https://theses.gla.ac.uk/

Theses Digitisation:

https://www.gla.ac.uk/myglasgow/research/enlighten/theses/digitisation/

This is a digitised version of the original print thesis.

Copyright and moral rights for this work are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This work cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

Enlighten: Theses
https://theses.gla.ac.uk/
research-enlighten@glasgow.ac.uk

System Support for Object Replication in Distributed Systems

Tor Erlend Fægri

September 30, 1998

ProQuest Number: 10390909

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.

ProQuest 10390909

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

GLASGOW
UNIVERSITY
LIBRARY

11807 (copy 2)

Abstract

Distributed systems are composed of a collection of cooperating but failure prone system components. The number of components in such systems is often large and, despite low probabilities of any particular component failing, the likelihood that there will be at least a small number of failures within the system at a given time is high. Therefore, distributed systems must be able to withstand partial failures. By being resilient to partial failures, a distributed system becomes more able to offer a dependable service and therefore more useful.

Replication is a well known technique used to mask partial failures and increase reliability in distributed computer systems. However, replication management requires sophisticated distributed control algorithms, and is therefore a labour intensive and error prone task. Furthermore, replication is in most cases employed due to applications' non-functional requirements for reliability, as dependability is generally an orthogonal issue to the problem domain of the application. If system level support for replication is provided, the application developer can devote more effort to application specific issues.

Distributed systems are inherently more complex than centralised systems. Encapsulation and abstraction of components and services can be of paramount importance in managing their complexity. The use of object oriented techniques and languages, providing support for encapsulation and abstraction, has made development of distributed systems more manageable. In systems where applications are being developed using object-oriented techniques, system support mechanisms must recognise this, and provide support for the object-oriented approach. The architecture presented exploits object-oriented techniques to improve transparency and to reduce the application programmer involvement required to use the replication mechanisms.

This dissertation describes an approach to implementing system support for object replication, which is distinct from other approaches such as replicated objects in that objects are not specially designed for replication. Additionally, object replication, in contrast to data replication, is a function-shipping approach and deals with the replication of both operations and data.

Object replication is complicated by objects' encapsulation of local state and the arbitrary interaction patterns that may exist among objects. Although fully transparent object replication has not been achieved, my thesis is that partial system support for replication of program-level objects is practicable and assists the development of certain classes of reliable distributed applications. I demonstrate the usefulness of this approach by describing a prototype implementation and showing how it supports the development of an example toy application. To increase their flexibility, the system support mechanisms described are tailorable. The approach adopted in this work is to provide partial support for object replication, relying on some assistance from the application developer to supply application dependent functionality within particular collators for dealing with processing of results from object replicas. Care is taken to make the programming model as simple and concise as possible.

Acknowledgements

I am in great debt and I am very grateful to a number of people for their encouragement, assistance and positive attitude during the period of study towards the M.Sc. in Glasgow. In particular I would like to thank the following:

My first supervisor, Dr. Peter Dickman: Without such a competent supervisor I would surely not have got this far. His firm and sound guidance through the duration of the project, his help with proofreading and his efforts to educate me for research are much appreciated.

My second supervisor, Prof. Malcolm Atkinson: As a secondary supervisor Malcolm filled the rôle as controlling instance. He asked the difficult questions which I had not yet considered important and suggested corrections on draft documents.

Prof. Derek McAuley and Dr. Lewis McKenzie: Discussions about networks, distributed system design, operating systems and a lot of other things.

My flatmate Vidar Hasfjord: Vidar's most important function during the time we shared in Glasgow, apart from being an excellent flatmate, was to let me 'bounce my ideas' off him. Sanity checks are always valuable, and through heated and inspired discussions we normally ended up reaching agreement on principles of object oriented design, object oriented programming languages and operating systems.

My Norwegian friends and e-mail associates Karl Martin Lund, Arne Hatlen and Øyvind Brande: Having access to electronic mail was enough to facilitate numerous and valuable discussions.

Research-associates Karim Dejame, Miguel Mira da Silva, Arthur Serrano and Huw Evans: Without the interesting research environment formed by these people in the department, doing research would have been much more difficult. I received a lot of input into my understanding of computing in general, distributed systems and RPC problems. Special thanks to Huw for valuable proofreading.

And finally to my parents, my brother and my good friends Sandra Cervino, Trond Olav Ronænes, Sissel Rong, Kolbjørn Helland, Dag Sønstebø, Vidar Røren, Bjørn Sundfær, Carmela Battibaglia, Dagrun-Haugen Breirem and Michael Edwards for both being such good friends and for their encouragments through the months of research.

Contents

Chapter 1

Introduction

Distributed systems have become an essential part of modern computing practice. They provide a scalable and adaptable structure on which many useful applications can be built. Applications involving sharing and manipulation of information among large and geographically dispersed groups of people and large process control applications are examples of applications that benefit from distribution and distributed systems.

However, distributed systems are inherently more complex than centralised systems. They must cope with heterogeneity, asynchrony and partial failures, and should also be extendible and scalable. This chapter provides an introduction to the diversity of distributed systems and motivations for their use. Also, some of the numerous challenges facing their developers are presented. Following that, the problem statement underlying this work is given.

1.1 Overview

A distributed system is a collection of cooperating, yet autonomous, computers (called PEs[1]) executing distributed system software. The system software is responsible for low level coordination among the computers and provides a layer upon which distributed applications are built. A component of the systems software is executing on each computer and carries out the task of local control and coordination with other computers in the system. Much like an operating system, distributed system software tries to hide most of the complexity stemming from the underlying system components. Figure 1.1 illustrates this general model of distributed systems.

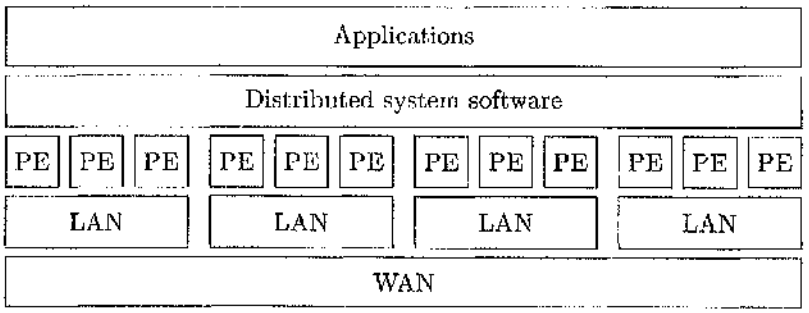

Figure 1.1: An abstract model for distributed systems

[1]Processing Elements. The definition can be found in section 2.2 p. 17.

The basis for any distributed system is a communication network that allows the PEs to exchange messages. The properties of the network are significant factors which influence system performance and the range of suitable applications within the particular system. Hence, the network will also have consequences for the design of distributed system software.

Small scale distributed systems use relatively simple network configurations. A LAN (Local Area Network), perhaps based on a small number of Ethernet segments, may be sufficient to support the necessary applications. LANs are well able to support distributed file systems, client-server databases, electronic mail and CSCW (Computer Supported Cooperative Work) applications for groups of up to several hundred people.

For support of larger and more geographically dispersed workgroups, communication services from public service carrier providers are often used to build WANs (Wide Area Networks). Large workgroups often contain multiple clusters of smaller groups, so they tend to employ internetworks based on a combination of LANs and WANs. In the general model depicted in figure 1.1 the distributed system is built up from a collection of LANs interconnected by a WAN. This hierarchical structuring is commonly used for large systems.

Distribution of computer systems is often recognised as a natural and elegant extension of centralised systems. Today, a large proportion of computing systems used for productive work are interconnected to give their users access to some kinds of networked applications such as shared file systems, client-server databases etc. These applications are often just 'networked' versions of a centralised application. Networked applications are often an extended version of the centralised version, with support for clients located in a network issuing requests to the server using some application specific protocol. A good example of a networked application is a file server which provides a shared repository for users' files.

A distributed system provides a stronger coupling between the computers where several computers cooperate to achieve some common goal. In contrast to client-server systems where servers are 'intelligent' and clients are 'dumb', distributed systems are composed of cooperating *agents*, i.e. computers that take on the role as both client and server thereby using services from other machines and concurrently offering services. A client-server application that employed several coordinated servers would, however, also qualify as a distributed system. As part of this cooperation the computers must maintain global properties such as information about configuration and failures within the system. Distributed systems software is necessary to coordinate all the operation requests and make sure that, for example, transactional properties like isolation and consistency are maintained during concurrent requests. Examples of such applications are distributed databases, multiuser editors and distributed CAD systems [40, 51].

In retrospect, it should be clear that distributed system software is built to coordinate several machines, with the aim of concealing complexity from the applications, providing an abstract and uniform platform for application software development [14]. It should be noted however, that not all applications running in a distributed system need to be distributed. Some applications are non-distributed and do not require the services offered by distributed systems software and are instead built using only services from local software, e.g. operating system software running on each computer.

1.2 Motivation

A distributed system allows for controlled sharing of physically dispersed computer resources, thereby allowing users on networked computers to cooperate on computerised tasks while still maintaining some degree of autonomy. As networks have become more commonplace throughout the computing society, the use of distributed systems is likely to see a significant increase. However, it is useful to investigate the motives and benefits of distributed systems further in order to understand their role in the future.

1.2.1 Inherent distribution and information sharing

Most large applications consist of a collection of nearly separate and physically separated subsystems. The subsystems often benefit from, or might even require, a degree of local administration or autonomy. A distributed system can provide parts of the necessary framework to build such applications.

New applications are made feasible by the availability of geographically distributed interconnected computers. For example, groupware applications, including group discussion databases, task scheduling, and whiteboarding applications, promote easier and more efficient cooperative working by enabling collaboration among large workgroups. Although the transition to groupware systems is not necessarily bringing instant profits to all organisations [160], it seems likely that groupware applications will become very useful as the computerisation of working practices progresses. System architectures that support these classes of applications will hence be valuable. Other classes of applications, for example distributed multimedia applications, distributed databases, electronic mail and distributed information systems can also benefit from system architectures that provide support for distributed coordination across the network. Allocating common functionality in system support layers, available to application developers, reduces the cost of application development. Principles for building system support layers are discussed in more detail in chapter 5.

Additionally, a geographically distributed workforce may justify the distribution of the computer system. By employing a distributed system for coordination between the subsystems, one can obtain a system configuration that more closely matches the structure of the workforce. This can help provide better locality of information, and may increase performance by reduced information access latencies. Furthermore, the physical distribution of computers reduces the probability of all machines failing concurrently. This, in turn, may make the application more available to the user.

1.2.2 Performance

A distributed system contains a number of computers, each with a certain amount of processing capacity, memory and optionally secondary storage. The cost of smaller computer systems has decreased favourably compared to the traditional mainframe and mini computers. A set of relatively fast workstations or PCs is often a more cost effective option than buying mainframe or mini computers supporting the same number of users [151], although the shift towards decentralised computing may incur higher *total* management costs [109] (cf. §1.2.3 p. 8). Also, high performance workstations are better suited to run interactive applications such as windowing systems, graphical presentation packages, database front-ends, spreadsheets and word-processors [51, 106].

Network technology is experiencing a narrowing of the gap between the traditionally fast LAN and the slower WAN networks. Fibre optical communication with extremely high bandwidths is now being employed both for LAN and WAN scale networks. Data communication is now possible at rates reaching gigabits per second, previously only found on specially designed parallel computer interconnects [140]. The availability of high capacity networks has increased the interest in very large scale applications and applications that exploit parallelism of multiple and heterogeneous computers [51]. By distributing tasks among several computers in the network, large gains in performance can be achieved. For example, the task of processing electronic mail within a department might be allocated to a particular workstation, thereby relieving the other computers in the department of this job. Also, many scientific applications require enormous processing capacity, and this demand might be met by, for example, workstations interconnected by high capacity networks [151, 32]. Parts of the application can then be run in parallel, exploiting the processing capacity of multiple workstations concurrently. In practice, relatively poor bandwidth and high communication latencies make it difficult to realise such systems, at least with the current communication infrastructures [123]. Only for certain classes of non-communication intensive applications are the benefits of wide-area parallel computing significant [32]. However, as networking technology evolves, this might become an important platform for demanding applications.

Some applications make copies of the shared data, and allow clients to access a nearby copy. An increase

in performance can be gained from the resulting locality, essentially reducing the access cost to storage and processors. However, the copies of the data must be kept consistent, and this incurs a cost of increased communication. A tradeoff in consistency can be made to reduce the communication but poses a challenge for system designers (see §1.3.3 p. 10). If the shared data can be used independently to a greater extent, the amount of communication necessary is reduced. The tradeoffs incurred are discussed in much more detail in chapter 4.

Also, powerful workstations linked by high bandwidth networks have made applications requiring processing and transmission of time based media such as video and audio feasible. However, such stream-based applications are not considered specifically in this dissertation. The success of these applications appears to be more dependent on appropriate operating system behaviour than on system support mechanisms [51].

1.2.3 Scalability

Dealing with large problems as a collection of smaller, related subproblems, is a well known paradigm in both engineering and science. Large computer systems are extremely complex, and the development of these systems is often simplified (or even made possible) by dividing them into smaller and more easily manageable subsystems. Distribution can be regarded as a mechanism for managing the scale of computer systems. In this respect, distribution deals with both the introduction of multiple management domains and geographical distribution of physical computing resources.

Cost efficient upgrades, and the ability to dynamically adapt the system to the current demand, are important motivations for distributed systems. Because the cost of small and relatively powerful computers is low, they can be added on demand (assuming system growth) thereby extending the system in small, yet affordable, steps. Accordingly, t he effort of local system maintenance and management is reduced. However, building extendible technology is non-trivial and remains a challenge for system designers (cf. §1.3.4 p. 10).

1.2.4 Sharing equipment

In a distributed system, it may be worthwhile to share expensive system resources like printers, scanners or high-capacity file servers. For example, a colour laser printer could be connected to the network and used by a large number of users. Expensive equipment can more easily be economically justified when shared. Such large-grain resource sharing might be the primary motivation for interconnecting the computers. As long as the necessary access structure is present, many resources in the system can be shared. However, sharing of resources raises important issues such as the enforcement of security and access policies (cf. §1.3.6 p. 13).

Other, more low-level resources, like processors and disks may also be shared in the system. However, while large-grain sharing of, for example, printers can be initiated by the users themselves, fine-grained resource sharing requires mechanisms in the system software, e.g. the operating system. Again, the issue of access policies must be addressed. Simultaneously depending on multiple distributed resources within the system decreases the reliability of the application, although replication mechanisms can partially alleviate this problem[2]. Furthermore, fine-grained resource sharing is likely to be more expensive in terms of scheduling overhead and system software complexity than large-grain sharing.

1.2.5 Reliability

Occasionally, distribution of system components is necessary due to an application's reliability requirements. It is very inconvenient if single failures stop the whole system. For example, on-line database

[2]Replication will be discussed in much more detail in the rest of this dissertation.

systems, process control systems and telecommunications systems commonly use redundancy to ensure continuous operation despite failures. Other distributed systems, not designed primarily for fault tolerance may also provide suitable environments for the incorporation of redundancy mechanisms to provide tolerance against failures. Given that copies of important objects can be located and accessed on different computers, a failure in a subset of them may be circumvented, such that the system can use the non-failed objects and continue to provide a service (possibly degraded) during the period of recovery. However, managing replicated components is non-trivial, and poses some difficult challenges (cf. §1.3.2 p. 10).

1.3 Challenges

Developers of distributed systems face several hard problems, e.g. the increasing complexity of software, poor system reliability and limited performance. These problems become more prominent as the systems grow in size, and without careful consideration they will impose severe overheads in terms of cost and performance on the large scale systems that are constructed in the future. This section elaborates on these and some other related problems, and discusses possible ways of addressing them.

1.3.1 Managing application complexity

As more of people's work is being computerised, the demand for more advanced computer systems is strengthened. Additionally, the increasing performance of computing equipment drives the development of applications solving computationally more demanding tasks [1]. Arguably, no limitation exists on the problems that computer systems are being used to solve. Large problems often have elements of distribution, e.g. due to reasons of reliability, scalability, performance or autonomy. Building reliable systems requires careful design and implementation [44], which, in turn, adds to the complexity of developing the software.

A useful paradigm for managing software complexity is that of composition [29, 136, 173]. By decomposing large, complex modules into hierarchies of smaller solvable submodules, very large problems can be handled, and the software is more easily maintainable if decomposed in such a tree-like fashion [187]. In the object oriented model, this decomposition can be even more fine-grained. A single class might contain the implementation of the solution to a sub-problem and collections of classes can be combined into modules which implement solutions to larger grain problems.

A significant benefit of the object oriented model is that it allows classes to reuse code from other classes through inheritance. A single parent class can implement functionality used in a number of child classes to save coding effort. Inheritance will thus result in (arbitrarily high) class hierarchies.

Good object oriented designs favour high class cohesion[3] and a low degree of inter-class coupling[4], which essentially means that a class is responsible for only one, well encapsulated task. A good composition also allows for greater flexibility because a submodule is easily interchangeable, i.e. it can be replaced with another upgraded version without changing the clients of the submodule. The low degree of coupling implies that there are only a limited number of dependencies among modules, and this in turn ensures that the interface of the submodule is moderately sized. The high degree of cohesion ensures that the implementor of the new submodule can focus on one particular problem, and this brings benefits to projects which require collaboration between many development team members.

The object oriented approach is particularly attractive for distributed systems because it can quite naturally be extended to model objects scattered around the network (cf. 2.5 p. 22). In this model, an application becomes a collection of encapsulated objects performing a common task by issuing operations on each other. Encapsulation and abstraction help to reduce the effort needed to understand parts of the system and increase the maintainability of the software.

[3] High functional relatedness [173].

[4] The measure of the strength of association established by a connection from a module to another [29].

Furthermore, due to strong emphasis on abstractions, the object oriented approach can provide good support for reuse. Reuse of designs, for example through the use of design patterns [75, 161], can bring benefits in terms of saved development cost and higher quality implementations. A useful approach to reuse is system support mechanisms that can provide reuse of implemented functionality among many applications. The system support approach to reuse is discussed in more detail in chapter 5.

1.3.2 Preserving system reliability

As computer systems continue to take on many critical tasks in our society, it is important to ensure that these systems are reliable. Distributed systems, often used by large numbers of people, should be the subject of particular attention. They are inherently less reliable than non-distributed systems due to the fact that they depend upon multiple components to work (see chapter 3). Distribution entails a new set of failure modes. Due to physical and electrical distribution the system components often fail independently. This increases the likelihood of a partial failure, but also implies that the probability of all the computers in the system failing simultaneously is low. Mechanisms for fault tolerance are thus essential in distributed systems, in particular systems providing vital services to a large number of users. A main subject of discussion in this dissertation is the mechanisms needed to make distributed systems resilient to failures, for example, chapter 4 is devoted to the techniques used to achieve this.

1.3.3 Distribution transparency

To simplify the task of developing distributed programs, system software should conceal as many of the distribution aspects from the programmer as possible. For example, programmers should not be required to write two versions of an application depending on whether it was going to run on an Ethernet or Token Ring LAN. Systems software should bridge heterogeneity so that applications could be written independently of underlying platform characteristics [14]. Similar ideas form the basis for Java, a portable programming language primarily designed for developing applications for use on the Internet [96]. A Java program is platform neutral, and is compiled into byte-code rather than machine specific instructions. A portable virtual machine executes the byte-code.

Furthermore, there should be uniform methods for accessing system services like file systems, location services or mail services. There are valid arguments against complete uniformity, e.g. reduced performance [194] and limited design freedom, but a conceptually simpler system model is usually worth the overheads.

An exception to the general goal of distribution transparency is related to failures. A programmer will usually want to know where failures occur so that they can be corrected, or at least reported to the user. However, this conflicts with the goal of concealing distribution. Maintaining distribution transparency while providing efficient access to failure status information is a challenging task for designers of distributed system software.

Also related to the challenge of distribution transparency is maintaining consistency. Commonly, distributed systems contain data replicated at several machines to exploit locality of the data and thus gain reduced latency while accessing the data. There is an inherent tradeoff between maintaining data consistency and allowing independent updates of the data. Users should be unaware of the fact that the data is duplicated and should have a consistent view of the data. As replication management is a main theme throughout the dissertation, this issue will be discussed in much more depth in the following text.

1.3.4 Maintaining scalability

A distributed system should be able to scale gracefully, meaning that it should allow for incremental growth and still provide reasonable efficiency. If a system architecture is scalable, the same architecture can be used in system configurations of widely varying sizes, thereby supporting the development of applications across a wide range of systems.

A scalable system architecture must cope with large variations in capacity requirements, and variations by several orders of magnitude should be expected. A small, departmental network might consist of tens of machines, a corporation wide network can contain thousands. The system architecture should therefore impose small overheads, and use available resources efficiently. Designing system architectures that are efficient in both scenarios is non-trivial. Any overhead that is reasonable in the small scale system, might cause overload in the large system. In contrast, an architecture which is efficient for a large scale system is not necessarily efficient for a small system. For example, a distributed coordination protocol that requires concurrent participation from all the computers in the network could be affordable in a small LAN-based system, but would be inappropriate in a large WAN-based network with large communication latencies.

To be scalable, the architecture must also cope with system extensions. If new components cannot easily be integrated into the system, scalability will suffer. For example, poor interoperability between system components from different manufacturers will reduce the system's extendibility and hence make it difficult to scale the system according to the requirements. (cf. paragraph "Managing heterogeneity" p. 13). Although not all distributed systems should be expected to reach the size of large corporate networks, it is difficult to predict in advance how large the system will grow, and if scalable technology is used throughout, incremental growth and efficient use of available resources is ensured.

1.3.5 Maintaining performance

Distributed systems should, like any other computer system, use the available resources efficiently so as to give good performance. However, achieving this in distributed systems is hard. Motivated by prospects of reduced application complexity, systems designers have advocated uniformity of mechanism and concept. However, providing uniformity often involves adding several layers of software which reduces system performance, and while processor speeds are currently doubling every two years, the benefits can easily be outweighed by layers of software bridging the heterogeneity of the hardware [185]. The challenge becomes to build well designed software architectures which minimise performance overheads.

Additionally, the communication infrastructure has traditionally been the bottleneck for performance in distributed systems. However, the arrival of high performance networking technology suitable for both local and wide area communication, has generated increased confidence in distributed systems as an attractive platform for many useful applications [14, 32, 108, 164, 170].

1.3.6 Other issues

The previously mentioned problems are the main focus of this dissertation. A number of other important related problems are discussed in this section. These problems are not directly addressed in this dissertation; they are outside the scope of this work. However, they are important issues for distributed systems designers and will influence the implementation of real systems.

Extensibility
Extensibility is often rather limited in traditional centralised computer systems. There are usually definite constraints both as to how new components can be added and which new components can be used. Both in terms of structure and allowed heterogeneity, there are strict rules confining the process of adapting the system to changing requirements. These constraints are typically imposed by manufacturers, leading to additional difficulties when equipment from different manufacturers has to be integrated.

Distributed systems tend to be more extendible than traditional centralised computer systems. An inherently loosely coupled distributed system is able to accommodate additions of new equipment more easily than a system based on a mainframe or minicomputer. Furthermore, a large proportion of networked computers run variations of UNIX. Uniformity of operating system platforms increases the extensibility of the distributed system.

Exploiting parallelism
Ideally, a distributed system architecture should give the programmer transparent access to all available computing resources such that applications could be written independently of the number of available processors. One approach, called the *processor pool* model, is the basis for several distributed systems, e.g. the Cambridge Distributed Computing System [152] and the Amoeba distributed system [133]. These systems model processing power as a globally available resource shared between applications. However, the majority of distributed applications today are partitioned explicitly, and are designed to make use of a particular number of processors.

Being able to exploit the available resources like processors, memories and disks efficiently, concurrently and transparently is not trivial. Problems such as load balancing and process migration are the focus of much research interest (see for example [62]). While research into parallel architectures has experienced significant progress, both in hardware architectures and in programming languages and tools (see for example [146]), there are however, many problems which still remain unsolved, most importantly is the tight coupling of programs to specific architectures, essentially rendering efficient parallel programs non-portable [168]. The end result is that applications aimed at exploiting parallelism are often required to make strict assumptions about the system architecture, and they are usually unable to cope with the heterogeneity found in traditional distributed systems. Any progress made in this research area is likely to have a big impact on the kind of applications people will use in distributed systems.

Also, despite the narrowing gaps in offered network capacity between traditional high-speed processor buses and networks, distributed systems still have to cope with inherent propagation delays in long haul communication links. Additional delays are imposed by the layers of communication software needed to bridge different networking and machine architectures. Arguably, the rapid increases in processor speeds are not matched by similar decreases in transmission latencies. This problem is present even in high performance multicomputer networks. Techniques such as caching and batching may amortise the latency cost over several requests, but for highly interactive and communication demanding programs the savings are limited. Consequently, the previously clearly distinct fields of parallel computing and distributed computing are becoming blurred [70].

An idealistic goal of distributed systems designers is to hide this heterogeneity and complexity, with the intention of giving users the illusion of a less complex uniprocessor system [51, 183]. Clearly, this is a major undertaking, but can produce systems which are easier to use.

Load balancing
Related to the problem of exploiting parallel execution of programs (cf. §1.3.5) and scalability, is the problem of load balancing or load sharing. In addition to reducing the scalability of the system, improper allocation of load among the computers will severely reduce performance. For example, it has been shown that significant amounts of processing capacity is wasted in networks of workstations [62]. In the extreme, load imbalance can reduce the availability of the system if certain important computers are overloaded with work. In addition to reduced availability, load imbalance can also reduce the system reliability. Overloaded machines are more likely to fail [105, 165], and overloaded networks are more likely to experience congestion and increased delays and jitter of data transmissions. Ideally, a load balancing scheme should allocate load evenly and dynamically among available computers. A key problem in load-balancing is to define what load is, i.e. the measure of cost. Many factors influence system performance, e.g. application memory requirements and the ratio of I/O versus computation. Optimal load balancing is a hard problem and most approaches to load-balancing assume relatively simple cost measures, e.g. the number of processes scheduled on a computer.

Further complicating the issue is the fact that many of the properties that determine efficient load are dynamic, and can change very rapidly with time. For example, spare capacity on a particular communication link might be large outside office hours, and relatively limited during office hours. However, significant variations can occur within much shorter time frames. Load balancing is not a concern of applications, it is a task that should be performed by system software. These kinds of problems are outside the scope of this work (cf. §2.7.1 p. 25). The reader might refer to chapter 15 in [131] and chapter 11 in [169] for more information.

Managing heterogeneity

Scalable distributed systems are often populated with heterogeneous components. It is therefore advantageous to integrate these components into a single, uniform framework to reduce the effort needed to access the various components. During extensions of a system, problems of interoperability often arise. Different components have different interfaces, and it can be a challenging task to provide cooperation across non-uniform platforms.

Heterogeneity originates at many different levels in the system hierarchy. Different processors can use different instruction sets. Different computers have various amounts of memory and disk space. There may be different operating systems installed together with various other kinds of system software like communication protocols and file systems. Without some kind of bridging software framework, applications would have to be written specifically for each particular machine.

Despite the heterogeneity, a distributed system must utilise the resources efficiently. This implies that knowledge about the properties of the resources must be available to the system, such that the system can determine a good utilisation strategy for each resource, e.g. processing power, memory and special hardware. For example, one particular computer may have special capabilities for numerical calculations, so a particular class of application that require a large amount of numerical calculations should be executed on that computer. Performing this kind of optimisation requires process migration mechanisms, another non-trivial problem (see e.g. [178]).

Data communication frameworks

The variety of hardware and software architectures that must be integrated can be very large. To achieve interoperability between different architectures, there must be a standard which defines a common structure for interaction. Traditionally, only relatively low level communication protocols were available. Among the most widely used, TCP/IP has been implemented on a range of platforms, and is hence providing interoperability between these platforms. While originally a 'UNIX only' protocol, it is now used on a wide range of computing platforms. Although a communication protocol suite is not sufficient for application level interoperability, a number of applications, for example World Wide Web browsers, file transfer, terminal emulation, and electronic mail are based on this protocol suite.

The OSI[5] framework reference model, was created in collaboration by ITU[6] and ISO[7] [153]. OSI attempts to provide a more complete framework for application interoperability than TCP/IP. The reference model is composed of seven layers, where the four lowest layers together provide similar functionality to a TCP/IP stack (peer to peer reliable data transfer). OSI uses the three upper layers, the session, presentation, and application layer, to add functionality for application interoperability. For example, in the application layer, OSI defines several ASEs[8] for direct use by applications. There are 'low level' ASEs for remote procedure calls, association management (an association is equivalent with a connection) and reliable data streams. Additionally, a set of 'high level' ASEs for directory services (X.500), mail services (X.400), file transfer (FTAM) and remote terminal emulation (VT) are specified. While the OSI model was a major undertaking, it has failed to reach wide acceptance in the computing community. This dissertation will not attempt to provide an answer for this, however, experts within communications research have indicated the severe overheads of inband communication as an important factor [122].

System threats

A distributed system is inherently less secure than a centralised system because the multiple components are each a potential threat to the security of the system. However, distribution of resources can also be a benefit because it normally requires more effort to tamper with all the components. If information

[5] Open Systems Interconnection.

[6] The International Telecommunication Union, formerly CCITT (International Telephone and Telegraph Consultative Committee).

[7] International Organization for Standardization.

[8] Application Service Elements.

is partitioned among several machines in a network, everything is not necessarily compromised from one machine. Additionally, heterogeneity among machines and interconnections will make it even more cumbersome to access all machines.

Issues such as encryption, authentication and identification need to be addressed to provide system wide security against attacks. However, there are also other, less obvious threats that have to be considered. For example, flaws in the design of distributed systems might lead to resource overloads and network congestion. This might severely reduce access to the system, essentially causing denial of service. It is therefore important that during the design and development of such systems, consideration is given not only to preventing direct attacks, but also to preventing some users maliciously or otherwise, limiting other users from accessing the system.

Correctness
Constructing correct computer systems is a significant challenge for both researchers and practitioners. Ensuring correctness is a hard problem in sequential systems, in distributed systems it is even harder due to added complexities such as heterogeneity, failures and asynchrony. Global coordination and administration requires access to some shared state, a globally valid property. Distributed consensus algorithms are able to achieve agreement on global properties [66], but they are normally expensive and complicated[9] due to failures and large communication latencies. Large distributed systems are often required to deal with incorrect or incomplete global state because of the high overheads incurred by traditional consensus algorithms, which further complicates their implementation.

1.4 Replication in Distributed Systems

Distributed systems offer poor availability if they are not designed to withstand partial failures. Replication is a recognised approach to increasing resilience against partial failures, but requires sophisticated data management to maintain consistency. With the increasing complexity of software, there is a demand for more system support software to keep application complexity under control. System support for replication can help developers of distributed applications attain suitable reliability without significantly increasing application complexity and therefore also application cost[10]. A system support facility provides generic abstractions that are applicable to a range of applications, and hence relieves the programmer from the task of reimplementing replication scheme code in multiple applications. The goal of this work is to present a usable approach to system supported object replication, and a proposal for such an approach is presented in chapter 6.

The fundamental issue for all replication schemes is the level of consistency offered. Strong consistency replication management schemes attempt to maintain full consistency among the replicas, thus offering a one-copy model of the replica group. But depending on the kinds of failure in the system, full consistency is not always attainable.

Weaker consistency replication management schemes achieve better availability, performance and scalability than full consistency schemes and are necessary for large scale distributed applications where full consistency is not practicable. However, due to the potential for inconsistencies among the replicas, weak consistency schemes are not appropriate as part of system support mechanisms as they require application specific intervention to sort out conflicting replicas. Thus, the application would have to include replication aware code which contradicts the aim of reducing application complexity.

In contrast, a full consistency scheme can be used without changing the semantics of the application as most programs are written under the assumption that there exists only a single copy of data items. Although weak consistency protocols have been used to support semantically simple applications where

[9] Worse still, consensus has been proven impossible in many realistic system models [69, 184, 186].

[10] Many markets are not willing to pay much extra for increased reliability of their applications [98]. Increased application reliability will however be a bonus if added at a small cost.

the rules for conflict resolution are straightforward, they are not very useful at a system support level in object oriented programming systems where reconciliation of objects generally cannot be automated. Only for some applications which are generally able to cope with inconsistent data due to their self-correcting characteristics, for example name resolution using a name server or reference databases [81, 134, 137], can weak consistency protocols be used transparently. The application is responsible for checking if the information is out of date, and if so, the application must be able to detect the error and guard itself by using a failure masking protocol such as retries.

Consequently, a full consistency replication scheme seems most suitable for system supported replication and forms the basis for the proposed system architecture.

1.5 Problem Statement

A range of challenges for distributed systems designers have been presented in the previous text, and a number of others probably exist. The problems discussed require substantial and continuous research, they are all important issues. The work described in this dissertation tries to address only a particular problem within the area of distributed computing; namely that of providing assistance to application builders developing reliable distributed software systems in an effort to help reduce application complexity and improve its reliability.

The aim of the work presented in this dissertation is to provide partial system support for object replication in a distributed system. Due to the inherent tradeoff between consistency and scalability, the system support mechanisms are aimed at supporting relatively small scale applications where high levels of consistency only incur moderate costs in performance. The small scale justifies the use of remote object references for sharing of objects between machines. However, the system support mechanisms allow the programmer to partially control the synchronisation of the replicas to better suit the particular application in hand. Through the use of the system support mechanisms it will be demonstrated that applications can employ replication in a simple and efficient manner.

My thesis, which will be supported by this dissertation, is that partial system support for replication of program-level objects is practicable and assists the development of reliable distributed object-oriented applications which require full consistency replication. I demonstrate the usefulness of this approach by describing a prototype implementation and showing how it supports the development of an example application.

1.6 Outline of the Dissertation

The remainder of the dissertation is composed of 9 chapters and 2 appendices; the first four (chapters 2–5) present the problem area, the next three (chapters 6–8) present my proposed architecture for system supported object replication, chapter 9 is a survey of related work, and chapter 10 contains concluding remarks about the achievements and open problems. Appendix A presents a sample collator which is part of the programming model, and appendix B presents a small amount of probability theory used throughout the dissertation. What follows is a more detailed description of each chapter.

Chapter 2: System Model. This dissertation is concerned with particular classes of distributed systems. The system model defines the characteristics of these systems by describing their structure and behaviour, covering both hardware and software issues.

Chapter 3: Computer System Failures. This chapter provides a presentation of some of the many failures that can occur in computer systems. Distributed systems are particularly vulnerable to failures, and must be designed to withstand them if they are to be useful. However, understanding the characteristics of these failures is necessary when attempting to build systems that should

withstand them. This chapter is to be regarded as an introduction to chapter 4 which covers techniques to mask these failures.

Chapter 4: Replication Techniques. Replication management schemes can be used to mask failures in distributed systems and a range of different approaches to replication do exist, of which a number are presented in this chapter. Additionally, replication in object systems, through the use of object replication, is distinct from traditional data replication techniques. The particular system model adopted in this dissertation, and the range of failures considered here, requires that important tradeoffs be made consciously when a replication scheme is chosen. Included in this chapter is a discussion of these tradeoffs and special considerations that must be made in an object replication scheme.

Chapter 5: System Support. Implementing system support is not trivial, but the availability of system support can be crucial for the construction of complex software systems. This chapter contains a discussion of various important issues that must be addressed during the development of system support mechanisms with an emphasis on those issues related to the provision of system support in distributed systems.

Chapter 6: System Architecture. My proposed architecture for system supported replication is presented in this chapter, highlighting its modular and flexible design.

Chapter 7: Programming Model. A main goal of the architecture described in the previous chapter is to present the developer with a simple programming model. In this chapter I show how my architecture extends an object oriented programming language with powerful mechanisms for managing object replication.

Chapter 8: Realising the Architecture. The architecture has been partially implemented as a prototype in Modula-3. Additionally, a toy application has been built exercising the system support mechanisms, and demonstrates the simple programming model. The application has also been instrumented for performance measurements. This chapter presents the prototype to illustrate how the architecture can be realised. A brief discussion of the application and the performance measurements is also included.

Chapter 9: Related Work. Vast numbers of research and commercial projects employ replication techniques to improve failure-resilience, availability or performance of applications. This chapter is divided in two; the first part focuses on those projects particularly aimed at providing programming level support for replication, such as replicated RPC or process groups. The second part contains a broader presentation of distributed applications that employ replication techniques.

Chapter 10: Conclusions. A range of valuable insights have been gained throughout the course of the project. The final chapter summarises these insights in a discussion of the limitations of the architecture, open problems and possibilities for future research.

Chapter 2

System Model

The system support mechanisms presented in this dissertation (see chapter 6) are built upon existing technology to reduce complexity and simplify their development. This chapter presents an abstract model of the distributed systems in which my proposed approach for supporting replication is appropriate. If similar technology is not available, implementing the replication mechanisms may not be practicable without reworking the architecture. The assumptions set forth in this chapter should therefore be considered prerequisites for the proposed architecture.

