Reconfiguring Distributed Computations

(Note 2014: this document has been compiled from separate files in a 1990 version of Word. There will be the occasional glitch in Symbol characters. Take page numbers and header/footer information with a pinch of salt.)

Timothy Paul James Gerard Kindberg

The Polytechnic of Central London

A dissertation submitted in partial fulfilment of the requirements
of the Council for National Academic Awards
for the degree of

Doctor of Philosophy

October 1990

Abstract

This dissertation presents a practical model for programming reconfigurable distributed computations (RDCs). The model is supported by an operating environment, Equus, which has been implemented on a multiprocessor computer system. This environment supports the concurrent execution of multiple RDCs, and RDCs can act as servers to client processes belonging to other RDCs.

RDCs can change their resource usage and therefore their configuration between run-times and/or during their run-time. They do so for a variety of reasons, and range from parallel applications that exploit a computer which is freed from another use and becomes available to them during their run-time, to server computations which expand their configuration to an extra computer when client demand reaches a threshold value.

The programming model provides primitives to establish an RDC and then reconfigure it by changing its process population, reconnecting its processes and migrating them between machines. The dissertation presents the design of the model's structural components and reconfiguration mechanisms, and shows their utility both for stand-alone applications and in particular for server RDCs with independent clients.

The dissertation discusses the transparency of reconfigurations with respect to application algorithms, and analyses general requirements of an RDC's application which must be met when reconfigurations are applied to it. Transparency is a requirement in the case of clients connected to multiple-process server RDCs which reconfigure to change the association between client and server processes. The dissertation shows the contribution of the model in making this achievable without interaction with the clients.

The dissertation gives the main features of the design of the kernel which is the major component of the implementation.

Acknowledgements

First and foremost, I should like to acknowledge the support of my supervisor, Professor Yakup Paker. Yakup has always shown his enthusiasm for my work and has given me much encouragement and guidance. Writing this dissertation sometimes seemed like climbing a mountain, and it is with many thanks to Yakup and his friendship that I made it this far.

My colleague and friend, Ali Sahiner, has worked on the Wormos and Equus projects almost as long as I have. Ali contributed to the kernel implementations, he has developed demanding applications, he has been a constructive critic, and I have very much enjoyed working with him.

Thanks also go to my brother, Chris, and my second supervisor, Dr. Jean Bacon, for their helpful criticisms and for their influential words of encouragement. Amongst members at the Computer Science Department of Queen Mary and Westfield College, University of London, I am especially grateful to Professor Richard Bornat, Jean Dollimore and Professor George Coulouris for giving me their support over the last few months.

Andrew Sherman did most of the implementation work on memory management and the pool manager, and Jack Leung on the kernel's time daemon and a prototype name service. Both became good friends. I wish that we could have been a team together for longer. Tony Deacon prototyped the code for stream space during his time at the short-lived Zebra Parallel. Martin Farncombe, Richard Harris and Debbi Lane tried with the rest of us to make Equus a commercial success. Notable for their contributions as Equus users have been Esin Onbasioglu, who has been investigating load balancing, and Matti Hiltunen and Philippe Garat, whose fault-tolerant and image processing applications provided valuable feedback for the system design.

Thanks to my mother, to Sal, Maria and Top and the rest of my family in Canada for their faith in me. My friends (especially Angelika) have all been a source of support and encouragement throughout, as were Rosemary and Maggie and the WPF group.

Last but not least, thank you Alice for helping me out so that I could concentrate, and thanks to both you and Gene for putting up with my preoccupations at home. It must have seemed that it might never end. I hope that I shall be able to help out in turn if either of you finds a mountain to climb.

The work presented in this dissertation was funded in part by the Science and Engineering Research Council under grant number GR/C69214 (*Self-managing Distributed Processes*), and by a Department of Trade and Industry Support for Innovations grant (*the Wormos Operating System*).

Trademarks

For Hunky Dory

List of Chapters and Appendices

Table of Contents

List of Tables

List of Figures

Glossary

buffer handle	A type of *reference* enabling the owner to read/write a data segment in another *incarnation*'s address space.
channel	A structural component used to route messages between *incarnation*s.
clients and servers	A server is a system-provided *process* or group of processes in a *distributed system*. It provides a shared facility such as access to a common resource, in response to message-based requests from independently created processes called clients.
configuration	The disposition of structural components of a *distributed computation*. The configuration of an *RDC* is its *structure* plus the *mapping* of its *incarnation*s onto underlying nodes.
distributed computation	A computation consisting of communicating *process*es running at the *node*s of a *distributed system*.
distributed system	A collection of computers linked by a communications network for sharing of resources and co-operative working.
dynamic	Performed at run-time; applied to configuring or reconfiguring a distributed computation.
Equus	The commercial name of the operating environment which together with Unix supports the execution of *RDC*s. Previously known as Wormos.
ghost	A kernel process which handles operations and event notifications for an incarnation.
incarnation	A *process* belonging to an *RDC*; an executing image of a *module*.

incarnation handle	A type of *reference* which enables the owner to migrate an *incarnation* and control it in other ways.
invocation	A basic *message*-based interaction between *incarnation*s.
kernel	The basic operating software resident at a computer in a *distributed system*.
mapping	The mapping of a set of *incarnation*s is the function giving each incarnation's current host *node*.
message	A discrete unit of data passed between *incarnation*s.
migration	An *incarnation* is said to migrate when it is moved intact between two *node*s.
module	A component of an *RDC*'s program, used as a template for creating *incarnation*s; a unit of functional division.
node	General term for a distinct computer in a *distributed system* - from a basic processing element to a workstation.
pincdata	**p**er-**inc**arnation **data**. A collection of kernel-managed data such as *references* pertaining to a single incarnation.
port	A type of *reference* enabling an *incarnation* to receive messages arriving over *channel*s.
port handle	A type of *reference* denoting a port and used for reconfiguring its attachment to *channel*s.
process	The main unit of execution supported at a *node*. Unless otherwise stated, a process is heavyweight: it has its own address space, which may be shared by more than one thread of execution. Processes in the *RDC* model are called *incarnations*.
processor pool	A collection of *node*s in a *distributed system* whose members are temporarily allocated to *distributed computation*s.

propagation	The transmission of a copy of a *reference* from one *incarnation* to another.
RDC	A *dynamic*ally configurable and reconfigurable *distributed computation* belonging to the model elaborated in this dissertation.
reconfiguration	A set of changes applied to a *distributed computation*'s *configuration*.
reference	A sealed data structure held by the kernel and owned by an incarnation, which refers to a kernel-supported entity such as an incarnation, and whose possession enables protected access to it.
stream	A type of *reference* enabling the possessor to perform *invocation*s over an attached *channel*.
stream handle	A type of *reference* denoting a *stream* and used for reconfiguring its attachment to *channel*s.
stream space	A repository into and out of which *incarnation*s copy *reference*s to achieve decoupled *propagation*.
structure	The structure of an *RDC* is the relation of ownership of *stream*s and *port*s by the constituent *incarnation*s, and the relation of interconnection of these streams and ports via *channel*s.
transparency	Something is transparent to a *process* if the process's execution is semantically independent of it. The "something" can be an event or change occurring to an entity in the same system, or a property of a related entity. In particular, a reconfiguration of a *distributed computation* might or might not be transparent to a constituent or client process.

Chapter 1
Introduction

This dissertation presents a programming model for reconfigurable communicating process computations, and their supporting operating environment which has been implemented on a multiprocessor computer system. It argues that the model and environment provide a practical and general framework within which solutions to problems involving reconfiguration can be programmed. It argues for a model in which distributed computations can dynamically establish and manage their own configuration. This is because of application-specific dependencies on run-time conditions – in particular, on the changing availability of computing resources.

The dissertation analyses the problems that arise when reconfigurable computations act as servers to independent client computations. It focuses upon the case of server computations consisting of multiple peer processes, between which clients are re-connected. A separation between reconfigurations and application algorithms cannot always be achieved, but the dissertation argues for – and the model provides – reconfiguration mechanisms whose operations are transparent to clients of these server computations. The dissertation develops, in relation to the mechanisms of the model, a general set of requirements which must be met by multiple-peer server applications, if reconfigurations affecting them and their clients are to take place consistently.

This chapter introduces the problem in the context in which it was initially investigated, by first considering schemes for dynamically allocating processing resources to distributed computations. It then introduces the programming model, and describes the computer systems on which the operating software has been implemented. It goes on to outline how the rest of the dissertation is structured to cover the topics at issue.

1.1 Reconfiguration

With the advent of relatively cheap and yet increasingly powerful microprocessors, many users have their day-to-day computing needs economically met by a personal computer or workstation containing a single processor. Intensive applications such as engineering or image-processing calculations are not uncommon, however, and tend to demand more processing power than these computers can individually provide.

When computers are connected as part of a distributed computer system, there are two established means aimed at improving the performance of applications through shared use of processing resources:

1] **Use of a processor pool**. The first scheme for sharing processors involves a purpose-built pool of basic processors, each usually consisting of a single processor card, with CPU, local memory and communications interface [AMOEBA90]. The pool is separate to the users' individual computers or terminals but accessible from them, and processors are allocated from the pool as demands for extra resources arise. When a processor is finished with, it is returned to the pool for further use.

2] **Use of idle workstations**. This scheme avoids buying extra hardware and instead attempts to maximise the usage of existing workstation resources: when an intensive application is to be run, an attempt is made to locate other workstations on the network which are not in use or are under-utilised [HAGMAN, NICHOL, WORMS]. Such workstations are regarded as constituting a fluctuating pool whose members can be allocated to intensive applications requiring them.

In both types of pool, computers or basic processors (henceforth collectively referred to as *nodes*, whichever kind of distributed system they belong to) are allocated to distributed computations *dynamically* – that is, at run-time. They are also withdrawn dynamically. For example, withdrawal can be made when a user logs back on to his or her workstation, or when a temporarily allocated node is required for a more privileged user of a purpose-built pool. Multiple allocations and withdrawals can be made from the pool during a single computation's run time, as the level of competition for nodes varies.

The extra processors acquired can be used to run application programs in parallel. Of main concern to this dissertation are *distributed computations*, for which many programming models have been developed. These are defined to be single applications consisting of communicating processes distributed across a number of interconnected machines without shared memory. A process is taken here to be the main component of execution according to the particular model concerned. Different models have different definitions of process, and some of these will be described in Chapter 2. Communication takes different forms in different models, but always involves a combination of synchronisation and/or data copying between processes.

To utilise a node, one or more processes belonging to the distributed computation are created there, each of which needs (in general) to be connected to other processes in order to communicate with them. When a node is withdrawn in the case of the Cambridge Distributed System's processor bank [CRAFT], processes executing there are terminated after a notice period has elapsed. Other processes should not continue to attempt to communicate with them there, and a strategy is also required to recover from this withdrawal so that essential results are not lost. In summary, node allocation and withdrawal imply the need to change the process population and the interconnection of the processes, whilst maintaining the application’s integrity.

1.1.1 Definition of Configuration

Informally, the configuration of a distributed computation is a combination of its logical process interconnection structure and the mapping of the processes onto nodes (Figure 1.1). The following more formal definition of configuration is based on the assumption that processes use local interfaces for communications. The term interface is used here in the restricted sense of that which serves to distinguish the streams of communications a process either initiates or accepts. Examples of this are found in remote procedure calls [RPC], and communication over CSP channels [CSP]. The process presents to the run-time system an interface identifier which serves at the application level to direct an outgoing communication or accept an incoming communication. The routing of communications is determined by the connections of the interfaces. The run-time system translates the interface indentifier into a physical address to deliver or pick up the communication.

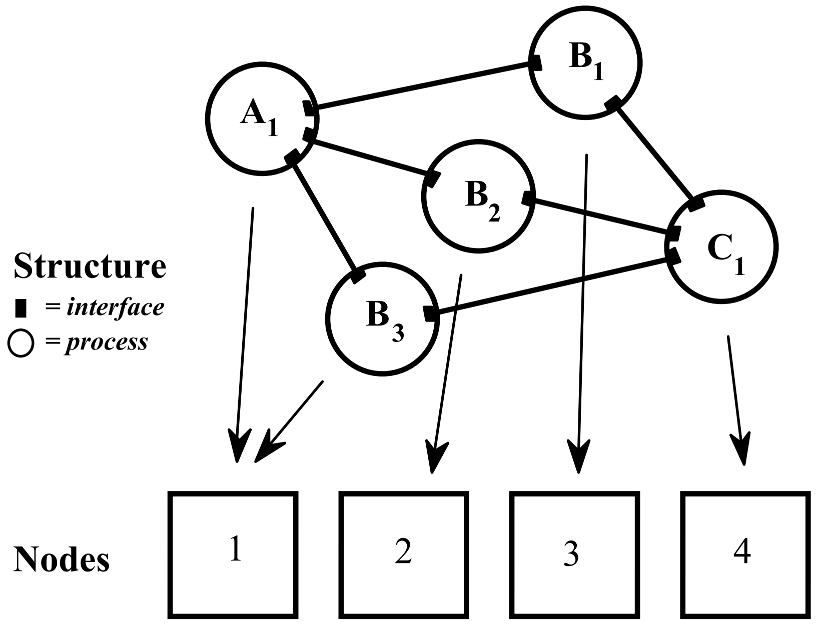


Figure 1.1: Configuration = Structure + Mapping onto Nodes

The configuration of a distributed computation d executing on a distributed system with set of nodes N is defined to be $<P_d, s_d, m_d>$, where:

P_d is the set of processes which belong to d.

Each $p \in P_d$ has a set i(p) of communications interfaces of which it has sole use.

s_d is the relation which gives the *structure* of the distributed computation.

This is the logical interconnection between the processes, achieved by the connections of their communications interfaces.

$s_d \subseteq I \times I$, where $I = \bigcup_{p \in P_d} i(p)$. $(i_1, i_2) \in s_d$ *iff* the two processes using the interfaces i_1 and i_2 are able to communicate using them.

m_d is the function which gives the *mapping* of d onto the underlying nodes.

m_d: $P_d \rightarrow N$. The node location of a process $p \in P_d$ is $m_d(p)$. The logical interconnections of processes are also mapped onto the underlying network, but this problem of routing will not be considered by the present work.

We require that the configuration of collections of client and server processes belonging to different applications can be considered. This definition of configuration is applied similarly to collections of communicating processes belonging to more than one distributed computation. P_d is replaced by P, the set of processes whose configuration is being considered.

1.1.2 Reconfigurability

Distributed computations which adapt to conditions of variable processor availability have their configuration dynamically established and changed. They are said to be *reconfigurable*: their configuration can change from run-time to run-time, and it can change during their run-time. Reconfigurability was originally motivated in this research by dynamic node allocation from a pool. But this is not the only reason for considering it. Altogether, the factors which make reconfigurability of research interest are:

R1] **Resource-based configurations**. Most generally, even "homogeneous" distributed systems can contain nodes with connections to specialised devices or other distinguishing characteristics. Also, nodes can be available in different numbers at different times for administrative or other reasons (e.g. failure), and not only due to systematic allocations from a pool. Thus configurations are dependent upon resources in a wider sense than that of a fluctuating provision of CPU power.

R2] **Data-dependent configurations**. Both the multiplicity of component processes and their inter-connection relation sometimes depend on the data upon which the distributed computation operates.

R3] **Load balancing**. In a distributed system with homogeneous nodes, multi-tasking at the nodes can be extended to a facility for balancing the load amongst them, by moving processes intact from more to less heavily loaded nodes.

R4] **Multiple-peer servers**. Multiple-process computations can be constructed to act as servers in performing resource-intensive processing for other applications. An interesting class of reconfigurations for these is that of adaptations to varying loads or other run-time conditions, made by altering the communications connections between the clients and server processes involved.

R5] **System evolution**. In software systems used in process control, it is sometimes necessary to maintain the running state of the system even when part of it is being changed. For example, when the implementation of a software or hardware component (device and/or process or group of processes) is changed, reconfigurations are needed to replace it with a new one whilst connected processes continue to execute.

R6] **Fault tolerance**. A failed or unreachable process belonging to a distributed computation can in principle be replaced by a backup version of it.

The research presented here was concentrated on a framework for programmers needing to solve problems associated with issues R1 - R4 and, to the extent that they are related, R5 and R6.

1.2 RDCs

This dissertation describes a model of reconfigurable distributed computations called RDCs, and their operating environment. The processes belonging to an RDC are called incarnations. Incarnations are not simply a unit of execution of user code. Other incarnations are able to perform generic reconfiguration operations upon them which affect their communications connections, and this distinguishes them sufficiently from processes in general to merit a different term. An incarnation has its own mapped and protected address space, and multiple incarnations can execute at a single node. Incarnations are single-threaded.

Incarnations possess communications interfaces, and the definition of configuration given above applies to RDCs. Much of the research presented here concerns the design and implementation of the interfaces and interconnection components.

1.2.1 Reconfiguration Mechanisms

The model provides four types of mechanism exercised by existing incarnations to establish the initial configuration of an RDC or part of an RDC, and to reconfigure it:

1] Mechanisms for setting up under program control from within an RDC a data-dependent and/or resource-dependent configuration of a new group of incarnations. This can be programmed a) without concern for the order of creation of the incarnations and their inter-connection components, and b) without requiring the programmer of the configured incarnations' code modules to be concerned with how the configuration is established. (see R1 and R2 above).

2] Mechanisms for altering existing connectivity between incarnations, so as to change the set of incarnations which handle communications deriving from given sources. (see R4, R5).

3] Mechanisms for propagating communications interfaces between incarnations to create dynamically-determined connections between them, including multicast connections. (see R1, R2, R4, R6).

4] Incarnation migration. (see R3).

In short, it is possible to create, migrate, destroy, connect, disconnect and reconnect incarnations dynamically, and the agents performing these operations are incarnations themselves.

1.2.2 Transparency

A principal design aim for the reconfigurability of RDCs is that reconfigurations can take place whilst the affected incarnations continue, in some sense, to run. At worst, an affected incarnation should be suspended, and then unblocked when the reconfiguration is over. This leaves unanswered, however, the question of whether an incarnation is semantically affected: the question of transparency of reconfigurations.

The second principal design goal is for controlled (that is, not failure-induced) reconfigurations to be transparent wherever possible and appropriate, in order a) to avoid complexity by separating concerns between application algorithms and reconfiguration management, and b) to avoid having to recompile code when new kinds of reconfiguration are required.

Transparency is not always appropriate. For example, this is the case where a server has to authorize a client newly connected to it before processing its requests. The server requires knowledge of the connection. In cases where transparency is appropriate, reconfiguration mechanisms are required to operate transparently. It will be shown in Chapter 6, however, that this is not a sufficient condition for transparency of reconfigurations with respect to application algorithms. We distinguish between:

1] **transparency for clients:** the transparency of a reconfiguration which an RDC or part of an RDC undergoes, with respect to a client incarnation – i.e. one which makes requests of it. Such a reconfiguration involves reconnecting the client to a new server incarnation, or it involves the migration of the existing server incarnation. This transparency can be achieved in the RDC model.

2] **transparency for servers:** the transparency of a reconnection of a client from one server incarnation to another, with respect to the server incarnations.

If a server has state with respect to its client in the second case, reconfiguration cannot be performed transparently if application-specific consistency is to be maintained. Application-dependent configuration management strategies have to be built upon the RDC reconfiguration mechanisms. The requirements for these strategies will be analysed in Chapter 6.

1.3 The Implementation

An environment has been developed which supports multiple RDCs run by multiple users. The RDCs have access to a shared collection of nodes which are managed as a pool. The main products of the research are:

1] a set of primitives to establish an RDC's configuration; primitives for communication between the incarnations; and a set of reconfiguration primitives. All are implemented as system and library calls made from the C language.

2] a purpose-built kernel which runs at each node. This was designed to be compact, with minimal facilities to support the communication and reconfiguration primitives. Effort was concentrated on support for communications and incarnation management, with incarnation migration regarded as a particular challenge. Apart from supporting the RDC programmer's model, the kernel was also designed to facilitate the implementation of node allocation management software as an RDC, i.e. in user-level code.

The kernel is part of an operating environment constructed to support RDCs – marketed recently under the commercial name Equus [EQUUS89a, EQUUS89b, EQUUS90], but previously known as Wormos [WORMOS86, WORMOS87] – which has been implemented on two distributed systems in distinct phases of its development. The first is a set of 7 Motorola 68000-based processor cards connected via a Cambridge Ring local area network and Multibus I bus, and the more recent a set of 11 Motorola 68030-based processor cards sharing a VME bus with a (host) card running the UNIX system V.2.2 operating system. Each system constitutes a pool of homogeneous nodes (although in the first system there are two different forms of hardware interconnection) hosted by a UNIX development, execution and input/output environment.

In addition to the kernel, Equus incorporates software to provide UNIX file system and windowing facilities to RDCs via the host workstation. Furthermore, the second implementation incorporates an RDC called the pool manager, which is responsible for allocating nodes to client RDCs, and also for performing load balancing by migrating the incarnations of client RDCs amongst a subset of the nodes.

1.4 Dissertation Overview

Chapter 2 describes in further detail the background to this research, examining resource sharing in existing systems with processor pools and idle workstations, models for distributed programming and approaches to reconfiguration connected with them. It ends with basic design decisions and the requirements to be addressed by this dissertation, including some illustrative problems to be solved which are based around an example RDC.

Chapters 3 to 8, the main chapters, elaborate the programming model and the implementation.

Chapter 3 begins with a description of the execution environment constituted by Equus. It goes on to describe incarnations: the ways they can be created and their main features. It outlines the basis upon which an RDC is run and allocated nodes from the pool. It describes an interface to node allocation software, which RDCs use to map their incarnations.

Chapter 4 describes inter-incarnation communication semantics and the structural components that make communications possible. The latter are motivated by the requirement to support streams of messages and queues of messages, both of which are the subjects of communications reconfigurations. The description relates the communications design to the implementation considerations that constrained the choices made.

Chapter 5 then develops the basics of the model presented in Chapters three and four. It shows how the configuration of a set of incarnations can be declared, and how the run-time system realises the configuration. It describes stream space, a construct first developed as part of this run-time system which has an independent rationale as a means of propagating communications interfaces and other components between incarnations.

Chapter 6 concentrates on reconfiguring incarnations considered as clients and servers of one another. It describes the mechanisms for changing the incarnation which is to receive and process streams and queues of message requests. It describes general requirements of the application-level response to these reconfigurations, and to reconfigurations in which an incarnation has to be withdrawn and not replaced.

Chapter 7 describes the mechanisms provided for controlling incarnations and monitoring events occurring in connection with them. They are described in the context of the requirements for node allocation and load balancing, and of the need to enforce RDC termination. Design choices made for incarnation migration, in particular, are covered in this chapter.

Chapter 8 is concerned with the main implementation issues not already covered. It describes the architecture of the Equus kernel. It gives an overview of the communications mechanisms upon which all interactions are constructed, and in particular it describes the re-routing algorithms used in the implementation of incarnation migration and the reconfiguration mechanisms described in Chapter 6. Finally, it gives performance figures for communication primitives and reconfiguration mechanisms.

Chapter 9 concludes the dissertation.

Appendix A presents the experiments which were performed to measure the implementation's performance.

Appendix B gives a brief description of each of the Equus calls.

Chapter 2
Background

This chapter outlines the distributed computer systems and programming models which form the background to this work. The starting points for the design are, firstly, requirements for managing processor pools in the distributed system upon which reconfigurable applications are to run, and, secondly, the model of processes and their interactions whose reconfiguration is at issue. Having examined related work for both these, the chapter goes on in Section 2.4 to give some examples of reconfiguration problems, and to state the design requirements addressed by the following chapters.

2.1 Processor Pools

The 1980s saw a shift away from centralised and time-shared mainframe computer systems to the provision of personal computers and workstations for individual users. The costs of hardware and the continuing need to serve groups of users with similar facilities have led in turn to distributed computing environments in which users' computers perform chiefly as intelligent user interface devices with fast interactive response times. Large discs are shared between them under the management of server machines, which are usually specialised for performance and run no other applications for protection reasons.

For these distributed systems, the chief advantage of a processor pool is that it provides large amounts of processing resources which can be shared and allocated dynamically in such a way that a number of users at any one time can benefit from processing power beyond that of a normal workstation, without the provision of individual extra processor nodes in every user's workstation. The capacity of a pool only needs to match the average instantaneous global demand within the distributed system, which is likely to be much less than that of all the users exerting their peak demand simultaneously.

2.1.1 Purpose-built Pools

In the Amoeba system [AMOEBA90], the process server chooses a node from the pool to run every new process it is requested to create. Each node runs the Amoeba kernel, but there is some hardware inhomogeneity in their pool, and clients of this service have to state whether floating point hardware is required and how much memory is required. The pool's computers are mainly single VME boards with 68020 and 68030 CPUs (these chips are binary upwards compatible). Although mention is made in the Amoeba literature of bulk allocation of pool nodes to run their parallel *make* application (a program to construct a binary from given targets), it is not clear how this facility is integrated with the process server's choice of node, or what happens if, for example, twenty nodes are requested but only ten are available.

The processor bank of the Cambridge Distributed System contains several incompatible machine types which run between them several different operating systems [CRAFT]. Software called the resource manager (RM), which itself runs on nodes in the processor bank, constructs for its clients high-level resources such as the Tripos or Mayflower operating systems, or a compiler, sometimes involving more than one machine. It also offers to its clients system services which have been constructed independently in the distributed system, and which are offered and withdrawn dynamically through its auspices. For the purposes of making allocation decisions, every resource managed by RM is in one of the states *free*, *worm* or *allocated.* A *free* resource can be allocated unconditionally. Before becoming *allocated*, a node preloaded with an operating system kernel can be used to run a process called a worm segment. In this case it can be reclaimed by RM at any time, but in the meantime by running the process it performs useful work for a low-priority application of a type known as a worm program, developed elsewhere originally, for utilising idle workstations (worm programs are discussed in Section 2.1.2).

Parallel Processing Machines

Whereas RM concentrates on managing a diversity of layered resources, the homogeneity of the Amoeba pool lends it to being regarded for some purposes as a parallel processor whose nodes can be allocated to distributed computations. This view is reversed when a parallel processing machine is added to a distributed system, for then its nodes can be regarded as the nodes of a shared processor pool, allocated in blocks for parallel processing by separate applications. In particular, there are now several commercially available machines containing up to the order of several hundred interconnected transputers [TRANSPUTER], and of the order of tens of transputers can be hosted within standard personal computers and workstations. Transputers are devices combining a powerful CPU with communications link controllers on a single silicon chip: they are designed to be linked to communicate with other transputers. They are aimed at applications constructed from processes passing messages to one another. In particular, transputers are aimed at applications constructed according to the CSP model [CSP], from which the occam language for programming transputers derives [OCCAM]. The Helios operating system design includes a task force manager (TFM) to map users' parallel applications onto a "network of communicating sequential processors which do not share memory" [HELIOS] – typically interlinked transputers housed in machines connected by a local area network. The TFM matches resource requirements specified by users (in terms of node characteristics such as CPU type and amount of memory) to available nodes. It partitions the nodes between users as it does so: current generations of transputers do not support memory protection, so that they have to be allocated exclusively to users to avoid interference from buggy programs.

2.1.2 Idle Workstations

A number of facilities to use idle or under-utilised workstations as a fluctuating pool of processing resources have been implemented: at Carnegie-Mellon University, originally for the Spice system [DANNENBERG] and later for the Andrew computing environment [NICHOLS]; at Xerox PARC, originally on a network of Alto workstations [WORMS] and later for the Cedar computing environment [HAGMANN]. Usage varies from a command available to run a single user's command at an idle machine (Andrew), to a facility to run distributed computations (the worms of Shoch and Hupp). A particular design problem is what to do when a user logs back onto their workstation. The designers of the Sprite network operating system [SPRITE] and the V-system [V85] cite the utilisation of idle workstations as a motivating factor behind their process migration facilities: when a user logs back on, processes utilising the workstation can be migrated back to their source machine or to another idle machine. Worm segments, however, are simply lost (whether at Cambridge or Xerox PARC).

Worm Programs

A worm consists of a number of segments, each running on a single machine, and each possessing the facility to communicate with the other segments of the worm. As implemented at Xerox PARC, there is no central resource management scheme, and therefore no arbitration between worms run independently, as to which can create a segment at which machine (each machine supports at most one segment). The worm mechanism provides facilities for worms to monitor the liveness of segments and to locate idle nodes and start up new segments there. Applications were built on top of this mechanism, and several examples were constructed, both to perform directly useful work and to exercise the mechanism – the Existential worm endeavours to maintain a constant population count of segments, somewhere in the network. A worm begins by locating idle machines and establishing an initial population of segments. If a segment becomes lost or unreachable through crashing, network partitions or when a user reclaims a workstation by logging on, the worm can reconfigure by locating new idle machines and creating replacement segments there. A worm is thus required to adapt not only to the number and locations of workstations which happen to be idle when it runs, but also to changes in this working set of machines whilst it runs. The main features of worm programs, then, are their ability to exploit whatever processing resources are available to them, their ability to recover from some failures, and their degree of autonomy: worm programs configure and re-configure themselves.

2.2 Distributed Programming Models

Many distributed programming paradigms have been developed over the last two decades, and although some have proved more widely used than others, no one model has proved generally superior, and nor is there a consensus on which paradigm best suits which problem. There is a broad split between the language-based approach (e.g. [ACTORS, ARGUS, BRINCH, CSP, ORCA, POOL]), and the distributed operating system approach (e.g. [AMOEBA90, CHARLOTTE87, CHORUS, DEMOS87, EMBOS, MACH, ROSCOE, V88]). The former approach is concerned primarily with programming single, autonomous applications; the latter is addressed additionally to the interactions between separately written and compiled programs, particularly those interacting according to the client-server model.

Whatever the model, there are two views of a distributed application. The algorithmic view is a static view. It is described by giving the function of each programmed component, including local knowledge of the functions of any components it interacts with. The other view, addressed by this dissertation, is concerned with the application's configuration, as defined in Chapter 1 in terms of its dynamic structure and mapping.

In some paradigms, structure is closely related to application algorithms. Object-oriented programming [JONES] is an important model in this category. Objects encapsulate procedures applied to local data, whose application is requested by other objects, viewed abstractly as operations or invocations made upon the object. Objects can, like processes, have one or more independent threads of control. The structure of a program written in the object-oriented style can be highly complex and dynamic. Suggestions for reconfiguring these programs have been made, however, and include schemes for migrating objects between nodes to improve performance [EMERALD], and the interposition of entities called proxies [SHAPIRO] which trap invocations and dynamically determine the objects which handle them, to improve performance or provide fault tolerance. ISSOS ([ISSOS], discussed in Section 2.3.1) does reconfigure collections of objects, but ones whose structure is defined to be largely static, undergoing small changes known as adaptations.

The parallel programming language Actor [ACTORS] is also used to create dynamic and potentially highly complex structures, this time of lightweight processes called actors which pass messages to one another. They pass messages asynchronously, to maximise concurrency. The actor model is notable for the high degree of inter-relation between structure and algorithm [HEWITT]. Networks of actors are dynamically constructed as part of the algorithm, to perform, for example, a recursive factorial computation with an individual actor evaluating each product.

We turn now to paradigms in which greater separation is possible between configuration management and application algorithms than in the object and actor models.

2.2.1 Loosely Coupled Processes

Since RDCs are to be able to act as servers, incarnations belonging to separate RDCs have to be able to interact, even though they were developed separately. This is a similar requirement to that met by the use of pipes in UNIX [UNIX] to couple the output of one program to the input of another. The versatility of this scheme derives from the fact that each program is transparent to the other one. Processes whose code is independently written and compiled, but which are nonetheless able to interact through software interfaces, have been called loosely coupled [LYNX]. This is in marked contrast to the concurrent programming language CSP, in which all processes that are to interact with one another have to be written, compiled and linked together.

The other properties which characterise loosely coupled processes are:

1] Loosely coupled processes do not share physical memory. This is not just because they are independently compiled. It is also because in general they run at physically disjoint nodes in a distributed system. Even if processes can reside at the same node, use of shared variables through physically shared data segments precludes a reconfiguration in which they reside at different nodes.

2] Loosely coupled processes are heavyweight. Running at a multi-user, multi-tasking node, they execute in separate mapped and protected address spaces which are expensive to create, and therefore they must perform a significant amount of work for their existence as separate processes to be justified. A number of distributed operating systems such as Mach provide kernel support for lightweight threads running together in a single processes address space [MACH], which are therefore less expensive to create; and they additionally facilitate the multiplexing of a process's work so that a task can be performed by a thread when another thread blocks. However, in all systems supporting threads it is the host process which is considered the main unit of configuration, and threads are encapsulated from other processes interacting with a multi-threaded process.

2.2.2 Approaches to Interaction

There are two commonly used models of interaction between loosely coupled processes: message passing and remote procedure call (RPC).

Message passing is the means of interaction in a number of languages besides Actor – for example, the Conic language [CONIC89a] – and it is a fundamental facility of distributed operating system kernels [AMOEBA90, CHORUS, MACH, V88]. A message is a collection of data which is copied between processes via, typically, *send*, *receive* and *call* (send and implicitly receive) operations. The destination of messages is a process in the V-system, but is more commonly an intermediate entity such as a Mach port, or a local entity such as a Conic exitport, both of which allow for the receiving process to be dynamically changed.

Remote procedure call [RPC] is an attempt to extend conventional procedure call semantics to the distributed case, in which the process which executes the code of the called procedure (the server) is in general in a machine which is remote from the calling process (the client). The call's input parameters are despatched to the server and any results are returned to the client. The binding of the procedure name to the address of the network location used by the process which executes the remote procedure code can be carried out at run-time. It is not normally possible in RPC implementations, however, for the binding to be changed transparently at run-time, as we require if reconfigurable server RDCs are to be constructed. The designers of the Mercury system at MIT have extended RPC semantics [WEIHL] in two respects. Firstly, the target of a remote call is not a procedure but a port possessed by some active entity which accepts the call. The designers plan to allow the ownership of ports to change. Secondly, they have addressed a disadvantage of RPC, which is that it is synchronous. Like conventional procedure call, a remote procedure call returns only when the procedure has been executed. The Mercury design provides for calls to be made asynchronously over what they refer to as call streams, thus enabling interacting processes to work in parallel.

We turn now to work which has investigated the problems of reconfiguring distributed computations.

2.3 Systems with Reconfiguration

Systems addressed directly to the provision of facilities for reconfiguration differ in the types of configuration and reconfiguration supported, and in the degree of separation between application algorithm and configuration management.

2.3.1 Language-based Systems

Conic and CSR

Conic [CONIC89a] is a distributed programming environment developed at Imperial College, University of London, which addresses itself to language support for reconfigurable distributed programs. They have developed two languages used together: one for application algorithms and one for configuration. CSR [CSR87, CSR89] is a software paradigm developed at Bristol university as part of a project which aims at reconfigurability of components written in the language CSP.

Their work exhibits several commonalities: i) the attempt to separate issues of configuration and reconfiguration from application algorithms; ii) the use of system-provided configuration managers; iii) the use of components (CSR's controllers, servers and resources, and Conic's logical nodes) which employ message passing for interactions; iv) the ability to connect and disconnect externally the set of interfaces each component uses for sending and receiving messages, thereby constructing and altering the set of components each interacts with; v) target applications for each include process control and instrumentation. Reconfigurations are required to take place whilst an application executes, and so as to preserve its consistency, safety conditions such as the avoidance of certain operating conditions, and liveness conditions – the absence of deadlock or livelock. A typical reconfiguration considered is the integration of a new software component when hardware is added or switched on, or the removal of hardware, and consequently any components it hosts. These problems are analogous to those of exploiting a node dynamically allocated from a pool, or coping with its withdrawal.

In both systems, the application writer creates separate code modules from which executing instances are created. The application component interfaces used are CSR's channel stubs and Conic's exitports and entryports, all of which are typed according to the data type of messages which can be sent or received using them. The interfaces are in each case local names which hide the actual destination or source of messages sent through them. Language declarations determine the set of interfaces of each component statically, up to parameterised lengths of arrays of interfaces. It is possible for an interface to become disconnected from any other interface.

To configure or reconfigure an application, the systems interpret commands which are typed by a (human) system administrator or, additionally in the case of Conic, supplied in a script written in the configuration language or contained in a request from an application component. It is not clear to what extent the latter has been tried. The systems also provide a set of queries for users to examine the state of running applications.

Configuration is established and altered ultimately by an application-independent configuration manager, which constrains the connections that are made in order that only interfaces of the same type can become connected. Apart from typing constraints, the configuration manager allows component interfaces to be connected arbitrarily in each system. Whilst Conic does not promote any structural models over others, CSR applications are by definition configured in a hierarchy according to strict rules. This simplifies the task of a second system component, the configuration controller, in implementing a general strategy to maintain required consistency, safety and liveness properties for any CSR application. A reconfiguration command issued by a user of the CSR system, such as one to remove a certain process, is handled by the configuration controller. It examines a configuration database to work out which components will experience a disconnection when the component is removed. It synchronises with and communicates with the application, so that the disconnections and removals it requests the configuration manager to carry out do not compromise the application's consistency or safety or lead to deadlock. Application code runs inside a harness which is designed to hide from it the details of interaction with the configuration controller.

Despite the original aim of separating application algorithms from the details of configuration, the designers of both systems were led to study the particular ways in which, it turns out, the two are necessarily inter-related in significant cases. The CSR application harness and configuration controller achieved separation of concerns for the CSR paradigm, but its designer described the paradigm as "over simplistic, imposing a high processing overhead and leading to unnecessarily complex implementations." ([CSR89], p. 62). The Conic team have looked at a generic model of interaction between the application and configuration management, which enables the latter to preserve what they refer to as the consistency of the application [CONIC89b, CONIC89c]. This relies upon the application writer incorporating code relevant to reconfiguration in each component's code module. Firstly, to determine and assert its state of activity or passivity. These are its possible states from the configuration management viewpoint. They are each a function of the interactions between it and the components it is connected to. Secondly, to implement what are called initialisation and finalisation actions, which take place when a connection is made or unmade. Whilst the Conic scheme abstracts configuration states away from the particular application for management purposes, the application writer is still very much involved in enabling reconfigurations to take place, because he or she is concerned with these states and their transitions from within the application.

