* and is responsible for ensuring that all other NAM instances are running. In order to minimise communication over the wireless link, any NAM on the static network is chosen to be the Master. With this arrangement, only one message in each direction between the walkstation and Master is required to assure both the walkstation and Master that the application is executing in all domains.

Each application must specify, using NLE at the time its NAM is initialised, a criteria which permits the Master NAM to determine the reachability of all the application's domains, and the policy for when it is found that a domain is

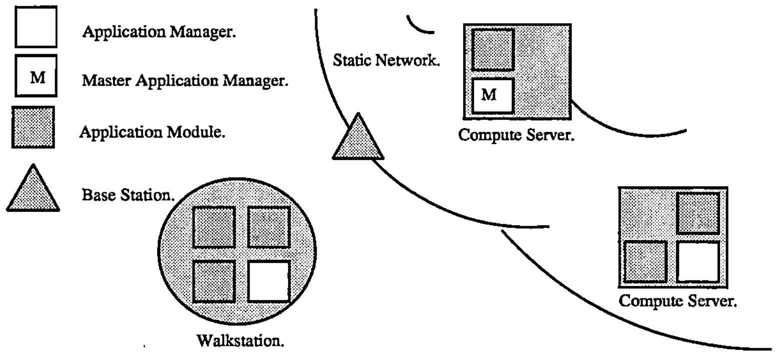


Figure 4.13: A Distributed Mobile Application

unreachable. The criteria may specify a timeout to be applied when attempting to contact other domains, together with an exception handler for failures. The default should be to propagate a termination request to all domains. If a walkstation intends to disconnect whilst leaving parts of its applications running on the static network, the Master NAM is informed so that the remaining domains are not terminated. When the walkstation reconnects, its NAM greets the Master NAM to ensure that the application is still running.

4.9 Security

Although the Notus architecture does not explicitly provide a secure environment for applications, it does not introduce any activities that greatly degrade security. Neither does Notus prevent the integration with a model such as Kerberos [Steiner88] which provides an authentication and authorisation service in an environment of untrusted workstations, using a trusted third party. The security issues which arise from the Notus architecture include:

- authentication and authorisation between clients and servers during the traded handoff,
- the movement of a walkstation between insecure and secure environments,
- authentication and authorisation between a walkstation and a compute server before commencing application migration activities, and
- the secure transfer of an application's execution state during migration.

These issues are addressed in the following subsections.

4.9.1 Security During Traded Handoffs

Mutual authentication and authorisation between a client and server is possible during a traded handoff, since the application is invited to partake in the same procedure as it made during initial establishment with the server.

The traded handoff protocol offers a sensible behaviour when a walkstation is carried behind a security firewall. After the traded handoff, the walkstation communicates using new bindings which have been created from behind the firewall. It would be hoped that most bindings are re-established to services within the secure network. However, if bindings are required to be established to services outside the firewall, then no encapsulation of network traffic is required and the firewall can apply normal scrutiny.

Finally, the Notus architecture does not make any assumptions concerning admission, and it is acceptable for the administrative environment to choose to accept or reject a walkstation which attempts a greeting based upon any criteria which it chooses.

4.9.2 Security During Application Migration

When a walkstation migrates an application to a target host, there should be some form of mutual trust between the target and the walkstation. In the Notus architecture, this is achieved by the target making its own authentication and authorisation of the walkstation and its user at the time when the walkstation attempts to start an instance of the application and the walkstation's user manually chooses the target. If the architecture were extended so that this process became automated, then steps would also need to be taken to ensure that the choice of target did not compromise the walkstation.

Once the target has been chosen, the walkstation must ensure that a suitable version of the migrating application executes on the target. This executable is chosen by the walkstation and may be transferred in advance from a trusted repository to the target. During migration, security of the state transferred between the applications can be ensured through the establishment of a secure channel between the two applications; for example through the use of the Needham and Schroeder key distribution protocol [Needham78], with the addition of timestamps.

Other attacks, such as eavesdropping over the wireless network after application migration, are possible, and might require the application to switch to a secure mode of communication between its modules. Since bindings are re-established through a traded handoff during migration, this may be arranged in a similar manner to the application's QoS negotiations. The subcontract mechanism, described in Section 5.4, which allows additional functionality to be incorporated after the marshalling stage of an RPC invocation, can be used to hook into a standard interface for security mechanisms, such as the GSS-API [Linn94]. Such issues have been addressed further [Wernick96].

4.10 Summary

This chapter has introduced the Notus architecture, which supports applications in a mobile environment through the provision of traded handoff and application migration support. The main components of the architecture are:

- the Notus Walkstation and Application Managers, which support the coordination and synchronisation of traded handoffs and application migration for both the walkstation and individual applications respectively, and
- the Notus Support Module, associated with each application module requiring Notus functionality and which is exposed to the application programmer through Notus Language Extensions.

These components and their interaction with applications as they perform traded handoffs and migration have been described. The principal points covered are:

- the components of the architecture and their functions,
- the assumptions made by the architecture regarding the programming and network environments in which it is expected to operate,
- the means by which the architecture supports traded handoffs for various cases when both client and server applications are mobile,
- the means by which application migration is achieved and the consistency issues when the application contains a number of concurrently executing modules,
- the interworking of the handoff and migration schemes, to ensure that an application's bindings to services are maintained during application migration,
- the extent to which the architecture should attempt to resolve the garbage collection issues raised when applications become distributed over a number of hosts after migration, and
- the security issues raised by the architecture.

The next chapter will describe an implementation of the Notus architecture which concentrates on the key features: traded handoffs and application migration.

Chapter 5

Implementation

5.1 Introduction

This chapter describes an implementation made of the Notus architecture. In Section 5.2 the implementation environment is described, followed by a description in Sections 5.3, 5.4 and 5.5 of the work undertaken to construct the remaining features which were required. This included a Remote Procedure Call (RPC) service providing default traded handoff support and a trader which supports Quality of Service (QoS) during service negotiation.

The chapter then describes the implementation of the various Notus components in Sections 5.7 to 5.10 and finally, examines the algorithm used to ensure the formation of a consistent checkpoint from a number of modules, in Section 5.11.

As a review of the interaction between the Notus components, this chapter first presents a brief overview of the traded handoff and application migration processes.

5.1.1 Traded Handoff Review

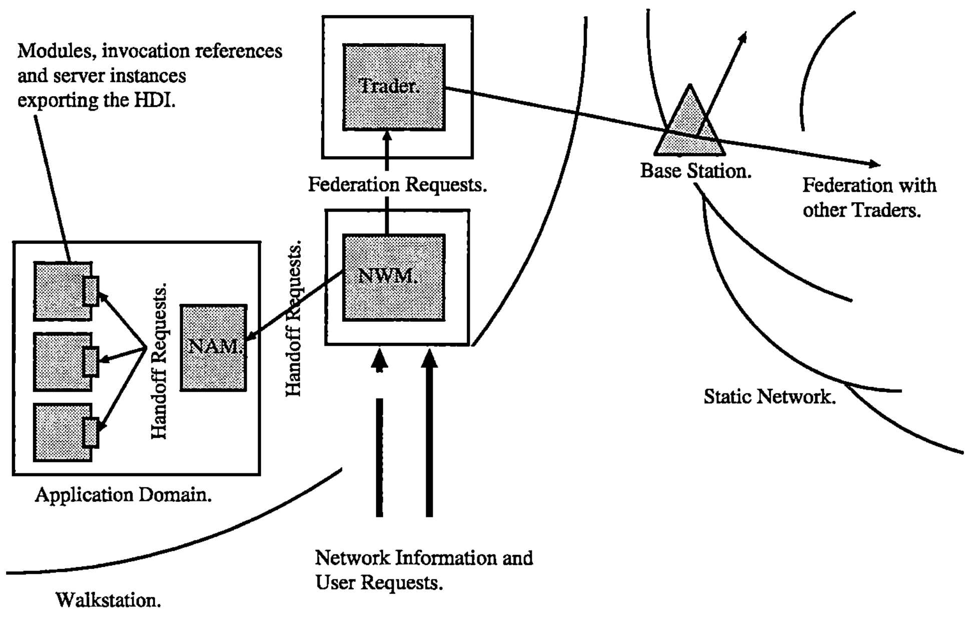

Modules, invocation references and server instances exporting the HDI.
Trader.
Federation Requests.
Base Station.
Federation with other Traders.
NWM.
Handoff Requests.
Handoff Requests.
NAM.
Static Network.
Application Domain.
Walkstation.
Network Information and User Requests.

Figure 5.1: Traded Handoff Review

Figure 5.1 shows a walkstation which is communicating via a base station with the static network. The walkstation is running a Notus application, an instance of the Notus Walkstation Manager (NWM) which receives network information and user requests, and a trader which is federated with other traders on the static network. The NWM responds to changes in the environment by instructing both the trader to federate via new base stations and the application to rebuild its connections.

Within the application domain is an instance of the Notus Application Manager (NAM), which responds to traded handoff requests from the NWM. The application also contains a number of modules which export the Handoff Interface (HDI). During a traded handoff, the NAM invokes operations on the HDI instances.

5.1.2 Application Migration Review

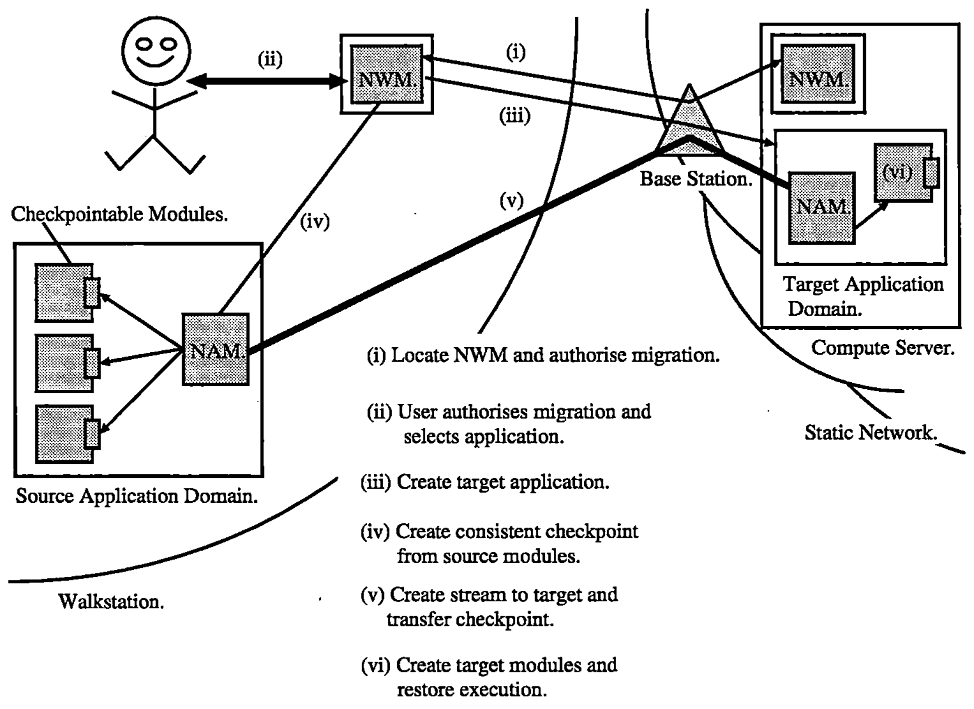

(ii)
NWM.
(i)
(iii)
NWM.
Base Station.
NAM.
(vi)
Checkpointable Modules.
(iv)
(v)
NAM.
Target Application Domain.
Compute Server.
(i) Locate NWM and authorise migration.
(ii) User authorises migration and selects application.
Static Network.
Source Application Domain.
(iii) Create target application.
(iv) Create consistent checkpoint from source modules.
Walkstation.
(v) Create stream to target and transfer checkpoint.
(vi) Create target modules and restore execution.

Figure 5.2: Application Migration Review

Figure 5.2 shows a walkstation in a high-bandwidth environment containing compute servers. The two NWM instances on the walkstation and compute server establish communication (i) and the compute server authorises the walkstation to transfer its applications from the walkstation to the compute server.

At this point, the user of the walkstation must be consulted (ii) to authorise application migration and to nominate particular applications for migration. The NWM then creates a target instance of the application on the compute server (iii) and instructs the source NAM to assemble a consistent checkpoint from the application (iv). The source and target NAM both cooperate, creating a stream between themselves (v) and a transfer of the application's state takes place. At the target, instances of the application's modules are created from the application's state (vi) and restored. Finally, the source application is killed and the target application continues execution.

5.2 Environment

A starting requirement for an implementation of the Notus architecture is the availability of a Distributed Programming Environment (DPE) with the following features:

- QoS support during service negotiation,
- an RPC service providing the default traded handoff support,
- support for a modular application structure and the evaluation of existing interactive applications, and
- a stable programming environment.

One potential implementation environment was the object oriented systems programming language, Modula-3 [Cardelli88]. This would have provided a distributed programming environment with a modular structure and strong typing. However, it was thought that implementation through this route would require extensive compiler modifications and would detract from the primary goal: an implementation of Notus which is sufficient for an evaluation of the core features of the architecture for interactive applications. Other options, such as Spring [Hamilton93a] and Spin [Bershad95] were not used because of the unavailability of resources or the lack of suitable applications with which to evaluate the implementation.

The chosen implementation route for Notus was to port the Nemesis type system and module support to the DEC MIPS and DEC ALPHA, UNIX based platforms. Nemesis was reviewed in Chapter 2 and its features include a heterogeneous, modular programming environment, a dynamic type system, and a uniform namespace.

The port was achieved quickly, and the resultant hybrid of a Nemesis and UNIX platform was stable and offered standard development tools. It was possible to incrementally port existing UNIX based, interactive applications to this hybrid environment.

It should be noted that whilst the Middl Interface Definition Language (IDL) used by the Nemesis operating system contains some unusual features which are not present in other mainstream object based DPE, such as OSF-DCE [OSF91], RM-ODP [ISO95b], and OMG-CORBA [OMG95], no unusual features were required for the Notus implementation. By using only the upper layers of Nemesis on a UNIX platform, any advantages from the single address space environment and QoS based scheduling were lost; however, neither were required for the experimental work undertaken.

This left a requirement for the implementation of an RPC service with default handoff support, a trader which provided support for QoS considerations during service negotiation, and the Notus components.

5.3 RPC Implementation

An RPC service was implemented as an extension to the Nemesis inter-domain communication model. This enabled all interfaces in the system to be described using the same IDL, with module invocations uniform in the cross machine and same machine cases, although with different failure properties. Stubs were created in the conventional manner, using the Middl stub compiler from the type definition of an exported interface. These stubs are dynamically available using the Nemesis type system and are stored in a repository. The implementation supports both TCP/IP [Postel81a, Postel81b] and MSNL [McAuley89] transports and some performance measurements are given in Chapter 6.

Using the RPC service, instantiating a service results in a number of actions taking place at the server:

- A thread is assigned the task of listening and accepting network connections from clients who wish to establish bindings with the server.
- A service offer is created which packages appropriate stubs, a subcontract (see Section 5.4), and a dispatch mechanism. The service offer is coupled to the interface of the server module and contains the *SvrControl* interface which enables the service to later be withdrawn.
- The application which has instantiated the service, or the RPC service itself, can export an offer for the service in a trader.

Given possession of a suitable interface reference, a client establishes a binding to a server. In the implementation, an interface reference consists of: a network address, which is used to establish communication with the server, a subcontract type, an interface type fingerprint, and a server identifier. The latter two are used to provide an assurance that an interface reference is used for the intended server. The interface reference does not contain QoS information from the exporting server. Such QoS information is likely to be inaccurate at the time a client establishes a binding and in any case, cannot reflect the QoS over all possible network routes to a server. QoS negotiations take place during binding establishment.

In the case that the client and server are on different machines, the client's request to establish an RPC binding with the server results in the establishment of a network connection between the client and server, and at the server-side a thread is dedicated to the receipt and processing of invocations. The client is returned an invocation reference (introduced in Section 2.4), which supports methods corresponding to two interfaces. One, the *ClntControl* interface enables the client to make control operations on the binding, for example to destroy it. The other, the *Proxy* interface, has the same type signature as the server module and is used by the client to make remote invocations of server methods. The following steps describe how a remote invocation made by a client progresses after the client has invoked a proxy method and is illustrated in Figure 5.3.

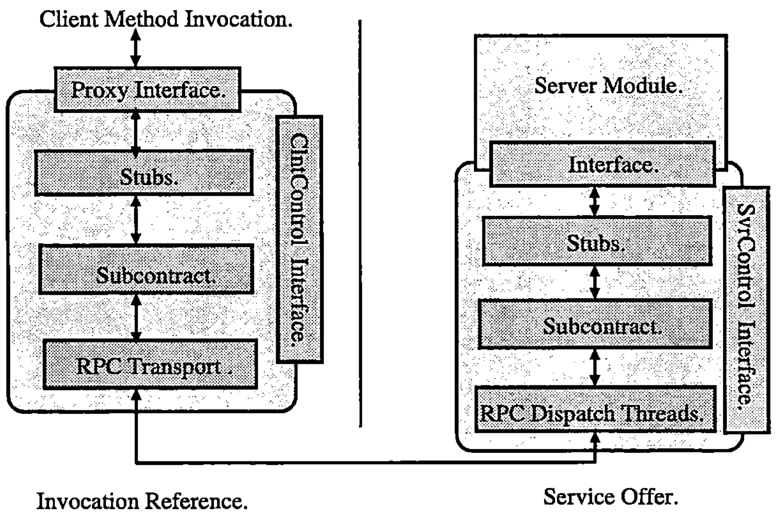


Figure 5.3: Method Invocation using the RPC Service

- The client identity, method, and argument values are marshalled in the client-side stubs.
- A marshalled invocation is transmitted over the network to the server handler thread, which passes it to dispatch code. The arguments are unmarshalled and a call made into the server.
- When the server module call returns, results or exceptions are marshalled by the server stubs and transmitted to the client.
- The client-side acknowledges the return from the server. Results or exceptions are unmarshalled by the client stubs and control returned to

the client, either as a return from the proxy method invocation with the results or by raising the appropriate exception.

A timer is set at the client-side after the server invocation is made and is cleared once results arrive. At the server-side, a timer is set after results are transmitted to the client and is cleared when an acknowledgement is received. A timeout at the client-side causes an exception to be raised and control passed to the client application, which may choose to retry the RPC. A timeout at the server-side causes the results to be retransmitted and the timer reset (for a small number of times). This corresponds to At Most Once (AMO) semantics [Birrell84].

5.4 Default Handoff Support

The principal features of the RPC service which was implemented to support Notus are present in other mainstream object based DPE. The Notus architecture however, requires additional support for default traded handoffs (described in Section 4.5.4). These features were implemented using a general mechanism for adding new behaviours to RPC bindings, which is heavily influenced by the Spring[1] *Subcontract* mechanism.

5.4.1 Notus Subcontracts

In the Notus implementation, a subcontract is a module positioned between the stubs and the RPC transport layer, and is used to modify the manner in which invocations are made to the server.

The positioning of subcontracts beneath the stubs enables their use to be transparent to the application. New subcontracts can be introduced to the system without requiring modifications to the base RPC service. Subcontracts are typed, and the type expected by a server is placed in its exported interface reference. This is used by a client as a hint, and selection of compatible subcontracts between client and server is made at the time a binding is established. A client can attempt to bind using a subcontract of a different type to that expected by the server. In this case, the server can dynamically select a different subcontract for the binding to match the client, according to the subcontract's subtyping rules.

For example, consider the *Encrypting* subcontract, which is specified as the

[1]The Spring operating system is reviewed in Section 2.5.

subtype of the *Null* subcontract. A server which specifies the Encrypting subcontract would not accept bindings from clients who attempt to use the Null subcontract. However conversely, a server which specifies the Null subcontract might dynamically select the Encrypting subcontract if a client requests it during binding establishment.