2.1 Overview

A distributed system is a collection of autonomous and cooperating computers which communicate via a network. A network of workstations is a good example of such a system which conforms to the system model presented in this chapter. Distributed software composed of cooperating modules execute within the network. An object-based programming model is chosen for this work, where objects are distributed among the computers in the system and interact by invoking methods on each other. Objects provide a simple and unifying concept used to decompose distributed applications into a collection of interacting, autonomous and maintainable components. Objects are convenient for the encapsulation of complex software mechanisms. Encapsulation and simplicity make objects useful for the construction of large distributed software systems [135].

In contrast to a centralised system, a distributed system must cope with a range of complicated problems such as asynchrony and partial failures. Also, distributed systems may often include heterogeneous components. The heterogeneity introduces variations in the underlying hardware and software architectures which must be concealed by distributed systems software. The following sections present the system model in detail.

2.2 Processing Elements

A collection of *processing elements* (PEs) cooperate to execute programs within the distributed system. Each PE has direct access to a limited amount of memory, and optionally, a limited amount of non volatile storage. Access to local memory is assumed to be fast, access to non volatile storage is assumed to be orders of magnitude slower[1].

[1]The cost of accessing non-volatile storage, unique in its ability to maintain the integrity of data during PE failures, can be amortised by employing techniques such as caching in combination with specially designed, failure resilient write-back policies. However, it is outside the scope of this dissertation to provide a thorough analysis of cache performance. For example, cache performance is found to be highly dependent on program behaviour patterns, a distinct field of research

A loosely coupled system architecture is assumed — each PE executes locally stored programs. The local memory, optionally augmented with a virtual memory mechanism, is partitioned into multiple address spaces. Each PE supports the concurrent execution of multiple, potentially multi-threaded, programs located in their own virtual address spaces. Due to variations in processing capacity, system resources and system load, PEs execute programs at variable and unpredictable rates. Each address space is protected against uncontrolled access from other programs by local operating system software. An attempt by a program in one address space to manipulate data within another address space without the appropriate access privileges will either be denied or cause a crash in the offending address space.

Potential architectural heterogeneity among PEs can cause problems during interaction. Differently sized address spaces and different rules for byte ordering will undoubtedly cause mishaps if an interconnectivity policy is not in place. However, these issues are assumed to be solved by existing systems software implementing inter-PE communication primitives.

A distributed program contains instructions both for local computation and for communication. A PE executing a communication instruction uses the facilities offered by the network to communicate messages with other PEs in the system. Each PE is uniquely identified within the network, and the communication network provides the necessary support for communication of messages between any two PEs. Hence, PEs are assumed not to be concerned with issues such as PE addressing and network routing. These issues are dealt with by lower level communication software.

Distributed shared memory (DSM) systems are not considered here. While DSM is a very powerful abstraction which potentially can simplify application development, current DSM systems tend to offer poor scalability and efficiency as compared to distributed memory systems. Because DSM systems do not support application partitioning (cf. §2.7.1 p. 25), the notion of failures is concealed, and this makes it difficult to build fault-tolerant systems. As in replication management schemes for distributed memory systems, maintaining consistency is the difficult issue. However, experiments with DSM systems can give valuable input into replication management strategies in distributed memory systems [36].

2.2.1 PE failures

Computing machinery is not able to sustain continuous failure free operation for arbitrarily long periods of time (cf. §3.2 p. 30). A PE may fail during execution of local programs and may trigger failures in other PEs. Normally, a failure will only affect the address space hosting the executing process. However, if the PE failure occurs during execution of critical code, e.g. operating system code, device driver code, etc., all activity on the PE may be affected, i.e. all address spaces local to the PE may fail. Also, it is assumed that a PE does not fail maliciously, i.e. the PE does not behave arbitrarily. Rather, when a failure occurs within the PE proper, it crashes and stops all processing permanently[2]. Some time after the PE has crashed, the PE may be restarted, most likely initiated by a human operator. Because a PE fails by crashing, such a failure can be detected in the time-domain by a timeout mechanism (cf. §3.2.3 p. 32).

2.3 Networks

Computer networks provide the necessary infrastructure for communication among PEs. The networks provide support for any pair[3] of PEs to communicate messages. Message passing is by definition not instantaneous. It is not possible to send data from a source to a destination in zero time. Additional delays are incurred by unpredictable traffic and congestion patterns throughout the network, leading to arbitrarily long delays.

[120, 56].

[2]Failures are discussed in more detail in chapter 3.

[3]Collection of PEs if the network supports multicast (cf. §2.3.3 p. 20).

The communication infrastructure is usually the most significant factor determining the characteristics of a distributed system. Most importantly, its structure, or topology, affects both performance, scalability, and failure modes in the system. A connected network is assumed, i.e. a PE can exchange messages with any other PE. However, not all PEs are directly connected by a single transmission path. A fully connected network is not feasible in practice due to the high cost; however, high connectivity[4] can improve the reliability of the network, as fault-tolerant routing algorithms can ensure that messages are routed around failed links and therefore provide service in the presence of link failures.

Furthermore, bus-based networks, while able to support efficient broadcast, do not scale to any significant sizes[5]. Accordingly, a network structured as a collection of broadcast subnetworks (LANs) interconnected by point-to-point long-haul networks (WANs) is assumed.

2.3.1 The latency problem

Routing, buffering and forwarding of messages in large networks incur overheads which are reflected in relatively long and unpredictable latencies. Additionally, significant latencies are present in large distributed systems due to the physical propagation delays in long distance communication links. Electrical and optical signals are inherently limited by the speed of light, and in many cases these signals travel at significantly slower speeds. For example, due to refraction, propagation speeds decrease to about 60% of the speed of light in optical fibres. As a result, a coast to coast connection in the continental U.S. can experience propagation delays of up to 30ms [141]. The problems incurred by physical signal propagation delays are naturally amplified by the geographical scale of the distributed system. Techniques such as caching and buffering can be used to amortise the propagation cost over multiple messages, but this does not bring much benefit to highly interactive applications which are dependent on rapid transmission of round-trip messages.

Timeouts
A timeout is a mechanism for dealing with the asynchronous behaviour of distributed systems. A timeout is an approach that introduces synchrony constraints into the communication channel to deal with benign failures. If a message is not received within a specified time interval, a timeout expires and the message is assumed to be lost. Timeouts essentially reduce timing failures to omission failures[6], making it possible to observe omission failures without arbitrary long delays.

A problem with timeouts is to find a timeout value which is efficient. Timeout values which are too small will lead to excessive numbers of timeouts, whereas overlarge timeout values will make the system inefficient by waiting too long before declaring a message as lost. Further complicating this issue is the greatly varying latencies found in internetworks combining LAN and WAN technology. The latencies also vary depending on competition for the channel. This dissertation assumes that the network technology has appropriate mechanisms for dealing with timeouts across heterogeneous networks such that communication among interconnected PEs is performed efficiently. Therefore, with high probability, failures are detected much faster within a subnetwork than in a long-haul WAN network.

2.3.2 Network failures

A network should allow PEs to exchange messages reliably. However, several kinds of mishaps are likely. Network failures are inherently less independent than PE failures. A network connects several PEs together, and a failure therefore normally affects multiple other components. Depending on the network's topology, architecture and population, there are large variations in the number of affected

[4]The connectivity of a network is the number of links that must be removed to obtain a single-connected network.

[5]This is due to the need for collision detection algorithms which limits the length of the bus. For example, an Ethernet segment cannot exceed 2.5km in length [182].

[6]Cf. §3.2.3 p. 32.

components. Highly connected networks with multiple alternative transmission paths can substantially reduce the effect of failures. Other networks, e.g. bus-based networks may cause disruption for many of the connected PEs.

Lost messages may cause PEs to observe *partition failures*, i.e. a group of PEs are not able to communicate with another group of PEs. Partition failures are extremely hard to deal with because PEs in each group might conclude that the PEs in the other group are just faulty. This, in turn, might lead to inconsistent behaviour within each group. In the proposed architecture, partition failures are treated pessimistically, i.e. only a partition containing a majority of the PEs is allowed to make progress (see §6.3.1 p. 68). Other solutions, assuming (optimistically) that conflicts are rare and can be dealt with later, would allow PEs in multiple partitions to continue [55].

Network failures are often transient and cause bursts of corrupted or lost messages rather than permanent partitions (cf. §3.2.4 p. 34). To increase the reliability of the network, failure resilient communication protocols are used to conceal many of these mishaps. For example, the TCP/IP protocol suite provides a reliable byte stream transport service over a virtual circuit [47]. The TCP/IP protocol guarantees that no bytes are reordered, duplicated or corrupted. Due to the connection oriented semantics and retransmission of lost data, TCP/IP connections have crash fault semantics and are failfast and reliable[7] (see also §3.2.3 p. 32).

2.3.3 Other network issues

Some network architectures have specific capabilities and strengths that can be of significant benefit in distributed systems. Among these are support for sessions (e.g. connections), broadcast or multicast, service guarantees, encryption and authentication. Although these features can be implemented in software, hardware support is likely to be much faster.

Network support for multicast
Replication involves keeping several copies of an object up to date. This can be supported by multicast network primitives. Multicast primitives allows a PE to send an update message to multiple recipients using a single network operation. Most bus-based networks, e.g. the Ethernet, but also ring-based networks such as FDDI and Token Ring support efficient broadcast. In these networks, multicast is similarly efficient; because all messages are seen by all stations, a station can just discard messages that are not from a transmitter in the multicast group. Network support for multicast is of benefit also because it reduces the amount of traffic on the network by making copies of the message only when strictly necessary. Traditionally, multicast in wide-area networks has been much more expensive, but new network architectures, such as ATM [190], and research into multicast on the Internet, such as the MBone [63], may reduce this problem. However, the proposed architecture does not require such support.

Isochronous datatransfer
Time critical media like video and audio require transfer of large quantities of data with little jitter and delay. Some network architectures, such as ATM [190] provide support for isochronous data transfer. Work is also being done to improve the performance of FDDI [43] and Ethernet [74] for time-based media. However, challenges still remain. Isochronous Ethernet has the scalability problems of Ethernet and will be most suitable for small scale installations. FDDI technology also has scale limitations, maximum network length is 200km. Communication latency in ATM networks is still a bottleneck for highly interactive applications [108], although promising progress has been reported [192]. For truly high speed networking, processing overhead in clients seems to be the bottleneck [108], work will be focused on improving device drivers and medium access protocols. Latencies below $200\mu s$ have been achieved in ATM LANs. Any efforts resulting in networks with less jitter and delay will be of benefit to the proposed system architecture.

[7] The connection is failfast because timeouts and checksums convert late or corrupted messages to lost messages, and is reliable because it retransmits lost messages [85].

2.4 Objects

A class implements an abstract data type (ADT) defined in a program, and is purely a programming language concept. A class may be a specialisation of some other class, in which case it inherits parts of its definition from the other class, or it may be a top-level (root) class. When a class is instantiated to construct an object, the object will contain all the fields and methods accumulated down the inheritance hierarchy, and the object will accept method invocations as declared in the ADT specification.

This dissertation is primarily concerned with the object concept. Issues related to class concepts, e.g. subtyping and polymorphism are not further considered[8]. It is assumed, in accordance with the traditional view of object-orientation, that data abstractions and procedures are first-class objects which can be manipulated as normal values [34].

An object is a structure that encapsulates a state and a set of methods (operations) that can be invoked to manipulate that state. A method is a non-instantaneous parameterised transformation of an object's current state [119]. Invoking a method on an object is the only mechanism available to other objects for accessing an object's state[9].

Objects exist during run-time in an application's address space, and the system support mechanisms described here are primarily concerned with programming language objects, i.e. objects instantiated by a program generated by a compiler. If used in the context of object-oriented operating systems, the same definition of objects would apply. However, other objects, such as traditional operating system objects (files, ports and processes) and hardware objects (displays, keyboards, disks etc.) are not part of the architecture.

2.4.1 Semantics of methods

A pure, encapsulated, object model is adopted in this dissertation. Method invocations on objects are assumed to potentially mutate the state of the object. Hence, the object's new state S' is a function of both the method m, any parameters p, and the state of the object S before the invocation occurred, $S' = f(S, m, p)$. Methods that do cause mutation are called *non-idempotent*, or *non-testable* [85]. The number of times such methods are invoked determines the final state of the object, hence, they must be executed exactly the number of times specified by the client. For example, invoking a method *deposit(£100)* on a bank account is non-idempotent, because it does not simply overwrite the object's internal state, but rather depends on the previous state to determine a new value (in this case the current balance). It is further assumed that methods are *non-commuting* and must be executed in the correct order. For example, the order of invocation is important for the two method calls *addInterest(10%)* and *deposit(£100)*.

Whether the invocation of a method on an object only reads the object's state or if it is also modified is not revealed to the holder of the reference to the object. An encapsulated object model means that the implementor of the object can guarantee that internal state invariants can be maintained. This relieves the client of the object from any obligations to deal with integrity constraints of concern only to the object itself and this in turn enhances the scalability of software designs.

Methods may define output parameters as well as input parameters. Input parameters are used to parameterise the method invocation, and output parameters return results of the invocation back to the caller.

Object's state
An object's state may contain any type, variable or procedure definitions allowed by the programming

[8] In this respect, the object model is *object-based* [135, 154]. However, the implementation of the architecture benefits from object-oriented features of the implementation language, which is *object-oriented*.

[9] Some object oriented languages do not enforce such strong encapsulation.

language and hence form arbitrarily complex constructs. For example, objects may contain dynamic data-structures such as references to files, monitors and threads [33]. Objects may also hold references to other, potentially distributed, objects[10] (cf. §2.5 p. 22).

During execution of a method invocation, the object might, in addition to performing computations on the local state, invoke methods on some of the referenced objects. Methods might, as input or result parameters, accept references. These references are just like any other reference, and the holder of the reference can use it to send invocations to the referenced object.

2.4.2 Concurrency issues

In the distributed systems considered here, multiple objects may concurrently invoke methods on a shared object and cause non-deterministic program behaviour. To prevent this problem, invocations must be serialised, using locks, semaphores or monitors. It is the responsibility of each object to ensure that multiple executing methods within the object do not cause incorrect state changes. The proposed architecture acknowledges the need for serialisation using built-in synchronisation primitives (see §6.2.4 p. 66).

The objects considered here are not active, i.e. there is no explicit coupling of objects and threads. A thread may visit arbitrarily many (local[11]) objects and an object may be visited by arbitrarily many threads. However, an active object model would also be suitable for the architecture, and would most likely reduce the complexity of the parallel RPC mechanism described in §6.2.2 p. 64.

2.5 References

A reference is a *handle* to a particular object, and is created when the object is instantiated. In a distributed system, references might span address spaces; the reference must then uniquely identify any object in any of the address spaces. Further, multiple objects can hold the reference to a particular object, facilitating sharing of the object [53].

Uniformity of references
Uniform references, i.e. indistinguishable local and remote references, have been the subject of some debate [26, 117, 194]. Uniformity is advocated as an approach to reduce application complexity. However, there is an inherent difference; dereferencing a remote reference may fail while this will never occur for a local reference. Remote references are therefore less reliable than local references. Furthermore, invoking a method on a remote object is more costly. If references are truly uniform, the programmer has no choice but to use local and remote objects in the same manner, thereby sacrificing either efficiency or reliability of the software.

This dissertation is based on an object model which makes it possible for the programmer to handle remote and local references differently through optional exception handlers for remote invocations. In case there is no exception handler for remote object invocation failures, the compiler will issue warnings. The benefit of this approach is that the programmer is only *reminded* about the additional failure modes of remote object invocations but *is not required* to handle these failures if the application can ignore them[12].

[10] Objects are never 'contained' within another object, nor are they 'owned' by another object; all objects exist independently in an universe of uniquely identifiable objects.

[11] Naturally, remote invocations will be processed by another thread in the remote address space.

[12] Any reliable application should be concerned with such exceptions however, and should not simply crash due to a remote object failure. Not adding exception handling for remote references should therefore be considered a dangerous programming practice.

Security issues
In a naïve implementation of a distributed object system, any holder of a valid object reference can invoke methods on that particular object. This opens up the possibility of security threats in the system, where arbitrary programs can manipulate objects. A solution to this problem is presented by Geihs et. al. where an authentication mechanism is integrated into object references [78]. The problem with such an approach is that it is likely to be very expensive, particularly in systems containing large numbers of mostly small objects. In such a system, an authentication check for each method invocation would incur severe performance overheads.

However, such security measures are neither assumed nor required by the architecture but could be used if present. The present version of the architecture assumes sharing of objects among programs residing in address spaces owned by the same user and protected by underlying system software, although extensions of the architecture might have to consider protection of object references.

Reference failures
A remote object reference is fragile. If the referenced object becomes unreachable, e.g. due to a network failure, the client will be notified by an 'object unavailable' exception. This is a problem for both the caller and the callee, and the failure of the client to be prepared for such events will most likely cause the client address space to crash. If the reference is remote, the remote address space might crash as well due to the execution of arbitrary instructions. Similar failures will also occur if the remote object is removed without updating the references that refer to it. Such 'dangling' references may cause failure in objects trying to dereference them (i.e. invoke a method on the referenced object), and potentially cause the execution of arbitrary instructions in the remote address space leading to remote address space failure as well. Dangling references might occur for several reasons, e.g. erroneous object migration or premature garbage collection [143].

The reference is the only mechanism available to invoke methods on another object. In fact, if no references to an object exist, the object is not reachable, and does not logically exist. Such objects are removed and their storage reclaimed by garbage collection technology [143].

2.6 Invocations

An object holding a reference to another object can invoke methods on the referenced object. Invoking methods on a local object is performed through a standard, local procedure call on the indicated object. The control is transferred to the method in the referenced object and if return parameters are specified for the method the caller waits until the method is completed. Invoking a method on a remote object requires transfer of control and data between address spaces. The calling thread is blocked before a remote thread starts executing the call in the remote object. The calling thread resumes execution when the call returns from the remote object. Issues such as locating the remote object, argument marshalling and unmarshalling, communication failures and remote object failures are handled by an object-oriented RPC mechanism[13].

Traditionally, in non-object based systems, the RPC [23] (remote procedure call) approach has been used for intra-address space procedure calls. In object based systems, RPC is quite naturally extended to *remote invocations*. Whereas a remote address space identifier must be supplied with each RPC call to identify the callee, a remote reference is sufficient identification of the callee in an object system [22, 27, 154]. This increases the uniformity of local and remote invocations.

Invocation failures
Invoking a method on an object can be regarded as equivalent to sending a message to the object. If the

[13] An object-oriented RPC mechanism extends the notion of a reference to include remote references. In contrast to non-object oriented RPC mechanisms which require a process identifier as parameter with each remote call, an object-oriented RPC hides the process-id within the object reference.

method defines return parameters the caller is blocked while waiting for the reply message, otherwise the caller proceeds. Due to the possibility of network failures, messages cannot be transmitted with complete reliability. The network may cause arbitrary delay of messages, due to disconnections or protocol failures. An invocation on a remote object may therefore block the caller until the timeout set for the return message expires. If the caller receives a timeout, it cannot accurately verify that the method has been executed at all. However, there is no exact way of deciding what went wrong; either the invocation message was lost, the remote object's address space crashed, the remote PE was too busy to respond in time, or the return message was lost.

Due to this uncertainty, remote invocations can only provide at-most-once semantics in unreliable asynchronous systems. The caller cannot accurately determine whether the invocation was executed one or zero times if a reply is not expected or expected but not received. However, if return messages are expected, at-least-once semantics can be achieved by retrying the invocation until a reply message is eventually received. This causes problems in the adopted system model, as methods may be non-idempotent and thus require exactly-once invocations.

The architecture assumes that network failures are rare, and occur mostly as transient failures which are masked by underlying communication protocols[14]. Also, it is assumed that a timeout mechanism reports untimely message arrivals. A client will therefore observe all invocation failures, in addition to some failures which are prematurely reported by the timeout mechanism.

2.7 Applications

The development of distributed, object oriented applications can be considerably simplified by the use of appropriate programming languages and systems. A number of programming languages and systems include support for distribution of applications, and often amend traditional object oriented programming languages with persistence technology such as stable storage and transactional functionality [154]. Although the architecture presented might benefit from persistency technology in some respects, such technology is not assumed. These system support mechanisms are aimed at amending a type-safe object oriented programming language with functionality for object replication.

Applications are composed of collections of interacting objects. Distributed applications, whose execution is supported by distributed systems, are composed of objects located in different address spaces, possibly on separate PEs. The programming language provides the facilities necessary to create, invoke and share objects. Thus, this must also be anticipated by distributed system software. Furthermore, system software technology is assumed to be present for the reclamation of non-reachable objects.

The distinction between so called client-server and peer-to-peer applications is important for distributed software. A client-server application is statically decomposed of clients requesting services from servers. While being the common approach to distributed computing today, this approach is limited by the static rôles of clients and servers. However, interaction between two objects is by nature a client-server relationship whereby one object invokes the method (the client) upon another (the server) to carry out a piece of work.

To increase the flexibility of distributed software, a model where the client/server rôles are dynamically changing is envisioned as the next step up from client-server computing. By allowing servers to request services from other servers, a peer-to-peer model is formed which assists collaboration and autonomy among *agents* [2, 106]. In a peer-to-peer structured application, objects are considered peers and may invoke methods on each other, essentially functioning as agents carrying out work on behalf of others.

Some software systems at a larger scale are composed of collections of cooperating objects which externally provide a server function. For example, a number of interacting objects might be cooperating to provide

[14]Transparent failure masking is a primary task for most communication protocols running in less than perfectly reliable networks. For example, transport protocols such as TCP/IP and OSI TP1–4 go to great lengths to recover from occasional transient failures [47, 182].

a file service to other components within the system. The particular group of objects providing the file service functionality will typically be located in the same address space to reduce the number of remote invocations and thus achieve reasonable performance. The architecture for system supported replication which is discussed in this dissertation assumes a client-server computational model where servers are internally composed of cooperating objects (see §7.2 p. 79). However, extensions of the architecture are suggested which can eliminate this restriction and allow a true peer-to-peer computational model (see §10.3.1 p. 109).

Multithreaded applications
A thread is a distinct flow of control within a process, potentially executing concurrently with other threads within the same process. Threads communicate via shared variables, and the synchronisation of threads is the responsibility of the application programmer. Multithreading is a useful and powerful concept for the construction of software because it increases parallelism [68] and consequently can reduce the performance penalties with synchronous method invocations [25]. Instead of simply waiting for a long-running invocation to complete, the application can allocate this task to another thread, and continue doing something else meanwhile. In this dissertation, it is assumed that an application will consist of multiple processes, each with potentially multiple threads of control. This, in turn, will occasionally trigger concurrent execution of methods in shared objects (cf. §2.4.2 p. 22). System support mechanisms must therefore be prepared to operate correctly despite concurrent invocations. The system architecture, described in chapter 6, supports object sharing. However, due to some inherent overheads, sharing among processes will incur reductions in performance.

2.7.1 Application partitioning

The application programmer determines the tasks each object is responsible for and their location among the collection of PEs within the distributed system. Because objects are relatively low level constructs, applications are built as a large collection of interacting objects. For the performance of the application it is important to minimise the number of interactions across PE boundaries because these are more expensive than local interactions. Both performance and scalability can suffer badly from poor locality. The application is therefore partitioned into groups of objects in such a manner that most object interactions occur within the group. Ideally, object location should be performed dynamically by system software that optimise application performance. However, dynamic load sharing and object migration are separate hard problems that are not investigated in this dissertation.

Static and dynamic partitioning
Application partitioning can be either static or dynamic. Static partitioning is done at compile time, whereas dynamic partitioning occurs at run time. The benefit with static partitioning is that object interaction can be type checked by a compiler to guarantee that only valid methods are invoked on objects. However, static partitioning is unrealistic for large scale distributed applications. Rather, it must be expected that these applications will be configured and changed during execution. Dynamic partitioning must therefore be supported for large software systems. Dynamic partitioning requires that type-checking of method invocations are checked at run-time.

Heterogeneity
During its lifetime, a large computer system is often required to interact with another, potentially heterogeneous computer system in order to cooperate on common tasks [142]. Because large distributed systems often consist of confederations of autonomously evolving components, problems might occur when evolution is not coordinated across component boundaries, for example, if a protocol between the two components is not updated simultaneously in both components, or schemata and datatypes are changed without prior agreement from both parties [142]. The problem is intensified due to the demands for increasingly *open* systems, i.e. systems which are designed to cooperate with other, potentially

unknown systems. These applications allow dynamic partitioning which requires careful planning of interaction mechanisms and well defined interfaces. Interoperability issues in object systems is a field of active research, see for example [118, 77] and Part 6 in [138]. Object orientation, with strong emphasis on encapsulation and abstraction can be a useful approach to reduce the cost of building interoperable systems [135], and the pure object model adopted in this dissertation acknowledges these principles.

So-called object request brokers (ORBs) have been proposed to alleviate the problem of integrating heterogeneous object systems by using repositories of interface contracts which define the interfaces available to the client of objects within the object store [14]. Network Objects uses the principle of subtyping [34] to allow a certain degree of evolution in the implementation of objects [22]. The implementation of the object may be extended (i.e. specialised) without necessitating any changes in the clients of the object.

2.7.2 Application failures

Applications are distributed over independently failing address spaces. Each address space may contain multiple objects which reference other objects, potentially contained in some remote address space. When an address space fails, all local objects fail, although this cannot be guaranteed with absolute certainty.

Other application issues
Software, like hardware, may fail. However, the nature of software is discrete, hence software failures can be avoided. In contrast to hardware components, a correct software component will never be the cause of its own failure[15]. Many challenges still remain before there can be any realistic hope of constructing provably correct substantial amounts of software.

It is unclear how the use of object oriented techniques will affect, if at all, software failures. One might suspect that increased encapsulation and better mechanisms for data abstraction will reduce the number of software failures, or at least reduce their effect outside the particular object. However, software designers using object oriented techniques are likely to build applications that continuously stretch the limits for comprehensible complexity, and thereby use up the benefits of better development paradigms. Failures in object oriented software might also exhibit more complex failure behaviour, unknown from procedural software due to polymorphic binding and very flexible interaction patterns among objects [16].

[15]Correct software remains correct over time. However, software must interact with hardware, and also often with other software; this will of course imply a probability of failure.

Chapter 3

Computer System Failures

This chapter examines computer system failures and their characteristics with an emphasis on failures in distributed systems. Failures are surprisingly common in distributed systems and often cause significant reductions in a system's usefulness. Generally, any large distributed system is likely to contain a number of failed components at any given time. Additionally, if other components are depending upon the failed ones even small numbers of failures can have large consequences throughout the system. Replication is one technique which has been used for some time to reduce the impact of failures, and this technique will be discussed in more detail in the next chapter. However, it is important to understand the nature of failures before embarking on the task of concealing them using replication techniques. One consequence of the asynchronous system model adopted in this dissertation is that failures cannot be accurately diagnosed, and this makes it harder to deal with them.

3.1 Dependable Computing Systems

A dependable computing system is one which allows users to depend on its service, for example by being reliable and available [104]. However, dependability is a metric which spans many aspects of a complete system, some of which are more abstract and may therefore be difficult to measure. This dissertation is primarily focused on the reliability and availability aspects of dependability as these can be improved using replication techniques. As such, other factors influencing dependability, for example security and maintainability, are not addressed.

A computing system which fails frequently is not very useful for any serious tasks; a user cannot *depend* on such a system. Even for the casual user such a computer system will soon become more of a nuisance than an efficient information processing tool. On the other hand, a dependable computer system can be used for such important tasks as the control of dangerous chemical processes, air traffic control systems, the running of business-critical applications such as a bank's databases or to ensure safe and continuous operation in nationwide telephone networks. As computers take over many important tasks in society, dependable computer systems will become more valuable and, in fact, dependability may be a common requirement of future users [44].

Dependability requirements are often greater for large and distributed systems, and undoubtedly the combination of large scale and distribution poses significant challenges for researchers in the area [45]. Sophisticated evolving software, complex dependencies among system components and heterogeneous computing platforms are issues which complicate the construction of dependable systems. However, large dependable systems are built recursively from smaller subsystems; to be able to build dependable computer systems it is necessary to understand why the subcomponents fail, and how they fail. Therefore, one must consider components individually; only then is it possible to construct dependable systems. After all, dependability is a system issue, all parts of a computing system must be assumed to play a rôle

in the dependability of the overall system [149].

Additionally, efforts to increase the dependability of a system should be focused to give the best effects for a given cost. No matter how many resources go into designing a dependable system there will be a non-zero probability of failure [1]. Consequently, the system's dependability requirements must be determined, as must efforts which will give the highest return in increased system dependability. Additionally, some failures are very costly to tolerate, while other failures are significantly cheaper to tolerate. During the design of a dependable system it must be clarified which failures should be addressed by mechanisms for failure tolerance, and which failures must be neglected.

3.1.1 Metrics

It is occasionally necessary to compare, or otherwise communicate, dependability measurements. A set of metrics is needed to facilitate this. If the terminology is simple and concise, it will reduce the effort needed to understand the principles of an area as complex as dependable computing. The literature is not always concise in its terminology, however, this section attempts to clarify the central metrics, and present them as they are used throughout this dissertation.

Reliability
Reliability is "the probability of a system performing its purpose adequately for the period of time intended under the operating conditions encountered" [150]. Most common is the use of *MTTF* (Mean Time To Failure) ratings to measure reliability [85, 104]. The MTTF rating is often determined through intensive testing or simulations, and is an indication of the expected failure rate of a component. It is important to notice however, that MTTF ratings do not indicate distributions of the failure probability, and that these measures are slightly limited.

Failure recovery
After a component has failed, a certain amount of time will be required to restore it to its operational state. This is called the *service interruption* or *MTTR* (Mean Time To Recovery). MTTR values are also estimates, and can only be used to suggest availability. Depending on the failure mode of a component, different actions may be required to bring the component back to an operational state.

In centralised systems a failure is often dramatic, and will normally cause disruption to the whole program. If a program crashes, it must be restarted and transactions in progress during the crash will need to be repeated either manually or automatically using transaction logs. A human operator is usually responsible for restarting the system, for example by restarting a program.

Occasionally, the crash is caused by permanent hardware faults, and in this case the operator will need to call an engineer to carry out the repair or replacement of hardware components. Hardware reconfigurations and repairs typically take much longer than simple system reboots. Often, the whole process could take minutes, or even hours[1]. In addition to the inconvenience of no access to the computer, individual users are likely to suffer from the loss of unsaved files and the need to manually redo work.

Availability
The *availability* A is the probability that the system is able to provide correct service at a given time [150].

$$A = \frac{MTTF}{MTTF + MTTR}$$

[1]The time it takes to repair a computer system is extremely unpredictable. According to [115], MTTR can sometimes be in excess of 20 hours on particular computer models, although an average of 4 hours is assumed in [150]. It is not hard to believe these numbers considering that they often include the time it takes engineers to arrive at the location with the correct spare parts.

Availability defines the percentage of time a service is available, so that an availability of 100% means that the service is always available. Most computing equipment today is very reliable, and availability is usually in the range 99.9–99.999%. To increase readability of availability figures, the notion of *availability classes* is introduced in [85]. The availability class is the number of leading nines in the availability figure, so for example availability class 5 implies 99.999% availability.

3.1.2 Reliability of computing systems

No computer component is completely reliable. That is; given enough time, they all fail [105]. This is due to physical deterioration caused by for example temperature changes, atmospheric radiation or material weakening. No known technique can be applied to change this process. However, if appropriate design and manufacturing procedures are adhered to, very low failure rates can be achieved, low enough to give satisfactory service. Hence, there are huge variations in the expected failure rates from different computer equipment. Generally, a complex component that is composed of several other components, is more likely to fail before a less complex component. A large proportion of computer equipment will also be exposed to other, even more damaging effects such as occasional power surges, dust particles and vibration. This further strengthens the point that dependability is a system issue; environmental, operational and even system maintenance procedures will have effects on dependability.

The reliability of computer and networking equipment has improved dramatically in recent years due to better manufacturing and material knowledge. Modern computing equipment, built from highly integrated circuits is very reliable compared to the machinery available 20 years ago [115]. Some MTTF values for common components in distributed systems are given by Gray and Reuter [85]. They indicate that most computers sold today have MTTF ratings between 3 and 5 years, MTTF ratings from 3 to 20 years are common for high quality disk drives. However, when the proper operation of a system relies on multiple components, possibly controlled by complex software, the MTTF rating for the system decreases rapidly. For example, a typical LAN has a MTTF rating of only 3 weeks. Likewise, a workstation running complex system and application software is likely to achieve a 3–4 week MTTF rating. It is therefore important to realise that if this problem is not addressed properly, distributed systems of any significant size will provide very poor dependability. As an example, Sriram's thesis contains an investigation of reliability of hosts on the Internet, arguably the largest computing infrastructure in the world, and finds that the expected MTTF is between 11 and 14 days [176]. This coincides with the rapid decrease in reliability as a function of increased number of dependencies among the individual components (see discussion on critical path length in §3.2.1 p. 31).

3.1.3 Reliable networks

The communication infrastructure has a great effect on the dependability of a distributed system. Unreliability of communication is typically a distinctive feature of distributed systems. However, distributed systems are built on top of a range of different networks, for example public networks, LANs and MANs which provide varying reliability.

Due to the potentially costly consequences of outages, public networks are designed to be very reliable. For example, most PSTN[2] networks are able to cope with failures through redundant links and specially designed networking software[3]. Typically, Western PSTN networks offer availability in the range of 99.7% with no outages lasting more than 30 minutes [85]. However, other continental networks do not achieve similar figures, for example, some African telephone networks are hindered by successful call-completion rates as low as 12% [147]. Consequently, building reliable wide-area computer networks becomes difficult

[2]Public Switched Telephone Network.

[3]It should be noted that there is an inherent conflict between economic issues here. For example, the huge bandwidths available in modern fibre optical links makes it possible to multiplex a vast number of communication sessions onto a single fibre. Economically this would be a cheaper option than using a number of redundant links, but to ensure good reliability, this fibre would have to be extremely well preserved.

in such environments as they must often rely on PSTN links to connect the hosts. This is also likely to be a restriction for the Internet as a communication infrastructure for global applications, the poor reliability of some continental networks will restrict the dependability of such software.

For smaller scale networks, such as LANs and MANs, the reliability is usually much better. Although poorly maintained networks naturally give lower reliability, most LANs and MANs achieve very reasonable reliability ratings. A probability 0.00001 of message loss has been indicated in LANs under normal conditions [129]. Additionally, some network protocols have fault-tolerance built into the architecture, such as the FDDI networks which use redundant rings to automatically tolerate single fibre and host failures through a specially designed self-healing protocol[182].

These variations in dependability will have effects on the kind of applications that are run on top of these networks. In general, a lower dependability of communication will motivate a more loosely-coupled application architecture, where interaction among the components is only occasionally necessary. Naturally, added to this argument is the fact that bandwidth is also normally reduced over long-distance connections. Autonomy is therefore necessary to achieve a reasonable performance. In contrast, LAN or MAN-based networks can facilitate a more tightly-coupled application architecture.

3.2 Failure Characteristics

A *failure* in a computer system is a deviation from its intended behaviour, and is observed outside the system. A failure occurs because the system is *erroneous*, i.e. it contains one or more *errors*. An error appears in the information domain, and is caused by a *fault* in the physical domain. Essentially, an error is the manifestation of a fault, and a failure is the effect of an error. For example, if a bit in a memory chip is stuck at value 0, this is a physical fault. When a program writes a 1 into it, but the bit remains 0, there is an error in the information domain. Later, when the program misbehaves due to this error, there is a failure which can be observed externally, for example by an operator observing mysterious or clearly incorrect behaviour [1, 7]. A fault need not cause errors however, and an error need not cause a failure. For example, if the faulty bit in the example above is not part of a program, or the erroneous 0-value is not used within the program, a failure will not occur.

The same notation can be applied recursively to subcomponents of the system and the relationship *fault* $\rightarrow$ *error* $\rightarrow$ *failure* can be thought of as a chain propagating up through the system component hierarchy [104]. For example, a distributed system which coordinates several components, may observe the failure of some of the components (e.g. a functional failure in a communication link) caused by internal faults and errors[4].

Failure sources

Failures may arise from several kinds of errors, and correct behaviour from a computer system depends on both hardware and software. Some basic failure sources such as material weakening and dust particles were mentioned in §3.1.2 p. 29. Although these can cause failures at different levels in the system hierarchy, for example bit errors and PE crashes, there are also other sources of failures which must be considered.

Table 3.1 summarises findings presented by Laprie et. al. [105] and by Wood [196]. The figures given by Laprie et. al. are from transaction processing environments whereas Wood's figures are sampled from a slightly wider selection of environments including university studies. This dissertation does not attempt to analyse the different findings other than to identify that the two surveys show only limited similarities. Laprie finds hardware and software/recovery sourced failures to be almost equal in importance, whereas Wood identifies software/recovery as a significantly more prominent source of failure than both hardware and operational difficulties.