ISSOS

ISSOS [ISSOS] is a programming system developed at Ohio State university which is aimed at prototyping and tuning parallel programs. These programs are adapted by changing their structure, their implementation in terms of program versions and parameter values, and their resource allocation: the placement of processes onto processors and use of shared memories. The applications consist of a collection of objects written in C which perform invocations upon one another. Their configuration is described with COOL, an object-oriented extension to C. The paper [ISSOS] describes facilities to monitor an executing parallel application by examining the values of variables. Adaptations can be specified by the programmer as being required when guard conditions expressed in terms of application variables evaluate to TRUE. An example is given of the addition of a worker process when the length of a job queue from which a number of workers extract and execute tasks exceeds a given value. ISSOS also allows adaptations in the form of changes to application parameters whilst it is running, in order to affect its behaviour without changing the application's structure. The example given is an alteration to a sleep time in the objects which add items to the queue. This has the effect of changing the rate at which jobs are added to it.

ISSOS's stated aim is the separation of adaptations from application algorithms, and for this reason the monitoring and adaptation mechanisms described are transparent to the application. However, it is not clear whether the synchronisation which sometimes proves necessary between application and configuration management, as shown by the Conic and CSR work, is addressed or provided for.

HPC

HPC (Hierarchical Process Composition) is a complex model of interprocess relationships developed at the University of Rochester, USA [HPC]. It provides a set of primitives used to connect and disconnect components dynamically, and is described as allowing "user-defined adaptations to failure or environmental changes." An HPC operation domain is a special case of an object of a type known as a shell. A shell exports communications interfaces to the outside world, and encapsulates the configuration of the objects inside it. In an operation domain, one of these objects is a process known as a controller, which is responsible for modifying and maintaining objects within it. This includes alterations to the population of objects and to their interconnection, both with one another and with the interfaces which the shell exports. Such alterations are the adaptations to failure or to environmental changes – such as load conditions – referred to above. HPC incorporates two features not found in the systems described so far:

1] there is no single, system-supplied configuration manager. Every application has a controller (or, rather, nested set of controllers) which acts as its own application-specific configuration manager. Moreover, the controller is able to establish a control channel with each of the objects in its domain, for reconfigurations which require the consent of the affected objects.

2] HPC provides for the interconnection of applications through exported interfaces. A design aim is for any reconfiguration performed by one application to be transparent to the other.

LADY

The LADY programming language and environment was developed at the University of Kaiserslautern, FRG, for the implementation of distributed software systems, with a special focus on operating systems. Because of this focus, it contains features for dynamically establishing and changing program configurations from within applications, as opposed to the external configuration management approach of the models above.

Like many other distributed programming languages, and like all distributed operating systems, LADY allows its main execution components, multi-threaded heavyweight processes called teams, to create further teams. LADY thereby allows for data- and resource-dependent structures. Teams are inter-connected via long-lived components called logical channels and logical buses, which also can be dynamically created so as to interconnect dynamically-created teams. Team interfaces called output ports and input ports come to be connected via these components. A logical channel is a one-to-one connection like a Conic or CSP link, but logical buses are for one-to-many or multicast communication.

Multicast Communication and Process Groups

A multicast facility has been incorporated into several languages, programming systems and distributed operating systems. Its uses include sending notifications, queries and updates to a group of processes performing parallel processing or managing distributed data [V88, KAASHOEK], and tolerance of failures through replication of processes [ISIS]. An important feature of multicast is that multicast addressing modes allow for the transparent addition and subtraction of processes to and from the multicast group that receive messages. How and whether to achieve reliability, atomicity and transparency for multicast semantics, and what tradeoffs are required to achieve efficiency, are all subjects of continuing research [KAASHOEK].

Distributed Shared Data

Linda [LINDA] is an extension to C (and other languages) comprising primitives to access data within a construct called tuple space. Linda processes only interact with Tuple Space, and never directly with one another. They communicate by depositing data items called tuples into tuple space, and taking them out or reading them. A tuple is an ordered set of data items. To request a tuple, the values of zero or more fields are given by the requestor. The other fields are each given a variable of appropriate type in the request. The request is returned a tuple which matches it in the given field values and types. The request is blocked if necessary until such time as a matching tuple has been deposited. Typically, a master process deposits a collection of tuples describing tasks, which are executed by a collection of worker processes which obtain them from tuple space. The coherence of shared tuple space is guaranteed by the run-time system, even though it is divided across machine boundaries.

Linda is one of several approaches to distributed programming using shared data surveyed by the designers of the language Orca [ORCA]. In Orca, processes communicate and synchronise via passive objects which encapsulate the shared data. A shared task queue is an example of such an object: a master process adds items to this queue, and worker processes retrieve tasks from it. The run-time system ensures that these operations are performed indivisibly.

These models emerged after the work presented in this dissertation was begun. Their attraction in relation to reconfigurability is the lack of direct process interaction. They avoid altogether the problem of connection and disconnection of processes. Only a master process overseeing an application need be aware of the addition or withdrawal of a worker process to or from a running computation. However, they do not satisfy the requirement of loose coupling of processes. They are aimed primarily at single applications with private shared data spaces, and it would be awkward to extend this to a system of multiple shared object/tuple spaces which overlap sufficiently for the connection of loosely coupled clients and servers. There is also a lack of agreement about their scalability to large collections of nodes, but the need to maintain coherence of the shared data across all nodes must impose some limits on the number of nodes that programs written according to these models can exploit.

2.3.2 Distributed Operating Systems

Distributed operating systems are designed to manage the activities of loosely coupled processes which interact largely as clients and servers, often employing RPCs. A kernel is typically constructed to provide a few process management and message passing primitives, whose design and implementation aims for location transparency and efficiency. Most designers try to keep the kernel small and robust, with higher-level services such as the file service implemented in user-level code.

Then they can be improved, relocated and extended without interference with the kernel or with other services.

The structural components supported by different distributed operating systems to connect loosely coupled processes are now examined.

Link-based Systems

Links are used in [CHARLOTTE87, DEMOS, EMBOS, ROSCOE] as the sole means of interprocess connection. They are intended for long-lived connections and are designed to provide protection, so that the set of processes able to send messages to a given process can be controlled. Unlike the linkage constructs of the models described above, links exist separately from processes, and link ends can be propagated from process to process by being enclosed in a message, thus establishing dynamic communications connections. In particular, a client can be connected to a server by having a link end connected to it sent to the client in a message, typically by a name service with which the server has registered itself. Link models vary in whether links are unidirectional or bi-directional. In the former case, the sending end can be duplicated so that multiple senders can be connected to the same link end held by a server process. Charlotte's bi-directional links can each connect only a single pair of processes. In some models (Charlotte, for example) a link end which is used for receiving messages can be sent in a message, thereby altering the server process which handles clients' messages. Link ends are represented by local names within a process, and this reconfiguration is transparent to the sender.

Link-based models are designed for programs to handle the connection of loosely coupled processes – i.e. for a case in which, unlike Conic/CSR/ISSOS/HPC processes, processes do not have a statically defined and comprehensively known set of communications interfaces. Moreover, for protection and to manage the multiplexing of different clients' requests, servers can require knowledge of the clients connected to them. So the requirements for configuration transparency differ in this respect from those of, say, the Conic model.

Port-based Systems

Ports are found in a number of important distributed operating system designs [ACCENT, AMOEBA90, CHORUS, MACH]. Each process possesses a number of ports, which are local message queues from which it has the sole right to receive messages. Other processes are able to send messages to these ports by virtue of possessing send rights. Send rights can be propagated in messages between processes, and in some cases receive rights can, too. The propagation of send rights and receive rights have the same effect on process connectivity as the propagation of link ends, and the Accent and Mach port models are no less connection- and protection-oriented than link-based models.

In some designs, ports differ significantly from link-based models, however. Firstly, the possession and propagation of send rights is not always constrained by the kernel, as it is with links. Access to server-managed resources is instead controlled at user-level through use of capabilities. Amoeba capabilities, for example, are data items fabricated by servers. The capabilities contain server port identifiers and refer to the objects which the servers manage. Capabilities determine the rights of the client possessing them to perform operations upon the corresponding objects. They are designed to be difficult to forge. A second difference from links is that Amoeba and Chorus ports can both be collected into port groups for the purposes of multicast communications, as in the LADY programming system described in Section 2.3.1.

Migration

Process migration mechanisms have been designed for DEMOS/MP, Accent, Charlotte, Sprite, LOCUS, the V-kernel and MOS [DEMOS83, ZAYAS, CHARLOTTE89, SPRITE, LOCUS, V85, MOSa]. Migration is motivated by:

1] **load-balancing**: It has been suggested for use in attempting to balance the computational load on a group of nodes so as to improve throughput for a set of jobs, over that possible with a static assignment [NI, MOSb].

2] **dynamic node withdrawal**: when this occurs in a pool-based distributed computer system, a possible means of recovery is to migrate the affected processes to another node, as already described for the V-system and Sprite network operating system.

3] **co-location of communicating processes**: If two processes communicate synchronously and frequently, it may be most efficient for them to be hosted by the same node rather than to incur the costs of remote communication. Process migration can be used to perform co-location as processes enter into and leave such patterns of communication with one another.

4] **an alternative to swapping**: If insufficient memory resources are available for a process to run any further, it may be possible to migrate it to another node which does have sufficient memory resources, rather than swap the process out to disc.

5] **fault tolerance**: Process migration has been suggested for building fault tolerant systems [RENNELS]. Under certain failure conditions, sufficient process state is available to re-establish the process at a working node.

The problems in the implementation of a migration mechanism stem:

i] from the need for transparency, both with respect to the migrated process itself and to those it interacts with.

ii] from the need for reasonable performance, so that efficacious load-balancing and co-location strategies can be supported; and so that minimum disruption is caused when processes are migrated away from nodes when they are about to be withdrawn.

iii] from the problem of residual dependencies. This is where a process has to leave kernel state or an object such as a file upon which it depends at a node it has migrated from. This means that previously local operations now require communications, and it means that failure of the previous host will affect the migrated process.

2.4 The RDC Model

This section now presents the basic design decisions made and design issues addressed by the research presented in this dissertation. It relates these to the survey of background work just presented.

2.4.1 Design Decisions

Node Allocation

The present node allocation requirement is to provide a framework within which a variety of policies can be implemented and tested. Existing practice as represented by the distributed systems just examined led to the following general characteristics of node allocation to base the rest of the design upon.

1] **It was decided to provide time-sharing at each node, and to allocate node resources to RDCs according to a system-defined policy exercised by a resource manager.** These decisions were taken in the light of the deficiencies of the Xerox PARC worm system. These were that there was no arbitration between worms, and no way for a worm to run if another worm had already gathered all available nodes (Section 2.1.4). Each of the distributed computer systems used for the implementation described in this dissertation consists of a pool of processors accessible via connected workstations, and the resource manager is therefore called the *pool manager*.

2] **Resources are to be managed at the level of nodes, and there are two types of node allocation.** In the first, the pool manager chooses a node for every incarnation it is asked to create (like Amoeba's process server). In the second, nodes are allocated in blocks on-the-fly, and the application decides how to exploit them, as is the case with worm programs in the Cambridge System (Section 2.1.1). The construction of resources at a higher level than nodes, as performed by Cambridge's resource manager, is beyond the scope of this work.

3] **Mechanisms are to be designed for use in enforcing priority schemes by withdrawing and re-allocating nodes.** Whether a pool is purpose-built or consists of idle workstations, existing practice is to allow some users or applications to take priority over others. In the idle workstation case, priority at a node means the eviction or destruction of other users' processes; in the Cambridge processor bank, priority over a worm segment leads to its destruction.

4] **Incarnation migration is to be implemented**. This is to make recovery possible from node withdrawal and to provide a framework for load balancing (Section 2.3.2).

The RDC Model

The RDC model is aimed primarily at long-running, computationally intensive applications such as image processing. RDCs can be programmed to operate as autonomous applications, or as servers providing facilities to loosely coupled clients.

5] **The computational model is connection-oriented**. This is to enable configuration management to take place, and to suit the long-lived series of interactions which incarnations are expected to engage in.

6] **Communications are to be in the form of message passing primitives.** These are to include asynchronous primitives for parallelism and multicast for addressing groups of incarnations (Section 2.3.1). Although the development of language-level RPC or object invocation semantics is beyond the scope of this work, run-time support for both can be implemented on top of an appropriate choice of message primitives. Synchronous message primitives supporting request-reply interactions efficiently are also necessary in view of this.

7] **Any incarnation can create other incarnations, and any incarnation can manage other incarnations – within an RDC, or as a service to other RDCs.** There is no system-defined functional hierarchy of incarnation types, like the nested controllers and basic processes of HPC (Section 2.3.1). The HPC approach to process management was considered too protection-oriented for the present concerns. But it was decided to provide protection against unwanted management operations and communications for the multi-user development systems.

8] **Declarative configuration establishment primitives are required.** These are to be called from existing incarnations which dynamically decide upon the configuration. One of the features of Conic's configuration language (Section 2.3.1) which is to be used is its declarativity. The programmer of both configuration code and application algorithm code is to be isolated wherever possible and appropriate from the operational details of creating incarnations and their connection components.

9] **Mechanisms are required which enable sets of incarnations to acquire connections with one another explicitly.** Applications written according to the object and actor paradigms exist in which active entities are dynamically created and connected according to – and not transparently to – the algorithms (Section 2.2). This suggests that having only reconfiguration mechanisms which are externally imposed upon application incarnations would be a limitation, and that application-initiated decisions to create a new incarnation and connect it dynamically to the existing computation should be possible in the RDC model.

The Operating Environment

10] **All the above mechanisms are to be realised by a purpose-built kernel.** This is so that they can be performed efficiently.

11] **User-level code implementations of node allocation and dynamic scheduling for load balancing are to be possible.** These facilities require testing and

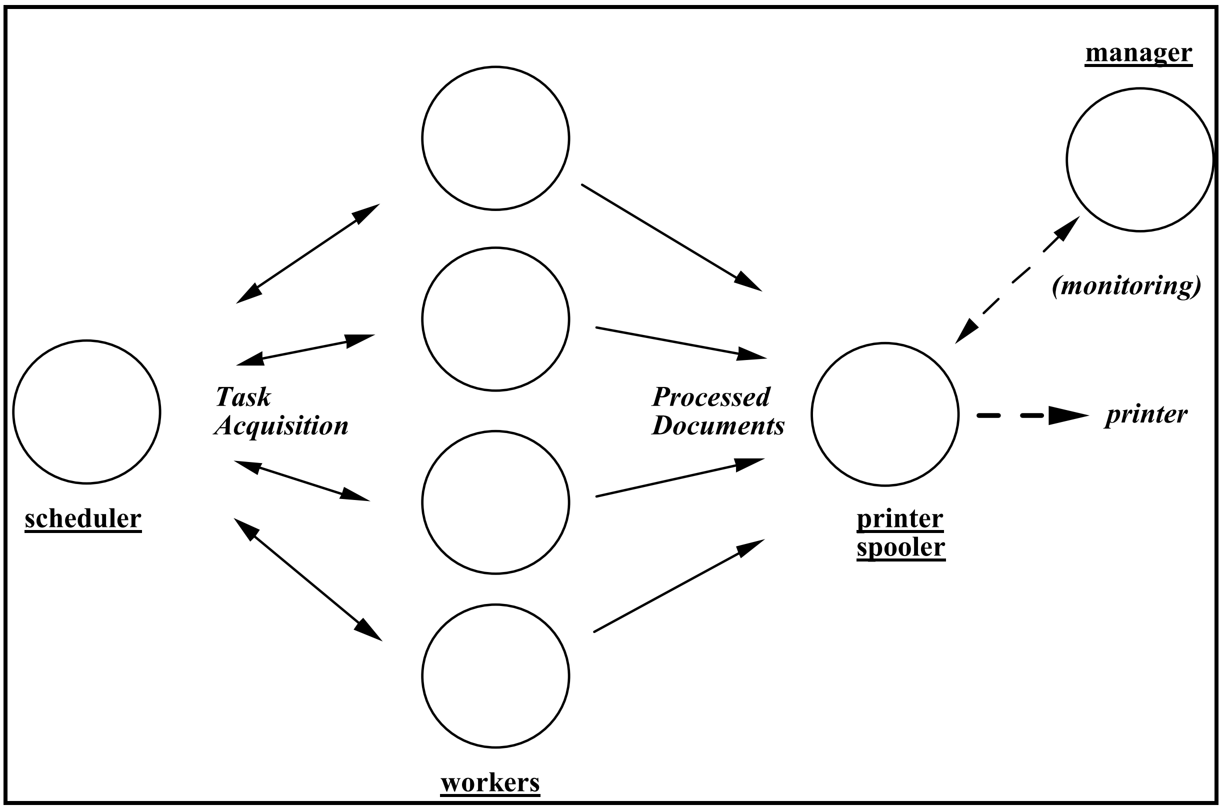


Figure 2.1: Initial Configuration of Text Processing RDC

adjustment. By implementing them at user level, they are made more amenable to change than if a kernel implementation was produced.

12] **The kernel is to provide mechanisms to enforce RDC termination.**

2.4.2 Aspects of Reconfiguration

To illustrate the aspects of reconfiguration addressed specifically by the research presented in this dissertation, several problems are now introduced. These are related to the design of a text processing RDC and a printer server RDC that derives from it. The examples exhibit features which are addressed and solved directly by the mechanisms of the RDC model. The examples also show that there are application requirements which must be met before reconfigurations can be safely applied.

Figure 2.1 shows a set of incarnations which between them process in parallel and print a set of documents.

The scheduler incarnation is given the list of documents to process at the start, and each of a set of worker incarnations repeatedly acquires a document from it (by name) and processes it. The scheduler-worker structure is a widely used paradigm in parallel processing of multiple independent tasks (see, for example, [HAMILTON]). All the worker incarnations, and therefore their nodes, are kept busy as long as there are tasks left to perform, since they acquire the next task as soon as the last is complete. It is a structure of particular interest here, since it is in principle possible to add a worker transparently to the scheduler.

The worker incarnations send the processed data asynchronously for printing. This data is received by a printer spooler incarnation which buffers and collates it as necessary before directing it to a local printer. It is monitored by a manager incarnation. For the present purposes, monitoring is assumed to consist of printing at a display messages concerning performance parameters such as the number of documents that have been printed so far, and the length of the queue of requests from the workers.

The manager incarnation is assumed to exist at the start, and it establishes the rest of the RDC. It has determined in this example that four processing nodes are to be used, and a worker incarnation has been created at each. Given the nature of the application, it is to be expected that little would be gained by having any of these nodes host more than one worker incarnation. The printer spooler requires local access to a physical printer. The scheduler performs a trivial computation when it allocates work; but it is a potential bottleneck, and whether or not it should share a node with a worker depends on the number of workers and the mean document processing time. The manager incarnation's mapping depends on the communications patterns caused by the monitoring activity, since otherwise it imposes little processor load. In short, the optimal mapping of the RDC is a function of the application's behaviour and the nodes' processor and communications performance heuristics.

The structure of this RDC is both data-dependent and resource-dependent: the optimum number of worker incarnations is at most the number of nodes available which can host them, but no more than the number of documents, since a document represents the quantum of work allocation. It is assumed that initially there is only one printer spooler. There are other printers attached to other nodes, but these are left by default to other users.

The aspects of reconfiguration to be considered are as follows:

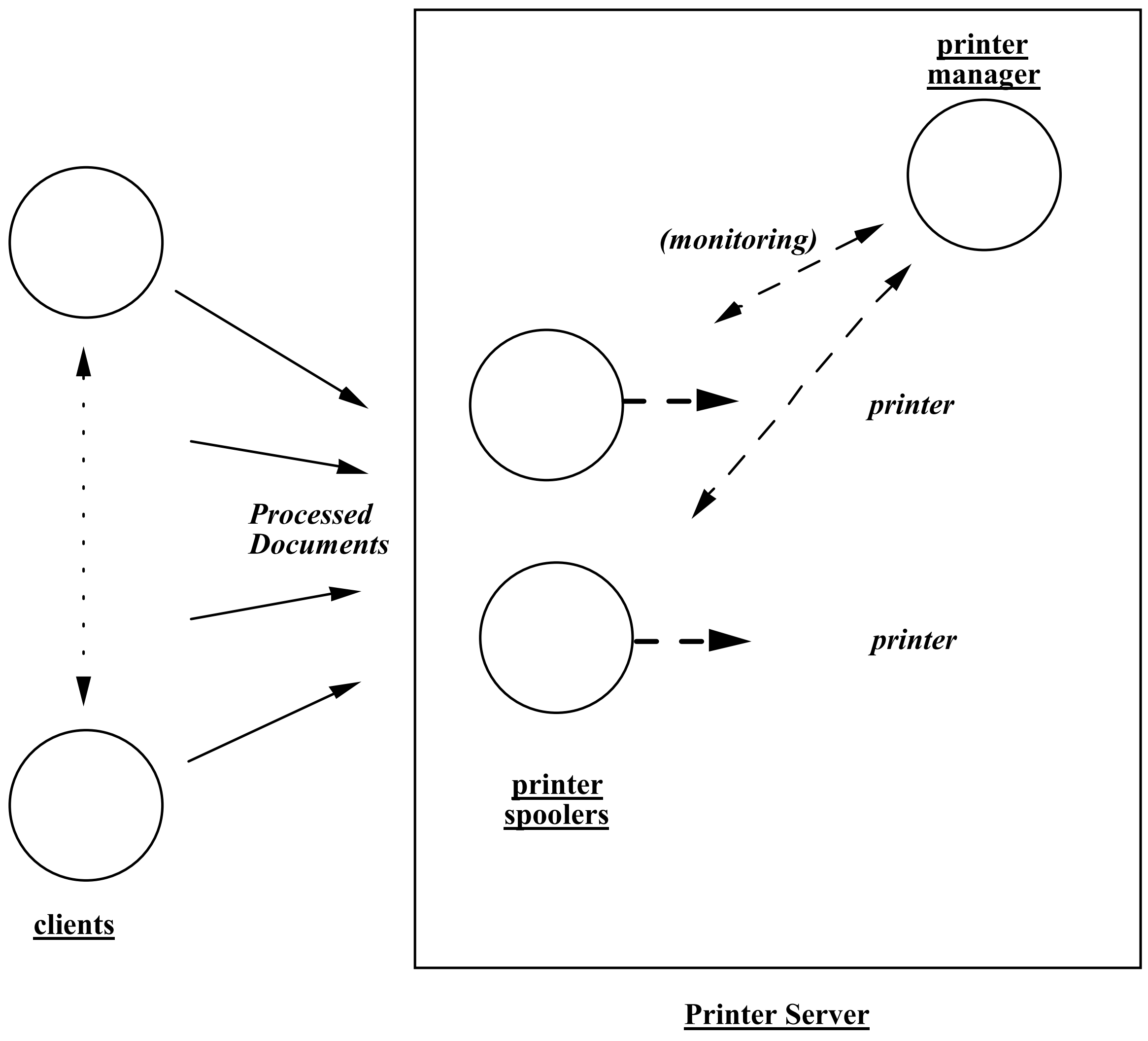


Figure 2.2: A Printer Server RDC with Clients.

1. **Configuration Establishment.**

   Through what mechanisms is the configuration established by the manager, given that the configuration is a function of run-time conditions (the number of nodes available and the number of documents)?

If there are still sufficient documents to process at a time when an extra node is available for allocation, it could be decided to create a new worker incarnation there:

2. **Dynamic Addition of a Worker.**

   Through what mechanisms can a new worker be dynamically created and connected to the scheduler and printer spooler? To which incarnations can this reconfiguration be made transparently?

In some circumstances, the rate of throughput of processed documents is important enough, and the current rate of throughput being monitored is low enough, to add a second printer node, and hence printer spooler incarnation, to the running RDC when one becomes available.

3. **Dynamic Addition of a Spooler.**

   Through what mechanisms can a new printer spooler be created and connected to the existing incarnations so that the workload is shared between the two printers? To which incarnations can the reconfiguration be made transparent?

Figure 2.2 shows a different view of the printer incarnation population, as being that of a separate printer server RDC which services a number of independent client incarnations that are dynamically connected to it. The server RDC uses one or two printers according to the level of demand being placed upon it, and according to whether the second printer is available or has been allocated for some other purpose which takes priority.

4. **Dynamic Addition of a Spooler to the Printer Server RDC.**

   How can the printer server be designed so that it can share work between the printers, but now given the variable number and unknown identities of the clients?

Finally, there is the question of reversing these reconfigurations:

5. **Dynamic Withdrawal of a Worker or Spooler.**

   After the reconfigurations of 1 - 4 have been made, how can recovery be made if the extra node is withdrawn and i) a worker and ii) a printer spooler are affected?

2.4.3 Discussion

Statically declared and externally connected interfaces, as used in the Conic and HPC models, are desirable for transparent configuration establishment and reconfiguration. Where interfaces are known, the example problems can be solved, at least at a mechanistic level, by disconnecting and reconnecting interfaces externally. But this is not suited to managing connections between independent clients and servers.

The ports and links of distributed operating systems, on the other hand, provide mechanisms for connections and reconnections between loosely coupled clients and the printer server. But these models are limited in their support for transparency. Connections would have to be explicitly acquired by the incarnations of the text processing RDC as their configuration was being established. And reconnections appropriate to spooler addition or withdrawal would have to be made explicitly by the spoolers, by transmitting port rights or link ends.

A synthesis of these communications connection models is therefore required. The design by which the RDC model achieves this will be developed in Chapters 4 to 6.

At an application level, both the scheduler and the printer spooler incarnations may retain state between communications with workers or clients which could lead to inconsistencies if they were disconnected arbitrarily. It will be shown in Chapter 6 how, in the unbuffered Conic connection model, clients of the printer server would have to be brought into a specific state in an application-dependent way before reconnection to a different printer spooler could take place safely. This is wholly unsatisfactory, and it will be shown in Chapter 6 how the RDC model dispenses with this requirement by incorporating message queues as objects of reconfiguration.

2.5 Summary

This chapter has considered related approaches to programming and reconfiguring distributed computations, and described pool-based management schemes. The research work has been set in the context of backgound work in the previous section, and examples have been given which manifest some specific issues of reconfiguration which are to be addressed. The next chapters go on to elaborate the design of the RDC model and operating environment.

Chapter 3
RDCs

This chapter begins the description of the RDC model and its execution environment with preliminaries for the next chapters' treatment of configuration. It covers the construction of RDC binaries from code modules. It describes and gives rationales for the basic model of incarnations, and the mechanisms available to create new incarnations. It describes the sequence of events which occur in order to initiate the running of an RDC, and gives a brief discussion of node allocation policies used with the development processor pools.

3.1 Modules and Incarnations

An RDC program is constructed from one or more separately compiled components called modules. Modules are used by the programmer to effect the functional division of an application. An example of this from the document processing program of the last chapter is that the scheduler and worker components would be programmed as two separate modules.

The term "incarnation" was chosen because each incarnation is created from a module, but there are in general several incarnations of each module executing as part of an RDC, with separate identities. Being executing images of the same code, incarnations created from a given module perform functionally similar roles within an RDC, but typically execute upon different data in parallel.

Each module is used for the creation of only one type of incarnation, which commences execution at the root of the module's main program. Incarnations are able to support recursion and standard libraries through stack extension and dynamic memory allocation. No language constructs have been added to the C language for programming RDC modules: access to the facilities available under this

```
main(argc, argv, envp, myparams)
    int          argc;
    char         *argv[];
    char         *envp[];
    IncParams    *myparams;
{
    ...
}
```

Figure 3.1: Main Program Declaration for an RDC Module.

model is exclusively through C function calls. Each incarnation runs in a separate, protected, virtually-mapped address space comprising a text segment and growable heap and stack segments. The kernel does not provide virtual memory or swapping to disc, however, and the programmer has to be aware of memory limitations. Since incarnations are heavyweight processes both in terms of set-up costs and execution management costs (context-switching between mapped address spaces), their creation is only justified if they are long-lived.

It is not essential to the RDC model that incarnations are single-threaded. There was insufficient time to develop either kernel facilities for multi-threaded incarnations, or a user-level threads package. There are sufficient facilities to monitor and manage asynchronous communication completion for the construction of a user-level package to be possible. These will be described in Chapter 4.

To minimise the set of changes necessary to existing C code, the main program declaration of any RDC module is of the form shown in Figure 3.1. All incarnations belonging to an RDC are able to access arguments and shell environment bindings supplied to the RDC at the time that it is run, through the normal arguments *argc*, *argv*, and *envp*. The additional argument *myparams* is for the sole use of the RDC programming scheme: it points to a data structure which comprises a block of incarnation-specific data arguments and to a list of names of communications interfaces through which it is connected to other incarnations. The declaration of and use of incarnation parameters are described in Chapter 5.

Executable Files and Module Names

A module is compiled into a file containing executable code, initialised data and kernel-dependent execution information. A module can be incorporated into any number of RDC binaries without modification. RDC binaries are single files comprising a concatenation of module files prepended by header information. In the code the modules are referred to by their index in this concatenation, the ordering of which is given at the time of the program's creation. The index can be derived from a character string module name which the RDC binary's creator can also supply if necessary.

3.2 Incarnations

3.2.1 Creating an Incarnation

To create an incarnation, a node must be selected, the incarnation to be created must be defined, and its initial parameters must be defined. Node selection will be covered in Section 3.4.

There are three ways to create a new incarnation: 1) by creating it from one of the RDC's modules (held on disc), 2) by creating it from an image of a module held in volatile memory, and 3) by forking.

1] *inc_create* with a module identifier as argument. Creating an incarnation from a module allows the caller to continue intact, but creates an incarnation which will begin execution from the start of the module's main program. Its initial state is specified to be either STARTED or INCHOATE. A STARTED incarnation proceeds to execute its main program when it has been created, an INCHOATE incarnation's execution is suspended at the point where all components have been assembled in main memory except its incarnation parameters (the IncParams structure which is passed to the main program).

2] *inc_create* with an INCHOATE incarnation's identifier as argument. INCHOATE incarnations can be substituted for modules in a next step of creating another incarnation of the same module. This makes creation faster since INCHOATE incarnations reside in volatile memory, not on disc. Although it may seem a low-level concern, this removes an unsatisfactory performance obstacle which was experienced when configuring an RDC. The host workstation became a bottleneck for fetching module text and initialised data. An INCHOATE incarnation is not wasted: a call exists to change its state to a STARTED incarnation in its own right. *inc_create* unblocks the caller as soon as the new incarnation's creation has been initiated, and before executable code and initialised data have been fetched. Therefore large numbers of incarnations can be created in parallel, by employing multiple INCHOATE incarnations created initially, from which others can be created independently. The use of INCHOATE incarnations is intended as being hidden from the applications programmer behind library calls.

3] *inc_fork*. Forking is similar to the UNIX facility of that name: the caller continues to execute, having created an incarnation with a copy of its execution state, but with a return value indicating which is the new incarnation and which the original caller. Forking allows new incarnations to be created whose state depends on processing so far. For example, a single server incarnation RDC with connected clients can be forked to another node to produce a copy with all the relevant data necessary to process requests from any client. The question of by what mechanisms clients can be divided between the original incarnation and the new one remains to be addressed in Chapter 6. Given the absence of kernel support for copy-on-write virtual memory segments, forking has to block the caller until the new incarnation has been given a copy of the caller's heap and stack segments (the text segment is shared).

In the original Wormos design, a user-level mechanism was provided for an incarnation to copy another one that is already in the STARTED state. This was intended for externally checkpointing incarnations, so that their copies could be used in case of the original's failure. However, it was never used. This was partly because such a mechanism could only ever be used in the context of i) user-level synchronisation, so that the checkpointed state would be determinate, and ii) user-level code to arrange for the two forked copies to identify themselves. When all that has been achieved, the copied incarnation could as well have forked itself anyway under the same synchronisation conditions. Only self-forking semantics were carried over to Equus.

3.2.2 References

Incarnations possess a kernel-managed vector of data structures called references, which are not readable or writable by user code. References are so-called because each corresponds to an identifier used in the incarnation's module code. Their data is used by the kernel to enforce protection against illegal accesses, and to bind the identifiers to the appropriate physical addresses.

For example, one type of reference is an incarnation handle, which is created whenever one incarnation creates another. The new incarnation's unique identifier is returned to the creator. Like all kernel-generated identifiers passed to user level, this identifier is an integer. When the incarnation uses the identifier of an incarnation or other kernel-created entity, a reference is looked up which corresponds with this identifier. Assuming it is found, the reference's data are used by the kernel to locate the corresponding entity, and also to enforce any restrictions on the types of operation the possessor can perform upon the entity. It is not enough to know an entity's identifier: if no corresponding reference is found, attempted operations are prevented.

There are six types of reference:

1] **incarnation handles** enable control of an incarnation;

2] **streams** are interfaces used for message passing;

3] **ports** are interfaces used for message passing;

4] **stream handles** are used to reconfigure communications connections;

5] **port handles** are used to reconfigure communications connections;

6] **buffer handles** refer to areas of a remote incarnation's address space.

These are fully described as they arise in Chapters 4 and 7. With the exception of the last, references refer to entities which are part of an RDC's configuration. Incarnations come to possess references in two ways: either they are created when the kernel-managed entity to which they refer is created, or when they are copied and transmitted – propagated – from one incarnation to another, using mechanisms to be described in sections 4.6 and 5.5.

The use of references is taken from protected link tables in DEMOS [DEMOS87] and other link-based distributed operating systems. The alternative paradigm to this is the use of user-level capabilities, as exemplified in the Amoeba distributed operating system design [AMOEBA86]. Possession of an Amoeba capability enables protected access to objects managed by kernel- or user-level processes. User-level capabilities are cheaper to create and propagate than references, and can be used to provide protection against illegal accesses to objects through encryption techniques, even when the kernels themselves cannot be trusted. They are therefore suited to control access to files and other protected objects employed in large numbers in insecure working environments. Security was not of great concern for the development environment in which the present work took place. Even if it were, security against kernel-tampering is relatively straightforward to provide for physically discrete processor pools, in that they can be locked away from users if necessary and the individual nodes booted only by system administrators with knowledge of the requisite password. It is therefore reasonable to assume that kernels running at pool nodes are trustworthy.

User-level capabilities cannot contain up-to-date physical address information if the objects they refer to can migrate independently. Therefore whenever a propagated capability is used first at a node, the referent object has to be located; and whenever it is used subsequently, the address of its referent has to be looked up in a cache. With references:

1] the latest address information can be transmitted when a reference is propagated, and, for frequently used streams and ports in particular, local user identifiers can be used which contain indices into the vector of references, for fast lookups of the reference data;

2] addressing information pertaining to an individual reference can be altered transparently under kernel control;

3] information about relationships between incarnations is available to the kernel;

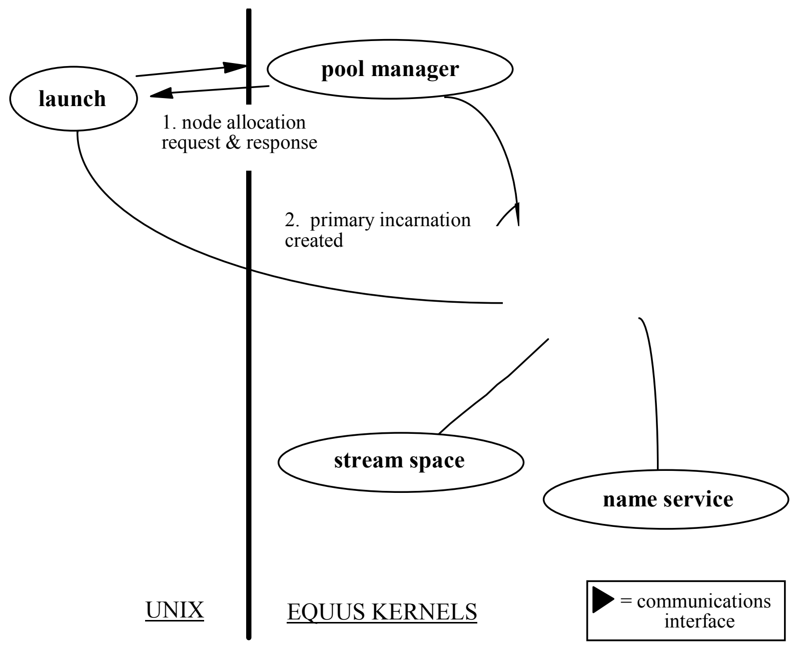
4] it is possible to constrain propagation to other incarnations to protect against unwanted operations.

3.2.3 Events

One of the operations available when an incarnation possesses an incarnation handle is to register an interest in certain classes of events that can occur to the referent child. By extension, these events can occur to the caller itself. Events include the completion of communications actions, and notification of changes of state of incarnations. These are described as they arise in Chapters 4 and 7. Incarnations are able to a) poll for event information; b) block until one or more of a set of events occurs; and c) experience a software interrupt when a specified type of event occurs, which can be handled by a user-supplied function.

3.3 Launching an RDC

The process of starting up an RDC on the processor pool is known as launching. Users run RDCs from the host workstation with a program called *launch* which initialises the RDC, acts as a server process for UNIX file and window services during its run time, and is used to terminate it.



launch
pool manager
1. node allocation request & response
2. primary incarnation created
stream space
name service
UNIX
EQUUS KERNELS
▶ = communications interface

Figure 3.2: Launching an RDC.