In the Notus implementation, three subcontracts were implemented to support traded handoffs: the *Null* subcontract, which supports unmodified method invocations; the *Flat* subcontract, which implements rebinding to the original end-points on a handoff; and the *Traded* subcontract, which rebinds to any server which matches the properties of the client's existing binding.

The subcontract mechanism allows applications which are unaware of mobility to respond to the traded handoff protocol. This includes legacy applications or applications which use stateless servers. An application which is aware of mobility, but wishes to use a default for some or all of its bindings should select either Flat or Traded subcontracts. For bindings where the application itself, as opposed to the underlying subcontract mechanism, wishes to take part in a traded handoff, a custom HDI interface should be exported.

5.4.2 Subcontracts for Mobile Clients

```
Handoff :  INTERFACE =

BEGIN

Unmount  : PROC [ synch : BOOLEAN ] RETURNS [ ];

Mount    : PROC [ ] RETURNS [ ];

Open     : PROC [ constraint : STRING ] RETURNS [ ];

Callback : PROC [ ref : InterfaceReference ] RETURNS [ ];

Redirect : PROC [ ref : InterfaceReference ];

END.
```

Figure 5.4: The Handoff Interface Definition

The Flat and Traded subcontracts both instantiate the HDI, which is used to coordinate the traded handoff for both mobile clients and servers (described in

Section 4.5). Figure 5.4 illustrates using Middl, the HDI definition used by the implementation.

In the case of a mobile client application, an offer for the HDI instantiated by the client-side subcontract is exported in the application's local trader and so is visible to the NAM. During a traded handoff, the NAM locates the HDI and invokes its operations. This is illustrated in Section 5.4.4 using the Traded subcontract as an example.

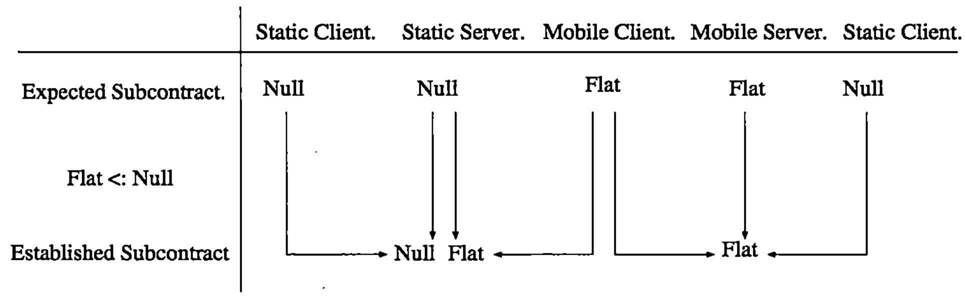


Figure 5.5: Subcontract Selection.

A server running on the static network might request the Null subcontract as a default. However, all the handoff subcontracts (for example, the Flat subcontract) are a subtype of the Null subcontract, and a static server offer would be prepared to dynamically select one of the handoff subcontracts if requested by a mobile client during binding establishment. This and other subcontract type selections are illustrated in Figure 5.5.

It should be noticed that in the current implementation, a mobile client does not require any form of redirection from a static server, and a handoff subcontract dynamically selected by a static server is used only for compatibility reasons. Hence, the Notus RPC service could communicate with a static server, which is unaware of the subcontract mechanism through the interposition of a translation module on the invocation path which strips subcontract information from the client's requests.

5.4.3 Subcontracts for Mobile Servers

A mobile server requires a client to select a handoff subcontract during binding establishment in order to receive redirection messages if the server moves.

The initialisation and binding process for the mobile server subcontracts is shown in Figure 5.6. At the time a server initialises, its subcontract creates an instance of the HDI with interface reference H_1. During binding establishment a client also instantiates a subcontract which creates an instance of the HDI with interface reference H_2. The client-side subcontract establishes communication with the server-side subcontract (i) passing the interface reference H_2 to the server, which returns H_1 to the client (ii). The server-side subcontract invokes the *Callback* method (iii) of H_1, stashing the client's interface reference.

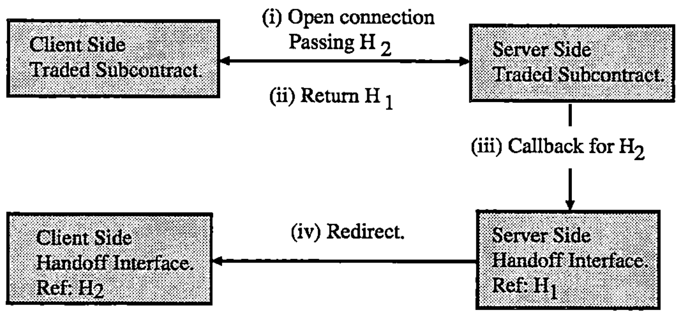


Figure 5.6: Subcontract Initialisation

The client can at any time make further invocations of the *Callback* method, either to register a new client-side HDI (useful if the client itself has performed a traded handoff), or to withdraw its requirement for a call-back altogether. If a binding is destroyed, the server-side subcontract ensures that the client's call-back is removed.

The server is requested to perform a traded handoff by an invocation of the *Open* method of its HDI. This causes the server to use the interface references of its stashed clients (here only H_2), establishing bindings with the client-side HDI, and invoking their *Redirect* methods (iv), passing the interface reference for the new server offer. At the client-side, the invocation of the default *Redirect* method causes further RPC invocations to block before rebuilding a connection to the server. If the redirection of a client fails, the client would find its bindings broken and would be required to re-locate the server using the trader.

5.4.4 Stashing

The RPC service implements the stashing facility which is used to store open connections during a traded handoff and client call-back interface references for mobile server bindings. The stash is implemented as an abstraction on top of the local trader interface, and is manifested as a set of options for use during binding establishment.

Figure 5.7 illustrates some of the stashing options, using the implementation of the HDI made by the *Traded* subcontract for a mobile client as an example. The ACQUIRE and RELEASE macros are used to block client invocations on the binding during a traded handoff, and MOBILE_IDC_DELETE is used to destroy a given connection. The MOBILE_IDC_OPEN macro, when invoked with a constraint, queries first the stash, then the trader for offers which match the constraint, and the macro then establishes a binding with any matched offer. The MOBILE_IDC_STASH macro carries out the same operations as MOBILE_IDC_OPEN, but additionally places the opened connection in the stash.

```
Open(self, c)
{
  MOBILE_IDC_STASH(self->context, self->constraint);
}

Unmount(self, synch)
{
  if (synch) ACQUIRE;
  MOBILE_IDC_DELETE(self->offer);
}

Mount(self)
{
  self->offer = MOBILE_IDC_OPEN(self->context, self->constraint);
  RELEASE ;
}
```

Figure 5.7: A Handoff Interface Implementation

During the traded handoff, the *Open* method is first invoked. This calls MOBILE_IDC_STASH to open a connection to a server with the appropriate properties, placing the opened connection in the stash. When the *Unmount* method is called, ACQUIRE prevents any further client invocations on the binding, and MOBILE_IDC_DELETE destroys the old connection. Finally, the *Mount* method invokes MOBILE_IDC_OPEN to open a new connection, replacing that which was destroyed. The opened connection is found in the stash, and RELEASE allows further client invocations to be made.

The main difference between this default implementation and an application exported HDI is that the *Unmount* and *Mount* phases would also be used by the application to cleanly unmount the old server and to transfer client state to the new server, so re-establishing the session. The stash is also used for the mobile server and client call-back cases. In all cases, it is possible for the application to consider and optionally stash all offers in a returned list. This should be compared with the example shown above, where the MOBILE_IDC_STASH macro binds to the first offer imported from the trader.

5.5 Trader Implementation

A trader implementation was required primarily to provide the standard set of trading functions for service mediation, based upon *properties* offered by servers and the *constraints* of clients [ISO94]. It was decided not to adopt existing trader implementations, such as [ANSA92, Beitz94] since the impact of mobility was not easy to predict at the time of implementation and its inclusion in existing trader implementations was considered potentially problematic. It was later found possible to implement these additional functions in a separate *QoS Filter module.*

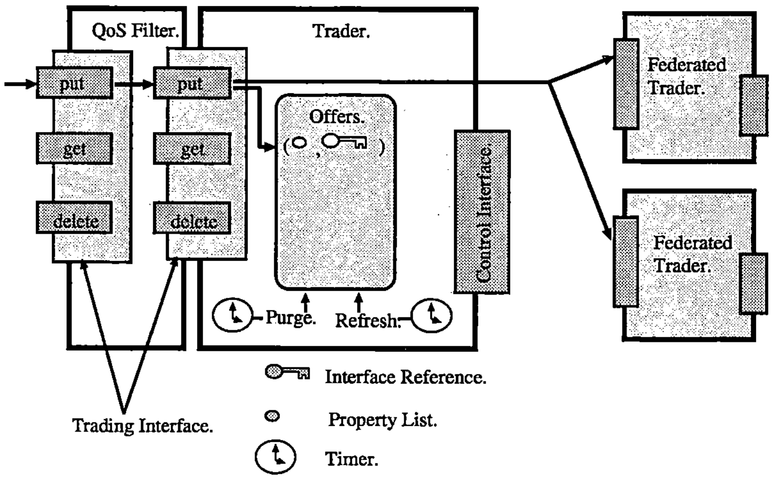


Figure 5.8: The Trader

The trader shown in Figure 5.8 supports two interfaces. One, the *Trading* interface, supports the methods: *Put* which allows clients to export new offers to the trader, *Get* which is used to query for offers using constraints, *Delete* to remove exported offers, and other methods with functions such as the modification of the properties of existing offers. The other supported interface, provides *Control* operations, such as initialisation and federation with other traders.

When querying a trader, a client requests a list of offers which match a set of *constraints.* The grammar used to specify both the properties of an exported offer and the constraints imposed by the client is similar to that defined in [ANSA93] and supports integer, string, and set data types, together with the usual operators. As a simple example, a video server offer in the context: `'services/videoPlayers'` might posses the following properties:

```
(Type = Mpeg ) , (Title = {'Bambi','Snow White'}).
```

A client wishing to watch the Bambi video would find a match in the above server offer with a constraint such as:

```
(Type = Mpeg ) and (Title = 'Bambi').
```

On receipt of a query, the trader examines its own offers, producing a list of those which match the given constraints. The trader then forwards the query to other federated traders and finally, returns the list of all matched offers to the client. For performance reasons, the trader stashes offers received from federated traders. These are subject to a timeout and are later removed by the trader, if the timeout elapses or if the offer is shown to be stale.

Similarly, when a server exports an offer, the trader stores the offer in its own stash. Again, according to the scope of the offer, the trader exports it to federated traders. As a garbage collection mechanism, exported offers which are received from federated traders are also subject to a timeout and must be periodically refreshed by the exporting trader. Hence for any offer in a trader's stash, two timeouts must be set: one which purges the offer and another which refreshes the offer in any federated trader to which the offer has been exported.

5.6 Quality of Service Support

Section 4.5.1 highlighted the fact that in a mobile environment, any properties which are related to the QoS available for an offer are difficult to express owing to the effect of mobility on QoS. Since it is not possible to determine the exact QoS available for a particular offer without direct negotiation with the exported server interface, a heuristic is used during trading. Offers from federated traders are matched only if the QoS experienced between the federated traders matches the QoS constraints from the client.

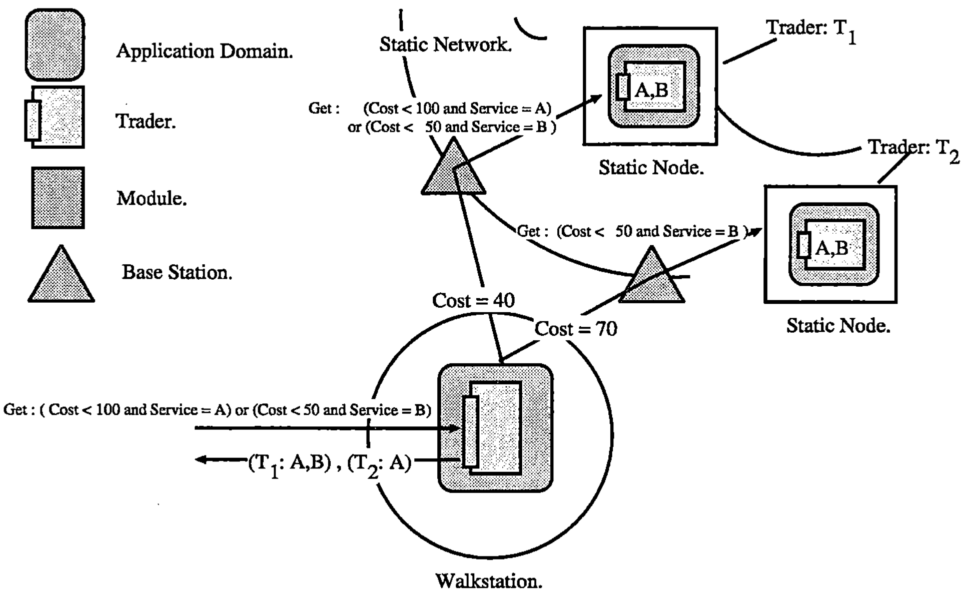

Application Domain.
Trader.
Module.
Base Station.
Static Network.
Get : (Cost < 100 and Service = A) or (Cost < 50 and Service = B)
A,B
Trader: T_1
Static Node.
Trader: T_2
Get : (Cost < 50 and Service = B)
A,B
Static Node.
Cost = 40
Cost = 70
Get : (Cost < 100 and Service = A) or (Cost < 50 and Service = B)
(T_1: A,B) , (T_2: A)
Walkstation.

Figure 5.9: QoS Constraints During Trading

This is illustrated by the example shown in Figure 5.9. Here, the walkstation is federated with two traders, T_1 and T_2, over links with respective *Costs* of 40 and 70. If the walkstation's client queries its trader with the constraint:

```
(Cost < 100 and Service = A) or (Cost < 50 and Service = B)
```

the walkstation's trader considers the QoS available to each of its federated traders T_1 and T_2 and sends a request to trader T_1 with constraint:

```
(Cost < 100 and Service = A) or (Cost < 50 and Service = B)
```

and to trader T_2 with constraint:

```
(Cost < 100 and Service = A)
```

The second `or` clause is dropped in the request to trader T_2, since it requires a link of Cost < 50 and the link to trader T_2 has a Cost of 70. These queries would result in the list of returned offers: $((T_1 : A, B), (T_2 : A))$. The walkstation's trader inserts the appropriate values for the *Cost* QoS property to each of these offers' property lists, so that they are considered by the client.

If the same server instance is available through a number of different routes, the offers returned to the client must be distinguishable. Different routes which use different transport types are distinguishable in the exported interface references. However, a choice of different network types for the initial wireless link from a walkstation would not be immediately distinguishable. In this case, the interface references returned to the client must be modified by the walkstation's trader to indicate the appropriate outgoing route.

Rather than modify the base trader implementation, these QoS considerations were implemented using the Filter module, interposed between the trader and its clients. The Filter module parses constraints before they are passed on to the trader. Clauses containing QoS properties are matched against the known QoS to each trader which is federated over a different network type and a number of modified queries are produced in the manner described above. Different traders are federated in a different context in the namespace and queries are targeted at different routes by modifying their contexts.

In the current implementation, the QoS properties which a client may use include *Cost* and *Bandwidth*. Queries from the client are expected to be a disjunction of clauses, $(C_1 \vee \cdots \vee C_n)$, where each clause C_i is a conjunction of QoS and service related properties, $(Q \wedge S)$. This restriction was made simply for implementation purposes and allows QoS constraints to be specified in an intuitive manner. However, more sophisticated QoS expressions could be handled through extensions to the parser used by the Filter module. Before the offers from the various queries are returned to the client, the Filter adds the values of *Cost* and *Bandwidth* to the properties of each returned offer.

The Filter module implementation requires the trader to provide QoS information concerning its federated traders via its control interface. Different network types are simulated using the different RPC transports available. Since these were immediately distinguishable by the client, the Filter module did not need to make modifications to the imported interface references.

If the implementation were to be used over a number of different network protocols, the Filter module could easily be extended to insert into the returned interface references – an indication to the client of the network level options required to select the appropriate outgoing network interface. This can be achieved for example, in the IP domain by using the loose source routeing extensions which were mentioned in Section 3.1.1.

By propagating the QoS constraints in the queries to federated traders, each trader in a chain of federated traders is able to consider the constraints and the QoS available between itself and the next trader, and so refine the constraints for the query. If a federated trader in the chain does not support QoS considerations, then propagation of QoS constraints cannot made. Using the above example, the constraint `( Service = A or Service = B)` would be propagated to trader T_1 and `( Service = A)` to trader T_2.

Where QoS constraints are dropped, the walkstation uses only an initial QoS consideration on the first step to select its own outgoing routes. Subsequent trading then proceeds along existing lines, without QoS support. However, for a walkstation importing an offer exported by another walkstation, it is essential that the *home* trader of the exporting walkstation supports QoS considerations in order that a comparison be made between any different wireless links from the exporting walkstation to the static network.

5.7 The Notus Walkstation Manager

The Notus Walkstation Manager (NWM) implementation shown in Figure 5.10, has the task of mediating between the user, network, and applications. It supports two interfaces: the *user* interface, which is invoked by the walkstation's user to query and control running applications; and the *network* interface, which receives connectivity information from the various network types supported by the walkstation. Information received via the network interface is used to initiate the federation of new traders, traded handoffs, and application migration. Since a wireless network was not part of the implementation, this information was simulated.

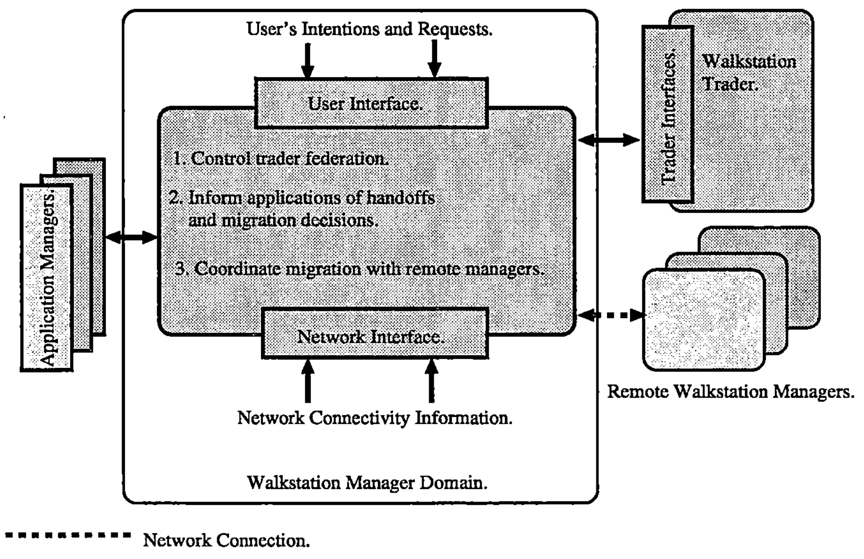

User's Intentions and Requests.
User Interface.
1. Control trader federation.
2. Inform applications of handoffs and migration decisions.
3. Coordinate migration with remote managers.
Network Interface.
Network Connectivity Information.
Walkstation Manager Domain.
Application Managers.
Trader Interfaces.
Walkstation Trader.
Remote Walkstation Managers.
Network Connection.

Figure 5.10: The Notus Walkstation Manager

Application migration from the walkstation to a compute server is initiated when the NWM perceives a high-bandwidth environment and is able to communicate with other NWM instances on compute servers. The source NWM contacts the prospective target NWM instances (which are running under an administrative user account), requesting permission to migrate the walkstation's applications.

The walkstation's user is then required to nominate, via the user interface, the target host and the applications which are required to migrate. For each migrating application, the user should specify the modules which should migrate using a trading constraint. The use of a constraint enables the specification of the modules required to migrate to be made in a flexible manner. For example, it is possible to specify every one of an application's modules, all modules of a specific type, or a particular instance of a module.