[4] The term fault-tolerance might therefore be slightly misleading; fault-tolerance is normally used to denote any system able to withstand faults, even if they are withstanding the failure of the subcomponents. A term like failure-tolerant might be more informative, but the term fault-tolerant is currently used throughout the literature.

Source	*Hardware*	*Environment*	*Software & Recovery*	*Operation*	*Other*
Lapric	40%	5%	30%	20%	5%
Wood	10%	4%	71%	15%	0%

Figure 3.1: Failure sources in computer systems

It can, however, be concluded from these results that both hardware and software play important rôles as failure sources. Although replication per se can only conceal effects of hardware failures and not software failures (as shown below), it is important to note that many software faults in distributed systems are caused by transient bugs in operating systems and other system software which occur in response to timing and system overload anomalies. It is reasonable to assume that some of these failures can be masked by replication of system components [44], which is the largest failure source reported by Wood. However, other techniques will probably be more effective at reducing the effects of software failures, such as improved development methods and tools.

Software failures
Essentially, hardware fails despite being correct, and software fails because either the hardware fails or the software is incorrect[5]. A well known approach for handling software failures is *n-version programming* [10]. Essentially, it involves replication of multiple, independently designed software components. Due to the severe cost of multiple development groups, only critical components are replicated. The usefulness of n-version programming has been investigated in an object oriented setting [197]. However, it appears that better results can be achieved using more conventional approaches, e.g. allocating more resources to develop correct software. Not only does the n-version approach suffer from the 'average IQ[6]' problem [85], but the approach also requires additional, complex application dependent system software, introducing the possibility of more failures.

3.2.1 Critical paths

Distributed systems consist of interacting components. Consider an object A invoking a method on a remote object B. To complete successfully, this interaction requires correct behaviour from a number of components. Not only must A and B behave correctly, but also so must the communication path between the PEs hosting A and B. A is *dependent* on B and the communication path between them. The set of components from which correct behaviour is required is denoted *the critical path* of the interaction. The number of components in the critical path is called its *length*.

Assuming that a component i fails with probability $p(i)$, and the failure modes of the components are independent, a service in the system depending on n correct components will have a probability

$$p(no\ failures) = (1 - p(i_1)) \cdot \cdots \cdot (1 - p(i_{n-1})) \cdot (1 - p(i_n)) = \prod_{i=1}^{n} (1 - p(i)) \tag{3.1}$$

of providing correct service. Improvements in reliability can be achieved by both reducing individual component failure probability and by reducing the number of components in the critical path. Replication, introducing redundant components, is essentially a technique that provides support for multiple parallel critical paths where each path has an independent mode of failure (cf. chapter 4).

[5] A software component may be vulnerable to other software failures as well if it is built using services from other software components.

[6] All programmers are more likely to make similar mistakes on the hard software problems. Why have n versions of software which crashes on the same inputs?

3.2.2 Independence of failures

Due to geographical distribution and physical heterogeneity, components in distributed systems often have independent failure semantics, i.e. the failure of one component does not affect the probability of failure of another (independent failure semantics imply that the probability functions are memoryless; see also appendix B.1). The failure will be limited to those components which either directly or indirectly depend on failed components, the number of dependencies determine how the failure *propagates* throughout the system. Increasing the number of dependencies causes an increase in components affected by the failure propagation.

Not all failures in distributed systems are independent. Often, a failure in a component causes a propagation of failure to other components [104]. For example, if two workstations use the same power source — they might share a wall socket — they are both vulnerable to an electric power outage at that socket. In a large building a power outage is likely to cause failures in multiple machines and network components. It is therefore important, when designing distributed systems, to ensure that an appropriate degree of failure independence is achieved (for example by installing redundant power supplies, introducing multiple administrative domains, using different machine architectures and different operating system platforms [114].).

Independence of failures distinguishes distributed systems from centralised systems; when a centralised system fails, the whole system normally become useless, and the system cannot offer any service until it has recovered from the failure. In contrast, the probability of all the components in a distributed system failing at the same time is extremely low. However, centralised systems are usually much better protected against accidents and other mishaps than distributed systems. For example, a centralised system can often be located in a single room, where access and maintenance can be well controlled.

The PEs (processing elements) in a distributed system are often workstations in peoples' offices, and they might be turned off at the end of a day. The autonomy of the components makes the distributed system vulnerable. For example, it would not be a good idea to use such a workstation as a central mailserver in a department. Part of the problem is that enforcement of computer usage policies can become very difficult in such environments. Secondly, some workstations might simply be moved or disconnected for some time (the workstation might even be a portable computer). However, it is the independence of failures which makes it possible to build fault tolerant systems. By 'masking' some failures through redundancy, the system can potentially continue to operate correctly.

3.2.3 Failure semantics

System components have different failure behaviour or failure *semantics*. Failure semantics describe how components are expected to fail. A clear understanding of failure semantics is important as only expected failure behaviours are likely to be tolerated by any failure resilient computer system. If an unexpected failure behaviour occurs, which is not considered by the failure resilient system, then, it is likely that the system will fail also.

To simplify the discussion of failure semantics, they are often classified according to how difficult it is to tolerate them [89, 163, 184], or how *strong* they are [13, 44]. A weak failure semantics implies that few assumptions are made about the component, it may exhibit a wide variety of different failures. In contrast, a strong failure semantics assumes that the component fails in only a small number of predefined ways. Because so few assumptions are made about the behaviour, a component with weak failure semantics is more difficult to tolerate. Commonly used failure semantics are listed below in ascending complexity order, i.e. the former are easier to tolerate than the latter. Faults no more complex than timing faults are denoted benign failures, other faults are denoted malign failures. An interesting feature of this classification hierarchy is that all benign failures are detectable in the time domain, whereas malign failures can only be detected in the data domain.

Benign failures. Benign failures are 'nice failures' in the sense that relatively cheap mechanisms can

be used to tolerate them. Some of these failures can be tolerated without the use of replication techniques. For example, omission failures are normally tolerated in communication protocols using techniques such as retransmission and message sequence numbers. However, the failures leading to PE or link halts can only be concealed with redundancy, but the failure masking capability of replication schemes will be greater if only benign failures are assumed.

Initially-dead failures. The PE does not execute any part of its program [184]. A communication link does not deliver any messages.

Fail-stop failures. A PE stops processing permanently in a controlled manner. A communication channel stops delivering messages. Other PEs are notified about the event [13].

Crash failures. A PE stops processing abruptly and loses its internal state [184]. A communication link ceases to deliver messages. Other PEs are not automatically notified about the event.

Omission failures. A PE fails to deliver (receive omission) or send (send omission) some messages. A link loses a subset of its messages.

Timing failures. A PE fails to respond within a specified timeframe (also called performance failures [44]). A link fails to deliver a message within a specified timeframe. Note that this failure mode is only applicable to synchronous systems. Asynchronous systems make no assumptions about timing of events.

Malign failures. Malign failures are 'hard' to tolerate because such failures can only be observed as erroneous results from computations. Therefore, redundancy must be used to tolerate them, a technique which might add significantly to the cost of constructing the system and also incur overheads during operation. Redundancy can be introduced at several levels in the system hierarchy, for example as redundant data in communication protocols in the form of error-correcting codes or as server groups. Although this redundancy adds a certain overhead, these techniques are able to completely conceal many failures from the client. For example, in contrast to a retransmission technique which adds delays to the service, many replication techniques do not result in such irregularities. A more thorough discussion of the failure-masking capabilities of replication techniques is presented in the next chapter.

Incorrect computation failures. A PE fails to produce correct output despite correct input [13, 44], for example a procedure which returns a list-element not stored within the list or a communication link which delivers corrupted messages.

Authenticated Byzantine failures. A PE behaves arbitrarily. However, an authentication mechanism is available so that other PEs can identify the faulty PE.

Byzantine failures. A PE or link behaves in an arbitrary or even malicious manner. For example a link that generates random messages or a PE which sends conflicting messages to other PEs.

An algorithm tolerates a failure class if it ensures correctness in the presence of a failure of that class. An algorithm tolerating a particular failure class also tolerates weaker failure classes. Clearly, an algorithm that tolerates arbitrary (Byzantine) failures also tolerates fail-stop failures. A fail-stop failure is just a special case of arbitrary behaviour.

It is often possible to reduce the complexity class of a failure. For example, timing failures are commonly reduced to omission failures by the use of timeouts. In asynchronous systems this is a conservative approach, because a timeout mechanism cannot correctly distinguish all omission failures from timing failures (the message might appear just after the timeout expired). Omission failures are simpler to handle than timing failures, and this brings benefits to the protocol using the channel. Retransmission of lost messages is a common approach to deal with omission failures.

Timeouts are conservative, a message arriving just after the timeout expired could be perfectly valid. However, because *most* messages arrive within the timeframe of the timeout, it does catch *genuine* omission failures most of the time[7].

[7] A genuine omission failure is a message lost forever.

Timeouts provide liveness by sacrificing accuracy [19]. That is, failures are reported within a finite time, but some operational components may be declared failed. The use of timeouts is also a mechanism used to make synchrony assumptions in asynchronous systems. However, if the timeout value is set sufficiently high, it is very likely to distinguish timing failures from omission failures. Some statistical information about the frequency of early timeouts is often used to improve its efficiency by dynamically adapting the timeout value to the current mean latency (cf. §2.3.1 p. 19).

Arbitrary failures are very costly to tolerate [69, 186]. Expensive consensus protocols, based on atomic multicasts and high levels of redundancy are required. For example, t-resilient non-authenticated Byzantine agreement among n PEs requires $n \geq 3t+1$ in addition to $t+1$ rounds of messages with potentially large message sizes ($O(n^{t+1})$). In contrast, Byzantine agreement in the authenticated case[8], requires $n \geq t$, $O(t)$ rounds and $O(n+t^2)$ messages [13]. The cost of computing the signatures must in this case be weighted against the added fault-resilience.

Additionally, expecting arbitrary failures may be questionable in many system contexts as not all components in a system can be allowed to behave arbitrarily. At some high level in the system hierarchy, one component will be the only client of the failure prone subcomponents, it is not reasonable to assume that all users of a computer system can take on the rôle of failure detector. It cannot be guaranteed that the only client component is not exhibiting arbitrary failures itself, as no-one remains to 'guard the guards'.

Partition failures

Partitions occur in distributed systems if communication failures prevent a subset of the PEs from communicating with other PEs. If the connectivity of the network is low, partition failures may be frequent, but a small increase in the connectivity of the network can reduce the probability of partition failures significantly. Partition failures can cause severe problems for distributed algorithms because PEs in different partitions can easily believe that they are the only PEs left in the system, and therefore make independent modifications to the global state. If PEs in different partitions are allowed to modify shared data the copies of the data must be reconciled when the partitions are again re-connected.

Partition failures are of great importance for the design of replication management schemes. The main characteristic of replication schemes, strong or weak consistency, determines whether or not the scheme allows independent updates in different partitions or not (see chapter 4). In this dissertation, a network model where partition failures are rare is assumed. Partition failures are handled pessimistically; at most one partition is allowed to make progress (cf. §6.3.2 p. 70).

3.2.4 Failure detection

In a distributed system it is occasionally necessary for an object to determine the failure status of other objects in the system, for example in a replication protocol. However, because the distributed systems considered in this dissertation have asynchronous behaviour, failures can only be suspected, not reliably detected (although they can be detected with arbitrarily high probability). All asynchronous systems are restricted by the impossibility result published by N. Lynch et. al. [69]. Essentially, if there are no bounds on the delay of messages, no two deterministic objects can reach agreement on a value in the presence of failures. For example, the asynchrony implies that a slow object cannot be distinguished from a failed object and vice versa. An object which does not receive a response from another object would theoretically have to wait indefinitely to distinguish between a slow and a failed object. Clearly, this is not practicable in any real system. Therefore, as a measure to gain efficiency for a small loss in accuracy, various assumptions which limits the asynchrony are used. Timeouts is a good example here; in case the object does not respond within a certain time it is assumed to be failed. The consequence of this is that it becomes difficult to guarantee correctness.

Fault diagnosis deals with efficient and reliable detection and localisation of faults. Accurate fault diagno-

[8] Authenticated messages are non-forgeable, all corrupted messages are detected and the message signature can be verified by any PE.

sis will in many cases significantly simplify the task of building reliable distributed systems. For example, the cost of Byzantine agreement is severely reduced if messages are authenticated [66, 13]. Because PEs cannot communicate arbitrary messages without being detected, the protocols for agreement on global properties require fewer rounds with smaller messages.

Standard communication services used in many distributed systems make it difficult to detect failures accurately. The weak failure semantics of the communication primitives requires that application programmers provide reliable failure detection in the application. For example, the popular RPC paradigm, widely used in many distributed systems because of its simplicity, cannot offer particularly strong invocation guarantees [19]. The only guarantee available to the programmer using standard RPC implementations is that the method will be executed *at most once* on the remote machine if it is initiated once, and *at least once* semantics are possible for RPC calls that are retried until the caller receives a positive acknowledgement message.

For non-idempotent operations, e.g. a method *inc(n : INTEGER)* which increments an object state variable, this becomes a problem. However, a solution to the problem has been used in communication protocols for some time, but it requires cooperation from both the caller and the callee. Through the use of unique *sequence numbers* for each invocation request, the callee can simply discard messages with duplicated sequence numbers, but should still send a positive acknowledgement message. The caller can thus keep on retrying the invocation (using the same sequence number) until a positive acknowledgement is returned without any danger of the method being invoked more than once. However, even this protocol cannot tolerate continuous loss of messages. The caller would in this case block forever without receiving any acknowledgement.

Transient failures
Some failures are transient, e.g. occasional omission failures, and might happen for just short periods of time. They will not always be detected. For example, during a period of congestion in a network, a switch might temporarily refuse to accept any more messages into the congested area. A PE that tries to send a message during this period is likely to observe this refusal of service, while a currently passive PE does not observe it. The usual approach to handling this kind of failure is to use a retry mechanism, i.e. the PE that observes a refusal of service will try sending the message again at some later time. If the resource is essential for the client so that the client cannot proceed without it, there might be no better alternative than to just keep trying until the resource is eventually available, possibly producing a warning or notification message. A danger of naive retry mechanisms is that they can generate enough messages to flood the network causing additional congestion and overfull buffers. This is further emphasised if several PEs are concurrently repeatedly sending retry messages due to a transient server failure. It is also important to avoid congestion as this will reduce the risk of denial of service.

A possible solution to this problem is to monotonically increase the interval between retry events. This is a well known technique from communications used in several network protocols, e.g. the Ethernet and in the TCP/IP protocols. However, there is a tradeoff that has to be made between fast recovery and the amount of traffic generated.

3.3 Avoiding Failures

Software failures are hard to avoid in distributed systems due to these systems' complexity. Replication as such cannot reduce the probability of system component failures, but it can increase the number of access paths to a resource thereby increasing the probability of finding a functioning path. Additionally, even small improvements in average component reliability can have a large impact on the total system reliability. For example, hardware component reliability can be increased by following manufacturers' guidelines for operating environments by providing suitable ventilation and maintenance of equipment. Protection against environmental damage such as electrical power instability, flooding and sabotage will also reduce the probability of failures.

3.3.1 Effects of software development methods

There are a number of approaches that can be followed to reduce the probability of implementing faulty computer systems, but there is usually a tradeoff that has to be made between reliability and cost. The cost of a failure in the system might justify increased efforts during development[9]. For example, when designing critical computer systems, formal methods might be used to specify and verify hardware and software designs. However, formal methods are still limited in their usefulness for large scale systems. They are most appropriate to model small components.

Additionally, appropriate testing procedures [191] can help locate software faults before the software is put to use. However, it is important to note that exhaustive testing is not feasible for any realistic distributed system, and that 'black-box' testing of key components is likely to be more appropriate although not able to guarantee correctness. Further, software is discrete by nature, and even small changes made during testing can therefore lead to large effects [45]. A key research area, now and in the future, is the development of methodologies and tools for producing correct software. At present, all significant pieces of software must be expected to contain bugs that can lead to failures.

3.3.2 Effects of overload

Designers and users of computer systems aim for the best possible performance. This requires efficient utilisation of system resources such that both overloading and excessive idle time can be avoided. Overloading of system components in a distributed system can lead to serious side effects[10] in the rest of the system (e.g. deadlocks due to overfull buffers or disks and excessive retransmissions due to slow responses from overloaded PEs). Arguably, such failures are only anticipated after they have caused major problems [46]. Also, it has been indicated in the literature that overloaded system components have a higher probability of failure [105]. By careful and efficient utilisation of the system resources, the reliability of the system will increase. The probability of overload can be reduced with the use of load sharing strategies, however this is in itself a difficult problem outside the scope of this work.

3.4 Summary

This chapter has focused on examining failures and their effects in distributed systems. As the scale and importance of distributed systems increases, failures will, if not managed appropriately, result in systems with poor dependability. This chapter has defined the failure terminology used throughout the dissertation and presented important motives for the construction of dependable computing systems.

Understanding the characteristics of failures is necessary for the proper use of replication techniques. The next chapter is concerned with replication techniques which are able to conceal failures. During the design of a dependable computing system, its dependability requirements must first be determined to balance the cost of failure resilience techniques with the benefits of increased dependability. Most important is perhaps the distinction between malign and benign failure semantics. If it can be assumed that the underlying components in the system only exhibit benign failures, replication techniques will be able to tolerate more failures, and it might be possible to avoid using replication altogether; simple error correcting procedures such as retries and retransmissions might be sufficient.

Additionally, this chapter has identified other important issues for the construction of dependable computing systems which replication techniques are not able to deal with. For example, the use of better software development methods might significantly improve system dependability and thus be an equally important factor.

---

[9]The flight control software for the space shuttle has been estimated to cost $1000 per line of code [98].

[10]See [165] for an example of how overload can spread chaos in a distributed system.

Chapter 4

Replication Techniques

Chapter 3 identified a large number of independent failures as a key characteristic of distributed systems. Without any mechanisms for failure tolerance, most failures incur extra delays or loss of data, thereby weakening these systems' usefulness.

Replication is an old and well known approach used to achieve resilience against failures. It dates back at least 40 years [193], and many techniques are available for this purpose. However, a number of conflicting issues must be considered to determine which replication strategy is most appropriate in a particular setting, e.g. system support. This chapter presents a number of replication techniques and a discussion of their advantages and limitations. Among the techniques discussed are object replication and replicated objects, two distinct approaches to replication in object-oriented systems.

4.1 Background and Motivation

The main focus of this dissertation is on the provision for replication support in object oriented programming systems. The goal is to provide generic support which can assist developers of fault-tolerant software. Replication is a complex issue, and if the problem can be solved by support software the application programmer's task will be simplified. However, as will be discussed throughout this chapter and the next (chapter 5), some particularly challenging problems arise in this setting due to strict encapsulation and arbitrary object interactions which inevitably require compromises.

Distributed computing systems are becoming an essential platform for modern computer applications. However, a high number of failures is a key characteristic of these systems. Failures, normally leading to extra delays or loss of data, can easily lead to critical situations or simply become a nuisance for users. The problem with failures becomes more prominent as the size of the distributed system increases and more inter-dependencies among the system components are created. Given that the trend towards increasing reliance on distributed computing systems continues, the requirements for dependability are likely to be heightened [44]. Therefore, it is worth investigating approaches that make such systems more resilient to failures and thereby increase their usefulness.

Replication has been widely used as an approach to increase a system's resilience to failures and to satisfy requirements for reliability and availability[1]. Replication is almost as old as computing itself; it was first suggested by John von Neumann in 1952 (and published in 1956) as a countermeasure to the accumulation of failure probabilities in basic organs when a computing automaton was built from a large network of such organs [193]. It was suggested that, by the use of a particular 'majority organ', a number of single organ failures could be masked. The majority organ was in fact the first version

[1] See §3.1.1 p. 28 for definition of terms.

of a function shipping replication scheme, composed of a number of independently executing machines[2] (see §4.3 p. 42). Replication is also a useful approach to reducing the consequences of sabotage and other physically destructive events such as fire and floods [40].

Traditionally, replication was most commonly used for critical applications like air traffic control systems, spacecraft systems, telephone networks and a number of military applications. The extra costs incurred by replication are justified by potentially substantial losses in case of system malfunctions. The danger of human deaths or injuries have been the primary issue of concern.

Following this, replication has been employed for less critical applications, e.g. banking systems and supermarket retailing databases. Due to the increased computerisation in these settings and the large material values at stake, replication of data is common and reduces the damage caused by occasional failures. Specially designed software, tailored for each particular application, includes techniques for managing the replicated data. Although manual backup procedures could alleviate part of this problem, replication is beneficial due to its speed and the reduced efforts required by human operators.

Replication has been a subject of research for some considerable time [55, 48, 79], and has led to the construction of a large number of systems incorporating some form of replication, of which database and file systems are prime examples [40, 158, 181]. Additionally, research into new approaches for replication techniques continues. Efforts are made to address the demand for availability and performance in object systems, for example object oriented database systems [111], programming languages [31, 5], and in persistent programming systems [113].

However, many replication schemes must be tightly integrated to a particular application, thereby necessitating substantial extra effort from the application developers and complicating the software construction process. Arguably, system support for replication can help reduce the incurred overheads in development cost. Quite recently, commercial database vendors have targeted wider markets with solutions for more generic support of replication [177]. Concurrently, distributed information storage systems with support for replication [128] and software development tools for reliable systems, such as the Isis toolkit [19], have appeared. This might be an indication of increasing reliability and availability requirements throughout a broader computing community.

In [145] David Powell makes the distinction between two different approaches to fault-tolerance; *distribution motivated fault tolerance* and *fault tolerance motivated distribution*. They denote two opposite views of maintaining reliability, and are each applicable to different application contexts. These are extreme views, and intermediates are also possible. For a particular system, a combination of the two views is likely to be the basis for fault-tolerance.

Distribution motivated fault tolerance

Distributed systems are more vulnerable to failures than centralised systems or traditional parallel computers. Large scale systems therefore tend to employ some degree of local administration which reduces the need for constant access to system wide information, and simplifies system administration. Naturally, local administration implies autonomy which speeds up decisions about local matters [8] (hence possibly increasing system scalability). Semi-autonomous clusters can be managed primarily by a local administrative body, and groups of such clusters can then be partially managed centrally for enforcement of global administration rules. Increasing the value of such arrangements, this often matches traditional hierarchical organisational structures.

Multi-cluster systems consist of a large number of system components (cf. §2.3 p. 18), and inter-cluster interaction tends to be less reliable than intra-cluster interaction. It is therefore normal to maximise the use of services local to the cluster, and only occasionally use services available from other clusters [196].

Distribution motivated fault tolerance is necessitated by this inherent distribution. Users have varying degrees of control of their own machine in traditional workstation environments. Depending on departmental policies, users might be allowed to do local configurations and management. Users of some

[2]Although von Neumann is usually best known as the creator of the sequential computer, he was also engaged in work on parallel machines.

machines like PCs may prefer to switch them off after work, although this might be uncommon for users of workstations. Additionally, users often run buggy software on various machines causing occasional crashes throughout the network.

Systems like these are also likely to grow substantially with time, to keep up with increasing demands within the organisation. In addition, because of their size and complexity, they are also harder to maintain and change, and one cannot expect the average user to fully comprehend the whole system[3]. When a failure occurs, it might require substantial effort and time to both locate and correct it. And indeed, a large distributed system will have relatively frequent failures [176]. During the period of repair, a large number of users might be affected, and the cost incurred by this can be substantial. Furthermore, system maintenance will cause occasional outages, preventing users from accessing shared resources [196]. It is therefore necessary to support fault tolerant operation through the use of replication techniques.

Fault tolerance motivated distribution
Some computer systems are used for critical applications where human life would be jeopardised or substantial costs incurred as the result of the system becoming unavailable or operating incorrectly. In a context like this, distribution and replication may be the only way to achieve the required system reliability. Some examples are flight control systems, process control systems, public telephone services and banking systems. The approach taken to replication will be more constrained, there will typically be only a single administrative body, so there is much less autonomy than in distribution-motivated fault tolerant systems. Secondly, more attention will be given to timing constraints, as most of these systems are running real-time, or time-critical applications. Because of this, much of the heterogeneity found in distribution-motivated fault tolerant systems is avoided by simply replicating existing technology like database systems and computing platforms. Additionally, the criticality of these applications can justify the cost of dedicated redundant network links between the computers which can reduce the jitter normally found in shared networks.

4.2 Problems with Replication

The key idea of using replication for fault-tolerance is to eliminate single points of failure by introducing a number of redundant replicas (or copies) of one or more system components, e.g. communication links or computers. If one component fails, the replication scheme automatically makes other replicas take over. Replication might also be used for other reasons; in some settings, e.g. in large or low-capacity networks, it is necessary for performance or autonomy reasons to make copies of a dataset to exploit locality. However, replication introduces other problems.

4.2.1 Maintaining consistency

If the replicated components store data, then there will be consistency requirements for the copies of that data. Maintaining full consistency requires careful synchronisation of replicas which might be very costly in some systems, and might not be feasible for large numbers of replicas. While reduced consistency can restrict this cost, some applications cannot accept out-of-date data. Also, it would be beneficial if the user of a system did not have to be concerned about whether the system used replication or not — a user sitting at a workstation should not observe any other difference than increased availability in the system using replication. Similarly, a programmer developing software for the system should not have to deal with replication issues. However, many replication schemes do introduce new complexity for the programmer, and the ones that don't depend on technology with limited scalability.

Both end-users and application programmers using non-replicated systems normally assume full data consistency. Consequently, most applications are written under the assumption that the data being

[3]According to Leslie Lamport, a distributed system is defined as one in which "A failure in a machine you haven't even heard of stops you from getting any work done."

manipulated is consistent. Central data repositories can implement serialisation on shared data, and thereby achieve full consistency, but it is more difficult in distributed systems.

Replica consistency is the property that the data stored in the replicas are valid. This means that operations manipulating the data leaves the data in a correct state, i.e. satisfying any integrity constraints[4] [85]. Due to the partitioned nature of the data, inconsistency can occur among the replicas during updates. To deal with this problem a protocol for update propagation must be used. Update propagation protocols, also called consistency protocols, are commonly classified as either pessimistic (strong) or optimistic (weak) depending on whether they guarantee that inconsistency can never occur (pessimistic/strong) or if they allow temporary inconsistencies that are reconciled later (optimistic/weak). In the case of optimistic/weak consistency, the user might be forced to request a reconciliation procedure after eventual updates to the data are made, and the failures in the other nodes or network links have been corrected.

Also, if the replicas are shared among several clients, operations on the replicas must be serialised by some isolation mechanism, e.g. locking, to maintain correctness in the presence of concurrent update requests. A solution is to use the replicas themselves as part of the locking mechanism; mutual exclusion can be guaranteed for example if some set of the replicas must be exclusively locked before an operation is allowed to execute. This is the approach taken by the voting and coterie-based replication schemes described in §4.4.4 p. 47 and §4.4.5 p. 48.

4.2.2 Providing replication transparency

Building a distributed system which appears to users as a 'powerful centralised machine that never fails' is a goal for many distributed system developers [51, 166]. Although it may not be feasible in practice, it is a useful goal to aim for in distributed system design as such a system could be used without noticing the complicated technology underneath the surface. Mechanisms which simplify the user's model of interaction with the system are of great value because distributed systems would otherwise require users to deal with unnecessary issues such as heterogeneity and locality. For example, a distributed file system which allows users uniform access to their files, independent of which workstation is used, makes it possible for users to easily switch workstation. Similarly, a client of a replicated service should observe a minimum of additional complexity compared to the equivalent non-replicated service.

Application programmers can also be considered clients of distributed systems, although at a lower level. It is also important that the programmers' model of the system be kept as simple as possible, this will reduce the cost of constructing software for the particular system. The view taken in this dissertation is that the issues of reliability and availability are orthogonal to most applications and that programmers should not be burdened with implementing mechanisms for replication in their applications. Rather, system support mechanisms for replication should be available to assist the programmers during software development, and be as transparent as possible to hide most of the underlying complexity (see chapter 5). This introduces an important tradeoff for replication techniques.

Replication transparency exceptions

Replication transparency implies that the replication protocol, replicas and inconsistent data should be invisible to the programmer. The system should hide all details about the redundancy, and just provide the user with ordinary but failure-resilient system abstractions. However, situations might occur which make it impossible to conceal underlying failures from the programmer. For example, if too many failures occur at the same time, the system abstraction will become unavailable. In this case it is important to ensure that the failure is adequately reported so that the programmer can take appropriate action, for example by retries or restarts.

[4] Mutual consistency among replicas is however not a sufficient criteria for correctness.

4.2.3 Maintaining performance

High performance in distributed systems results from asynchronous operation which allows PEs within the network to make use of inherent parallelism [18]. However, replication management schemes may introduce complex and costly synchronisation protocols which contradict this principle by trying to synchronise replicas on different PEs to maintain mutual consistency. In particular, for high consistency schemes, the tight synchronisation incurs performance overheads which can result in low scalability. When high consistency schemes are necessary, it is important to consider factors which can regain some of the benefits from asynchronous operation. For example, network support for multicast can reduce the cost of synchronisation in the high consistency schemes. Such functionality within the network proper reduces the amount of network traffic when multiple recipients must deliver the same messages. Network supported multicast will also relieve PEs from the task of managing multiple connections to replicas. Multicast is available in many LANs, but are uncommon in larger networks which tend to be composed of point-to-point links and connect heterogeneous LANs. Architectural support for multicast is therefore more difficult to provide in larger scale networks, and is therefore rather uncommon. However, research efforts such as the MBone have led to improvements and might give rise to more widespread availability of multicast support in future networks [63]. Additionally, many distributed applications may benefit from internal concurrency, e.g. from the use of concurrent threads or processes to execute various parts of the application. Weak consistency schemes, requiring less synchronous update protocols, do not impose the same overheads and are therefore more useful in larger systems.

Several other issues also influence the performance of a particular replication scheme:

Number of replicas. The cost of maintaining consistency increases with an increasing number of replicas. In high consistency schemes, replicas are updated synchronously. Processing and network latencies can therefore potentially severely reduce performance. Optimistic concurrency control policies, allowing concurrent updates, can reduce this cost, but only if both the number of conflicts and the cost of resolving them is small (see also §4.5 p. 49).

Replica placement. The location of replicas determines the cost of accessing them. Finding the best placement for the replicas is an optimisation problem which is dependent on the access patterns to the replicas and the cost of accessing replicas at different locations within the system. In distributed systems with large differences in the cost of accessing local and remote information the choice of replica placement should be addressed carefully. Because load in distributed systems is dynamic, a good replica placement map cannot be determined statically. Rather, ensuring effective object locality will be a continuous process in which objects move about, dynamically adapting to the current network loads, failure patterns and object interaction patterns [114]. However, dynamic replica placement requires object migration support [178], and complex algorithms for determining costs and good location policies dynamically.

Failure patterns. The failure patterns in distributed systems have an important influence on replica scheme performance. For example, in a weighted voting scheme it is a good idea to give more weight to replicas located on PEs which are reliable and not overloaded. Some network connections might be known to be more reliable than others, and this knowledge can also be used during replica placement decisions[5].

Nature of transactions. Some replication schemes are optimised for certain types of transactions, e.g. query-transactions, which do not make modifications in replicas and hence do not incur consistency problems.

4.2.4 Providing high availability

Availability is the probability that a system is able to provide correct service at a given time (§3.1.1 p. 28). The replication protocol determines the number of replicas necessary to perform operations upon

[5] Overloaded PEs may also cause communication failures such as lost messages due to overfull buffers.

the replicas. A voting-based protocol might require that a client gathers votes from n out of m replicas. If less than n replicas respond to the request the client will not be able to perform the request on the replicas, and must accept unavailable service.

Furthermore, high availability can conflict with performance for full consistency schemes. If n is large, the client must do a lot of work before carrying out the request, even if the scheme can tolerate a large number of failures ($m - n$ replicas might be crashed).

Optimistic schemes are normally able to allow any operation to proceed on any set of replicas, even if the set only contains a single replica. Additionally, the client has to do much less work, for example, contacting just a single replica can be enough to carry out a request. However, optimistic schemes are troubled with the requirement to resolve potential conflicts. Thus, optimistic schemes tend to offer greater availability than pessimistic schemes by reducing the frequency of lost opportunities at the cost of resolving conflicts [55]. Optimistic consistency schemes are discussed further in §4.5.

4.2.5 Other problems

Replication management schemes entail the added costs of providing the replicated units (e.g. diskspace, CPU-capacity and memory). This cost is directly proportional to the number of replicas required, and can hence be accurately determined.

Replication is essentially a technique to eliminate single points of failure. However, for reasons outside the realm of the application developer a software system might depend on single components further down in the system hierarchy. Multiplexing in communication protocols is an example of how a single point of failure is re-introduced. Even if the application maintains a set of replicated objects on different PEs in a LAN, the network may itself be a single point of failure and hence compromise the reliability of the whole application if it fails. This problem might be solvable in small systems where the developer has more control and could initiate the installation of redundant network cables and interfaces. However, if the distributed system uses leased PSTN links for LAN interconnections, the developer has less control over the allocation of redundant communication channels and must trust the provider of the WAN link to ensure the required reliability.

Security is a very important issue in distributed systems. Large scale systems are used by a large number of people, and it is therefore essential to provide support for authentication and access control to data. In systems employing replication of data it is further important to maintain the same security among all the replicas. In autonomous distributed systems, maintaining information security may prove to be very difficult due to potentially non-uniform security policies.

4.3 Replication in Object Systems

Two distinct approaches to replication are recognised within distributed object systems. The data shipping approach, also called passive replication [113] or simply data replication [59], regards objects as passive data-structures which are passed between a replicated object store and the computer performing the processing of invocations on the object.

In contrast, the function shipping approach, also known as the state machine approach [162], active replication [113] or object replication [59], deals with replication of both objects and their invocations. The objects are stationary and they replicate the *computation* of a method invocation.

Data Shipping

A replication scheme based on data shipping usually depends on a central resource for the processing of invocations on objects. A server, the manager of the replicated object, receives invocations addressed to the object. The server fetches replicas of the object from a reliable (usually persistent) object store, and processes the invocation locally. The data shipping approach is often adopted in systems where

mechanisms are already available for the transportation of objects such as in distributed database systems [3]. For example, the data shipping approach is used in the Thor database system developed at MIT, and a variant of the primary backup scheme[6] is used to ensure consistency [111].

A data shipping replication scheme has several disadvantages:

1. Complex mechanisms for transportation of objects are required, and there is an inherent conflict with the encapsulated object oriented system model; rather than viewing objects as passive data structures, they should be used as servers accepting service requests across a network. For systems adhering to this system model, a data shipping approach might introduce significant architectural mismatches and added complexities, for example if objects are residing in heterogeneous object stores.

2. Only the server performs processing — and this removes any benefits of parallelism in the replica hosts. Additionally, objects execute their methods by both manipulating local state and invoking methods on other referenced objects. Thus, if messages are smaller than the potentially large number of objects that need to be shipped across the network, the data shipping approach will give lower performance.

3. The server will easily become both a hotspot and a single point of failure which reduces the reliability of the replication scheme[7].

Although the data shipping approach is commonly used in database systems, in the form of client caching, the disadvantages using this approach for other replication schemes appear to be significant in systems where less direct manipulation of object state is appropriate. Data shipping also complicates locking and consistency protocols (see §9.3.2 p. 100 for a discussion of replication in database systems).

Function Shipping

Function shipping, as the name implies, involves sending the function to the data for remote processing. For example, traditional client-server relational database systems employ function shipping; SQL queries are sent to a server which executes the query on local data [155]. However, as queries to relational database systems may construct large result tables, there are inherent tradeoffs between shipping the query or shipping the data. Optimisation of distributed query operations is a non-trivial problem [40].

In the context of replication schemes, a function shipping approach also replicates the invocation on objects. In contrast to data shipping, function shipping is a desirable approach because it is tolerant of failures during processing of the invocation [59]. The function shipping approach is also particularly beneficial in object systems as it may eliminate the disadvantages mentioned for the data shipping approach, in particular the need for potentially expensive object migration [178]. However, the problem of data shipping reoccurs if objects need to be copied as method arguments, and for large scale distributed systems such copying may be necessary for autonomy reasons [52]. However, in smaller scale systems, as the ones of interest to this dissertation, sharing objects by reference is practicable.

Assuming that a remote invocation is smaller than the object which is being invoked, less communication capacity is required in an object replication scheme[8], although it incurs higher demands on low-latency communication protocols [192]. However, function shipping comes at a cost of utilising processing capacity at multiple machines, but this is usually not a big problem as networked machines are commonly underutilised anyway [62]. Additionally, in a data shipping approach, processing capacity is also required to fetch objects and transmit them across the network. Furthermore, function shipping conforms well to

[6] The primary backup scheme is discussed in more detail in §4.4.1 p. 46.