At launch time the following are carried out (Figure 3.2):

1] The user running launch, as identified by his or her UNIX user name, is authenticated, and, if he or she is deemed eligible to run an RDC and the request for nodes as given in launch options can be satisfied, the RDC is assigned an identifier. The identifier is a bit-string generated so as not to be repeated at a subsequent launch over a reasonably long period. The RDC is allocated an initial set of nodes to operate upon. The allocation is normally made by the pool manager, to which the launch program makes a request. Certain users are able to launch system RDCs, such as the pool manager itself. The launch program directly allocates all nodes to these.

2] The RDC's first incarnation, the primary incarnation, is then created by the launch program. This is of the module which was declared as the 0'th module when the RDC binary was created. If possible, the initial location of the primary incarnation is at the node given as a hint by the user as a launch option. But it can be located elsewhere if this node could not be allocated.

3] This incarnation (and all subsequent incarnations created as part of the RDC) is provided with communications interfaces which library calls employ to obtain file services, X11-based window services, and other, Equus-specific services:

- a name service. Through this service a server RDC is able to register its services under a well-known text name. Other RDCs interrogating the name service with this name are returned a communications interface which they use to communicate with the server RDC. This is discussed further in Section 5.4.

- the pool manager. After launch time this continues to be responsible for allocating nodes to RDCs. It also provides automatic load balancing for them. This is dicussed further in Section 3.4.

- a service implementing a per-RDC construct to be described in Chapter 5, called stream space.

Equus was not designed to provide a full emulation of any version of UNIX. Only a small subset of UNIX system calls are available, and they are not accurate emulations due to difficulties in matching certain failure semantics. Given this, it would be more satisfactory to the programmer for it to be possible to compile some modules to run as incarnations under UNIX, and to allow for any UNIX system calls to be made within such modules. Use of an incarnation running under UNIX to perform UNIX system calls would require the programmer to use RDC communications primitives as a common interface between UNIX and Equus incarnations, through which data is passed to and from the latter. This would be justified for sophisticated use of UNIX (e.g. full access to X11 facilities). On the other hand, it is convenient for UNIX file operations, in particular, to be available directly to Equus incarnations, despite the emulation inaccuracies. This would be retained.

3.4 Node Allocation

```
#
# Pool nodes under machine "kathleen".
#
201:kathleen1:systems,all:jt68030:4mb
202:kathleen2:systems,all:jt68030:4mb
203:kathleen3:balanced,all:jt68030:4mb
204:kathleen4:balanced,all:jt68030:4mb
205:kathleen5:balanced,all:jt68030:4mb
206:kathleen6:timeshared,all:jt68030:4mb
207:kathleen7:timeshared,all:jt68030:4mb
208:kathleen8:exclusive,all:jt68030:4mb
209:kathleen9:exclusive,all:jt68030:4mb
20a:kathleen10:exclusive,all:jt68030:4mb
20b:kathleen11:exclusive,all:jt68030:4mb
```

Figure 3.3: Contents of Node Descriptor File.

When a node is allocated to an RDC, any incarnation belonging to the RDC can create incarnations which are hosted there. An incarnation belongs solely to the same RDC as the incarnation which created it – that is, it bears the same RDC identifier (initially borne by the primary incarnation). It also bears the same UNIX user and group identifiers as its creator, so that file accesses can be constrained according to standard UNIX conventions. The kernel, pool manager and launch program are together responsible for security against fraudulent use of these identifiers, and they are kept in sealed kernel tables which can be read by user code but not altered.

Integer names were chosen for the nodes of the development computer systems which reflected the underlying topologies. These names are interpreted directly by the kernel to perform efficient routing. A node descriptor text file with a well-known pathname is created by the system's administrator to describe nodes' attributes. For the multi-68030 computer system, one of the form shown in Figure 3.3 was used.

This file defines for each node: i) its kernel-recognised name; ii) a humanly-recognisable string name; iii) a list of names of groups to which the node belongs; iv) the node type (jt68030); and v) a list of hardware characteristics of the node (they all have 4 megabytes of main memory).

```
#
# User groups for nodes under machine kathleen.
#
tim:systems,timeshared
andrew:systems,timeshared
alivahit:exclusive,timeshared
esin:exclusive,timeshared
equus:all
```

Figure 3.4: Contents of Pool Users File.

There is also a pool users text file to describe users who are able to launch RDCs. One of the form shown in Figure 3.4 was used.

This is a list of UNIX user names associated with a list of node groups from which nodes can be allocated to RDCs launched by the user. The two files taken together show groupings in which the nodes are divided into: those for use by the systems programmers for testing development versions of the kernel ("systems"); those used by the pool manager to run RDCs' incarnations with load balancing ("balanced"); those timeshared between incarnations without load balancing being attempted ("timeshared"); and those to be allocated to RDCs exclusively so that they can perform their own mapping without interference ("exclusive").

Experience with node allocation policies and load balancing has been limited by lack of time. Although each type of allocation represented by the node groups in the above example has been found to be desirable for different types of user requirement, their simultaneous use has not been attempted, and only limited experimentation with sub-combinations has been possible. Most commonly, the "systems"/"exclusive" division was used, but with different numbers of nodes required in each grouping at different development stages; developers of autonomously mapped RDCs were therefore constrained to write ones which could adapt to whatever nodes they were allocated at run-time. Limited node numbers have meant no more than two "exclusive" allocation groups could be used without making the groups too small for most experiments.

There is much room for improvement in the understanding which has been gathered of pool allocation policies based around the types of node usage represented by the above groupings. However, the following facilities were prototyped in collaboration with the author's colleague, Mr A. Sherman [SHERMAN]:

1] At launch time, it is declared through a launch program option whether the mapping of the RDC's incarnations is to be performed by the system on the "balanced" nodes, or whether the RDC is to perform its own mapping. A generalisation of this option which was felt by users to be desirable was for a list of node groups from which the user wishes nodes to be allocated to be declarable as a launch option. Then the user could try launching onto exclusively-allocated nodes, and if this failed launch onto timeshared or load-balanced nodes by changing the launch program options.

2] Incarnations can find out the RDC's current node allocation – the number of nodes, their kernel identifiers and current state of loading. The number of nodes allocated can be of interest even when an RDC runs under the load balancing service: ten computationally intensive and functionally identical incarnations running on two nodes may be merely wasting the kernel's time in context switching, rather than giving a performance gain over two such incarnations. The nodes' long-term characteristics (for example, processor type and amount of main memory) can be looked up from their identifiers in the node description file. The load information is measured as a time average of the number of processes which are running or eligible to run. It is provided so that RDCs can measure the effects of their load balancing strategies. On the multi-68030 machine, load information is obtained cheaply by accessing global variables stored in the nodes' memories, directly over the VME bus – that is, by breaking the design assumption of distributed memory nodes. The cheapness of this operation means that any incarnation is allowed to look up this information. For the first development computer system, which was a truly distributed memory system, it would create undesirable message traffic if any incarnation could gather current global load information by interrogating the kernels. Therefore the kernels periodically exchanged loading information, and this information was read locally when incarnations requested it. The question of incorporating communications load into the kernel load measurements has not been studied.

3] Nodes which were exclusively allocated are automatically released back to the pool when the RDC terminates.

In addition, the applications work of the author and another colleague, Mr A. Sahiner [SAHINER], employed the following interface to pool management facilities, for the design of server RDCs which adapt their node usage according to varying client demands. The RDCs run on their initially-allocated nodes, until the demand for their services increases beyond a threshold value. They then utilise additionally-allocated nodes, which they release as demand declines again. These management facilities were simulated through having the "clients" and the "pool manager" incorporated as incarnations belonging to the server RDCs themselves. The interface consists of:

4] A call to request extra nodes – ones available immediately if possible, and optionally ones which become available later. The request may or may not be satisfied.

5] The reception of a notification informing the caller of either a) a new node allocation, which the recipient has to confirm it still requires, or b) an impending node withdrawal, which will occur in a fixed time. These notifications only arise when a call of type 4] has been made. There is possibly a restriction on the allocation (the node could be allocated from the "timeshared" group, or from an "exclusive" group, but with a time limit).

6] Incarnations can voluntarily release a node back to the pool.

Mapping

The node at which a new incarnation is to be created is specified explicitly by its integer identifier, passed as an argument to the creation call. For RDCs which have been launched such that the incarnations are mapped automatically by the pool manager, the node identifiers are discarded by these calls, and a node chosen by the pool manager is used instead. This is performed transparently to the caller, to avoid alterations to code being necessary in order for an RDC written to perform its own mapping to be launched onto load-balanced nodes.

A deficiency of this implemented scheme is that it rests on the assumption that nodes are homogeneous. If they were not, then the node identifier passed through to the creation call would not suffice for the pool manager to determine what categories of node to choose from for the initial and subsequent mappings of the incarnation. This could be improved by requiring a data structure to be supplied as an argument to each creation call which contains a node identifier and/or flags which specify the required characteristics of the node to be used.

3.5 Summary

RDC binaries are constructed from separately compiled components called modules. These are the blueprints for incarnations. Incarnations are single-threaded and run in their own address space. They each possess a vector of references which they use to perform operations upon local and remote entities using location-independent names, and which constrain their rights to perform these operations. Incarnations are able to register interest in and monitor the occurrence of events concerning other incarnations and communications events.

At launch time, the RDC is allocated a set of nodes to operate upon, and the primary incarnation of the RDC is created by the operating environment. This incarnation and its children are able to create new incarnations from modules, from INCHOATE incarnations and by forking. The user launching an RDC can declare whether the RDC is to map its own incarnations, or whether they are to be mapped automatically by the pool manager's load balancing service. The node allocation schemes used with the development computer systems have been outlined.

Chapter 4
Communications

This chapter covers the communications facilities available to the programmer in the RDC model. It describes the primitives that incarnations use to communicate, and the structural components via which communication takes place.

4.1 Message Passing Requirements

The message passing primitives to be described are designed to be used both directly as they are and as part of the run-time support for RPC and object operations:

1] an incarnation sends a "request": a message, a remote procedure call or an operation on an object;

2] the request is queued;

3] the request is received by one or more incarnations which process messages/ implement remote procedures/ implement objects;

4] as a result, these may generate a reply message/ return output procedure parameters/ return output operation parameters.

The RDC model incorporates primitives for asymmetrical message exchanges of the form 1] - 4], called *invocations*. Under invocations, requests are streamed so that reconfigurations can be made which change the request stream's association with one or more incarnations that process the messages, implement the procedure calls or implement the object operations. A request stream is not to be confused with a byte stream: requests are discrete units of data, and can only be transmitted or received discretely. A request stream does not by itself constitute a complete communications interface. It is a structural component which serves to distinguish a series of requests, but for a remote procedure call system, for example, there are additional functional requirements of naming and binding, interface type matching and parameter marshalling and unmarshalling. All of these are orthogonal to the present configurational concerns and are left to other layers to manage.

The association of request streams with receiving incarnations is achieved by intermediate routing constructs called *channels*. Channel-based reconfigurations affect the destination of future request messages sent over given streams.

The other structural component supported directly by the kernel and of relevance to communications reconfigurations is the message queue. Firstly, message queues are necessary to allow for asynchronous interactions. Secondly, as long as a request is in a queue and has not been received by an incarnation, a decision can still be taken as to which incarnation consumes the request. This issue will be taken up in the discussion of peer server incarnations in Chapter 6.

4.2 Invocations

To match the interactions 1], 3] and 4] above, an invocation consists of: i) the sending of a message – the invocation message – by an incarnation using a type of reference called a *stream*; ii) the receipt of this message by one or more incarnations, each using a type of reference called a *port*; and, in some cases, iii) the sending back of a reply message using a handle generated automatically when the invocation message was received. By using the appropriate handle, reply messages are routed back to the call which transmitted the corresponding invocation message, whatever reconfigurations may have occured meanwhile.

A message consists of:

- A 32-byte untyped header, used either for containing the entire message data or for identifying accompanying extra data;
- An optional block of untyped extra data which can be as large as fits contiguously in the virtual address space of the sender;
- An optional reference descriptor, used for propagating – sending a copy of – a reference to another incarnation, as described in Section 4.6.

Although use of untyped messages can be error-prone, the basic message-passing primitives have been found to suffice for writing applications running on

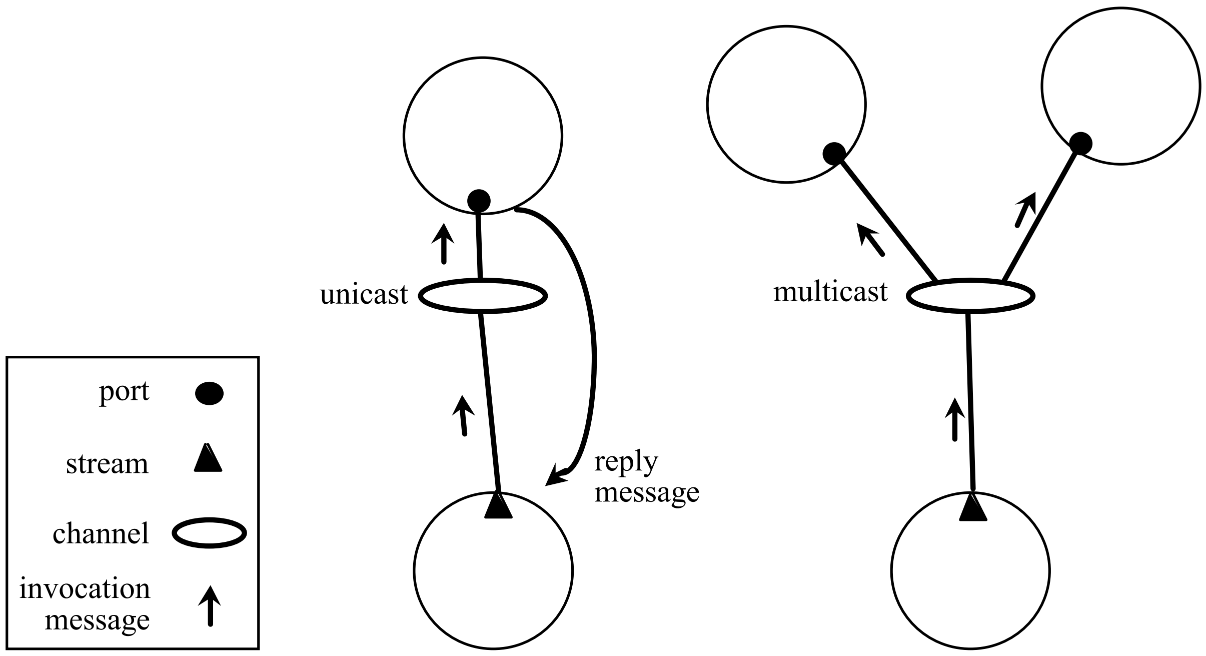


Figure 4.1: Invocations Take Place Using Streams and Ports Connected via Channels.

the homogeneous target processors. Facilities could be improved through library-supplied utility functions: Isis [ISIS], for example, provides the *msg_put* function to marshal a list of C data items and structures on to an array of bytes, and the *msg_get* function to unmarshal the data. Dynamic buffer management for marshalling is available through the *malloc* and *free* library calls.

4.3 Channels

Incarnations come to be connected via channels, which are the referents of streams and ports (Figure 4.1). A channel is an unbuffered routing medium, over which invocation messages are sent using streams attached to it. These messages are then received using ports attached to the same channel. Whereas streams are attached to at most one channel, a port can be attached to more than one so that the output of several channels can be merged at it. Furthermore, an incarnation can receive messages from more than one port. By connecting and disconnecting incarnations to and from channels by changing their stream and port attachments, different communications relationships are made possible, potentially transparently to the incarnations concerned.

A channel is either unicast – in which case only one port can be attached and so be a destination for invocation messages at any one time; or it is multicast – in which case the kernels attempt to deliver identical copies of each invocation message sent

primitive	synchronous?	message size	reply?	delivery	multicast?
inc_invoke	yes	unrestricted	optional	reliable	no
inc_asinvoke	no	unrestricted	no	reliable	no
inc_send	no	up to 1.5k	no	reliable[1]	possible
inc_dgram	no	up to 1.5k	no	unreliable	possible

Table 4.1: Invocation Primitives.

over it to all incarnations with attached ports. By dynamically acquiring and destroying ports, incarnations join and leave multicast groups, without the knowledge of any incarnation sending invocation messages over the channel. The same port can be simultaneously attached to both types of channel.

4.3.1 Unicast Message Primitives

The complete set of primitives for sending invocation messages is given in Table 4.1, and the other communications primitives are given in Table 4.2.

Only *inc_invoke* is synchronous, and it is the only primitive that can obtain a reply. No separate action is necessary to get the reply: if a reply is made, the reply message overwrites that used for transmission whilst the call is blocked. This is satisfactory for many cases, and using separate message buffers for transmission and reply would increase the cost of this call. The receiver sets a flag to choose whether a caller of *inc_invoke* is to be unblocked immediately upon receipt of the invocation message (therefore without receiving a reply), or unblocked later with a call to *inc_reply*. If no application-level reply data are required, it is cheaper for the kernel to provide an acknowledgement than for the receiver to make a redundant call to *inc_reply*, involving a user-kernel context switch. When the receiver blocks the sender, a reply handle is returned to it. With this, it can either reply after due processing of the invocation message (and in the meantime it can receive other invocation messages); or, to allow a programming scheme in which one of its peer incarnations replies instead (on the basis, for example, of its specialised functionality), it can forward the received message over another stream, using *inc_forward*. In that case, the next recipient receives a copy of the message which is identical except that its header can have been modified by the forwarder.

[1] Used over a multicast channel, *inc_send* guarantees delivery only to at least one attached port

primitive	function
inc_receive	receive an invocation message at a port
inc_reply	reply to an invocation message using reply handle
inc_forward	forward an invocation message over a stream
inc_copyto	copy data to a remote buffer using a buffer handle
inc_copyfrom	copy data from a remote buffer using a buffer handle

Table 4.2: Non-Invocation Communication Primitives.

Meanwhile, the original sender remains blocked until replied to, and the message forwarding is transparent to it.

inc_asinvoke is used for sending invocation messages asynchronously. Its name reflects the fact that it was at an early stage planned for it to be possible to obtain a reply message resulting from a call to it, as with *inc_invoke*. Kernel mechanisms to support such a feature would be redundant, however, since, as will be described, a stream can be propagated in the message sent using *inc_asinvoke*, which a receiver can then use to send a reply message back with an existing invocation primitive.

inc_send is used as an alternative to *inc_asinvoke* for sending small amounts of data (up to 1.5 kilobytes). Whereas extra message data sent using *inc_send* are copied immediately out of the sender's address space, extra data supplied to *inc_asinvoke* remain buffered in the caller's address space. This is discussed further in Section 4.4.2.

inc_dgram is the same as *inc_send* except that no underlying delivery acknowledgement is sought by the kernel, making message transmission times up to half those of *inc_send*. Although delivery is usually achieved with *inc_dgram*, unlike the other invocation primitives it cannot be relied upon.

4.3.2 Multicast

Only *inc_send* and *inc_dgram* can be used to send invocation messages over multicast channels. *inc_send* is deemed to be successful if the message was delivered to at least one port, so it can be used to find out whether at least one member of a multicast group of incarnations is reachable, but does not guarantee delivery to all ports attached to the channel. As in the unicast case, no delivery guarantees are made for *inc_dgram*.

This choice of low-level multicast facilities is to be compared with, for example, Isis's reliable broadcasts [ISIS] or LADY's facility to stipulate a number of implicit delivery acknowledgements which are to be received for a multicast to be deemed successful [LADY]. The choice made for the RDC model is a result of i) the decision to add multicast after the original unicast primitives were designed, and ii) designing for simplicity of the basic model and its implementation. The following factors had to be taken into account:

1] *inc_invoke* is only designed to get one reply. If multiple replies were to be received as a result of a multicast, either a separate multicast version of this invocation primitive would be required, or the other replies would have to be obtained with separate system calls – the approach of the V-system.

2] Transparent reliable multicast is only possible if the number of ports attached to the channel is known to the issuing kernel. The kernels would in that case have to maintain a consistent view of this, even though ports can be dynamically created and destroyed, and streams attached to the channel and requiring this knowledge can be propagated to arbitrary kernels. As things stand, no global knowledge has to be kept of ports attached to a given channel; the design relies upon an underlying physical broadcast facility by which a packet can be received at all nodes. Only those kernels managing ports attached to the designated channel accept the arriving packet.

3] The problem of reliability in the absence of global knowledge about ports extends to *inc_asinvoke*.

4] The utility of LADY's partial-success multicast semantics (defined by specification of the minimum number of acknowledgements required) was unclear in the absence of hard application requirements. It was decided to leave partial-success and reliable semantics to the application layers, rather than provide kernel support for them. To implement higher layers of multicast, *inc_dgram* can be used to multicast messages, which recipient incarnations acknowledge using streams connected to the multicaster. Bulk data can be transferred by enclosing in the small multicast message a buffer handle referring to the bulk data to be multicast (buffer handles are described in Section 4.5). Multicast atomicity and serialisability, which are addressed in the Isis work and the Amoeba work ([KAASHOEK]), are also left to application layers.

4.4 Ports

A port is a queue of invocation messages which have arrived at an incarnation over one or more channels. Ports can become unattached as a result of reconfigurations: an unattached port can have a non-empty queue of messages which arrived during (a) previous attachment(s), although no further messages will be appended to it in this state. Messages are queued in the order in which they arrive at the port. Two messages sent using different streams could appear in a different order to that in which they were sent, but the kernel guarantees that messages sent using the same stream are queued in sending order (unless *inc_dgram* was used).

Invocation messages are received using the same primitive, *inc_receive*, regardless of which invocation primitive was used to send them. *inc_receive* removes messages from the queue. If no suitable messages are present, the caller is blocked indefinitely, or for up to a specified timeout period. The kernel can distinguish between the streams used to send the messages queued at a port, and it is possible to receive only those messages sent from a particular source stream. This selectivity makes it possible to receive, say, a second message from a specific client (or rather, stream) without interleaving with the reception of messages from other clients at the same port.

An incarnation is able to possess multiple ports in order that:

1] Different ports can be associated with messages of different functionalities. An incarnation receiving messages of (application-specific) type *a* at port *A* can continue to do so until it is ready for a message of type *b* at port *B* without the programming problem of interleaving the processing of two kinds of message at a single port.

2] Clients can be grouped together. By associating a set of clients with one port, a server is able to handle their invocation messages equivalently with respect to the data they operate upon.

3] Any port can be set so that a software interrupt is generated upon the arrival of a message there; a handler function is specified, to be called asynchronously when this event occurs. This is useful for implementing management operations which execute independently of the incarnation's main thread of control (but which can be blocked during critical code sections, like hardware interrupts). An example of such a management operation is a monitoring operation, which returns the current state of the incarnation to the requestor.

Using the event mechanism, it is also possible to find out which of a set of ports have messages for receipt, to avoid blocking in awaiting a message from one port when another port has messages of interest.

Where considerations 1] - 3] above do not apply, the need to use the event mechanism is an argument against the use of multiple ports for multiple clients, as opposed to use of a single port with clients and functions distinguished by message data. It adds to the performance overheads of reception by necessitating regular event-related system calls, and adds to program complexity. A second argument in favour of using one port is that, if clients are anyway indistinguishable, a client can be connected or removed transparently.

4.4.1 Message Queues

By queuing invocation messages in ports local to the incarnations which receive them:

1] The set of messages awaiting reception at an incarnation is known locally. Examining the number of these messages and their application headers is then potentially a cheap means of being able to estimate the amount of processing these messages represent when considering whether a reconfiguration is desirable on the basis of workload.

2] The current message queue can be divided and some of it moved to a different incarnation's port if required as part of a reconfiguration. This will be described in Chapter 6.

3] But, for a port local to another incarnation to become the new destination for messages, all connected streams have to undergo address rebindings.

The alternatives considered were:

a] to enqueue invocation messages locally to their issuing kernel. This would leave unreceived messages untransmitted regardless of any reconfiguration to change the identity of the incarnation to receive them, and so amortise the costs of achieving this. But it would make the information of 1] expensive to obtain, since the queue could be distributed at more than one kernel. And it would complicate the logic necessary for reception of invocation messages sent using multiple streams.

b] to queue invocation messages at a fixed location for each channel. The information of 1] would again be available at a single place. And this arrangement has the advantage over the implemented scheme of not requiring new address bindings to be made if a new incarnation takes over receiving the messages. But it can be expected to add significantly to the cost of receiving each message (as would alternative a): it could as much as double the end-to-end transmission times of individual messages, by making the reception of a message potentially as expensive as the operation of delivering the message.

On balance, it seemed preferable to keep message transmission times at a minimum during normal, unreconfigured operation, and to concentrate on efficient mechanisms to rebind streams and move message queues.

The concept of a shared queue with multiple receivers arises naturally out of the idea of a queue with a fixed location. The LOCUS distributed operating system [LOCUS] supports pipes with multiple readers. The pipes are implemented at a storage node that remains fixed as the set of nodes hosting readers and writers varies. Although kernel support for this construct could make its implementation more efficient than is achievable through use of an incarnation to manage a shared queue at the application level, the application level approach is preferable. For a queue is only one of the different abstractions of shared message collections which can be required at the application level. These include, for example, sets of messages distinguished according to task types, and multiple queues of prioritised messages. It is the job of an application-dependent scheduler to provide such facilities, and not that of the kernel.

4.4.2 Asynchronicity and Buffering

After the experience of the first implementation, it was decided that the kernel would not buffer messages bigger than could fit into a single underlying communications packet – the largest block of data which can be transferred contiguously over the underlying physical network. The size restriction on messages sent using *inc_send* is chosen to match the packet size. Each kernel message buffer either contains all of a message's data, or is a header containing a packet's worth of the message data, and address information to enable the rest of the data to be fetched when an incarnation performs a call to *inc_receive*, and thereby provides a user-level buffer for the data. The rationale for this consists of two points:

1] By imposing a limit on the amount of each message's extra data which can be held queued awaiting reception in a kernel message buffer, and by furthermore imposing a limit on the total amount of this buffer capacity which the kernel is prepared to devote to a single incarnation, the overall buffer capacity required by the kernel is limited and buffer management is reduced to handling buffers of a small number of fixed sizes (currently two).

2] If a queued message is transmitted to another port (queue) as part of a reconfiguration, only the first packet of the message data is transmitted twice. The rest remains in the caller's address spaces until the message is eventually received.

It is the responsibility of the programmer not to re-use a message buffer until the kernel has finished reading it. The event mechanism can be used to find out when a particular call to *inc_asinvoke* has completed. As an alternative to using the event mechanism, the programmer is able to set the value of the so-called buffer limit of each individual stream, in order to control the use of buffers automatically. If the number of calls to *inc_asinvoke* from and including the first one which remains uncompleted equals the value of this limit at the time another call to *inc_asinvoke* is made, the call will be blocked automatically until the first such call has completed. By maintaining a circular list of buffers which are in number one more than the value of the buffer limit, and using them in strict rotation, no buffer can be overwritten prematurely, and no event management code is necessary. These mechanisms are intended for use in the implementation of library calls. The constraint on asynchronicity which this scheme makes is not an unusual shortcoming of this design: any run-time support system for asynchronous message passing has a buffering limit which when reached will cause the caller to be blocked.

A shortcoming of this implementation of asynchronous message passing is that it limits concurrency. The transfer of bulk message data sent using *inc_asinvoke* could in principle overlap with the execution of the incarnation which later receives it. Instead, it is forestalled until the receiver is ready. Charlotte [CHARLOTTE87] provides an asynchronous receive primitive, which provides a user buffer into which the kernel can receive bulk message data whilst the caller executes concurrently. It has not been investigated how such a primitive could be incorporated into the RDC model, nor what performance advantages can in fact be achieved when message reception and incarnation execution, although concurrent, take place via the time-sharing of a single CPU. Asynchronous reception appears to violate the separation between message queueing and message reception which is presupposed when considering reconfigurations which re-queue unreceived invocation messages at other ports. In fact, however, it would be possible for the kernel to "unreceive" messages by resetting its pointer to those user buffers which had been overwritten. Buffers supplied in a call to the asynchronous reception primitive would have to be assumed by the programmer to be filled with garbage thereby, until notification by the kernel that they contained message data.

Failure

All the invocation primitives except *inc_dgram* attempt at the time of the call – whether they are asynchronous or not – to deliver the message or part of the message to at least one destination port, and await a reply message or a kernel-generated acknowledgement. If neither is received despite a system-defined number of attempts, the call fails with an appropriate return code. If an invocation message is sent using *inc_asinvoke*, however, it is possible for the node to which it is delivered to fail after it has been queued but before it has been received. In that case, it is the application writer's responsibility to decide that a problem has occurred: the kernel will regard the call as one that is never completed, and provide no information about the failure.

Events and Threads

The event mechanism could be used in the construction of a user-level threads package, using which threads can be assigned ports from which they receive and process requests asynchronously with respect to one another. Using the event mechanism, the run-time support system for such a package could tell whether a communications call issued by a thread would block the incarnation. If so, it could deschedule the current thread and schedule another thread which can make a non-blocking communications call.

4.5 Buffer Handles

A buffer handle is a reference which enables the possessor to read from or write data directly into a data area (buffer) in the address space of another incarnation. The data area is specified as a vector of bytes which is readable and optionally writable, to which a handle is propagated to another incarnation. The possessor of the handle then uses the primitives *inc_copyto* and *inc_copyfrom*, which are based upon the *copyto* and *copyfrom* primitives of the V system [V84]. They each take as arguments the offsets from the beginning of the remote buffer from which data are to be written or read respectively, the start address of a corresponding local array and the number of bytes to be copied. Unlike the V-system, their use implies no synchronisation with the remote incarnation. If synchronisation is required then it has to be achieved through message passing. The rationale for buffer handles is as follows:

1] If transmitted message data is not to be overwritten by a reply, the sender is able to include a buffer handle with the invocation message, so that the reply data can be written into a separate buffer in the caller's address space.

2] They allow for selective data transfer, as opposed to only being able to transfer entire or truncated arrays with ordinary message passing. Work on image processing applications has revealed a requirement to copy rectangular sub-sections of two-dimensional data in addition to contiguous linear sections: so that an image section could be obtained in a single primitive. This remains to be implemented, however.

3] They can be used to avoid transmitting data more than once. By sending a buffer handle instead of transmitting data in a message, a client enables a server incarnation to choose one of its peers or worker incarnations to process the client's data, without the data being transmitted twice. Instead, the server incarnation propagates the buffer handle, and its peer or worker fetches the data directly from the client's address space.

4] They can be used to increase concurrency: one or more incarnations can use buffer handles to read or write a buffer multiply before synchronising with the buffer's owner using invocations.

5] In particular, they can be used to multicast bulk data, by including a buffer handle in a multicast message, which the recipients use to copy the data concurrently and then issue acknowledgements to the sender.

4.6 Sending References in Messages

The semantics of sending a reference in a message is that a copy of the reference is sent – not the original. Propagation semantics was chosen over transmission semantics because i) it does not require a separate primitive to duplicate a reference before sending it in a message; ii) it extends naturally to multicast, whereas a single reference cannot be transmitted to multiple destinations.

A copy of a reference is one which has the same referent. It is a handle to the same incarnation, a stream or port attached to the same channel(s), a handle to the same remote buffer or a handle to the same port or stream. The semantics of propagating a port attached to a unicast channel requires further description, which will be given in Chapter 6. The reference descriptor component of a message contains a field for the local identifier of the reference to be propagated, and a flag which if set causes this reference to be destroyed upon propagation. Reference transmission can therefore be simulated. However, a propagated reference is not in general completely equivalent to that used for propagation. Firstly, a stream handle or port handle referring to the original does not refer to the new reference, even though the original may be destroyed and the new one is thenceforth functionally equivalent to the old. Secondly, flags in the reference descriptor can be used to modify the copy relative to the original:

1] The rights flags are currently used for altering read/write rights to read rights in the case of a buffer handle, for reducing the set of control rights in the case of an incarnation handle, and for determining whether the recipient of a port obtains the queue of messages which otherwise remains at the original port. The last two cases are taken up in context in Chapters 6 and 7.

2] A flag can be set to remove propagation rights from the recipient in relation to the received reference. This is for use where a server's designer wishes to ensure that invocation messages received over a certain port can issue from only one stream. This might be either because its message processing algorithm assumes an ordering which could be violated by multiple clients, or because a change in the set of its clients would affect the ability to reconfigure the server.

Propagation through message passing is the only kernel-supported means of propagating references. It will be shown in the next chapter how this basic mechanism is used to underpin higher-level mechanisms for the establishment of an RDC's configuration and for the propagation of references between its incarnations.

4.7 Summary

Incarnations can communicate using messages exchanged during synchronous or asynchronous interactions called invocations. They also can pass data asynchronously using buffer handles.

Invocation messages are transmitted via references called streams over routing components called channels, which can be unicast or multicast. Messages are queued for reception at references called ports. The use of message queues is an important aspect of the model in relation to reconfigurations, which will be taken up in chapter 6.

It has been described how incarnations can be dynamically created and mapped, by forking and by use of modules and INCHOATE incarnations. It is now possible to go on to show how they are created in a state of connection to other incarnations, as part of an initial RDC configuration.

Chapter 5
Configuration & Propagation

This chapter describes the mechanisms by which the initial configuration of an RDC is established. The first problem set down in Section 2.4.2 was how to achieve this for the text processing RDC, and that example is taken up again here. The chapter describes stream space in sections 5.4 and 5.5. This is a construct used in the run-time support for configuration establishment. Stream space has a wider role, for propagating references, and this is described.

5.1 Experience with the First Design

In the design of the original operating environment, Wormos, no special provision was made for the establishment of an RDC's initial configuration. An RDC had to be set up at run time using the following facilities, by the primary incarnation (and other incarnations in the creation tree):

1] There are primitives to:

- create a new channel with an attached stream and port;
- create a port attached to an existing multicast channel, referred to by a stream.

2] Every new incarnation is created with a so-called standard port, analogous to the standard input of a UNIX program. The creator obtains automatically a stream connected to this port. Any structure can be established by using this automatic connection to propagate streams attached to new channels created by the new incarnations and their creator, with the primitives of 1].

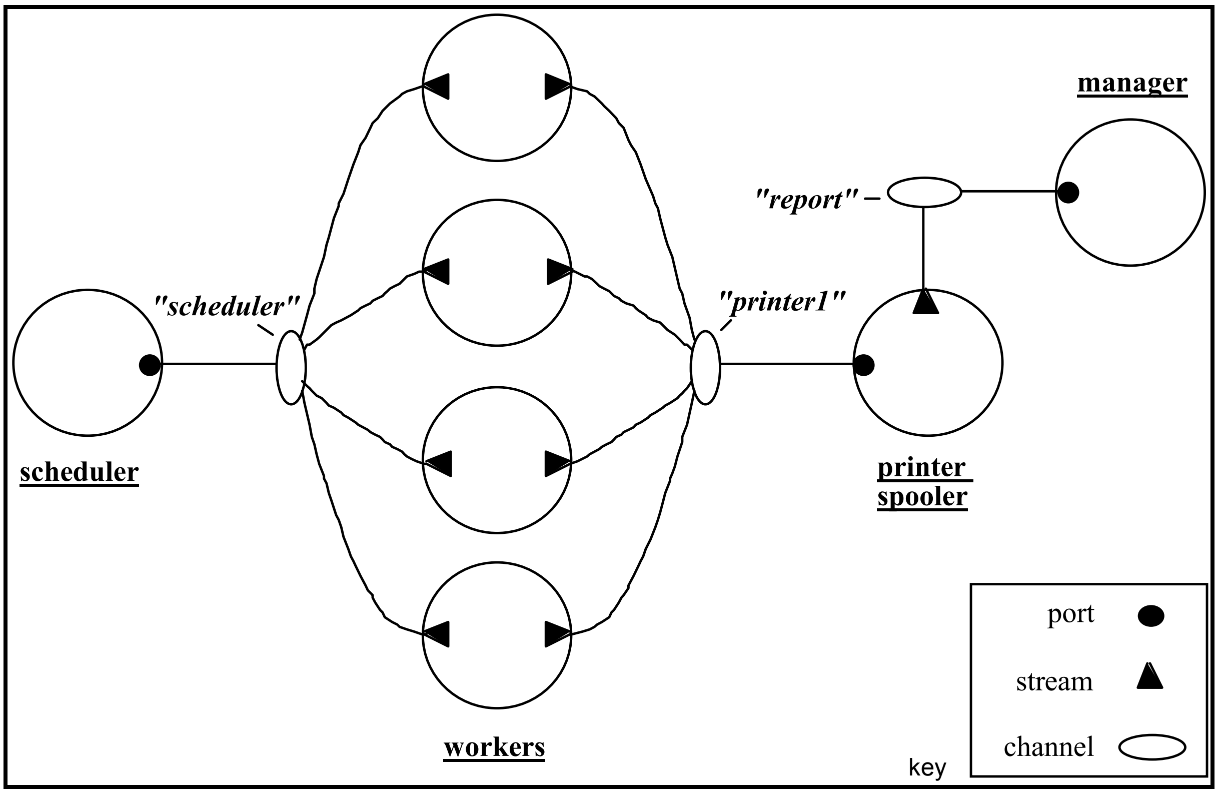


Figure 5.1: Initial Structure of Text Processing RDC.

This was a tedious and error-prone programming activity. Library calls were written to lessen the burden of this, but these still had to be explicitly made as part of the preamble to application code in every non-primary module.

5.2 Establishing a Configuration

The improvement achieved in the most recent operating environment, Equus, is for the programmer to be able to declare the connectivity between incarnations, and not be concerned with how connections are realised. No kernel changes were needed to achieve this. The facilities described in 1] and 2] above are still provided by the Equus kernel. But their use is hidden in a user-level run-time support system which makes declarative connection possible. This is now described through the text processing example.