The implementation assumes the availability of an executable version of the application on the target host[2], and the source NWM uses the UNIX `rsh` mechanism to start the application under an account belonging to the user of the walkstation, specifying via command line arguments that the application should initialise from the state of another application instance. The target application instantiates only the NAM and the remainder of the migration process is handled by the NAM at both the source and target.

A traded handoff is initiated when the network interface receives simulated events indicating a change in the connectivity available through the walkstation's network links. If the walkstation loses connectivity through a particular network type, the NWM informs the walkstation trader to cease its federation with other traders over that network, and then instructs each instance of the NAM on the walkstation to perform a traded handoff. Conversely, if a new network becomes available, the walkstation's trader is instructed to federate with a trader over the new network and all NAM instances are similarly informed.

5.8 The Notus Application Manager

An instance of the Notus Application Manager (NAM) shown in Figure 5.11 is created whenever an application starts and supports the modules of an application, in coordinating registration, traded handoff, and migration activities. By convention, applications which are a migration target are informed via their command line arguments, and this information must be passed on to the NAM as it is initialised.

In the case that an application starts normally, the NAM accepts the registration of modules which wish to be notified of migration or traded handoff events. Otherwise, when an application is started as a migration target, the NAM creates a stream on which it expects to receive the state of the

[2]A transfer of an executable from a repository could be made in advance of migration.

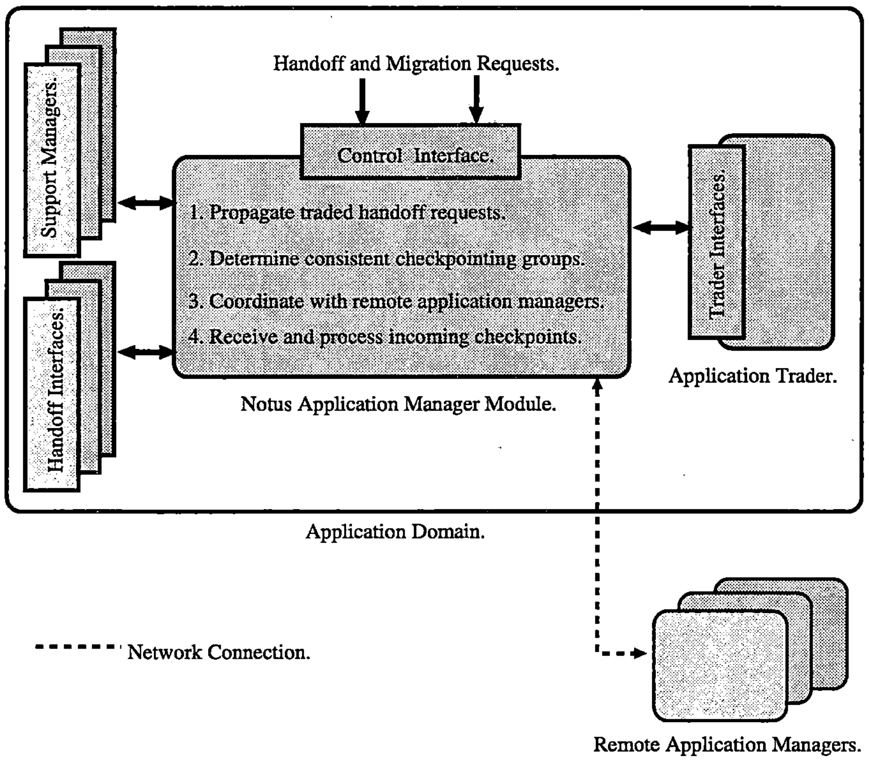

Handoff and Migration Requests.
Control Interface.
1. Propagate traded handoff requests.
2. Determine consistent checkpointing groups.
3. Coordinate with remote application managers.
4. Receive and process incoming checkpoints.
Notus Application Manager Module.
Support Managers.
Handoff Interfaces.
Trader Interfaces.
Application Trader.
Application Domain.
Network Connection.
Remote Application Managers.

Figure 5.11: The Notus Application Manager

source's checkpointed modules and passes back an interface reference to the stream's endpoint to the source NAM. While the source is creating its consistent checkpoint[3], the source NAM connects to the stream and is later responsible for coordinating the transmission of each migrating module's checkpoint to the target NAM.

On receipt of each source module's checkpoint, the target NAM first determines the type of the source module from the checkpoint state. This is used by the dynamic type system to create a new module instance. The target NAM then creates a Notus Support Module (NSM) instance filled with the received checkpoint, and a thread which will invoke the new module's *Restore* method with its associated NSM as an argument. The new module will then, with the assistance of its NSM, restore from the checkpoint state and continue execution.

After each module has been migrated, the source NAM instructs the module at the source either to terminate or continue. The former corresponds to migration, while the latter might be used as part of a replication scheme, but is not considered further in this work.

[3]The algorithm for creating a consistent checkpoint is described in Section 5.11.

The second function of the NAM is to receive and propagate traded handoff requests from the NWM. Given a traded handoff request, the NAM uses the application's trader to locate all the HDI interfaces exported by the application and passes on requests for each phase of the traded handoff protocol, by making invocations on these HDI interfaces, as described in Section 4.5.

5.9 The Notus Support Module

It was mentioned in Section 4.2 that Notus Support Module (NSM) instance is associated with any module which uses Notus functionality. In the implementation, this association is made during the module's initialisation, by passing a reference to the associated NSM as an argument of the module's *Restore* method. This means of association was chosen over other mechanisms, such as class inheritance, simply for convenience in the Nemesis programming environment.

The NSM is intended to draw functionality away from, and so simplify the implementation requirements of the Notus Language Extensions (NLE). It provides the application with simple synchronisation primitives for use during a traded handoff. However, its main requirement is to support the construction of a module's checkpoint during migration.

When the NSM initialises, it registers with its NAM, providing an interface reference to a *Control* interface, with properties which include the module type, the name of its checkpointing group, and the level of consistency (selected from the set: *strong, intermediate, or weak*) which, must be maintained with the other members of the checkpointing group. The consistency information is communicated to the NSM via NLE, and at any time during the module's execution, the NSM can inform its NAM of changes to these properties. This for example, allows a module to change its consistency requirements or checkpointing group membership at any time.

Part of the state of an active module transferred during migration is its marshalled thread call trace. This state is maintained by the NSM using a stack which represents the call trace and contains pointers to marshalling code for the state local to each stack frame. At the time a checkpoint is required, this stack is unwound and the marshalling code invoked.

The NSM is also required to provide a marshalled representation of the global module state (the *self* closure in Figure 5.14). This additional state can be marshalled automatically if it is described in the module's interface, but if not, must be marshalled using NSM call-backs to functions provided by

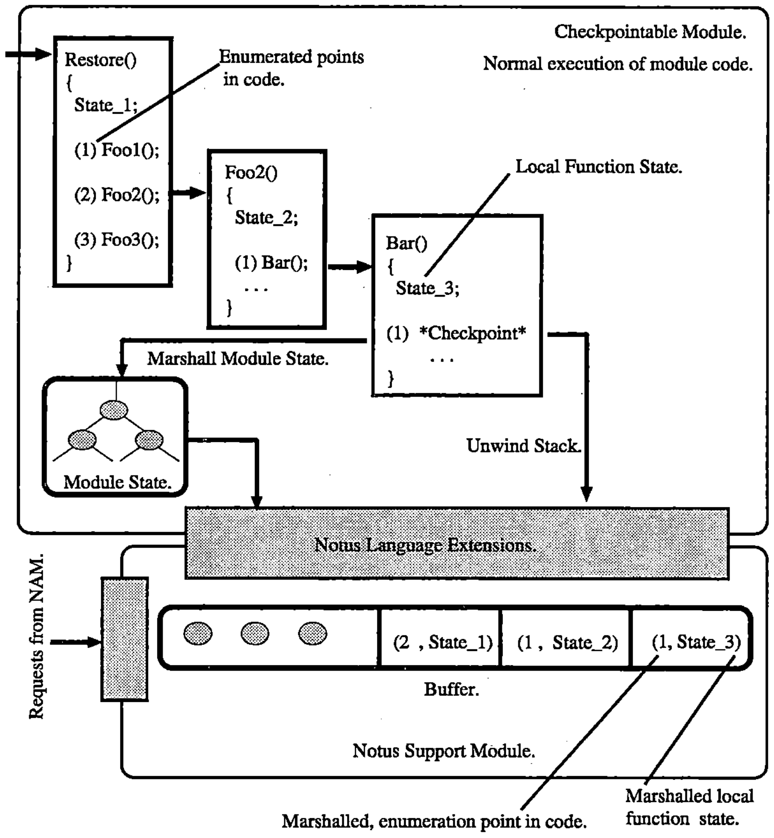


Figure 5.12: Checkpoint Assembly

the module. The programmer is assisted in creating these application specific marshalling functions, with access to marshalling code for all the data types defined in interfaces imported by the application.

The example shown in Figure 5.12 illustrates how a checkpoint is produced by the NSM. The programmer has specified that a checkpoint be produced whenever execution reaches `CHECKPOINT` in the function `Bar`. At this point, the NSM stack contains a trace of the function calls and marshalling code references. The NSM unwinds the stack and invokes the marshalling code to fill its buffer with the call sequence leading to `CHECKPOINT`, the local state for each function call, and the module's global state. The checkpoint held is now available to the NAM and can be used as part of the consistent checkpoint for the module's checkpointing group.

5.10 Notus Language Extensions

A module which wishes to use the functions provided by the Notus implementation must interface with its associated NSM. The architecture requires that applications partake in migration and traded handoffs. This potentially requires each module with an associated NSM to implement a number of functions.

- During its own initialisation, a module must detect when its associated NSM contains a *filled* checkpoint and if so, must undertake to retrieve the checkpoint and restore itself. The module is also required to register itself during initialisation, with its NAM, specifying its checkpointing group and required consistency.
- An active module must inform the NSM whenever a function call or return is made, and of the nominated state to be marshalled for each call.
- The module must specify points where traded handoffs and migration can take place and synchronise with the NSM at these points when required.
- The module should have a means of requesting that a checkpoint be formed for the application.

Also, when an application first starts, it must initialise an instance of the NAM. During its initialisation, the NAM must be informed if the application is being started normally or as a migration target. In the case that it is a migration target[4], the NAM does not return control to the application and instead coordinates the restoration. Otherwise, the NAM allows the application to initialise as normal, creating and registering its modules and associated fresh NSM instances.

Without Notus Language Extensions (NLE) support, these requirements would be a burden on the application programmer. The NLE extends the programming environment, automating to various degrees, all the above tasks. The implementation supports the C programming environment, with NLE extensions appearing as a set of macros. These are resolved during pre-processing into both embedded code and calls to the NSM. This implementation enables migration to take place between two executable instances of an application which have been compiled for different architectures using different compilers.

Figure 5.13 shows an example of expanded NLE macros which are used to restore the execution state of a module. The *Restore* method of this module has been invoked and the module's associated NSM contains a filled checkpoint.

[4]This is detected by the application through its command line arguments.

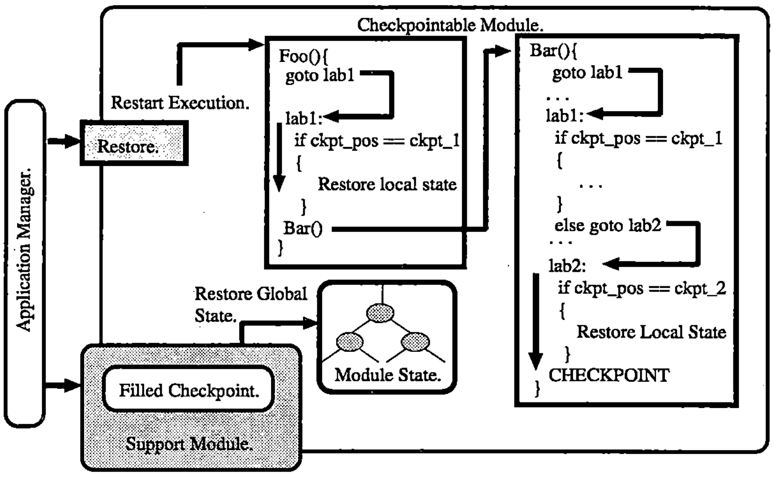


Figure 5.13: Module Restoration

The module's global state is unmarshalled and the expanded macros extract the thread call chain.

In Foo, the macros pass execution to the point where the call to Bar is about to be made and unmarshal local Foo state. Bar is then invoked and the restoration continues until the point is reached at which normal execution continues. During this process, the NSM's thread call stack is also reconstructed, enabling the restored module to produce another checkpoint as soon as normal execution is resumed.

5.10.1 Example Annotations

When using NLE to adapt a module for migration, it is necessary to annotate all *checkpointed functions*. Here, a checkpointed function either contains a checkpoint or calls another checkpointed function. Figure 5.14 shows a checkpointed function which contains two checkpoints, indicated by the macro POINT. The function initiates an application wide checkpoint, using the macro SCHEDULE, based upon a test of the variable iteration. The VARS macro is used to indicate local state which should be marshalled during a checkpoint, and the ARG macro is used to introduce the associated NSM to the NLE.

As the pre-processing stage stands, there is a requirement for several further annotations. The call to the checkpointed function Bar must be decorated with the CALL and AFTER macros. These are used to maintain the thread call stack, and also expand into code which passes execution on to the correct point when the module is restored. Finally, TOPS and BOTS are required to mark the scope of

```
Foo( self )
{
    uint32_t i;
    uint32_t temp;

    CKPT VARS uint32_t &i;
    CKPT ARG  (self->checkpoint);
    CKPT TOPS;

    CKPT POINT;            /* checkpoint */

    if(self->iteration == 100)
      CKPT SCHEDULE;       /* initiate an application wide checkpoint */

    CKPT CALL;
       Bar(self);
    CKPT AFTER;

    CKPT POINT;            /* checkpoint */
    CKPT BOTS;
}
```

Figure 5.14: An Annotated Function

the checkpointed function, and expand to code which detects a filled checkpoint and initiates restoration.

With an improved version of the pre-processor, it should be possible to eliminate the need for the `TOPS`, `BOTS`, `CALL`, and `AFTER` macros, and a tool may be envisaged which, given programmer assistance in determining the points in a program where checkpointing can occur, will determine and annotate the function call paths to each point. Along the way, the tool might also assist the programmer in determining the state which should be nominated for transfer.

5.11 Checkpoint Consistency

The source NAM is required to ensure that a consistent checkpoint is available from an application's modules during migration. This is achieved in the implementation through the use of the algorithm shown in Figures 5.15 and 5.16.

In general, a consistent checkpoint might be required from a number of modules which are members of a number of checkpointing groups (see Section 4.7).

Here, the algorithm first determines the grouping of modules and the consistency requirements of each group, by querying the properties of the modules through the trader. This allows groupings to be flexible and modules can enter or leave groups during their execution, so long as they update their properties in the trader. Since the consistency of a group is as strong as the strongest requested by any one module, the addition or removal of a module from a group can change the group's overall consistency.

```
1. Open stream.
2. Determine and group all modules to checkpoint:
   let $g_i$ := { m ∈ modules | ∀ m' ∈ $g_i$ · GroupName(m) = GroupName(m') }
   let CG := $\bigcup g_i$ · modules ⊆ CG
3. Request that all members of strongly consistent groups produce a checkpoint.
4. Write the last checkpoint from all weakly consistent group members on the stream.
5. As each strongly consistent group reaches a checkpoint, write it on the stream:
   while CG ≠ ∅
   do
      let G := { g ∈ CG | g has reached a consistent checkpoint }
      if G = ∅
      then
         wait
      else
         ∀ g ∈ G, ∀ m ∈ g
         do
            5.1 Write out the checkpoint from module m on stream.
            5.2 Signal m to allow module to continue execution.
         done
         let CG := CG - G
      fi
   done
6. Close stream.
```

Figure 5.15: Checkpoint Consistency: Manager

The algorithm is only required to directly consider *strong* and *weak* groups, with an *intermediate* group interpreted as a number of strong groups, each containing one element.

It is the intention that at all times the algorithm takes advantage of any possible progress in forming the application's global checkpoint. While strongly consistent checkpointing groups are in the process of synchronising to produce their consistent checkpoints, the checkpoints from weak groups are output. The algorithm then waits for strongly consistent groups to synchronise and immediately writes out the checkpoint from groups which have synchronised. This process continues until either all the groups have synchronised or until the migration is aborted.

```
if Checkpointing required.
then
    Prepare checkpoint for the module.
    if module is not weakly consistent.
    then
        if All other modules in this group have produced a checkpoint and are waiting.
        then
            Signal manager that a group has reached a consistent checkpoint.
        fi
        while Manager has not released module Wait.
    fi
fi
```

Figure 5.16: Checkpoint Consistency: Client

The implementation of the algorithm assumes that all members of a checkpointing group are resident on the same machine. This is because it is only intended to be used prior to the transmission of a checkpointing group from one host to another. If the implementation were to be extended such that a checkpointing group may be distributed over a number of machines, the possibility of a communication failure leading to the unavailability of a consistent checkpoint for the group must be considered.

5.12 Summary

The implementation of the Notus architecture and programming environment, described in this chapter, has concentrated on enabling an evaluation to be made of the key features the architecture, with:

- an RPC service which supports default traded handoff behaviours,
- a trader implementation supporting queries which include QoS constraints in an environment where there are a number of different routes to services, and
- an implementation of the Notus components, with a particular concentration on the formation of a consistent checkpoint for an application consisting of a number of asynchronous modules. This involved the implementation of an algorithm which ensures the correct synchronisation of modules for consistency, and a mechanism for the creation of the heterogeneous representation of a module's state.

This implementation is able to support an experimental evaluation of the traded handoff and migration services, in a programming environment which supports a large number of representative applications.

Chapter 6

Notus Evaluation

6.1 Introduction

The evaluation of the Notus implementation aims to demonstrate its performance in conjunction with specially written applications, as well as with a number of existing interactive applications which have been modified to use Notus.

In this evaluation, all walkstation mobility is simulated by applications running on static hosts. This was for two reasons: first, a wireless testbed which could have offered connectivity in both the local and wide areas, was unavailable to the author at the time of the implementation; second, that it is possible to fully exercise the implementation without a wireless component.

6.1.1 Experimental Program

The experimental work which was undertaken may be considered in three parts.

- In Section 6.3, the Null Server application is introduced to provide a basic evaluation of the performance of the implementation.
- Then, in Sections 6.4, 6.5 and 6.6, a suite of existing applications which have been adapted to use the Notus implementation, consisting of a video player, news reader, and shell, are evaluated. These applications are representative of those expected to operate in a walkstation environment, demonstrating applications for which it is important to minimise the disruption to the end user during migration and handoffs, and applications which transfer significant amounts of state during migration.
- Finally, in order to fully exercise all the options for consistency during application migration, a simple application which consists of a number of periodically activated modules is evaluated, in Section 6.7, against an analytical model.

These experiments demonstrate all the features of the architecture including: default traded handoffs for both client and server applications, a traded handoff with application involvement, application migration for a number of existing interactive applications, the interworking of the traded handoff protocol during migration so that the application's bindings are re-established as the migrated application is restarted, and all the checkpoint consistency options supported by the architecture.

6.1.2 Metrics

There are two measurements which can usefully be taken for migrating applications, namely disruption and latency. *Disruption* is the amount of time for which the migrating application is not able to do useful work because of the migration process. If the application is interactive, this is the time during which the end user would perceive the application as being unavailable. *Latency* is the total elapsed time from the initial migration request, to the completion of the transfer of application state to the target.

Disruption is perhaps the most important metric, since it is the time during which the end user is affected. The application is still available to the user during much of the measured latency time, since this includes the time spent in preparation for the migration[1]. However, latency is a useful measurement when considering the various checkpoint consistency algorithms and becomes an issue when the time available for completion of the migration is limited. For example, when a walkstation is quickly moving from a low-bandwidth to a high-bandwidth environment, it is important to complete any required application migration before the high-bandwidth connection is lost.