[7] In the case of a primary copy scheme, this disadvantage can be ignored.

[8] If both objects and invocations are smaller than the message size actually transmitted across the network, this argument can be ignored. However, communication protocols might fill empty space in data link frames with data from other concurrent communication sessions.

the philosophy of the object model, as data are viewed as units encapsulated by methods and invocations are seen as messages being sent between objects (see §5.9.3 in [38]).

However, as in all replication schemes, a coordinator is necessary to coordinate the replication of invocations, and will constitute a single point of failure[9].

4.3.1 Object replication

Approaches for replicated objects, e.g. [31], involve designing objects for replication. In contrast, object replication deals with the replication of already existing objects. Object replication is a function shipping approach where each member of the replica group executes invocations in parallel.

Problems with encapsulation
A particular problem occurs in object systems due to the encapsulation principle (cf. §2.4 p. 21). Whereas data in a traditional data-oriented system are simple overwritable values, the data inside an object are protected with methods that define the allowed operations upon the data. Operations for overwriting data may of course be implemented on such objects, but so can operations that *mutate* the data inside the object (e.g. a method on an object which increments a counter variable)[10]. Therefore, to maintain full consistency among a group of object replicas, each replica must receive the same sequences of invocations. This principle incurs some restrictions for replication techniques in object systems. For example, consistency protocols based on updates of subsets of the replicas, like voting or coterie-based schemes (see §4.4.4 p. 47), assume that data are directly exposed and therefore can be directly compared and overwritten. Data within replicas belonging to a subset which was not updated in some previous round will have an earlier timestamp, and are subsequently marked as stale and directly overwritten with the most current values during following rounds of the replication protocol.

Additionally, object replication schemes, while conceptually general and elegant, are restricted by the rich behaviour of objects [59]. Objects may reference each other in arbitrary patterns. If an object A is replicated and contains a reference to another object B, it can only be determined from the semantics of the application if invocations from A to B should be replicated. In case object B is shared among the replicas of A, object B should receive exactly one invocation from the replicas. However, to avoid single points of failure, more than one invocation should be sent from the replicas of A. This arrangement is not transparent, the designer of the object must decide whether calls from an object replica are made to another shared object or not. As an object's methods cannot in general be assumed to be idempotent and commutative, the object might mutate, therefore it is of crucial importance to invoke an object's methods the correct number of times.

4.4 Strong Consistency Replication Schemes

A strong consistency replication scheme ensures that replicas are consistent, i.e. mutually consistent and correct, between each operation on them. During the operation, there will be a non-negligible period of time in which this property is not valid, therefore it is important for a full consistency scheme to serialise operations to allow only one operation at a time. If an operation was allowed during the time the replicas were being manipulated it might observe incorrect data. From this definition it is clear that such a replication scheme has three main tasks:

1. A replication scheme's main task is to eliminate single points of failure by introducing redundancy. The scheme must be able to conceal failures so as to provide continuous service despite such malfunctions among the replica group. However, if there are many simultaneous failures, these might

[9] Although some schemes, e.g. process groups, can tolerate failing coordinators, they ultimately depend on a single client to make the fail-over decision [28].

[10] However, the encapsulation principle is relaxed in some object-oriented database systems [142].

have occurred due to a partition failure, in which case it must be ensured that at most one partition continues manipulating the replica group. Because operations on the replicas must be serialised, full consistency schemes can generally mask fewer failures than weak consistency schemes due to the risk of partition failures.

2. The scheme must ensure that clients only observe consistent data, which is the main benefit of full consistency schemes; the simple data model — namely that all the replicas contain correct data — means that programs using such schemes will not need to deal with stale or incorrect data and can be constructed as if they were not using replicated data. However, if the consistency requirements are not satisfied, programs expecting it cannot be expected to work correctly either.

3. The scheme must serialise operations on the replicas. The execution of operations on several replicas is normally not instantaneous, and therefore clients must not be allowed to observe the replicas during this period as they would then see inconsistent data, and hence evade the simple data model. Serialisation is typically achieved through some locking mechanism, e.g. by locking a majority of the replicas before carrying out a client request. Consequently, the need for mutual exclusion also limits the failure masking capability of the scheme (cf. item 1 below). Naturally, such pessimistic locking reduces the performance of the scheme in case of multiple competing clients, and may also increase the likelihood of deadlocks [85].

It will be explained how these three tasks are performed for each replication scheme presented. Numerous full consistency schemes have been discussed in the literature [79, 92, 162, 175, 181]. Arguably, the simplest way to achieve these two tasks is taken by the primary copy scheme, where only one replica is used at a time and the other replicas are used only if the first fails (see §4.4.1 below). Maintaining the consistency of this single replica is therefore simple, and serialising access to it is trivial. However, there are some problems with this approach which can arise during particular failure semantics. Another approach that ensures full consistency is voting protocols which achieves mutual exclusion of client requests by vote assignments (see §4.4.4) and relies on timestamps to detect the most recent copy of the data at the next operation.

General problems

The requirement that operations on the replicas are serialised restricts the performance of full consistency schemes, and generally, performance will not improve significantly by introducing more replicas although availability might increase. For applications that require the use of replication simply for performance or availability reasons, full consistency schemes are usually a suboptimal choice. Weak consistency schemes might be more appropriate (see §4.5 p. 49). Furthermore, in some systems, for example large federated databases, full consistency might not be practicable due to large numbers of partitions and low communication capacity [142]. Instead, weaker consistency must be tolerated, and optimistic concurrency control together with special conflict repair procedures should be used (see §4.5).

However, the performance of full consistency schemes can be improved by reducing the lock granularity during serialisation. The probability that two clients want to update the same item concurrently is thereby reduced, and the likelihood that one operation must wait for another will be lowered. For example, in the multi-user CES editor (Collaborative Editing System), documents are divided into sections, which are the units of serialisation [86]. While full consistency is ensured, people can edit the same document as long as they perform updates on different sections. Conflicts become rarer, and performance therefore increases among concurrently executing clients.

Additionally, full consistency schemes, due to the need to guarantee serialisation of replica operations, will incur extra communication overheads because many replicas must be locked to guarantee mutual exclusion. There is a chance that this extra load can increase the risk of network congestion and PE overload, and hence lower the reliability of the system (see §3.3.2 p. 36).

4.4.1 The primary copy scheme

The primary copy scheme, also known as the primary-backup scheme, is used in many existing systems, for example in the Thor database [111] and the Echo file system [181]. In both systems the primary copy scheme is used for replication of servers (object repository servers and file servers respectively). A primary copy scheme is often chosen due to the relatively simple behaviour and low overheads during periods of no failures. The scheme is also appropriate for data replication in object systems because it maintains a single, and thereby consistent, copy of the object graph[11].

The primary copy scheme is based on the idea of 'hot standby replication'; during fault-free periods, a single primary replica receives all the operation requests and carries them out sequentially. Serialising access to the replica is therefore a simple question of only allowing a single client at a time to manipulate the primary, and this can be done by simple locking primitives. A collection of backup replicas are periodically synchronised with the primary, and if the primary fails, one of the backup replicas takes over as the new primary. Although the scheme would work well for benign failures, malicious failures cannot be masked. If the primary fails maliciously, the client cannot generally determine this failure[12]. Additionally, the scheme can behave incorrectly during partition failures where several primaries can be elected at the same time, thus causing inconsistencies. Some variations of the primary copy scheme use just a single backup, in the form of process pairs [85]. Although the backup has a simpler job of deciding whether it is supposed to take over from a failing primary, conflicts are still possible if partition failures prevent the backup from determining the correct status of the primary. Echo, a fault-tolerant file system, solves the problem of electing multiple primaries by using a second level of replication; a primary is only elected if it manages to get a majority of votes from replicated disks [181]. Echo will thus block if a majority of disks are not responding rather than allocate two or more primaries[13].

The critical factor determining the scheme's performance is the frequency of synchronisation between the primary replica and the backups. If few failures are likely, the performance of the scheme improves with reduced frequency of synchronisation. However, when the primary fails, more work must be done to get the elected backup synchronised, and if the primary fails just after an update, but before it is synchronised with at least one backup, the scheme cannot provide full replication transparency to applications. Some data will have been lost, and the client of the scheme must therefore be prepared for this during interaction with the current primary. The scheme can be implemented such that the backups are synchronised for each operation that updates the primary, but this reduces the performance of the scheme to that of an available copies scheme (see §4.4.4 p. 48). Essentially, if the frequency of synchronisation is high, very little work needs to be done to switch to a backup, because the backup is likely to closely match the primary, but then the performance drops dramatically [94]. Additionally, as the primary receives all the requests, the scheme is not very scalable. The primary can very easily become a 'hotspot' and a bottleneck to performance.

4.4.2 Replicated RPC

Full consistency can be achieved in an object oriented system by sending the same sequence of invocations to each replica in the group, closely resembling the state machine approach to replication [162]. In addition to the process of replicating the invocations, procedures are necessary to mask failures and provide support for serialising access to the replication. However, because all non-failed replicas will remain mutually consistent, there is no need for potentially expensive replica state transmissions to bring replicas up to date as is the case for voting schemes. The RPC approach is attractive because it replicates

[11] Although, during synchronisation and fail-over the backup copies must be brought up to date.

[12] Naturally, proprietary failure detection mechanisms such as error-detection codes can be implemented, but this requires special design of the replica and is not enforced by the replication scheme.

[13] This illustrates a common tradeoff in distributed computing; liveness versus consistency [19]. Achieving one of them is simple, achieving both is hard.

the *invocation* and will thus normally avoid complicated object shipping[14].

A useful way to encapsulate this functionality is within RPC stubs [23] or proxies [167], which are commonly used paradigms for encapsulating distributed programming. This approach is very convenient and is used in a number of systems, e.g. it is known as MultiRPC [159] in the Andrew and Coda File System [157] and simply as replicated procedure calls in Circus [48, 49]. It is also the approach chosen in the proposed architecture. Replicating invocations to separate object replicas may trigger nested invocations however, and this complicates the design of a replicated RPC mechanism. For example, in [49], invocations are uniquely identified by low-level system code if they originate from replicas and a particular filtering mechanism at the invokee ensures that duplicate messages are ignored. A similar approach is followed in [121].

The main benefit of the replicated RPC approach is the simple programming model which is a copy of the standard RPC [23]. Replicated RPC can also give relatively good performance, even in large scale systems with a high number of replicas [159]. Automatically generated stubs interface with the client to provide a replication transparent object which can be manipulated much like a normal non-replicated object. Additionally, full consistency schemes are beneficial in object systems, as they do not conflict with the encapsulation principle (cf. §4.3.1 p. 44). A more thorough discussion of approaches to replicated RPC is in §9.1.1 p. 95.

4.4.3 Process groups

A process group is a synchronised collection of processes, where a message sent to one member guarantees it to be delivered by all or none of the group members [18]. Special broadcast protocols ensure that different members of the group receive the same sequences of messages by using some form of atomic broadcast algorithm. Additionally, a set of routines for changing the configuration of the group, group membership protocols, is defined within the protocol to provide support for reconfigurations such as new members joining the group. Due to the elaborate protocols for maintaining the atomicity property process group protocols generally do not scale to groups larger than, say, a few tens of replicas.

The process group approach can be useful as a tool for replication in object systems because it is guaranteed that each member of the group (i.e. each replica) receives all the invocations sent to the group. Process group protocols do not generally support nested invocations, rather, the approach makes it difficult to provide support for this. If replicated processes must invoke other shared processes, specially designed mechanisms must be used to filter out duplicate invocations. Additionally, a process group is a relatively low-level mechanism, and is not sufficient to implement a replication management scheme on its own. For example, messages can normally only be sent among members of the group. If another object, not belonging to the group, wants to send a message to the group, it has to use a special contact (a representative) for that group. This contact will obviously be a single point of failure, and the object must therefore implement a fail-over mechanism to guard itself from failing contacts (see also §9.4.2 p. 104).

4.4.4 Voting schemes

Voting is a kind of pessimistic concurrency control scheme for data replication, and achieves full consistency by ensuring that all operations on the data are serialised and that data are timestamped such that stale data are updated in subsequent rounds of the protocol [79, 92]. Serialisation in voting schemes is based on the principle of overlapping vote sets, or quorums. Before an operation can be executed on the replicas, a quorum must be obtained by the replica manager, and because the votes are arranged in such a way that only concurrent operations will be able to gather quorums simultaneously, strict serialisation of operations is guaranteed. Consistency is provided by a timestamping mechanism which is able to

[14] Object shipping is occasionally required if objects are parameters to the RPC and the objects cannot be remotely referenced.

distinguish between up to date and stale replicas. All replicas which are contained within the quorum are updated with the data from the most current replica, and then the requested operation is applied to the quorum. This enforces another requirement from the vote assignment; it must be guaranteed that each quorum will contain a replica with the most current data. Otherwise operations might be missed by some replicas. The fact that replicas are brought up to date by copying data from the most current replica makes voting schemes unsuitable for object replication regimes. Additionally, the problem of nested invocations is present in voting schemes as well.

The main drawback with voting schemes, is the high cost and relatively low availability as compared to weak consistency schemes. For each operation to be applied to the replicas, the appropriate number of votes must be gathered from the replica set. This can involve multiple rounds of potentially expensive remote invocations. However, the individual numbers of votes may be optimised to reduce the cost of certain operations [76, 79, 92]. For example, a replica known to be very reliable might be assigned a high number of votes so as to achieve better performance[15]. Additionally, it is common to exploit knowledge of read/update ratios to favour the most common operation type. If most of the operations are read operations, these operations can be assigned smaller quorums thus increasing the performance of reads by reducing the performance of updates. Another possibility is to exploit more advanced semantic properties of the data, for example by allowing increased concurrency for particular data types [92].

A widely known variation of the voting approach is the majority voting scheme. In a majority voting scheme quorums are simply determined as containing any majority of votes. The scheme does not optimise for reads or writes, but simply requires that $\frac{n}{2+1}$ votes are collected before an operation can be issued, and hence does not provide an increase in availability unless $n > 2$. However, for large n, the scheme offers good availability (see §B.2 p. 116). A majority, by definition, can be gathered by only one replica manager at a time, and thus guarantees mutual exclusion. As long as a majority of the votes are available, the scheme can allow for operations to proceed without sacrificing consistency. In its simplest form, majority voting assigns a single vote to each replica, however, different number of votes might be given to replicas to optimise the scheme. The weighted voting and the available copies schemes exploit optimisation of the vote assignments in slightly different ways.

Weighted voting is a specialisation of majority voting where different replicas are given different weights (or votes) [79]. This approach can be used to optimise the vote assignments, such that particular features of the distributed system can be exploited. Determining optimal assignment of votes can be computationally expensive, but less so than coterie assignments. This makes weighted voting more appropriate for larger numbers of replicas than coteries (see §4.4.5).

The available copies scheme[16] is a variation of the majority voting scheme, although highly optimised for read operations [55]. A replica is available as long as a majority is present in the current partition. Read operations are performed on the nearest replica (or the replica with the smallest cost of performing the read). Updates are performed on all available replicas in the group.

4.4.5 Coterie schemes

A coterie scheme is similar to quorum voting schemes in that serialisation is achieved through the locking of subsets of replicas. However, in contrast to voting schemes, coterie-based schemes have predefined sets of overlapping replicas. That is, instead of requiring the gathering of votes, sets of replicas are used to satisfy the execution of an operation. Replicas are arranged in sets, the coteries, so that two potentially conflicting operations have overlapping members [76, 17]. Similarly to the voting scheme, when a coterie is achieved, and an update operation is performed on the respective replicas, the replicas are also updated with a new timestamp which ensures that a following operation accesses the latest updated replica.

Although coterie schemes can give slightly better availability than voting schemes for high numbers of

[15] However, by directing more load to a replica, its probability of failure will increase as well (cf. §3.3.2 p. 36).

[16] Sometimes called the accessible copies scheme.

replicas[17], coteries are more expensive to compute [76]. However, determining the optimal assignment of coteries is computationally very expensive [17, 76, 175] making this approach most appropriate for smaller numbers of replicas (cf. §4.4.4 p. 47). A variation of the coterie scheme is the read-rows write-columns approach, where overlapping subsets are ensured through organising the coterie groups in a matrix. The fact that each row will overlap with any column ensures mutual exclusion.

4.5 Weak Consistency Replication Schemes

Strong consistency replication schemes depend on synchronous updates of replicas. In some distributed systems this synchronisation is too costly due to large numbers of replicas and relatively poor communication capacity. For example, federated multidatabase systems require, for performance and availability reasons, that global serialisation be relaxed [8, 142]. Other systems, e.g. systems where failures occur frequently or where replicas must be available during periods of disconnection makes full consistency unrealistic. Thus, a weakening of the consistency requirements is necessary.

Weak consistency replication schemes allow for asynchronous updates of the replicas. Some weak consistency schemes use epidemic algorithms for update propagation where replicas cooperate to 'infect' each other with the updates [57, 81]. Others, such as the optimistic protocol scheme, try to update as many replicas as possible, but in case of partitions a *precedence graph* is constructed which detects conflicting transactions [55]. Due to the asynchrony, an operation updating a replica need not wait for all other replicas to be updated, rather, the updates are propagated in the background while the client that issued the update can continue with another task. This means that if some of the replicas are unavailable, due to failures or disconnection, they can receive the update when they are once again available. However, weak consistency schemes rely on a mechanism to detect inconsistencies in the replicas that have been independently updated, and this normally requires substantial processing [55].

Weak consistency schemes in general incur lower overheads in the clients than full consistency schemes. They are therefore better suited for large scale applications using networks of 'data servers', for example global name and directory services [137, 41, 21], distributed file systems [157], and large autonomous information systems [100, 128] where replicas may be located in database servers with relatively high performance. At the cost of potentially stale data, weak consistency replication schemes assist in enhancing performance and availability.

However, weak consistency protocols have some limitations. Resolving conflicts can potentially be very expensive, and this requires more processing to be done in the replica servers. Due to the fact that replicas might contain stale information, weak consistency schemes must be used with care. Some applications are able to deal successfully with this problem, e.g. if it is obvious from the information content that it is stale the application can use another replica or simply retry the operation in the hope that the data will become valid soon. By reducing the granularity of the replicated item, the probability of conflicting updates can be reduced significantly. However, this comes at the cost of introducing larger numbers of items which is not always practicable due to higher overheads for a given amount of data.

Reconciliation

Weak consistency replication schemes require that mechanisms are provided to deal with potentially inconsistent data. Some conflicts can be reconciled automatically (and correctly) without loss of replication transparency. For example, some of the inconsistencies that might occur in replicated distributed file systems can be reconciled by the system following given rules. Consider a file system with the usual operations for reading and updating files and directories replicated at two PEs. Many of these operations commute, i.e. the order in which they are performed is insignificant and they can therefore be reconciled automatically by applying the missed operations. For example, the creation of two distinct files or directories, updating two distinct files, and simultaneously reading the same file are commuting operations.

[17] For more than five replicas the computation of optimal coteries can be *very* expensive, with exponential running times.

However, if the same file is updated independently in both replicas, knowledge not available to the file system is necessary to merge the updates.

When inconsistencies occur that cannot be reconciled automatically, a reconciliation authority is required. In the case of files in a file system, the content of the two conflicting files must be examined to determine how the reconciliation should proceed. The examination might be possible for a range of file types, e.g. structured text documents or source code where 'direct manipulation tools' are available [93]. For example, Lotus Notes arbitrates automatically among conflicting updates, and flags the overridden update [14]. Lotus Notes programmers are therefore required to construct programs which must tolerate sudden overwrites. However, such approaches are unsuitable in the context of programming-language objects. Due to the encapsulation principle, access to the internals of the objects is limited. The key idea behind encapsulation is that integrity constraints are enforced by the object itself, not by external objects. Besides, tools for investigating the internals of objects are generally not available.

A serious problem with weak consistency replication schemes is that some operations should *never* be allowed to happen in a partitioned network because they cannot (realisticly) be reconciled later [40, 55], for example the firing of a missile or the scheduling of aeroplanes at an airport. There might thus be a need to enforce different consistency requirements on different data, which further complicates the replication scheme. Additionally, the fact that weak consistency is sought, often implies that the potentially inconsistent data will be used outside the system [42], introducing covert channels which can confuse the causality property of the data.

Other disadvantages

The main cost of weak consistency schemes is the added overhead for logging of operations in the different partitions. However, in database systems this overhead might not be very significant as logging mechanisms are already present [55]. Secondly, reconciliation can be costly in systems with many updates during a partition. While the probability of conflicts can be reduced by reduced granularity of data, long periods of independent updates can cause large numbers of rollbacks which require further processing in the servers.

Variations of weak consistency schemes

A number of approaches to weak consistency replication have been discussed in the literature, but they are mostly based on similar ideas, e.g. epidemic processes depending on randomisation.

Version Vectors is an approach based on keeping track of the number of updates from each site holding a replica by recording pairs of *site,v* tuples in a vector, and using the notion of vector *domination* to flag a conflict. Domination follows from a replica having seen a superset of the updates of another replica. Because each replica is associated with a vector, a dominating vector implies that this replica has seen more updates than the other, and can be reconciled automatically. However, if neither vector dominates, a conflict has occurred and this must be resolved manually [55].

The *optimistic protocol* is an approach based on designing conflict graphs of transactions, and by analysing the graph, conflicting transactions are forced to roll back. The graph analysis is computationally expensive however, and implementations often depend on heuristics [55].

The *anti-entropy* approach is based on random selection of partners for exchange of new data [57]. Instead of propagating updates, two partners compare the entire dataset and resolve their differences after that. This makes the approach very expensive for any significantly sized databases. A variation of this approach, called timestamped anti-entropy, has been proposed by Golding et. al. [81]. While this approach is fine for data with simple semantics, e.g. a name database where weak ordering is sufficient, it is expensive for stronger ordering regimes.

In the *direct mail* scheme, upon receiving an update, a direct mail replication protocol notifies all other replicas about the update via buffered messages (mail messages) [57]. The update is asynchronous, i.e. the client does not wait for the messages to be propagated (it is similar to a best-effort multicast). The disadvantage with this approach is that there is a chance that a receiving replica is crashed or its message

is dropped by the network. The receiving replica will therefore miss the update and become permanently inconsistent. Due to this limitation, the direct mail scheme is sometimes combined with other, more fault-tolerant schemes such as anti-entropy [165]).

Rumour mongering is an epidemic-style update propagation protocol [57], but in contrast to anti-entropy it is not completely reliable. This technique is based on the idea that a new update is a 'hot rumour' that should be distributed to as many of the other neighbouring PEs as possible. The rumour gets 'cold' after a certain number of attempts to spread the update and there is hence a probability that a PE does not receive all updates.

4.6 Concluding Remarks

This chapter has shown that replication is a problem area with many conflicting concerns. Building fault-tolerant distributed systems is not cheap, but the added value gained from increased availability will become increasingly more important as our dependence on dependable computing systems rises. Similar to many other problem areas within computing, the tradeoffs depend on the system in which the problem must be solved. This dissertation is concerned with system support for object replication, and thus the requirements to transparency and a simple programming model are of paramount importance; scalability and availability are secondary concerns.

In this chapter strengths and weaknesses of several replication schemes have been presented, and quite clearly, none of the schemes stand out as 'the perfect solution' to any replication problem. Optimistic protocols maximise availability at the expense of repairing eventual inconsistencies. If conflicts are rare and the cost of resolving them is low, then they will offer benefits in terms of better scalability and performance. For some applications, resolving conflicts is cheap and the correctness of the conflict detection mechanism might not be of great importance. In some settings such as replicated name servers where data is only minimally encapsulated, a client will often be able to distinguish between correct and incorrect data[18]. The name server client can take corrective action by making repeated requests to the name server in the hope that the name server will eventually be updated. The client could also send its request to another name server agent.

However, in the context of system support for replication of generic objects — where any object might be replicated — it cannot be assumed that conflicts can be detected so easily, yet alone corrected. Pessimistic replication schemes do not require corrective action to be taken by clients as these schemes will always ensure consistent data. These schemes have reduced availability compared to optimistic schemes, but they are predictable and offer a simple programming model; the model the programmer is used to from writing non-replicated programs. Additionally, if conflicts are frequent and the cost of repairing them is high, then pessimistic schemes will offer better performance than optimistic schemes.

To summarise; pessimistic schemes do have significant associated costs. However, so do optimistic schemes, and in addition they cannot provide the transparency necessary in a system support architecture. Their cost will be unpredictable in systems where only weak predictions can be made about the frequency of conflicts. In the context of this work — system support for replication — maximum replication transparency is necessary. Any other approach would complicate the programming model rather than simplify it and thus counterfeit the goal which this dissertation was intending to achieve.

Based on these observations, a full consistency replication scheme based on replicated RPC has been chosen in the proposed architecture. Function shipping is a means of replication which corresponds well with the object-oriented system model, and does provide the best resilience against failures by also replicating the execution of objects' methods. From a system support view it offers a clean and transparent programming model which will be very useful to build system support mechanisms that can reduce the application complexity.

[18] For example, the reference obtained from the name server does not refer to an existing object any longer.

Chapter 5

System Support

This chapter presents the concept of system support mechanisms and motivates their availability during distributed software development. Different issues relating to the realisation of system support are discussed; in particular, those issues related to the provision of system support in distributed object systems are emphasised.

5.1 Overview

The rôle of system support is to provide abstractions of complex system structures and components to the application programmer and to increase reuse of common program patterns among multiple applications. System support is provided to the programmer in the form of APIs (Application Programmer's Interface) as collections of abstract datatypes, procedure libraries or class libraries. APIs can be made available to programmers through library code or operating system calls, facilitating reuse by a number of applications. System support functionality need only be implemented once, justifying increased effort to ensure high quality implementations.

System support is of greatest value if the same functionality is needed by many applications and when the functionality requires complex, error prone and substantial programming effort. Significant savings can then be achieved. Not only does it reduce the effort of developing each application, it can also increase the quality of the applications because the system support mechanisms are well tested. System support can help application developers concentrate on application functionality. Orthogonal issues, such as reliability and availability, are important for the satisfaction of non-functional requirements, and should as far as possible be delegated to reusable software components.

Furthermore, the demand for increasingly large and powerful applications necessitates reuse to ensure efficient realisation. Not only does efficient reuse lower implementation costs, it can also help increase interoperability between different applications. The use of standard, low-level system support mechanisms can provide the interface necessary for integration. This makes system support particularly interesting for the development of distributed applications [14]. Due to the complexity of distributed control, resource management and coordination, system support for these tasks can reduce application complexity and thus simplify their development [86, 170].

However, to be useful, a support layer must not introduce overwhelming additional complexity to the application programmer. If application programmers are required to familiarise themselves with many new and potentially radical concepts, or fundamentally change the way they reason about their programs, they might choose simply to implement the functionality themselves[1]. Also, system support mechanisms must be generally applicable to aid the construction of a wide range of applications. This might occasionally conflict with the goal of efficiency, as designs optimised for generality or performance may result

[1]The "not invented here" syndrome [72].

in different implementations [156]. To cater for this problem, application programmers should be allowed access to parts of the internals of system support mechanisms in cases when doing so does not jeopardise other important issues such as system security. Naturally, providing such access can only be realised with the understanding that system support integrity may be violated.

5.2 Providing System Support

System support mechanisms are mechanisms that are available to application developers to reduce the effort of developing software. Application developers on common computing platforms already have access to a vast range of software libraries and operating system calls [14]. As the demand for increased programmer productivity continues, the sheer volume of APIs is likely to rise. For the suppliers of these APIs it is therefore important to optimise their coherence and brevity so that the programmer is not inundated and hindered. During design and development of system support functionality, these, and other issues, need to be addressed carefully. This section elaborates on issues of importance to builders of system support software.

Generally, principles for good software engineering should be adhered to during development of system support mechanisms. For example, sound procedures for documentation and testing should be followed. Additionally, simplicity of design is more likely to increase implementation reliability. The fewer components a system support mechanism must incorporate, the greater the probability of a reliable mechanism. Furthermore, simplicity by design often leads to elegant and more easily maintained implementations.

5.2.1 Procedure and class libraries

With the increasing popularity of object oriented development techniques and programming languages, traditional procedure libraries have been partially superseded by class libraries. Class libraries consist of a collection of class definitions which can be incorporated into an application. For example, a supplied class can be specialised through inheritance with appropriate new methods and data fields. This kind of system support, i.e. language level software, provides a flexible and efficient approach to software development [75, 161].

However, in contrast to operating system support, this flexibility can lead to oversized libraries which can be very hard for the developers to completely understand. This is especially important for class libraries where classes might have complex inter-dependencies, such as libráries for window systems like Microsoft Windows [30] or X11.

5.2.2 Operating system support

System support mechanisms might occasionally have to be integrated within operating systems due to requirements for security and access to low-level devices. However, implementing system support at this level has a tendency to swamp the operating system. A goal among operating system designers is to provide only the most essential services which require protection and access control as kernel functionality, for example; processes, inter-process communications and address space administration [107]. Remaining system services can be implemented outside the kernel in user level processes. Furthermore, operating systems are required to support large, existing bases of software through system support mechanisms. Therefore, operating systems are forced by the application base to restrict changes in system call interfaces to a minimum.

There is research into object oriented operating systems, promising easier customisation and more flexibility for programmers, e.g. Spring [125] and Spin [15]. Object oriented operating systems try to reduce the traditional tension between generality and specialisation, essentially providing the benefits of language level support with the added facilities of protection and resource scheduling [95]. However, it is likely

to take some time before such systems come into widespread use, and it is likely that the first ones to appear will have standard supplied interfaces for popular operating systems, e.g. UNIX.

This familiar operating system, dating back to the 1970's, provides its support to programmers in the form of a collection of system calls, i.e. application callable procedures which are executed by the kernel. The amount of software written on this platform would make any radical changes to the system call interface unthinkable.

5.2.3 Stability of interfaces

System support closely relates to layering in software engineering, as a collection of system support mechanisms can be seen as a layer. Implementing system support mechanisms becomes the process of building such a layer and documenting its interface to the application builder.

As more software is developed for a particular system support layer, it becomes harder to change it. Small changes in the API can lead to cascading changes being necessary in the software written for the API. It is important to consider this during development of low level software. Low level software interfaces must be designed for stability and longer life than application level software. However, this does not affect the implementation. The implementation of the mechanisms are allowed to change independently of the interface.

An inherent problem with system support mechanisms is making them applicable to the widest possible range of applications. Reuse, in practice, is difficult, and often, several iterations of refinement may be necessary to end up with good, generic abstractions [75, 170]. There is an inherent conflict between the stability of the interface and the functionality it implements, however, as the functionality underneath the interface changes, the interface might not be the optimal interface to this functionality any longer. Research and experience with reuse, and patterns in particular, will need more time to mature and clearly demonstrate their benefits [73].

5.2.4 Conflicts and overlaps

Having large numbers of interfaces introduces problems with conflicts and overlapping functionality. For example, if two interfaces are used simultaneously by a programmer, slight differences in their conceptual models might severely increase the effort needed to build reliable code. This motivates strict adherence to software architectures; not only their concrete representation, but also their underlying assumptions. Lack of common understanding of the assumptions made during design of reusable software components appears to be a very significant problem [54]. Design decisions should be clearly documented to reduce the danger of mismatch between software components. This is most important for the part of the reusable software which defines the interface to other software packages. For example, if one software component assumes that all I/O is via pipes or files and the other component assumes RPC calls, then clearly the effort needed to make the components interact can be substantial.

Additionally, using interfaces from multiple suppliers, the programmer might experience namespace conflicts. Such conflicts can complicate the development process. The advice from T. Vayda is to check with suppliers in advance and use tools for namespace control [189]. The same advice is probably well worth considering for developers of such interfaces as well. Additionally, organisational naming conventions, for example using a global naming space strategy as in Java [82], can help reduce this problem.

5.2.5 Visibility of code

System support software cannot be expected to be free of bugs, although due to its importance, it should undergo very thorough testing. For application programmers it is important to be able to determine the source of faults in their software. However, system support software may make this difficult due to lack of access to the source code. It may be based on code precompiled into libraries which are not generally

open for inspection, or the code might simply be compiled into the operating system itself, where it remains invisible to the application developer. In some circumstances it may therefore be beneficial to supply not just the precompiled code, but also the source code of the system support software to the application developer.

Some system support software is based on preprocessing using automatic code generators. In this case it should not be assumed that the generated code is always correct. If the generator produces code which is not accessible to the programmer, errors in the generated code cannot be corrected by the programmer. Occasionally, the programmer might want to perform corrections, or at least make sure that errors in the implementation are not due to the code generator. Any intermediate code, e.g. output from stub generators, should be visible. However, because of the automation, any corrections made by the programmer will be over-ridden by subsequent invocations of the code generator. These corrections can only be undertaken by the supplier of the code generator, but the programmer might have a better chance of producing a useful fault-report for the supplier.

Another problem with generated code is that it might be difficult to read by humans. Automatic code generators cannot always be expected to produce code with intelligent variable names and readable comments.

5.3 System Support in Distributed Object Systems

In addition to adhering to the principles discussed in section 5.2, system support for distributed object systems must assist the development of distributed object oriented applications. A model of what distinguishes a distributed application from a non-distributed application is therefore necessary[2]. A particular feature of these applications is that they contain concurrently executing objects on physically dispersed computers. Any object might therefore be invoked by several other objects simultaneously, and hence, reliable and efficient sharing of objects must be supported.

5.3.1 Object sharing

Sharing of objects is facilitated by passing references to objects among other objects, and references can be passed among objects as invocation parameters or invocation results. Any object holding a valid reference can invoke methods upon the referenced object. An application will therefore typically consist of a collection of objects which invoke methods on each other, and which share objects with other objects. If correctness is to be maintained, this sharing of objects requires coordinated access, i.e. concurrency control [53].

A system support facility should be safe in the presence of multiple threads, i.e. it should behave correctly independently of the number of concurrent clients. Special care must therefore be taken during the development of system support mechanisms for these kinds of systems. The programmer should be shielded as far as possible from the fact that there are other programs executing in the system. However, unnecessarily complications of the programmer's model should be avoided and functionality related to concurrency control should be made as transparent as possible.

5.3.2 Concurrency control

This dissertation is concerned with applications written in a distributed programming language using lightweight concurrency primitives, e.g. threads synchronised using locks [25]. In contrast to distributed programming languages, distributed programming systems extend the programming model with support for persistent objects and more sophisticated concurrency control inherited from database technology, such as atomic transactions and recovery functions [9, 154]. A distributed application containing several

[2]The application model adopted in this dissertation is described in §2.7 p. 24.

threads of execution must be designed to avoid dangerous situations such as deadlocks, livelocks, starvation, and race-conditions occurring due to the concurrency [25, 29]. In distributed programming systems this is the responsibility of the application developer.

Objects can be considered as a unification of the data and process concepts; each object contains its own 'processor' (the thread) and the data upon which this processor executes (the object's internal state). This model is called an active object model, and is implemented in some programming languages such as Objective Linda [101]. However, the unification concept is valid even if the actual programming language does not restrict threads to execute only within a single object[3], although more care must be taken by the programmer when building concurrent programs in a passive object model[4]. In fact, in an active object model, the system itself is responsible for synchronising access to the object's state. The benefits of this unification become clearer when concurrency must be controlled, as each object can be held responsible for maintaining the correctness of its own state. In a passive object model, objects which require synchronised access are programmed with this in mind, and the other objects do not need to pay the performance overheads associated with synchronisation.

Distributed programming systems often include support for transactions and persistent objects [154]. For example, Argus implements transactional semantics on distributed data-objects using the notion of Guardians (objects) and Actions (atomic invocations) [110]. However, due to the need for stable commit and abort mechanisms, Guardians are relatively heavyweight, and can easily impose too much overhead for interactive applications if they are not carefully designed such as to minimise the use of guardians for frequently updated data [86]. Similarly, the Arjuna system groups object invocations into atomic actions using specific programmer declared primitives [171]. Arjuna uses a persistent object store which maintains the state of objects (state servers), and assumes that particular object servers contain the code for the methods. For each update to an object, the new state of the object must be forced to disk, and this approach also results in relatively high overheads for fault-tolerant objects [113]. Transactional support is also necessary for dealing with nesting of invocations; an invocation which trigger the invocation of multiple child-invocations may not need to be aborted if only a subset of the child-invocations abort. However, a thorough discussion of transaction models is outside the scope of this dissertation (see e.g. [9, 85, 154]).

Care must therefore be taken when providing system support for concurrent programs. In programming systems, concurrency control is a central part of the system, commonly provided through some form of transaction mechanism. In programming languages, the programmer is normally left with more of this responsibility, and the amount of work is dependent on the object model. In an active object model, the programming language is responsible for ensuring serialisation in the presence of concurrent threads. In contrast, a passive object model, which is the most common model today, requires that the provider of system support functionality ensures correct operation during concurrent processing.