5.2.1 Setting Up the Text Processing RDC

Figure 5.1 shows the complete structure for the text processing RDC introduced in Section 2.4.2. The manager is assumed to be the primary incarnation. This

```
typedef struct {
    int            ip_datasize;  /* size of data arguments in bytes*/
    char          *ip_dataargs;  /* pointer to data arguments       */
    int            ip_nifaces;   /* number of interfaces            */
    IncInterface  *ip_ifaces;    /* interface declaration structures*/
} IncParams;
```

Figure 5.2: Data Structure Used to Declare Arguments and Interfaces (streams and ports) for New Incarnation.

incarnation establishes the rest of the RDC's configuration, and initialises the RDC before entering into a monitoring mode. The details of its decision about the number of workers to create and about the RDC's mapping are not of concern here.

In order to be able to declare the configuration of any RDC, every channel employed in the structure is assigned a character string label of the programmer's own choosing. The means by which channel labels are assigned at run time will shortly be described.

In the example, each worker has a stream attached to the channel labelled "scheduler". Task acquisition by the workers can be implemented using *inc_invoke* over this stream. The scheduler possesses a corresponding port attached to the same channel, which it uses to receive these requests. Each worker also possesses a stream attached to the channel labelled "printer1", using which it sends processed document data asynchronously with *inc_asinvoke*. The printer spooler uses a port attached to this channel to receive this data. The printer spooler is programmed to send information periodically using a stream attached to the channel labelled "report". The information is received by the manager.

The call *inc_create* (used to create a new incarnation: Section 3.2.1) requires an argument to declare the new incarnation's arguments and interfaces. This argument is given as a pointer to an instance of the C structure shown in Figure 5.2, IncParams.

Before calling *inc_create*, an incarnation ("the configurer") makes declarations concerning the interfaces of the new incarnation. It does this using calls to library functions which set fields of the interface structures in the array pointed to by the field *ip_ifaces* (Figure 5.2). The data arguments and interface structures are copied automatically to the new incarnation (a pointer to another IncParams data structure appears as a main program argument). Each IncInterface structure in the new incarnation, however, has a field containing the identifier of a stream or port that the run-time system has generated. The stream and port identifiers can be extracted by the new incarnation on the basis of the corresponding indices in the interface array (which are known by compile-time convention or through data in the argument structure).

The configurer declares interfaces as being attached to labelled channels. By specifying the labels in a consistent way, the desired connectivity is effectively declared between the port of one incarnation and the stream of another. Connectivity will be achieved via the common channel.

The following are example calls of the interface declaration functions. *interfaces* is an array of IncInterface data structures pointed to by a data structure of type IncParams, and *n* is the index of the interface in this array.

- **ss_isStream**(&*interfaces[n]*, *label1*, *handle_label1*)
  Interface is a stream attached to channel labelled *label1*; a stream handle referring to the stream is to be generated, and labelled *handle_label1*, if this label is non-null.

- **ss_isPort**(&*interfaces[n]*, *CHAN_NEW*, *label2*, *handle_label2)*
  Interface is a port attached to a unicast channel to be created as a result of this call (*CHAN_NEW*) and labelled *label2*; optionally, a port handle referring to the port is to be generated, and labelled *handle_label2*.

- **ss_isPort**(&*interfaces[n]*, *CHAN_OTHER*, *label3*, *handle_label3*)
  Interface is a port attached to a multicast channel declared to be created elsewhere (*CHAN_OTHER*) and labelled *label3*; a port handle is optionally generated, and labelled *handle_label3*.

In addition, there is a function which is necessary for a configurer to create a labelled channel to which it itself owns an attached port:

- **ss_createChan**(*label4*, *chantype*, &*port*, &*stream*)
  The channel is declared to be labelled *label4*, to be unicast or multicast according to the value of the argument *chantype*, and the identifiers of an attached port and stream are returned in *port* and *stream*.

We now return to the example. To set up the rest of the text processing RDC, the manager incarnation makes *inc_create* calls to:

1] create an incarnation from module **scheduler**. It is declared to have a single communications interface: a port attached to channel *scheduler*. The channel is to be created as a result of this declaration.

2] create an incarnation from module **worker**. It is declared to have two interfaces: a stream attached to channel *scheduler*, and a stream attached to channel *printer1*. Repeat for the other workers.

3] create an incarnation from module **printer_spooler**. It is declared to have two interfaces: a stream attached to channel *report*, and a port attached to channel *printer1*. The latter channel is to be created.

Mapping is realised by supplying the appropriate node identifier to each *inc_create* call.

The manager must also declare the remaining channel to be created. It calls *ss_createChan* to:

4] create a unicast channel, named *report*. The call creates a local port attached to the channel.

This completes the description of the actions which the manager incarnation has to make in order to declare the structure shown in Figure 5.1.

5.2.2 Discussion

The general programming scheme just presented for establishing a configuration has the following features:

- The programmer is neither concerned with how stream-port connectivity is realised nor with the order in which channels and incarnations are created. The manager's actions 1] - 4] above could in fact have been made in any order. Moreover, for incarnations dynamically configured in this way, no module code is expended on the configuration.

- Channel labels, being character strings, are conveniently manipulable within the C language. The labels can be chosen for human readability to reflect both function (e.g. "scheduler.task_rqst", "scheduler.manage_rqst") and multiplicity ("worker0", "worker10").

- Any RDC structure can be created under this scheme. Character string labels can be generated for an arbitrary set of channels to connect any run-time variable set of incarnations with any combination of streams and ports.

- The programming scheme has been described in a context in which only one incarnation (the configurer, or the manager in the example) sets up the rest of the configuration. It is natural for the manager to execute step 4], since this results in the creation of a port which the manager requires. The steps 1] - 3], which declare the creation of the other incarnations, however, could in principle be followed by any other incarnation belonging to the RDC. For example, if 100 worker incarnations were to be created rather than 4, or any number sufficiently large for the cost of calling *inc_create* iteratively to be non-negligible, then the manager could fork itself so as to divide this task in two.

- By declaring that a unicast channel is to be created in the same call as the declaration of its attached port, inefficiencies are avoided in the operations of the run-time support system. For this convention enables the system to create the port at the incarnation which is to possess it, and to propagate streams with the correct physical address information corresponding to the port. If the creation declaration of a unicast channel was made in a separate call to the attached port declaration, the ultimate physical location of the port would not necessarily be known at the time attached stream interfaces were created. When these streams were first used, a kernel location mechanism would have to be invoked to determine the port's physical address. Any invocation messages already sent over the channel before the port was created would also have to be re-located.

5.2.3 Dynamically Adding A Worker

The manager can add a new worker dynamically to this RDC. The manager is assumed to have registered interest in extra node allocations from the pool manager. It created a channel and sent a stream attached to this channel as an argument to this registration call. Upon receiving a notification message at the corresponding port, informing it of an extra node allocation, it creates a new worker. To do this is no different to creating the original workers. The channel labels remain valid throughout the run-time of the RDC. Worker addition is transparent to the worker and its extant worker peers. The scheduler and printer spooler may or may not require notification of the addition by the manager, according to application requirements.

5.3 Forking

The forking facility, by its nature, involves design decisions about how to treat the references of the incarnation which performs the fork when they are copied to the new incarnation, rather than about how to declare new interfaces. The overiding concern is for it to be possible to integrate a forked incarnation into the existing RDC structure in such a way that maximum choice is left to the application writer, but for useful defaults to be provided.

When an incarnation forks itself, non-port references are automatically copied – with the same local identifiers and without modification of referent or rights – to the new incarnation, unless their propagation was disallowed, in which case they become invalid in the new incarnation. It seemed most useful for the new incarnation to possess, by default, streams attached to the same channels, and incarnation, buffer, stream and port handles referring to the same incarnations, buffers, streams and ports as in the original.

Ports attached to unicast channels cannot be attached to the same channel in the two incarnations, whereas corresponding ports attached to multicast channels can be. It was decided that, by default, ports in the new incarnation would be unattached. This means that the application writer can control the point at which the new incarnation becomes integrated into the rest of the structure, for messages cannot immediately arrive at its ports.

To overide the default, the forker can specify a list of its ports which are to be attached in the new incarnation. It may be necessary to use existing port identifiers in the forked child, to avoid inconsistencies with stored values, so these ports retain their identifiers. The ports are specified as either a) to be attached to a multicast channel, so that the incarnation can be automatically included in a multicast group; or b) to become attached to a new unicast channel, in which case a stream attached to the new channel is returned to the forker. The original can then propagate these streams to other incarnations, or employ them in reconfigurations to be described in the next chapter. Attachments not covered by this scheme can if necessary be programmed. As an example, Figure 5.3 shows code to establish two connected stream-port pairs which can be used to propagate references in either direction.

```
int                         forkstreams[1], forkports[1], fork_result;
int                         stream_to_parent, stream_to_child, my_port;
unsigned int   new_inc, new_node;

inc_chanCreate(CHAN_UNICAST, &my_port, &stream_to_parent);
forkports[0] = my_port;
fork_result = inc_fork(new_node, &new_inc, 1, forkports, forkstreams);
switch(fork_result)   {
 case 1: /* original                       */
          stream_to_child = forkstreams[0];
          inc_close(stream_to_parent);
          ...
 case 0: /* new incarnation  */
          /* stream_to_parent connected to original's my_port       */
          /* my_port is here attached to a new channel              */
          ...
 default: ...
}
/* Each now has a stream connected to the other's my_port       */
```

Figure 5.3: Code to Fork a Child.

5.4 Propagation

Every incarnation possesses by default a stream connection to a name service (Section 3.3). A library call to look up name entries takes a well-known character string service name and sends it using this stream to the name service. The service name is looked up. If it is registered, a stream connected to an incarnation belonging to the service RDC and supplied by it is associated with it. A copy of this is propagated to the caller. Conversely, RDCs wishing to supply a service are able to register themselves under an agreed name. This is the only way that an incarnation can come to possess a stream to communicate with an incarnation belonging to another RDC, apart from the few system-supplied streams it owns by default.

Once a stream to a service is obtained, it may, according to the service, either be used to make requests directly, or be used to request a connection to the service. The server incarnation initially contacted may want to a) validate the connection request, or b) associate the client subsequently with another incarnation belonging to the RDC, according to the type of connection request, for example, or the load conditions. A successful connection request can therefore be returned a new stream connected to a different server incarnation.

If a service request includes references – for example, a stream to return results asynchronously, a buffer handle to read or write request data – then these will have to be propagated by the client in messages. The details of this are liable to error and are cumbersome, and so are hidden behind a service stub procedure supplied as a library call by the service provider.

The details of the propagation of references in messages are, sometimes at least, hidden behind procedural interfaces to system services. Details of reference propagation are also hidden behind the mechanisms for configuration establishment described above in Section 5.2. The remaining case is that in which it is necessary to add a new connection between two incarnations belonging to the same RDC which were previously configured with no direct connection between them. The next section describes reference propagation facilities which improve upon the use of messages for this purpose.

5.5 Stream Space

When an incarnation propagates a reference to another in a message, a connection (or series of connections) is required for sending it, but the two incarnations might have no other use for this connection. Stream space was introduced into the model as a means of decoupling incarnations when references are propagated between them, to avoid this state of affairs. It is a repository, into which references can be placed, and out of which they can be fetched. To propagate a reference, one incarnation puts a copy of it into stream space, and the other takes a copy of it out. Since stream space can contain multiple items, a character string label is used to label the reference at the time it is deposited, and to identify it when a copy is requested.

Despite its name, any kind of reference can be deposited in stream space. A reference can either be propagated to indefinitely many incarnations in this way, or it can be specified at the time of deposition that only a certain number of copies of the reference can be taken out of stream space at any one time. The ability to restrict the number of copies is provided for programming schemes in which it is required that only, say, one incarnation is able to perform certain invocations until it yields the necessary stream for use by another, by placing it back in stream space. For example, it can be ensured in this way that at most one writer at a time is updating shared data managed by an incarnation, by using separate ports for write and read invocations, and limiting the number of extant streams connected to the write port to one. Qualifications applying to the special case of propagation of a port attached to a unicast channel are described in the next chapter.

Placing a reference in stream space is asynchronous: no results need be returned and the caller of the relevant primitive is not blocked. Two primitives are required: *ss_put* is used to deposit a reference initially in stream space, *ss_return* is used to return one of a limited number of copies after it has been obtained from stream space. The primitive to fetch a reference, *ss_get*, is synchronous. If no reference with a matching label exists, the caller is blocked indefinitely until one is deposited in stream space.

Apart from the fact that stream space can contain all reference types, and not just streams, it differs from a name service in two fundamental respects:

1] stream space exists privately and independently for each RDC, and does not persist beyond the RDC's execution. The contents of stream space are assumed to be needed only by the RDC's current incarnations and any they may create.

2] a name service request will return immediately with a failure code if supplied with what turns out to be a non-existent service name. In the case of stream space, the run time system is not in a position to decide how long to keep an unsatisfied request pending: it may be a programming error, or there may legitimately be an indefinite delay. A separate stream space fetch call which polled or waited for up to a specified period of time before returning if unsatisfied was considered, but it was decided that this would only complicate matters for the programmer. A primitive is provided, however, that prints a description of the current contents of stream spaces and the current set of requested labels, in order to aid in debugging.

5.5.1 Stream Space and Initial Configuration

The labelling of channels in the configuration declaration scheme of Section 5.2 are realised through labelled streams attached to the channels, which are deposited in stream space. The application is free to obtain copies of these from stream space, but only after the call which declares the creation of the channel has taken place. An attempt by the configurer to fetch a stream attached to a labelled channel before this point will block forever.

Similarly, labelled handles to stream and port interfaces (declared in the calls of Section 5.2.1) are also deposited in stream space with the given labels. The programmer is in this case required to observe a stronger convention to avoid the possibility of deadlock: no attempt should be made to fetch them from stream space until all creation declarations necessary for the complete configuration have been made.

The run-time system devolves configuration establishment to the configured incarnations themselves. They run user-level preamble code before the main program is called, to fetch and generate their parameters. This strategy means that the processing involved is carried out concurrently and therefore potentially in parallel. It also means that ports attached to new channels can be created at the correct incarnation automatically. Each new incarnation deposits in stream space labelled streams attached to new channels it creates, and fetches streams it requires, either to use directly as interfaces or to refer to other channels to create an attached port. Freedom from deadlock is simply assured, by causing each incarnation first to create all channels it is required to create and depositing them, and then to fetch streams referring to channels created elsewhere.

5.5.2 Operational Considerations

Stream space has two drawbacks when compared to propagation by message passing: the first is that it is a critical point of failure, the second is the performance overhead incurred in using it. Stream space is currently implemented as a per-RDC incarnation, with put/get operations implemented as invocation messages sent to it. This implementation choice avoided extending the kernel and led to an implementation which can relatively easily be changed and extended with monitoring and debugging facilities.

Replication of stream space contents using replica incarnations would be necessary to provide resistance to failure, but this has not been attempted. Ensuring atomicity of *ss_get* operations in cases where the number of reference copies is limited would make replication a non-trivial undertaking. Atomic reference propagation (i.e., in which it is known either that propagation did successfully take place to all of a set of incarnations, or that it did not take place to any of them) is left to applications to provide, based on message passing.

Stream space is not employed in forking, so that a forked incarnation can be linked into the existing structure without dependence upon it. The forker is anyway blocked in the current implementation until the new incarnation can begin execution, and any streams to the new incarnation which the parent has declared are then obtained immediately from it by the run-time system.

Using the current user-level implementation of stream space, the minimum cost of a call to *ss_get* (for an uncached reference) can be estimated as 3 milliseconds – i.e. slightly more than the cost of the invocation call necessary to propagate the reference (see Section 8.6 and Appendix A). This seems a small price to pay for stream space semantics. Message-based propagation remains as an option for those programmers concerned with optimising performance.

5.5.3 Caching

The run-time system caches references obtained from stream space. *ss_get* first looks for the label in a cache of references obtained from previous calls, and the request is passed on to the software implementing the stream space repository only if not found in the cache. Caching affords a considerable performance advantage over non-caching, since a synchronous invocation takes at a minimum several milliseconds, as opposed to a local search conducted in the caller's address space, which takes typically a small fraction of a millisecond. Moreover, stream space caching automatically obviates redundant propagations which might occur unless incarnations kept track of reference propagations. For example, a scheduler processing tasks for clients can label the references pertaining to their requests and deposit them in stream space. The task descriptions it gives to its workers include these labels. Because of caching, the workers only fetch the corresponding references from the stream space repository at most once, even though they can execute several tasks requiring the same set of references.

References whose copying from stream space is restricted in multiplicity are not cached, since otherwise the restriction would be unenforceable. When such references are returned to stream space the local copies are automatically destroyed. Also, stream interfaces which are declared with labelled handles at the time of initial configuration are not cached. This is because the handles can be used to attach the streams transparently to a different channel, as described in the next chapter. If such a stream were held in a cache, a call to *ss_get* with the original channel label would return this stream erroneously.

5.5.4 Monitoring Propagation

The propagation and destruction of streams can lead to configurations in which no streams exist that are connected to a certain port, or in which streams continue to be propagated and invocations attempted using them, even though no port is attached to the same unicast channel. No way could be seen for the operating environment to decide upon these conditions except at the expense of considerable kernel-to-kernel notification traffic. For at any instant propagated references could exist potentially at any node, or could be contained in a message in transmission or awaiting reception at a port. A facility whereby an incarnation can request that it be notified explicitly through the event mechanism when another incarnation has propagated a reference of interest to it (e.g. a stream connected to one of its ports) has been considered. It could straightforwardly be built upon existing kernel mechanisms, but has not been implemented.

Control over Clients

When a server propagates a stream to a client belonging to a different RDC, the stream itself can be made to fall under the management of the server. It has been described how the stream's propagation can be prevented by setting a reference descriptor flag. Further control is available through a stream handle which is generated by the server at the point of propagation (*inc_reply* can be given a flag to cause it to return a handle to either a propagated port or a propagated stream). The ability to manage streams is based upon the principle that the server needs to be able to control access to its facilities in order to protect itself and also, as will be fully described in the next chapter, to perform reconfigurations in relation to its clients. The client can, however, autonomously destroy its stream. It is presently possible to obtain notification of the destruction of a stream, so that a server can clean up when a client disconnects unexpectedly. A server can freeze and unfreeze a remote stream, thereby blocking and unblocking attempts to send invocations using it, as a means of preventing one client from over-running others in its use of the server. Finally, a facility to invalidate clients' streams could also be added straightforwardly, using the existing stream control mechanism.

5.6 Summary

This chapter has described mechanisms by which an incarnation can declare the creation and integration of new incarnations into an RDC's configuration. The programmer does not have to be concerned with the operational details of how the new incarnations' connections to other incarnations are realised. The chapter described the use of stream space to propagate references between incarnations which do not have a stream-to-port connection Propagation of ports has not yet been fully described, however. It is one of three related mechanisms to be described together. To complete the description of structural reconfigurations in the RDC model, it remains to examine reconfigurations which change the server incarnation that processes requests from a given client incarnation.

Chapter 6
Reconfiguring Clients and Servers

This chapter is concerned with reconfigurations applied to fluctuating sets of multiple peer incarnations, which between them process requests from a set of client incarnations. The chapter describes the necessary reconfiguration mechanisms available under the RDC model, and analyses and defines application requirements for performing the reconfigurations. It illustrates the use of these mechanisms and shows how application requirements are met in the cases of the example text processing and printer server RDCs (introduced in Section 2.4.2).

6.1 Multiple-peer Servers

The rationale behind the use of multiple peer incarnations includes the following considerations:

1] load balancing – if more than one printer exists, for example, then the printing load can be divided between them.

2] separation of concerns – requests from clients of certain types are handled by incarnations of appropriate functionality (laser printing versus line printing).

3] parallel computation through data division – for example, the incarnations implement a database, each handling requests pertaining to a discrete section of it.

4] replication for fault tolerance – every request is multicast to a set of incarnations which process it and reply together.

Of related concern are configurations in which a single peer is normally used, but one which can be replaced by another:

5] replacement as part of node administration or system evolution.

6] replacement for fault tolerance – clients are connected to a replacement incarnation if the original fails.

Configurational and reconfigurational requirements for fault tolerance – particularly where multicast is used – are largely beyond the scope of this work. The RDC model does, however, take a general approach to those of the reconfigurations implied by the above which take place in the absence of failure (1-3 & 5). The types of reconfiguration which are to be addressed here are to:

Add a server peer and redistribute clients.
The printer server RDC of Section 2.4.2 can expand its operations to two printers and therefore two printer spooler incarnations. Some clients connected to the first can then be re-connected to the new spooler incarnation so as to balance the load.

Change the association between existing clients and server peers.
If, for example, all clients connected to one printer spooler close their connections to it whilst the other has more than one client connected, then some of these can be re-connected to the first.

Withdraw a server peer and replace it with an extant one or a new one.
When a printer node is withdrawn, clients connected to the corresponding printer spooler are to be reconnected to the remaining one.

The problem of achieving these reconfigurations falls into two parts. A server peer such as a printer spooler may manage state related to a client's activities, which it retains between client calls. The implications of this are analysed in Section 6.3. The other component of the problem, to be dealt with first, is how invocation messages are to be switched between server peers.

An intermediate incarnation could be used to manage these reconfigurations transparently by forwarding requests. Each time a client invocation is received, it is forwarded to the server incarnation able to handle it or best able to handle it on the basis of load, functionality or data. Such an incarnation could be made to reside at the client's node in situations where fault tolerance is required (although client migration would then be a problem).

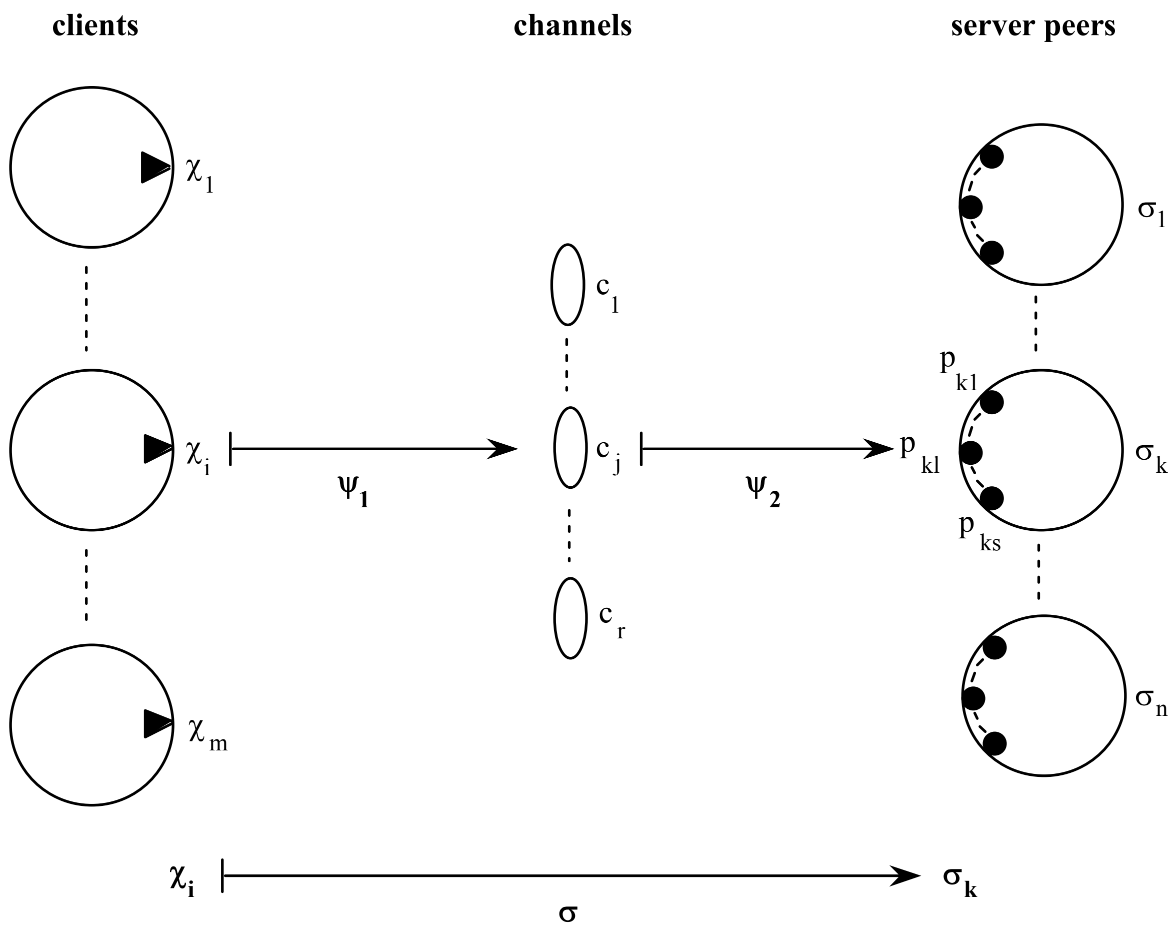
With some servers, clients usually make successive invocations which are handled by the same server incarnation. For example, this is the case if there is state held per client. A potentially more efficient alternative to forwarding is then to bind the client to this server incarnation for the duration of the calls it handles from the client. A logically direct connection between client and appropriate server peer a) has the advantage of not involving an intermediate incarnation in performing extraneous processing, and b) it potentially allows the underlying message routing software to choose a more efficient route than via the intermediate incarnation's node. Such a direct connection is that between a stream held by the client and a port owned by a server peer, and reconfiguration mechanisms applicable to these connections are described in the next main section, 6.2.

Adding or removing a client.

The other reconfigurations not included in the three just given are the addition and removal of a client from a multiple-peer server. There is little to be said about these reconfigurations from the configurational standpoint. If server peers manage separate clients at separate ports, then adding or removing a client means creating or destroying a port. Otherwise what is involved is the creation or destruction of a stream attached to the appropriate channel. The new attachment can be imposed externally by a manager within an RDC, or a stream directly connected to an appropriate server peer can be passed back when the client requests connection to a server RDC. Destruction is purely local to the stream involved.

6.2 Mechanisms to Change the Receiver

Clients can be grouped a) by means of the channels to which their streams are attached, and b) by means of the ports from which their invocations are received. The channels exist to distinguish server peers as they are introduced into the server RDC, but can persist even after they are withdrawn. Ports are used by server peers or their managers to group clients.



The function $\psi_2 \circ \psi_1$ induces the function σ from client streams to server peers.

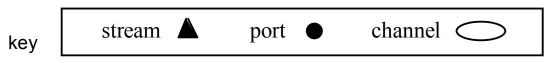


Figure 6.1: The Mappings Relating Clients to Server Peers.

In a given configuration of clients connected to server peers (Figure 6.1), there are server peer incarnations $\{\sigma_1,..., \sigma_n\}$ which service requests from clients $\{\chi_1,..., \chi_m\}$ through channels $\{c_1,..., c_r\}$. Each client is identified with a stream: an incarnation with several streams to the same service is regarded in this discussion as several clients. A given server incarnation σ_k has a set of ports $\pi(\sigma_k) = \{p_{k1},..., p_{ks}\}$.

The configurational relationship can be expressed through two functions: the function from clients to channels is:

$\psi_1: \chi_i \rightarrow c_j$ (for some $j \in \{1, .., r\}$, for each $i = 1, .., m$)

and the function from channels to ports in the unicast case is:

$\psi_2: c_j \rightarrow p_{kl}$ (for some $p_{kl} \in \pi(\sigma_k)$, for each $j = 1, .., r$. $k \in \{1, .., n\}$)

and in the multicast case is:

$\psi_2^j : c_j \rightarrow \{p_{k_1l_1}, .., p_{k_tl_t}\}$ (with each $p_{k_il_i} \in \pi(\sigma_{k_i})$, for each $j = 1, .., r$)

In the unicast case we then have a function giving the unique server peer which currently processes requests for a client:

$\sigma: \chi_i \rightarrow \sigma_k$ *iff* $\psi_2(\psi_1(\chi_i)) \in \pi(\sigma_k)$.

There are three mechanisms for changing this configuration (Figure 6.2), which are now described.

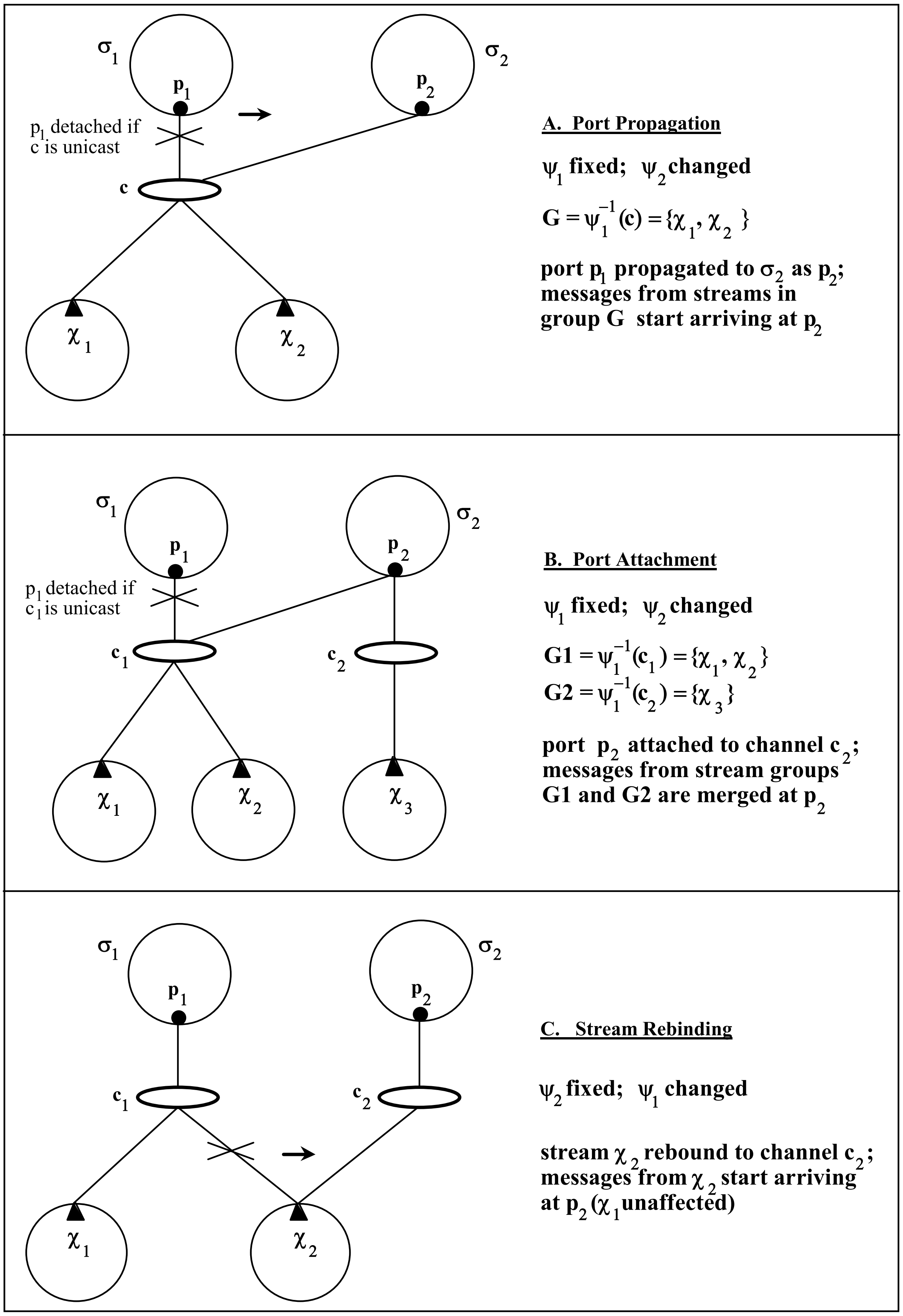
6.2.1 Propagating Ports

The first mechanism is that of propagating a port to another incarnation (Figure 6.2 A). To simplify semantics, only ports attached exclusively to either multicast or unicast channels can be propagated (although for other purposes a single port can be attached to both types).

- If the original port is attached to a unicast channel, the new port becomes the sole port attached to it, and the original port is, as it has to be, automatically detached from it. Invocation messages sent using a stream attached to this channel are routed to the new port when it is attached. In the case of propagation using stream space, the system automatically limits to one the number of copies of the port which can be taken out at any one time.

- If the original port is attached to a multicast channel, the new port becomes added to those to which invocations are multicast.

This mechanism does not require knowledge of what streams are attached to the original port's channel(s), but has a blanket effect on all such streams. A "client" of a multiple-peer server RDC could in fact be a group of incarnations, created as a result of propagating an original stream connected to one of its ports. If a different incarnation becomes the implementor of the service for the multiple-incarnation "client", then port propagation is appropriate since all members of the group are affected together.



A. Port Propagation

ψ_1 fixed; ψ_2 changed

$G = \psi_1^{-1}(c) = \{\chi_1, \chi_2\}$

port p_1 propagated to σ_2 as p_2; messages from streams in group G start arriving at p_2

B. Port Attachment

ψ_1 fixed; ψ_2 changed

$G1 = \psi_1^{-1}(c_1) = \{\chi_1, \chi_2\}$
$G2 = \psi_1^{-1}(c_2) = \{\chi_3\}$

port p_2 attached to channel c_2; messages from stream groups G1 and G2 are merged at p_2

C. Stream Rebinding

ψ_2 fixed; ψ_1 changed

stream χ_2 rebound to channel c_2; messages from χ_2 start arriving at p_2 (χ_1 unaffected)

ψ_1 is the function from client streams to channels; ψ_2 from channels to server ports.

key: stream ▲ port ● channel ⬭

Figure 6.2: Three Mechanisms to Cause Invocation Messages to Arrive at a Different Incarnation.

In the first implementation [WORMOS87a], ports could be attached to at most one channel, and port propagation was the only mechanism which allowed clients to be grouped for the purposes of reconfiguration. Whilst some applications need to separate clients or groups of clients using multiple ports, use of this mechanism implies that the server peers must manage variable numbers of ports at which they receive requests. It therefore provides no scope for separating reconfiguration from the application algorithms, even when this is otherwise partially or wholly possible.

6.2.2 Attaching Ports

Another primitive *inc_attachPort* was introduced which can be applied externally by a manager incarnation. This causes a port to become attached to a channel (Figure 6.2 B). If it is a unicast channel, then the other port currently attached is normally automatically detached from the channel. If it is a multicast channel, the newly attached port becomes a member of the corresponding multicast group. When *inc_attachPort* is externally applied, the port is referred to by a port handle, whose deposition into stream space was specified as part of the port's declaration when the peer was created.

A printer spooler incarnation belonging to the printer server RDC uses only one port to receive requests and therefore, unlike peers using port propagation, does not distinguish structurally between its clients. In this case, groups of clients become connected to or disconnected from this port by attaching or detaching it to or from channels. The output of invocation messages from channels is thereby merged or de-merged with or from that of other channels at the port.

In cases where source and functionality are distinguished by message data, port attachment can also be usefully applied locally when the incarnation knows the channel attachments of its ports. Messages arriving over multiple channels – both unicast and multicast – can then be merged at a single port to simplify reception. Moreover, the messages can be de-merged by separating them out from one port to others. Thus port propagation is always in effect feasible, even if the port has to be split into two (with channels of either type divided between them). The two ports can be propagated separately, and the new ports recombined by the recipient.

6.2.3 Rebinding Streams

Both of the above mechanisms change the channel-port configuration function ψ_2 (or ψ_2') and leave the client-channel grouping function ψ_1 intact. They do not suffice for reconfiguring individual clients' associations with server peers, unless each is associated at any one time with a single channel. In cases where clients cannot be distinguished by message data, this means that reconfigurable servers are bound to carry the overheads associated with reception from as many ports as there are clients.

The next and last mechanism is that of rebinding an individual stream: of causing it to become attached to a new channel (Figure 6.2 C). Rebinding is only provided for streams attached to unicast channels; there is no need to rebind streams to different multicast channels. Its effect is to cause invocation messages to be sent to the (unique) port attached to the new channel. The stream to be rebound is referred to by a stream handle, which was either deposited into stream space at the time of the stream's creation, or was created at the time that a server propagated the stream back to a connecting client.

Stream rebinding differs operationally from the previous two mechanisms. The operation is applied directly to the incarnation possessing the stream, whereas in the other two cases re-routing is effected using a search algorithm that locates the new physical destination for the messages affected, based on the identifier of their channels (this is described fully in Chapter 8).

This operational distinction is provided, firstly, in case of failure. If the original destination, which is where re-routing information would be located, becomes unreachable, then there is no way that invocation messages can be routed to a port attached to a different channel unless they bear its identifier. The stream's host resorts eventually to broadcasting, but will attempt to find the original channel unless it has been directly rebound with the new channel's identifier. Secondly, since no acknowledgements are given for datagrams, stream rebinding is the only means available for reconfiguration when *inc_dgram* is used.

6.2.4 Moving the Message Queue

The above mechanisms are concerned with re-routing invocation messages to new ports: but this leaves open the question of what happens to messages which have been sent but not received when a reconfiguration takes place. These are messages still "on the wire", and those queued at the original port. The first issue to be addressed is consistency in meeting delivery guarantees. The second is the set of options available in the model.

1] Messages which have been sent but not queued at a port at the time reconfiguration takes effect will be routed to the new address, so that any invocation message is consistently sent to one port or another attached to a unicast channel – never to both. In the multicast case, such a message may or may not be delivered to the new port. This is consistent with the unguaranteed delivery which anyway applies to invocations over a multicast channel.

2] Messages which have already been delivered to the port can be dealt with as the reconfiguring incarnation determines. The reconfiguring incarnation specifies whether the message queue of the port affected is to be left intact, or whether relevant queued messages are to be moved to the new port. When rebinding a stream or detaching a port (in attaching another one), some queued messages do not derive from the relevant stream or channel, in general, and movement of these is not in any case in question. When propagating a port, the entire queue is eligible for movement.