The principal metric for the traded handoff is the disruption to the end user. During the experimental program, disruption was measured both for the case where the client is executing on a walkstation and must reconnect to its server on the static network, and the case where the server is on the walkstation and must redirect its clients as it moves.

6.2 Platform

Unless stated otherwise, all measurements are the average of 100 runs and were taken on 150 MHz DEC ALPHA platforms running the Digital OSF1 3.2 UNIX operating system. These were connected via a shared 10 Mbit/s Ethernet and used TCP/IP. This configuration does not contain a wireless component, but is able to provide a useful demonstration of the implementation. Current Wireless LAN (WLAN) implementations are achieving the performance of this Ethernet, so measurements taken using this platform should be comparable to those from a WLAN.

6.2.1 System Metrics

Thread Creation	2.9 ± 0.5 ms
Bound Null RPC	1.4 ± 0.2 ms
Unbound Null RPC	6.9 ± 1.7 ms

Table 6.1: System Performance Metrics

Prior to conducting the experimental program, measurements were taken in order to give an indication of the performance of the communications environment. These measurements included the time to create a single thread, and cross machine Null Remote Procedure Call (RPC) times on both a bound and initially unbound interface[2]. The average of 5000 measurements is shown in Table 6.1.

[1]The time spent opening a stream, instantiating a target, during module synchronisation.
[2]Binding is described in Sections 2.4 and 5.3.

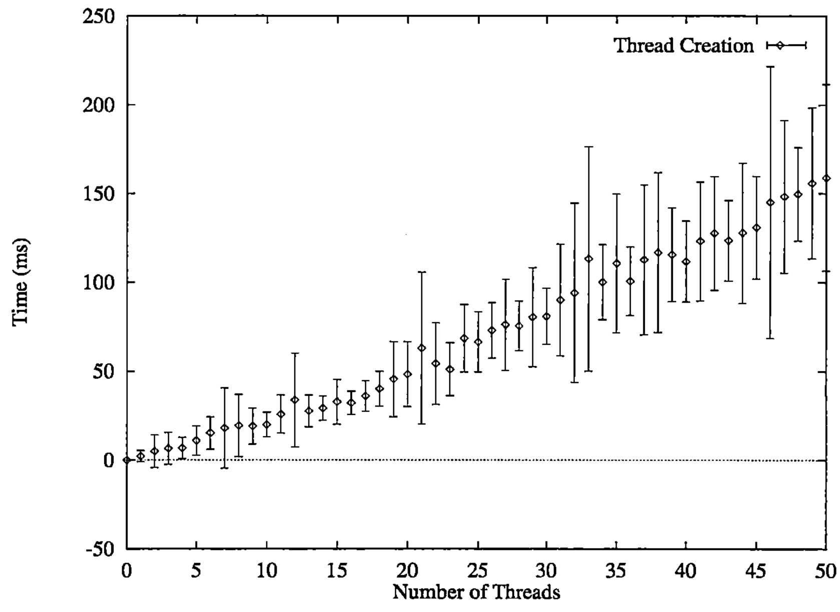
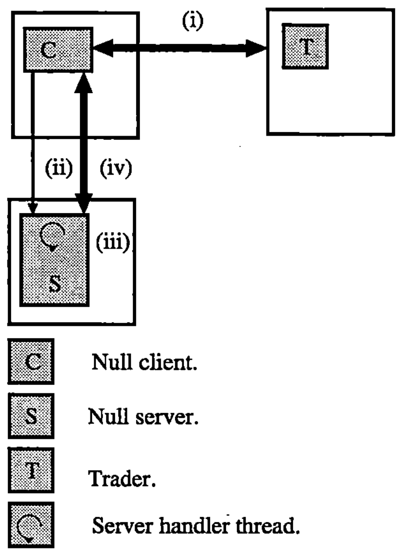


Figure 6.1: Thread Creation Time vs Number of Threads

Thread creation time is not linear and is subject to larger deviations when a number of threads are required to be created in a single batch. This is shown in Figure 6.1 and has ramifications for the Null Server experiment in Section 6.3.



(i) Import server offer from trader	1.7
(ii) Open network connection to server	2.3
(iii) Dedicate handler thread for binding	2.9
Unbound Null RPC time:	6.9 ms
(iv) First client invocation overlaps (iii)	1.4 ms

Figure 6.2: Unbound Null RPC Invocation Time

The Null RPC times are given for a Null server interface which contains one method called *Ping*. This takes an integer argument and returns the string 'Hello There!'. A marshalled invocation of *Ping* on a bound interface results in the transfer of 16 bytes from the client to the server and returns 32 bytes. The Null client's unbound invocation time, illustrated in Figure 6.2, is comprised of an RPC to a federated trader (i) to import an offer exported by the Null server, (ii) the time to open a network connection with the Null server interface, and (iii) the time taken on the server-side to create a dedicated handler thread for the binding. In the unbound case, the time elapsed during the invocation of *Ping* (iv) is masked by the time spent at the server creating the handler thread.

6.3 Null Server

The Null Server application was developed as a first test of the migration and traded handoff facilities of the Notus implementation. The application is configured to create a given number of independent modules, with each module implementing a single server which exports the Null interface.

6.3.1 Migration

The application was migrated to another host with all the server modules restored at the target machine. This resulted in the transfer of a checkpoint of

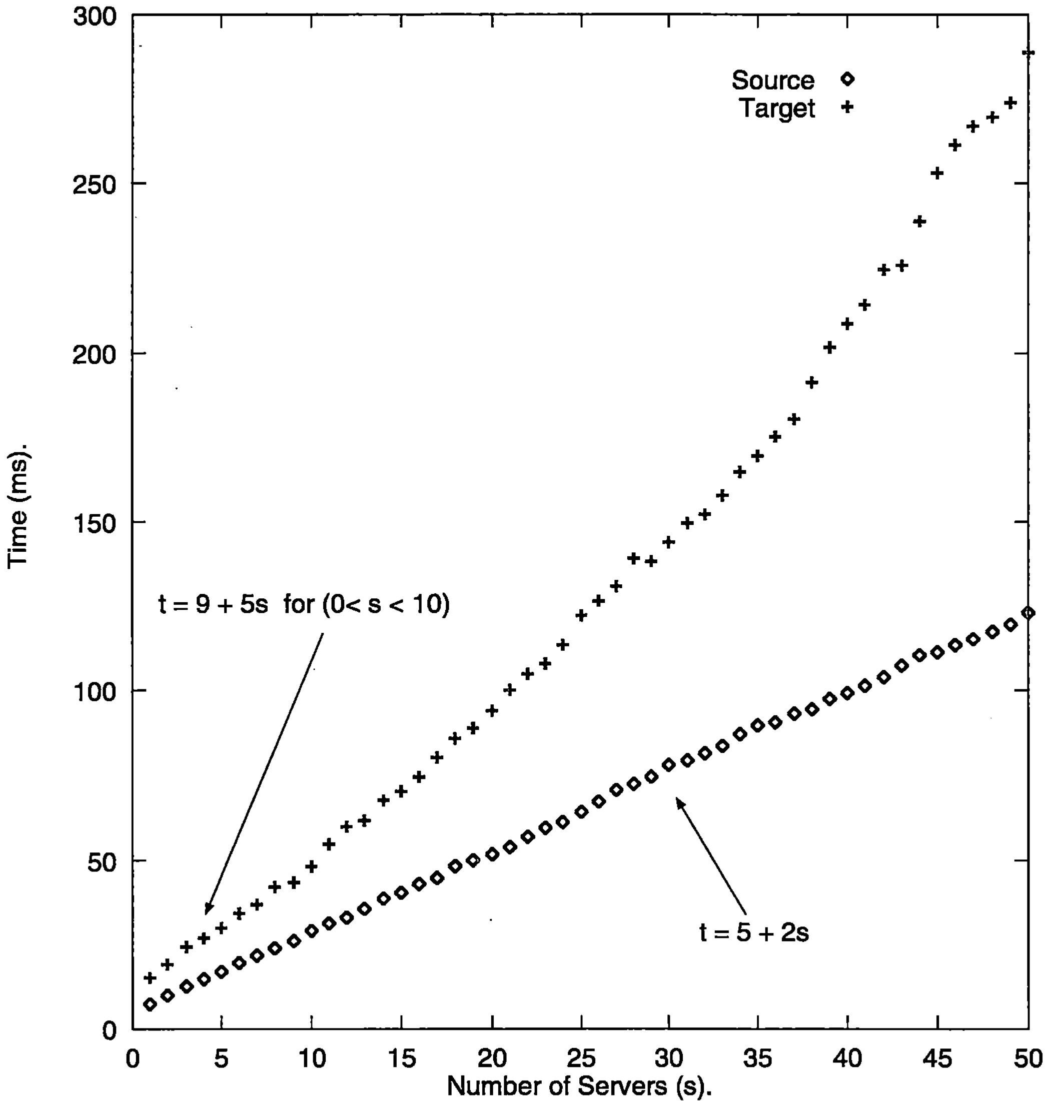


Figure 6.3: Server Migration Time vs Number of Servers

size 156 bytes per module. In Figure 6.3, the gradient of the *Source* plot gives the application's latency as 5ms, with an additional 2ms per server module.

The *Target* plot shows the elapsed time at the target, after accepting a connection from the source, until each module has restored from a received checkpoint. The required time was 9ms, with an additional 5ms per module for small numbers of modules. For larger numbers of modules, the Target plot is non-linear as a result of the cost of creating larger numbers of threads. The disruption to the application can be estimated from the Target plot, since this also includes the time spent preparing the checkpoint at the source.

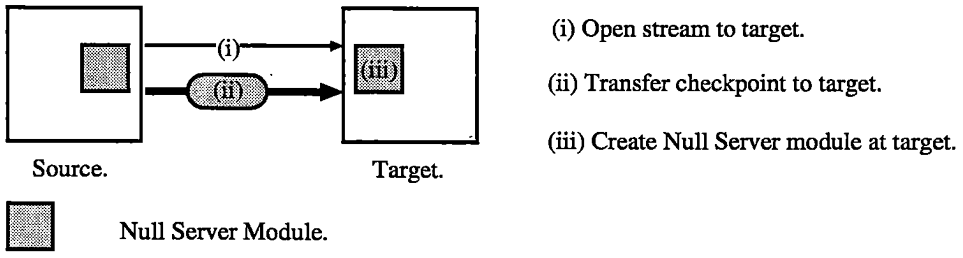

(i)
(ii)
(iii)
Source.
Target.
Null Server Module.
(i) Open stream to target.
(ii) Transfer checkpoint to target.
(iii) Create Null Server module at target.

Figure 6.4: A Migrating Null Server Module

For one module ($s = 1$) the components of the time measured at the Target are shown in Table 6.2 and illustrated in Figure 6.4. Here, *Source* represents the time spent at the target, once the stream has been opened (i) while waiting for the source to prepare and transmit the checkpoint. *Read* represents the time taken to transfer the checkpoint from the source to the target (ii). *Thread* represents the time taken to create the server module at the target (iii), and *Offer* represents the time taken for the Null server to initialise at the target (iv).

Target Restore Time (ms)			
Source	Read	Thread	Offer
7.4	1.8	2.9	2.3

Table 6.2: Disruption for a Single Null Server Module During Migration

It was also found that as the number of modules increases in the range ($0 < s < 10$), the *Source* and *Read* times remain constant, while the *Thread* and *Offer* times both increase as a multiple of the number of modules.

These results have demonstrated application migration for a simple application. When considering a single Null Server module, the overheads of migration above the time spent at the source preparing the checkpoint are similar to those required to establish an RPC binding.

6.3.2 Traded Handoffs

The Null Server application was then configured so that a single server instance was created which requested the *Flat* subcontract when exporting an offer. Flat subcontracts were discussed in Section 5.3 and provide default traded handoff semantics where, on a traded handoff, connections are rebuilt to their original end-points without the application being aware of mobility. There are two (simulated) cases to consider: first, where the client is executing on a walkstation and must reconnect to its server on the static network; second, where the server is on the walkstation and must redirect its clients as the walkstation moves.

In the first case, the client-side Flat subcontract exports an instance of the Handoff Interface (HDI) which is invoked during the traded handoff (see Sections 4.5.3 and 5.8). The average time taken for the subcontract to *Open* a new connection to the server in the background, was measured as 2.3ms. This is essentially the time taken to create a binding given an interface reference and is a component of the unbound RPC invocation time shown in Figure 6.2. Since there is no client state to be transferred, the *Unmount* and *Mount* phases, and hence the disruption to the client were unmeasurable using the experimental platform.

In this simulated environment, both the old and new connections are simultaneously available and there is little disruption to the client. This would not always be the case, and might even depend upon the direction of the handoff. For example, when moving from a Mobile Radio Network (MRN) to a WLAN environment, it might be expected that offers made using the MRN would be available from the WLAN. However, moving in the other direction, a WLAN offer would be immediately unavailable as soon as the walkstation moved into the MRN environment.

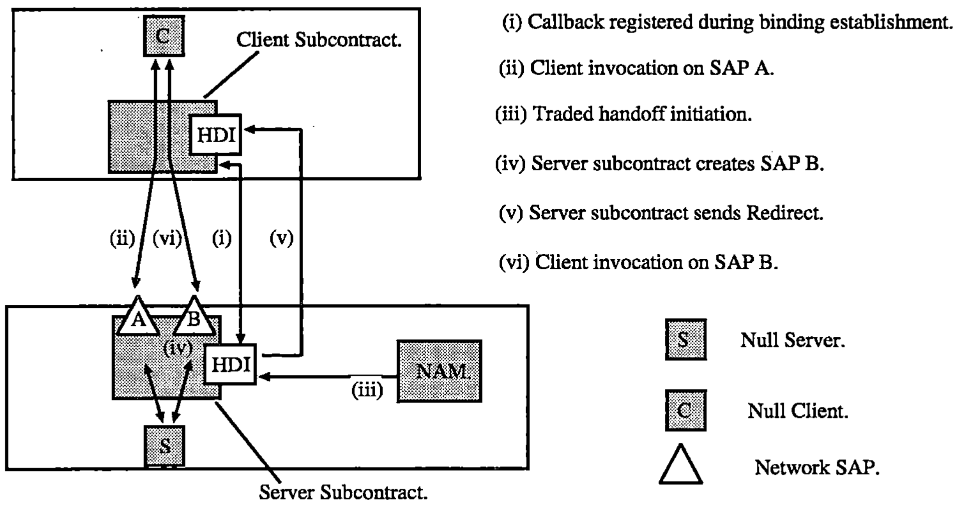

C
Client Subcontract.
HDI
(ii)
(vi)
(i)
(v)
A
B
(iv)
HDI
NAM
(iii)
S
Server Subcontract.
(i) Callback registered during binding establishment.
(ii) Client invocation on SAP A.
(iii) Traded handoff initiation.
(iv) Server subcontract creates SAP B.
(v) Server subcontract sends Redirect.
(vi) Client invocation on SAP B.
S Null Server.
C Null Client.
Network SAP.

Figure 6.5: A Default Traded Handoff for the Null Server

The second case of a mobile server is illustrated in Figure 6.5. The server's subcontract exports an instance of the HDI at the time the Null RPC server offer is created. When the client establishes a binding using the offer, the requirement to use the Flat subcontract is detected. During binding establishment, the client and server-side subcontracts initialise, and the interface references of the client and server-side HDI are exchanged. The server's *Callback* method is invoked (i) to register and stash the client's HDI interface reference, as described in Section 4.5.5. At this point (ii) client invocations are made through Service Access Point (SAP) A.

When the (simulated) walkstation moves, the server is required to perform a traded handoff and is informed by the Notus Application Manager (NAM) (iii). It creates (iv) the new SAP B and exports a modified offer to the trader. The server then binds to the stashed HDI of each of its clients, instructing each client (v) to *Redirect* and use the new SAP. The traded handoff is completed once each client has reconnected to the new server offer. After the traded handoff, client invocations (vi) will be made through B.

In the experiment, a single client was connected to the Null server. The server was then requested to perform a traded handoff and the time taken for a new server offer to be *created* was measured, together with the time taken to *inform* the client of the new offer and to *destroy* the old offer. At the client-side, the *disruption* time during the redirection was measured. These results are shown in Table 6.3. .

Server Handoff (ms)			Client Disruption (ms)
Create	Inform	Destroy	
15.2	3.0	3.1	2.1

Table 6.3: Traded Handoff Measurements for the Null Server

The results demonstrate the default traded handoff facilities of the Notus implementation, which require no application modifications. By performing much of the handoff in advance, the disruption experienced by the application is comparable to other implementations which perform mobile handoffs within the network. For example, in Section 3.1.3 mobile handoffs performed within an Asynchronous Transfer Mode (ATM) switch are simulated and show a 7ms disruption to the client.

6.4 Video Player

A Public domain MPEG player [Rowe93] was obtained and modified to take input from a network video server, rather than a file. This application was intended to represent a scenario where a video player running on a walkstation receives, decodes, and displays compressed video from a replicated network server. As the walkstation moves, the client application is required to take advantage of replicated servers and the changing network environment by performing traded handoffs, and rebuilding its session with a suitable video server. If the walkstation moves into a WLAN environment, it could be appropriate to migrate the client to a static compute server, in the process performing a traded handoff and ensuring that the video stream is rebuilt as the application is restored. This application was chosen because it is important to minimise any disruption to the playing out of the video stream.

The components and normal operation of the video player application are first described in Section 6.4.1. Then Section 6.4.2 describes the modifications made to support the migration of the video player client and presents results obtained by measuring the migration time. The migration of the video player interworks with the traded handoff mechanism and the migration results include a traded handoff. However as an illustration of an application which is fully involved in the traded handoff, Section 6.4.3 describes the traded handoff for the video player client in isolation.

6.4.1 Components and Operation

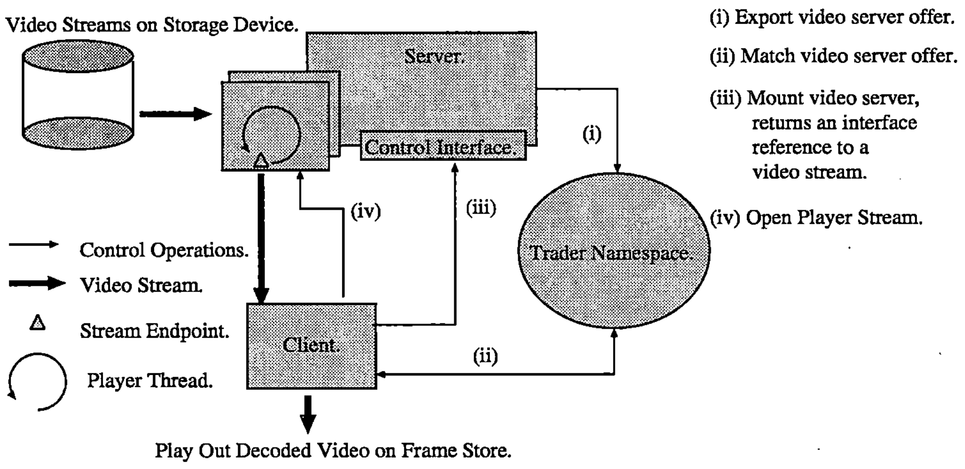

Video Streams on Storage Device.
Server.
Control Interface.
(i)
(iv)
(iii)
Trader Namespace.
Control Operations.
Video Stream.
Stream Endpoint.
Player Thread.
Client.
(ii)
Play Out Decoded Video on Frame Store.
(i) Export video server offer.
(ii) Match video server offer.
(iii) Mount video server, returns an interface reference to a video stream.
(iv) Open Player Stream.