5.3.3 Other services

Other services are also important within the framework of system support in distributed object systems. Applications commonly require system-wide services which cannot easily be implemented by each application, and should rather be accessible through reusable components. Examples of such services are name services, binding services, RPC-facilities, garbage collectors, and load balancers. Although this dissertation is primarily concerned with replication facilities, these will become only services within application development frameworks [14].

[3]Which is the case in the passive object model adopted here.

[4]Active objects do however introduce another point of failure. Because an active object must accept invocations asynchronously, it must either be augmented with some form of message buffer or create new objects with their own threads dynamically [154]. The message buffer can potentially overflow, and the pool of threads can grow without bounds, if care is not taken to introduce some form of flow control. In a passive object model, a thread is typically blocked on a lock before entering the object and flow control is in this case provided by the mutex itself.

5.4 System Support for Object Replication

As software becomes more distributed, more people will depend on the same application, and its availability requirements are likely to increase. In §3.1 p. 27 it was indicated that a high level of interaction among components in a distributed system makes distributed software vulnerable to even single failures. System support for object replication could be a significant benefit to alleviate this fragility, and could help to reduce the difficulty of developing more reliable applications. The system support mechanisms proposed in this dissertation (chapter 6) are suggested as a possible, although partial, approach to object replication.

Object replication implies replication of objects not specifically designed for replication (see §4.3.1 p. 44). Ultimately, it should be possible to replicate any object within a distributed system, and do so transparently for the application programmer. Many parameters in a distributed system are very dynamic, and it would be unrealistic to demand that application developers deal with these [114]. Rather than burden the programmer with tasks such as replica placement, failure masking and replication protocols, system services should perform these functions automatically based on simple metrics such as the desired object availability.

However, system support for object replication is a challenging problem [59]. While data replication can be supported in a generic manner, object replication does incur tradeoffs for transparency. Successful implementations of such mechanisms can therefore only be provided after making careful judgements of the inherent tradeoffs. Furthermore, replication management is itself non-trivial, requiring both distributed coordination, inter-object communication and distributed resource management. It would be unrealistic to assume that application programmers would have the resources necessary to implement replication schemes with the appropriate reliability [18, 44]. Hence, there is a danger that the implementation of the replication mechanisms themselves cause failures, reducing the benefits of replication. However, system support mechanisms, due to the benefits of reuse, can justify the cost of higher quality implementations.

5.4.1 Replication transparency

The conflict between consistency, performance and availability is inherent in replication schemes. For a system support layer, consistency is the factor which determines its usefulness as increased consistency implies increased replication transparency. Further conflicts arise in object systems due to complex interactions among objects in such systems, and the encapsulation principle, rendering voting and coterie-based replication schemes unusable for object replication (see §4.3.1 p. 44).

The achievable transparency in object replication schemes is limited [59]. For example, invocations from a replicated object A on some other shared object B will result in multiple, potentially unwanted invocations on B because each replica A_r of A holds the reference to B. However, if the invocation from a replica A_r was to a non-shared object, e.g. a temporary object created by A_r itself, there should be no change in semantics, and all invocations from A_r should reach B.

The problem is that the wanted behaviour is application specific; firstly, if B is not shared, no particular action needs to be taken. In the case that B is shared, maximum fault tolerance is achieved if as many methods as possible are invoked on B, and this does not cause any semantic violations as long as these methods are either reads or pure overwrites. Multiple invocations would not change the final state of the invoked object B (assuming overwrite methods are executed at least once). However, in the case where objects mutate, i.e. the methods are non-idempotent, the number of invocations is significant for the resulting state in B. According to the object model adopted in this dissertation, any method may potentially cause a mutation in the invoked object, and the builder of A objects should have to worry about the internal semantics of B objects.

A mechanism which automatically 'filtered' invocations from a replicated object could solve part of the problem, although at the cost of added programmer complexity and filtering even invocations to non-shared objects [121]. Additionally, multiple invocations might be what the application semantics dictate. Consider for example an object which counts the number of replicas which correctly executes the call.

Filtering all replicated invocations to shared objects would then prevent the correct behaviour of the program. Only the application builder can determine the correct semantics.

Transparency disadvantages
Even in a system support setting there are valid arguments against full replication transparency. Essentially, these arguments have much in common with those for and against completely distribution transparent systems [51, 166, 183, 194]. For example, replicating an object changes its failure semantics, because a replicated object cannot conceal all possible combinations of failures. During periods of many failures, the replicated object might be unable to execute any method calls and would in this case have to report an error or simply block while waiting for failure recovery. This would never happen using a local object[5]. The programmer might want to know which objects are replicated so that the program can be built to resume execution despite blocking calls on certain objects.

Worse still, if the replicated object fails, rendering the application's reference to it invalid, the application might fail too if it didn't contain code to deal with the exception. A tradeoff between transparency and application complexity therefore seems inevitable. For example, the approach taken by the system support mechanisms presented in chapter 6, is to let the application developer decide which objects should be replicated and to expose the new failure modes.

5.4.2 Applicability

Due to the inherent conflicts between transparency and availability in replication schemes there will undoubtedly be certain groups of applications which can be supported more efficiently than others by a particular replication scheme. The wide range of application classes developed in distributed systems makes it difficult to find a common substrate of replication functionality useful to the whole range. Clearly this reduces the usefulness of a system support mechanism as it would be advantageous to support any class of application with the same support layer. However, if assumptions are clearly stated, the application developer can make a conscious decision before starting to use a specific support facility.

Applicability also deals with concrete assumptions such as programming language bindings, data models, existing system support functionality, system management interfaces and protocols [14, 54]. It is crucial that such assumptions are made as explicit as possible. The conformity assertions for the architecture proposed in this dissertation are presented in §6.1 p. 60.

5.4.3 Increased availability

System support for replication would not be very useful if it didn't increase application availability by masking failures. For an application designer it might be important to determine how much more available the application will be [150]. This necessitates that information is available about the particular replication scheme used, such that reliability predictions can be made accurately. As discussed in chapter 4, tradeoffs between availability and consistency are inevitable in replication schemes, but for the strong consistency scheme used in the presented architecture, availability can be reasonable for many applications (cf. §B.2 p. 116).

5.5 Concluding Remarks

This chapter has presented some of the issues to be considered before embarking on the task of constructing system support software. Although there are a number of challenging problems, system support can

[5]An address space is the failure domain of an object. A local object resides in the same address space as the caller (see §2.7.2 p. 26).

bring large benefits in terms of reduced application complexity and consequently help increase productivity. System support for object replication will be particularly useful as a means of increasing application dependability. Although some compromises on support transparency are necessary, the value of increased fault-tolerance can not be ignored.

This chapter marks the end of background and motivating discussions in this dissertation; in the next chapter the proposed system architecture for system supported object replication is presented.

Chapter 6

System Architecture

This chapter presents the architecture of the proposed system support mechanisms. The presentation covers a range of aspects; logical partitioning into modules and objects, internal functionality such as performance and synchronisation issues, failure resilience, and the realisation of the architecture in a real distributed system including issues such as the physical mapping onto, and requirements which must be met by, the existing infrastructure. Limitations of the architecture are also discussed, and enough detail is provided to facilitate further refinement and extension. Read in combination with chapter 7, which describes the programming model of the architecture, it will give the necessary understanding for using the system support mechanisms.

6.1 Overview

Object replication deals with the replication of objects not specifically designed for fault-tolerance. That is, the programmer should be able to define objects as if they were not replicated. Similarly, modifications should not be necessary in already defined objects if they are to be replicated. Similarly, other objects, which use replicated objects, should not have to be modified either. Thus, this architecture is distinct from research efforts such as Adaptable Replicated Objects [31] and the earlier work on Fragmented Objects [117] where the replicated objects must be explicitly implemented with replication in mind.

The architecture partially fulfils these goals in the sense that only limited adjustments of application code are necessary to adapt non-replicated objects for replication. A tradeoff has been made between efficiency and generality. For example, the programmer is responsible for indication of classes that should be replicated, thereby complicating the programming model with the benefit of only incurring extra costs for a subset of the defined classes. The alternative, making all classes replicated, would be very inefficient as the dependability requirements for individual classes are likely to vary and the programmer's opportunity to optimise the application would be reduced.

The proposed system support mechanisms encapsulate replication functionality within particular surrogate objects which to the application programmer appear very similar to the equivalent non-replicated objects. This reduces the amount of modifications necessary in the clients of the replicated objects. Issues related to programming with replicated objects are discussed further in chapter 7. The rest of this chapter is primarily discussing the internal structuring of the replication mechanisms.

Applicability
System structures similar to the one outlined in chapter 2 are assumed to be the platforms on which the replication mechanisms will be used. Most importantly, programming language objects are the unit of replication in this system, and a number of distinct computers with independent failure modes are able to support the execution of methods in locally stored replicated objects. Thus, the architecture advocates process distribution by following the usual function-shipping principle in object oriented systems [38, 162].

Objects are stationary, and method invocations are passed among the replicas as messages on the network.

The replication mechanisms, being quite lightweight, are primarily aimed at supporting the development of distributed applications with high availability requirements without incurring severe overheads. They will also be useful for adding fault-tolerance to existing applications, for example when they are redesigned to run in a networked environment. However, as these mechanisms offer relatively low-level services, their greatest benefit will probably be as a component within a complete distributed computing environment, such as a middleware framework [14]. Distributed application development requires services not directly supported in this architecture, such as authentication services and access control, which could then be used to provide a more complete application development framework.

The architecture is designed to support the construction of relatively small scale distributed software systems such as groupware or other multi-user data-sharing software used in LAN-like networks. It is assumed that users are most dependent upon servers local to the network, and that it is the servers themselves which are the most common cause of service outages [196]. By replicating server functions within the system, significant savings may be achieved. However, larger scale application frameworks could be built on top of these mechanisms, for example by using more loosely coupled clusters of networks utilising this architecture internally to provide high local cluster consistency. Communication latency is the primary performance bottleneck in this architecture, application developers must therefore carefully consider how the latencies of communication in a particular network configuration will influence application performance. Different network characteristics will influence issues such as replica placement and the number of replicas used in a particular application context.

The failure-resilience offered by the architecture reduces the frequency of application restarts necessary due to failures. Due to the consistency constraints enforced by the object model, the mechanisms are not suitable for large scale systems, where components are geographically widely dispersed and problems such as disconnected operation and long communication latencies must be weighted against full consistency (see §6.3.2 p. 70). Consequently, applications using replication primarily for reasons of autonomy and high performance are not adequately supported by this architecture although performance for asynchronous invocations on objects is good (see §6.3.4 p. 74).

The pessimistic concurrency scheme will be inappropriate for some classes of applications which might require long-duration transactions [99], such as software engineering tools and multi-user CAD systems where direct access to data is most important, and the encapsulated object model adopted here would be in the way for efficient data manipulation. Also, such applications tend to require more fine-grained concurrency control, for example by distinguishing between read and update operations on the data. Majority locking would in this case severely reduce performance of the application. This architecture is better suited for development of service-functions within a distributed system, for example information management or system management services such as name services or accounting services [14]. The architecture is based upon the assumption that object invocations are short-lived and that multiple clients avoid long periods of exclusion from the object replicas. However, investigation of concurrency schemes allowing more concurrency is an interesting topic for further research (see §10.3.2 p. 109).

Additionally, the system architecture does enforce some limitations on the structuring of the application. Primarily, object replicas may not invoke non-idempotent methods on other shared objects as this will result in multiple (possibly harmful) invocations on the shared objects. An extension of the architecture to remedy this limitation would probably lead to less replication transparency for the designer of the objects to be replicated [59, 121], but support for this should be considered in eventual continuations of the architecture (cf. §10.3.1 p. 109). By partitioning the application into separate object graphs located in disjoint address spaces this problem is avoided, but naturally complicates the programming model and restricts the applicability of the architecture. See §6.2.1 p. 63 for a more thorough discussion of these issues.

6.2 Main Components

The system support mechanisms are built as a collection of reusable software components which interact to support the replication of objects. A major feature of this proposal is the separation of generic and application specific components into distinct entities. Surrogates are generic objects which are automatically generated from interface definitions, and collators are application specific and used to customise processing of replica result data. Surrogate objects encapsulate the parallel invocations and the collators, and appear to the programmer as normal, non-replicated objects. Collators allow the programmer to tune the synchronisation and failure-masking requirements as demanded from the application in a simple yet flexible manner. Surrogates and collators are described in more detail in the following text.

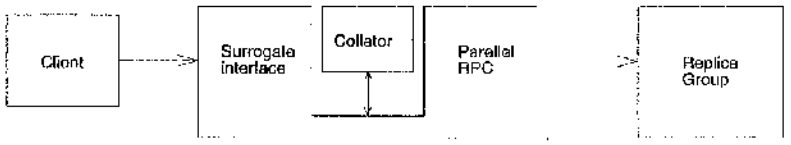

Figure 6.1: Overview of the main components.

Figure 6.1 illustrates the interaction among the main components. The surrogate is invoked by the client as the normal non-replicated object, and the surrogate manages the parallel RPC module. The programmer is responsible for 'slotting in' collator objects as necessary (cf. §6.2.3 p. 65).

6.2.1 Surrogates

A natural and simple extension of the object encapsulation principle is adopted in the system architecture. Particular surrogate objects hide the details of replication from the programmer while acting as a manager of a collection of object replicas. The surrogates described here are similar in principle to Gaggles [28] in that they encapsulate multiple other objects using a single object. However, the Gaggle is a more generic abstraction than the surrogate. Gaggles are not directly designed for object replication because it is assumed that only one, the *clerk*, of the multiple objects should receive an invocation. This is appropriate within a primary copy scheme, or as the fail-over mechanism required in a process group used for active replication. Because both the primary copy and the process group scheme work by fail-over in case of failures in the currently used primary or contact group member respectively, the Gaggle is well-suited for this task[1]. In contrast, the surrogates described here also encapsulate functionality for failure masking and consistent updates of object replicas and are therefore specialised for replication.

Programmer interaction

A surrogate defines the same methods as the object replicas, and the programmer can therefore manipulate the surrogate much as if it was an ordinary, non-replicated object. Maintaining the illusion of only a single object increases the transparency of replication. This means that only small modifications are necessary in the application code if it is adapted to use objects that have been replicated. To further simplify the use of these replication mechanisms, surrogates can be generated automatically by a stub generator given the interface of the object to be replicated.

However, surrogates exhibit a somewhat richer behaviour than non-replicated objects due to extra failure modes in the replicas it manages. More details on how the application developer can use this architecture are presented in chapter 7.

Other benefits of surrogates

Surrogates, similar to proxies [167], are commonly used in distributed systems because they introduce an

[1] Gaggles are described in more detail in §9.1.2 p. 96.

extra level of indirection, thereby allowing additional functionality to be added without making modifications to neither the client or the original object. In addition to hiding complexity such as functionality for replication, they are also useful for other tasks such as implementing access control policies and caching [75]. For example, a surrogate might support only 'public' methods on a particular object which would prohibit the client from calling 'protected' methods. The surrogates used in this system could be extended to support such tasks.

Due to the extra level of indirection, a local surrogate object can act as a forwarder of messages to remote objects, such that the object it represents can be manipulated as if it was local [23, 58]. The surrogates discussed here are always local to the client object and they relieve the programmer of some of the communication aspects of remote method invocations such as argument marshalling and unmarshalling of method arguments and results.

New failure modes

Normally, clients use a surrogate as if it was a normal, local object. Local objects fail[2], if and only if, the address space fails. However, by hiding the distribution aspects of objects, new failure semantics appear in the surrogate. For example, the surrogate might be unable to carry out an invocation if too many of the replicas have failed. For convenience, the surrogate returns exceptions if this happens which the programmer can use to detect and possibly correct surrogate failures. The programming model, presented in chapter 7, discusses how to handle such exceptions in the application (see §7.7 p. 86).

Problems with nested invocations

Objects can contain multiple references to other objects as part of their internal state. Additionally, multiple objects may hold references to the same object. Within a program there might therefore exist graphs of arbitrarily interconnected objects. This exposes a limitation with object replication as envisioned by this architecture, namely that object replicas which hold common references to another object will cause multiple invocations in the shared object. Maximum fault-tolerance is achieved by transmitting multiple invocations from the replicas, and performing it multiple times in the shared object. As long as the invocation on the shared object is idempotent, this is the desired behaviour, and this is what would happen using the proposed architecture. However, if the invocation on the shared object executed a non-idempotent method, the program would behave incorrectly.

Solving the problem requires involvement from the programmer. Depending on the semantics of the application, the programmer could distinguish between calls to shared and non-shared objects. Method calls to shared objects could then be 'coordinated' or 'filtered' such as to only execute a single invocation on the shared object (cf. §5.4.1 p. 57). Arguably, this reduces the usefulness of the replication scheme by requiring the implementor of each class to distinguish between different method invocations. In fact, the basic idea of making object replication fully transparent seems to be impracticable due to this rich behaviour of objects [59]. However, investigating the possibilities of automating the process of distinguishing coordinated calls from non-coordinated calls might be an interesting topic for further research within this architecture (cf. §10.3.1 p. 109).

Despite this fundamental limitation in the architecture, it is still believed to be useful. The problem only occurs if replicas share another common object, and this can be avoided if the application is structured into separate object graphs without inter-graph references (similar to *troupes* in [48]). There are at least three benefits stemming from keeping separate object graphs on distinct PEs.

1. Better locality which increases performance. Compared to local invocations, remote invocations are very expensive [40, 84]. Because the objects in such an object graph will be local to the same PE, fewer method invocations will have to be executed over the network. It is indicated in [4] that most applications are in fact structured this way to maximise local processing.
2. Reduced overheads incurred by the replication protocol because of the small number of surrogates

[2]Failure here means that the object does not behave according to its specification.

needed. If the application is structured as a collection of large separate object graphs, only one surrogate is required for each set of object graphs.

3. Shared objects, which would cause this problem, introduce single points of failure and reduce the reliability of the application, so should be avoided. It is important to maintain replication at all system levels to increase reliability (cf. §6.4.2 p. 76), and in this respect, shared objects also include non-object data, such as files.

Structuring applications in such a manner could be appropriate for implementing larger-scale modules within a system. The modules would then benefit from the three properties mentioned above and form an efficient, yet manageable unit of replication. However, this restriction is a complication of the programming model, and does restrict the programmer during construction of the application (see §7.2 p. 79).

6.2.2 Parallel invocations

The system support mechanisms are based on the assumption that replicas are hosted by individual PEs. Method invocations on the replicas can therefore be executed in parallel rather than in sequence, potentially gaining performance benefits leading to reductions in service time. Although parallel execution of methods will require processing capacity on all the PEs hosting replicas, studies done elsewhere indicate that, for example, workstations are commonly severely underutilised [62]. This dissertation assumes that this is the common case, so PEs in the network have the spare capacity necessary to execute additional object method invocation.

Parallel invocations are designed as a collection of independently executing threads within a surrogate, similar to MultiRPC within AFS [159]. However, rather than being restricted to always waiting for all threads to finish, each thread is associated with a shared collator object which controls the level of synchrony appropriate for the client (see §6.2.3 p. 65).

Orphan computations

It is assumed that replicated classes implement deterministic methods which eventually will complete, and that replicas execute identical methods. However, jobs will be less than optimally shared among the PEs in the network, leading to unpredictable execution speeds and some replies from replicas will be received long after the collator has returned its result to the client. In particular, if the method call is executed using a wait-for-first collator (see §6.2.3 p. 65), invocations are left running in the replicas after the client request has returned, and this in turn creates orphan computations. The problem with such late replies is that the surrogate cannot continuously execute new requests from the client as this might cause concurrent execution of methods in the slower object replicas. Depending on the client object(s), the rate of requests to the surrogate can become too high for the slower replicas to keep up, and the slowest replica will become a bottleneck for performance in the surrogate. This argument favours smaller grain surrogates which would increase concurrency and reduce the probability of overloading single surrogates. The programmer should recognise this fact, and make careful judgements depending on application characteristics (see also §7.2 p. 79).

It is important that late invocations are not simply abandoned. The probability of late invocations being delayed due to failures is significant, and even if the client has already got its result from the collator, the surrogate must still be available to diagnose potential failures in late replicas. In the architecture this problem is addressed by not allowing the processing of a new client request before the replicas are finished processing the previous request. Used in conjunction with a wait-for-first collator, the surrogate ensures this by waiting for the outstanding invocations before allowing another. Naturally, if some of the remote invocations crash, the surrogate will wait only until the corresponding timeout expires before releasing the lock on the replica. In the case that multiple surrogates must be coordinated, the majority locking scheme guarantees that a new request is not carried out before all replicas are finished processing

the previous request. A majority of locks will not be available until the previously active surrogate is finished.

6.2.3 Collators

The system support mechanisms allow the programmer to control the synchronisation among the replicas based upon the application's requirements. Particular *collator* objects, based on an idea by E. Cooper [48], are used to process the results returned from the method invocations on replicas, and can be specialised to support different method result types. For example, collators for the processing of integers, real numbers and strings require different implementations. Adhering to sound software engineering principles, collators are able to reduce the amount of application specific knowledge within the support layer without significantly complicating the programming model.

The collator design described here is an extension of that described by Cooper, such as to make them useful in an object oriented programming language. Most notably, this collator design covers the generation of new surrogates containing object reference return types. A collator, ignoring the parallel RPC connected to it, is also similar to *promises* and *futures* [4], but are not automatically generated as in Argus or the CRONUS System. Rather, the programmer is responsible for defining collator objects.

A collator has a simple and understandable interface which makes them easy to use. This is achieved by locating generic and complicated code in the surrogates themselves which can be automatically generated, and locating application specific code in the collators. Examples of useful collators which can be easily implemented are:

1. Return on first result. This collator might be useful for methods on objects where low latency is more important than error detection and correction [48, 64]. Additionally, some object methods might not return equivalent results despite being correct, e.g. if the value returned is determined from random number generators or local timestamps. In this case the first result is just as correct as any other.

2. Wait for all results, returning the most common value (i.e. a voting collator). If arbitrary behaviour is expected from any of the replicas a collator may implement a voting process on the results.

3. Wait for all results, and return a processed average value. This collator might be useful for methods where exact answers are less important than statistically sound answers. For example, a process control application which takes input from a number of replicated sensors might want to weight the sampled values to increase confidence in the data.

It is worth noting that collators of type 3 and 2 implement resilience against failures in the data domain, e.g. Byzantine failures (cf. §3.2.3 p. 32). A voting collator can tolerate t data-domain failures among $2t + 1$ replicas.

Although all correct replicas receive the same sequence of invocations, the collator defines how many invocations must finish before the method invocation on the surrogate returns. For example, if only a single return value is needed, the collator waits until just the first invocation finishes and then returns the answer to the caller (the client). This will alleviate parts of the performance problem normally found in full consistency replication schemes by allowing the client of the surrogate to continue before all invocations are completed [64]. The throughput of the surrogate is not increased; a new invocation on the surrogate cannot commence until all[3] the invocations from the previous invocation have returned.

Composite return types

The usefulness of the collator design becomes clearer when methods on replicated objects return more complicated types. While single types such as integers and real numbers can undergo quite generic

[3] If replicas crash during an invocation, "all" means all non-faulty replicas.

processing such as weighting and majority voting, types such as strings, records and arrays normally require more specialised treatment because operations such as weighting and majority are not immediately obvious for these types. The correct processing of values of such types could not realistically be automated, the programmer must therefore be given a simple mechanism to handle such results. Collators allow the programmer to provide such refined processing methods within an encapsulated module (cf. §7.5.2 p. 83).

Object reference return types
Reference types as results give the opportunity to increase the functionality of collators beyond what is possible in Cooper's system model [48], which is not based on object-oriented technology. Because a programmer observes a surrogate as a single local object, reference return parameters would create a semantic mismatch if a set of references, referencing remote replica objects, were visible as individual values. Furthermore, it would make no sense to perform voting or weighting of references, as they are intrinsically unique. Intuitively, a collection of references returned from a replicated invocation should be treated as a new surrogate, thus triggering automatic creation of a new surrogate object which would be returned as the result of the invocation.

For example, if a replicated class defines the interface of a replicated file server, invoking a method *open(name:FileNameType):File*[4] on a surrogate of this class would return a new surrogate acting as the manager for a set of file objects. The code generator must therefore recursively generate surrogate code also for reference types as return parameters from methods in classes tagged as replicable. The programmer can then easily construct collators which return new surrogates *managing* the references returned from the call (cf. §7.5.2 p. 83). Such collators would most likely be of a type which waited for all results from the replicas so as to maintain the availability of the new surrogate. By giving it as many replicas as possible the maximum resilience against failures is achieved.

Backdoors
System support can never be completely generic, some applications might want to implement slightly different mechanisms than the ones offered. In light of the end-to-end argument [156], the architecture allows control over lower level abstractions. Collators may be used more primitively to return sets of replica results back to the client. This might be necessary for some applications that require more detailed control over the object replica results (see §7.5.2 p. 83).

6.2.4 Object replicas

Although the replicated objects themselves are not directly part of the proposed system architecture, they are discussed here as the architecture enforces some requirements on them. Because object replication deals with replication of objects not directly designed for replication, all requirements on the object replicas themselves are reducing the benefit of this approach. The architecture tries to keep the set of special requirements to a minimum however, and no modifications of *existing* object functionality should be necessary. For example, the methods *lock* and *unlock* described below can normally be added to the objects without changing existing code[5].

Serialising object access
Object replicas must be extended with two additional methods to support serialisation among multiple surrogates. Assuming that the potential for name conflicts is eliminated, these methods can be added by automated code generation tools because the semantics of the methods are simple and generic across all replicated classes.

[4]Practically a function named open which accepts a filename as an argument and returns a result of type File.

[5]Assuming methods with the same names do not exist already. Note that the name of the methods need not be *lock* and *unlock* in an implementation of the architecture. The requirement is that two methods implementing this functionality can be uniquely added to every replicated class. Other names, even less likely to cause naming conflicts could therefore be chosen. They are given short names here to simplify presentation.

A method to lock the object is required to 'mark' the object as being currently used for a surrogate request. The method is specified as:

lock(s:SrgtId; #locks:integer; activeSet:ReplicaSet):LockReply

where *LockReply* is a record containing result parameters from the lock request. If the object is currently unlocked then the object is marked as locked by surrogate *s* and *lockGranted* is returned. Otherwise, if the object is currently in use by another surrogate (and therefore locked), *lockDenied* is returned together with the name of the locking surrogate. However, due to the potential for competition for the locks, a mechanism to arbitrate among competing surrogates is necessary. The parameter *#locks* is used for this purpose, and the surrogate which locked the replica with the highest value for *#locks* wins and can continue the attempt at gathering the necessary locks. Now, if the surrogate loses, the call still returns *lockDenied*, but also returns *giveUp* as part of the *LockReply* parameter. The surrogate then knows that it should give up all its currently gathered locks, pause, and start over. A competing surrogate, which does not receive a *giveUp* result, will thus be able to make progress.

The parameter *activeSet* is used to propagate failure status among the replicas, and is also used in conjunction with the reconfiguration protocol, and is described in more detail in §6.3.2 p. 70. Similarly, the unlock function takes the form:

Unlock(s:SrgtId; activeSet:ReplicaSet):LockReply

It resets the object to an 'unlocked' state and returns *lockReleased* if the object is currently locked by surrogate *s* or currently unlocked. The method returns *lockNotReleased* if the object is currently locked by another surrogate. The parameter *activeSet* is used similarly to the *lock* method.

Before a surrogate can invoke the object replicas a majority of them must be locked using the method above. This ensures that multiple surrogates cannot jeopardise serialisation of invocations. When the invocation returns, the replicas must be unlocked by the same surrogate. Problems with unreleased locks due to crashing of surrogates can arise during sharing of replicas by multiple surrogates and are discussed in §6.3.3 p. 72.

The serialisation and consistency protocol, executed by the surrogate, is divided into three sequential rounds of synchronous invocations. Rounds 1 and 3 guarantee serialisation by mutual exclusion, round 2 ensures that all non-failed replicas are mutually consistent. During each round, observed failures are recorded in the surrogate's active set. To inform other surrogates, the *activeSet* parameter is used to record detected failures in other replicas during rounds 1 and 3 (see also §6.3.1 p. 69).

1. Gather locks from a majority of the replicas to ensure serialisation of replica requests.
2. Invoke the client's requested method on all replicas.
3. Release locks granted in round 1.

A benefit of these generated methods is that they are guaranteed to be idempotent, in fact they have no effect on the object's original internal state at all. If there is contention, achieving a majority of locks can require several rounds of competition among surrogates. During the process of gathering a majority of locks on the replicas, each surrogate might therefore retry these methods as many times as is found necessary without any danger of violating the integrity of the object replicas. Additionally, if the underlying communication infrastructure is believed to be unreliable and the surrogate does not receive the required replies, it can initiate extra retries of the lock method to check if the replica is still alive. This will increase the probability of giving a correct diagnosis of real replica crashes.

Encapsulated object replicas

Object replicas must have a completely encapsulated state, i.e. no part of their state must be accessible

other than through the use of methods defined on the object. This restriction is needed to avoid replicating state in the surrogate and the object replicas themselves, but does also conform with the object model advocated in this dissertation (see §2.4 p. 21). If an object replica was to have non-encapsulated state, the surrogates would have to contain this state to maintain the semantics of the object[6]. This, in turn, would lead to problems when multiple surrogates exist throughout the network. Essentially, all modifications to replica object state contained in the surrogate would have to be propagated to all other surrogates to reflect the changes, incurring a need for another consistency protocol among the surrogates. The data stored in the surrogates would create another virtual (covert) communication channel between clients of surrogates, and this could result in violations of the causality relation [42].

Additionally, allowing parts of an object's state to reside in its surrogate would also complicate the underlying remote invocation mechanism, as any method on the object which manipulates state residing in the surrogate would have to access this state via another set of (potentially expensive) network messages.

6.3 System Functionality

The components described above interact to implement the replication scheme. Briefly, the surrogate is the interface used by clients. It receives an invocation (parameterised with a collator object in case of function-type methods), passes it on to the parallel invocation module which invokes the methods on the object replicas, and the collator processes eventual results from the invocations (see figure 6.1 p. 62).

More details on the programmer interaction aspects are given in chapter 7. The rest of this section elaborates on the internal functionality of the mechanisms.

6.3.1 Masking failures

Resilience against failures is achieved if the system can be reconfigured to operate despite failures or if the system can be brought back to a state before the failure occurred [104]. This architecture implements replication which is a technique to mask failures by redundancy; failure recovery requires transactional support such as logging, state restoration and grouping of actions [85] which is not readily available in this system model. To mask failures, a surrogate object maintains as part of its internal state a data structure which contains information about the collection of object replicas being managed. This structure is called the *active set*, where each replica is tagged with a failure status. If the surrogate detects and diagnoses a replica failure, the replica is tagged as such in the active set, and the surrogate does not pass any more invocations on to this replica. Note, however, that all initial members remain in the set during the lifetime of the surrogate, only the status flag changes. During reconfiguration the old entry is reused, reconfigured replicas are only installed in PEs specified by the programmer at initialisation (see §7.4 p. 81). Consequently, all failures are reduced to crash failures by passivisation [104]. This achieves mutual consistency among all correct replicas, but it also means that transient failures can lead to exclusion.

A tradeoff between accuracy and performance must be made here. To improve performance replicas can be tagged as failed quite rapidly (and somewhat pessimistically). Slower, but more accurate error detection is obtained if the lock-method is called multiple times[7]. The architecture allows implementations to optimise the number of retries to fit the network's failure characteristics and underlying communication protocols[8].

[6] The programmer would expect to be able to access this state directly without the use of indirect methods.

[7] Note that only the lock-method on the replica can be retried, the actual method requested from the client cannot generally be retried as it might be non-idempotent.

[8] Note that many communication protocols are already perfectly able to mask many transient failures. Retries of the lock method can only increase the accuracy of this functionality.

Failure resilience
The architecture uses a majority voting scheme to ensure mutual exclusion among multiple surrogates. Hence, in a collection of n replicas, $\lfloor \frac{n}{2} \rfloor - 1$ replica crashes can be tolerated. For example, to mask two crashes, three correct replicas are required. This might seem restrictive, but the majority avoids conflicts during partition failures, and the availability of majority voting schemes is good even for relatively small numbers of replicas (cf. §B.2 p. 116). The use of special collators, e.g. weighting or voting collators can amend the architecture to also tolerate some malicious (data-domain) failures (see §6.2.3 p. 65). Adding to the failure resilience of this architecture is the requirement that clients do not share surrogates between address spaces. This means that the failure of other client address spaces does not affect the availability of a surrogate.

Detecting failures
The architecture depends on timeouts and 'alive' messages to detect failures[9]. Alive messages are necessary due to the large variations in running time for different method executions; using timeouts would be very inefficient. Because timeouts would have to be set large enough to allow for even the most lengthy computation, periodic alive messages are used instead to check if the address space hosting the replica is still responding when the replica is executing a lengthy invocation, thus achieving more efficient and accurate failure reporting. Alive messages are retried a small number of times to reduce the impact of transient failures; the exact number might be determined by a particular implementation of the architecture, but might also be set by lower-level software, such as the remote invocation facility (see §8.2.1 p. 88).

A replica failure is detected by the surrogate, and only as a consequence of erroneous behaviour during a surrogate's manipulation of the replica. If a replica is transiently incorrect between two such requests, the failure cannot be observed. However, the architecture assumes that such transient failures do not affect the state of the replica itself, transiently disconnected replicas are rather regarded as a period of time in which the replica is not responding to requests. All transient failures can therefore be treated as transient network failures, while the replicas themselves behave according to the crash-failure semantics described in §3.2.3 p. 32. It is further assumed that transient network failures will be masked by the communication facilities.

The serialisation protocol, executed by the surrogate, is divided into three sequential rounds of synchronous invocations. During each round, observed failures are recorded in the surrogate's active set. Consequently, a replica failure can be observed only during one of these rounds. The first round can naturally fail to observe some replica failures; if the first $\lfloor \frac{n}{2} \rfloor + 1$ replicas responds positively to the lock request, failures in the remaining replicas will be missed, although serialisation is still ensured. The probability of not observing failures in the first round is therefore high. However, failures which are detected during this round can be more accurately established due to the idempotent behaviour of the lock primitive on the replicas. Round 2 will observe all non-transient failures in the replicas. The last round may experience additional failures happening after round 2. Clearly, there is no need to attempt releasing locks on failed replicas.

Decreasing number of replicas
The fundamental difficulty with this approach is that the number of non-failed replicas are monotonically decreasing. Even if a surrogate observes that a replica which has been tagged as failed recovers, it cannot easily be re-integrated into the replica set because the surrogate does not record old invocation requests, and therefore cannot bring the replica up to date through its method interface. An approach to add new replicas to the replica set is necessary to solve this problem. This is discussed further in §6.3.2. If the system support mechanisms are used for long-running applications, or systems experiencing frequent failures, this can lead to rapid complete surrogate failures because no replicas remain failure-free indefinitely.

For other applications, which only have to run reliably for short periods of time, or applications which

[9]Similar to the probes described in [23].

can be quickly restarted, this problem will be less important. The architecture will significantly reduce the probability of restarts being necessary due to failures.

Surrogate failures
A surrogate may fail while performing the parallel invocations on replicas, i.e. round 2 of the consistency protocol (p. 69). Only invoking a subset of the replicas may cause inconsistencies, and because messages are not logged, there is no way of 'replaying' the missing invocations. Thus, the architecture cannot implement the atomicity property, i.e. the "all or nothing" property. Atomicity requires a copy of the previous replica state which is not available in this system model. The probability of failure during execution of the parallel invocation module is unpredictable because it is impossible to know in advance how long each execution will take. If the duration of round 2 is short, the probability of a surrogate crashing while executing them is low. However, the architecture makes sure that subsequent surrogates observe the potential for inconsistency and let the application programmer decide whether to abandon the object or continue using it (cf. §6.3.3 p. 72).

6.3.2 Maintaining consistency

A system support mechanism should require very limited knowledge about the internal structure of the objects it replicates such as to provide a generic service for a range of objects. Full consistency programming models are beneficial because the programmer never observes an out of date object, and can always regard an object as being non-shared. This also makes replication more transparent. Consistency is maintained among the replicas by ensuring that all non-faulty replicas receive identical sequences of invocations (round 2 of the protocol described in §6.2.1 p. 66). A communication protocol which prevents reordering of messages (such as TCP/IP, used in Network Objects [22]) is assumed by the architecture, and ensures that object replicas receive invocations in the same order as they are issued by the surrogate. Additionally, the majority locking scheme ensures that surrogates serialise their access to the replicas. This guarantees that both the consistency and isolation properties known from transaction processing are preserved[10].

To increase fault tolerance, a surrogate is always located in the same address space as the client object referencing the surrogate. This reduces the length of the critical access path to the replicas, thereby further increasing the reliability of the object. While the surrogate is still a single point of failure, the system model assumes that a failure within an address space causes all objects in it to fail (i.e. including the client).