When a port is propagated and the original port is kept, then the current message queue is retained and these messages can be received as normal. In the unicast case, a flag can be set to specify whether or not invocation messages continue to be queued until the new port has been received and messages are re-routed. If not, then in the meantime all transmissions of invocation messages sent over unicast channels which previously would have been queued at the original port have to be made pending, in the absence of a known destination. This amounts to choosing between, on the one hand, allowing the message queue at the original port to grow in the period before the propagated port is received and, on the other hand, blocking incarnations which are trying to send. The latter particularly affects those incarnations attempting to make asynchronous invocations – synchronous invokers are anyway blocked until their messages are received.

Reconfiguring Under Failure

An exceptional case of applying a communications reconfiguration is when the owner of a port has become unreachable from a given stream or streams, through network partition or node failure. Then the queue of messages at the port is also unobtainable, but nonetheless it may be desirable for subsequent invocation messages sent using these streams to arrive at a backup incarnation which is reachable from them (i.e., in the same network partition or on a functioning node) [HILTUNEN]. In this case, the stream rebinding or port attachment primitive is used by an incarnation in the same partition which has detected the unreachability, with a flag to indicate the absence of a port from which queued messages are to be moved. In the current implementation, an invocation attempt to a failed node gives up and returns a failure code after a system-defined number of retries lasting about five seconds. If a manager took longer than this to react, a client would perceive the failure condition, and would have to repeat the invocation primitive for the reconfiguration to take effect.

An alternative considered was a type of port which could be registered with the local kernel as being a backup port: i.e. which was associated with but not attached to a particular unicast channel. When unable to locate an attached port, a sending kernel could request the attachment of the backup port (whose existence was notified in responses to messages searching for an attached port), and then deliver the invocation message to it. This would take place transparently: clients need not know of the first failure. The incarnation owning the backup port-turned-attached-port could either be a peer of the unreachable incarnation able to provide a substitute service, or an intermediate manager. A manager could apply a reconfiguration to reroute the unreceived messages to a backup incarnation, when notified through the event mechanism that they had arrived. Backup ports have not been implemented.

Application techniques for recovering from failures using the features of the RDC model and its Wormos-based predecessor have been examined elsewhere [HILTUNEN, LEUNG]. The features employed include the creation of duplicate incarnations through forking, the use of multicast and use of the above communications reconfiguration mechanisms.

6.3 Application Requirements

Applications have global requirements to be maintained under the reconfigurations of Section 6.1. These are to maintain:

- Distributed data and device state consistency. For example, the states of the scheduler and worker of the text processing RDC are each consistent when an entire task has been obtained by a worker. For the printer server RDC, consistency means printing each document entirely at one printer or another, and not partly at each of several printers.

- Safety and liveness properties. These are that incoherent stream-port connections are not made, invocation messages are not lost, and deadlock and livelock are avoided. It is assumed for the purposes of this treatment that manager incarnations which connect and re-connect other incarnations do so with sufficient knowledge of the incarnations concerned, and are not liable to make incoherent connections. Protection against message loss will be considered with the consistency requirement. Strategies to guarantee against deadlock and livelock are application-dependent, and beyond the scope of this work.

Consistency is the central concern here. Consistency is achieved by the completion of *transactions.* A transaction is semantically defined in application-specific terms: the acquisition of a task by a worker from a scheduler (acting as a task server) is an example of a transaction. It leaves the scheduler's task list in a consistent state, and the worker with a well-defined task description.

Transactions are defined operationally as a sequence of communications exchanged between an initiator (the client) and a recipient (server peer). An incarnation can act as both a client and a server simultaneously to other incarnations, so the "client" and "server" labels used here are to be taken as relative to the transaction taking place.

A client χ enters into a series of transactions $t_u,\ldots, t_v,\ldots, t_w$ whilst connected to the server peer $\sigma_j = \sigma(\chi)$ (because of reconfigurations, σ is a function of time). Each transaction t_v operationally consists of an initial invocation made by the client χ, followed possibly by other communications made by χ, and communications made back to χ by the server peer σ_j. Communications can either be:

- invocations (made perhaps using auxiliary streams and ports), or
- data copying performed using auxiliary buffer handles.

So that transaction initiation is well-defined with respect to σ_j, these communications must affect it after it receives the initial invocation. In particular, χ may use a buffer handle in connection with t_v only after the initial invocation is known to have been received.

- It is assumed that the client can always tell locally when a transaction is complete. Up to this point it is said to be *engaged in a transaction*. It can in principle always confirm completion, either through i) completion of a synchronous invocation, ii) use of the event mechanism where *inc_asinvoke* is used; or iii) by receipt of an invocation message back from the server. Note that, when *inc_send* is used, a transaction can be complete as far as the client is concerned, even when the server has not yet received the invocation message.
- It is further assumed that none of the communications belonging to a transaction transfer data or references belonging to another transaction. Thus

transactions can overlap in time, but are cleanly separated in terms of the data passed in the communications involved.

Transaction processing by an incarnation can entail entering into another, *consequent*, transaction. Transactions are required to take place in bounded time, regardless of the existence of consequent transactions upon which they depend for completion. In particular, they are assumed to take place without deadlock.

6.3.1 The Conic Reconfiguration Criterion

[CONIC89c] describes a model of managing reconfigurations. Although the reconfigurations are achieved through different mechanisms to those used here, they achieve the same ends as the client-to-multiple-peer server reconfigurations of Section 6.1. Instead of using the reconfiguration mechanisms introduced above as specific to the RDC model, the Conic model employs basic link and unlink operations between communications interfaces. Because of its relevance, the Conic work is now outlined. It will be shown, however, that it does not extend to the case of loosely coupled clients and server peers, and the RDC model solution for this case is given.

The notion of transaction defined above is derived from the one given in [CONIC89c]. Two notions of state are used to establish when a Conic node (a process which is the equivalent of an incarnation for the purposes of this treatment) may undergo a reconfiguration. These states are *activity*, *passivity* and *quiescence*. These states are defined in terms of transactions, and not the underlying communications primitives.

Quiescence is the state required of a Conic node in order that it can undergo a reconfiguration, whether this be its complete removal, or its being linked to or unlinked from another Conic node. In this state:

(Q1) it is not currently engaged in a transaction that it initiated;

(Q2) it will not initiate new transactions;

(Q3) it is not currently engaged in processing a transaction;

(Q4) no transactions have been or will be initiated by other Conic nodes which require processing by this node.

It is shown [CONIC89c] that by making each of a well-defined set of Conic nodes passive, quiescence can be reached, and it can be reached in bounded time. A node is passive if and only if it processes transactions and initiates any consequent transactions, but is not currently engaged in a non-consequent transaction it initiated, and it will not initiate a non-consequent transaction. The set of nodes which are passivated for a node Q to reach quiescence is: i) Q itself, ii) all nodes which can directly initiate transactions on Q; and iii) all nodes which can initiate transactions which result in consequent transactions on Q.

Quiescence is also a well-defined and sufficient criterion for the complete removal of an incarnation from an RDC. The above definition of quiescence and the auxiliary definition of passivity can be immediately applied to incarnations instead of Conic nodes. In regard to the validity of quiescence as a precondition for applying a reconfiguration to an incarnation, the only possible stumbling block is the allowance of asynchronous transactions based on the RDC communications model. However, the definition of passivity implies that an incarnation processes all requests and leaves its queue empty after no more requests are sent. This guarantees that an incarnation is not removed with requests in its queue. If a client sent an invocation message reliably, then the completion of its invocation implies the arrival of the message in the server's queue: the message cannot be on the wire. And the definition given above of "engagement in a transaction" precludes the withdrawal of a node whilst or before another incarnation attempts to copy data to it or from it asynchronously.

On being directed to the quiescent state, an incarnation may, additionally, perform finalisation actions before notifying the manager that it is ready to be withdrawn. For example, the printer spooler incarnation may send final data for printing. It would be erroneous to withdraw the incarnation immediately the data were received by the spooler, even though the transaction, operationally defined, is complete.

6.3.2 Limitations of Quiescence

The state of quiescence cannot necessarily be reached when a server peer's clients belong to independent RDCs, rendering it unsatisfactory as a precondition for reconfiguration in this case. This is because the server's management cannot interact with the clients at an application level. Passivation, however, can only take place via an interaction with the client's algorithms directing it to reach this state. Quiescence cannot then necessarily be reached by a server peer in bounded time, since it may be always engaged in some client or other's transaction.

Quiescence at the level of incarnation is also too strong a condition in circumstances where what is required is for a single client to become connected to a different peer. There is no reason, in principle, why a peer should not respond to transactions from other clients whilst reaching a state of consistency with respect to the client to be reconfigured.

6.3.3 Server-Quiescence

Because invocation messages are queued at ports, it is possible to forestall processing of a new transaction by not receiving its first invocation message, even if it has arrived at a port. Once the final invocation message belonging to a transaction has been received by a server peer, it is then able not to receive further messages from the same source. This can be achieved either:

- by selectively receiving messages only at certain ports (if necessary, an individual client stream can be rebound to a new channel to which a local port is attached, thereby isolating invocation messages at this new port);
- or by using a call available to examine the origins of queued messages at a single port, and the facility to receive only messages deriving from a specified stream when using *inc_receive*.

Given these facilities in the RDC model, it is possible to define a criterion for being able to perform client-server reconfigurations which does not suffer from the shortcomings of quiescence.

Definition of Server-quiescence

An incarnation Σ is said to be in a state of *server-quiescence* with respect to one of its clients χ if Σ is not engaged in a transaction initiated by χ, and will not receive the first invocation of a next transaction from χ, if such is made.

The following observations can be made when a server Σ is server-quiescent with respect to a client χ:

- χ has completed any last transaction processed by Σ (otherwise Σ would still be engaged in it), and invocations belonging to any next transaction have been or are about to be queued at Σ, but have not been processed.
- a reconfiguration is then possible in which some other peer Σ' processes remaining transactions, including any whose initial invocations were queued at

Σ. If it processes them as Σ would have done had χ remained connected to it, then this reconfiguration is transparent to χ.

Crucially, server-quiescence is a state which can be reached by Σ itself, and does not require interaction with χ's algorithms.

Transferring a Client Between Server Peers

To re-associate the client χ consistently from server peer Σ to a new peer or another extant peer Σ', Σ is made server-quiescent with respect to χ, so that it is in a consistent state with respect to it. This state has two components: external device state and internal state. The internal state consists of local volatile data and auxiliary references (buffer handles, streams, ports) associated with processing transactions from χ.

To achieve the transfer of the client:

1] One of the three reconfiguration mechanisms described above has to be used, incorporating movement of the appropriate queued messages to the new server peer port.

2] Any local device state associated with χ has to be made consistent, and the internal data and references representing the state of processing for χ must be transferred to Σ' before it can process the next transaction from χ. For example, in the case of a printer spooler, this means that any data belonging to a particular document that have been buffered but not printed (for want of completion) have to be sent to the new spooler.

Withdrawing a Server Peer

A server peer to be withdrawn may itself be a client of one or more other incarnations. What is then required is a combination of Q1 - Q2 from the definition of quiescence, together with server-quiescence to replace Q4. An incarnation is *replacable* if:

(R1) it is not currently engaged in a transaction that it initiated;

(R2) it will not initiate new transactions;

(R3) it is server-quiescent with respect to all of its clients.

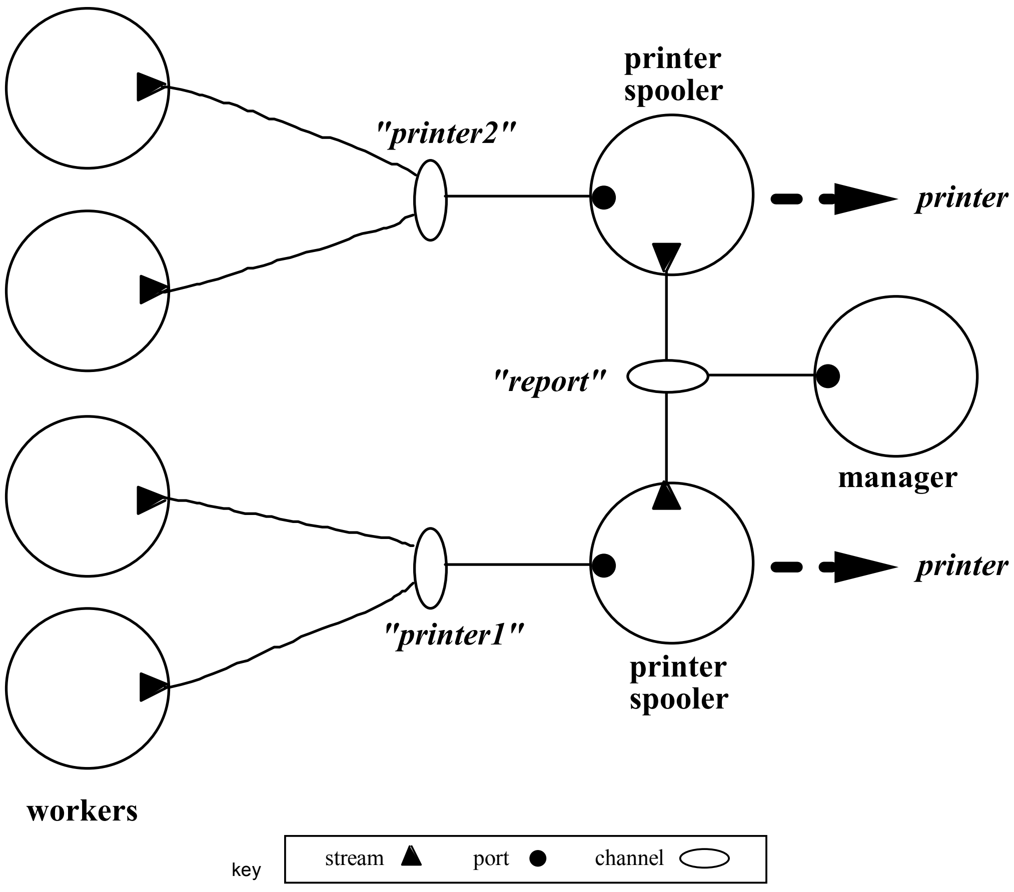


Figure 6.3: Configuration with Two Printers.

Removal Without Replacement

If an incarnation (whether or not it is a server peer) is to be withdrawn from an RDC without providing a replacement for its clients, then the clients must be made fully quiescent. This is so that there are no outstanding invocation messages from the clients which are buffered but which will not be processed. If there were to be such messages, then they will be lost.

In summary:

- To replace a server peer with respect to a client, it has to be made server-quiescent with respect to that client.
- To remove a server peer but continue to service its clients with other peers, it has first to satisfy R1 - R3, and to perform any necessary finalisation actions.
- To remove an incarnation entirely from an RDC, without replacement, it has first to satisfy Q1 - Q4, and to perform any necessary finalisation actions.

6.4 Reconfiguring the Example RDCs

In the example text processing RDC with its four workers, the initial configuration of the RDC employs only one node with a printer. The issue now addressed is how to add a second printer node, and hence printer spooler incarnation, to the running RDC. This can be achieved as follows:

1] The manager receives from the printer spooler a report that the workload represented by the queue of unreceived messages at its port has exceeded a threshold value. There is a primitive *inc_testPort*, which returns the number of invocation messages currently queued, and their (user-level) headers, containing worker identifiers and document lengths. This can be used to measure workload.

2] On inspection of this performance value (and perhaps others), a decision is taken as to whether the addition of a printer is required – either by a human administrator interacting with the manager incarnation, or independently by the manager incarnation. The availability of another printer node can be ascertained by interrogation of the pool manager.

3] To effect the reconfiguration, the manager firstly creates a second printer spooler at the new node, with once again a stream attached to the "report" channel (for continued monitoring), but with a port attached to a new channel, "printer2".

4] The manager then rebinds the streams of two of the four workers, so that they become attached to this new channel (Figure 6.3). The workers are evenly divided on the assumption that they are likely to make requests amounting to approximately equal total workload. Stream rebinding is the only mechanism which can be used, given that all worker streams are initially attached to the same channel.

```
1  node = ..NEW PRINTER NODE..;
2  ss_isPort(&incparams.ip_ifaces[0], NEW_CHAN, "printer2", "printer2.han");
3  ss_isStream(&incparams.ip_ifaces[1], "report", NO_HANDLE);
4  inc_create(MODULE_SERVER, node, &printers[1],.., &incparams);
5
6  new_stream = ss_get("printer2");
7  port_handle = ss_get("printer1.han");
8  for(workerid = ..SET OF WORKERS TO REBIND..) {
9      sprintf(handle_label, "worker%d", workerid);
10     strm_handle = ss_get(handle_label);
11     inc_rebindStream(strm_handle, new_stream, port_handle);
12 }
```

Figure 6.4: Manager Code to Integrate New Printer.

Fragments of the manager code for performing the reconfiguration are given in Figure 6.4. After establishing the identity of the new printer node, the manager creates the new printer spooler there. It declares the interfaces (lines 2 - 3), and then issues the creation call (line 4: details are omitted for the sake of clarity). At line 6, *ss_get* returns a stream attached to the new printer's channel, and it is to this channel that some of the worker incarnation streams are to be rebound. The set of workers to reconfigure, given the homogeneity of the work they are carrying out, is any two of the current configuration of four workers. *ss_get* is used in line 7 to get a handle to printer1's port, whose creation was stipulated when this incarnation was created. *ss_get* is used in line 10 to obtain from stream space a handle to the streams of each of the chosen workers, and finally *inc_rebindStream* is used in line 11 to effect the rebinding. The argument *port_handle* is provided so that unreceived messages sent from the stream are automatically forwarded from the referent port before the stream is rebound. Thus all queued and future messages from these two workers are directed to the port of the new printer spooler, but the other two streams are unaffected.

The following points can be made about the reconfiguration of this RDC:

- If this example were to be altered to that of the printer server RDC with clients belonging to other RDCs, then reconfigurations aimed at balancing load across the printers would still be possible using the same mechanisms. Connection to the printer server would be made via the manager, which propagates back to the client a stream to one of the spoolers, generating a stream handle referring to it as it does so. The decision as to which clients should be associated with which spooler would have to be based on several factors. These include the workload that is currently queued for receipt, the component of this which is due to each client, and possibly the workload each has imposed in requests so far, as reported by the spooler in terms of numbers of documents and sizes. This is not further discussed here.

- The worker and spooler incarnations do not call any reconfiguration primitives when a new spooler is added. The only code mentioned in connection with this reconfiguration is for reporting printer state. What has not been described, however, is whether the printer spooler is blocked by the manager when it reports its condition, and whether this is the only reconfiguration code included with the application algorithms.

The consistency requirement for the text processing application is that every document is printed, but integrally at a single printer. Although it has been implicitly assumed so far that the printers are sufficiently close together for it not to matter if the set of documents is divided between them, it is unsatisfactory for an individual document to be split up.

If the application is such that document data is always transferred integrally with a single invocation, then the spooler can send the data to the printer directly it is received. This constitutes a single transaction. In this case, an alternative reconfiguration is to attach *printer2*'s port to *printer1*'s channel, leaving *printer1*'s port unattached but retaining its queue of messages, and causing subsequent messages to arrive at *printer2*'s port. The attachment is then reversed as necessary to maintain a queue of work at each printer. Whatever the mechanism, the manager blocks the spooler when it reports its state whilst the mechanism takes effect, but no other interaction with the application algorithms is required.

Otherwise, the data have to be buffered, and there are two possible definitions of transaction for the purposes of changing the spooler which handles a particular client's requests:

i) in which a transaction consists of the sending of all document data to the printer server, even when this takes several invocations; the data are buffered until complete, and then printed.

ii) in which a transaction consists of a single asynchronous invocation, which transfers possibly only part of a document's data.

On the first definition, the spooler has to be directed by the manager to continue to operate to reach a state of server-quiescence with respect to the client concerned. It can then print the complete data and notify the manager of this state.

On the second definition of transaction, the spooler is in a state of server-quiescence with respect to all of its clients at the point when it notifies the manager of its queue length. However, it will have to be directed by the manager to send partly-buffered data to the other spooler before the reconfiguration can be mechanically effected.

Dealing with Node Withdrawal

We turn, finally, to the question of possible responses to the imminent withdrawal of an extra node by the pool manager. In the case of the printer spooler, the incarnation must be removed since no printer device remains for it to control, and the clients affected must be re-connected to the remaining spooler. It is therefore directed to reach a state of readiness for removal by the manager, at which point it synchronises with the manager. In this case, it is sufficient for it to become server-quiescent with respect to its clients (R3), since the spooler is a client only of the manager (R1, R2).

In the case of the worker, Q1 - Q4 apply. It is a client of both the scheduler and the printer server. A transaction with respect to the scheduler is a complete task acquisition. Despite the existence of two possible definitions of a transaction considered earlier in connection with the printer spooler, when the worker is to be removed the only definition of a transaction which leaves the spooler in a consistent state is the sending of complete document data to it – otherwise the worker could disappear having sent only part of a document to the spooler. Possible finalisation actions for the worker are i) to complete current task processing, or ii) to abort it, if no data have been sent for printing, and send the task description back to the scheduler, to be processed by another worker. For a worker to be given a directive by a manager, a channel is required, to which the worker has an attached port (not shown in the figure). The worker would be required to test for a message at this port, or to set up a software interrupt handler called when a message arrived from the manager.

The other option in this case is the migration of the affected worker to a node which remains allocated. This option has the merit that migration would be transparent to all but the manager incarnation. If task processing times are indefinite, it could be the only option that can be carried out before the pool manager's withdrawal notification period expires, at which the worker would be lost if still resident there. Migration has the disadvantage that it would increase the load on one of the nodes, but once migrated the worker can be removed when it is convenient. Migration is also not always possible, for lack of memory resources. This is also a problem when the client-related internal state has to be transferred to another peer, but is less liable to overflow memory resources since less stack and heap duplication is involved. Migration is discussed in full in the next chapter.

6.5 Summary

This chapter has considered sets of peer server incarnations as a structural unit, serving either clients within the same RDC as the server peers, or independent clients belonging to other RDCs. It has described how incarnations can be removed from these configurations and replaced in them. Three reconfiguration mechanisms are available, for use either by the server peers themselves or a separate manager incarnation. The mechanisms differ in the incarnations they affect, whether or not they can be applied transparently, and in their utility under failure conditions.

The chapter went on to discuss application requirements to maintain consistency when reconfiguration takes place, and it arrived at general conditions to be met according to whether an incarnation is to be replaced or removed without replacement, and according to whether its clients belong to independent RDCs.

The application of these developments to the example problems introduced in Chapter 2 shows the possible extent of interaction between application algorithms and reconfiguration management algorithms. There needs, in general, to be both synchronisation and transfer of monitoring and state information between management and application before reconfiguration mechanisms can be brought into play.

Chapter 7
Control and Migration

This chapter moves on from the control of communications between incarnations to describe the mechanisms available for controlling the incarnations themselves. In particular, it covers incarnation migration. The mechanisms are described in the context of the requirements implied by node allocation and load balancing as performed by the pool manager, and of the need to enforce termination.

7.1 Parents and Control Mechanisms

If incarnation *A* possesses an incarnation handle for incarnation *B*, *A* is said to be a parent of *B*, and *B* the child of *A*. When a parent first creates a child, it is by default the only parent (in a load-balanced RDC, a pool manager incarnation is made a second parent at creation time – see Section 7.2). However, if the incarnation handle is propagated to other incarnations, the recipients become parents, and thus the child can come to have multiple parents. In addition, every incarnation possesses by default an incarnation handle referring to itself, and is its own parent. This is so that incarnations can migrate themselves.

A parent is able to perform the following types of control operation upon a child:

C1] to **destroy** it: this terminates the incarnation – without giving it an opportunity to flush buffers or perform any other "last wishes".

C2] to **freeze** its execution (after completion of any current system call).

C3] to **unfreeze** it. This is necessary both to re-commence the execution of an incarnation frozen by an operation of type C2, and to cause an INCHOATE incarnation to become STARTED. A parameter to this call sets the local timeslicing priority of the incarnation at its host node.

C4] to **migrate** it. If successful, this causes the child to move to a node specified as an argument to the operation. This operation is described in Section 7.4.

C5] to register interest in **events** concerning the child.C6] to set the **dependency** of the child upon the parent, as described in Section 7.3.1.

C7] to enquire about the incarnation's **state**.

There was insufficient time to include an operation to examine and set a child's machine-level execution state, which is necessary for the purposes of debugging.

Except when the operation is invoked upon the caller, all of the above operations are performed transparently to the child concerned. In some cases, however, an application-dependent operation is required that is implemented at user level and which is performed asynchronously with respect to the application. For example, a monitoring operation which reads the value of a variable might be required to be performed in this way. This can be achieved using software interrupts caused by the arrival of a message at a port. When a parent first creates a child, it is returned a stream to an auxiliary port possessed by the child. This port is for use by the child in setting up software interrupt handlers that are vectored according to an identifier in the data of arriving messages. The parent interrupts the child by sending a standard format invocation message over its stream.

The facility for a child to have multiple parents is necessary, firstly, since some of the operations listed above are independent and are appropriately applied by different incarnations. For example, an incarnation can register interest in the event of its child's death at the same time as the pool manager migrates it to perform load balancing. Secondly, if a parent fails, another parent can take over.

The application determines the set of incarnations which may control a child. It does this by propagating incarnation handles, which are references. The kernel cannot control an incarnation, except to destroy it when a node is withdrawn. On the principle that the kernel should provide mechanisms and that policy should be flexibly implemented in user-level code, the kernel does not employ an arbitration policy between parents. So, for example, it is possible for two parents to try to migrate a common child to two different nodes simultaneously. What is enforced by the kernel, however, is the serialisability of control operations.

7.2 Support for the Pool Manager

7.2.1 Load balancing

When an RDC is run under the pool manager's load balancing service, the requirements are that all control and communication semantics are identical to those pertaining to the case in which incarnations were autonomously mapped by the RDC, except:

- the mapping of incarnations is to be controlled by the pool manager;
- and it is not to be possible for the application to interfere with this control by overriding mapping decisions.

It was decided to spread the cost of incarnation creation away from the pool manager by designing the implementation of the creation call *inc_create* so that it first creates the incarnation at a node determined by the pool manager, and then transfers control to the pool manager to enable it to migrate the incarnation. This decision is in fact forced for the case when an incarnation is forked, because no mechanism exists for an incarnation to copy another one that is STARTED. The reception of the requisite incarnation handle in a message causes a fraction of the overhead to the pool manager of that of executing a creation call. Even though an *inc_create* call potentially returns before code and initialised data are fetched, it is nonetheless blocked while a protocol is exercised to establish that the incarnation can be created at the chosen node, and while other resources necessary to create it are reserved and identified.

When a parent creates a child in a load-balanced RDC, the kernel automatically removes the right to migrate from the parent's incarnation handle, and removes the child's right to migrate itself. System RDCs, and in particular the pool manager, are able to override this deficiency in the incarnation handles propagated to them.

7.2.2 Node Allocation

In the Wormos design [WORMOS87a], no allowance for extra node allocations was made. At launch time, the RDC was allocated a set of nodes whose identifiers were stored as references. This meant that access to nodes could be protected locally to the incarnation. At first it was planned for extra allocations to be made by propagating node references to incarnations which required them. This suffers from two disadvantages: i) some allocations are made to RDCs, and others to individual incarnations; ii) to withdraw a node, it is necessary to place the incarnation's identifier on a list held by the node's kernel, which causes it to refuse subsequent attempts to host an incarnation. Such entries have to be held indefinitely and not removed until it is known that the RDC as a whole has been terminated. The entries have to be deleted if the node is later re-allocated.

In view of these considerations, it was decided to use access control lists instead in the Equus design. Each kernel maintains an access control list, whose entries are added and deleted by system calls available only to system RDCs (the pool manager). If an RDC's identifier is on this list, any incarnation belonging to the RDC can migrate or create an incarnation there. If no entry appears, these types of access are refused. If incarnations belonging to the RDC are hosted at the time access is withdrawn, they will, optionally, be destroyed by the local kernel (unless they are migrating away from the node). The same mechanism serves for the termination of whole RDCs. The disadvantage of using access control lists compared to the original scheme is that the pool manager has to contact each kernel concerned when nodes are initially allocated to an RDC, whereas previously the launch program simply wrote the nodes' identifiers into a protected data structure.

When an RDC is load-balanced, it is trusted user-level code in the current implementation which creates a new incarnation at a node dictated by the pool manager. This is a point of potential abuse. A solution would be for the incorporation of "node token" references into the model, which could be propagated from a system RDC (the pool manager) to an incarnation belonging to a load-balanced RDC which has requested a creation site for a new incarnation. Such a token would provide creation rights overriding the access control list scheme (a load-balanced RDC would be allocated no nodes according to these lists). The token would be destroyed as soon as it was used; and at most one could be possessed at any one time. This has not been implemented.

7.2.3 Monitoring

In order to monitor its state of execution, a parent can register interest in several types of event that can occur to its child. The events are detected and forwarded to interested parents by the child's local kernel. The events are detected either by awaiting them or polling for them in a generalised event call, or by the generation of a software interrupt when the events occur. The events are:

1] **termination**: the child's termination by its own action or the action of another parent or the withdrawal of its node.

2] **faulting**: it enters a faulting state when it attempts an illegal instruction or other hardware-detected error condition. The child is blocked when it enters this state if a parent has registered interest in it. This facility is provided in anticipation of the parent being able to inspect and change its register values and address space contents for debugging.

3] **creation** or **migration complete**. Incarnation creation and migration operations are asynchronous; and migrations can be blocked until the migrated incarnation reaches an acceptable state. These events return the total times taken for incarnation creation or migration to occur, for the purposes of measuring gross system performance. The pool manager is able to keep track of incarnation locations using this mechanism. Migration completion events are also provided for applications in which server incarnations are to remain loosely co-located with their clients for performance reasons. The client would have to pass an incarnation handle as part of its connection to the server. Upon receiving a migration completion event, the server incarnation would be migrated to the same node.

In addition to event-based monitoring, parents can poll to establish the state of their children. The control operation [C7] referred to above, *inc_status*, returns a child's node location and its gross execution state: executing, frozen, faulting or unreachable. The last state is returned if the local kernel receives no reply from a repeated request for information about the child (requests are eventually broadcast). It has been suggested [HILTUNEN] that parents could register interest in the event of their children becoming unreachable. This would cause the local kernel to probe for the child periodically, and generate this event when it failed to receive a reply, as with *inc_status*. This would be a consistent use of the event mechanism which could be straightforwardly implemented. The criteria of the number of probe retries carried out over a specified time, after which the state of unreachability is deemed established by the kernel, is application-independent, since it is a kernel process which responds to probes, and it does so preemptively over any user processes. An alternative considered is a pool manager service based on application-level probes, which informs incarnations that have registered interest with it when it has established the unreachability of a node. This would be less satisfactory than the event-based scheme suggested, in that the latter indicates the unreachability of incarnations, which is the primary concern, rather than that of nodes. In terms of message traffic generated, however, the event-based scheme can be expected to present more communications load, because incarnations, rather than nodes, are probed for.

Finally, *inc_status* also returns a child's memory resource usage and message queue length, to allow estimation of the cost of migrating it; and crude figures for its CPU usage, to estimate the load-balancing benefits of migrating it. Research has still to be done, however, into which are the most appropriate statistics for this purpose.

7.3 Termination and Dependency

In the initial design, the destruction of an incarnation caused the recursive destruction of its descendants: the children it had created and those they had created, etc. The destruction of the primary incarnation therefore caused the termination of an RDC. This design was made as a way of achieving termination in the absence of RDC access control lists held at nodes. It suffered from several drawbacks:

- unintentional termination of the launch program through bugs sometimes led to orphaned RDCs which could not be terminated;
- the presence of an incarnation at a failed node meant that its descendants could not be terminated;
- even when a parent is useless to a computation, it has to be made to persist through the lifetime of its children.

The inclusion of RDC access control lists provided a mechanism through which all incarnations belonging to an RDC at reachable nodes can be terminated. In the current implementation, by default, the termination of an incarnation has no effect on its children or other incarnations it created. Experience of orphaned incarnations, however, led to the provision of a facility whereby applications could selectively enforce termination dependency. This is required when an incarnation should not continue to exist without another, because it depends upon the other's functionality.

Dependency

A parent can make a child dependent upon it for its continued execution. The child can be made absolutely dependent upon the particular parent, or effectively dependent on a group of parents, each of which has individually made it depend upon it using a group dependency flag. Group dependency allows for schemes in which a parent can replace a peer it has detected to have become terminated or unreachable. If a child is absolutely dependent upon a parent, it will automatically be terminated if:

- its parent terminates;
- or if its kernel, upon making a periodic test, discovers that the parent is unreachable.

If a child is group-dependent upon a parent, termination occurs only if these conditions are met and there are no other reachable parents upon which it is also group-dependent.

Every incarnation belonging to an RDC is by default made absolutely dependent upon its launch program (which acts, for some purposes, as an incarnation), so that if the latter terminates unexpectedly or becomes unreachable, all the RDC's incarnations are eventually terminated (the launch program is normally responsible for terminating an RDC).

7.4 Migration

7.4.1 Migration Semantics

To migrate an incarnation, a call to *inc_migrate* is made, which takes two arguments: the identifier of the incarnation to be migrated, and the identifier of a destination node. Migration decisions (when and where to migrate) are not made by the kernel, but by user-level code.

The semantics of migration are that the migrated incarnation's complete data and execution state are transferred consistently from one node to another. This transfer is transparent, just as timeslicing is transparent to processes on a uniprocessor. No special devices are connected to nodes in the development computer systems, and in particular there are no local files. An incarnation's state therefore resides in its address space and in its local kernel. The transferred state includes:

- program text, heap, stack and processor registers;
- kernel-held data inherent to the incarnation (the per-incarnation data, or *pincdata*), including the incarnation's reference data, list of pending events and dependencies;

- the incarnation's message queues.

The foregoing design of incarnation communications mechanisms and port propagation mechanisms makes the communications aspects of migration straightforward to implement, without any additional mechanisms. Migration is transparent with respect to all interactions involving the migrating incarnation:

. The movement of streams is transparent to the migrated incarnation. They retain attachment information and physical addresses in their reference data.

. The movement of a migrating incarnation's ports is transparent to it and to its clients. Port movement as part of a migration is equivalent to port propagation between incarnations, and the same mechanisms are used. As in port propagation, communications taking place with the incarnation whilst or after it migrates are subject to changes in latency and throughput from those experienced before migration, as messages are redirected and as the incarnation changes its co-locality with others.

. Event notifications and operations upon the migrating incarnation – control operations (including stream and port operations) and data transfers using buffer handles referring to buffers in the incarnation's address space – can continue to be made with identical semantics. Further migration operations are delayed until the current migration has completed. A destruction operation causes the abortion of the migration. Redirection of notifications and operations is achieved through the same mechanisms used to propagate incarnation ports.

7.4.2 Operational Considerations

There is no limit to the number of migrations which can take place concurrently, whether or not they involve the same kernel. All migrations are independent. They are carried out almost entirely by the source node S (at which the incarnation currently resides) and the destination node, D. The node M hosting the manager incarnation that requests the migration only transmits a migration request to S, and receives back a response indicating either denial or a confirmation that the migration will begin, at which point the migration has been committed. Thus the manager can move on to other migration decisions without being blocked, and the task of migration is assigned to S and D. If the aim of migration is load balancing, then it is desirable for the majority of the processing to be carried out by D – this being, by hypothesis, the less loaded node.

There are other operational factors to be considered, which have a bearing that depends on the intended use for migration:

1] Migration can be blocked until certain actions have completed.

2] There is a point at which the migrating incarnation is halted at S.

3] The migrating incarnation remains halted for a non-negligible time from this point until it commences execution at the destination node – the freeze time.

When the aim is withdrawal of an incarnation so that a node can be otherwise used, then blockings are best avoided. However, an incarnation is not migrated:

- during the transfer of bulk data (e.g. as part of message transfer), to keep transport protocols simple;
- and when it is involved in a synchronous invocation.

During these activities, a migration request can be processed, but the migration itself is not performed until they have completed. Bulk data transfers are a limited problem, because the time for these is bounded by memory size. A synchronous invocation can, however, last indefinitely. This restriction is discussed further in Section 7.4.6.

Once control has returned from any such blocking activity, the incarnations to be migrated still consume resources during migration. By default, a migrating incarnation continues to execute until the kernel can no longer sustain consistency with the incarnation being established at D. It is an option, however, for the manager to freeze it directly or reduce its local timeslicing priority. Even when frozen, it still consumes memory resources. The need for swapping to disc in such circumstances is discussed in Section 7.4.6.

Migration blocking is less of a problem for load balancing. Whilst engaged in a synchronous invocation, an incarnation presents very little load on S. It is impossible to state what the load-balancing requirements in relation to migration blocking and incarnation halting are, in the absence of characterisation of the incarnation's behaviour. The activity of migration itself exerts a load at both S and D, which may or may not be comparable with the incarnation's exerted load. In this case also, the manager has the option of freezing the incarnation explicitly just before issuing the migration operation, thus overriding the kernel's attempts to continue its execution during migration.

7.4.3 The Freeze Time

Continued execution is incorporated by default because it is preferable for the incarnation involved if it is progressing towards the end of a finite task, and it is preferable for the incarnations with which it synchronously interacts. Once frozen, incarnations attempting to communicate synchronously with it are blocked until it is established at D and unfrozen again. The designers of the migration facilities of the V-system and Accent [V85, ZAYAS] concentrated on reducing the freeze time, and the latter additionally on minimising the part of the address space which is transferred on migration. Both schemes are based on the expectation that most processes touch only a small fraction of their address space segments in the time taken to migrate. V iteratively copies address space pages whilst the process executes, starting with unmodifiable pages. It re-sends those pages which the process is found to have dirtied meanwhile. It freezes the process if necessary after several iterations, to ensure termination of this procedure. The Accent copy-on-reference mechanism is employed so that a) the incarnation commences execution at the destination almost immediately, and b) only the pages it actually requires are copied across the network. They are copied as they are referenced by the process executing at the destination site. The Accent scheme is reported in [ZAYAS] to achieve a 58% reduction, averaged over test applications, in the number of bytes transferred as a result of migration.