Figure 6.6: Mounting a Video Server

The application consists of two modules shown in Figure 6.6 – a server which plays streams of MPEG compressed video out from a storage device and a client which decodes and displays the video. The server accepts requests on a control interface to play out streams. It exports an offer in a trader (i) consisting of an interface reference to the control interface and the properties of the video streams which it is able to play out. The client queries the trader (ii) and imports. an appropriate offer. The client then binds to the server's interface and invokes a method to *Mount* the required stream (iii). This causes the server to dedicate a *Player* thread to the playing out of the stream. The *Player* accepts a network connection from the client, and an interface reference to this end-point is returned to the client as the result of the *Mount.* The client is then able to connect to the *Player* (iv) and receives the video stream. The bandwidth requirement of the video clips used was approximately 226 Kbit/s.

6.4.2 Migration

The MPEG [ISO91] compression scheme uses an interframe coding, with frames of three types, Intra Picture (I), Predicted Picture (P), and Bidirectional Picture (B). An Intra Picture is directly decoded and displayed, providing a random access point within the stream. A Predicted Picture is coded with reference to a previous frame, and a Bidirectional Picture is coded possibly with reference to both a previous and a future picture.

If the restoring video player were required to correctly decode and display the next frame in the stream, it would have been necessary to transfer as part of the application's state, a number of Predicted frames, together with the previous and future Intra Picture frames. Since Intra Picture frames tend to appear frequently in the stream[3], it was decided that during migration, the client should skip forward through the stream to the next Intra Picture frame. From this point, the stream is decoded without reference to any frames held in the source application.

The application is structured around a tight loop, each iteration decodes and displays a single frame of video, requesting new data from the video server as required. It was sufficient to place a single checkpoint within this loop, at a point where a complete frame had been decoded and displayed. This required the annotation of a single function using the Notus Language Extensions (NLE). Other state identified for checkpointing was limited to the video's title, offset, aspect ratio, bit rate, and the thread call chain. In the experiments, the size of the checkpoint was 240 bytes.

Since the size of the transferred checkpoint was small, the application was modified to produce a checkpoint while processing each frame of video with no noticeable impact on performance. This enabled the weak checkpoint consistency option from the Notus implementation to be used, thereby improving the latency of migration. When requested to migrate, the NAM is able to immediately access the most recently produced checkpoint.

During migration, the client application is moved from one host to another. In order to ensure the rebuilding of the video stream, a traded handoff is in-

[3]Although this depends upon the encoding parameters used, but is typically about every 8 frames.

terleaved with migration. The source application cleanly unmounts its server and the restoring target application mounts a new server at the correct offset in the video stream. The client application is configured so that during a traded handoff, a query is made for a (different) server which supports the required video stream.

The time taken to migrate the application is shown in Table 6.4 below. After the source application is disrupted, it must first *Unmount* its server connection, then *Read* in the checkpoint from the Source. *Restore* is the time taken from when the checkpoint has been read in, to when the first frame of video is displayed. This time has been further broken down into the time taken in the *Open* phase of the traded handoff protocol to import and establish a binding with a suitable Video Server offer[4], the time taken to *Mount* the server, and to *Connect* to a new stream of video. *Display* is the time taken to reset the user interface and could have been optimised out of the process through pre-initialisation.

Video Player Disruption Time (ms)					
Source	Target				
Unmount	Read	Restore			
		Open	Mount	Connect	Display
		7.0	11.3	2.3	9.6
1.4	1.8	30.2			
33.4					

Table 6.4: Video Player Disruption During Migration

These results show that it is possible to migrate the video player with the end user disrupted and unable to view the video stream, for a time which is comparable to the loss of a single frame of video. For this application the goals of migration in a mobile environment, particularly heterogeneity and low-latency are both achieved.

Examining the components of the migration time, it appears that much of the time is spent mounting the new video server offer. If the Flat subcontract had been used, which would have re-established the binding to the original server, the *Open* time would have been reduced to the 2.3ms of the Null Client in Section 6.3.2. The largest component of the *Mount* time is the new server opening and seeking through the video file. If the same server were used, this time would also be significantly reduced, since it is likely that a file handle at the correct place would be available. Further optimisations to the traded handoff protocol are suggested in Section 8.1.

[4]This is equivalent to the unbound Null RPC shown in Figure 6.2.

6.4.3 Traded Handoffs

The video player application was then instructed to perform a traded handoff of its connection with the server, while in the process of playing out a video stream. The time taken was measured and shown in the Table 6.5.

Video Player Handoff Time (ms)			
Open	Unmount	Mount	Connect
7.0	1.4	11.3	2.3

Table 6.5: Video Player Handoff Results

Here again, *Open* is the time taken to import a new server offer from the trader and to establish a binding. *Unmount* is the time taken by the client to close down its connection with the old server, while blocking any attempt to read the video stream. The new server is then *Mounted*, the client indicating the correct point in the stream at which to play out, and a *Connection* is made to the new stream. It should be noticed that these results are a component of the migration results for the video player, shown in Table 6.4.

During the traded handoff, the client must transfer its session to the server and so makes use of a custom HDI, rather than using a default. This illustrates an example application where client-side synchronisation during the handoff involves more than just swapping a network SAP. Here, it is also required to block operations on the video stream, which is a completely separate entity to the video server control interface.

In this example, the *Open* phase of the traded handoff is performed in the background and the client is disrupted for only the time taken to *Unmount* and *Mount* a server, and to *Connect* to a new video stream (about 15ms). This corresponds to a time which is significantly less than the play out time for a frame of video.

If the traded handoff were being made in an environment where network latencies were much greater, then an estimation of the expected performance might be made based upon these results. For example, where a traded handoff is being made because of the availability of local services, it would be expected that the time required to *Unmount* would reflect the additional latency from the invocation of the existing (now remote) server, while the *Mount* of the new server would be local and hence a low-latency operation. In the limit, the total handoff time would be expected to be dominated by the latency of the *Unmount* operation. However, this time would be quickly recouped by the use of local services.

When comparing the traded handoff time against schemes which perform handoffs at lower levels in the network, it should be noted that for this stream based application, it would be likely that there are large amounts of buffered state requiring transfer between base stations during a handoff. The results described in Section 3.1.1 show transport layer handoffs requiring times of about 300 ms in the presence of about 4 Kbytes of buffering.

6.5 A Migrating News Reader

As a demonstration of the migration of a desk-top application with significant state, a public domain news reader [Sentovich88] was modified to be migratable using NLE. The application is structured around a state based model, with the behaviour of the application depending upon the application's current mode and events received through the user interface.

For each available news group, the unmodified news reader application records information regarding the articles which the user has read into a file when the application exits. This action makes it possible for subsequent instances of the application to display only the unread articles for the user. This existing facility was incorporated into the migration process to transfer the application state directly to the target (using the checkpoint stream rather than a file). The other required state, including: the current and previous modes of the application, the current news group, and article being read – were described using the Middl Interface Definition Language (IDL) and automatically transferred during migration.

In order for the migrated application to remain consistent with received interface events, and because of the performance impact of periodically producing the checkpoint, strong checkpoint consistency was chosen.

It was found that the time taken to produce, transfer, and parse the (16Kbyte) description of the read articles for each news group, and to resolve this information against the available articles provided through the new connection to the news server, dominated the disruption time experienced by the application, and amounted to the order of 2 seconds.

Although this disruption time is tolerable, it was thought that an order of magnitude reduction could be achieved both by transferring only the changes made by the source application to the article state, and also by the target application not re-querying the news server during restoration. The modifications made to the news reader program for migration were minimal, requiring only a superficial understanding of the structure of the program. Extensive modifications to the program would have been required to implement the above improvements and were not undertaken.

6.6 A Migrating Shell

Work by Ashton has involved modifying a public domain shell [Joy80] in order to explicitly identify state for migration [Ashton96]. This work was motivated by a requirement for load balancing in a cluster of homogeneous computers. During migration, an instance of the shell is created on the target host, with a pipe arranged for communication with the source. Various ad-hoc methods (including writing state to a file) were used to transfer state between the two instances, until the target is able to take over execution from the source. Ashton observed that much of the work in modifying the application was spent in ensuring the robustness of the mechanism used for transferring state and restoring the target application.

Ashton's migrating shell was adapted by the author to use the Notus implementation. It was found that the required modifications to the application for migration were greatly reduced in comparison to those made by Ashton. In particular, Notus transparently instantiates and connects to the target application. This left only a consideration of the state to be transferred:

- the values of all the shell variables which had been defined.
- the values of any aliases, which are used to introduce new keywords to the shell.
- the history of commands executed by the shell, used for example by a user of the target shell, who might request the last command executed by the source shell to be re-executed at the target.
- the value of the path, which is a list of directories in the file system to be searched to find a command.

In the case of the shell's history, there already exists a mechanism to store the history in a file when the shell exits, and a new instance of the shell is then able to load the history as it starts. This existing facility was used to transfer the history directly to the target. The implementation allowed this transfer to be made directly on the checkpoint stream, thus avoiding use of the intermediate files of Ashton's migrating shell.

As well as demonstrating another migrating desk-top application with significant state, this example gives anecdotal evidence that use of the Notus implementation reduces the work required in adapting applications for migration and ensures that all participating applications can be uniformly controlled by one manager.

It was found that the size of the checkpoint, for the migrating shell, depended upon the particular instance of the shell, but was typically around 6Kbytes, giving rise to a measured disruption time of 87ms during migration. This disruption time is half that of Ashton's migrating shell, because even though the same state is transferred, there is no requirement to use various intermediate files to store the state. Additionally, this state was correctly marshalled during migration, enabling migration between heterogeneous computers.

6.7 Checkpoint Consistency

The following experiments are intended to evaluate the tradeoffs associated with the various checkpoint consistency options of the Notus implementation. The experiments are focussed on the formation of a globally consistent checkpoint within the migrating source application. Consequently, there was no requirement to consider a target application. To reduce experimental complexity, the source application was configured to store its checkpoint in a file. The main difference in measurements taken using this configuration to the previous measurements is that the time required to open the file using the Sun Network File System (NFS) [Sandberg85] was much greater than would have been required to open a TCP/IP connection between the source and target.

```
while(true)
{
  wait(random(T));
  if(iteration == 100)
    CKPT SCHEDULE;
  CKPT POINT;
}
```

Figure 6.7: Consistency Test Module: Main Loop

An application was developed to test the algorithm, which aims to form a consistent checkpoint from a group of modules. The application initialises a number of modules, each of which performs the loop shown in Figure 6.7. Every iteration of the loop blocks the module for a random, discrete interval of up to a maximum of T ms. An iteration also contains a checkpoint, marked by the macro POINT, where the module can synchronise during the checkpoint consistency algorithm. After a fixed number of iterations, one module, termed the *initiator*, uses the SCHEDULE macro to initiate a consistent checkpoint for all of the application's modules.

The NAM performs the algorithm described in Section 5.11, to form the checkpoint according to the consistency requirements specified by the application. For example, if strong checkpoint consistency were specified and all the modules placed in a single checkpointing group, the NAM would block each module as it reaches its next checkpoint, until all the modules have reached this point. Before presenting experimental results, the application is first modelled analytically for the *strong* and *intermediate* consistency cases. This model is intended to provide a benchmark for the performance of the implementation.

6.7.1 Analytical Model

The time taken to reach a consistent checkpoint is the elapsed time between the initiator I starting the checkpoint process (by reaching the `SCHEDULE` macro), and every other module reaching a checkpoint (at the macro `POINT`). If there are only two modules in the checkpointing group (I and A), the probability of A reaching its checkpoint at any time t after I has initiated checkpointing can be determined. Generally, checkpoints can occur at any time, however since this application was implemented using a discrete time model, the analysis is somewhat simplified by also assuming discrete intervals of time.

If an *event* is said to have occurred when a module reaches a checkpoint (with a maximum period T), then Figure 6.8 shows three events: E_1 at time e_1 when A reaches a checkpoint, E_2 at time e_2 when I initiates checkpointing, and E_3 at time e_3 when A reaches its next checkpoint. The uniformly distributed, random delay means that the probability of an independent event at any time is $1/T$. Hence the probability of E_3 given E_1 is $1/T^2$.

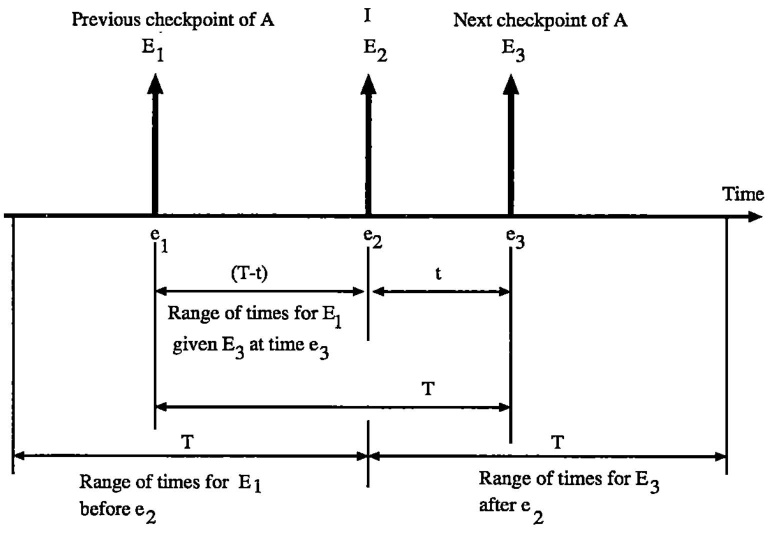

Previous checkpoint of A
E1
I
E2
Next checkpoint of A
E3
Time
e1
e2
e3
(T-t)
t
Range of times for E1 given E3 at time e3
T
T
T
Range of times for E1 before e2
Range of times for E3 after e2

Figure 6.8: A Model for Strong Consistency Checkpoints

In general, it is possible for E_1 to occur at any time within the period T before E_2, and for E_3 to occur at any time within the next period T after E_2. However, if E_3 is constrained to occur at the given time t such that $e_3 - e_2 = t$, then E_1 is also constrained such that $e_3 - e_1 \leq T$, hence $e_2 - e_1 \leq T - t$.

Hence, there are $T - t$ discrete possibilities for the occurrence of event E_1 given E_3 at a time t after E_2 and the probability $\alpha(t)$ of such an event E_3 is thus:

$$\alpha(t) = \frac{T-t}{T^2}$$

The probability β, of an event in some interval $0 \leq t \leq t'$ is thus:

$$\beta(t') = \sum_{t=0}^{t'} \frac{T-t}{T^2} \tag{6.1}$$

Given $N+1$ checkpointing modules (including the Initiator I), the probability γ, of all N events from synchronising modules occurring in the interval $0 \leq t \leq t'$ after I is:

$$\gamma(t') = \beta(t')^N$$

Once all N events have occurred, the modules have synchronised and a consistent checkpoint can be taken. Hence, the average time $\bar{t}$ to synchronise the modules can be found from the normalised probability of all the events occurring within the interval $0 \leq t \leq \bar{t}$, and is given by:

$$\frac{\gamma(\bar{t})}{\gamma(T)} = 1/2$$

$$\Rightarrow \beta(\bar{t})^N = \gamma(T)/2$$

$$\Rightarrow \beta(\bar{t}) = \beta(T)/2^{1/N} \tag{6.2}$$

The sum of the arithmetic progression for $\beta(t')$ in Equation 6.1 gives:

$$\beta(t') = \sum_{t=0}^{t'} \frac{T-t}{T^2}$$

$$\Rightarrow 1/T \sum_{t=0}^{t'} 1 - 1/T^2 \sum_{t=0}^{t'} t$$

$$\Rightarrow \frac{t'+1}{T} - 1/T^2 \frac{(t'+1)t'}{2}$$

$$\beta(t') = -t'^2/2T^2 + t'(1/T - 1/2T^2) + 1/T \tag{6.3}$$

The Equations 6.2 and 6.3 (substituting $\bar{t}$ for t'), can be combined forming a quadratic in $\bar{t}$:

$$-\bar{t}^2/2T^2 + \bar{t}(1/T - 1/2T^2) + 1/T - \beta(T)/2^{1/N} = 0$$

and solving for $\bar{t}$ gives:

$$\bar{t} = (T + 1/2) \pm \sqrt{T^2 + 3T + 1/4 - 2T^2\beta(T)/2^{1/N}}$$

In the application $T = 250$, hence $\beta(T) = 0.502$. For these values, the roots of the quadratic are approximated to:

$$\bar{t} \approx T(1 \pm \sqrt{1 - 2^{-1/N}})$$

Since synchronisation occurs within the period T, the lesser root gives the average time $\bar{t}$ taken to reach a consistent checkpoint for the $N + 1$ strongly consistent checkpointing modules.

$$\bar{t} \approx T(1 - \sqrt{1 - 2^{-1/N}}) \tag{6.4}$$

6.7.2 Measured Latency

The plots shown in Figure 6.9 compare the three consistency schemes, *strong*, *intermediate*, and *weak*, introduced previously in Section 4.7, together with the points *Expected* for strong and intermediate consistency derived from Equation 6.4 in the analytical model when T = 250ms. The plots show latency against the number of checkpointing modules, encompassing the total time elapsed from the initial request to form the application's checkpoint, to the completion of writing out the checkpoint to a file. The size of the checkpoint produced by each module was 176 bytes.

Checkpointing using weak consistency allows the NAM to take the most recent checkpoint from each module. This requires no synchronisation with the module, except where there is contention if a module is updating its stored checkpoint at the time the NAM requires access. The latency for weak checkpointing in this experiment was dominated by the NFS file open operation, which took on average 49ms. The additional time required for larger numbers of modules resulted from the time required to write out each module's checkpoint (approximately 1ms per module). This time would be expected to be more significant as the size of the checkpoint was increased, as would the cost of periodically creating unnecessary checkpoints. The cost of checkpoint creation was unmeasurable in the experiment, because of the small checkpoint size.

The *strong* plot (Figure 6.9) shows the latency experienced when all modules are placed in a single, strongly consistent group. The *intermediate* plot shows the modules placed in a single group of intermediate consistency[5]. For small numbers of modules in both the strong and intermediate cases, the time spent opening the checkpoint file results in the measured latencies being greater than

[5]Which is equivalent to placing the modules in strongly consistent groups where each group contains one member.

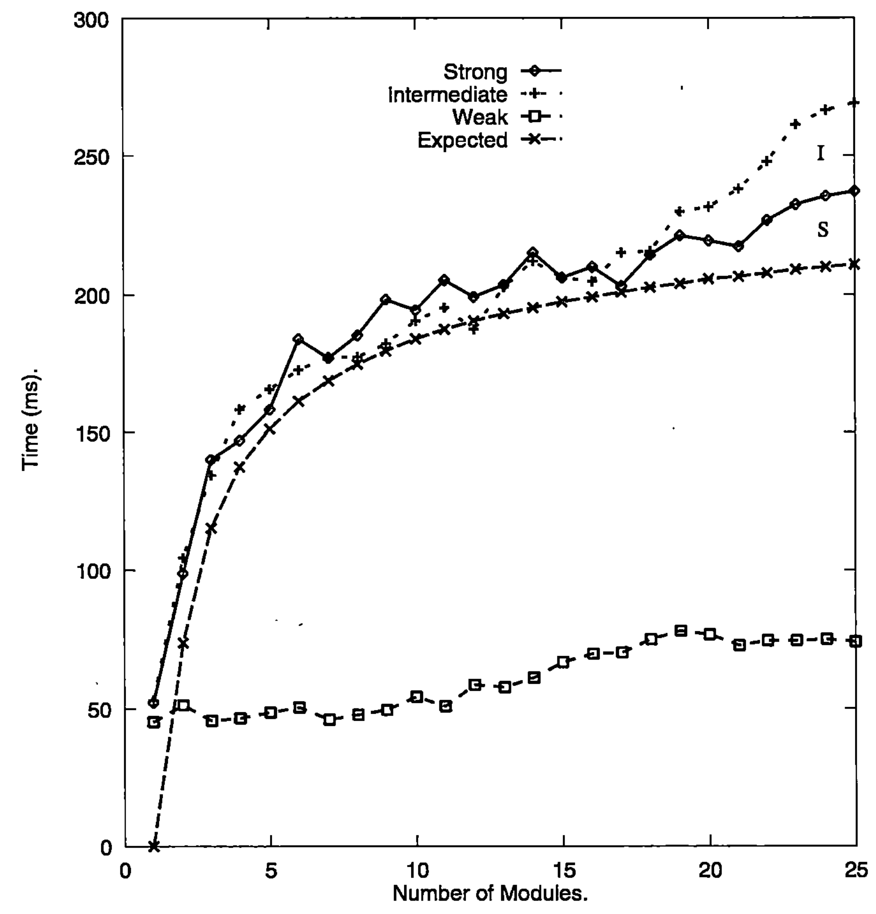


Figure 6.9: Latency vs Number of Modules

the *Expected* latency. As the number of modules is increased, the file is opened while the modules are synchronising, and both plots more closely match the *Expected plot.* For larger numbers of modules, the plots diverge to a much greater degree. Here, the additional time (S) in the strong plot, is caused by the increasing overhead from writing out the global checkpoint once all modules have synchronised. During intermediate consistency checkpointing, each module writes out its checkpoint immediately upon synchronising with the NAM. However, for small checkpoint sizes, any saving from this is offset by the time (I) required for additional locking, signalling, and scheduling between the manager and each group.