However, this requirement complicates the architecture somewhat during the sharing of object replicas among surrogates in different address spaces. The serialisation protocol, through the *lock* and *unlock* methods, requires that failures are recorded through the *activeSet* parameter. This parameter does increase the overheads in the concurrency protocol, but not excessively so, as its size is of order $O(N)$.

Reconfiguration of replicas
During operation, the active sets are monotonically decreasing. For long-running applications, a way of reconfiguring replicas is necessary. The object encapsulation principle and lack of recoverable objects makes it difficult to reconcile failed, and potentially stale, replicas into the active set. Reconciliation typically requires very application specific knowledge, not available to the system support mechanisms. Without access to, or knowledge about, the local state of the objects, an inconsistent object cannot be brought back to a consistent state by the system support layer alone. An approach based on *regeneration* of failed replicas is presented in [148]. Each replica object must implement a *CopyMe* method, which is used to make new copies of an object in case a replica fails. Cooper suggests a similar approach, relying on automatic marshalling and unmarshalling of replicas to implement special *get_state* procedures [48]. Implementing such methods increases the burden upon the application programmer and reduces the

[10]The C and I in ACID [85].

benefits of object replication, but must be added to make the replication scheme useful for long-running applications. In contrast, the Delta-4 architecture assumes simpler conventions for copying replicas; when reconfiguration is necessary, single process contexts are copied transparently [145]. However, that approach is not appropriate in this system model. Simply making a copy of an address space and reinstalling it in another PE would most certainly violate a number of bindings with underlying system components such as open files and thread-contexts. Breaking and re-initialising such bindings would have to be implemented manually as they are application specific, thereby reducing the approach to the one proposed in [148].

Some distributed object systems may provide direct support for copying objects, and in this case the surrogate can simply initiate a copy of a non-failed replica to another address space. This can be achieved by creating a copy of another (failure-free) replica and installing it into the replica set. However, care must be taken on several accounts.

1. To avoid race-conditions, a new replica must be re-installed within a single indivisible action. The surrogate must not perform any updates on other replicas during the process of installing a new replica as this could lead to inconsistency. Rather, the surrogate should temporarily halt its processing of client requests, install the new replica, update the replica set, and only *then* resume accepting method invocation requests from clients.

   In the case of multiple surrogates for the same replica group, this implies that also the other surrogates must be blocked during the period of reconfiguration. By first acquiring a lock on the majority of the replicas, a surrogate prevents interference from the other surrogates. These locks are set with the current number of active replicas flagged as non-failed using the parameter *activeSet* in the *lock* method. However, after the new replica is added, it is unlocked with the updated version of the active set, indicating a new member in the replica group. The surrogate's active set is modified accordingly. And, because both the newly installed replica, and a majority of the older replicas are unlocked with this parameter updated, the next surrogate to attempt locking the replicas will notice that the active set has increased. The surrogate noticing that the active set has increased, queries the PEs currently marked as failed in its active set for a reference to the new replica objects.

2. Defining the scope of the object-graph to copy into the new replica is a hard problem, and compromises must be made when the decision is taken [52]. A shallow-copy approach, i.e. copying only a single object and maintaining the existing references, will result in poor locality and reduced fault-tolerance due to the number of remote references. It also introduces the problem of increased object sharing. In contrast, a deep-copy approach may be very expensive and has confusing semantics as it duplicates objects and thus ruins the notion of object sharing. An intermediate solution is to require that the programmer defines the rules for making a copy of the object graph, although this introduces extra complications in the programming model.

3. Changes to persistent data. Related to the above problem is the problem which arises if objects within the copied graph have references to persistent objects such as files (as most realistic applications will). An application specific procedure to make a copy of the object graph could deal with this problem. The cost of this approach is introduction of extra complications in the programming model and a reduction of the transparency of the system support functionality.

4. During the creation of a new replica, it does not make much sense to directly try to install it in a failed PE. With high probability, the PE is still failed, and the operation will therefore not succeed. However, the PE must be in the set of originally specified PEs for the replicas by the programmer to maintain the semantics of the initialisation of the surrogate (see §7.4 p. 81). Therefore, only when it has been determined that one of the faulty PEs has recovered can a new replica be installed in it. This is not a problem however if the reason for the replica failure was a single address space failure within the PE. A new replica can be installed in another non-failed address space on the PE.

The problem of reconfiguration is not addressed in any further depth in this dissertation. It is clearly an issue requiring further investigation and should be investigated more carefully in the light of real applica-

tions (cf. §10.3.3 p. 109). For applications which can be restarted occasionally, dynamic reconfiguration will, however, be less important.

6.3.3 Supporting object sharing

Objects in a distributed system are commonly shared by passing references as arguments in method invocations or registering objects with name services. Object sharing introduces problems with serialisation of invocations [53, 99]. Two distinct sharing scenarios are possible, and they are handled differently for reasons of fault-tolerance:

1. Sharing replicas among multiple clients in the same address space.
2. Sharing replicas among multiple clients in different address spaces.

To support isolation among multiple concurrent clients in the same address space (type 1), the surrogates hold a lock as part of their internal state. The lock is acquired before a client request is executed within the surrogate, and the lock controls the queue of blocked outstanding client requests.

Special care must be taken when two clients residing in separate address spaces need to share a collection of object replicas (type 2). Consider an object A holding a reference to a surrogate B_s managing a collection of replicas B_r. Now, a remote object C wants to use B_s, and asks A for a reference to B_s. The naïve approach of simply passing to C the reference of B_s would result in C's use of the group B_r being dependent on any faults in A's address space. The chain of references from C to B_r should instead be kept as short as possible so as to maximise its availability. Most importantly, the failure masking code which resides in the surrogate, should be located in the same failure domain as the client. The surrogate will therefore always be available. If this was not ensured, the failure of the surrogate address space would render the client's reference invalid and thus weaken its resilience.

Therefore, a new surrogate B'_s is created in C's address space before C starts using B_r. After creation, B'_s contains an identical collection of replica references B_r. This set of references need no special treatment however, they are ordinary remote references which refer directly to the replicas. The set of replica object references is determined during the instantiation of the surrogate, and will never extend to other replicas than those specified by the programmer (see §7.4 p. 81). To ensure that the surrogate has enough information available to rebind to a reconfigured replica, the active set contains an identifier for the PE in which the replica previously existed. During reconfiguration the surrogate is thus able to relocate the new replica on that particular PE (cf. §6.3.2 p. 70). Note that this approach would be meaningless unless the underlying remote object referencing policy worked similarly.

Another problem appears during synchronisation. Clearly, the lock stored within the surrogate is unable to synchronise access to the replicas when several surrogates exist. Rather, a shared resource must be used, and the replicas themselves are used in this architecture by requiring that a majority of them respond positively to a lock method invocation[11]. The majority ensures that only one surrogate at a time is able to execute an invocation. A surrogate that fails to acquire a majority of locks must wait until the currently executing surrogate is finished (cf. §6.3.1 p. 69).

The problem of unreleased locks (see §6.2.4 p. 66) becomes apparent when multiple surrogates share object replicas (type 2). If a surrogate fails after having been granted locks, these locks must be released. There are at least two possible approaches which can be used to solve this problem (without resolving to logs[12]):

[11] The lock within the surrogate is not strictly necessary, but improves performance in the case of multiple clients sharing a single surrogate because it eliminates the need to execute the two-round majority locking protocol and simplifies the construction of surrogates (cf. §6.3.4 p. 74).

[12] With access to logs the approach of extermination can be used. By recording the proceeding RPC on disk the client can release the lock itself after reboot [183] (chapter 10). The expense of logging each RPC to disk might be high however, and the approach assumes that the client will eventually reboot.

Expiring locks. By using dedicated timeout mechanisms within the lock methods, a lock could be designed to automatically expire after a certain time interval. While the use of timeouts is a probabilistic approach, and may cause havoc if they are released too early, the timeout interval could be set long enough to make conflicts very unlikely. However, this approach would be very inefficient as the duration of a method invocation will have large variations and thus require very long timeouts. A slightly more sophisticated approach is suggested in [183]. Instead of using a single large timeout value, the callee can be responsible for periodically renewing a 'contract' with the client. However, this complicates the construction of replicas by enforcing particular conventions for defining methods and would reduce the transparency of replication.

Explicit surrogate unlock. Due to the limitation of the above mentioned approach, the architecture uses the following slightly more complicated technique. A surrogate which fails to lock a majority repeatedly simply suspects another surrogate of having left unreleased locks. The suspecting surrogate can then check if the suspected surrogate is still alive. If it is, the suspecting surrogate must wait, and retry the process of gathering locks later. However, if the suspected surrogate is believed to be dead (using an appropriate failure detection algorithm), it can be assumed to have left the locks unreleased due to a crash. The locks are then explicitly released by calling *unlock* with the id of the crashed surrogate, and then set again by the suspecting surrogate[13]. Additionally, by using this approach, a surrogate can suspect that the previous surrogate using the replicas died during invocation, and report a 'potential for inconsistency' exception back to the client as a warning (see §7.7 p. 86).

Both approaches have their limitations however, and they are both probabilistic. The asynchronous system model does not allow completely reliable failure detection, so the problem of unreleased locks cannot be managed with absolute certainty. There is a small chance that locks are released prematurely which will endanger the consistency of the replicas.

Worth noting is that unreleased locks do not pose a problem unless there are several surrogates sharing the replica objects. If the last surrogate fails, its replicas not be required any longer because no surrogate references them any longer, and they will be reclaimed automatically by the garbage collector. Another problem arises if the orphan has acquired locks or has initiated unrecoverable actions, simply killing it is not preserving correctness in the system. However, the problem of reclaiming distributed garbage objects is not discussed in any further depth in this dissertation (e.g. see [143]).

Distributing failure status

Maintaining multiple surrogates also introduces a new consistency problem. The replicated surrogates should have a consistent view of the replica group status in terms of failures. In systems which provide atomic message delivery, ensuring consistent group views can be costly for large numbers of surrogates [18, 20]. Atomic message delivery was sacrificed in this architecture as it would seriously reduce the scalability of the system, and as a high number of surrogates is expected for the kind of applications this architecture is aimed for, it would work against the goal of the architecture. Atomic message delivery is provided by group communication protocols by closely synchronising all participating processes with an orthogonal protocol to propagate group view changes, i.e. failures and reconfigurations. This synchronisation thus requires that all processes are able to communicate with all other processes (an $O(N^2)$ overhead). In this architecture, it would imply that all surrogates maintained references to all other surrogates, and synchronised themselves by gathering locks from each other.

To avoid such overheads, this architecture requires surrogates to detect replica failures rather than relying on propagation of active sets among the surrogates. The replicas' fail-stop failure semantics makes this possible. Although a surrogate could achieve faster determination of failures through specific intra-surrogate messages, the fact that replicas simply crash means that all surrogates will eventually detect replica failures. However, the lack of a group membership protocol among the surrogates means that more cooperation is required from the replicas during reconfigurations, where replicas are reintroduced into the system. The reconfiguration protocol described in §6.3.2 p. 70 facilitates this.

[13]This is possible because a failing *lock* call returns the id of the currently locking surrogate.

6.3.4 Maintaining performance

The achieved performance in distributed systems is primarily dependent on the level of asynchrony allowed among interacting objects. Close synchrony is usually wasting CPU-cycles in both the invoker and the invokee. Asynchronous collators, i.e. collators returning on the first reply improve the throughput for casual surrogate invocations. The architecture can therefore give good performance if surrogates are lightly loaded. Under high loads however, the pessimistic concurrency scheme cannot allow another surrogate request before all the previous replies are gathered.

Further, the architecture is not dependent on significant amounts of disk I/O, the performance of this scheme is primarily dependent on communication latencies. While storing objects on disk is necessary to support certain kinds of recovery strategies, the overheads can be substantial. Further, as the gaps in speed between the levels of the memory hierarchy in computer systems are likely to increase [12], the dependence on extensive disk I/O might become too expensive for some applications.

The ratio of communication latency to object method execution time determines the efficiency of a function shipping replication approach as used in this architecture. Maintaining performance also requires attention to scalability issues. Although there are many factors which affect the scalability of this architecture, the main factor is communication latency. The replication protocol presented requires only a single RPC to each replica in case there is one surrogate in use. When several surrogates are used, and thus must be serialised, three rounds of RPCs are necessary, with the first and third round requiring at least $\lfloor \frac{n}{2} \rfloor + 1$ parallel RPC calls to lock and unlock a majority of replicas (cf. §6.2.4 p. 66) and the second round requiring n parallel RPCs. More RPCs might be necessary in case of competing surrogates.

Communication latency

Communication latencies within high-speed networks are already very low, in many cases lower than the average access time for disk storage systems. Although communication latencies are inherently limited by physical propagation delays, other factors such as processing overheads and media competition are currently more significant. Research aimed at reducing latencies of popular protocols have shown promising results with round-trip delays around 200μs (157μs for very small TCP messages) in ATM-based LANs [192]. It appears unlikely that similar latencies are achievable in disk storage systems in the foreseeable future, currently providing mean access times around 10ms (a factor of 50 higher).

Competition for the communication media incurs non-predictable delays, particularly in long-haul computer networks which must do a lot of buffering due to bursty traffic patterns [122]. This is easily observable in the Internet, for example, where latencies may vary greatly during the day. If the architecture was going to be used in a wide-area network, such as the Internet, at least with its current characteristics, would probably be an unsuitable networking infrastructure. Dedicated, perhaps leased PSTN-based, WAN links should be used instead, to guarantee low latencies for priority communication. However, the physical propagation delays become more prominent as well. For an optical communication channel of ca. 900 kilometers length, its round-trip propagation delay equals the latency of a disk access[14]. Naturally, potentially multiple stacks of communication protocols will increase this latency, but similar contention-dependent processing overheads are also present in disk systems. For both approaches, caching is a technology that can reduce the number of such accesses and thus give substantial performance gains. Modern communication technology also provides high bandwidth, at least comparable to that of disk systems [140, 192].

Replication of objects on other PEs on the network may therefore be a good alternative to the storage of objects on disk for the purpose of survivability, as accessing objects over the network will be faster than accessing them from disk[15]. Essentially, the architecture implements the durability property by replication on several independently failing PEs[16]. Naturally, this alternative requires more memory

[14] Assuming a signal propagation speed of $1.8 * 10^8$ m/s in the fibre (cf. §2.3.1 p. 19).

[15] This is a motivating factor behind current state of the art research within distributed file systems as well [6].

[16] The D in ACID [85].

capacity in the PEs, increasing hardware costs of the system. If additional physical memory is not installed, greater proportions of the PEs' address space will be stored on disk anyway by virtual memory mechanisms, thus reducing the advantages of this approach.

6.4 Physical Mapping Issues

Underlying an architecture is a collection of assumptions made about the physical mapping, i.e. allocating the architecture onto a real distributed system and should be made explicit to reduce integration and reuse efforts [54]. This section elaborates on the physical mapping issues assumed by this particular architecture.

The architecture is meant to be used within the system model defined in chapter 2 and is intended to be used as a system support facility to assist with replication of programming language objects. The key component in the architecture is the surrogate which slots in between the client and a collection of object replicas while taking on the rôle of encapsulating replication. Mapping the architecture onto a real distributed system therefore involves the locating of clients, surrogates and object replicas onto the PEs in the network.

As mentioned in the introduction to this chapter, a surrogate is a relatively light-weight construction. If used by clients residing in the same address space the performance overheads due to sharing are limited by the implementation of locks in the particular programming language. During sharing of replicas among clients in distinct address spaces synchronisation of multiple surrogates is required which uses locking of replicas, and this incurs extra communication overheads.

Of critical importance to the mapping issue is the network failure characteristics. Some networks are often partitioned, for example large internetworks like the Internet [81]. However, as this architecture is primarily aimed to assist the development of distributed applications within smaller scale networks, it is assumed that partition failures are relatively rare, and do not persist for very long periods of time. If the particular network used does experience frequent partition failures, fault-tolerant network designs should be considered as these can reduce this problem significantly [19, 196].

6.4.1 Clients and surrogates

In a general distributed object system, any object can invoke methods on another object if it has the reference to it. Thus, any object may potentially become a client of another referenced object. It is therefore difficult, if not impossible, to predict the localisation and number of clients which obtain references to surrogates. Some applications might be contained within just a single address space, i.e. running as a single process, and hence only require the use of a single surrogate. Other applications, for example groupware systems which support cooperation among several users on different PEs, might be partitioned into large numbers of processes dispersed throughout the network, each process making use of the same replica group and thus requiring distinct surrogates.

This dissertation makes no attempt at classifying a client's usage pattern of a particular surrogate. Program behaviour is difficult to predict, although some programs exhibit very characteristic behavioural patterns [195]. Depending on the semantics of the application, a client may use a surrogate sequentially in a tight loop, or may use it sporadically or not at all. However, with increasing numbers of clients, it likely that eventual burstiness of activity is smoothed out.

Further, the time necessary to execute a method call might vary significantly, again depending on application semantics. Naturally this will increase the amount of generated load on the PEs hosting replicas, however, it is assumed that the PEs are normally underutilised and therefore are not significantly slowed down by this (see §6.4.2).

The architecture, due to the strong consistency assumption, performs strict serialisation of client requests to the surrogate. Therefore, clients of a particular surrogate cannot make valid predictions about the

service time of the surrogate. If the number of concurrent clients is high, relatively long service times must be expected.

Surrogates are always located in the same address space as the client to increase resilience against failures (cf. §6.3.2 p. 70). The creation of new surrogates is automated so clients need not be concerned with this issue. However, surrogates do naturally introduce extra computations necessary in the PE.

6.4.2 Replicas

The architecture assumes replicas are hosted in separate failure domains, i.e. address spaces, and most likely on separate PEs. Of particular importance for the physical mapping is the placement of replicas and their number. Although the programmer is responsible for the actual mapping in this architecture, the particular choice must be made carefully to achieve good benefits from replication. As indicated in section B.2 p. 116, even configurations with less than 10 replicas will give substantial availability improvements, and configurations with 3, 5, or 7 replicas might be sufficient for many applications. Additionally, the generated load on the PEs hosting replicas will increase as well, and some care should be taken to avoid overloading these.

Replica placement
Resilience against failures can only be achieved if failures are independent and partial. Thus, it is essential that any sharing is minimised among the replicas. This implies that common resources such as file systems, databases, system services and physical components should be replicated as well. However, replication at all levels in the system hierarchy might not be practicable and some tradeoffs must be made. For example, many applications are written to make use of whatever file system is available, and in some scenarios this will be a shared, distributed file system. If this file system is NFS for example, which is a non-replicated file system, the benefits of replication will be lost if the file system server crashes.

Additionally, sharing of resources between replicas introduce problems of nested invocations whereby each replica will attempt to perform the same sequence of operations on the same resource. If the operations on this resource are non-idempotent the application will not behave correctly. Some operations on file systems, for example, are non-idempotent, such as the creation of directories and files. An approach to avoid this problem is to make use of file services on local disks only, eventually implementing the replicas such that they can tolerate this behaviour.

Independence of failures requires that failures are hindered from propagating [104]. In a replication scheme, it is thus essential that the replicas are placed in independently failing address spaces. The probability that all of a PE's address spaces fail simultaneously cannot be ignored, and this will normally justify that replicas are located on separate PEs within the network. Additionally, many network failures affect multiple PEs, for example broadcast storms, babbling nodes and routing conflicts [196]. Naturally, such network failures will have a dramatic effect on the availability of the replicas, and consequently on the availability of the surrogate. Full consistency schemes suffer from very low availability in such circumstances, and even weak consistency schemes would give unavailability unless a replica was stored locally on the PE.

However, assuming that such network failures are rare, a good placement of replicas will increase the probability that enough replicas are available for the surrogate to achieve its majority of replica locks. Depending on the network's topology and failure characteristics, the optimal placement of replicas will vary and might require expensive computations[17]. Schemes have been presented to automate the process of replica placement [114, 124]. However, automated replica placement requires access to sophisticated support functionality such as replica relocations, failure statistics calculations, object interdependencies assessment and dynamic compensations for changes in network topology. This architecture assumes that the programmer is reasonably knowledgeable about the reliability of the PEs within the network and

[17] Although near-optimal placement can be performed much more cheaply for some network topologies (Ethernet and fully connected networks) [175].

therefore is able to make the decision on replica placement.

Generated load
A PE hosting a replica is responsible for executing the method call in the replica object graph. The time required to execute a method call cannot easily be predicted, rather, it is assumed to exhibit large variations. The proposed architecture relies on programmer directions to locate replicas and thus decide which PEs are able to support the extra load of replica method executions. Although this is clearly a task which should be automated in an extension of the architecture, the programmer could relatively easily develop application functionality which queried a collection of PEs for their load and thereafter selected those with the best prospects of giving the fastest execution times. These PEs could then be used as input to the surrogate creation procedure (cf. §10.3.3 p. 109).

6.5 Limitations and Future Work

The system support mechanisms lack support for coordination of invocations from multiple replicas to shared objects. A given implementation of the replica object might therefore trigger multiple, non-idempotent methods in referenced objects. The semantics of the application must be considered, and the programmer is responsible for the correct implementation. For example by implementing particular filters in shared objects that filter out redundant invocations.

Further, the serialisation protocol has not been formally verified. Although it has undergone informal reasoning, a formal approach should be taken to provide the necessary guarantees for correctness if the architecture was used in critical settings. These limitations would be interesting directions for future work, and are also discussed in §10.3.

6.6 Concluding remarks

This chapter has presented the architecture of the proposed system support mechanisms in detail while focusing on the internal structures and functionality. The strong decomposition of generic and application-specific code has lead to a design with good cohesion and extensibility.

Understanding how the architecture works is important in its own right, but the main benefits of the architecture will only become clear after observing its effect on application program complexity. In the next chapter a programmer's model of the architecture is presented which explains how the architecture is used by an application developer.

Chapter 7

Programming Model

This chapter describes the external interfaces of the system support architecture introduced in chapter 6, and explains how an application can benefit from the functionality to use failure-resilient objects. The syntax used in this chapter is Modula-3's to simplify presentation and to give the model a concrete appearance. Small changes in the notation are therefore likely if the architecture is implemented in other languages.

7.1 Overview

An unfortunate attribute of distributed applications is their inherent vulnerability to failures in other system components, and replication can often be used to reduce this problem. The main goal of the architecture proposed in this dissertation is to provide a simple, yet flexible programming model such that developers of distributed applications are given access to relatively transparent object replication. This can help the developer to focus more attention on application specific functionality rather than availability requirements which are orthogonal to the application. Object replication is a beneficial approach to replication because it aims to use replication as a generic service; using this approach, objects need not be specifically designed for replication. This is particularly beneficial from a system support point of view as it reduces the involvement required by the developer of the object.

Another goal for object replication techniques is to conceal replication for clients to minimise changes necessary in objects using a replicated object. In the proposed architecture, surrogates are used to hide details of replication. The surrogates, which are very similar to ordinary objects, take on the rôle of concealing replication functionality. Much like a Gaggle [28], a surrogate encapsulates a group of replica objects. A surrogate provides a new but very similar interface to a collection of object replicas. Figure 7.1 p. 79 shows how this is realised. In the figure, the surrogate and replica objects are composed of an interface part and an implementation part. The surrogate Srgt is referenced by an object Client, and the surrogate provides a functionally equivalent interface to the client as the replicas.

Additionally, to increase flexibility, the architecture supports the use of special collator objects, which allow the programmer to define customised processing of method return values from replicas. The use of collators is discussed further in §7.5.2 p. 83.

Declarative object replication
The programming model is based on programmer declared replication. The programmer is responsible for specifying individual classes of objects that will be replicated and code generation technology produces replication code for such classes. Individual classes, whose instances are to be replicated, are defined as normal, but they are tagged with a keyword so that a code generator is able to recognise which classes should have extra surrogate code produced. This approach is chosen based on the observation that the programmer is the only authority with enough knowledge about the application requirements to

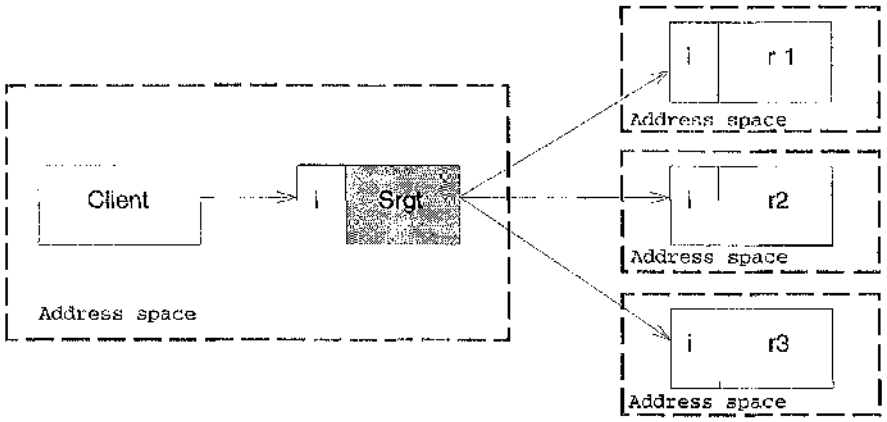

Figure 7.1: A surrogate Srgt managing a set of replicas r1, r2, r3.

determine if the extra overheads for replicated classes can be justified. The only other realistic alternative would be to replicate all classes. However, the added costs of using replication should be strictly controlled because most realistic applications must be developed using available resources efficiently, and replicating all classes would most likely not be necessary (see also §7.3 p. 80).

Design issues

The architecture is designed as a collection of software modules and automatically generated stub code which provides assistance to the management of object replication. The prototype built to experiment with the architecture (described in chapter 8) implements the mechanisms as library code included in the application's code space. This seemed reasonable in the prototype due to a sufficiently compact implementation. However, other and more complete implementations of the architecture might justify the use of e.g. shared libraries to house the mechanisms. Additionally, if implemented on a range of different platforms, the architecture could be included as a service within distributed system middleware; distributed application development frameworks supporting interoperability [14]. Further, some operating systems, for example SPIN [15], could allow for the architecture to be implemented as extensions to the operating system (cf. §5.2 p. 53). However, the programming model, as presented here, should not change significantly as a result of the localisation of the code, although some of the procedures used to *build* the applications might change slightly (e.g. makefiles etc.).

The process of developing software with this technology is extended with an extra preprocessing stage, and after the surrogate code is generated, a compiler is used to produce the executable(s) while ensuring that type safety is maintained.

7.2 Application Partitioning Assumptions

A limitation of the architecture is that replicas will cause multiple invocations on shared objects. Let us consider the problem in more detail. As dealing with replication adds to the complexity of the application semantics, the programmer wants to design objects as if they were not replicated. The objects are thus implemented as if they were singletons. However, one feature of objects is that they may invoke methods on other referenced objects. Hence, when an object is replicated, it may trigger multiple invocations on referenced objects. Naïvely, the problem can be solved with some programmer interference. Invocations to shared objects can be distinguished and sent through a filtering mechanism which makes sure that only one of the invocations is passed on to the destination object. Similarly, the filtering mechanism must make sure that results from the invocation are passed back to the calling replicas [121] (see also §10.3.1 p. 109). Naturally, this requires that the programmer is made aware of replication, and will result in

increased application complexity. Without such a mechanism the programmer is currently restricted in the way the application can be partitioned.

The limitation does not have any effects if the application is partitioned into separate object graphs as discussed in §6.2.1 p. 63. As objects within these object graphs do not reference shared objects, the problem of multiple invocations is avoided. Although distributed applications in general are likely to be partitioned in this way for efficiency reasons, a small amount of interaction with other, shared objects may be necessary.

Enforcing this partitioning restriction on distributed software may not always be feasible. For example, in a scenario where objects can dynamically take on the rôle as agents, and thereby invoke methods on other arbitrary objects, methods on shared objects would be invoked multiple times. Further, for software structured as layers of libraries, it might be difficult to ensure that objects are not shared among the object graphs.

7.3 Defining Replicable Classes

The architecture assumes that a programmer defines classes by writing separate class interfaces and class implementations. Support for replication of particular classes is achieved by tagging the class with the keyword `<* REPLICATED *>`, which is recognised by a stub generator.

For example, in an application controlling the temperature of some process using multiple thermometers, the user might define the interface for this function as the replicated class `Thermometer.T` like this:

```
INTERFACE Thermometer_Replicated;
TYPE
  T <* REPLICATED *> <: Public;
  Public = OBJECT
    temperature : REAL;
  METHODS
    readTemperature() : REAL;
    calibrateTemperature(t : REAL) : BOOLEAN;
  END;
END Thermometer_Replicated.
```

The generated stub file will contain the new interface code for the type `Thermometer.T`, which is the type to be used by the calling application and which is linked into the program. Similar naming is required for the implementation of the class, the tag is recognised by the code generator to add the lock and unlock methods to the class (see §6.2.4 p. 66):

```
MODULE Thermometer_Replicated;
IMPORT Sensor;
REVEAL
  T <* REPLICATED *> = Public BRANDED OBJECT
    theSensor := NEW(Sensor.T).init();
  OVERRIDES
    readTemperature := ReadTemperature;
    calibrateTemperature := CalibrateTemperature;
  END;

<Other methods on the objects.>

PROCEDURE ReadTemperature(self : T) : REAL =
  BEGIN
```

```
    self.temperature := self.theSensor.read();(* Read the actual sensor *)
    RETURN self.temperature;
  END ReadTemperature;

BEGIN END Thermometer_Replicated.
```

On the basis of these two components the code generator produces an interface file `Thermometer.i3` and an implementation file `Thermometer.m3`.

This code uses the Modula-3 facility for encapsulation called partial revelation [33]. The phrase `T <: Public` says that `T` is a subtype of `Public`. However, this is not the complete specification of `T`, it is revealed later (in this example, in the module `Thermometer` using the phrase `REVEAL T = .` This encapsulation facility is not required by the replication architecture however.

The choice of using tags and preprocessing is not arbitrary. A similar effect could be achieved by relying on subtyping principles, so that a class could be automatically replicated if it was subtyped from a 'replicated root' or some such. However, some potential implementation languages do not support multiple inheritance (e.g. Modula-3 [33]). If a class were to be subtyped from another replicated class, e.g. the replicated root, it could not inherit from any other class. Thus, relying on subtyping would mean that dual type-trees would be necessary for all classes which the programmer might want to use in a replicated fashion. This would further imply that potentially large amounts of existing code would have to be adapted for replication, resulting in consistency problems and increased amounts of code to maintain. By tagging a class explicitly in the interface, the programmer makes very direct choices, which remain visible in the application code and will assist debugging and maintenance. Also, new keywords could have been introduced to distinguish replicable classes. However, code containing replicated classes could then not be processed by unextended compilers.

Most of the code enclosing the replicas is automatically generated on the basis of the object's interface. Naturally, the implementation of the object must adhere to the interface specified. Inconsistencies between the interface and the implementation will be detected by the compiler.

7.4 Instantiation of Replicable Classes

An instance of the replicated object type is instantiated by the following example code fragment:

```
IMPORT Replicated, TextList;
...
PE-list := TextList.Cons("host1", PE_list);
PE-list := TextList.Cons("host2", PE_list);
PE-list := TextList.Cons("host3", PE_list);
TRY
  myThermometer := NEW(Thermometer.T).rInit(PE_list);
EXCEPT
| Replicated.Fatal =>
  (* Couldn't instantiate any replicas *)
END;
```

If the statement terminates normally, `myThermometer` becomes a reference to the local surrogate object which intercepts the calls to the replicas. After the surrogate is created, the client can invoke methods on `myThermometer` much as if it was of the original type. It will not be completely identical because the generated stubs for the type `Thermometer.T` require that function-type methods are given collators as arguments (see 7.5 p. 82).

The method `rInit` is defined by the system support layer and prepares the active set within the surrogate by instantiating replica objects on the PEs specified in `PE_list` and setting up references to these within

the active set. The number of PEs specified will determine the maximum number of replicas that will be used for this particular surrogate. During failures, the number of replicas will decrease, and procedures for automatic reconfigurations may be initiated by the system software depending on the particular implementation of the architecture (see §6.3.2 p. 70).

A realistic implementation of the architecture would include support for automatically selecting replica hosts, for example by maintaining a set of hosts able to support objects of the selected type (cf. §10.3.3 p. 109). Such an extension would allow the programming model to be simplified by avoiding list of PE names such as in the code examples given.

Garbage collection
Experience has shown that garbage collection is an essential part of distributed programs; managing the reclamation of distributed objects is a task to be handled by lower-level system software [58, 143]. This architecture assumes that appropriate technology exist to handle this problem. For example, after an application process is finished using a surrogate, its storage should be reclaimed by the garbage collector. Because surrogates do not normally reference each other, surrogates can usually be reclaimed dynamically at process termination. Additionally, when no more references are kept to the object replicas themselves, they are removed.

7.5 Method Invocations

The local surrogate implements the interface of the replicas and will hence accept the invocation of any methods defined for the replicas. However, function-type methods which return arguments are invoked with an extra argument, the collator (the use of collators is discussed further in §7.5.2 p. 83). Method invocation is synchronous in the sense that an invocation does not return until either the invocation has completed, or too many failures have occurred preventing a normal invocation return. In the latter case, the exception `Replicated.Failure` is thrown during the invocation on the surrogate after a timeout given by the underlying RPC mechanism.

If a method is procedure-like, i.e. does not have return arguments, the surrogate returns control to the client as soon as the parallel RPC component has issued invocations to the required replicas (cf. 6.2.2 p. 64). That is, procedures are invoked asynchronously

```
IMPORT Replicated, RealCollator;
VAR
  myTempCollator := NEW(RealCollator.T).init();
BEGIN
  ...
  TRY
    currentTemp := myThermometer.readTemperature(myTempCollator);
    myThermometer.calibrateTemperature(currentTemp);
  EXCEPT
  | Replicated.Failure =>
     (*
       Too many things went wrong at the same time.
       Abandon myThermometer object.
     *)
  | Replicated.Warning =>
    (* There is a potential for inconsistency. Retries may be ok. *)
  END;
  ...
END Application.
```

Guarding each method invocation on a replicated object with exception handler code clearly adds to the complexity of writing the program. However, as in all distributed applications, handling partial failures in a secure manner can significantly increase the reliability of the application, and it might bring benefits in terms of a more maintainable program. Note also that statements can be grouped within a guarded block, thus amortising the cost of writing extra exception handling code. It is often worth considering what the application should do to avoid crashing in the event of insignificant mishaps and failures. However, use of exception handlers is not enforced by the system, although the compiler will produce warning messages when it encounters potentially unhandled exceptions. Unhandled exceptions lead to run-time failures, and will therefore halt the application. Reliable applications should therefore include code to handle exceptions.

7.5.1 Method parameters

Argument passing in distributed object systems is limited by several compromises [52]. Many RPC systems therefore limit the range of types that can be passed as arguments or use non-scalable techniques. The main difficulties arise with objects as arguments. Two main approaches to parameter passing in object systems are recognised:

Call by value will copy all objects reachable from the argument object between the caller's and the callee's address space[1]. For large object graphs, in which a vast number of objects may be reachable from the argument object, this approach can be very costly, and it also introduces problems with duplication of objects which eventually lead to consistency problems.

Call by reference simply passes a remote reference to the object as the argument. Passing objects by reference is the natural approach to argument passing in object systems, and the architecture proposed here, constructs new surrogates for objects passed as reference. Call by reference is beneficial in terms of efficiency, simplicity and consistency. However, a disadvantage with this approach is a lower availability than a deep copy approach. As the size of a distributed system increases, the number of components which are required to work will increase for a remote object to be available. Essentially, a remote reference is fragile.

This architecture does not address this problem in any further depth as underlying RPC technology will largely determine how arguments are passed among invoker and invokee.

Exceptions
Exceptions defined for the methods in replicated classes are not handled by the current architecture. An extension of the collator design is necessary to do this (see §10.3.4 p. 110). As exceptions are a special kind of result parameter from invocations, and because different exceptions should be handled differently by the programmer, the interface of collators could be extended with new add methods for each exception returned by the remote invocation.

7.5.2 Collators

Normally, the surrogate will receive identical results from the object replicas. However, there are situations where this is not the case, for example during failures, when replicas might compute different results. It might also happen that different results are the correct behaviour. The methods executed in the different replicas could compute results depending on state local to the hosting address space or PE such as random numbers, timestamps or replicated sensors as in the thermometer example above. In this situation, only the application semantics can determine the correct interpretation of the results. By

[1] There are variations of the call by copy approach which are determined by the depth of the copy. See [52] for a more complete discussion.

using a specially designed collator, the application builder can easily provide these semantic rules to the surrogate.