The VME bus used in the current Equus implementation provides a raw bandwidth of an order of magnitude larger than the Ethernet used for communication in these systems. About 3 Mbyte/sec. is realisable (in contrast, [V85] reports about 0.3 Mbyte/sec.), so that the largest possible incarnation address space, occupying most of a node's memory (4 Mbytes), can be copied in just over a second. In the light of this, the advantages of copy-on-reference and iterative pre-copying were not felt to justify the implementation work entailed in this context (the 68030 processor includes memory management facilities capable of supporting both). Pre-copying the unmodifiable text segment is, however, a mechanism for reducing freeze time which is obtained at little implementational cost.

7.4.4 The Migration Algorithm

A migration involves the following actions (Figure 7.1, overleaf):

[M1] In response to the migration request from M, S sends a nested request to D. The request includes the RDC identifier and the memory resources required by the incarnation. If D is able to host the incarnation, it commits the resources and replies affirmatively. Whatever the reply, S echoes it to M. The incarnation continues execution at S.

[M2] At this point, the incarnation is locked against increasing the size of its main components: heap, stack or pincdata, since otherwise these would exceed the resources allocated at D. If it attempts to grow one of these segments, it is frozen. (It would be possible to request of D that it also increase the size of the corresponding segment, but this has not been further examined or implemented).

After resources are committed, D executes two concurrent series of actions, M3 and M4:

[M3] In the first, the pincdata is fetched from S. This is examined for port references. Appropriate message queue structures are created, and routing entries are created in anticipation of messages being forwarded from S. At S, the pincdata segment is locked against changes at the point that it is fetched, so that D continues to have a consistent copy. Changes to the pincdata are unlikely to be required before migration is complete; if they are, the incarnation is blocked. Operations and notifications are redirected to D from the time that the pincdata has been transferred. They are handled there unless data is required which has not yet been fetched from S, in which case they are made pending.

[M4] The other series of actions performed concurrently by D is the fetching of the incarnation's text, data, stack and register state (in that order). When the heap segment copying has begun, the incarnation's execution is frozen, to avoid inconsistencies between the copies of the volatile data segments at S and D. Once the incarnation execution is frozen, S is able to perform the algorithms for moving its message queues. Any messages which have been received but not replied to must be moved first, before the incarnation commences execution at D (in case it attempts a reply).

	M	S	D	S	D	
[M1]..	*inc_migrate*	∅ request mig ∅	memory resources committed			
	(*returns*)	♦ echo reply ♦				
[M2]..		segments locked against growth				
[M3]..		pincdata locked against change	♦ fetch pincdata		♦ fetch text	*..[M4]*
time		divert operations and notifications to D	create message queues and address entries	freeze incarnation	♦ fetch heap, stack and register state	
			handle/stall operations and notifications	forward and re-route messages		
					execute	

Figure 7.1: Actions Performed to Achieve Incarnation Migration.

As it stands, migration is not atomic in the face of failure. It is possible for D to fail after an operation has been performed on the incarnation at D, without any knowledge of this at S. In addition, copies of messages forwarded from S are not currently kept. To achieve atomicity, it would be necessary to make all operations and notifications pending and to retain copies of forwarded messages at S, until confirmation of the incarnation's re-creation at D.

7.4.5 Migration Performance

A migration facility has to perform well if it is to be useful for load balancing and for responding to node withdrawal. This section reports measurements taken of the total time to migrate an incarnation, from the point at which the destination node is committed to the migration, until the incarnation re-commences execution at this node. This is an upper bound for the incarnation's freeze time. The details of the measurements are described in Appendix A, Section A.2.

It was found that the elapsed time taken to migrate an incarnation is given approximately by the following relationship:

migration time = $19 + 1.2p + 0.39s$ milliseconds, where p is the number of its ports, and s is its total address space size, in kilobytes.

It takes about 59 milliseconds to migrate a 100 Kbyte incarnation with one port, and about 420 milliseconds to migrate a 1 Mbyte incarnation with one port.

These can be compared with the following times which are extrapolated from figures given for other migration implementations[2]:

Charlotte (implemented on VAX-11/750 computers): $45 + 12.2s$
100 Kbyte process: 670 ms; 1 Mbyte process: 6300 ms.

Sprite [SPRITE] (implemented on Sun-3 workstations): $190 + 3.6s$
100 Kbyte process: 550 ms; 1 Mbyte process: 3880 ms

V [V85] (implemented on Sun-2 workstations): $32 + 3s$
100 Kbyte process: 330 ms; 1 Mbyte process: 3100 ms.

These comparisons should be treated with caution, because of significant differences in the performance of the computers and networks used, and because of the different mechanisms employed to perform migration. In particular, the time taken to migrate a process in any implementation is mainly a function of the process's address space size. The networks used in the other implementations have a bandwidth of the order of 10 megabits per second. The VME bus in the Equus implementation can sustain a throughput of about 24 megabits per second when data is copied in four-byte units in a tight loop between two nodes, with no other bus activity at the time.

The Sprite figure assumes that all pages of a process are transferred upon migration. Transferring a page involves writing it to disc from the source node, and demand-paging it from the destination node. In practice, only dirty pages are written to disc, and some pages may not be required at the destination node. The Sun-3 workstation is of comparable performance to the 68030 nodes; a Sun-2 is about half as fast and a VAX-11/750 about a quarter as fast.

If allowance is made for network bandwidth as probably the main determinant of the coefficient of s in the above formulae, and if allowance is made for processor speed as probably the main hardware determinant of the constants in the formulae,

[2] A similar comparison was made by the designers of the Charlotte migration facility [CHARLOTTE89]. Some of their extrapolated figures do not seem to be borne out by the information sources they quote. In particular, they estimate a migration time of $80 + 6s$. for the V system.

then the Equus migration facility appears to be of superior or comparable performance to these other implementations. The absolute migration times make incarnation migration a very good basis for load balancing and for reaction to node withdrawal compared to the other implementations, which practice both of these uses for migration.

7.4.6 Comments on the Migration Implementation

Since the kernel does not provide virtual memory or swapping, migration is not always possible, for lack of memory resources. In the case of load balancing, it is not anyway disastrous if a migration cannot be carried out for this reason. In the case of node withdrawal, the problem is more serious.

In retrospect, a facility to swap incarnations to disc should have been provided, so that an incarnation can always be moved away from a node, even if this in fact means swapping to disc until sufficient memory becomes available at the destination node. A separate version of *inc_migrate* would be necessary, which caused the child to migrate away from the current host even when insufficient memory is available. Where swapping was necessary, the destination D would have to have sufficient resources to receive the pincdata (containing ports information), and in the meantime provide queuing for messages sent to the incarnation. This and the necessary disc transfers involved could be implemented using existing mechanisms. The proto-incarnation at D could be migrated to another node which had sufficient memory resources for the full incarnation before D has them.

The restriction of not being able to migrate an incarnation until it has completed any synchronous invocation is also unnecessary. A receiving kernel keeps a copy of reply data, in case of a retransmission of the invocation message by the sender when it has timed out awaiting a reply. The problem is that this data is kept, currently, for a fixed time, which can be exceeded by the time taken to migrate. So if the incarnation's invocation were to be prematurely terminated at S and retried at D, the receiving kernel R might decide erroneously that the repeated invocation message was a new one. To interrupt synchronous invocations, it would therefore be necessary to cause a copy of any reply generated as a result of the invocation to be locked at R. After a retry was made, the reply copy could return to its normal status.

7.5 Summary

This chapter has described the facilities implemented by the kernel for the control of RDCs and of individual incarnations. Their rationale includes the need to terminate RDCs, to control node allocation to them, to perform load balancing upon incarnations and to monitor and control their execution state. It has described how control relationships are themselves reconfigurable. A dynamically variable set of incarnations can be parents to a given child, so that a backup can be provided in the case of parent failure, and so that a pool manager incarnation or debugger, for example, can be introduced at run time to exercise control over a child.

The chapter has described how incarnation migration was implemented, and discussed the operational features of this in relation to its intended use. Migration performance has been shown to be favourable compared to other implementations. The chapter concluded that kernel facilities should be modified so that migration away from a node is always possible in bounded time.

Chapter 8
The Kernel

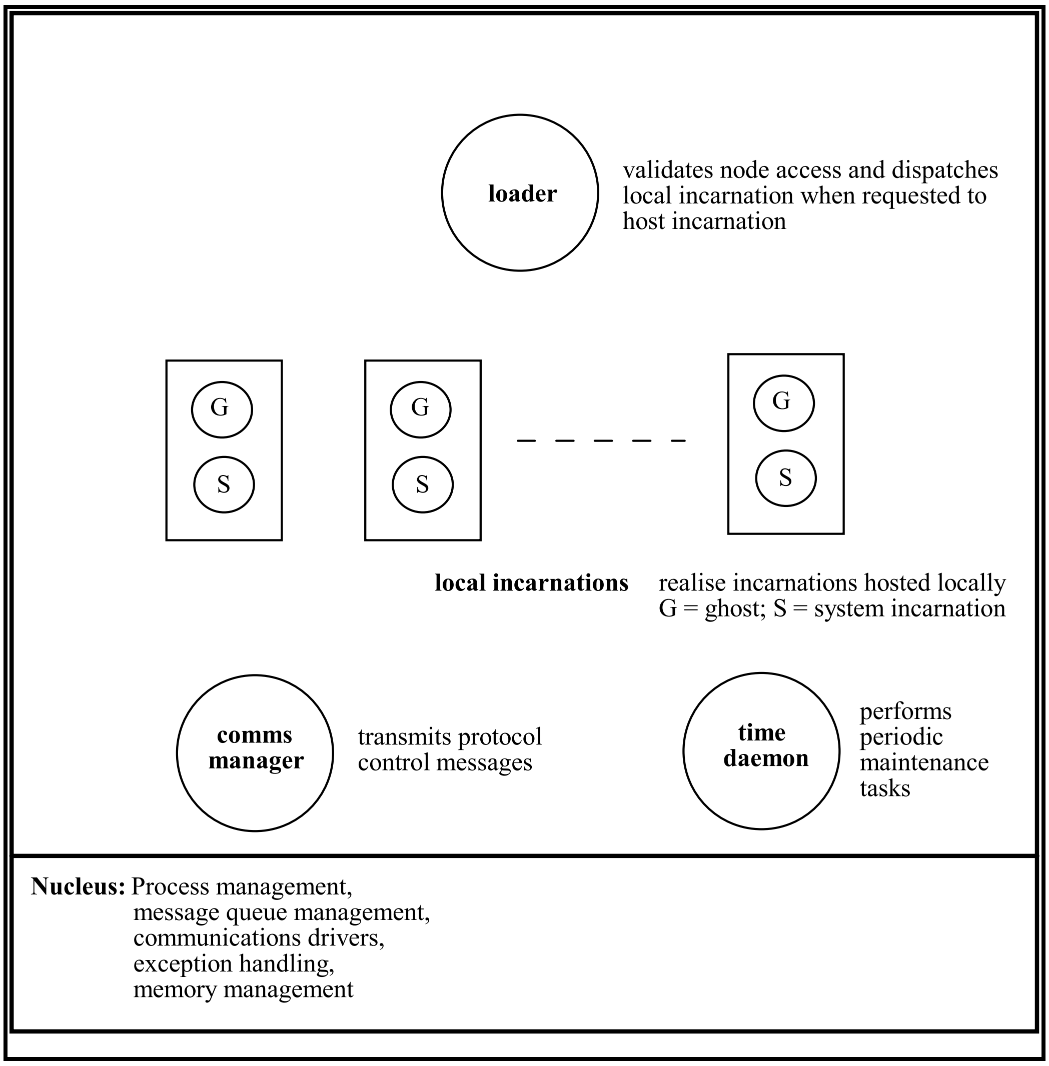
This chapter outlines the design of the Equus kernel. It describes its component processes and their inter-relationships, and in particular the use of so-called ghost processes to achieve concurrency and modularity in the kernel's operations. The communications implementation is outlined, and a description is given of the algorithms used to locate destinations and re-route messages. The chapter ends with some performance figures measured for invocation primitives and communications reconfiguration primitives.

8.1 Kernel Architecture

The kernel is written largely in C, with a small amount of MC68030 assembler. Its total program size is 186 Kbytes, of which 109 Kbytes is text and 77 Kbytes initialised and uninitialised data (not including kernel stacks, message buffers and memory mapping tables, which are obtained dynamically). It consists of processes supported by a small nucleus of mainly low-level code (Figure 8.1: the ghosts and system incarnations are described in Section 8.2, and the others in Section 8.4). The nucleus provides: local process management (for example, creation, scheduling, stack management); message queue management; communications between local processes; packet-level network drivers for external communications; protocol stacks; hardware exception handling and memory management.

The main actions which in general involve interactions between processes residing at more than one node are the following:

- incarnation creation and migration, and other operations upon incarnations;
- invocations and data copying;



loader
validates node access and dispatches local incarnation when requested to host incarnation
G
S
G
S
G
S
local incarnations
realise incarnations hosted locally G = ghost; S = system incarnation
comms manager
transmits protocol control messages
time daemon
performs periodic maintenance tasks
Nucleus: Process management, message queue management, communications drivers, exception handling, memory management

Figure 8.1: Main Kernel Components.

- locating incarnations and ports;
- propagating and re-attaching ports and rebinding streams;
- event propagation;
- incarnation termination and constraint of RDCs' access to nodes.

All of these actions can take place within a single node, but no fundamental distinction is made in the design between this and the inter-node case. They are all achieved by passing messages between processes.

Of the processes, the loader, communications manager and time daemon are constantly present, but the other processes are used to realise incarnations, and vary in number as the kernel hosts them and as they are migrated away or destroyed. Processes run in overlapping protected and mapped virtual address spaces which are backed by pages resident in local physical memory and (for the purposes of message passing over a VME bus) pages resident in the memory of other nodes but directly accessible there. The processes execute kernel code in supervisor mode (MC68030 master and interrupt modes [MC68030]), except when the system incarnations execute user-level incarnation code – which is their primary function. All processes have separate master mode stacks for local variables, and share kernel text and heap segments. Where interactions are necessarily local (for example, in buffer acquisition), processes share tables and variables, using low-level synchronization primitives to co-ordinate access.

8.2 Local Incarnations

Every incarnation executes at a host node as two processes, together referred to as a local incarnation (Figure 8.2). These processes are:

- A system incarnation. This executes the user-level incarnation code, except that it executes kernel code when the incarnation is being established at the node, and when it processes exceptions, such as system call traps.
- A ghost process.

8.2.1 Ghost Processes

Ghost processes:

1] handle all operations upon the incarnation, including control operations, stream and port operations and reads and writes of segments of the incarnation's address space. Reads and writes occur as part of incarnation creation or migration, or as part of an asynchronous message receipt or use of *inc_copyto/from*.

2] handle notifications of events sent by other incarnations, and handle events detected locally – causing software interrupt processing if necessary.

3] detect local events of interest to other incarnations and send notifications.

4] handle status probes from other incarnations.

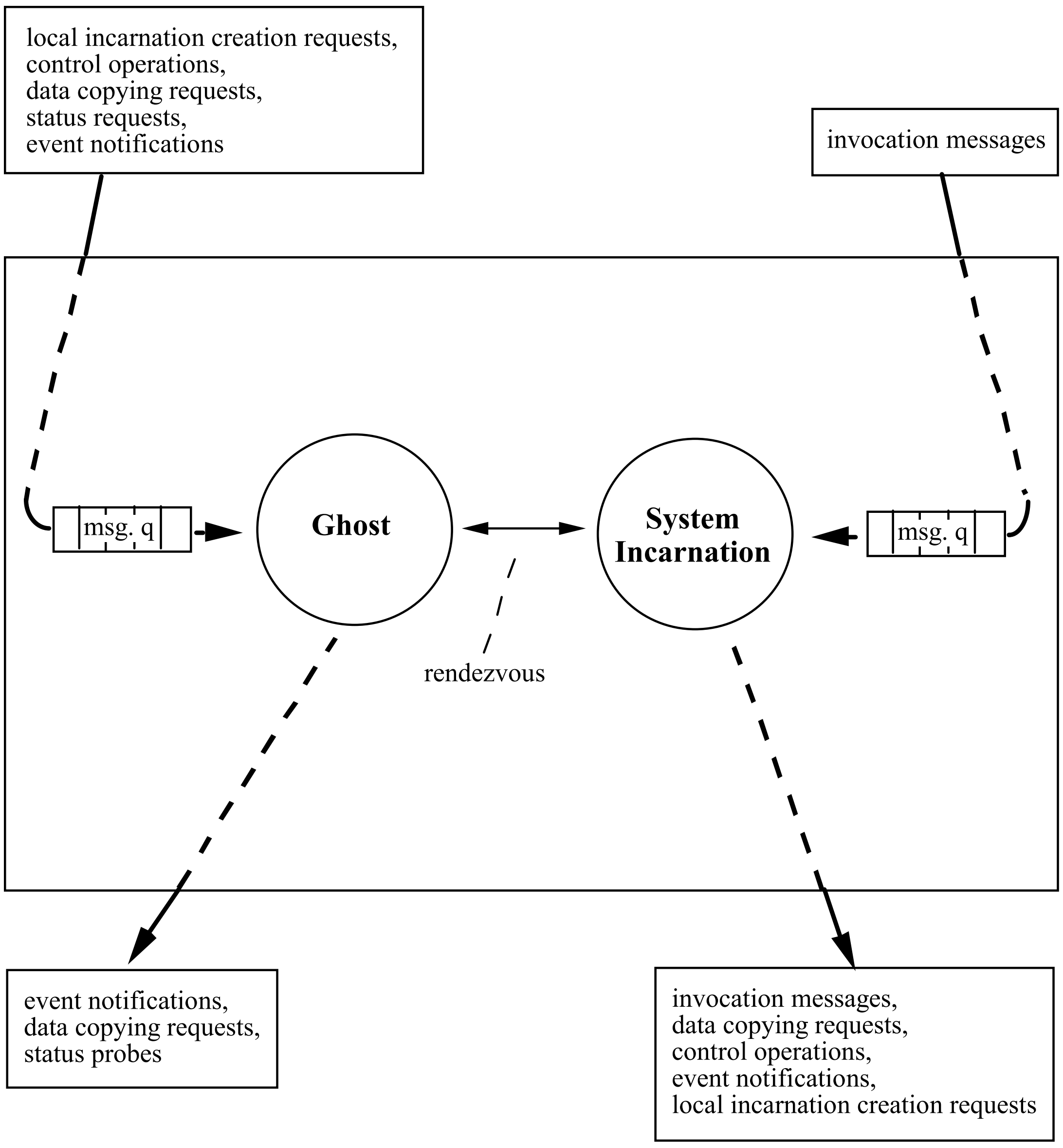


Figure 8.2: Local Incarnations and Their Interactions.

5] generate periodic status probes when a parent has made the incarnation dependent upon it.

Ghost processes were introduced into the kernel architecture because of:

- the concurrency between the operation and event processing of 1-5 above and the execution of the incarnations' user-level code;
- and because of the independence of, and therefore concurrency possible between, much of this processing as carried out for different incarnations.

Ghosts do not have to synchronize with one another except during migration and when one ghost handles the other's status probe. Ghosts do require their corresponding system incarnations to synchronize with them in certain circumstances. Although the synchronization points had to be programmed with care to avoid deadlock, they are few. Per-incarnation ghosts simplify the kernel programming by eliminating the multiplexing of operations between incarnations that would otherwise be necessary, and by allowing state to be maintained in local variables. The concurrency afforded by ghosts maximises potential processor utilisation. When one ghost or system incarnation blocks, either its counterpart in the same local incarnation or one associated with a different incarnation can still run.

8.2.2 Other Features of Local Incarnations

Dispatching a Local Incarnation

A pool of local incarnations is managed by the kernel. Each member of the pool has its two processes and associated data structures already created. When a request is made to host an incarnation – either a new incarnation or one migrated from another node – the loader either dispatches a local incarnation from this pool, if one is available, or creates a new one to handle this. The ghost and system incarnation are engaged concurrently, the system incarnation in loading program text and data, and the ghost in fetching and processing the pincdata (per-incarnation data, introduced in Section 7.4.1). This is the realisation of the concurrency described as part of migration in the last chapter.

The Pincdata

The need to copy and migrate incarnations led to the collection in the pincdata data structure of all intrinsic data associated with a particular incarnation's operations, so that this can be simply copied to another node. The pincdata is memory-mapped to a fixed location. This is firstly so that its internal pointers used in linked list structures are still valid when copied to a new node (where it is mapped to the same address). Secondly, the data can be extended in size continuously under this scheme, as references and other data items are added.

Incarnation Address Space

Ghosts and system incarnations share the same per-incarnation MC68030 supervisor-mode address space (Figure 8.3). Each address space includes the common kernel text and heap segments. This means that the loader, communications manager and time daemon can execute in any context. And the

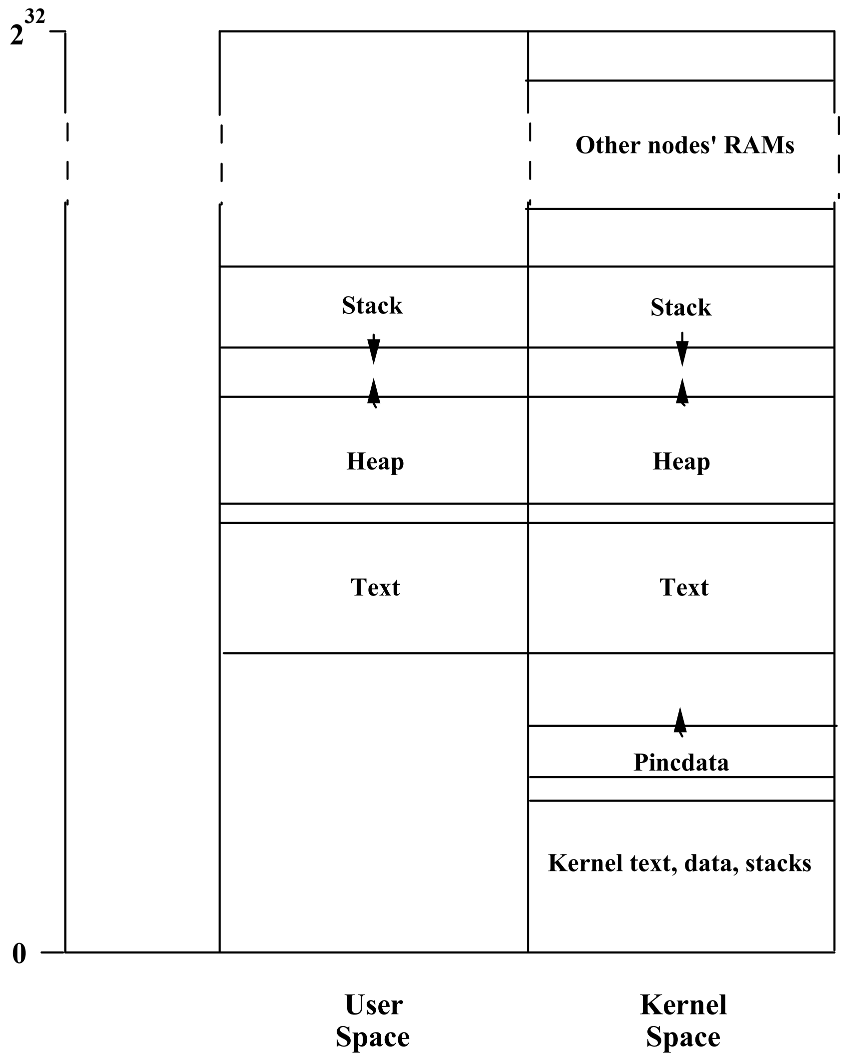


Figure 8.3: Incarnation Address Space.

system incarnation and ghost of a local incarnation operate in the supervisor and user counterparts of the same MC68030 memory management context, so that memory management switches are not required between them. The address mappings of the user-level incarnation stack, heap and text are identical between user and supervisor modes. This eliminates the need for address translation between kernel and user mode for the same incarnation when system calls are processed. It also allows a buggy kernel to overwrite incarnation address spaces; but the kernel is sufficiently robust for such bugs not to have been encountered.

Each processor card occupies a unique range of addresses accessible over the VME bus. The memories of the other processor cards are mapped identically into each supervisor context. This is in order that message-passing can be implemented using memory-to-memory copying. The MC68030's transparent translation registers [MC68030] are used to map the large address ranges involved; they do so at little operational expense, and their use avoids the need for corresponding memory mapping tables in local memory.

8.3 Other System Processes

The Loader

This process controls access to the node and monitors its activities. There are two categories of requests processed by it. Belonging to the first category are requests concerned with overall node management in the context of the pool. These are: to enable incarnations belonging to a given RDC to be hosted at the node; to withdraw this right and optionally terminate any current incarnations belonging to it there; and to report the status of the node, in terms of processor loading, memory utilisation and numbers of incarnations hosted.

Belonging to the second category are requests that a particular incarnation be hosted at the node. A host request to loader contains:

- the incarnation memory requirements;
- a bit string identifying the associated module text, so that this can be shared if an incarnation associated with the same module is already present at the node;
- the RDC's unique identifier, which the loader looks up against its access list.

Communications Manager

The communications manager is a utility process which sends responses such as acknowledgements for asynchronous messages to processes in other nodes. For reasons of efficiency, an interrupt handler rather than a separate process is used to do the majority of the processing associated with attempting to deliver incoming messages. Using a process would entail context-switching overheads additional to those of the handler itself. However, if a response message is required, then the interrupt handler cannot send it. If it did, this could cause it to wait for a low-level condition, which would deschedule the unrelated process that happened to be executing at the time the interrupt occurred. Therefore the handler generates the response message but passes it to the communications manager for transmission.

Time Daemon

The time daemon runs periodic tasks. In particular, it garbage-collects reply message data buffers after these are found to have aged more than an amount which is dependent upon the retry time used in the transmission protocol. It also periodically checks the integrity of kernel data structures. In particular, it checks that kernel stacks have not overflowed.

8.4 Communications

8.4.1 Delivery Mechanisms

The communications subsystem of the kernel consists of a few basic mechanisms for the delivery of single-packet kernel messages and for the reliable transport of bulk data. They are used directly for message passing between kernel processes, and they are used to implement invocations between incarnations and communications reconfigurations.

A kernel message consists of:

- a delivery header, containing the kernel message's address and other data used by the delivery mechanisms;
- a fixed-size header set by the sending process which identifies the function of the message to the recipient process and provides parameters (the entire data for many kernel-kernel requests and replies fit into this header);
- optional extra data – as much as fits into a packet.

Each process possesses one or more data structures used to control the enqueuing and reception of kernel messages, called kernel ports. In the case of system incarnations, these are used to implement incarnation ports: each incarnation port is associated with a kernel port.

There are three types of delivery mechanism available to any sending process: request-reply, asynchronous send and bulk data transfer. Each is based upon an unreliable packet transmission service over the underlying network.

1] In a request-reply interaction, a process sends a single kernel message and receives back a kernel message. One or more processes receive the request, and any can send a reply. One of these replies is returned to the sender by default (the first to arrive back), but it can obtain others. This service attempts to be reliable in the face of packet loss or corruption: at regular intervals the request packet is re-sent in case previous packets were dropped. If requested, the delivery service ensures that at most one copy of the request is delivered to any recipient, despite repeats. When a kernel message has been queued but not received it is said to be *arrived*. All arrived messages are held together on a per-process queue, but distinguished by their destination kernel port address. The use of a single queue facilitates the movement of all arrived messages when an incarnation migrates. When a kernel message has been received but not replied to it is said to be *attached*, and is placed on another per-process queue. When it has been replied to, a copy of the reply data is held on a per-process reply queue, so that the reply can be repeated if a retry is received. Only the last, unacknowledged packet of reply data is retained. Keeping a copy of this reply packet saves an extra acknowledgement message which would otherwise have to be added to the reply protocol.

2] A kernel message which is sent asynchronously is acknowledged immediately by any receiving kernel which possesses a matching destination kernel port; no reply data are returned to the sender. The asynchronous delivery service retries a stipulated number of times when no acknowledgement is received.

3] Finally, the bulk transfer service is used to transport reliably blocks of data which are larger than can fit in a single kernel message. This service splits the data to be sent into separate packets sent out over the network, which are reassembled in the correct order at the destination. To keep protocols simple, it is design policy for the endpoints of a bulk transfer not to be moved (as a result of a reconfiguration) whilst it takes place.

A synchronous invocation is realised by a request-reply interaction; extra data is transferred between receiving and replying to the kernel message which is the header of the invocation message – whilst the invoker is blocked. Asynchronous invocations are realised by sending the header (or entirety, in the case of *inc_send*) of the invocation message using the asynchronous kernel message delivery service. If there is extra data belonging to the invocation message to be transmitted, then this is obtained from the sender's ghost process, using the bulk transmission service. *inc_copyto* and *inc_copyfrom* also achieve data copying by bulk transmission to and from the corresponding ghost.

Operations are directed either to a port belonging to the loader, or to a port belonging to the ghost process of the incarnation concerned; event notifications are directed to ghost ports. Each such ghost port is managed as if it were an incarnation port attached to a channel, so that it can be reconfigured as an incarnation port when migration takes place.

8.4.2 Addressing

A kernel message is addressed with one of two types of address:

A kernel port address
This is a kernel port identifier: an integer which can be rapidly decoded into a kernel node identifier and the memory location of the port data structure within it;

A logical address
This consists of:

i) a node's kernel identifier or a flag denoting broadcast;

ii) a global identifier (a bit string which is constructed to be globally unique over a reasonable time period);

iii) a type value which qualifies the global identifier.

Kernel port addresses are used alone when the destination kernel port is known and is known to be fixed. For example, bulk data transmissions take place under these circumstances. Logical addresses are used to reach and locate kernel ports using logical destinations.

At each node there is a database of global identifiers. A record in the database consists of an identifier such as is used in logical address field ii), a type value such as is used in address field iii), and the identifier of a kernel port. There can be more than one record for a given global identifier. The values of the global identifiers and types are not interpreted by the delivery system, but are used to match given key values in logical addresses. The database and logical addresses are used in conjunction as follows:

1] A kernel message normally carries a kernel port address. This is decoded to reach the node whose kernel identifier appears in it.

2] To reach a process using a global identifier at a particular node – such as the loader at a given node – logical addresses are used with part i) containing the node's kernel identifier, ii) the global identifier and iii) the type. The type is

DLVR_NODE_SERVICE in the case of the loader (a well-known global identifier is used to identify the loader). The type DLVR_GHOST is used in the case that ghosts are to be reached which have a kernel port registered in the database at that node. Only entries which match in type as well as identifier are used for delivery. For example, a control message concerning a channel can be delivered to all ghosts interested in such messages, without its being delivered to incarnation ports attached to the channel at the node. The latter are registered with type DLVR_CHANNEL.

3] A multicast invocation message is broadcast to all nodes. It carries a logical address in which the global identifier is that of the channel involved. Nodes deliver the message to any kernel ports with database entries under this identifier with the type value DLVR_CHANNEL, and drop the message otherwise. (The kernel could support multicasts with other global identifer types, but these are not used).

8.4.3 Locating a Kernel Port

It is sometimes necessary to locate the appropriate destination kernel port when a unicast kernel message is to be sent. Either no kernel port address is known, or the kernel port is invalidated as a destination after certain types of reconfiguration have taken place (this is described in Section 8.5).

If the only address known for a unicast kernel message is a logical address, a destination kernel port is located and the message is delivered by the same broadcast. If a node possesses a database entry of matching global identifier and type, it delivers the kernel message locally. The kernel port identifier is automatically returned to the sender by the delivery system in any acknowledgement or reply. The sender is then able to use this kernel port address in subsequent transmissions.

If the kernel port is one used to implement an incarnation port or is a ghost port, however, then kernel messages sent to this kernel port address may later be subject to redirection as the result of a communications reconfiguration or incarnation migration. The old and new destination kernels are co-ordinated so that: i) The new site is given a database entry with the appropriate global identifier and type DLVR_CHANNEL. ii) The database entry is removed from the original destination kernel. iii) Redirection information is placed between them. Redirection entries are looked up by a kernel port identifier. Redirection entries are made according to the reconfiguration protocol described in the next section.

The delivery system discovers that redirection is required when it attempts to deliver a unicast message at a kernel port which proves not to exist. The kernel messages are forwarded as datagrams: no acknowledgement or reply is required (if any) until they have been queued or received, depending on the transmission semantics. Whatever the means by which the kernel port is located, any acknowledgement or reply message carries the final address with it, so the sender can update its version.

The delivery system can follow a chain of redirection entries if necessary. However, the oldest redirection entries are discarded when the kernel is running out of table space, and redirection entries are useless when the node they reside at fails or becomes unreachable. When no more redirection entries can be found or none can be reached, the delivery system at the sender's node is either informed of this by the terminal node of the chain, or times out through lack of any response. It then resorts to broadcasting for an entry with the global identifier of the logical address, with type DLVR_CHANNEL (which is used with all ports subject to redirection).

8.4.4 Multicasting

When a multicast is made, every node incurs an overhead, even if no destination kernel ports are located there. But the alternative, as has been pointed out in chapter 4, is for the locations of multicast address bindings to be known and the bindings updated whenever incarnation ports are destroyed and propagated. The overhead of deciding that an incoming message is unwanted is mostly that due to accepting the packet (around 0.5 milliseconds), and a search of the database – which can be estimated at up to the order of 0.1 milliseconds for the order of a hundred entries on a 2 - 3 MIPS MC68030. The overhead could be eliminated from the main processor by employing a co-processor dedicated to handling communications at every node. A co-processor would also be an advantage in a distributed system with a topology which requires routing, to offload this function from the nodes' main CPUs.

The VME-based development computer system does not have hardware support for broadcast or multicast, and point-to-point packet delivery is used instead. The cost of this is acceptable because of the small number of nodes involved, and because the implementation has been carried out such that the messages are transmitted concurrently.

8.5 Changing the Receiver

The following algorithms were devised for the reconfiguration mechanisms described in Chapter 6, and for communications reconfigurations consequent upon incarnation migration. Their aim is:

1] to forward kernel messages already queued at a port;

2] to cause subsequent messages to be given the new, correct kernel port address at source, and to re-route them if necessary before this has been achieved;

3] to ensure that delivery guarantees of reliability are maintained despite reconfigurations;

4] to ensure that kernel messages sent by the same process are received at a port in the same order as they were sent (when delivery is reliable), despite reconfigurations[3].

5] to prevent unintended gaps in the sequence of messages received at a port affected by a reconfiguration, by ensuring that the forwarding of messages from a port involved in a reconfiguration is not interleaved with reception from the port.

8.5.1 Port Propagation

When a port attached to one or more channels is propagated, an incarnation *receiver* receives a kernel message containing:

- the global identifiers of the channels concerned;
- kernel port and logical addresses for the ghost port of the incarnation *sender* from which it is being propagated.

In lines 3-8 of Figure 8.4, *receiver* creates a port and, for each channel, associates an attachment with it. Each attachment to the port is realised by a separate kernel port. Although it is not shown in the figure, the kernel ports are made proxies for a single, main kernel port: a kernel message addressed to a proxy port is delivered to the main kernel port it is a proxy for. This arrangement allows messages arising

[3] When a port is propagated but the message queue is retained at the original port, it is possible for invocation messages to be *received* out of order, at different ports. But it is the application's responsibility not to perform such a reconfiguration if it violates consistency.

receiver:

```
1    handleProp(propRqst: propagation_rqst)
2    begin
3        p = portCreate()
4        ∀ c ∈ propRqst.channels
5        do
6            k = kernelPortCreate()
7            portMarkAttach(p, c, k)
8        od
9        PropActivate(propRqst.ghost, propRqst.portid)
10       ∀ a ∈ p.attachments
11       do
12           dbaseEnter(a.kernelPort, a.chnl, DLVR_CHANNEL)
13       od
14   end handleProp
```

Figure 8.4: The Algorithm Followed by a Recipient of a Propagated Port.

from multiple attachments to be received at a single port, which requires simpler and cheaper logic than reception from multiple kernel ports. And at the same time it allows control over routing messages at the level of an individual channel.

Receiver then requests that *sender* activate the reconfiguration and blocks until this has been done (line 9). Thereafter it remains only to enter the database entries for the channels (lines 10-13).

Figure 8.5 shows the algorithm followed by *sender*. If the incarnation port was not marked for retention on propagation, the current message queue is forwarded (lines 6-8). Messages which have already received acknowledgements are forwarded reliably. A single destination kernel port address is shown for the sake of clarity, but in fact each message is forwarded to the proxy kernel port set up by *receiver* to correspond with the message's channel. The identifier of this proxy port will then be carried in any acknowledgement, causing the sender to use the proxy port's identifier as the address in subsequent messages.

sender:

```
handlePropActivate(propActRqst: activate_prop_rqst)
begin
    p = portLookup(propActRqst.portid)
    k1 = p.mainKernelPort
    k2 = propActRqst.destKernelPort
    if(not p.retained)
        forwardMsgs(k1, k2)
    fi
    reply
    ∀ a ∈ p.attachments
    do
        dbaseDelete(a.kernelPort, a.chnl, DLVR_CHANNEL)
        forwardPendingMsgs(a.kernelPort, k2)
        redirectionEnter(a.kernelPort, k2)
        kernelPortDelete(a.kernelPort)
        portMarkDetach(p, a)
    od
    if(not p.retained)
        portDelete(p)
    fi
end handlePropActivate
```

Figure 8.5: Activating Propagation on Request from Recipient.