It is interesting to note that for intermediate numbers of modules, both the strong and intermediate consistency plots show that the time spent performing the NFS file open operation does not contribute to the overall latency. This is because the modules start to synchronise before the file is opened.

6.7.3 Measured Disruption

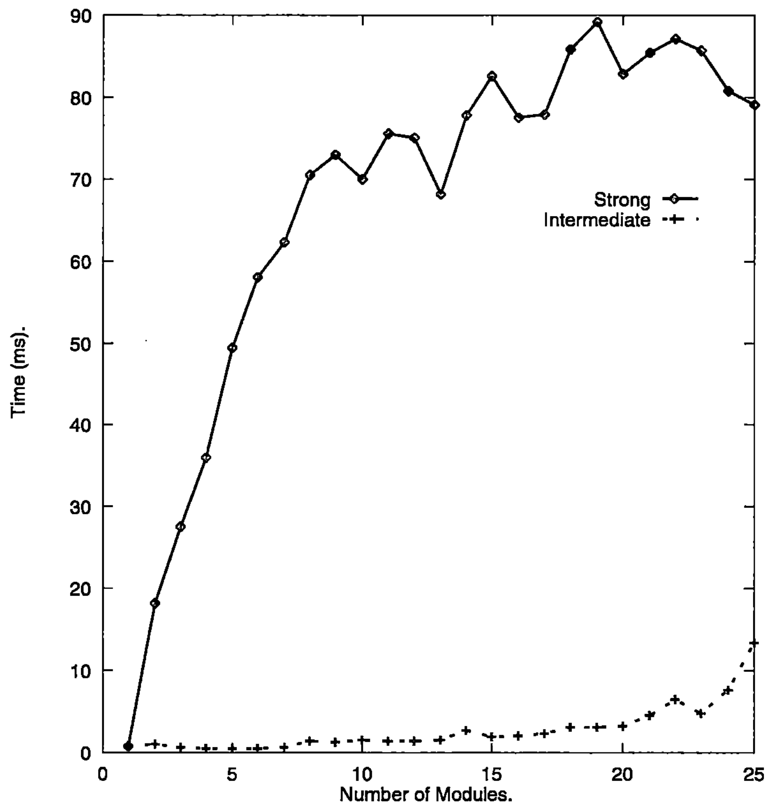


Figure 6.10: Disruption vs Number of Modules

The plot shown in Figure 6.10 gives the average disruption experienced by each module during the checkpointing of 20 modules.

The results show that the disruption experienced by modules during checkpointing is reduced when using intermediate rather than strong consistency. For strong consistency, a module is disrupted from the time that it synchronises until all other modules in the checkpointing group have synchronised. For intermediate consistency, a module continues execution once it has synchronised with the NAM. Where there are small numbers of modules, each module is disrupted only for the time taken for the NAM to respond to the module's signal, write out the checkpoint, and release the module. As the number of modules increases, there is a greater chance of a module synchronising while the NAM is in the process of dealing with another. This would result in an additional delay to the newly synchronising module. As expected, the intermediate plot appears flat until high numbers of modules are reached.

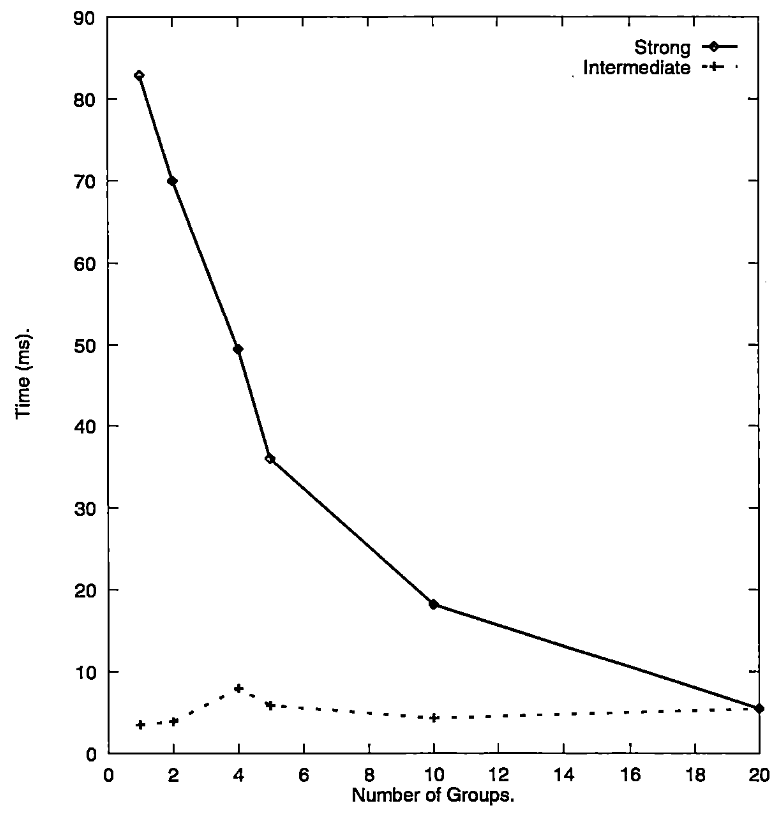


Figure 6.11: Disruption vs Groupings of 20 Modules

Figure 6.11 shows how the ability to group strongly checkpointing modules enables an application to minimise the disruption experienced by each module. The plot shows the disruption for various groupings of 20 checkpointing modules. When the number of strongly consistent groups equals the number of modules, the algorithm performs as for an intermediate consistent grouping.

6.7.4 Larger Checkpoints

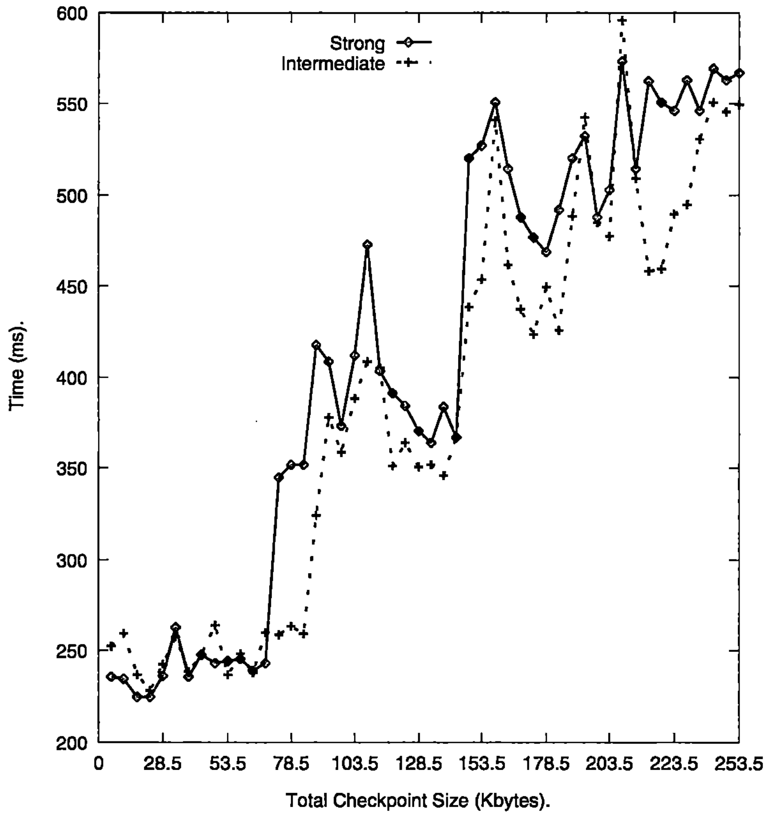


Figure 6.12: Latency vs Checkpoint Size

Finally, the experiment was modified to evaluate the increase in latencies as checkpoint sizes were increased. Each module was configured to add extra data to its nominal checkpoint of 179 bytes, the extra data varying in steps of 256 bytes from 0 to 12.5 Kbytes. Figure 6.12 shows latency plotted for 20 modules in the *strong* and *intermediate* cases against the size of the total checkpoint produced by all 20 modules. These results show that the implementation does not diverge significantly from the *Expected* plot (208ms) until the size of the checkpoint reaches about 73Kbytes. After this point, the latency becomes dominated by the time required to write out the checkpoints using NFS. The latencies then appear to increase significantly as the checkpoint size is increased through steps of 73Kbytes. This is probably a scheduling artifact, since the algorithm used by NFS batches small requests up to 8 Kbytes.

6.8 Conclusions

During the modification of a number of existing applications to use Notus, it was found that traded handoff support was easily accommodated. The required functionality was generally already contained within the application, for example to handle network errors, and it was usually only necessary to encapsulate the functions in an interface.

For application migration, programmer assistance was required to identify the state for transfer. It is likely that this task would have been eased had the applications been originally written using Notus modules, particularly because of the explicit state in module interfaces and the requirement for no global variables. However, all the applications considered in this chapter were well structured and amenable to modification. The general steps which should be taken to adapt the applications for migration are as follows.

- The application is compiled with a NAM and NLE added which will correctly initialise the Manager during application startup. The application's command line arguments must be processed to determine whether the application is a target for migration and if so, the NAM informed. At this point it is important to ensure that the application is able to coexist with the Notus implementation environment, in particular the trader and RPC service.
- The structure of the application is examined for points at which migration may take place. At these points, there should be no outstanding transactions and few partial calculations in progress. For example, in the case of the video player, it would have been inappropriate to attempt migration during the decoding of a video frame.
- Once migration points have been identified, function calls are traced to these points, local function state which requires transfer identified, and NLE annotations added.
- The application's global state which must be transferred is identified and described using Middl. In general, attempts should be made to determine the minimal amount of state to be transferred. Much state, such as that associated with the user interface, for example, can be recreated at the target.
- Finally, it must be ensured that all the established bindings use either the default traded handoff or directly respond to traded handoff requests.

For the applications which were adapted, there was no requirement for a deep understanding of the application's internal structure or for extensive modifications. It was noticed in the case of the news reader, that migration through application modifications at a high level of abstraction resulted in large amounts of state being transferred and hence, a significant disruption during migration. In the case of the video player, migration considerations at a high level of abstraction resulted in inconsistencies in the video stream, but reduced the state required for transfer during migration.

There appear to be two trade-offs between migration at different levels of abstraction: first, if the behaviour of the application is approximated at high levels of abstraction, inconsistencies after migration may result, but the amount of state requiring transfer can be reduced; second, without approximating the application's behaviour, migration considerations at a lower level of abstraction also reduces the amount of state required for transfer, since the state is specified at a fine granularity. However, this requires a greater understanding of the application and more effort in its modification. It also appears that at lower and lower levels, the potential reduction in state which is transferred does not outweigh the effort required to achieve that reduction.

It was also found that care should be taken in the selection of the appropriate checkpoint consistency options. The *strong* and *intermediate* checkpoint consistency options are suitable for larger checkpoint sizes, and give applications direct access to the checkpoint stream between the source and target. The synchronisation required to form a consistent checkpoint means that applications which consist of many active modules are likely to experience considerable disruption, especially if some of the modules are blocked for other reasons. Applications which use the *weak* consistency option maintain a small, continually available checkpoint. This reduces the disruption to the application, provided that the application is able to tolerate the inconsistencies caused by the transfer of checkpoint state which is not current.

The video player application was suited to the weak consistency option, since its checkpoint size was small and it was required to minimise the disruption experienced during migration. Also the application was tolerant to slight inconsistencies in the video stream. Conversely, the news reader and shell applications are both more suited to the *strong* consistency option, since their checkpoint sizes are larger and access to the checkpoint stream allows use to be made of the application's existing state saving mechanisms.

In summary, this chapter has evaluated a suite of applications using the Notus implementation. The results show that it is possible to migrate interactive applications in a heterogeneous manner and with little disruption. Traded handoffs are easily accommodated by existing applications and impose little disruption on the end user.

Chapter 7

Related Work

This chapter compares other work which is related to the Notus architecture. Section 7.1 describes work providing support for handoffs in a distributed programming environment, which is similar to the default traded handoff behaviour (described in Section 4.5.4) of the Notus Remote Procedure Call (RPC) service. Section 7.2 describes other work which is relevant or has addressed some of the requirements of application migration in a mobile environment.

7.1 Mobile RPC

The **M-RPC** [Bakre95b] architecture is an extension to the Sun RPC service, which supports binding re-establishment for mobile client applications in a manner similar to the Notus traded handoff. *Agents* on the static network perform an indirection of RPC calls from a walkstation. These agents collaborate, performing mobile handoffs by transparently creating new bindings to servers. This transparency removes the requirement, of the Notus traded handoff protocol, to synchronise the handoff process with the application's threads. However, all servers which use the M-RPC architecture must be stateless.

It was suggested that future extensions of the M-RPC architecture would support stateful servers. If this were the case, synchronisation with either the client or the server would have to be introduced. The M-RPC implementation described, uses a variant of the Sun RPC *portmapper* service for mapping service names onto offers. This naming scheme does not allow for flexibility in choosing a service and there is no equivalent to the trading concept.

The Notus architecture provides support for applications which use both stateless and stateful servers, and offers the application various levels of involvement in the handoff process. Stateless servers are supported in Notus through the use of the default handoff subcontracts. These provide a number of different handoff semantics, without requiring application modifications. Notus RPC invocations are not indirected through user-level agents on the static network, thus avoiding major performance bottlenecks from which the M-RPC scheme is likely to suffer.

7.2 Application Migration

Work by Ashton has involved modifying a public domain shell [Joy80] in order to explicitly identify state for migration [Ashton96]. This was motivated by a requirement for load balancing in a cluster of homogeneous computers. During migration, an instance of the shell is created on the target host, with a pipe arranged for communication with the source. Various ad-hoc methods (including files) are used to transfer state between the two instances, until the target is able to take over execution from the source. It was observed that much of the work in modifying the application was spent in ensuring the robustness of the mechanism used for transferring state and restoring the target application. Section 6.6 describes how this work was modified to use the Notus implementation, enabling the shell to be migrated between heterogeneous computers. It was also noticed that by using the Notus implementation, fewer modifications were required to be made to the source code of the original shell.

The **Emerald** distributed programming language has been extended so that objects can be migrated between heterogeneous computers [Steensgaard95]. It uses a mechanism for marshalling thread and object state in a manner similar to Notus, but with no requirement for a separate Interface Definition Language (IDL) description of state. Instead, the Emerald compiler generates *templates*, describing objects and activation records in detail. The compiler also enumerates points in the code where migration can take place and code is generated, such that restarting an object at the same enumerated instruction on different architectures will always be valid. Between these points, the different instruction formats and code optimisations used for different architectures can make direct migration difficult. It was proposed that for migration between these points, code patches be generated on the fly to bridge between the different architectures. However, this is likely to be difficult to implement efficiently over a large number of architectures. Emerald programs require a close coupling with a large run-time system, and have an unusual model of computation which is not likely to be adopted in the near future by walkstation applications. Probably the greatest limitation of Emerald is the lack of popularity of the language.

Tui [Smith96] is a heterogeneous process migration scheme. Programs are written in a type safe subset of ANSI C and compiled using a specially modified compiler. When migration is requested, the memory image of the running process is scanned for all data values. The type of these is determined from tags introduced by the compiler and they are marshalled and stored on disc. On the target host, a new process is created and the global variables, heap, and stack are recreated from the marshalled state and execution is restored at the correct point. As with Emerald, the compiler enumerates points at which migration can take place, inserting *preemption points* at the beginning of loops and at the end of each compound statement. During optimisation, code must not be moved across these points. Tui does not address the reconnection of communication links to migrated applications and has only been demonstrated with very simple applications. Since all the state of the running application is marshalled and transferred, migration using Tui or Emerald results large disruptions to the user. The use of a specially modified compiler introduces a requirement for the compiler's availability on all target architectures. In contrast, Notus requires the programmer to specify suitable points in an application where migration can take place and also to nominate application state for transfer. The Notus implementation also ensures that an application's connections are rebuilt after migration and requires no compiler modifications.

Empire [Bates96] is an example of the use of migration techniques for interpreted languages. Modifications have been made to a Scheme [Steele75] interpreter, allowing the state of a running application to be captured and migrated. The implementation has been used as part of a framework for co-operative working in a ubiquitous computing environment. As workers move, objects dependent on the user's physical location are migrated. The authors report that due to performance limitations, interpreted objects were only used for management functions and compiled stateless objects were used for tasks such as media processing, which required better performance. During migration, these stateless objects were restarted on the target host and reconnected with their media streams. Notus does not migrate all the state of an application and has been shown to be suitable for the migration of compiled modules which perform video processing.

Another example of the migration of user interface applications has been made using the **Obliq** [Cardelli94] interpreted scripting language [Bharat96]. During migration, the application's user interface is traversed to identify, marshall, and transfer to a target host all mutable state using the Obliq *network copy* facilities. At the target, the unmarshalled representation is used to recreate a user interface using the local user interface toolkit (which potentially can be different to the source). The implementation uses a heavily modified Obliq run-time to achieve user interface migration and does not yet consider either the reconnection of an application's network connections after migration, or applications for which the migration of thread state is important. The paper reports migration times for "small to medium" sized applications of 5 to 45 seconds over a local area network.

Recently interest has been revived for compiling applications into a machine independent format. This ensures the portability of one distribution of the application over different architectures. **ANDF** [Macrakis93, Toft94] defines a low level, machine independent program representation. Programs are statically specialised for specific architectures by *installers*. **Java** [Sun95] is an object oriented programming language which compiles to a portable byte code format. This is then interpreted or compiled at load time. It is intended that applications written using Java are available dynamically over the network and attempts have been made to reduce the security risks caused by their execution. In both these cases, because of the intermediate representation and subsequent translation into a native format, it is possible to use a single distribution of an application over heterogeneous architectures.

When applications are compiled in the current Notus implementation environment, they are targeted at different architectures. Prior to migration, it is necessary to ensure that an appropriate executable version of the application is available on the target computer (described in Section 4.6.2). The use of a single distribution format would have removed the requirement for the existence of a repository for executables targeted at different architectures. The Notus architecture places no requirements on any particular compiler or execution format and so does not preclude the adoption of a single distribution format for the implementation environment.

It should also be noted that these schemes, while providing for a single code distribution over heterogeneous architectures, do not permit the interruption of a running application and its migration to another host. There has been great interest in Java, and it is probably only a matter of time before a migration service appears. However, the security implications of migrating Java applications should be seriously considered.