More interestingly, results from the replicas may be references. For example, if the replicated service implements a handle-like coordination model [2] the replica may return a reference to another service within the system. Consider the code fragment below specifying a manager of service objects. The manager is responsible for creating and returning handles on temperature sensor service objects.

```
INTERFACE Service_Manager;
IMPORT Thermometer;
TYPE
  T <* REPLICATED *> <: Public;
    obtainTemperatureSensor() : Thermometer.T;
  END;
END Service_Manager.
```

The method `obtainTemperatureSensor` returns a reference (a handle) to a thermometer, and because the object `Service_Manager` is replicated, multiple references are returned to the calling object (the surrogate for the replicas). They are individually unique references to distinct objects local to each service manager, and are essentially a new group of object replicas. To maintain the illusion of surrogates concealing replication, the returned references are used as replicas for a new surrogate. The new surrogate must be created in the client address space, constituting a new manager for the objects referenced by the returned collection of references (cf. §6.2.3 p. 66).

Programming interface
The collator encapsulates a single task; processing method invocation results from object replicas. The programming interface of a collator is presented below (see §A p. 111 for an example implementation):

```
(* Class interface for Collator.T type *)
TYPE
  T <: Public;
  Public = OBJECT METHODS
    init() : T;
    prepare(nReplicas : INTEGER);
    add(e : Elem.T) : BOOLEAN;
    addFailure() : BOOLEAN;
    getResult() : Elem.T RAISES {Fatal};
  END;
```

As collators are only directly used by the system support mechanisms, they must comply with this exact set of methods. The type of the result parameter `Elem.T` must naturally correspond with the type of the result from a particular method.

All collators must define the four methods `prepare`, `add`, `addFailure` and `getResult`, a method named `init` is not required but the surrogate requires a correctly initialised collator for each method invocation. The four required methods are used as follows by the architecture:

`prepare` notifies the collator about the number of replicas which are currently active.

`add` is called by each thread to input results for processing to the collator. The method returns `TRUE` if this was the last result required by the collator and `FALSE` otherwise. A `TRUE` response signals the surrogate that `getResult` is ready to retrieve a processed result.

`addFailure` is called by a thread if the replica failed to return a result. The method returns `TRUE` if the failure of this replica makes normal result processing impossible, for example if a majority

of replicas is required and this failure implies a majority of failures. A `TRUE` response signals the surrogate that an exception should be returned to the client.

`getResult` is called by the surrogate to retrieve the processed result. This method blocks until the result is ready. If exception `Fatal` is raised, the surrogate returns the `Replicated.Failure` exception to the client.

The fact that the number of PEs specified during initialisation is the maximum number of replicas that will be available must be recognised by the application programmer during construction of specialised collators. Collators should not be dependent on particular numbers of replicas, but rather use majorities or some other relative measures.

The method *prepare* is used by the surrogate to inform the collator about the current number of replicas in the active set, and this information should be used by the application programmer to define rules for relative numbers of replies necessary to produce valid results (see §7.5.2 p. 83). The programmer should make few, if any assumptions about the number of replicas in the specification of the collator objects.

As long as this interface definition is adhered to, any processing allowed by the particular implementation programming language can be performed within a collator. This allows for very flexible and powerful collators to be built. Additionally, once a collator is constructed they are simple to reuse. For example, it is a trivial job to modify a collator for integers to a collator for floating-point numbers. Similarly, it does not require much effort to modify a standard majority voting collator to a collator which also performs weighting of the results.

Backdoors
A collator is an object which is passed in to the surrogate via method invocations that have return parameters. During the processing within the surrogate, results are added to the collator as they arrive in from the parallel RPC module, and the `getResult` method on the collator is used by the surrogate to retrieve the processed result. The `getResult` method has the same return type as the corresponding method on the surrogate, and therefore returns a single value (although it might be composite).

However, the programmer might occasionally need to manipulate sets of results, rather than the singleton which is returned from the surrogate via the `getResult` call. The programmer is free to implement other methods on the collator which can return other results, although care must be taken to avoid causing name-conflicts with the required interface of the collator. A backdoor method could for example return an array containing all the results returned via the `add` call.

7.6 Sharing of Surrogate Objects

Sharing in an object oriented system is achieved by passing references as parameters in method invocations (both input and output parameters). For example, object A can initiate sharing of an object B by giving an object C a reference to B. Both A and C are now able to invoke methods on B.

The architecture enforces some particular procedures for sharing of surrogates. Because a surrogate should always reside in the same address space as the client, a new surrogate is created if a client A passes a surrogate reference to another object C in a different address space (cf. §6.3.3 p. 72). However, this will be performed automatically and is transparent to the programmer. Due to automatic creation of surrogates, two references to surrogates in different address spaces will generally not be equivalent even if they manage the same set of replica objects. Sharing of a surrogate among to objects within the same address space does not require any particular processing.

7.7 Failure Semantics

A replicated object will have different failure semantics from local or non-replicated remote objects, and although it will be more available than a normal distributed object, there are failure situations which cannot be concealed by the replication mechanisms. To allow applications to handle the new failure modes, the programming model defines a new exception *Replicated.Failure* which is raised when the serialisation scheme fails to gather a response from a majority of the replicas or a collator fails to receive results from enough replicas. This exception therefore reports a very critical situation within the system. Normally, an application will have to abandon such an object and create a new surrogate with new replica objects.

Additionally, during serialisation of replica operations, the majority locking scheme might detect unreleased locks which might be due to a premature surrogate death (cf. §6.3.3 p. 72). The surrogate raises an exception `Replicated.Warning` if unreleased locks were detected and had to be explicitly unlocked. If so, the surrogate does not attempt to invoke the replicas, but leaves the replicas unlocked before returning the exception. Because the replicas may still be mutually consistent (if the locking surrogate crashed after the invocation was executed), an application may choose to retry the invocation.

7.8 Concluding Remarks

This chapter has described the programming model of the proposed architecture. Evidently, some complications are necessary in programs using the architecture, in particular because an application might need a relatively large number of different collators to suit the different method return arguments. However, as collators are simple to reuse, in particular for different result data types, the added complexity will mainly be observed as an increased number of objects in the application.

In summary; most of the underlying complexity is hidden by the architecture. The programmer is completely shielded from for example the serialisation protocol and failure masking functions. It is therefore believed that the complications necessary will be outweighed by the benefits the application receives in the form of increased reliability.

Chapter 8

Realising the Architecture

This chapter presents a prototype implementation of the proposed architecture as described in chapter 6. The prototype is by and large experimental, it is not a complete implementation of the architecture. However, it does demonstrate the key benefits of the design, such as the simple programming model which is described in 7. The application which has been built to exercise the prototype shows that very limited programmer effort is necessary to use the system support functionality. Additionally, the chapter might be valuable for later implementation efforts, perhaps on other platforms; the chapter describes how existing system support software influences its functionality and ease of implementation. The performance measurements might be beneficial for comparative studies of other architectures, but also for later implementation on platforms with other system characteristics.

8.1 Overview

The prototype is implemented in Modula-3, a statically typed, type-safe, compiled, object-oriented programming language [33]. A range of useful features justified this particular language; for example support for remote object invocations, concurrent programming via threads, exception handling, strong emphasis on the separation of interfaces and implementations, a vast range of libraries and built-in automatic garbage collection both for local and remote objects [22, 24]. Having these facilities available meant that the construction of the prototype itself could be significantly simplified. However, the implementation does also expose some limitations in this programming environment. For example, the remote object facility assumes a quite static partitioning of the objects within the application which somewhat complicates the programming model in the prototype (cf. §8.3.2 p. 91).

The prototype is built as a collection of static library code and surrogate code derived from programmer specified classes and the abstractions herein have been derived from previous experiments [67]. Some collators have been built as well to demonstrate the simplicity of the design. Applications import the library code and use the derived surrogate code in place of the originally specified classes to gain support for replication.

A sample application has been built to experiment with the replication support code and to act as an instrumented testbed to allow for performance measurements. Although the application is far from a realistic application, it confirms the simplicity of the programming model. Applications' use of the prototype is further described in §8.4 p. 92.

The rest of this chapter is structured as follows. Section 8.2 describes the environment in which the prototype has been implemented. Section 8.3 presents the internal design in detail. Section 8.4 presents the sample application exercising the prototype. Section 8.5 contains a discussion of the prototype performance. Finally, section 8.6 contains a summary of the chapter.

8.2 Implementation Platform

The prototype is implemented using DEC SRC's version of the Modula-3 compiler for SunOS 4.1.3 running on Sun SPARC workstations interconnected via Ethernet networks. Although this compiler is ported to several other platforms as well, these platforms have not been used in conjunction with the prototype. However, if porting to other platforms is necessary, it should be a relatively small effort, as only small parts of the code are bound to the hardware and operating system platform (cf. §8.2.3 p. 89).

Only non-persistent Modula-3 objects have been considered in the implementation, but extensions of the implementation might draw some benefits from Modula-3 persistence technology, for example to support atomic invocations on the replicas, so that in case the surrogate crashed before invoking all the replicas, the invoked replicas could be rolled back to the previous state. Further, reconfiguration of failed replicas could be simplified to increase survivability for long-running applications. A big advantage of DEC SRC's compiler is that it comes with a large collection of useful library code which can be reused in other applications. Some of these libraries are generic, and their instantiation can thus be parameterised for a collection of types. This can be a big advantage for the construction of new collators (see also §7.5.2 p. 83).

8.2.1 Existing system software

Some existing system software has been used to build the prototype. Most important is the library developed for Modula-3 to support distributed objects (Network Objects), but also other features such as the library for IP (Internet Protocol) functions, threads, generic lists and tables have been used to speed up the implementation work.

Remote invocations
An RPC facility to invoke methods on remote objects is necessary to implement the architecture. The prototype is built using Network Objects[1], a powerful RPC mechanism developed for Modula-3 [22]. Network Objects extends the notion of invoking methods to include remote objects, and supports arguments much like local method invocations do. References are valid both as input and output parameters in method invocations, and because Network Objects differentiates between local and remote object references, network objects are passed by reference while other objects are passed by copy. Furthermore, Network Objects ensures that all remote references are direct references between two address spaces by constructing surrogates in each referencing address space which communicates with the referenced object directly. In contrast to forwarders [71], this mechanism is more resilient to failures, but requires additional communication to avoid reclamation of non-garbage objects [22, 143]. Further, by always creating local surrogates with direct references to the remote object, Network Objects directly supports the proposed approach to object sharing (see §6.3.3 p. 72).

Of crucial importance for a reliable implementation is the manner in which failures are managed by the RPC technology. Network Objects do not support asynchronous calls, and it is therefore able to provide at-least-once semantics using exceptions to notify the caller if the remote invocation failed. In case an invocation has no specified return parameters, the client of a remote object blocks until a dummy result[2] is returned. All invocations are hence synchronous, i.e. the caller is blocked until the invocation returns or the runtime reports a failure.

Network Objects amends the fail-stop TCP/IP protocol with mechanisms for simpler failure detection and reporting by raising exceptions in the client if the remote address space has died[3]. However, this

[1]By convention, Network Objects (capitalised) refers to the Modula-3 RPC mechanism, network objects (uncapitalised) refers to objects which are invoked remotely.

[2]Such a fake return parameter is added automatically by the Network Object stub generator [22].

[3]Death is conservatively assumed if the remote address space doesn't answer ping messages.

exception may be raised due to transient network errors causing either a remaining orphan object method execution in the remote address space, or a prematurely collected object in the server.

Premature garbage collection in Modula-3 Network Objects might happen. However, the probability of such events is very low. Only if the Network Object runtime, running on the same machine as the owner of a remote object, erroneously decides a client has failed will a remote object be reclaimed too early [24]. In the current implementation, this check is done using a sequence of TCP/IP 'ping' messages[4]. Additionally, if the client tries to invoke a prematurely collected object, the Network Object runtime will raise an exception to warn the client.

8.2.2 Nameservice

Location independence in distributed object systems is achieved through the use of *nameservers* together with remote object references appearing like local references. A nameserver is essentially a simple database which stores *(name,location)* tuples. Clients can query the nameserver for the location of named objects, and indirectly retrieve a reference to the object. However, in a realistic system most objects will not be registered within the nameserver. Rather, they only exist as anonymous objects, only referenced from other objects, e.g. the object that created it [58]. Additionally, [41] reports that name lookup operations have a significant impact on system performance. As a natural consequence, nameservers are often replicated using weak optimistic consistency schemes (cf. §4.5 p. 49).

Network Objects provide access to a simple (non-replicated) nameserver which is used by the surrogate to locate object replicas on PEs specified by the client. Hence, this nameserver must be running on each PE in the network which will be used to host replicas for the prototype.

8.2.3 Portability issues

System support mechanisms should, as far as possible, be portable. Underlying software and hardware should be expected to evolve, hence system support mechanisms should not make excessive assumptions about their constancy. However, by nature, system support mechanisms are closely associated with a certain system model.

Although the prototype has been implemented in Modula-3, other object oriented languages should be possible to use. A very interesting platform for further experiments would be Java from Sun [82].

Implementing the architecture requires an object oriented programming language with support for remote method invocations and a failure reporting mechanism which allows the caller to determine failures. The prototype makes use of very few platform dependent functions, one of which is the use of some communication protocol functions (IP) to support the location and naming of PEs in the network. However, such functionality is likely to exist in other network environments.

8.3 Prototype Design

The prototype follows the module structuring presented in §6.2 p. 62. The object diagram shown in figure 8.1, using the notation of Booch [29], illustrates the internal design.

Essentially, the client instantiates the surrogate object (much as it would instantiate the corresponding non-replicated object) and passes collators in as a method argument to the surrogate in case the method requires result processing. The surrogate maintains an active set containing the replicas specified by the client in the instantiation call. The parallel RPC module updates the failure status of individual replicas in the active set as failures are detected. It also records failures in the active sets stored in the replicas

[4] A 'ping' is a special message in TCP/IP which checks if the remote connection is still alive by echoing a message in the remote address space.

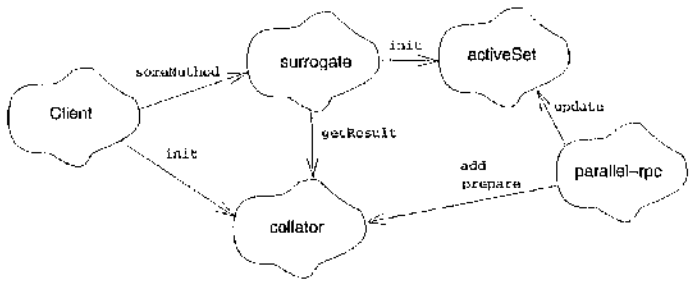

Figure 8.1: Surrogate object diagram

via the *lock* and *unlock* methods. What should be emphasised in the figure is the very simple interface to the collator; although the code residing within the surrogate and the parallel RPC module is complex, the collator is manipulated using very simple method calls (cf. §7.5.2 p. 83). The rest of this section presents these components in more detail.

8.3.1 Surrogates

Surrogates are directly derived from programmer specified classes. Although they are hand-crafted in the prototype, surrogates are relatively generic and could thus be automatically generated by a stub generator for example. Most of the code within surrogates is invariant over different replica class types, and this will simplify an eventual code generator.

The issues regarding naming conflicts and implementation platform[5] must however be considered if such stub generator technology was to be built. For example, the problem of name conflicts can normally be solved by generating identifier names which are concatenations of the application identifier names and a substring specific to the code generator. Naturally, this assumes that the compiler technology allows identifiers with such length.

Figure 8.2 presents the class diagram of the central programmer defined classes, generated classes and static library classes. Note that the names used in the prototype implementation are not consistent with the programming model specified in chapter 7 due to the experimental nature of the implementation. A client in the prototype gains access to the surrogate through the name foo_srgt.T rather than foo.T as would be the case for a realistic implementation of the architecture.

Class foo.T is the interface for the class to be replicated and is defined by the programmer, and the class foo_server.T implements foo.T (the reason for this particular partitioning is explained in §8.3.2 p. 91). Only the class foo.T forms the basis for the generated code however; both foo_srgt.T and foo_act.T are generated from foo.T. This is illustrated by the dotted lines.

The prototype separates some of the generic functionality of the surrogate into the class Replicated.T which foo_srgt.T inherits. The class foo_act.T might seem unnecessary, foo_srgt.T could just manipulate a collection of remote foo.T objects to implement replication. However, to support sharing of replicas among multiple surrogates, serialisation must be enforced. The class foo_act.T simply amends the interface foo.T with the methods *lock* and *unlock* to support the majority locking scheme described in §6.2.4 p. 66.

The surrogate uses the built-in Network Objects exceptions `NetObj.Failure` and `Thread.Alerted` to detect and mask many failures from the client. However, in accordance with the programming model (§7.7), the surrogate may return the exception `Replicated.Failure` which signals that the surrogate cannot carry out any client requests, or the exception `Replicated.Warning` if the surrogate had to break locks in the replicas and thus potentially infringe on consistency.

[5] Discussed in §5.2 p. 53.

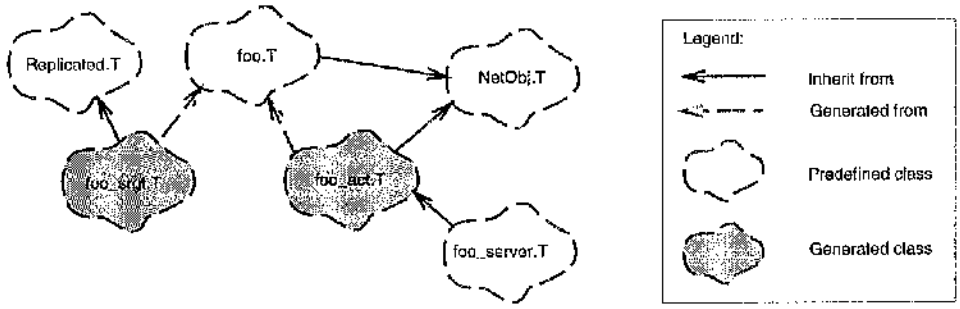

Figure 8.2: Code generation class diagram

8.3.2 Implementing object replicas

The Network Object library enforces restrictions on the implementation of remote objects. However, violations of these restrictions will be reported during compilation. A replica object, due to the fact that it is remote and consequently must be a subtype of `NetObj.T`, can not be instantiated quite as simple as a local object. A replica object is not mobile, it must be instantiated by a server program running on the PE where it will become accessible. Replicas are only available while the server is running. Such replica servers are registered with the name server process, `netobjd`, running on the PE. The replica servers must export object names which can be recognised by the surrogate code, and in the prototype this name is foo_act.T.

The programmer is responsible for building and starting replica servers on the PEs which will be used[6]. In more realistic applications where servers are long-running processes, the system might be configured to start up the servers during the booting of the machine.

Further, Network Objects are restricted to be *pure* objects, meaning that interfaces cannot expose part of the class' internal state. However, this restriction conforms to the object model advocated in this dissertation (cf. §2.4 p. 21) and does not incur problems for the architecture.

Separation of interface and implementation
Network Objects require that the type T is fully revealed to generate stubs for it. Thus, T cannot be declared as opaque in an interface T.i3 and revealed in a module T.m3. As the architecture assumes that replicable classes are implemented in separate interface and implementation files, the programmer must implement type T in another module, the module called foo_server.T in the diagram. This restriction is slightly cumbersome, as it increases the complexity of the software. However, other implementation platforms, such as Java, might not enforce this restriction.

8.3.3 Parallel invocations

Parallel invocations are implemented by a collection of threads managed by the surrogate. In fact, a separate thread, the thread manager, within the surrogate is responsible for managing the parallel invocations. Other threads are responsible for synchronising the gathering and releasing of locks with the parallel RPC thread manager. This separation of responsibilities within the surrogate increases the performance of the surrogate, for example by allowing the surrogate to return results back to the client before the releasing of locks has started. Additionally, as most of the surrogate contains boilerplate code, the added complexity does not affect the application programmer using the surrogate.

Each thread in Modula-3 is a closure, which defines the thread's shared variables and the procedure in `foo_act.T` which is called by each thread managed by the thread manager.

[6]The process of starting servers can of course be automated using, for example, startup-scripts.

8.4 An Example Application

A small application has been built to exercise the prototype implementation, mainly to facilitate the performance measurements, but also to experiment with different collator designs. This testbed application has the form described below.

```
MODULE Client EXPORTS Main;
IMPORT IO, Fmt, TextList, IntCollator, RObj_Surr, Replicated;
VAR
  myFibCollator := NEW(IntCollator.T).init();
  rRef          :  RObj_Surr.T;
  hostList      :  TextList.T;
  result        :  INTEGER := 0;
  objName       :  TEXT := "RObj_Act.T";
BEGIN
  hostList := TextList.Cons("unimak", hostList);
  hostList := TextList.Cons("campbell", hostList);
  hostList := TextList.Cons("agattu", hostList);
  TRY
    rRef := NEW(RObj_Surr.T).rInit(objName, hostList);
    result := rRef.fib(n, myFibCollator.init());
  EXCEPT
  | Replicated.Error, Replicated.Warning =>
    IO.Put("Fatal srgt error. Exiting.\n");
  END;
  IO.Put(Fmt.F("result = " & Fmt.Int(result) & "\n"));
END Client.
```

The code fragment above shows how an application program can include support for object replication. As a result, `rRef` is a more reliable object. If the probability of failure is 0.01 for each of the workstations unimak, campbell and agattu, then this application has reduced the probability of failure due to failures in `RObj_Surr.T` to 0.0003, corresponding to an increase in MTTF from 100 days to over 9 years[7].

8.5 Performance Measurements

This section presents a few initial performance measurements which have been sampled from the prototype. The results are generally assuring; the prototype yields a performance similar to what should be expected for the given system platform. However, the architecture has not been carefully optimised, although the design has focused on efficient and light-weight implementation principles. The samples were gathered during the night in periods when the workstation and network usage was low. A number of samples[8] were taken for each configuration and these were averaged to give the results presented. This can naturally be only an indication of the performance of a prototype implementation in a given application context. The performance measurements on the prototype are, like the prototype itself, initial and incomplete but do nevertheless give an idea about the efficiency and overheads in the architecture.

It is not clear from the samples taken what is the most important source of overheads, or if there is only one such source. Presumably, large overheads, e.g. round-trip delays, exist in the network communication subsystem, but it is difficult to say how these vary with increasing numbers of communication channels. The measurements used a number of similar workstations to avoid extra factors of uncertainty such as different operating systems, processing power and memory performance.

[7] Refer to appendix B for calculations of failure probabilities.

[8] Either 100 or 1000 samples for each configuration.

The table below shows the times in milliseconds for different numbers of replicas, n. During this particular test he servers invoked empty procedures in the replicas and the majority locking scheme was used in each sample. The coloumn "only RPC" is the number obtained from using a surrogate not containing any replication code.

n	only RPC	1	3	5	9
execution time	4.7	15.6	24.0	41.6	119.7

Figure 8.3: Execution times in milliseconds for different numbers of replicas.

During other, single-sample experiments, replicas were crashed manually to determine the influence of the failure detection mechanisms in Network Objects. No difference in the execution times could be identified which suggests that the failure detection mechanisms are very efficient.

Other performance factors
The measurements were taken on a single user of the surrogate only. A more extensive performance sampling should include measurements on multiple clients residing in different address spaces.

In this scenario, the generated load on the surrogates is likely to be rather evenly distributed, and thus give small performance advantages from the use of wait-for-first collators. Additionally, due to the extra rounds of network messages necessary for the serialisation protocol, performance will be lower than for only a single surrogate.

A 'wait for first' collator can result in improved performance only in the case that the surrogate is not constantly loaded with client requests [64]. The performance with different collators is therefore dependent on application behaviour and on the processing capacity in the PE hosting the surrogate. If clients keep the surrogate busy the surrogate will have to wait for the replicas to be unlocked anyway to maintain consistency. Other collators, such as weighting collators, will increase the processing time in the surrogate, but this processing is distributed, and will therefore scale well with even high numbers of clients.

8.6 Summary

This chapter concludes the discussion of the proposed architecture. It has demonstrated the usefulness of the architecture by describing a prototype implementation and providing some initial performance measurements. The next chapter focus on how this architecture relates to other work in the area.

Chapter 9

Related Work

Replication is an important approach to increasing availability in distributed systems. A large number of systems using replication and research efforts investigating various replication management techniques have been presented in the literature — a number far too great to allow for individual treatment within the scope of this work. For that reason, a selection has been made; this chapter covers work of particular relevance to the architecture described in the dissertation. This includes a discussion of other system software components of particular importance to application builders such as highly available distributed file systems. Some overlap of material occurs, intentionally, in particular with respect to chapter 4 which makes several references to the work discussed in this chapter. The discussion here is more focused, however, on comparisons with the proposed architecture. The chapter is divided into four sections; the first two emphasise system support for application developers, and the last two sections are focused on replication used within application components such as middleware and databases.

9.1 Language Level Support for Replication

Language level support involves the provision of libraries and automated code-generator tools to assist developers with reusable components to be included within the application. Language level support is a flexible approach to system support, although it normally incurs some complications to the programming model.

The encapsulation principle is the foundation for the architecture presented in this dissertation, but similar ideas have been investigated before. The remote procedure call (RPC) has been the traditional mechanism to facilitate interaction among programs in separate, potentially geographically distributed, address spaces (see e.g. [132]). In [48] the remote procedure call abstraction was extended to deal with both replicated invokers and invokees.

Object oriented techniques made possible an even more high-level abstraction; that of the remote invocation. Remote invocations conceptually integrate binding and service handle in the reference mechanism and provides a uniform mechanism for service invocation [135]. Potentially, an invoker need not observe any difference between a local method call and a remote method call, although full transparency is not normally desirable[1]. The reference essentially conceals locality and access mechanism. A particularly beneficial attribute of such uniform references is that they facilitate the construction of proxies [167] which creates a potential for concealing much of the added complexity with replication.

[1] See also §2.5 p. 22.

9.1.1 Replicated procedure calls

A major influence on the mechanisms proposed in this dissertation is the work of Eric C. Cooper on replicated procedure calls [48, 49]. Cooper's approach, based on groups of independently executing state-machines called *troupes*, supports both dynamic reconfiguration and call coordination. Replication transparency is maintained using a full consistency scheme where each troupe member receives each client request. This is necessary due to the mutating nature of the troupe members, which are similar to the object graphs suggested here (cf. §7.2 p. 79). Furthermore, automatically generated stub-code conceals the notion of multiple server handles within the run-time system, and the name service's interface supports troupe-handles. Similarly to the proposed architecture, clients of troupes do not need to be aware of replication.

Troupe members are assumed to be completely independent program modules behaving like identical deterministic state machines. However, in contrast to the architecture proposed here, Cooper's work is based upon a transactional system model with persistent troupe members and requirement to network support for multicast. The persistent troupe members allow for more loosely synchronised serialisation of concurrent clients. An optimistic scheme for serialisation of client requests is suggested. This serialisation scheme allows two transactions to be committed concurrently if they are committed in the same order in all troupe members. Although it is not completely clear from the presentation how it detects that two transactions are in conflict, it can be assumed to depend on some inherent global ordering of client requests[2]. Similarly to the proposed architecture, Cooper does not require any synchronisation protocol among the replicas themselves as troupe members are completely unaware of each other. Rather, the client troupe members are responsible for gathering replies from replicas that are ready to commit. This is likely to give good scalability properties for high numbers of clients, although it does enforce extra functionality within the client. Cooper can guarantee atomicity for this optimistic scheme due to the use of non-volatile storage and specially designed procedures in the troupe members to implement specific *ready_to_commit* procedures, thus requiring some interference from the programmer of the troupe members.

Rather than using light-weight threads allocated to each replicated call as in this architecture, Cooper assumes that the network itself is able to multicast messages to the troupe members [49], although the actual implementation simulates this parallelism with sequential messages [48]. The dependence to network multicasts also complicates Cooper's architecture in case of heterogeneous networks, where timeouts will need to be individually adjusted to achieve good performance. The architecture proposed here does not require network support for multicast, rather a connection is made from the surrogate to each replica. The added cost of maintaining multiple connections is not negligible, but for relatively small numbers of replicas as envisioned by this architecture (normally less than 10 replicas, see 6.4.2 p. 76), it will not constitute major overheads. Individual connections can therefore have distinct timeouts, improving the performance in heterogeneous networks. Results gained from evolutions of the RPC2 and the MultiRPC system [159] used in Coda [158], suggest that using concurrent threads for replicated calls can incur significant overheads for larger numbers of replicas however, and, naturally, multicast primitives help reduce network load.

In contrast to this architecture, Cooper addresses the issue of nested invocations; the replicated procedure call manages both many-to-one and many-to-many calls using a specially designed binding agent (a name service) for troupes. Duplicated calls are always filtered at the invokee, thus achieving high fault-tolerance at the expense of generating more network traffic. Call coordination, i.e. filtering of calls from replicated callers, is achieved by assigning a unique ID to a set of calls originating from troupe members. A special ID is reserved for calls from non-replicated troupes. Invokees, normally other troupe members, are therefore able to detect duplicated calls and can ignore all but the first which is executed normally by the invokee. The binding agent makes Cooper's approach more replication transparent for troupe clients; troupe members can dynamically join the group at run-time by joining a troupe. This is invisible to clients. In contrast, this architecture assumes that clients are responsible for specifying replicas during initialisation of the surrogate. The benefit of this approach is that the programmer can focus on the

[2]The commit protocol is only sketched out, it is not part of the implemented architecture.

particular needs of the application without being concerned with the replicas. Cooper's approach puts this responsibility onto the designer of the troupe members themselves.

Additionally, the concept *collator* was suggested by Cooper as a mechanism to allow various levels of synchrony between the client and the troupe server. Cooper's collators are exploited further in the mechanisms proposed here to benefit from object-oriented concepts (cf. §6.2.3 p. 65). If references are the return argument of function-type methods, new surrogates are created automatically to conceal the notion of replica references. This extension of Cooper's architecture follows naturally from the object oriented system model adopted in this dissertation.

Recently, a more direct extension of Cooper's work has been suggested with the name CopyCat [102]. In contrast to the architecture proposed here, CopyCat is a non-object approach based on the standard RPC paradigm. The main feature of CopyCat is the flexibility of semantics; the programmer can, depending on the application, relax the ordering constraints enforced among the invocations. Three ordering types are supported; causal, forced and immediate. Causal ensures that replicas deliver messages in the same order as they are sent from the client, forced ensures causal ordering among multiple clients, and immediate guarantees full ordering among all messages from all other clients. The architecture proposed here provides a more encapsulated approach than CopyCat, but does enforce full ordering of all requests. Assuming that clients of object replicas will be able to optimise the ordering of calls, CopyCat is able to achieve better performance than the proposed architecture, but this kind of optimisation does introduce extra coupling between clients and the invoked modules. CopyCat is an example of a replication approach in which transparency of replication is traded off for higher performance.

9.1.2 Gaggles

Another important influence on this work is the *Gaggle*, described by Andrew Black and Mark Immel [28]. A Gaggle is a software construct that implements a non-deterministic choice among a collection of clerk objects. The main idea of Gaggles is that they appear to the client just like a normal, singular object. The basic assumptions behind this work and the Gaggle are thus very similar; presenting the programmer with a surrogate which can conceal the notion of multiple server-objects so as to hide complexity and provide a layer in which different functionality can be implemented without needing to change the client of the surrogate.

However, in contrast to the surrogates described here, and the replicated procedure call abstraction proposed by Cooper [48], a Gaggle is neither primarily concerned with consistency nor serialisation, it only implements the selection of a new replica if one is discovered to have failed. Essentially, a Gaggle implements the failure masking functionality necessary for replication, but some form of underlying replication scheme is assumed to be available for the purpose of consistency and serialisation, for example ISIS process groups [18]. Thus, a Gaggle must be extended with a replica consistency scheme such as voting or process groups to be used for replication. The architecture described here combines the idea of a Gaggle and a consistency mechanism into a single abstraction, while maintaining the transparency of replication.

9.1.3 Fragmented objects

The Fragmented Object (FO) model for replication is a programming paradigm which is an extension of the proxy principle [60, 116, 117]. This approach suggests that fault-tolerant objects are structured into fragments which communicate using special connective objects. Fragments are always local to the invoker, and the connective objects are responsible for maintaining consistency among the multiple fragments. The key idea behind the FO-model is that of client transparency; each client is presented with a local interface, a proxy, to a local fragment [117]. This is in correspondence with the notion of local surrogates as advocated in the proposed architecture; they provide distribution transparency while increasing the fault-tolerance of the system.

However, fragments are specifically constructed for cooperation and consistency management. As a

consequence, the programmer must design objects especially for replication, as connective objects must be used explicitly. The key benefit of these connective objects is they can implement different consistency schemes, and can be used to optimise the performance of synchronisation between fragments. The FO-model is not an approach that advocates replication transparency for the designer of the objects, and therefore it is not an object replication strategy. Rather, the FO-model is a good example of a replicated objects approach.

While the FO-model gives the programmer a great deal of flexibility, it comes at a cost of increased efforts necessary for the creator of the fragments. Additionally, the problem of nested invocations must be addressed by the programmer in the FO-model. If a fragment contains references to other shared objects, potentially fragmented objects, it is the responsibility of the invoker to ensure that the shared object is invoked the correct number of times through the use of appropriate connective objects.

Adaptable Replicated Objects (ARO) [31] has been proposed as an approach to extend the Fragmented Objects model with technology from the BOAR libraries of support code for replication management such as consistency managers [83]. Instead of having to implement connective objects from scratch, the idea is that the creator of the fragments can simply use consistency managers from the library and thus reduce the efforts needed to construct the fragments. However, the FO-model is maintained; the approach trades low-level concurrency control efficiency for replication transparency.

9.1.4 Reflective programming

Some object-oriented programming languages support the notion of reflection. Reflection allows a program to change its own behaviour by modifying, at run-time, its meta-data. For example, a reflective program might dynamically change the way it reacts to method invocations, which can be used to implement serialisation functionality during concurrent access [179]. This property has been exploited in Open-C++ to implement various replication techniques, in particular an object replication scheme [65]. However, solutions to inherent problems with object replication are only briefly mentioned; nested invocations are not addressed in this work.

The approach of reflective programming is not fundamentally different from the approach suggested in this dissertation. It is simply a more flexible approach to the *implementation* of replication management. Rather than depending on code generators to produce intercepting surrogates, a reflective program can *dynamically* produce such surrogates.

9.2 Replication in Programming Systems

Programming systems encompass more extensive support for development of long-lived and usually concurrent software than do programming languages. In particular, distributed object-based programming systems normally provide functionality for persistence, transactions and object sharing among multiple applications [154]. Although the programming systems discussed here are implemented on top of operating systems, the distinction between programming systems and operating systems is blurred with the introduction of more flexible object-oriented operating systems such as SPIN and Spring [15, 125]. In the future, programming systems might be fully integrated into the underlying operating system. Similarly, a clear trend drives the integration of traditional database persistence technology with programming languages, which further blurs these boundaries [9].

The support for persistence and transactions can have a significant impact on the kinds of replication mechanisms that are used. In particular, it can facilitate the use of more optimistic replication techniques.

9.2.1 GARF

A system supported approach to object replication has been developed as part of the GARF system [121]. GARF is a programming environment which provides run-time support for object-oriented distributed applications, and also partial support for object replication.

Fault-tolerant objects in GARF are implemented in two layers, normal application objects and a communication layer used for managing invocation filtering among replicas. If they need to be replicated, application objects are associated with such communication objects. A communication object is a symmetric extension of the traditional proxy [167], it exists as a representative on both the invoker's and the invokee's node. These objects have two responsibilities; pre-filtering of invocations from a group of invokers, and replicating invocation replies back to the invoking replica group. After the invocation replies have been filtered, identical copies are passed back to the invokers. Although Cooper's replicated procedure call architecture filter duplicates at the invokee [49], GARF appears to be similar in replication functionality. However, the system model adopted in GARF, based on non-persistent objects, achieves atomic multicast by being mapped onto ISIS process groups [18].

In contrast to the proposed architecture, GARF does not support the use of collators. Therefore, GARF cannot tolerate any replica failures in the data-domain or optimise performance as with the semi-asynchronous wait-for-first collator suggested here. GARF can only tolerate fail-stop failures among the replicas. However, the use of ISIS process groups implies that GARF can provide stronger guarantees for consistency, in particular in the presence of client failures.

9.2.2 Arjuna

The Arjuna system [171], developed at Newcastle upon Tyne, is a distributed programming system that supports replicated persistent objects. Compared to the architecture presented here, Arjuna trades high survivability and transactional correctness for lower performance. It combines an extensive collection of tools and building blocks to form a programming system for distributed software development in C++, and has been tested in a number of applications [170].

Fault-tolerance in Arjuna is based upon the notion of (nested) atomic actions which are transactions encapsulating object invocations, and the programmer is responsible for declaring transaction closures—using special directives. Essentially, groups of invocations on replicated persistent objects are explicitly grouped to indicate atomic actions. Replication in the commercial version of Arjuna is based on a primary-copy scheme, but support for active replication has been investigated although not implemented [170, 113].

Arjuna does not support collators, i.e. programmer defined processing of replica results. Rather, as in GARF it is assumed that replicas return identical results and that any result is as good as any other.