The routine *forwardMessages* allows new messages to arrive whilst it is forwarding. Before forwarding any messages, it marks the kernel port as "moving". This flag causes the delivery system to make arriving unicast kernel messages pending: it places them on the port's arrive queue, marks them as pending, but does not acknowledge asynchronously sent kernel messages. Multicast messages are dropped when the kernel port they would otherwise be delivered to is moving. The alternative to queuing pending messages would be to drop them but send an acknowledgement which causes senders to retry later. But estimating a retry time is difficult, and can result in senders being rebuffed the next time they retry, or blocked longer than is necessary.

If *forwardMessages* encounters a pending message, it finishes. These are forwarded later, as will be described. This forwarding routine will terminate, because making arriving messages pending bounds the number of messages to be forwarded by the number of senders. Without the use of pending messages, a zealous sender of asynchronous messages could keep the forwarding routine indefinitely occupied.

After forwarding messages, *sender* then replies to and thus unblocks *receiver* (line 9), which proceeds immediately to set up DLVR_CHANNEL database entries. For each channel, *sender* forwards pending messages[4] just before setting the redirection entry (lines 13-14). When the kernel ports are deleted (line 15), the redirection entries become used by the delivery system to forward arriving messages. The attachments are unmade from the port data structure (line 16). The port itself is deleted if it was not retained on propagation (lines 18-20).

Although multicast messages are dropped whilst the main kernel port is marked moving, subsequent retries will find an entry at *receiver*'s node, unless they all find the same state of a new reconfiguration. This situation has been left as possible but of neglible probability. It would lead the transmitter of the messages to give up.

8.5.2 Port Attachment

In this case, a manager causes a port belonging to incarnation *attach_inc* to become attached to a channel (assumed unicast), and a port belonging to an incarnation *detach_inc* to become detached from it. The manager operates upon *attach_inc*, which responds by creating a new kernel port and attachment entry for its incarnation port, and then operating upon *detach_inc* to activate the detachment of its port and the forwarding of any arrived messages. The algorithms employed by *attach_inc* and *detach_inc* follow those for port propagation between them, except that:

- only one channel is involved, and only messages which have arrived over this channel are forwarded from *detach_inc*'s port;
- and the incarnation port has to be locked against reception whilst forwarding takes place, to guard against reception by *detach_inc* being interleaved with the forwarding of arrived messages. Interleaving would cause gaps in the sequence of messages arriving at the newly-attached port.

4 If the port is retained, there is the option to detach it when propagation is initiated, so that no more invocation messages are queued at the port. This is achieved by marking the main kernel port as moving. If this option was selected, there may be pending messages at the kernel port at this point.

8.5.3 Rebinding a Stream

In this case, a manager rebinds a stream belonging to incarnation *sender*. Messages sent using the stream are currently delivered to a port belonging to an incarnation *current_receiver*. If the stream is to be rebound without affecting messages already sent but not received – for example, under failure conditions – then the manager itself rebinds the stream. Otherwise, messages sent from the stream and queued at *current_receiver*'s port are first de-queued and forwarded over the stream's new channel. The manager requests *current_receiver* to perform this operation; *current_receiver* is then able to co-ordinate message re-routing.

Figure 8.6 shows the algorithm used by *current_receiver*'s ghost in performing stream rebinding. Only messages sent using the given source stream are forwarded (line 9). The incarnation port is locked against receives before forwarding takes place and unlocked afterwards (lines 8 and 12). The forwarding routine sets the delivery system to make pending any kernel messages sent using the stream, after it has forwarded existing arrived messages. *Current_receiver* operates upon *sender* to rebind the stream at line 10, and when this operation returns, it forwards any message which arrived in the meantime (there can be at most one). In handling the rebind operation from *current_receiver*, *sender*'s ghost only has to set the new channel and destination kernel port values. Any invocation in progress is not affected by this rebinding operation itself. There is no need, since the invocation message will be routed correctly when pending messages are forwarded (line 11), if it has not been already by the forwarding routine.

current_receiver:

```
handleRebindAtCurrentReceiver(rebRqst: rebind_at_curr_receiver_rqst)
begin
    p = portLookup(rebRqst.portid)
    k1 = p.mainKernelPort
    s = rebRqst.streamToRebind
    c = rebRqst.newChannel
    k2 = rebRqst.newKernPort
    portLockReceives(p)
    forwardArrivedMsgsByStream(k1, k2, s)
    rebindAtIncWithStream(rebRqst.GhostWithStream, s, c, k2)
    forwardPendingMsgByStream(k1, k2, s)
    portUnlockReceives(p)
    reply
end handleRebindAtCurrentReceiver
```

Figure 8.6: Handling a Rebind Operation at the Current Receiver.

The destination kernel port used – *k2* – is a value held in some stream attached to the appropriate channel, and it could be out of date. But the forwarding procedure will still locate the correct kernel port destination, using the same location algorithm as is used for an invocation.

8.5.4 Migration

When an incarnation migrates, the source kernel reliably delivers all the kernel messages which have been received but not replied to. It then forwards all arrived kernel messages and places redirection entries as for port propagation. It notifies the destination ghost when forwarding is complete, whereupon the latter is able to make database entries, as for port propagation.

8.6 Performance

This section presents figures for the implementation's performance for invocations and the mechanisms to change the receiver of invocations. The figures are given as a guide only, to judge gross system performance. A full description of each of the experiments made to obtain these measurements is given in Appendix A.

System	Processor	Call Time (milliseconds)
Cedar [RPC]	Dorado – custom	1.1
Amoeba [AMOEBA89]	MC68020	1.4
V [V88]	MC68020	2.5
Sprite [SPRITE]	MC68020	2.8
Equus	MC68030	2.9
Firefly [FIREFLYRPC]	MicroVAX-II	4.8
Amoeba/Unix [AMOEBA89]	MC68020	7.0

Table 8.1: Performance of Request-reply Primitives.

Invocations

Table 8.1 gives figures comparing the performance of the *inc_invoke* primitive with comparable calls in other implementations. These are all request-reply calls which transmit a minimal amount of user data (0 - 32 bytes) in each direction. In all cases, one sending process or thread of control sends to one receiving process or thread of control on a different node.

The implementations used for comparison all employ local area networks whose bandwidth is an order of magnitude less than the VME bus used in the Equus implementation. It is not surprising that the bulk data transfer rates achieved for the Equus implementation are considerably higher, and these figures are not given here but appear in Appendix A, Section A.1.

The figures for other implementations also appear in a comparison made by implementors of the Firefly RPC system [FIREFLYRPC].

These figures constitute only a rough guide for comparison, because of variations between the implementations in call semantics, in computer systems and in whether the calls are made from user space to user space, or from kernel space to kernel space. Nonetheless, it can be seen from Table 8.1 that the Equus implementation gives a respectable performance when measured against these other systems. Moreover, lack of time has meant that little effort has been put into optimising the

Equus implementation, whereas considerable optimisation effort has been reported for Amoeba, V and Firefly RPC.

Changing the Receiver

Table 8.2 shows the results of measurements taken to gauge the overhead to a client of a reconfiguration to change the server peer which processes its requests. The client uses *inc_send* to send 10,000 "requests" asynchronously in a simple loop without pausing. Each request consists only of the 32-byte message header. There are two server peers, which receive the requests but do not perform processing on them.

Measurements were taken of the total time taken by the client to complete all the *inc_send* calls, in four different situations:

1] Only one peer receives all the requests.

2] The peers alternate in receiving the requests. Each receives 100 requests and then propagates its port in a message to the other.

3] The peers are alternated by a separate incarnation which periodically re-attaches their ports to the request channel.

4] The peers are alternated by a separate incarnation which periodically rebinds the client's request stream.

In 3 and 4, 100 reconfigurations take place, with about 100 messages being received in between reconfigurations. In all reconfigurations, the queue of messages at the current peer's port is forwarded as a result of the reconfiguration. This is expected to be a worst case, since the sender has to be blocked whilst messages are forwarded.

The figures for the overhead are calculated from the total elapsed time by subtracting the corresponding figure for the case with no reconfigurations, and dividing by 100 (the number of reconfigurations). The number of messages forwarded refers to acknowledged messages; at most one pending message can also have been forwarded per reconfiguration (see Appendix A, Section A.3).

Experiment	Elapsed send time (milliseconds)	No. of reconfig-urations	No. of messages forwarded (both kernels)	Overhead per reconfig-uration (milliseconds)
no reconfig-urations	21900	0	0	0
port propagation	22900	100	201	10
port attachment	22710	100	84	8
stream rebinding	22640	100	77	7

Table 8.2: Overhead to Client Due to Communications Reconfigurations.

It is not surprising that port propagation turns out to be the most expensive form of reconfiguration of the three. In the time it takes to propagate the port in a user-level message (about 2 milliseconds), neither peer is receiving the requests. Arriving requests therefore accumulate at the port being propagated. When the propagated port is received, these queued messages (about 2 of them per reconfiguration) have to be forwarded before the reconfiguration is complete and new messages can start arriving at the new port.

In the cases of port attachment and stream rebinding, the reconfiguration operation is applied whilst the current peer still attempts to receive requests. Reception is in fact blocked whilst queued messages are forwarded, but the queue of acknowledged messages cannot grow during this interval. This is reflected in the smaller numbers of messages which the kernels forwarded.

It may be possible to improve the reconfiguration algorithms. Asynchronous invocation messages could be delivered to the new port and acknowledged – thus allowing senders to continue – at the same time as queued messages are forwarded between the old and new ports. To preserve invocation order, message reception at the new port would have to be inhibited until previously delivered messages had been forwarded. Nonetheless, at 3.5 - 5 times the cost of one of the invocation calls involved, the above figures for reconfiguration costs are encouraging. Further measurements are required, in which request processing is performed (or simulated) by the peers, and in which a variety of invocation primitives and amounts of extra data are used by the clients. The implementation was designed so that the existence of extra data in invocation messages would not significantly increase reconfiguration costs (Section 4.4), but this requires verification. These additional measurements were not made for lack of time.

8.7 Summary

This chapter has presented the design of the major components of the kernel which are specific to the RDC model. It has described the kernel's architecture in terms of the communicating processes between which its tasks are divided. The most important among these are the loader and the ghost processes. The loader controls access to the node by RDCs, and the ghost processes handle operations upon invocations and event notifications sent to them. There is one ghost process per incarnation at the node, so as to maximise the concurrency of the kernel's operations. Concurrency was required to simplify the programming of the kernel by eliminating explicit multiplexing (many of the tasks it performs can be processed independently for different incarnations), and to reduce the number of bottlenecks in its operation.

The communications subsystem of the kernel has been described, and the algorithms employed to change the destination of invocation messages have been presented. Performance figures have been given which show that invocation times are comparable with those found in other implementations. Figures have also been given for reconfigurations which change the destination of invocations. Further measurements are required, but these initial figures are promising.

Chapter 9
Conclusion

9.1 Summary of Contributions

This dissertation has presented a model for reconfigurable distributed computations called RDCs, and its implemented operating environment, Equus.

The model and its environment are motivated by the need to program distributed computations which can adapt to run-time conditions, such as processor availability and the load imposed upon servers by their clients.

The main argument of the dissertation is that the RDC model and Equus provide a practical and general framework for programming solutions to problems involving reconfiguration. This framework is realised in the following main contributions:

- A comprehensive set of mechanisms has been provided for establishing and changing a distributed computation's configuration.
- The programmer only has to declare a configuration, rather than be concerned with its fabrication. Reconfigurations can be made in the form of migration and in the form of changes to the incarnation population and interconnection structure. The two types of reconfiguration can be made independently.
- Using stream space, interconnectivity can be expanded through naming conventions. This eliminates the programming problem of having to use existing connectivity.

- The implementation has been divided between a small kernel with good performance, and more easily modifiable user-level code for configuration establishment, reference propagation (stream space), node allocation and load balancing.

- A synthesis of existing communications and reconfiguration paradigms has been made, so that reconfiguration management can be made transparent to a server incarnation connected to loosely coupled clients.

- An analysis has been given of the conditions which must be met before any communications reconfiguration can be applied, if application consistency is to be preserved.

- In the important case of servers which reconfigure in relation to loosely coupled clients, the design of the model enables the servers to reconfigure whilst preserving consistency, without the need for interaction with their clients.

Overview of RDC Model and Implementation

Incarnations are heavyweight processes upon which reconfiguration operations can be performed, including migration and re-connection of their communications interfaces. As currently implemented, they are single-threaded, but this is not an essential restriction in the model. Incarnations are mapped explicitly as they are created, or they can be mapped transparently by an external load balancing facility (the pool manager).

The structural components that go to make up the interconnections between incarnations – their streams and ports and the channels to which these are attached – have been designed to provide for all possible communications relationships to be established. Interactions are based upon the client-server and object-invoker paradigms, and include multicast communications.

Equus creates the primary incarnation of an RDC. Thereafter, the initial configuration is established by declaring a set of incarnations to be created, and a set of channels with character string labels. Each incarnation is declared to have an interface associated with one of the channels. The desired connectivity is created between the incarnations whose streams and ports are declared to be attached to the same channels. Streams, stream handles and port handles can be deposited in stream space as a result of setting up a configuration. Stream space provides a means of propagating these and other references, to serve dynamic needs such as those due to requests from clients. Incarnations which provide references via stream space do not have to synchronise with incarnations requiring their references. Nor do they have to be concerned with their connections to the incarnations they supply. Stream space caching eliminates redundant fetches from the stream space repository.

An important structure which subsumes a number of problems of reconfiguration is that of a set of server peer incarnations with clients. There are three alternative reconfiguration mechanisms to change the identity of the server peer that processes requests from given clients. The design of the interconnection components and these mechanisms means that clients can be affected singly or can be grouped as required with respect to these reconfigurations. The mechanisms can be applied transparently by external manager incarnations, or applied by mutual co-operation of the server peers involved if a decentralised management scheme is required.

The design of ports as queues of invocation messages which can be manipulated as part of reconfiguration means that re-associations between clients and server peers can take place consistently in bounded time. A sufficient pre-condition for such a reconfiguration to take place is for the current peer not to receive further requests from the client once it has completed any current transaction. Meeting this condition does not require interaction with the client concerned, and does not have to affect server processing for other clients.

The control network represented by an RDC's incarnations and any external management incarnations which may be employed (such as ones belonging to the pool manager) is itself reconfigurable, so that control can be passed to server RDCs and so that controllers can be replaced in the event of failure. The control mechanisms enable user-level code to be used to implement the node allocation and load balancing facilities provided by the pool manager. The design of the migration facility has been presented in relation to its use for load balancing and reaction to imminent node withdrawal. Its performance for load balancing is good when compared with other implementations in which migration-based load balancing is practiced, allowing for differences in communications bandwidth and processor type. Some improvements are necessary in relation to node withdrawal, but these could have been carried out straightforwardly given more time.

The chief feature of the kernel's architecture is the presence of per-incarnation ghost processes for the handling of operations and event notifications sent to incarnations. Operations and event notifications can be processed concurrently with the incarnation's execution, and concurrently with processing due to other incarnations at the same node. The use of ghosts simplified the multiplexing of the kernel's activities. For example, the ability to perform multiple migrations concurrently is an immediate consequence of this. The organisation of the communications infrastructure was a key component of the implementation effort. The algorithms used to achieve redirections of invocation messages as a consequence of communications reconfigurations and incarnation migrations have been presented. These preserve delivery guarantees and the ordering of invocation messages. The performance of the (unoptimised) invocations implementation is respectable when compared with other implementations. Initial figures for the performance of the reconfiguration mechanisms to change the destination of invocations are encouraging.

9.2 Further Work

There are two main areas in which the contributions described by this dissertation can be developed further. Firstly, there is the area of system servers providing shared facilities to client applications. Servers can improve their performance in relation to clients by matching their utilisation of hardware resources with their current client load. Secondly, there are those application domains which require intensive calculations, such as image processing. Computationally intensive applications can benefit by being able to harness every available processor in an under-utilised distributed computer system.

Whilst the RDC model and Equus together provide a framework within which adaptive servers and parallel applications can be realised, there is a lack of programming tools to simplify this task.

During the course of their development, the computational model and its implementation underwent considerable improvements in the light of implementation experience and application requirements. But what remains to be done is to write RDCs which really utilise the mechanisms developed, for applications with hard performance or consistency requirements. This practical experience is necessary if the right tools are to be developed. It is also necessary if the components of the operating environment upon which optimisation effort should be focused are to come to light.

Server Harnesses

Systems programmers writing adaptable servers must be directly concerned with reconfigurations. The RDC model allows for reconfiguration management and service provision to be separated, between manager and server peer incarnations associated with different modules. What is required for both manager and server peers are generalised harnesses. These would perform the necessary monitoring, synchronisation and reconfiguration actions, whilst presenting the server writer with an interface which is largely free of such concerns. Monitoring functions and reconfiguration descriptions would have to be supplied by the application. Declarations made at initialisation are all that should be required for this.

Programming Systems for Applications

For applications writers, the ideal tool would be a parallel programming language which avoided concern with configuration as much as possible, whilst allowing for the run-time support system to reconfigure transparently. The languages Linda and Orca go some way towards this goal (section 2.3.1). But neither provides a means for the programmer to declare which processes can or should be dynamically created and destroyed in response to variations in node availability during run time. Neither Linda nor Orca allows the programmer to declare under what application-specific circumstances these reconfigurations are permitted. Both programming models present scaling problems. It may be that problems of scale could be reduced by transparently partitioning the virtually shared data space and dynamically re-connecting processes to partitions. It seems a promising line of investigation to enquire whether virtually shared data could be efficiently implemented as a multiple-peer server, of which processes belonging to parallel programs are clients.

Persistence

The present implementation assumes that every RDC runs to completion in one period of execution. This may not be desirable if, for example, loading conditions arise under which very only slow progress can be made. Very long-running applications could be inactivated altogether until conditions improve, rather than be forced into a state of poor performance which nonetheless increases the load on the system as a whole.

The swapping mechanism required to improve the node-to-node migration facility could be extended naturally to that of migration to disc. By migrating all of its incarnations to disc after making them quiescent (as defined in Chapter 6), an RDC becomes a persistent object whose execution can be re-established. The mechanism by which it is currently envisaged that the pool manager informs an RDC of additions to its node allocation could in principle be extended so that it would cause the requesting incarnation to be automatically migrated back to a node for continued execution. This could then set about migrating the other incarnations from disc. The present kernel port location algorithms would suffice to re-connect incarnations, as they use their streams upon re-execution.

References

[ACTORS] G. Agha.
ACTORS - A Model of Concurrent Computation in Distributed Systems, MIT press.
1986.

[AMOEBA86] A. Tanenbaum, S. Mullender, R. van Renesse.
Using Sparse Capabilities in a Distributed Operating System.
Proc. 6th. International Conf. on Distributed Computer Systems, IEEE, pp. 558-563.
1986.

[AMOEBA89] R. van Renesse, Hans van Staveren, A. Tanenbaum.
The Performance of the World's Fastest Distributed Operating System.
Software – Practice and Experience, vol. 19, pp. 223-234.
Mar. 1989.

[AMOEBA90] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, H. van Staveren.
Amoeba - A Distributed Operating System for the 1990s.
IEEE Computer, vol. 23, no. 5, pp. 44 - 53.
May 1990.

[ARGUS] B. Liskov.
Distributed Programming in Argus.
Communications of the ACM, vol. 31 no. 3, pp. 300-312.
Mar. 1988.

[BRINCH] P. Brinch Hansen.
Distributed Processes: A Concurrent Programming Concept
Communications ACM, vol. 21, no. 11, pp. 934-941.
Nov. 1978 .

[CHARLOTTE87] Y. Artsy, H. Chang, R. Finkel.
Interprocess Communication in Charlotte.
IEEE Software, vol. 4, no.1, pp. 22-28.
Jan. 1987.

[CHARLOTTE89] Y. Artsy, R. Finkel.
Designing a Process Migration Facility - The Charlotte Experience.
IEEE Computer, vol. 22, no. 9, pp. 47-56.
Sep. 1989.

[CHORUS] M. Rozier, L. Martins.
The Chorus Distributed Operating System: some Design Issues.
Distributed Operating Systems: Theory and Practice, (ed. Y. Paker et al), NATO ASI Series, vol F28, Springer-Verlag, pp. 261-287.
1987.

[CONIC89a] J. Magee, J. Kramer, M. Sloman.
Constructing Distributed Systems in Conic.
IEEE Trans. Software Engineering, vol. 15, no. 6, pp. 663-675.
June 1989.

[CONIC89b] J. Kramer, J, Magee, A. Young.
A Refined Model of Change Management in Distributed Systems.
Proc. 3rd Workshop on Large Grain Parallelism, SEI/CMU.Pittsburgh, USA.
Oct. 1989.

[CONIC89c] J. Kramer, J, Magee and M. Sloman.
Managing Evolution in Distributed Systems.
Software Engineering Journal, pp. 321-329.
November 1989.

[CRAFT] D. H. Craft.
Resource Management in a Distributed Computer System.
Technical report no. 73, University of Cambridge Computer Laboratory.
1985.

[CSP] C.A.R. Hoare.
Communicating Sequential Processes.
Communications ACM, vol. 21, no. 8, pp. 666-677.
Aug. 1978

[CSR87] M. Barton, N. Edwards.
Occam in a Reconfigurable Local Area Network.
Proc. of the IFIP Conf. on Distributed Processing, Amsterdam, pp. 295-304.
Oct. 1987.

[CSR89] N. Edwards.
Dynamically Reconfigurable Systems.
PhD dissertation, Dept. of Electrical Engineering and Information Technology, Univ. of Bristol, UK.
Mar. 1989.

[DANNENBERG] R.B. Dannenberg.
Resource Sharing in Network of Personal Computers.
Tech. report CMU-CS-82-152, Carnegie-Mellon university, USA.
Dec. 1982.

[DEMOS83] M. Powell, B. Miller.
Process Migration in DEMOS/MP.
Proc. 9th. Symposium on Operating Systems Principles, ACM, pp. 110-119.
Nov. 1983.

[DEMOS87] B. Miller, D. Presotto, M. Powell.
DEMOS/MP: The Development of a Distributed Operating System.
Software-Practice and Experience, vol. 17, no. 4, pp. 277-290.
Apr. 1987.

[EMBOS] R. Olson.
Parallel Processing in a Message-Based Operating System.
IEEE Software, vol. 2, no.7, pp. 39-49.
July 1985.

[EQUUS89a] A. P. Sherman, T. Kindberg.
Horsebox.
Parallelogram, no. 14, pp. 18-19.
May 1989.

[EQUUS89b] D. Pountain.
Equus: A Parallel Operating System.
Byte, pp. 80IS-3 - 80IS-8.
Sep. 1989.

[EQUUS90] T. Kindberg.
Making Waves: Computation Dynamics on a Processor Pool.
Proc. Microsystem Design Show, pp. E/2/1 - E/2/15.
1990.

[FIREFLYRPC] M. Schroeder, M. Burrows.
Performance of Firefly RPC.
Proc. 12th Symposium on Operating System Principles, ACM, pp. 83-90.
Dec. 1989.

[HAGMANN] R. Hagmann.
Process Server: Sharing Processing Power in a Workstation Environment.
Proc. 6th. Int. Conf. Distributed Computing Systems, Vambridge, Mass., pp. 260-267.
May 1986.

[HAMILTON] A. Hamilton
A Transputer Farm Accelerator for Networked Computing Facilities
INMOS Technical Note 54, INMOS Ltd.
Sep. 1988

[HELIOS] *The Helios Operating System*, Perihelion Software, Prentice Hall.
1989

[HEWITT] C. Hewitt.
Viewing Control Structure as Patterns of Passing Messages.
Artificial Intelligence, vol. 8, no. 3, pp. 323-364.
June 1977.

[HILTUNEN] M. Hiltunen.
Building Fault-Tolerant Programs with Equus.
Tech. Report, Dept. of Computer Science, University of Helsinki.
June 1990.

[HPC] T. Le Blanc, S. Friedberg.
HPC: A Model of Structure and Change in Distributed Systems.
IEEE Trans. Computers, vol. C-34, no. 12, pp. 1114-1129.
Dec. 1985.

[ISIS] K. Birman, T. Joseph.
Exploiting Replication in Distributed Systems.
Chapter 15 of *Distributed Systems*, S. Mullender ed., acm press, pp. 319-367.
1989.

[ISSOS]	K. Schwan, R. Ramnath, S. Vasudevan, D. Ogle. A system for Parallel Programming. *Proc. 9th. Int. Conf. on Software Engineering,* IEEE, pp. 270-282. 1987.
[JONES]	A. Jones. The Object Model: a Conceptual Tool for Structuring Software. *Operating Systems: an Advanced Course*, LNCS vol 60, Springer-Verlag. 1979.
[KAASHOEK]	M. Kaashoek, A. Tanenbaum, S. Flynn Hummel, H. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, vol. 23, pp. 5-19. Oct. 1989.
[LADY]	D. Wybranietz, R. Massar. An Overview of LADY - a Language for the Implementation of Distributed Operating Systems. *Tech. Report 12/85*, SFB 124, University of Kaiserslautern. 1985.
[LEUNG]	C. Leung. The Survival Worm. *Tech. Report*, Polytechnic of Central London. Dec. 1987.
[LINDA]	S. Ahuja, N. Carriero, D. Gelernter. Linda and Friends. *IEEE Computer,* vol 19, no. 8, pp. 26-34. Aug. 1986
[LOCUS]	G. Popek, B. Walker. *The LOCUS Distributed System Architecture*, MIT Press. 1985.
[LYNX]	M.L Scott. Language Support for Loosely Coupled Distributed Programs. *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 88-103. Jan. 1987.
[MACH]	A. Tevanian, Jr. MACH: A Basis for Future UNIX Development. *Tech. report CMU-CS-87-139,* Carnegie-Mellon University, USA. 1987.

[MC68030] *MC68030 User's Manual,* Motorola Inc.
1987.

[MOSa] A. Barak, A. Litman.
MOS: A Multicomputer Operating System.
Software Practice and Experience, vol. 15(8), pp. 725-737.
Aug. 1985.

[MOSb] A. Barak, A. Shiloh.
A Distributed Load-balancing Policy for a Multicomputer.
Software Practice and Experience, vol. 15(9), pp. 901-913.
Sep. 1985.

[NICHOLS] D. Nichols.
Using Idle Workstations in a Shared Computing Environment.
Proc. 11th. Symposium on Operating System Principles, ACM, pp. 5-12.
Nov. 1987.

[NI] L. Ni, C-W. Xu, T. Gendreau.
A Distributed Drafting Algorithm for Load Balancing.
IEEE Transactions on Software Engineering, vol. SE-11, no. 10, pp. 1153-1161.
Oct. 1985.

[OCCAM] *occam 2 Reference Manual*, INMOS document number 72 occ 45 02, Prentice Hall International.
1988.

[ORCA] H. Bal, A. Tanenbaum.
Distributed Programming with Shared Data.
IEEE Conf. on Computer Languages, IEEE, pp. 82-91.
1988.

[POOL] P. America.
Definition of the Programming Language POOL-T.
Doc. no. 0091, Project 415, Philips Research Laboratories, Netherlands.
1986.

[ROSCOE] M. Solomon, R. Finkel.
The Roscoe Distributed Operating System.
Proc. 7th. Symposium on Operating System Principles, ACM, pp. 108-114.
Dec. 1979.

[RPC] A. Birrell, B. Nelson.
Implementing Remote Procedure Calls.
ACM Trans. on Computer Systems, vol. 2, ACM, pp. 39-59.
1984.

[SAHINER] A. Sahiner.
Self-Adapting Parallel Servers.
Tech. Report QMW/CPC/AVS/2, Centre for Parallel Computing, Queen Mary and Westfield College, University of London.
Sep. 1990.

[SHAPIRO] M. Shapiro.
Structure and Encapsulation in Distributed Systems: the Proxy Principle.
Proc. 6th. International Conf. on Distributed Computer Systems, IEEE, pp. 198-204.
1986.

[SHERMAN] A. Sherman.
The Equus Pool Manager - a Technical Overview.
Tech. Report, Zebra Parallel Ltd.
Mar. 1989.

[SPRITE] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelso, B. Welch.
The Sprite Network Operating System.
IEEE Computer, vol. 21, no. 2, pp. 23-36.
Feb. 1988.

[TRANSPUTER] *The Transputer Databook,* INMOS document number 72 TRN 203 00, INMOS Ltd.
1988.

[UNIX] D. Ritchie, K. Thompson.
The UNIX Time Sharing System.
Communications of the ACM, vol 17, no. 7, pp. 365-375.
July 1974.

[V84] D.R. Cheriton, W. Zwaenepoel.
The V Kernel: A Software Base for Distributed Systems.
IEEE Software, vol. 1, no. 2, pp. 19-42.
Apr. 1984.

[V85] M. Theimer, K. Lantz, D. Cheriton.
Pre-emptible Remote Execution Facilities for the V-system.
Proc. 10th. Symposium on Operating System Principles, ACM, pp. 2-12.
Dec. 1985.

[V88] D. Cheriton.
The V Distributed System.
Communications of the ACM, vol 31, no. 3, pp. 314-333.
Mar. 1988.

[WEIHL] W. Weihl.
Remote Procedure Call.
Chapter 4 of *Distributed Systems*, S. Mullender ed., acm press, pp. 65-86.
1989.

[WORMOS87a] T. Kindberg, A.V. Sahiner, Y.Paker.
Worm Programs.
Distributed Operating Systems: Theory and Practice, (ed. Y. Paker et al), NATO ASI Series, vol F28, Springer-Verlag, pp. 355-379.
1987.

[WORMOS87b] A.V. Sahiner, T. Kindberg, Y.Paker.
A Distributed Architecture for Image Programming.
Proc. British Computer Society Parallel Architectures and Computer Vision Workshop, Oxford, pp. 187-199.
Mar. 1987.

[WORMS] J. Shoch, J. Hupp.
The 'Worm' Programs - Early Experience with a Distributed Computation.
Communications of the ACM. vol. 25, no. 3, pp. 172-280.
Mar. 1982.

[ZAYAS] E. Zayas.
Attacking The Process Migration Bottleneck.
Proc. 11th. Symposium on Operating System Principles, ACM, pp. 13-24.
Nov. 1987.

Appendix A
Performance

This appendix describes experiments made to measure the performance of the implementation. Experiments were performed to measure communications times, migration times and the overhead due to communications reconfigurations.

The measurements are given as a guide only, to judge gross system performance. Efficiency was one of the goals of the design, but no attempt has been made to optimise the implementation.

The processor cards used in these experiments are based on Motorola 68030s running at 20 MHz. The cards are based on a 68020 design, which was not optimised for the 68030. In particular, the 68030's burst mode data access capability was not exploited in the card's design. The VME bus system clock runs at 16 MHz.

The system call *inc_time* is used in the communications experiments to determine elapsed times. The kernel handles timer interrupts at a rate of 60 Hz, and therefore the timing resolution is only about 17 milliseconds. In each experiment the action to be timed is therefore performed multiply, to reduce the indeterminacy per action.

Variations were found in the values measured for repeats of the same experiment. Factors which may account in part for variations in the recorded values in any of the experiments described here are:

- lack of clock accuracy. In addition to the clock's intrinsic lack of resolution, clock interrupts are suspended during certain kernel operations, to ensure that atomic updates are made to kernel data.
- the card running Unix accesses the disc over the shared VME bus. Unix-related activities which could not be suppressed, such as superblock updates, may compete for the bus with the test incarnations in some experiments.
- the kernel's time daemon process runs essential maintenance procedures periodically. In practice these run at different times in relation to each experiment, since the starting points of the experiments were not synchronised with the time daemon.

A.1 Invocation Timings

Table A.1 shows the results of experiments made to measure the cost to an incarnation of making the various types of invocation call. The times are shown graphically in Figures A.1 and A.2. In each experiment a sending incarnation makes the same invocation call iteratively in a simple loop. A corresponding incarnation at a different node calls *inc_receive* in a simple loop. No other incarnation is active during the course of the experiments.

In each experiment (apart from those measuring *inc_dgram* – see Table A.1) the sender makes the invocation call 1000 times, thus reducing the indeterminacy per call due to the clock resolution to about 0.02 milliseconds. The recorded results are average values taken from the elapsed times, with 10 experiments made in each case. The figures in brackets are the standard deviations of the results.

For comparison when considering bulk data transfer rates, a C loop employing register variables to copy data between nodes in four-byte units (kernel-to-kernel) achieves a maximum data transfer rate of 2.6 megabytes per second.

inc_invoke

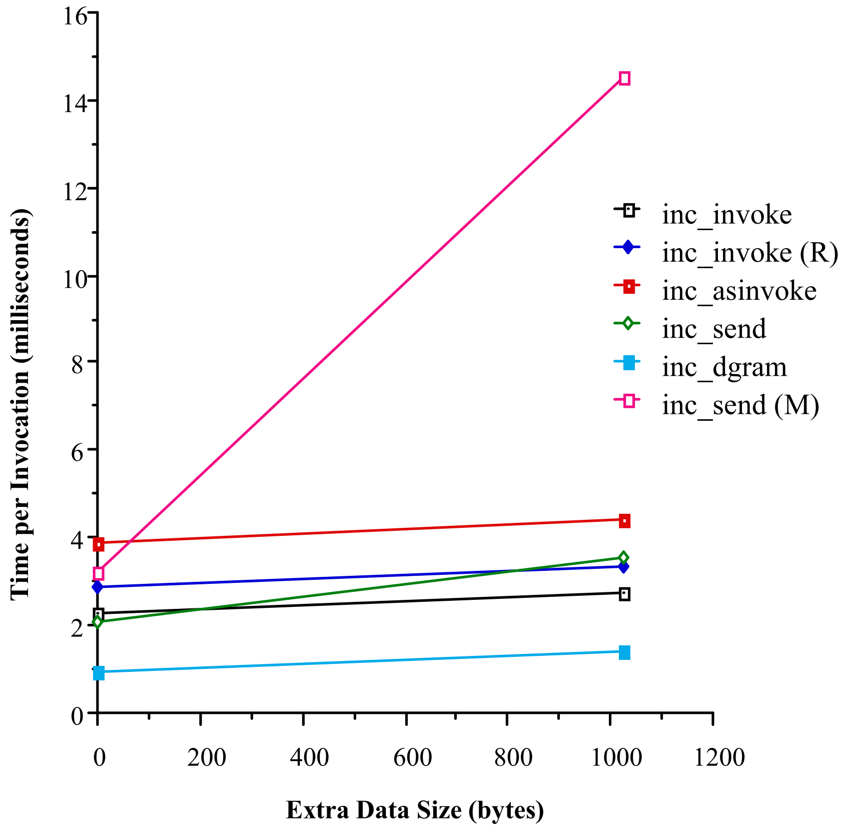
In the first experiment, *inc_invoke* is called by the sender, and the receiver does not make an explicit reply (Table A.1, first row). The time per call is 2.3 milliseconds if no extra data and no references are sent with the invocation message. If extra data are sent, the data transfer rate is 0.4 megabytes per second for 1 kilobyte, and 2.5 megabytes per second for 64 kilobytes of extra data.

primitive	0 extra bytes	1k	32k	64k
inc_invoke (no reply by rcvr)	2270 (22)	2771 (13)	13957 (14)	25480 (10)
inc_invoke (R) (rcvr replies)	2863 (22)	3377 (25)	14570 (21)	26100 (15)
inc_invoke (rcvr replies; stream each way)	3288 (31)	-	-	-
inc_asinvoke	3860 (11)	4430 (18)	14783 (20)	25493 (24)
inc_send	2098 (14)	3553 (7)	-	-
inc_dgram	940 [x500] (7)	1374 [x250] (8)	-	-
inc_send (M) (multicast to 2 receivers)	3228 (127)	14537 (10)		
inc_dgram (multicast to 2 receivers)	2494 [x250] (8)	-	-	-

Each entry gives the average over 10 experiments of the elapsed times for 1000 calls, in milliseconds. Standard deviations are given in round brackets. In the case of *inc_dgram*, the number of calls made in each experiment was lower because of buffering restrictions. The actual number of calls made is given in square brackets. The values given for *inc_dgram* have been muliplied by the appropriate factor to obtain a figure for 1000 calls.

Table A.1: Invocation Times Between Nodes.

The difference in transfer rates is to be expected, because invocation message delivery has a fixed and non-negligible overhead due to kernel message preparation, transmission and acknowledgement. Once a message header has been delivered, however, transferring extra data over the VME bus is relatively cheap. There is a limit of 64 kilobytes on the size of data which the kernel copies from a single message over the VME bus, so as to prevent a process from hogging the bus.