Approaches which enable checkpoint consistency for distributed applications were discussed in Section 4.7. However, there has been some recent work [Acharya94] which is relevant to the checkpointing of distributed applications in a mobile environment. This work is based upon the assumptions that a walkstation is not suitable for the long term storage of checkpoints due to its physical vulnerability and that disconnection should not prevent the recording of the global state of an application. The paper presents a roll-back algorithm, which is similar to those described in Section 4.7.1. Essentially the scheme requires an application's modules to leave a trail of checkpoints on the static network which may later be resolved to form a consistent checkpoint. There are two potential problems with this scheme. First, checkpoints are left at many different locations on the static network, requiring a significant effort when attempting to determine those which are consistent. Second, a walkstation is required to *checkpoint* before disconnection. This makes the voluntary, short term disconnection of a walkstation an expensive option.

In general, roll-back algorithms defer the work of creating a consistent checkpoint until the time at which it is required, usually when recovering from an application's failure. In the case of application migration, the formation of a consistent checkpoint must take place at the time of migration, so no deferment is possible. This, and other roll-back algorithms, are unsuitable for the Notus architecture because the formation of a consistent checkpoint must take place at the time of migration, and because all roll-back algorithms potentially restart applications from state which is very old, thereby causing much work to be repeated.

7.3 Summary

This chapter has described:

- the M-RPC mechanism for re-establishing RPC bindings after a mobile handoff, but which imposes undesirable restrictions on applications and is thought likely to suffer from performance problems,
- an ad-hoc solution for application migration which would have benefited from the support of a general suite of migration facilities, and
- a number of application migration schemes which have used compiler or interpreter modifications to automatically marshall an application's state for migration. Without programmer hints for both the state to be transferred and the most appropriate execution points for migration, all these schemes tend to migrate more state than is necessary.

None of the work reviewed has offered the suite of facilities which are provided by Notus, namely:

- traded handoffs with various levels of application involvement, supporting binding re-establishment for both mobile clients and servers,
- programmer support for low-latency application migration between heterogeneous platforms, without compiler or operating system modifications,
- facilities which ensure consistency between an application's modules during its migration, and
- the rebuilding of an application's connections after migration.

Chapter 8

Conclusion

8.1 Further Work

Time and resource constraints during the research of Notus have meant that there remains much scope for further work in this field. The following section describes some possible extensions to the current implementation and future directions for the Notus architecture.

8.1.1 Extensions to the Current Notus Implementation

Wireless Network Evaluation: The implementation environment did not contain a wireless network, and consequently the parts of the Notus architecture which interface with a wireless network have been simulated. Further work could concentrate on evaluating the architecture over wireless network protocols, so as to consider the issues for traded handoffs between different network types or over large distances. For application migration, other tradeoffs might be considered, such as increased bandwidth usage against power consumption.

Further Application Support: Future work could include producing a complete suite of mobile applications such as: electronic mail, World Wide Web (WWW) browsing, and multi-media broadcasting. It should be noted that for each of these applications there are different considerations to be made when operating in a mobile environment.

For example, if a WWW client browser [Berners-Lee94] were running on a walkstation which moves into a Wireless LAN (WLAN) environment, migration of the browser from the walkstation to a compute server might be useful. In this case, it would be appropriate to distribute the application between the walkstation and compute server, since the transfer to the compute server of data cached at the client is not likely to be productive.

Traded handoffs might also be considered for the WWW browser. Given the relatively static nature and large distribution of WWW pages, their caching has been found to be useful in reducing the otherwise large numbers of long distance requests. Current browsers can be configured with the fixed address of a *proxy* cache. However, use of a single proxy cache can become counter productive if a mobile browser is expected to be carried over large distances. On performing a traded handoff, a browser can locate and use the most appropriate proxy. An exported offer from a proxy might contain other properties, allowing a particular proxy to specialise in caching pages relating to a particular subject.

Extensions to Marshalling: The Notus implementation is restricted in that only data types described using an Interface Definition Language (IDL) can be automatically marshalled during migration. For other data types it is necessary for the application to provide marshalling code using base types defined in the IDL. Further work could extend the marshalling facilities of Notus, integrating with existing packages which allow marshalling of complex and self referential data types (using techniques described in [Herlihy82]). It might be the case that these aims require a closer coupling with the compiler than is defined in the current architecture.

8.1.2 Extensions to the Notus Architecture

Adaptive Architectures: These were described in Chapter 3 and enable applications to adapt to changes in their environment. One such adaptation is an application changing its data representations on the basis of long term variations in bandwidth. For example, an application presenting news dispatches might move from a video to a text-based representation. Other work such as [Hyden94] has demonstrated architectures which enable applications to adapt to changes in their allocated Quality of Service (QoS) over short time-scales; for example a video decoder application which adapts to QoS fluctuations on a per-frame basis. The Notus architecture can be extended so that applications are informed of changes over a greater range of conditions. This might involve an integration with the techniques used by other mobile aware architectures.

However, it remains an unresolved issue as to how an application is able to effectively adapt to new resources after migration. In such circumstances, it is likely that the application would be required to make use of services with a different QoS, possibly requiring the use of a different set of algorithms at the target from those at the source.

Extensions to the Traded Handoff Protocol: The traded handoff protocol enables the establishment of new bindings to servers to be performed in the background. The application is disrupted if client state corresponding to a session is transferred to a new server.

Further work might investigate how this state transfer can also be made in the background. For example, the video player application described in Section 6.4 could be made to predict where in the stream the new server should continue to play out the video. This would enable the client, in the background, to mount a new server, stashing a connection to the video stream endpoint. The client application might then use this stashed stream endpoint during the traded handoff, removing the requirement for any communication with the server while the client is disrupted.

Optimisations of this nature would also be useful where traded handoffs are interworked with application migration (see Section 4.6.4). Another useful optimisation might be that prior to migration, the source application sends to the target the properties of the network services which it requires. The target application could then use this information to establish and stash bindings to servers which match these properties, reducing the time required to restore the migrating application at the target.

All these optimisations require cooperation with the application. In the former case, session level state can never be transferred without some application involvement, and inconsistencies in the predicted state must be reconciled. In the latter case, the target cannot choose to establish a binding with one offer from a list of imported offers without assistance from the application.

Inter-Server Handoffs: It was mentioned in Section 4.5.3 that where there are very large amounts of session level state, a client might wish to reconnect with the original endpoint rather than suffer a disruption caused while state is being transferred from the client to the new server. In these circumstances, it might also be appropriate for the client to locate a new server and then request that the servers cooperate with each other to transfer the state.

The above scenario is illustrated in Figure 8.1, with a client first communicating with server A. During a traded handoff, the client first opens a connection (i) with a new server B and requests (ii) that B retrieve the client's state from A. Given an extension to the Handoff Interface (HDI) which enables one server to *Retrieve* the state from another, server B is then able to request (iii) that the client's state be transferred from A over the static network.

Such inter-server handoffs would benefit from interworking with the Notus application migration facilities when creating a channel between the two servers,

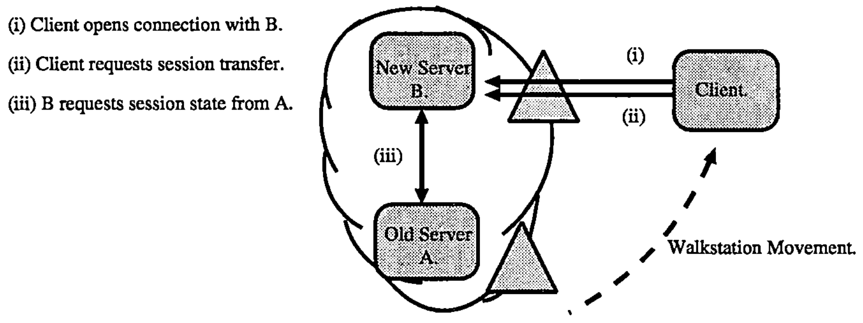

(i) Client opens connection with B.
(ii) Client requests session transfer.
(iii) B requests session state from A.
New Server B.
Old Server A.
Client.
(i)
(ii)
(iii)
Walkstation Movement.

Figure 8.1: Inter-Server Handoff

the marshalling of a client's state, and the transfer and restoration of the client's state at the target. Authentication between servers must be ensured, and the point at which there is a performance gain to be had from using this mechanism (compared to a client-server handoff) investigated.

It would be expected that for small amounts of state, the time spent authenticating and transferring state between the servers would be greater than simply transferring the state from the client to the new server. Client-server handoffs are facilitated by the fact that existing applications are structured in such a manner that an HDI implementation is often easy to produce.

8.2 Summary

This dissertation makes two direct contributions to the field of mobile computing: first, the use of application migration to exploit available compute resources; second, the introduction of the traded handoff concept, where applications are able to participate in the handoff process, thereby rebuilding connections to the most appropriate services.

An implementation has been made of a new **application migration** service which satisfies the requirements for application migration in a mobile environment, in that:

- applications can be migrated between heterogeneous platforms,
- there is little disruption experienced by applications during migration,
- applications are written using a standard compiled language,
- and applications running normally suffer little overhead.

Additionally, the implementation required no compiler modifications and provided facilities for the consistent migration of multi-threaded applications. The migration service has been demonstrated in the local area using three pre-existing applications which are representative of those expected to be used by walkstations. In each of the three cases, it was found that the application had been originally designed using a modular approach, and neither a detailed understanding of the application's internal structure or extensive modifications to the application were required when adding the capability for migration.

By involving the application programmer in the process of adding a migration capability to an application, the following issues can be addressed:

- policies for migration under different circumstances,
- the favoured distribution of the functional modules of an application between a walkstation and compute server,
- the consistency requirements between different modules, and
- the minimal state required for transfer during migration.

A system which transparently migrates applications, without programmer involvement, cannot fully address these issues. Future work might be directed towards the development of programming methodologies and tools which assist the application programmer in developing migration aware applications. For example, a simple tool might assist a programmer in choosing appropriate execution points and nominating application state for migration. It would be desirable that such tools are integrated into the program development environment.

A **Traded Handoff** requires coordination between the wireless network, a federated trader environment, and the walkstation's applications. An implementation has been made which enables traded handoffs to be evaluated in the local area and has challenged the currently held belief that the provision of mobile handoffs is purely a function of the underlying wireless network.

Applications participating in traded handoffs are required to implement a standard interface. It was noticed that an implementation of the interface requires functionality which is already an important feature of many applications in a distributed programming environment (to recover from network errors). For legacy code or applications which wish to remain unaware of mobility, an implementation of an Remote Procedure Call (RPC) service has been made which provides default traded handoff semantics.

It was found necessary for the trader to consider of the QoS of different routes so as to allow an application to choose between offers of service in a mobile environment. These QoS considerations were implemented through an extension to a standard trader and are also relevant to trading in a general distributed programming environment.

This dissertation has argued that mobility awareness and the support from appropriate tools, can enable walkstation applications to better adapt to a changing mobile environment, particularly when the walkstation is carried between different network types or over great distances.

Bibliography

[Accetta86]	M Accetta, R Baron, W Bolosky, D Golub, R Rashid, A Tevanian, and M Young. *Mach: A New Foundation for UNIX Development.* In Proceedings of USENIX Summer Conference, pages 93–112, 1986. (pp 15, 32)
[Acharya94]	A Acharya and B Badrinath. *Checkpointing Distributed Applications on Mobile Computers.* In 3rd Intl. Conf. on Parallel and Distributed Information Systems, September 1994. (p 122)
[Adams93]	N Adams, R Gold, B Schilt, M Tso, and R Want. *An Infrared Network for Mobile Computers.* In Symposium on Mobile and Location-independent Computing, pages 41–52. USENIX, August 1993. (p 3)
[Agrawal88]	R Agrawal and H Jagadish. *Partitioning Techniques for Large-Grained Parallelism.* IEEE Transactions on Computers, 37(12):1627–1634, December 1988. (p 30)
[Alonso90]	R Alonso, B Barbara, and H Garcia-Molina. *Data Caching Issues in an Information Retrieval System.* ACM Transactions on Database Systems, pages 359–384, September 1990. (p 45)
[ANSA92]	Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK. *An Overview of ANSAware 4.0*, March 1992. Document RM.099.00. (pp 13, 17, 28, 44, 76)
[ANSA93]	Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK. *ANSAware 4.0 DPL Reference Manual*, 1993. Document TR.032.00. (p 77)
[Ashton96]	P Ashton. *Migrating Tcsh.* Personal Communication, 1996. University of Canterbury, New Zealand. (pp 105, 120)

[Ayanoglu95] E Ayanoglu, S Paul, T LaPorta, K Sabnani, and R Gitlin. *AIRMAIL: A Link-Layer Protocol for Wireless Networks.* ACM Wireless Networks, February 1995. (p 24)

[Aziz94] A Aziz. *A Scalable and Efficient Intra Domain Tunnelling Mobile IP Scheme.* ACM Computer Communication Review, 24(1), 1994. (p 21)

[Bach86] M Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs, N.J., 1986. (p 15)

[Baker96] M Baker, X Zhou, S Cheshire, and J Stone. *Supporting Mobility in MosquitoNet.* In Technical Conference, pages 127–139. USENIX, January 1996. (pp 22, 40)

[Bakre94] A Bakre and B Badrinath. *I-TCP: Indirect TCP for Mobile Hosts.* Technical Report TR-314, Rutgers, August 1994. (pp 4, 23)

[Bakre95a] A Bakre and B Badrinath. *Handoff and System Support for Indirect TCP/IP.* In Second Symposium on Mobile and Location-independent Computing. USENIX, April 1995. (p 23)

[Bakre95b] A Bakre and B Badrinath. *M-RPC: A Remote Procedure Call Service for Mobile Clients.* In Proceedings of Mobicom, Berkeley, 1995. IEEE. (p 119)

[Balakrishnan95a] H Balakrishnan, S Seshan, E Amir, and R Katz. *Improving TCP/IP Performance over Wireless Networks.* In Proceedings of Mobicom, Berkeley, 1995. IEEE. (p 23)

[Balakrishnan95b] H Balakrishnan, S Seshan, and R Katz. *Improving Reliable Transport and Handoff Performance in Cellular Wireless Networks.* ACM Wireless Networks, December 1995. (p 24)

[Balakrishnan96] H Balakrishnan, V Padmanabhan, S Seshan, and R Katz. *A Comparison of Mechanisms for Improving TCP Performance over Wireless Links.* ACM Computer Communication Review, 26(4):256–269, October 1996. (p 24)

[Barak85] A Barak and A Shilow. *A Distributed Load Balancing Algorithm for a Multicomputer.* Software Practice and Experience, 15(9):901–913, September 1985. (p 31)

[Bates96] J Bates, D Halls, and J Bacon. *A Framework to Support Mobile Users of Multimedia Applications.* ACM Mobile Networks and Nomadic Applications, 1996. (p 121)

[Beguelin94] A Beguelin, E Seligman, and M Starkey. *Dome: A Distributed Object Migration Environment.* Technical Report CMU-CS-94-153, Carnegie Mellon University, 1994. (p 33)

[Beitz94] A Beitz and M Bearman. *An ODP Trading Service for DCE.* In Proceedings of the First International Workshop on Services in Distributed and Networked environments (SDNE), pages 42–49, Prague, Czech Republic, June 1994. IEEE Computer Society Press. (p 76)

[Benmohammed94] K Benmohammed and P Dew. *A Periodic Symmetrically initiated Load Balancing Algorithm for Distributed Systems.* ACM Operating Systems Review, 28(1):66–77, January 1994. (p 31)

[Berners-Lee94] T Berners-Lee, R Cailliau, H Nielsen, and A Secret. *The World Wide Web.* Communications of the ACM, 37(8):76–82, August 1994. (p 125)

[Bershad89] B Bershad, T Anderson, E Lazowska, and H Levy. *Lightweight Remote Procedure Call.* ACM Operating Systems Review, 23(5):102–113, 1989. (p 16)

[Bershad95] B Bershad, S Savage, P Pardyak, E Sirer, M Fiuczynski, D Becker, C Chambers, and S Eggers. *Extensibility, Safety and Performance in the SPIN Operating System.* ACM Operating Systems Review, 29(5):267–284, December 1995. (p 68)

[Bharat96] K Bharat and L Cardelli. *Migratory Applications.* Technical Report Research Report 138, Digital SRC, February 1996. (p 121)

[Birrell82] A Birrell, R Needham, and M Schroeder. *Grapevine: An Exercise in Distributed Computing.* Communications of the ACM, 25(4), April 1982. (p 12)

[Birrell84] A Birrell and B Nelson. *Implementing RPC.* ACM Transactions on Computer Systems, 2(1), February 1984. (pp 12, 71)

[Birrell93] A Birrell, G Nelson, S Owicki, and T Wobber. *Network Objects.* Proceedings of the 14th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review, 27(5):217–230, December 1993. (p 15)

[Biswas94] S Biswas. *Handling Real Time Traffic in Mobile Networks.* PhD thesis, University of Cambridge Computer Laboratory, 1994. Technical Report 351. (p 26)

[Black88] A Black, N Hutchinson, E Jul, H Levy, and L Carter. *Distribution and Abstract Types in Emerald.* IEEE Transactions on Software Engineering, 13(1):65–76, January 1988. (p 13)

[Black95] R Black. *Explicit Network Scheduling.* PhD thesis, University of Cambridge Computer Laboratory, 1995. (p 17)

[Brown95] P Brown. *The Electronic Post-it Note: A Model for Mobile Computing Applications.* In Proceedings of Mobile Computing and its Applications, Savoy Place, London WC2R OBL, UK, November 1995. IEE. (p 29)

[Cardelli88] L Cardelli, J Donahue, L Glassman, M Jordan, B Kalsow, and G Nelson. *Modula-3 Report.* Technical Report Technical Report no. 31, Digital SRC, 1988. (pp 13, 68)

[Cardelli94] L Cardelli. *Obliq: A Language with Distributed Scope.* Technical Report Research Report 122, Digital SRC, 1994. (p 121)

[Casas95] J Casas, D Clark, R Komuru, S Olto, R Prouty, and J Walpole. *MPVM: A Migration Transparent Version of PVM.* Technical Report CSE-95-002, Oregon Institute of Science and Technology, 1995. (p 34)

[Chandy85] K Chandy and L Lamport. *Distributed Snapshots: Determining Global States of Distributed Systems.* ACM Transactions on Computer Systems, 3(1):63–75, February 1985. (p 56)

[Cheshire96] S Cheshire and M Baker. *Internet Mobility 4x4.* ACM Computer Communication Review, 26(4):318–329, October 1996. (pp 22, 23)

[Comer94] D Comer and V Russo. *Using ATM for a Campus-Scale Wireless Internet.* In Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications, Dream Inn, Santa Cruz, CA, U.S.A, December 1994. (p 4)

[Condon95] J Condon, T Duff, M Jukl, and C Kalmanek. *Rednet: A Wireless ATM Local Area Network Using Infra Red Links.* In Proceedings of Mobicom, Berkeley, 1995. IEEE. (pp 3, 24, 25)

[Davies94a] N Davies. *An Object Management System to Support Distributed Multimedia Design Environments.* PhD thesis, Lancaster University, February 1994. (p 33)

[Davies94b] N Davies, S Pink, and S Blair. *Services to Support Distributed Applications in a Mobile Environment.* In First International Workshop on Services in Distributed Networked Environments. IEEE, 1994. (p 28)

[Deering95] S Deering and R Hinden. *Internet Protocol Version 6 (IPv6) Specification.* RFC-1833, December 1995. (p 22)

[Demers94] A Demers, K Petersen, M Spreitzer, D Terry, M Theimer, and B Welch. *The Bayou Architecture: Support for Data Sharing among Mobile Users.* In Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications, Dream Inn, Santa Cruz, CA, U.S.A, December 1994. (p 28)

[Dijkstra78] E Dijkstra, L Lamport, A Martin, C Scholten, and E Steftens. *On the Fly Garbage Collection: An Exercise in Cooperation.* Communications of the ACM, 21(11):966–975, November 1978. (p 59)

[Douglis91a] F Douglis and J Ousterhout. *Transparent Process Migration: Design Alternatives and the Sprite Implementation.* Software—Practice and Experience, 21(8):757–785, August 1991. (pp 8, 32)