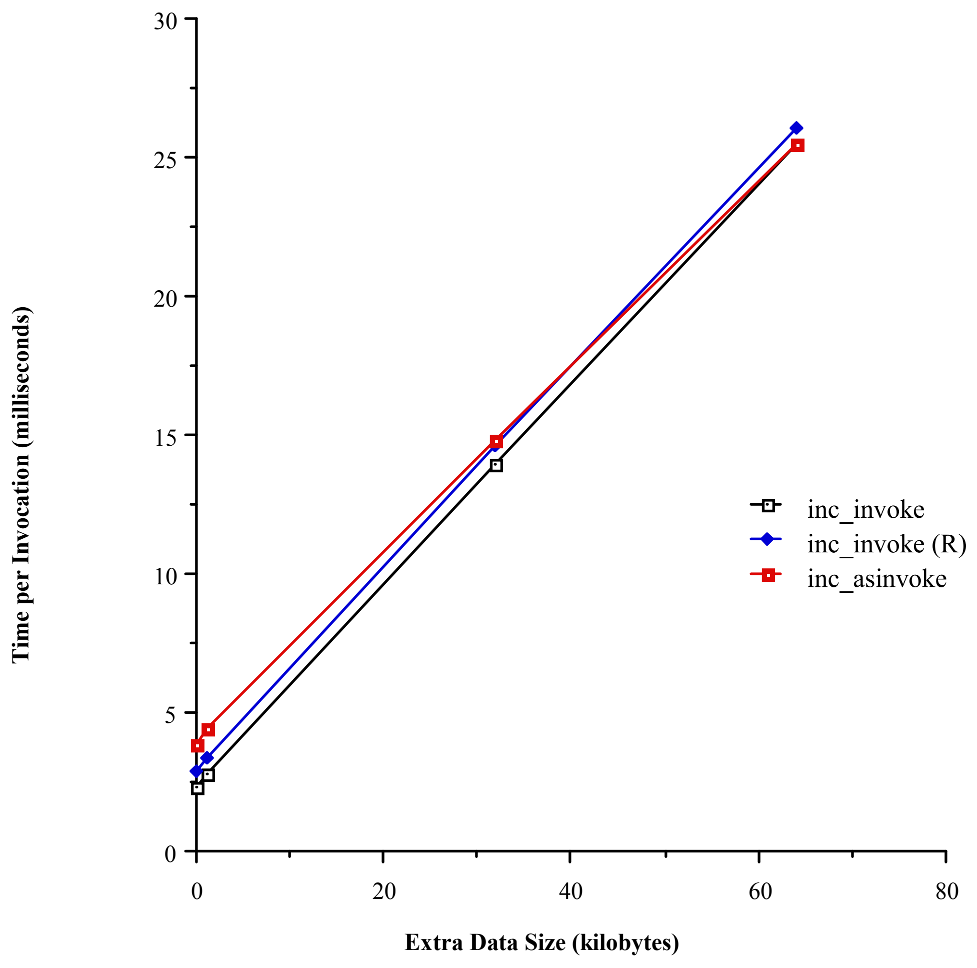
Data taken from Table A.1. (R) = with reply; (M) = multicast to two nodes.

Figure A.1: Invocation Times Between Nodes (0-1k Extra Data).

A single *inc_invoke* call can therefore transfer up to this amount of data with a total of only two associated kernel messages (one for the invocation message, one for the reply or acknowledgement). If the extra data exceeded this size, the transfer would involve the transmission of further kernel messages.

In the second experiment the receiver replies (using *inc_reply*), but without extra data or a reference (Table A.1, second row). The time per call is increased by 0.6 milliseconds to 2.9 milliseconds. The data transfer rate at 1 kilobyte of extra data drops to 0.3 megabytes per second, and to 2.4 megabytes per second when 64 kilobytes of extra data are sent in the *inc_invoke* call.

In the third experiment, a stream is propagated in the invocation message and the reply message (Table A.1, third row). No extra data are sent in either direction. The stream used for propagation is destroyed each time, so that the system limit on the number of streams held by a single incarnation is not exceeded. From the



Data taken from Table A.1. (R) = with reply.

Figure A.2: Invocation Times Between Nodes (0-64k Extra Data).

figures in the second and third rows of Table A.1, the overhead of propagating a stream in a message using *inc_invoke* or *inc_reply* (assumed to be the same in each case) is (3.3 - 2.9)/2 = 0.2 milliseconds. The additional overhead due to propagating any other type of reference other than a port in a message is expected to be similar to the value for a stream, since similar processing is involved. Also, this overhead is expected to be similar across the different invocation calls for the same reason.

inc_asinvoke

In the experiments using *inc_asinvoke* (Table A.1, fourth row), a buffer limit of one was used, with the consequence that each attempt to make the call would be blocked until the previous invocation had been completed. This buffer limit is expected to produce a worst case for the sender's asynchronous performance (if the buffer limit is zero, sending is effectively synchronous). The time with no extra data sent is 3.9 milliseconds, and with 64 kilobytes of extra data the transfer rate is 2.5 megabytes per second. When an invocation message is received, the receiver's kernel sends a message to the sender's kernel informing the latter of completion. The sender's kernel has to acknowledge this. This accounts for the large difference between the first figure (no extra data) and the comparable figures for *inc_invoke*. The experiment with no extra data was made for comparison only. *inc_asinvoke* is only intended for sending bulk data asynchronously.

inc_send and inc_dgram

When used over a unicast channel, *inc_send* is the fastest reliable invocation call for messages with no extra data, at 2.1 milliseconds (Table A.1, fifth row). With 1 kilobyte of extra data, the time rises to 3.6 milliseconds. Extra data sent using *inc_send* or *inc_dgram* is copied twice: once at the time of the call, and once at the time of reception. This probably accounts for the fact that *inc_invoke* is faster than *inc_send* with 1 kilobyte of extra data.

Buffering limitations restricted the number of times *inc_dgram* could be called in each experiment (Table A.1, sixth row). At 0.9 milliseconds and 1.4 milliseconds for 0 and 1 kilobytes of extra data respectively (over a unicast channel) *inc_dgram* is considerably faster than the other primitives. No messages were dropped in the experiments using this call – in either the unicast or multicast case.

The figures for *inc_send* and *inc_dgram* sending to two receiving incarnations over a multicast channel show a marked drop in performance over the unicast case (Table A.1, last two rows). This is to be considered in the light of the following points regarding the implementation. Each multicast has to be achieved using a point-to-point transmission to each node running the Equus kernel (an attempt has been made to make these transmissions overlap to some extent, however). And every node has to process the multicast message, even though only two accept it for delivery to local kernel ports.

Intra-node Invocations

Table A.2 shows the results of experiments in which the sender and receiver reside at the same host node. It may seem surprising that some figures recorded with 0 or 1 kilobyte of extra data are actually higher than corresponding figures between incarnations at different nodes. This is probably due to an attempt to optimise the scheduler, by which a process awaiting a low-level event is not descheduled unless there is another process ready to run. A process switch is thereby avoided if the waiting process is the next to be made ready to run. This optimisation is brought into effect when two nodes are used: the only other process which might run when the sender is blocked is the time daemon. When the same node is used, however,

primitive	0 extra bytes	1k	32k	64k
inc_invoke *(no reply by rcvr)*	2370 (7)	2733 (1)	9022 (8)	15467 (1)
inc_invoke *(rcvr replies)*	2918 (9)	3777 (8)	9568 (5)	16007 (22)
inc_asinvoke	3790 (8)	4318 (5)	13536 (7)	23016 (1)

Experimental conditions are the same as for those whose results are given in Table A.1, except that the sending and receiving incarnations reside at the same node. The average over ten experiments is recorded in each case. Standard deviations are given in brackets. All figures are in milliseconds.

Table A.2: Invocation Times Between Incarnations at Same Node.

the receiver and the sender are alternately the current, executing process. A full context switch is required every time this alternation occurs.

primitive	1k extra bytes	32k
inc_copyto	2802 (9)	13990 (13)
inc_copyfrom	2779 (8)	13240 (8)

The times in milliseconds for 1000 calls of *inc_copyto* and *inc_copyfrom* between incarnations residing at different nodes. The average value over ten experiments is given, with the standard deviation given in brackets.

Table A.3: Times to Copy Data Asynchronously Between Nodes.

inc_copyto and inc_copyfrom

Table A.3 shows the times taken for 1000 calls of *inc_copyto* and *inc_copyfrom*, each averaged over 10 experiments. The figures are close to those for *inc_invoke* with no explicit reply (Table A.1). This is to be expected, since the calls involve a similar exchange of a single kernel message in each direction. Data copying takes place after the request is received, just before an acknowledgement is generated. In the case of *inc_copyto* and *inc_copyfrom*, a ghost process handles the request; in the case of *inc_invoke*, a system incarnation handles the request.

address space size (kilobytes)	**migration time (1 port)**	**migration time (9 ports)**
70	46 (15)	56 (14)
134	73 (24)	79 (14)
196	96 (27)	107 (15)
390	173 (34)	185 (23)
518	223 (40)	234 (22)
1030	419 (43)	426 (19)

Times averaged over 100 migrations (standard deviations in brackets). All times are in milliseconds. Size is that of text + heap + stack + pincdata, each rounded to next multiple of 2 kilobytes.

Table A.4: Migration Times.

A.2 Migration

Table A.4 shows the results of experiments to measure the elapsed time taken to migrate incarnations of various sizes. In each experiment the incarnation executes a simple loop to receive messages from a port, to which no messages are sent.

The incarnation is migrated 100 times, over a set of ten nodes at which no other incarnation executes. The incarnation which migrates it executes at a separate node which the child never visits.

The elapsed time is measured by the kernel at the migration destination node. The results of these measurements are supplied via the event mechanism to the incarnation which migrates the child. The kernel begins timing at the point that the request to host the incarnation has been validated by the loader, and finishes timing immediately before the incarnation re-commences execution at the new node. This time is therefore an upper bound for the freeze time experienced by any incarnation of the same address space size. The address space components which are included in the total size are the text, heap, stack and pincdata segments. For convenience, the sizes of these segments are rounded up to the next multiple of two kilobytes (the page size).

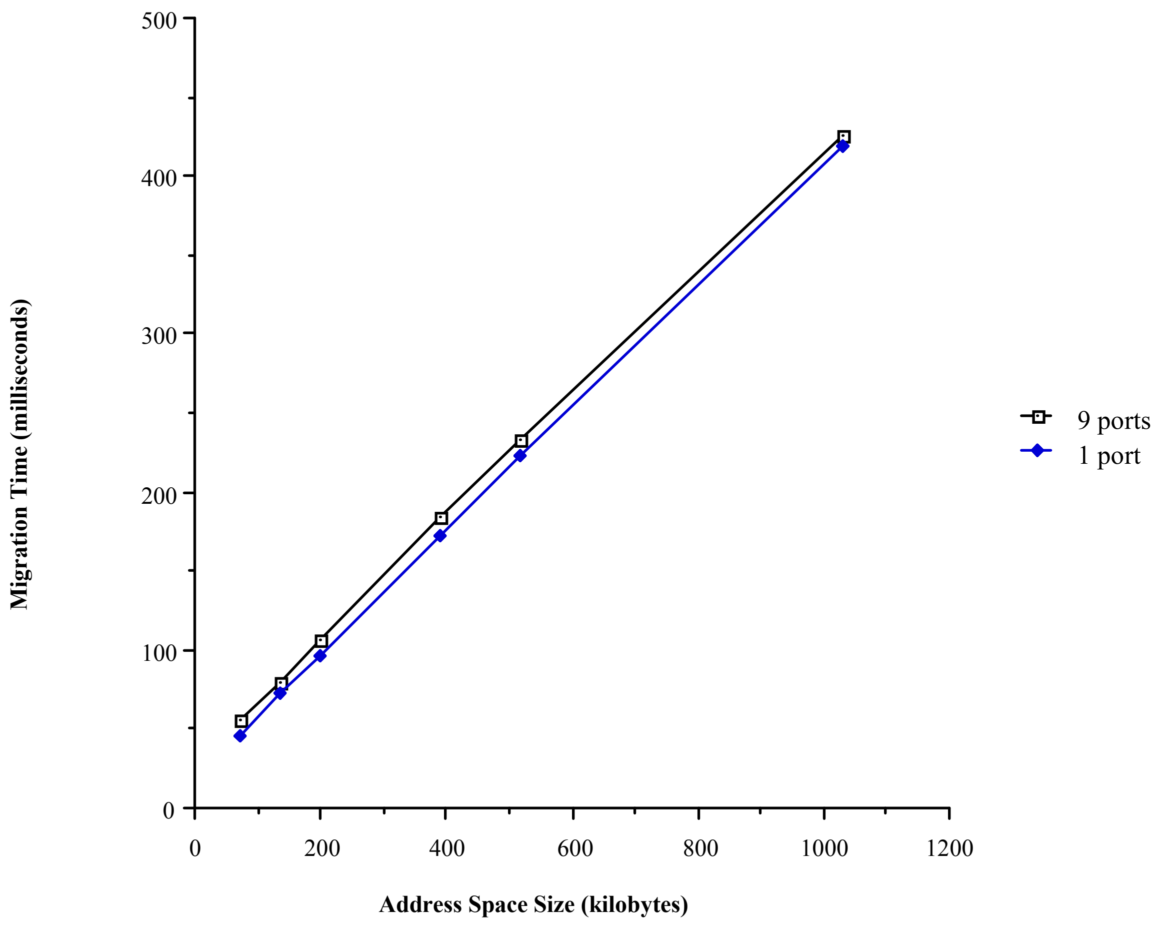


Figure A.3: Migration Times.

Two sets of experiments were performed. In the first, the child has nine ports (which happens to be the number needed for an all-purpose experimentation incarnation), and in the second it has one port. Each port requires processing during migration. It has to be initialised at the destination node with a kernel port and a logical address database entry.

The results are shown in graphical form in Figure A.3. Linear regression performed upon these data gives the following relationships:

migration time =	$20.27 + 0.388s$	(1 port)
	$30.17 + 0.387s$	(9 ports)

where s is the total address space size. Together these imply that the overhead per port is:

$$(30.17 - 20.27)/(9 - 1) \sim 1.2 \text{ milliseconds.}$$

We then obtain, approximately:

migration time $= 19 + 1.2p + 0.39s$ milliseconds,

where p is the number of ports and s is the total address space size.

Migration is expected to take longer if the incarnation was active during migration (instead of being blocked in an *inc_receive* call). This is because the ghost's activities in performing the migration would be interleaved with the execution of the incarnation, until its heap segment was requested (see Section 7.4). The ghost runs at a higher scheduling priority than the system incarnation, however, so degradation is expected to be slight.

If messages were sent to the migrating incarnation in these experiments, then this would be expected to degrade migration performance, since any attached or arrived messages have to be forwarded from the source node.

A.3 Changing the Receiver

The following experiments involve an incarnation making invocations which are received alternately by two receiver incarnations. The experiments are designed to measure the overheads which are incurred by the sender as a result of the reconfigurations which alter the receiver. Measurements are taken for the three mechanisms of port propagation, port attachment and stream rebinding. The results are given in Table A.5.

The same sending incarnation used in the invocation timing experiments is employed. In this case, however, the sender makes 10,000 calls to *inc_send*. Each call sends a simple message with no reference or extra data. In each experiment 100

experiment	elapsed send time	no. of reconfig-urations	no. of messages forwarded	overhead to sender per reconf.
no reconfigurations	21897 (35)	0	0	0
port propagation	22897 (145)	100	201 (2)	10
port attachment	22713 (241)	100	84 (9)	8.2
stream rebinding	22635 (183)	100	77 (17)	7.4

Times are for 10,000 calls to *inc_send* made by one sender to two receiving incarnations in alternation. Invocation messages queued at the current receiver's port are forwarded to the other receiver. The figures given are averages taken over ten experiments for each case. Standard deviations are shown in brackets. All times are in milliseconds.

Table A.5: Communications Reconfiguration Measurement Results.

reconfigurations take place, and approximately 100 invocation messages are received by each receiver before the sender is re-connected to the other one.

The receiving incarnations are different to those used in the invocation timing experiments. This is reflected in the time measured for the case of no reconfigurations, which is slightly larger than an extrapolation from the measurement for 1,000 *inc_send*s would suggest (Table A.5, first row). The sender records a sequence number in the header of the invocation messages, and the receivers record the gaps they find in the sequence numbers of the messages they receive. This information is used to confirm the number of reconfigurations that take place, and to check that messages are not lost or duplicated as a result of the reconfigurations. A second difference from the receivers in the invocation timing measurements is that they set a timeout for each *inc_receive* call. This is so that each receiver eventually times out when the sender has finished, and reports the gaps in message sequence numbers it detected.

In each experiment, the queue of messages at the current receiver's port is forwarded. The kernel keeps track of the total number of acknowledged messages which it forwards to the other port in each experiment. The forwarding of messages which have already been acknowledged is carried out reliably (see Section 8.5). The kernel may, additionally, forward a pending message during a reconfiguration. An experiment to account fully for the total time spent during a reconfiguration would have to record whether a pending message was forwarded. This was omitted for lack of time available to make these experiments. There could be at most one pending message per reconfiguration in these experiments (since there is only one sender); and forwarding a pending message can be expected to take less than half the time it takes to forward an acknowledged message.

Port Propagation

In this experiment each receiver:

1] Receives 100 messages from the current *receive* port.

2] Propagates its port in a message with no extra data to the other incarnation. It uses *inc_send* for this (*inc_invoke* could have been used), over a stream connected to the other's *swap* port. A flag is set in the message's reference descriptor to cause the destruction of the port upon propagation (and therefore the forwarding of the message queue).

3] Receives a new receive port in a message, using its swap port.

4] Repeats the above from step 1.

Comparing the elapsed send times for this case and the case in which no reconfigurations take place (Table A.5), the overhead to the sender per reconfiguration is:

(22897 - 21897)/100 = 10 milliseconds.

On average:

201/100 ~ 2 acknowledged kernel messages are forwarded per reconfiguration.

Port attachment

In this experiment, a manager incarnation (separate to the sender and receivers) performs a port attachment operation affecting the two receivers (*inc_attachPort*). The receivers remain in a receiving loop and do not actively take part in the reconfigurations. The manager pauses between the reconfiguration operations. The duration of this pause was determined heuristically so that 100 reconfigurations were performed altogether whilst sending continued, and so that the last took effect just before sending stopped. This meant that about 100 messages were received by a receiver each time before a reconfiguration took place.

Comparing the elapsed sending times from Table A.5, the overhead to the sender per reconfiguration is:

(22713 - 21897)/100 ~ 8.2 milliseconds.

On average:

84/100 ~ 0.8 acknowledged kernel messages are forwarded per reconfiguration.

The total number of messages forwarded is less than when port propagation was used. This is to be expected, because in the latter case messages are given time to arrive while the port is propagated in a user-level message and the receiver incarnation does not attempt to receive. With port attachment, the receiver is active up to the point when the reconfiguration operation takes effect.

Stream Rebinding

This experiment is the same as the last, except that the manager incarnation performs a stream rebinding operation instead of a port attachment operation.

Comparing the elapsed sending times from Table A.5, the overhead to the sender per reconfiguration is:

(22635 - 21897)/100 ~ 7.4 milliseconds.

On average:

77/100 ~ 0.8 acknowledged kernel messages are forwarded per reconfiguration.

Appendix B
Equus Calls

This appendix gives a brief description of each of the basic Equus-specific calls available to the RDC programmer. There are around 60 calls, but there are only 30 system calls handled by the Equus kernel. This is because a number of Equus calls share one or more underlying system calls to which they provide their own distinguishing arguments. In this case the system call may have no other utility and is not defined as part of the system interface.

The relevant structure definitions are first defined. The calls are then described in groups according to their functionality.

B.1 Definitions

```
/*          INCARNATION CREATION     */

/* node allocation entry */
typedef struct    {
    unsigned int ne_node;           /* node identifier              */
    unsigned int ne_load;           /* node's processing load   */
}   NodeEntry;

/* structure used to specify interfaces to child */
typedef struct    {
    unsigned int ii_flags;          /* pertain to interface creation     */
    RefDescrip           ii_ref;          /* child's interface                 */
    char                 ii_chanlabel[20];      /* channel label                    */
    char                 ii_hanlabel[20];       /* handle label                     */
}   IncInterface;

/* parameters supplied to child */
typedef struct    {
    int                   ip_datasize;  /* size of data arguments in bytes          */
    char            *ip_dataargs;/* pointer to data arguments           */
    int              ip_nifaces;   /* # of interfaces                     */
    IncInterface    *ip_ifaces;        /* array of interface descriptors       */
}   IncParams;

/* Structure provided to inc_create */
typedef struct    {
    unsigned int ic_state;       /* INCHOATE/STARTED        */
    IncParams           ic_params;    /* decaration of child's parameters      */
}   IncCreateParams;

/*          COMMUNICATIONS      */

/* specify buffer in local inc's address space */
typedef struct    {
    char            *bf_base;      /* start of buffer                  */
    unsigned int           bf_cnt;/* # of bytes in buffer              */
}   Buf;

/* specify buffer in remote inc's address space */
typedef struct    {
    int                     ab_bufhandle;         /* handle on remote buffer         */
    unsigned int ab_cnt;                 /* # bytes in remote buffer         */
}   AlienBuf;

/* reference identifier   */
typedef union     {
    unsigned int oi_inc;         /* incarnation handle       */
    int                     oi_stream;            /* stream                    */
    int                     oi_port;              /* port                      */
    int                     oi_streamhandle;      /* stream handle             */
    Buf                     oi_buf;               /* local buffer              */
    AlienBuf                oi_alienbuf;  /* alien buffer              */
}   Objectid;
```

```
/* reference descriptor */
typedef struct    {
    unsigned char        rd_type;       /* type of reference          */
    Objectid             rd_obj;        /* local name/descrip of ref         */
    unsigned int  rd_keep: 1;   /* 1 iff keep ref on propagation    */
    unsigned int  rd_domain: 3; /* propagation restriction*/
    unsigned int  rd_flags: 28;  /* type-specific flags              */
}   RefDescrip;

/*message structure */
typedef struct    {
    char                 m_head[32];  /* header                              */
    RefDescrip           m_ref;         /* ref to propagate                    */
    unsigned int  m_cnt;          /* # bytes extra data sent out       */
    unsigned int  m_nreply;             /* # bytes in reply                    */
    char                 *m_base;               /* buffer for extra data                */
}   Msg;

/* structure to get info about msgs from selected streams using inc_testPort */
typedef struct    {
    int      sm_smhan;            /* stream handle                         */
    int      sm_nmsgs;            /* no. msgs arrived from stream  */
}   SelectMessages;

/*            EVENTS & INCARNATION STATUS            */

/* Structure describing events */
typedef struct    {
    unsigned int  ed_reference;/* reference identifier              */
    unsigned int  ed_evtype;            /* event type              */
    unsigned int  ed_info1;             /* extra info              */
    unsigned int  ed_info2;             /* extra info              */
}   EvDesc;
```

```
/* structure describing incarnation state       */
typedef struct    {
    unsigned short      is_state;       /* state of incarnation                      */
    unsigned short      is_load;        /* load due to inc.                          */
    unsigned int        is_node;        /* location of inc                           */
    unsigned int        is_usertime; /* time executing user code so far  */
    unsigned int        is_systime;     /* time executing sys code so far  */
    unsigned int        is_realtime;    /* total elapsed time since creation         */
    unsigned short      is_ntext;       /* no. of pages of program text              */
    unsigned short      is_nheap;       /* no. of pages of program heap              */
    unsigned short      is_nstack;      /* no. of pages of program stack             */
    unsigned char       is_npinc;       /* no. of pages of pincdata          */
    unsigned char       is_nmsgs;       /* no. of arrived invocation messages        */
}   IncState;

/* local incarnation times returned by inc_time */
typedef struct    {
    unsigned int  tb_sys;        /* system time        */
    unsigned int  tb_user;       /* user time  */
    unsigned int  tb_elapsed;    /* elapsed time       */
}   TimeBuf;
```

B.2 Incarnation Creation

inc_nodes(nnodes, node_list)
    int   nnodes;
    NodeEntry   *node_list;

This returns in *nodes_list* up to *nnodes* entries giving the caller's RDC's node allocation (their identifiers and states of loading). It returns the number of entries filled in.

inc_getModByName(modname)
    char   *modname;

This returns the non-negative integer identifier of a module from its null-terminated character string name given by *modname*.

inc_create(blueprint, node, inchan_p, stream_p, create_params_p)

```
    unsigned int        blueprint
    unsigned int        node;
    unsigned int        *inchan_p;
    int                 *stream_p;
    IncCreateParams     *create_params_p;
```

blueprint is either a module identifier or the incarnation identifier of a frozen, INCHOATE incarnation created by a previous call to **inc_create**. *node* specifies where the new incarnation is to be created, and *stream_p* is used to return the identifier of a stream to the new incarnation's standard port, *inchan_p* that of the new incarnation's handle. *create_params_p* points to a data structure which must be set to determine the new incarnation's initial state (STARTED or INCHOATE); its data arguments; and a declaration of its interface arguments.

ss_isStream(iface_p, chan_label, handle_label)

```
    IncInterface     *iface_p;
    char             *chan_label, *handle_label;
```

This stipulates a stream interface for a child, using the structure pointed to by *iface_p*. The stream is to be attached to a channel labelled by the null-terminated character string *chan_label*. If non-NULL, *handle_label* is to point to a character string label: a stream handle referring to the new stream is be deposited in stream space with the given label.

ss_isPort(iface_p, do_create, chan_label, handle_label)

```
    IncInterface     *iface_p;
    int              do_create;
    char             *chan_label, *handle_label;
```

This stipulates a port interface for a child, using the structure pointed to by *iface_p*. The new port is to be attached to a channel labelled with *chan_label*. If do_create is CHAN_NEW, a new channel is to be created and a stream attached to the channel is to be deposited in stream space with the character string label *chan_label*. The new port is to be attached to this channel. Otherwise it is to be attached to a channel created by another declaration. If non-NULL, *handle_label* is to point to a character string label: a port handle referring to the new port is be deposited in stream space with the given label.

ss_createChan(chan_label, stream_p, port_p, chan_type)

```
    char        *chan_label;
    int *stream_p, *port_p;
    int chan_type;
```

This creates a channel. The new channel is to be of type *chan_type* (CHAN_UNICAST or CHAN_MULTICAST), and a stream attached to it is deposited in stream space with the label *chan_label*. If non-NULL, *stream_p* and *port_p* are used to return the identifiers of a stream and port respectively, attached to the new channel.

inc_paramRef(incparams_p, interface_index)

```
    IncParams       *incparams_p;
    int         interface_index;
```

This is a utility to extract and return the identifier of the reference of the interface with index *interface_index* referenced in a child's parameters given by *incparams_p*.

inc_fork(node, inchan_p, nports, ports_list, stream_list)

```
    unsigned int    node;
    unsigned int    *inchan_p;
    int         nports;
    int         *ports_list;
    int         *streams_list;
```

This forks a copy of the calling incarnation to node *node*, returning the new incarnation's identifier using *inchan_p*. The identifiers of ports to be attached in the forked child have to be supplied in *ports_list*. Ports attached to a multicast channel must have the corresponding stream entry set to the identifier of a stream attached to the channel. When the call returns in the parent, the corresponding entries in *streams_list* are filled with the identifers of streams attached to the new channels.

B.3 Stream Space

```
ss_put(label, refp, ncopies)
    char            *label;
    RefDescrip      *ref_type;
    int        ncopies;
```

```
ss_get(label, refp)
    char            *label;
    RefDescrip      *ref_type;
```

```
ss_return(label, keepqueue)
    char            *label;
    int        keepqueue;
```

ss_put puts into stream space the reference described in type and identifier by *refp* with label *label*. The reference is kept by default, but this can be overridden in the case of a port. *ncopies* is either SS_UNLIMITED – an unlimited number of copies can be taken out and will be cached; SS_DONTCACHE – an unlimited number of copies can be taken out, but will not be cached by the run-time system; or a positive integer. The latter case is used to specify that at most this number of copies can be obtained simultaneously from stream space.

Copies are obtained using **ss_get**, which returns the type and identifier of the reference it has obtained, using *refp*. When only a limited number of copies can be taken out, they are replaced in stream space using **ss_return**. *Keepqueue* is examined only in the case of propagating a unicast port. If TRUE, the incarnation calling **ss_return** retains the port.

```
ss_delete(label)
char   *label;
```

This deletes from stream space any entry with the label *label*. This does not delete cache entries, however.

```
ss_print()
```

This prints onto the standard output the contents of stream space.

B.4 Communications

B.4.1 Message Passing

inc_invoke(stream, msg)
    int stream;
    Msg *msg;

Send the message *msg* down stream *stream* and optionally get a reply, overwriting first two fields of msg and possibly the messages's extra data block.

inc_asinvoke(stream, msg)
    int stream;
    Msg *msg;

Send the message *msg* down stream *stream*. This time no reply can be given, but the message can contain arbitrarily much extra data which is sent asynchronously from the caller's address space – the buffer should not be overwritten whilst it is in use by the system.

inc_send(stream, msg)
    int stream;
    Msg *msg;

Send the message *msg* down stream *stream*. This time no reply can be given, and the block of extra data cannot be larger than a system-defined amount (1.5K bytes). In this case, however, the extra data is copied from the caller's address space before the call completes, so the user need not be involved in buffer management.

inc_dgram(stream, msg)
    int stream;
    Msg *msg;

Send the message *msg* down stream *stream*. This call is the same as **inc_send** except that message delivery is unreliable.

```
inc_receive(port, msg, block_invoker, selector, timeout)
    int port;
    Msg     *msg;
    int block_invoker;
    int selector;
    int timeout;
```

Receive a message into the message buffer *msg* using the port *port*. If *block_invoker* is non-zero, a caller of **inc_invoke** is blocked until the receiver issues a call to **inc_reply** using the reply handle returned from **inc_receive**. Otherwise the invoker is unblocked implicitly, and its message data is not overwritten with a reply message. If *selector* is a valid stream handle identifier, the call will attempt to receive only a message sent using the stream referenced by this stream handle. If *timeout* is negative, **inc_receive** waits indefinitely for an invocation message to arrive. If it is zero, it receives a message if one is currently available, and otherwise returns immediately. If positive, it is the maximum number of milliseconds the call is to wait until a message arrives. A return value of 0 indicates no messages were received, of 1 indicates a message was received, and a positive value greater than 1 is the identifier of a returned reply handle.

```
inc_reply(reply_handle, msg)
    int reply_handle;
    Msg     *msg;
```

Reply to an **inc_invoke** caller referenced by *reply_handle* (obtained from **inc_receive**). The message *msg* is sent back in replying, and overwrites the invoker's. An extra data block can be present in the reply message, as can a reference. On obtaining the reply message, the invoker is unblocked.

```
inc_forward(stream, reply_handle, msg)
    int stream;
    int reply_handle;
    Msg     *msg;
```

Forward the message whose receipt returned *reply_handle* down the stream *stream* (which must be attached to a unicast channel). The forwarded message data and reference is an exact copy of that received, except that its data in the *m_head* field is overwritten by that of **msg*.

B.4.2 Data copying

inc_copyto(buffer_handle, offset, data, nbytes)

inc_copyfrom(buffer_handle, offset, data, nbytes)

```
    int             buffer_handle;
    unsigned int    offset;
    char            *data;
    unsigned int    nbytes;
```

buffer_handle refers to a buffer in another incarnation's address space, and these two calls copy data to/from it (starting at offset *offset*) from/to the caller's local buffer (address *data*, size *nbytes*).

B.4.3 Auxiliary Calls

inc_testPort(port, nheaders, header_array, nselect, select_array)

```
    int             port;
    int             nheader;
    char            *header_array;
    int             nselect;
    SelectMessages  *select_array;
```

This call returns the number of messages at the port *port*. It also a) copies up to *nheader* bytes of arrived message headers in the array *header_array*, and b) fills up to *nselect* entries in *select_array* with the number of arrived messages at the port *port* originating from the streams to which stream handles supplied in these entries refer.

inc_setBufferLimit(stream, limit)

```
    int stream;
    int limit;
```

This call sets the buffer limit associated with stream *stream* to be *limit*, which must be positive. This value determines conditions under which a call to **inc_asinvoke** will block.

B.5 Channels, Streams and Ports

inc_chanCreate(chantype, stream_p, port_p)
```
    int          chantype;
    int          *stream_p;
    int          *port_p;
```

This creates a new channel of type *chantype*. *chantype* can be CHAN_UNICAST or CHAN_MULTICAST. It returns the identifiers of a stream and a port attached to it, using *stream_p* and *port_p* .

inc_portCreate(stream)
```
    int stream;
```

This creates a new port attached to the same channel as the stream *stream*. If successful, the identifier of the new port is returned.

inc_hanCreate(stream_or_port)
```
    int stream_or_port;
```

This creates and returns the identifier of a stream or port handle referring to the stream or port *stream_or_port*.

B.6 Reconfiguring Communications

inc_rebindStream(stream_handle, stream, port)
```
    int stream_handle;
    int stream;
    int port;
```

This rebinds the stream referenced by *stream_handle* with the stream *stream*. If port is not NO_PORT and is a valid port or port handle identifier, then the call forwards any messages which have already arrived from the stream referenced by *stream_handle* at the port referred to by *port*, before this rebinding takes place.

```
inc_attachPort(newport, oldport, stream, move_q)
    int newport;
    int oldport;
    int stream;
    int         move_q;
```

This attaches the port referred to by *newport* to the channel referred to by *stream*, and optionally detaches the port *oldport* from the same channel. If *move_q* is TRUE, then any messages queued at *oldport* which have arrived over this channel are first forwarded to *newport*. *oldport* and *newport* may be referred to either by a local port identifier or a port handle.

```
inc_freezeStream(stream_handle)

inc_unfreezeStream(stream_handle)
    int stream_handle;
```

inc_freezeStream causes any attempts to send invocations using the stream referenced by *stream_handle* to cause the invoker to block before sending the invocation (an invocation being made at the time of this call is allowed to complete). Such an invoker can only be unblocked by a call to **inc_unfreezeStream**.

B.7 Control

```
inc_makeDependent(child, type)
    unsigned int    child;
    int         type;
```

This call makes the incarnation *child* dependent upon the caller. If *type* is DEP_ABSOLUTE the child will be made absolutely dependent on the caller; if it is DEP_GROUP it will be made group-dependent upon it; if it is DEP_NONE, any dependency is cancelled.

inc_status(child, statep)
unsigned int child;
IncState *statep;

inc_status returns one of the values INC_IS_EXECUTING, INC_IS_FROZEN, INC_IS_FAULTING, INC_IS_UNSREACHABLE in *statep*->is_state If *child* is not unreachable, it fills in the fields of **statep* with relevant information.

inc_nodeAlloc(node, rdcid)
  unsigned int node;
  unsigned int rdcid;

This causes the node *node* to be allocated to the RDC whose identifier is *rdcid*. This call can only be successfully called by RDCs launched as system RDCs.

inc_nodeWithdraw(node, rdcid, kill)
  unsigned int node;
  unsigned int rdcid;
  int kill;

This causes the node *node* to be withdrawn from the RDC whose identifier is *rdcid*. If *kill* is TRUE, any incarnations belonging to the RDC and not migrating away from the node are destroyed automatically on withdrawal. This call can only be successfully called by RDCs launched as system RDCs.

inc_destroy(child)
  unsigned int child;

Terminate the incarnation *child*.

inc_freeze(child)
  unsigned int child;

Freeze the execution of the incarnation *child*.

inc_unfreeze(child, priority)
    unsigned int   child;
    unsigned int   priority;

Unfreeze the execution of the incarnation *child*. It is to re-commence execution at local timeslicing priority *priority*.

inc_migrate(child, node)
    unsigned int   child;

Migrate the incarnation *child* to the node *node*.

B.8 Identification

inc_self()

This returns the caller's incarnation identifier.

inc_where()

This returns the caller's current location.

inc_rdcId()

This returns the caller's RDC identifier.

B.9 Events and Software Interrupts

inc_eventWait(nevents, event_list, timeout)
    int nevents;
    EvDesc   *event_list;
    int timeout;

This call awaits or polls for one or more of a list of events to occur. The types of events of interest are specified using fields in the *nevents* entries of the array *event_list*. Events can concern: a message arriving at a port (EV_PT_HASMSG); a stream becoming unfrozen so that invocations can take place without blocking for this reason, or a call to *inc_asinvoke* completing so another can be made without blocking (EV_SM_INVOK); a stream being closed (EV_SM_CLOSED); or an event associated with some related incarnation – the death of a child, etc. (EV_INC_FAULTING, EV_INC_DEAD, EV_INC_MADE, EV_INC_MIGRATEDONE). Information about these events is returned in the array *event_list*. If *timeout* is negative, the call waits indefinitely for an event to occur. If zero, it returns immediately with information about any events which have occurred. Otherwise it waits for up to the specified positive number of milliseconds for at least one such event.

```
inc_setPortHandler(port, handler, priority)
    int port;
    void        (*handler)();
    int priority;
```

This associates current message and the arrival of future messages at the port *port* with the calling of a software interrupt handler, specified by *handler*. These events are handled with interrupt priority *priority*.

```
inc_setStreamHandler(stream_handle, handler, priority)
    int stream_handle;
    void        (*handler)();
    int priority;
```

This associates the event of closure of the stream to which *stream_handle* refers with the calling of a software interrupt handler specified by *handler*. Such an event is handled with priority *priority*.

```
inc_setIncInterest(set_interest, child, event_type, handler, priority)
    int         set_interest;
    unsigned int      child;
    int         event_type;
    void              (*handler)();
    int         priority;
```

If *set_interest* is SET_INTEREST, this call causes the incarnation *child* (which may be the caller) to notify the caller of events of type *event_type*. The caller can obtain event information either with a call to **inc_eventWait** or through a software interrupt handler. The call associates the occurence of the events with the calling of a software interrupt handler specified by *handler* – if non-NULL – to be run with priority *priority*. If the handler is NULL, any previous handler is no longer effective. If *set_interest* is UNSET_INTEREST, the handler is unregistered and *child* ceases to have notification responsibility.

inc_setInterruptPriority(priority)
    int priority;

This call changes the current interrupt priority to *priority* and returns the old priority value.

B.10 Other Calls

inc_close(reference)
    unsigned int   reference;

This closes the reference whose (local) identifier is *reference*. The identifier becomes invalid.

inc_testReference(reference)
    unsigned int   reference;

This returns RT_NONE if there is no reference with the identifier *reference*. Otherwise it returns the type of the reference in the lowest 8 bits of the returned value, and reference-specific type flags in the upper 24 bits.

inc_rdcExit()

This causes the termination of the caller's RDC.

inc_brk(new)
    int new;

This is used to emulate the Unix brk() and sbrk() system calls.

inc_sleep(nmillisecs)
    int nmillisecs;

This causes the caller to block until *nmillisecs* milliseconds have elapsed.

inc_time(time_buf)
    TimeBuf *time_buf;

This returns in *time_buf* the current user/system/elapsed times for the caller.

inc_pause()

This does nothing except block the caller until a software interrupt occurs, when it returns.