[Douglis91b] F Douglis, J Ousterhout, M Kaashoek, and A Tanenbaum. *A Comparison of Two Distributed Systems: Amoeba and Sprite.* Computing Systems, 4(4):353–384, 1991. (p 31)

[Eager86] D Eager, E Lazowska, and J Zahorjan. *Adaptive Load Sharing in Homogeneous Distributed Systems.* IEEE Transactions on Software Engineering, 12(5):662–675, may 1986. (p 31)

[Eager88] D Eager, E Lazowska, and J Zahorjan. *The Limited Performance Benefits of Migrating Active Processes for Load Sharing.* In Conf. on Measurement and Modelling of Computer Systems, pages 63–72, Santa Fe, NM (USA), May 1988. (p 30)

[Eckhardt96] D Eckhardt and P Steenkiste. *Measurement and Analysis of the Error Characteristics of an In-Building Wireless Network.* ACM Computer Communication Review, 26(4):243–254, October 1996. (p 4)

[ETS95] ETSI Radio Equipment and Systems. *High Performance Radio Local Area Network (HIPERLAN)*, functional specification version 1.1 (draft) edition, January 1995. (p 3)

[Evers93] D Evers. *Distributed Computing with Objects.* PhD thesis, University of Cambridge Computer Laboratory, September 1993. (pp 13, 59)

[Evers94] D Evers. *Implementing Modules in Nemesis.* Pegasus Cambridge Working Document 009, February 1994. (p 17)

[Findlay96] A Findlay. *The Multi-Media Telephone: Directory Service and Session Control for Multi-Media Communications.* In Proceedings of the Third International Workshop on Services in Distributed and Networked environments (SDNE), pages 169–173. IEEE Computer Society Press, June 1996. (p 40)

[Francis94] P Francis and R Govindan. *Flexible Routing and Addressing for a Next Generation IP.* ACM Computer Communication Review, 24(4):116–125, October 1994. (p 22)

[Fraser93] S Fraser. *Early Experiments with Asynchronous Time Division Networks.* IEEE Network, 7(1):12–26, January 1993. (p 19)

[Friday96] A Friday. *Extensions to ANSAware for advanced mobile applications.* In Proc International Conference on Distributed Platforms, Dresden, 1996. (pp 13, 28)

[Geist93] A Geist, A Benguelin, J Dongarra, W Jiang, R Manchek, and V Sunderam. *PVM 3.0 Users Guide and Reference Manual*, February 1993. (p 33)

[Goodman91] D Goodman. *Trends in Cellular and Cordless Communications.* IEEE Communications Magazine, pages 32–40, June 1991. (p 26)

[Hager93] R Hager, A Klemets, G Maguire, M Smith, and F Reichert. *MINT - A Mobile Internet Router.* In Proceedings of IEEE VTC, May 1993. (p 3)

[Hamilton84] G Hamilton. *A Remote Procedure Call System.* PhD thesis, University of Cambridge Computer Laboratory, December 1984. Technical Report 70. (p 12)

[Hamilton93a] G Hamilton and P Kougiouris. *The Spring Nucleus: A Microkernel for Objects.* Technical Report 93-14, Sun Microsystems Laboratories, Inc., 2550 Garcia Avenue, Mountain View, California 94043, April 1993. (pp 13, 15, 68)

[Hamilton93b] G Hamilton, M Powell, and J Mitchell. *Subcontract: A Flexible Base for Distributed Programming.* Technical Report 93-13, Sun Microsystems Laboratories, Inc., 2550 Garcia Avenue, Mountain View, California 94043, April 1993. (p 16)

[Harter93] A Harter and F Bennett. *Low Bandwidth Infra-Red Networks and Protocols for Mobile Communicating Devices.* Technical Report ORL Report no. 93-5, Olivetti Research Limited, Cambridge, 1993. (p 3)

[Harter94] A Harter and A Hopper. *A Distributed Location System for the Active Office.* Technical Report ORL Report no. 94-1, Olivetti Research Limited, Cambridge, 1994. (p 3)

[Hayton96] R Hayton and O Seidel. *MSRPC3.* Technical Report, University of Cambridge Computer Laboratory, 1996. An Object Based RPC Package. (p 44)

[Herlihy82] M Herlihy and B Liskov. *A Value Transmission Method for Abstract Data Types.* ACM Transactions on Programming Languages and Systems, 4(4):527–551, October 1982. (p 126)

[Hopper78] A Hopper. *Local Area Communication Networks.* Technical Report 7, University of Cambridge Computer Laboratory, 1978. (p 19)

[Hyden94] E Hyden. *Operating System Support for Quality of Service.* PhD thesis, University of Cambridge Computer Laboratory, January 1994. (pp 17, 126)

[Imielinski92] T Imielinski and B Badrinath. *Mobile Wireless Computing: Solutions and Challenges in Data Management.* Technical

Report CUCS-007-92, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, 1992. (pp 1, 4)

[Ioannidis91] J Ioannidis, D Duchamp, and G Maguire. *IP-based Protocols for Mobile Internetworking.* ACM Computer Communication Review, 21(4):235–245, September 1991. (pp 4, 20, 41)

[ISO88] ISO/CCITT. *Recommendation X.500: The Directory - Overview of Concepts, Models and Services*, 1988. (p 40)

[ISO91] ISO/IEC. *Coded Representation of Picture, Audio and Multimedia/Hypermedia Information*, December 1991. Committee Draft ISO/IEC CD 11172. (p 101)

[ISO94] ISO/IEC. *ODP Reference Model: Trading Function*, October 1994. Committee Draft ISO/IEC/JTC1/SC21 N807. (pp 14, 76)

[ISO95a] ISO/IEC. *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*, 1995. ISO/IEC 8824-1:1995. (p 12)

[ISO95b] ISO/IEC. *ODP Reference Model: Overview*, January 1995. Draft Recommendation X.902, International Standard 10746-1. (pp 13, 68)

[Jacobson88] V Jacobson. *Congestion Avoidance and Control.* ACM Computer Communication Review, 18(4), August 1988. (p 23)

[Johnson94] D Johnson. *Scalable and Robust Internetwork Routing for Mobile Hosts.* In Proceedings of the 14th International Conference on Distributed Computing Systems. IEEE, June 1994. (pp 22, 41)

[Joy80] W Joy. *Tcsh V6.0.* Distributed from: ftp.deshaw.com, 1980. (pp 105, 120)

[Keeton93] K Keeton, B Mah, S Seshan, R Katz, and D Ferrari. *Providing Connection Oriented Network Services to Mobile Hosts.* In Symposium on Mobile and Location-independent Computing. USENIX, August 1993. (p 25)

[Kuo74] F Kuo. *The Aloha System.* ACM Computer Communication Review, 4(1), January 1974. (p 3)

[Lambley84]	R Lambley. *Developments in Cellular Radio.* Electronics and Wireless World, June 1984. (p 26)
[Lauer78]	H Lauer and R Needham. *On the Duality of Operating System Structures.* Technical Report, Xerox Palo Alto Research Centre, March 1978. (p 11)
[Leslie94]	I Leslie, R Black, and D McAuley. *Experiences of Building an ATM Switch for the Local Area.* ACM Computer Communication Review, 24(4):158–167, October 1994. (p 26)
[Liedtke93]	J Liedtke. *Improving IPC by Kernel Design.* ACM Operating Systems Review, 27(5):175–187, December 1993. (p 16)
[Linn94]	J Linn. *Generic Interface to Security Services.* Computer Communications, 17(7):483–491, July 1994. (p 62)
[Liskov85]	B Liskov, M Herlihy, and L Gilbert. *Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing.* Technical Report CMU-CS-85-168, Carnegie-Mellon University, 1985. (p 11)
[Litzkow92]	M Litzkow and M Solomon. *Supporting Checkpointing and Process Migration outside the UNIX kernel.* In Proceedings of the Winter USENIX Conference, pages 283–290, January 1992. (pp 8, 31)
[Macrakis93]	S Macrakis. *Delivering Applications to Multiple Platforms using ANDF.* Technical Report, Open Software Foundation, 1993. (p 122)
[McAuley89]	D McAuley. *Protocol Design for High Speed Networks.* PhD thesis, University of Cambridge Computer Laboratory, September 1989. Technical Report No. 186. (p 69)
[Milojicic93]	D Milojicic, W Zint, A Dangel, and P Giese. *Task Migration on top of the Mach Microkernel.* In Proceedings of the 3rd USENIX Mach Symposium, Santa Fe, U.S.A, April 1993. (pp 8, 32)
[Montenegro95]	G Montenegro and S Drach. *System Isolation and Network Fast Fail Capability in Solaris.* In Second Symposium on Mobile and Location-independent Computing, pages 67–78. USENIX, April 1995. (p 28)

[Mullender92] S Mullender, I Leslie, and D McAuley. *Pegasus Project Description.* Technical Report Memoranda Informatica 92–75, University of Twente Faculty of Computer Science, September 1992. (p 16)

[Needham78] R Needham and M Schroeder. *Using Encryption for Authentication in Large Networks of Computers.* Communications of the ACM, 21(12):993–999, December 1978. (p 61)

[Needham82] R Needham and A Herbert. *The Cambridge Distributed Computing System.* International Computer Science Series. Addison Wesley, 1982. (pp 11, 12)

[Noble95] B Noble, M Price, and M Satyanarayanan. *A Programming Interface for Application Aware Adaptation in Mobile Computing.* In 2nd Symposium on Mobile and Location Independent Computing, Ann Arbor, Michigan, April 1995. USENIX. (p 28)

[Olsen92] M Olsen. *A Persistent Object Infrastructure for Heterogeneous Distributed Systems.* In Proceedings of 2nd International Workshop on Object Orientation in Operating Systems, pages 49–55. IEEE, September 1992. (p 33)

[OMG95] OMG (Object Management Group). *Common Object Request Broker: Architecture and Specification 2.0,* July 1995. Technical Document PTC/96-03-04. (pp 13, 15, 16, 44, 68)

[OSF91] OSF (Open Software Foundation). *Distributed Computing Environment: An Overview,* April 1991. (pp 13, 68)

[Otway95] D Otway. *ANSA Phase III: The ANSA Binding Model.* Technical Report Document 1392.01, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK, January 1995. (p 15)

[Perkins96] C Perkins. *IP Mobility Support.* Internet Draft, May 1996. (p 21)

[Pope96] S Pope. *Application Migration for Mobile Computers.* In Proceedings of the 3rd International Workshop on Services in Distributed and Networked Environments, pages 20 – 27, Macau, June 1996. IEEE. (p i)

[Porter94] J Porter and A Hopper. *An ATM Based Protocol for Wireless LANs.* Technical Report ORL Report no. 94-2, Olivetti Research Limited, Cambridge, 1994. (pp 3, 24, 41)

[Postel81a] J Postel. *The Internet Protocol.* RFC-791, September 1981. (pp 19, 69)

[Postel81b] J Postel. *Transmission Control Protocol.* RFC-793, September 1981. (pp 23, 69)

[Prouty94] R Prouty, S Otto, and J Walpole. *Adaptive Execution of Dataparallel Computations on Networks of Heterogeneous Workstations.* Technical Report CSE-94-012, Oregon Institute of Science and Technology, 1994. (p 34)

[Rahnema93] M Rahnema. *Overview of the GSM System and Protocol Architecture.* IEEE Communications Magazine, pages 92–100, April 1993. (pp 3, 26)

[Rajagopalan95] B Rajagopalan. *Mobility Management in Integrated Wireless ATM Networks.* In Proceedings of Mobicom, Berkeley, 1995. IEEE. (p 24)

[Randell75] B Randell. *System Structures for Software Fault Tolerance.* IEEE Transactions on Software Engineering, SE-1(3):220–232, June 1975. (p 54)

[Rashid81] R Rashid and G Robertson. *Accent: A Communication Oriented Network Operating System Kernel.* Proceedings of the 8th Symposium on Operating System Principles, pages 64–75, December 1981. (pp 8, 31)

[Richardson93] T Richardson, F Bennett, G Mapp, and A Hopper. *Teleporting in an X Window Environment.* Technical Report, Olivetti Research Laboratory, Cambridge, November 1993. (p 29)

[Roscoe94] T Roscoe. *The Middl Manual.* Technical Report Pegasus Working Document, University Of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, 1994. (p 17)

[Roscoe95a] T Roscoe. *Clanger: An Interpreted Systems Programming Language.* ACM Operating Systems Review, 29(2):13–20, April 1995. (p 17)

[Roscoe95b] T Roscoe. *The Structure of a Multi-Service Operating System.* PhD thesis, University of Cambridge Computer Laboratory, April 1995. (pp 13, 15, 16, 17)

[Rouche95] E Rouche. *The Fast Freeze Algorithm for Process Migration.* PhD thesis, University of Illinois at Urbana Champaign, 1995. (p 32)

[Rowe93] L Rowe, K Patel, and B Smith. *MEPG Video Software Decoder.* Distributed from: toe.cs.berkeley.edu, 1993. (p 99)

[Rozier89] M Rozier, V Abrossimov, F Armand, I Boule, M Gien, M Guillemont, F Herrmann, C Kaiser, S Langlois, P Leonard, and W Neuhauser. *CHORUS Distributed Operating System.* Technical Report CS/TR-88-7.8, Chorus Systemes, February 1989. (p 15)

[Saltzer79] J Saltzer. *Naming and Binding of Objects.* In R Bayer, R Graham, and G Seegmuller, editors, *Operating Systems: an Advanced Course*, volume 60 of *LNCS*, chapter 3.A, pages 100–208. Springer-Verlag, 1979. (p 14)

[Saltzer84] J Saltzer, D Reed, and D Clark. *End-to-End Arguments in System Design.* ACM Transactions on Computer Systems, 2(4), November 1984. (p 6)

[Sandberg85] R Sandberg, D Goldberg, S Kleiman, D Walsh, and B Lyon. *Design and Implementation of the Sun Network Filesystem.* In Proc. Summer 1985 USENIX Conf., pages 119–130, Portland OR (USA), June 1985. (pp 29, 43, 106)

[Schilit93a] B Schilit, N Adams, R Gold, M Tso, and R Want. *The Parc Tab Mobile Computing System.* Technical Report CSL-93-20, Xerox Palo Alto Research Centre, December 1993. (p 29)

[Schilit93b] B Schilit, M Theimer, and B Welch. *Customising Mobile Applications.* In Proceedings of Usenix Symposium on Mobile and Location Independent Computing, pages 129–138, August 1993. (p 29)

[Sentovich88] E Sentovich and R Spickelmier. *Xrn: News Reader.* Distributed from: ftp://ftp.com.ov.com/pub/xrn/xrn.tgz, 1988. (p 104)

[Shapiro90] M Shapiro, D Plainfosse, and O Gruber. *A Garbage Detection Protocol for a Realistic Distributed Object Support System.* Technical Report Rapport de Recherche 1320, INRIA, November 1990. (p 59)

[Shearer95] E Shearer. *TETRA - A Platform for Multimedia.* In Proceedings of Mobile Computing and its Applications, Savoy Place, London WC2R OBL, UK, November 1995. IEE. (p 3)

[Shum96] K Hong Shum. *Adaptive Parallelism for Computing on Heterogeneous Clusters.* PhD thesis, University of Cambridge Computer Laboratory, August 1996. (p 34)

[Smith96] P Smith and N Hutchinson. *Heterogeneous Process Migration: The Tui system.* Paper published at University of British Columbia: http://www.cs.ubc.ca, February 1996. (p 121)

[Steele75] G Steele and G Sussman. *Scheme: An Interpreter for the extended Lamda Calculus.* Technical Report Memo 349, MIT Artificial Intelligence Laboratory, 1975. (p 121)

[Steensgaard95] B Steensgaard and E Jul. *Object and Native Code Thread Mobility Among Heterogeneous Computers.* In Proceedings of the 15th Symposium on Operating Systems Principles, Colorado, December 1995. ACM. (p 120)

[Steiner88] J Steiner, B Neuman, and J Schiller. *Kerberos: An Authentication Service for Open Network Systems.* In Usenix Conference Proceedings, pages 191–202, February 1988. (p 60)

[Strom85] R Strom and S Yemini. *Optimistic Recovery in Distributed Systems.* ACM Transactions on Computer Systems, 3(3):204–226, August 1985. (p 54)

[Sun87] Sun Microsystems. *XDR: External Data Representation Standard*, June 1987. (p 12)

[Sun88] Sun Microsystems. *RPC: Remote Procedure Call, Protocol Specification, Version 2*, 1988. (p 12)

[Sun95] Sun Microsystems. *The Java Language: A White Paper*, 1995. (p 122)

[Tait92] C Tait and D Duchamp. *An Efficient Variable Consistency Replicated File Service.* In Proceedings of File Systems Workshop. USENIX, May 1992. (p 29)

[Tantawi85] A Tantawi and D Towsley. *Optimal Static Load Balancing in Distributed Computer Systems.* Journal of the ACM, 32(2):445–465, April 1985. (p 30)

[Teraoka93] F Teraoka and M Tokoro. *Host Migration and Transparency in IP Networks: The VIP Approach.* ACM Computer Communication Review, 21(1):45–65, January 1993. (pp 4, 21)

[Theimer85] M Theimer, K Lantz, and D Cheriton. *Preemptable Remote Execution Facilities for the V-System.* In Proc. 10-th ACM Symp. on Operating System Principles, December 1985. (pp 8, 32)

[Toft94] J Toft and J Nielsen. *Formal Specification of ANDF.* Technical Report, DDC International, A/A G Lundtoftevej 1B, 2800 Lyngby, Denmark, 1994. (p 122)

[Toh95] C Toh. *The Design and Implementation of a Hybrid Handover Protocol for Multi-Media Wireless LANs.* In Proceedings of Mobicom, Berkeley, 1995. IEEE. (p 26)

[Trotter95] J Trotter and M Cravatts. *A Wireless Adaptor Architecture for Mobile Computing.* In 2nd Symposium on Mobile and Location Independent Computing, pages 25–31, Ann Arbor, Michigan, April 1995. USENIX. (p 3)

[Want92a] R Want and A Hopper. *Personal Interactive Computing Objects.* IEEE Transactions on Consumer Electronics, 38(1):10–20, February 1992. (p 29)

[Want92b] R Want, A Hopper, V Falaco, and J Gibbons. *The Active Badge Location System.* Technical Report ORL Report no. 92-1, Olivetti Research Limited, Cambridge, 1992. (pp 3, 4, 29)

[Wernick96] P Wernick. *MMN URI: Work Package 5 (Security) The Trial Implementation and its Lessons.* HATS Project Document, University of Cambridge Computer Laboratory, March 1996. (p 62)

[Wilson92] P Wilson. *Uniprocessor Garbage Collection Techniques.* In Y Bekkers and J Cohen, editors, *Memory Management: International Workshop IWMM 92,* volume 637 of *LNCS,* pages 1–42. Springer-Verlag, September 1992. (p 59)

[Yarvin93] C Yarvin, R Bukowski, and T Anderson. *Anonymous RPC: Low Latency Protection in a 64 Bit Address Space.* In Proceedings of the Summer USENIX Conference, pages 175–186, Cincinnati, U.S.A, June 1993. (p 16)

[Zadok93] E Zadok and D Duchamp. *Discovery and Hot Replacement of Replicated Read Only File Systems with Application to Mobile Computers.* In Proceedings of Summer Conference. USENIX, June 1993. (p 29)

[Zayas87] E Zayas. *Attacking the Process Migration Bottleneck.* In Proc. 11-th ACM Symp. on Operating System Principles, pages 13–24, 1987. (pp 8, 31)

[Zhou88] S Zhou. *A Trace-Driven Simulation Study of Dynamic Load Balancing.* IEEE Transactions on Software Engineering, 14(9):1327–1341, 1988. (p 31)

[Zhou93] S Zhou, X Zheng amd J Wang, and P Delisle. *Utopia: A Load Sharing Facility for Large Heterogeneous Distributed Computer Systems.* Software—Practice and Experience, 23(12):1305–1336, December 1993. (p 31)