9.3 Replication in Application Components

Replication techniques have also been used in application components such as file-systems and database systems. The main difference between replication in such settings and generic support functionality of replication, is that application components exploit knowledge of the semantics of the data being managed. Additionally, some systems are built on the assumption that inconsistencies are visible outside the system as failures, and that clients are able to take corrective actions. This is in contrast to the system model adopted here, in which replication system support should be generic and not make such assumptions about data and clients. However, because of the assumptions made, some of the application systems can use quite sophisticated replication techniques which achieve high availability and performance.

9.3.1 Distributed file systems

File systems provide one of the most fundamental system services in any computing environment. Arguably, files are the most common structure used to share and store information among both applications and users in distributed systems. This critical dependence on file services has led to a number of efforts to build distributed file services such as NFS [180], AFS [157], Coda [97, 157], Harp [112], Echo [181, 94] and xFS [6]. These systems simplify the sharing of files by providing uniform location and naming schemes. Of the systems mentioned, only the first version of AFS (AFS-1) and NFS do not directly exploit replication, although they make use of caching at the client side and thus depend on cache coherence protocols to maintain integrity of data. Distributed file systems are of interest because of their different, and occasionally extreme, approaches to consistency, availability and performance.

As most applications are written assuming a one-copy update model, distributed file systems attempt to provide a high degree of data consistency to clients. Echo is an example of a replicated file system which attempts to provide full replication consistency to the file system clients and uses a primary copy scheme to improve availability of servers. Clients of the replicated file servers contain Echo-specific code within *clerks* which intercept calls to the file system and perform the fail-over to a new primary if one fails. Echo employs redundant disks to store replicated files, and as an additional level of replication, primaries compete for election if they manage to claim ownership of a majority of disks. The replication scheme in Echo is thus fairly transparent to clients. If the filesystem is available to a client, the client will always observe correct and consistent files.

Full consistency is also achieved in the xFS system [6], a file system which is tailored for high-capacity, switched LANs such as ATM. The idea is that the high aggregate bandwidth provided by such networks can exceed the bandwidth of local disks, and thus invalidates the underlying assumptions of for example the Andrew file system policy of using local disks as caches. The most novel feature of xFS is its truly distributed design. By allowing all machines within the file system group to maintain files it can reduce the problem of server overload found among centralised file system designs such as Echo, NFS and Andrew. xFS assume that there is a high probability that the creator of a file is also the most frequent writer on the file. A file is managed by the machine on which the file was created, and later invalidation and write requests are passed to this machine. In this way xFS dynamically shares load among the machines cooperating in the system. However, any number of machines may hold copies of a file's data blocks, thus allowing fast access to the data for other machines as well. Replicas of the file are kept consistent by only allowing one writer to a particular data block at a time. Other machines must aquire ownership of the data block via the manager of the corresponding file. xFS exploits two techniques to achieve fault tolerance. Firstly, data blocks are striped across multiple disks, and enough redundant information is stored at each stripe group member to allow for single machine failures within the stripe group. Reconstruction of a new stripe group is performed automatically. Secondly, xFS is based on a transactional, log-based file structure which can be restored using roll-forward techniques in cooperation with clients.

Some of the systems make compromises with consistency to achieve better scalability and performance. For example, NFS uses periodic checks of timestamps between clients and the server to decrease the likelihood of update conflicts. AFS-2 goes a step further, due to higher scalability requirements cache coherence is only checked during *open* and *close* calls. Coda takes the most extreme approach to availability; in Coda clients are allowed to operate even if they cannot communicate with any of the servers. Conceptually, Coda defines two classes of replication; first-class and second-class replicated files, both optimistic [97]. The result is that there is a non-zero probability that clients which share files will observe inconsistent data. If the client is connected to a server, a first-class replication scheme is used among the servers to detect potential conflicts, this requires manual repair of unrecoverable conflicts [158]. If the client operates while disconnected, an optimistic second-class replication scheme exploits client caching which also may lead to conflicts. These conflicts are also repaired manually. The server-replication scheme used in Coda is a variety of the approach used in Ficus where any server can be sent an update and subsequently attempts to notify other servers about the update [144, 91].

NFS and AFS-2 assume that the probability of observing inconsistent data is small enough to be sacrificed

for the increased performance, although applications using NFS and AFS-2 should strictly be aware of the fact that files might be out of date. Coda acknowledges the fact that applications should be prepared for inconsistent files, and supplies special tools which will assist the user in reconciling conflicts that are not automatically repairable. Unix files have very simple structure, they are typically interpreted by convention from application to application. Therefore it is not possible to devise generic conflict resolution procedures for files. However, directories within file systems have a very limited operation set, typically *create* and *delete*. This makes automated conflict resolution possible for most independently executed directory operations in the Coda system, although conflicts within files must be reconciled manually by the users.

Similarly to most object-oriented databases (discussed in §9.3.2 p. 100), distributed file systems use caching at the clients to increase performance. Research has shown that there are a number of distinctive usage patterns within file systems which naturally lend themselves to caching [51]. For example, most files are used by only a single user, and if a file is shared, it is normally modified by only one user [6]. This observation reduces the probability of update conflicts, and will normally improve the performance of caching strategies. Also, read operations are much more common than write operations, which is the main justification for the extensive use of caching in some distributed file systems such as AFS and Coda. xFS is also based on this assumption, and exploits it also for load sharing. AFS and Coda clients store large volumes of cached files on local disks, and use a cache invalidation protocol to maintain cache coherence.

Caching in file systems can be compared with the data-shipping approach to replication in object systems (see §4.3 p. 42). This makes sense in file systems as files are embodied with little extra structure; clients access data in a file without going through a closely constrained operation interface. In contrast to the architecture proposed in this dissertation, distributed file systems exploit semantic knowledge about applications and data to optimise replication strategies. This is not possible here, where very little is assumed about applications and the data stored within objects. For example, building tools which would facilitate manual repair of conflicting objects within a system support layer seems impractical. Firstly, many objects will contain data which are not meaningful without significant application specific knowledge; tools which could easily be used to repair them would probably be as costly to implement as the application itself. Secondly, because these objects are not persistent, but rather very rapidly changing in response to method invocations; the required frequency of repairs might be far too great for manual intervention.

9.3.2 Database systems

Database systems manage persistent data, normally stored on disks; they do not normally consider operations on non-persistent data such as processes [19]. It is therefore possible for database systems to exploit other kinds of replication schemes than is possible in the proposed architecture. Transactions maintain integrity constraints on the data, explicitly separating application programs from data managers and assuming an inherent classification of data operations as either reads or replacing writes.

The encapsulation principle is an issue of debate in the object-oriented database community, and it is reasonable to assume that strict encapsulation will not normally be enforced [38]. Non-idempotent operations need not be too problematic in database systems; operations such as *deposit* and *withdraw* are commonly decomposed to reads and overwrites [51]. By exposing the data to the database, the data can be directly compared and overwritten, thus making it possible to use voting or coteric-based replication schemes, for example. Externally, queries and updates can use the encapsulated object interface[3].

Distributed databases are motivated by several factors, although the most important justifications are increased performance and support for autonomy [40, 8]. Additional complexities are introduced when database systems are built from a collection of existing databases; maintaining interoperability among potentially heterogeneous components and ensuring dynamic growth. Many distributed databases are

[3]Object-oriented databases such as Thor maintain the object encapsulation principle at the application level while transforming object invocations into simple read and writes at the data manager level [111].

very large — their size is commonly the reason for distributing them — and hence such systems tend to employ replication such as to achieve increased performance and autonomy for shared data. Synchronous replication protocols are therefore unsuitable; the need for autonomy dictates scalable, asynchronous protocols. Consequently, these systems favour loose synchronisation and weak consistency replication protocols at the added cost of reduced replication transparency.

Database systems are normally equipped with elaborate support for logging, checkpointing and grouping actions into atomic transactions. These facilities are normally not available in programming languages, and make it possible to use more sophisticated concurrency control for replicated data in database systems. In contrast to the system model adopted here, where objects can only be kept consistent if they receive the same sequence of method invocations, database systems can support the use of optimistic schemes discussed in §4.5 p. 49 and [55]. Additionally, some object-oriented database systems allow objects to be explicitly identified as either mutable or immutable depending on whether they can change state or not [3]. Immutable objects will never change, and access to those does not have to be serialised. This fact is exploited in the optimistic concurrency scheme used in Thor [3]. Timestamping combined with logging allows transactions to roll back upon discovery of conflicting updates. However, with optimistic concurrency schemes, there is a danger that other transactions have used the data already, perhaps even having committed. If the other transactions are not already committed they can simply be aborted, however, if they are already committed the crucial issue is whether or not it is possible to revert its effects; in many cases it is not. Aborts may appear in cascades, thereby incurring significant extra costs. However, optimistic schemes will tend to perform well in systems where conflicts are rare [55].

Most distributed relational database systems are built using a function-shipping approach. Queries and updates are sent from the client to a database server which executes the transaction. Later, the results are passed back to the client. The function-shipping paradigm can be exploited for process replication, such as in the hot standby approach [85]. The primary performs all the processing and a backup receives the log records from the primary and performing redo-actions continuously on these records thereby making sure that the recorded transactions are safely logged in case of fail-over.

Traditional client-server relational database systems, such as Sybase, Oracle and IBM's DB2 are also recognising replication as a means to increase performance or satisfying availability constraints [177], although they follow different routes. Using the log from transactions committed at the primary, Sybase System 10's Replication Server transfers these logs to replicas which have subscribed to the data. Replicas are 'backups' in the sense that all updates must be performed at the primary, only reads are allowed at the replicas. Although this approach might give reasonable performance, it raises the problem of maintaining causality relations. If a client reads data at a replica, and later, on the basis of this information, performs an update on the primary, it requires a very strict synchronisation of primary and replicas. Naturally, the primary will attempt to push the updates out to the replicas as fast as possible, but nevertheless, distributed systems are asynchronous and there will be a gap in time in which the replica is lagging behind the primary. Sybase addresses this problem by storing the logs in case communication with the replica fails, and although it is not explicitly stated, the replica will probably be denied permission to perform updates on the primary if it has pending logs at the primary [177]. Oracle's Symmetric Replication facilities also supports this push-model by registering, from the primary, asynchronous RPCs for subscribing replicas. Modifications in the primary trigger these RPCs which are executed at the replicas, following the function-shipping approach. However, Oracle also allows replicas to perform updates. Due to the potential for conflicts, subscribing clients can implement particular 'conflict-resolution' procedures which are automatically invoked by the Oracle database server upon detection of a conflict.

IBM's Copy Management and Oracle both support a pull-model of synchronisation. Clients are allowed to request a 'refresh' of replicated data (called 'snapshot' in Oracle's system), and the primary passes any updates on to the replica if necessary.

In contrast to relational databases, most object-oriented databases adopt a data-shipping computation model. Objects are shipped across the network and copied into the cache on client workstations where the objects can be manipulated. In contrast to the architecture proposed here, OODBs treat objects as passive (although complex) data structures which are passed between the persistent store at servers and the clients' caches. The data-shipping approach might therefore give better scalability as a result of

leaving more of the processing to clients.

The replication schemes used in OODBs are therefore not object replication schemes. The rationale for this approach is that the application normally resides on client workstations, and bringing the data closer to where it is processed increases the performance of the system [35, 3]. Most database applications interact very closely with the data, for example through object-graph navigation and query processing. However, caching, like replication, requires coherence and synchronisation protocols and introduces the same problems as replication schemes. Cache consistency can be achieved in several ways; [35] contains a survey of some common approaches and argues that adaptive callback locking schemes give the best performance.

9.3.3 Name services

Name services offer a fundamental and important function in distributed systems; they facilitate the sharing of named objects. A name service is essentially a database which stores *(name,reference)* tuples, and allows clients to perform both query and update operations on this information. The name introduces a level of indirection which makes it possible to assign a meaningful name to an object rather than a memory address. The name might be a human-readable text-string or any other identifier. Further, it has been noted that the performance of name services is critical to many applications; name lookups may constitute more than 40% of the system call overheads in UNIX according to [41]. Although many objects within a system will not be registered with a name service[4], the name service is crucial during bootstrapping[5] and to maintain references to important shared services. The large scale of some distributed systems motivates system-wide naming services which provide uniform access to objects anywhere within the system, thus forming potentially vast namespaces. These requirements introduce significant challenges for distributed name services.

Several designs have been proposed for reliable and scalable name services, for example Grapevine [21], The Clearinghouse [137], The Global Name Service [103], The Internet Domain Name System (DNS) [126, 127], the architecture of Cheriton and Mann [41] and CCITT's X.500 recommendation (although strictly a directory service[6]) [39] with suggested extensions for replication [90]. To address the problem of scale, the naming space is usually hierarchical to allow for autonomous administration and better locality of data. For example, in DNS, which is the system used for naming hosts in the Internet, the root-level entries denote top-level domains such as countries and large groups of institutions. The hierarchy is divided into zones which are the units of replication, and the names belonging to each zone are replicated at a minimum of two independent sites. A replication scheme classifies zones into two groups, primary and secondary servers. Primary servers fetch data directly from master files. Secondary servers download data from primaries, and periodically[7] query the primary for new updates.

Generally, replication is used extensively in name services to improve their availability and performance. Because name services have some rather distinctive characteristics, weakly consistent schemes are normally used. For example, it is normally appropriate to assume that the frequency of updates are much lower than the frequency of queries [39, 126]. The DNS architecture exploits this fact by caching the addresses of recently resolved names [126]. Cached data are non-authoritative and associated with timeouts. Clients may therefore suspect the cache to be stale if the resolved address is unusable or if the timeout has expired. Additionally, many name service designs assume that clients are able to tolerate temporary inconsistencies in the data by detection [137] and failure-masking using retries[8] [165]. The

[4] It is reasonable to assume that most objects are anonymous, i.e. they are not given explicit names and are only shared among a small number of objects using their direct references [58].

[5] Name services can be located at well-known PEs within the network [137] at the cost of more complicated reconfiguration, or clients may issue broadcasts to find a name service provider at the cost of more messages sent across the network [21].

[6] A directory service, in contrast to a name service, also contains more general information such as personal information about users.

[7] The frequency is defined using adjustable timeouts.

[8] The use of retries are not always sufficient to tolerate inconsistencies however, for example when the stored information

simple semantics of the data — names and references are simply read or overwritten — means that ordering constraints can be relaxed. For example, in [21] it is assumed that clients very rarely communicate directly with each other, and that clients therefore are more tolerant to different ordering of operations on the data. However, it is still desirable that replication schemes used in name services are convergent, so that the data stored will eventually become correct [21]. In consequence, due to their extreme requirements for scalability and rather modest consistency requirements, name services will normally exploit weakly consistent schemes with success.

9.4 Replication Support in Middleware

Middleware is commonly defined as a software layer that supports the development of interoperable, distributed applications [14, 106, 155]. Although this concept is rather vague, the term middleware is currently used to denote a vast range of software components. Services as diverse as RPC, object request brokers, transaction monitors, name services, configuration management services, communication systems and even distributed database systems have all been classified as middleware [14]. The main contribution of middleware components is a bridging function which allows programmers to develop distributed programs without too much concern for underlying heterogeneity. Many middleware components are not concerned with replication and are therefore not discussed. However, two distinct middleware components are concerned with replication; Lotus Notes and group communication systems. Although they have inherently different intentions, these two components are examples of software systems which can provide significant benefits to the development process of various classes of distributed applications.

CORBA is probably the most significant effort to date which attempts to define a framework for interoperable distributed processing, it does not currently specify any details about replication services. The need for replication has been recognised in CORBA, but no architecture for the actual implementation has yet been made available[50, 106, 198].

9.4.1 Lotus Notes

Lotus Notes is a relatively comprehensive application development environment which supports the manipulation, storage and distribution of documents [128, 14, 80]. It is a scalable system used in both small LANs and large corporate internetworks with heterogeneous network architectures. Lotus Notes is based on a client-server structure; servers act as document repositories while clients (personal workstations) retrieve and manipulate documents using a proprietary interface. Documents are comparable with text-files, although they may have a composite structure, e.g. containing attachments.

Lotus Notes uses replication extensively to achieve good scalability and for support of disconnected operation. As the central unit of data within Notes is the document, documents are also the units of replication. Release 4 of the system allows for replication of so-called *fields*, which correspond with the internal structuring of documents into subcomponents. Replication in Lotus Notes is flexible; it is customary to employ consultants to optimise the replication strategy for large installations, as it has a significant impact on the overall performance of the system. The flexibility is gained from the use of dedicated *replicator* processes which allow pairs of servers to exchange updates to the documents according to specified replication schedules. If either of the two databases has been modified, new and updated entries are pushed from the newly updated copy to the other. An increase in performance was the main motivation for reducing the granularity of replication from whole documents in Release 3 to fields in Release 4. The reduction in replication granularity also reduces the probability of update conflicts, which are detected automatically by Lotus Notes. Naturally, however, the reduced granularity increases the management overheads for each exchange session. Reconciliation is performed by arbitrating among conflicting updates, flagging the likely looser, and letting the client decide what action should be taken [14].

---

is indirect as may be the case for mailing-lists [134].

In contrast to the architecture proposed here, Lotus Notes takes an 'application-aware' approach to replication. A client must accept that shared documents might be updated simultaneously on another server. Naturally, this increases the complexity of the application, but because documents within Lotus Notes have such a simple structure, inconsistencies will normally be easily observed and eliminated by users of the system.

9.4.2 Group communication facilities

Group communication protocols have gained popularity as an approach to distributed computing because they can simplify the task of coordinating activities among a collection of active processes, called a group. A group communication protocol normally implements some sort of fault-tolerant multicast within a group of processes [89]. This approach to coordination is well suited to applications which consist of a relatively small number of cooperating processes, less than 20 say, where the processes are located in PEs connected to broadcast networks such as Ethernets or FDDI. The process group approach is also flexible, there is no requirement for processes to run the same code. Due to this flexibility, a process group system could be used to coordinate non-replicated activity, such as load-sharing. This is in contrast to the architecture proposed here, object replicas managed through the surrogate are identical.

Process group protocols ensure that any message delivered to a member is delivered to either all or none of the processes and therefore provides stronger delivery guarantees than the replication protocol proposed here. Examples of group communication protocols are the ISIS toolkit [18, 20] and the later Horus system [188] both developed at Cornell University, the Transis system [61] from the Hebrew University of Jerusalem and the Totem system [129] from the University of California at Santa Barbara.

Group communication protocols can be useful for the support of replication, although these protocols alone do not include all the facilities necessary to implement replication schemes. Generally, group communication protocols do not encapsulate plurality, clients of the group must be aware of the fact that they are using a group. Indeed, this lack of functionality was a motivation for the Gaggle [28] described in §9.1.2 p. 96, and process groups are also used as underlying technology in GARF to implement reliable message delivery among representatives (see §9.2.1 p. 98). For example the ISIS toolkit includes software tools for failure monitoring, an interface to support automated recovery of failed process group members and support for group reconfiguration in replication groups. The Transis system includes facilities which allow the programmer to merge data which have been updated in different partitions. These tools are used by the programmer to implement application specific procedures for replicated data management.

In ISIS, the programmer must design the processes specifically for replication by including statements for joining a particular group and sending update messages to other group members. This is in contrast to the architecture proposed here, which is tailored solely for replication and therefore can automate the use of replication. Furthermore, the proposed architecture can reduce complexities in the programming model and does not require any substantial modification to classes being replicated other than an extra stage of preprocessing to generate surrogates.

Clients of a replicated service need not be significantly complicated by the fact that it uses a replicated service however. Sending a message to the group is sufficient, although it is more efficient if the sending process is a group member [18]. Most applications are therefore likely to be structured to include most processes within the group. Scalability might therefore be a critical issue for these systems, and applications with a large number of clients, such as a multiuser editor or a distributed workgroup scheduler, might observe high performance overheads regardless of whether the clients are members of the group or not.

Additionally, process group protocols can guarantee consistency only if a majority of all the process group members are available. A problem of scalability is present in the proposed architecture as well, but by not requiring an intra-surrogate protocol, more surrogates can be accommodated, and surrogate failures do not normally reduce the availability of the replicas for non-failed surrogates. Only if surrogates fail while holding locks in the replicas can they affect availability of the system, and if so, locks are broken and another surrogate is left the choice of retrying the invocation. Because surrogates are coordinated

via the replicas only, using the locking protocol discussed in §6.2.4 p. 66, scalability in this architecture is mainly limited by replica lock contention and not by process group member failures as in ISIS. The architecture proposed here is therefore more geared towards efficient sharing among large numbers of clients, the number of clients does not influence the overheads of the replication scheme.

Some group communication systems support several message ordering policies to increase performance. For example, the ISIS toolkit implements two types of multicast; CBCAST (Causal Broadcast) and ABCAST (Causal Atomic Broadcast) [18]. The CBCAST primitive exploits application semantics to provide asynchronous multicast based on a notion of *virtual synchrony*. Programmers might use this primitive if they are sure that causal dependency is the only necessary relation among messages. In contrast, the ABCAST primitive is simpler, it ensures that all active processes within the group deliver messages in lockstep. ABCAST is therefore a potentially much more expensive primitive than CBCAST as ABCAST does not allow any asynchrony within the group. However, the performance is gained by sacrificing application complexity, as the programmer must show great care when deciding which primitive to use. Erroneous use of CBCAST could lead to misbehaviour in the program.

Commonly, group communication protocols have been tailored to non-partitioned operation, for example a call to a process group in ISIS will block if the partition contains less than a majority of the processes in the group. Recently, some systems have been tailored towards larger systems where partitions are common. Transis, allows partitioned (disconnected) operation [61]; however, this introduces a danger of conflicting updates, and Transis requires that programmers implement procedures for reconciling conflicting updates.

9.5 Summary

This section contains a summary of the various approaches that have been presented in this chapter. The chapter has shown that replication schemes in different system contexts are implemented to benefit from particular features of the surrounding system and to adhere to specific requirements set by the application. The most distinct factor is the tradeoff between consistency, performance and scalability. Generally, consistency requirements are sacrificed for many large scale systems such as name services and scalable distributed file systems, although the Echo file system [181] is a counter-example, providing even stronger consistency than NFS [180]. In contrast, system support for application programmers is normally based on full consistency models, for example Arjuna [171], the work of Cooper [49] and Mazouni et. al. [121].

Various approaches to programming language level support for replication have been presented. Due to the different assumptions underlying these, some differences to the approach proposed in this dissertation are evident. Cooper's approach, similar to the one proposed here, is based on the idea of building replication functionality into the usual RPC-stubs. The system model underlying Cooper's work is different, the use of persistent troupe members allows the use of an optimistic serialisation protocol. The programmer is also required to be more involved during the construction of replicas; modules are responsible for joining troupes. The application programmer is therefore given less freedom to specify the degree of replication compared to the proposed architecture. GARF implements a subset of the functionality described by Cooper; it implements a low-level invocation filtering mechanism which ensures that replicated objects coordinate invocations to avoid multiple executions of objects' methods. However, GARF is based on filtering at the invoker rather than filtering at the invokee as in Cooper's work. GARF can therefore map object replicas onto ISIS process groups and use this as the message delivery module.

The Gaggle is similar to the surrogate in the proposed architecture, but is not specifically designed for replication, it requires the implementation of additional functionality to achieve this. Fragmented objects is an approach advocating replicated objects, and is thus distinct from the proposed architecture in that the creator of the fragment is responsible for synchronisation and update propagation. However, the approach based on reflective programming, is similar in goals to the proposed architecture, and can, for example, be implemented using the reflective features of Open-C++.

Replication support in application components is fundamentally different from programming support for

replication. Whereas file systems, name services and databases make explicit assumptions about the data they manage, such assumptions cannot generally be made about objects that are replicated. The most extreme consequence of this is found in name service designs. Name services have very high availability requirements, and relatively weak consistency requirements. This, in addition to the very high read vs. write ratio, means that optimistic replication schemes can be used with great success.

Although files are manipulated in somewhat more complex patterns than the name bindings stored in name servers, distributed file systems have demonstrated that, with some assistance from the user when conflicts occur, optimistic replication schemes can be used to achieve very good scalability and performance.

In contrast, database systems rely on their sophisticated support for logging and transactions to employ optimistic concurrency schemes with success. Object-oriented database systems, depending more on caching than traditional replication techniques, show how important appropriate cache invalidation schemes are to achieve good performance.

The two middleware systems discussed, Lotus Notes and process groups, take near opposite directions to replication; Lotus Notes chooses to let the programmer take control of the consistency of the data. Lotus Notes will therefore be able to support system configurations of widely differing scale. By adjusting the scheduling of the replicator processes the propagation of changes in the data can be adapted to fit even large distributed systems interconnected with low-capacity networks. The process group approach does not normally sacrifice consistency, although recent systems such as Transis provide functionality to deal with disconnected operation, and might therefore be an example of a new trend within process group computing.

Chapter 10

Conclusions

This chapter concludes the dissertation with a summary of the main contributions and some directions for further work. The chapter is divided into four main sections; section 10.1 gives a summary of the main insights and results presented throughout the dissertation, section 10.2 presents the importance and implications of these contributions. Section 10.3 identifies important directions for further work within this area, and finally, section 10.4 compares the achievements with the thesis set forth at the start of the dissertation.

10.1 Summary of Contributions

This dissertation has argued that useful distributed systems must be constructed to withstand partial failures. One possible approach is to introduce redundant components and apply replication techniques to manage this redundancy. Chapter 4 presented a range of replication techniques which have been developed to mask failures and maintain consistency among replicated components. That chapter also argued that object replication is a technique which is especially beneficial in object systems. In contrast to other techniques, such as replicated objects, object replication reduces the effort required of the programmer to gain increased fault-tolerance. Various benefits and disadvantages were identified for the different techniques, the most fundamental tradeoff being that between consistency, transparency and scalability. In the adopted system model, based on program-level, fully encapsulated objects, some replication techniques were identified as inappropriate, for example those techniques which are based on overlapping replica groups and thus violate the encapsulation principle by assuming fully exposed object state.

Furthermore, the problem of increasing application complexity motivates the provision of system support, i.e. commonly available software components which can be used for several applications. A replication scheme based on object replication was identified to be most appropriate in this setting, as it attempts to minimise the changes needed when objects are replicated. Chapter 5 identified some of the problems which must be addressed to successfully realise system support, in particular system support for object replication where full transparency is not generally achievable.

Based on these observations, I have presented an architecture for system supported object replication. The architecture provides assistance to software developers constructing fault-tolerant object oriented programs, and the simple yet flexible programming model together with built-in support for object sharing adds only small complications for the programmer.

10.1.1 Programming model

The most distinctive feature of this architecture is the relatively simple programming model. Surrogates, which replace programmer defined objects, are manipulated very similarly to the original object. The

main difference is the addition of collators as parameters to methods which return results. Collators which encapsulate application specific reply processing are easily constructed and therefore add little to the complexity of using replication in object-oriented software.

Additionally, surrogates define a small set of specific exceptions which allow the programmer to handle surrogate failures. In summary, little added effort is required to extend an application to use the replication mechanism.

10.1.2 Object sharing

Object sharing is a natural consequence of the object model adopted; object references may be passed as arguments in method invocations and thereby allow multiple objects to share other objects. In programming languages the synchronisation problems incurred by object sharing are normally left to the programmer. However, this architecture directly supports object sharing among multiple concurrent objects by the inclusion of serialisation functionality.

Additionally, the architecture supports sharing among objects in multiple address spaces without reductions in fault-tolerance. This is achieved by allocating surrogate objects within each address space, thus increasing the failure resilience of surrogates. For multi-user applications which consist of many separate clients this is a significant benefit.

10.2 Discussion

Replication is only one among several approaches to increase computing system dependability. Other approaches, such as improved development methods, and n-version programming can also be useful to achieve this goal. N-version programming, in contrast to replication, is able to reduce the ill effects of software failures at the cost of developing several versions of the same software. Thus, n-version programming can not be used for transparent system support. The use of system supported replication is therefore a useful technique for increasing application dependability at a relatively small cost in application development overheads.

A reduction in application complexity is the main motivation for the proposed architecture, and this has been achieved using a full consistency programming model based on a strictly serialised concurrency scheme. Pursuing a full consistency paradigm is costly and has scalability limitations. The architecture is clearly unsuitable for very large distributed systems where other factors such as autonomy and loose synchronisation are more important than a simple programming model. The architecture presented here trades transparency and genericity for performance and scalability.

10.2.1 Architectural limitations

In its current form, without support for call coordination, the architecture enforces limitations on application partitioning. The separate object graphs described in §7.2 introduce new complexities for the application developer. Clearly, investigations into mechanisms to support call coordination are a natural issue for further work.

A small probability of inconsistency has been favoured rather than relying on more expensive and less scalable group communication protocols. Because the replicas themselves are not actively participating in forwarding requests to other replicas in the group, a surrogate which crashes during the replica update round might introduce inconsistency. Thus, the protocol does not implement the atomicity property in the presence of surrogate failures. However, this is only a problem when replicas are shared, and other surrogates sharing the replicas are informed about potential inconsistencies.

10.3 Future Work

This dissertation has identified a number of directions for further work within the area of system supported object replication. The rest of this section discusses these in more detail.

10.3.1 Coordinated calls

A very useful addition to the architecture would be a mechanism for handling replicated invocations to shared objects, i.e. many-to-one calls. The limitations on application partitioning could then be reduced (cf. §7.2 p. 79). However, solving this problem is non-trivial and may incur other limitations on program behaviour [59]. The problem occurs as a consequence of multiple replicas triggering redundant invocations on the same shared objects, and would require special method invocation protocols such that multiple identical invocations can be detected by the invoked object [48, 121]. When a replicated invocation is detected, the invocation must be executed only once by the object and the result from the method invocation should be copied and passed back to all of the invoking replicas.

10.3.2 Experiments with other transaction models

The proposed architecture is suboptimal for large scale systems, where synchronous updates are impracticable. Other, optimistic concurrency schemes could alleviate this problem. More specifically, long-duration transactions supporting shared locks would probably be more appropriate for application classes such as CAD, software engineering tools and CSCW [99]. However, automation of conflict resolution should be studied more carefully in such scenarios, as it is essentially the lack of semantic knowledge within the system support layer which makes this problematical.

Access to object persistence technology would increase the viability of such experiments. Most importantly, lower-level functionality for storing previously committed objects are necessary to implement optimistic concurrency schemes [55].

10.3.3 Higher level abstractions

A goal in distributed systems is to make them at least as reliable as a centralised system. When the system becomes large, it would be very inconvenient if a failure in a single machine implied a reduced level of service in the rest of the system (see page 27).

For an application developer it would be useful if the level of availability could be indicated by the application, and if a replication support facility was responsible for achieving this by using the necessary degree of replication and computation of (sub)optimal replica placement [114, 175]. The application programmer will in many cases be unable to make a good judgement of the placement of replicas, due to e.g. dynamically changing failure behaviour and object interdependencies [124]. However, making availability guarantees is difficult because the achieved level of availability depends on many factors such as the probability of failures, system load, external events like power outages and other environmentally caused failures. A step on the way to achieve this would be to relieve the programmer from having to indicate PE names during the initialisation of surrogates (cf. §7.4 p. 81). Rather, a pool of PEs could be maintained by system software which would perform the necessary analysis of the reliability of the PEs to achieve the required availability. A good placement of replicas would also ensure a relatively well shared load among the PEs in the network.

Real applications will have different requirements on the period of reliable operation. While some applications can safely be restarted occasionally, other applications might require continuous operation thus necessitating automated replica reconfiguration (cf. §6.3.2 p. 70). An important factor for such a scheme is the frequency of attempted reconfigurations. A high number of replicas results in higher survivability, but over-frequent reconfigurations might be very costly. This tradeoff should probably be determined by

the application programmer on the basis of the period of expected operation and the expected failure rate among the replicas.

10.3.4 Exception processing in collators

The current architecture does not support the processing of exceptions from replicas. Solving this problem could be done in two ways, both introducing some complexities to the presently simple collator programming model:

1. Add exception handling in the collator itself. By restructuring the surrogate, a collator could be made 'responsible' for triggering remote invocations, and thus be able to handle exceptions from these invocations directly. However, this approach would significantly complicate the programming model of the collator as the collator now would have to include code for each remote object type. Collator reuse would also be complicated.
2. Extend the collator interface with new add methods to notify the collator about exceptions. The main difficulty with this approach is the need to define a naming convention for matching the new methods with particular programmer defined exceptions. This approach also significantly complicates reuse, as different objects are likely to define independent exceptions.

10.3.5 Protocol verification

The architecture, and in particular the serialisation protocol, should undergo a formal verification process. The current protocols have only been subjected to informal reasoning and testing. Designing correct protocols is extremely complicated as the state-space is very large, and there might be special failure situations which trigger incorrect behaviour. A formal protocol verification process should be applied on a realistic implementation of the architecture.

10.4 Final Remarks

This dissertation has verified the thesis underlying this work; partial system support for object replication is feasible and such support assists the development of dependable distributed applications. The thesis was proved by demonstrating the usefulness of a prototype implementation of the proposed architecture.

Appendix A

Designing Collators

This appendix presents the sample code for implementation of reusable collators as it has been used within the prototype implemented in Modula-3. The presented code demonstrates the relatively simple programming necessary to construct collators.

A.1 A Specialised Collator

The code below implements the interface for an asynchronous collator for integer types. This particular collator is a specialisation of an abstract class `IntCollator.T` which defines template functions for the methods `init`, `prepare`, `add`, `addFailure` and `getResult`. The abstract class is presented in the next section.

```
(*
  File: IntFirstCollator.i3
  Documentation:
    A specialisation of IntCollator. This one returns on first
    result.
*)
INTERFACE IntFirstCollator;
IMPORT IntCollator;
TYPE
  T <: Public;
  Public = IntCollator.T OBJECT
  END;
END IntFirstCollator.
```

The code below is an implementation of a specialised asynchronous collator for integer types. Because the abstract class `IntCollator.T` implements the necessary functionality for the methods `init`, `add` and `getResult`, only `prepare` needs to be implemented here. The class `IntCollator.T` implements a collator which waits for all replies and returns a random reply. Therefore, only the number of wanted replies needs to be redefined through the `prepare` method.

```
(*
  File: IntFirstCollator.m3
  Documentation:
    A specialised version of the IntCollator. This one
    returns on the first reply.
*)
MODULE IntFirstCollator;
REVEAL
  T = Public BRANDED "IntFirstCollator.T" OBJECT
  OVERRIDES
    prepare := Prepare;  (* Only this method needs a new implementation *)
  END;

PROCEDURE Prepare(self : T; nReplicas : INTEGER) =
  (*
    Documentation:
      This method specialised the method in IntCollactor so
      as to return on the first result.
  *)
  BEGIN
    LOCK self.m DO
      self.nReplicas := nReplicas;
      self.nResultsWanted := 1;
    END;
  END Prepare;

BEGIN END IntFirstCollator.
```

A.2 A Basic Integer Collator

This section presents the code for a basic integer collator which could form the basis for a number of different specialised collators, such as `IntFirstCollator.T` presented in the previous section.

```
(*
  File: IntCollator.i3

  Documentation:
    A simple, generic, collator for integers which waits for all
    results and returns the first added.
*)
INTERFACE IntCollator;
IMPORT Thread, IntList;
EXCEPTION TooManyFailures;

TYPE
  T <: Public;
  Public = OBJECT
    m : MUTEX;
    collecting      : Thread.Condition;
    results         : IntList.T;
    nReplicas       : INTEGER;
    nResultsAdded   : INTEGER;
    nResultsWanted  : INTEGER;
    nFailures       : INTEGER;
  METHODS
    init() : T;
    prepare(nReplicas : INTEGER);
```

```
    add(e : INTEGER) : BOOLEAN;
    addFailure() : BOOLEAN;
    getResult() : INTEGER RAISES {TooManyFailures};
  END;
END IntCollator.


(*
  File: IntCollator.m3

  Documentation:
    Collects, manipulates and presents results from replicated
    invocations. This is a generic int-collator which waits for
    all the results and returns the first.
*)
MODULE IntCollator;
IMPORT IntList, Thread;

REVEAL
  T = Public BRANDED "IntCollator.T" OBJECT
  OVERRIDES
    init := Init;
    add := Add;
    addFailure := AddFailure;
    prepare := Prepare;
    getResult := GetResult;
  END;

PROCEDURE Init(self : T) : T =
  BEGIN
    self.m := NEW(MUTEX);
    self.collecting := NEW(Thread.Condition);
    self.results := NEW(IntList.T);
    self.nReplicas := 0;
    self.nResultsAdded := 0;
    self.nResultsWanted := 0;
    self.nFailures := 0;
    RETURN self;
  END Init;

PROCEDURE Prepare(self : T; nReplicas : INTEGER) =
  (*
    Documentation:
      Default is to wait for all the replies.
  *)
  BEGIN
    LOCK self.m DO
      self.nReplicas := nReplicas;
      self.nResultsWanted := self.nReplicas;
    END;
  END Prepare;

PROCEDURE Add(self : T; e : INTEGER) : BOOLEAN =
  (*
    Documentation:
      Adds e to the collection of results gathered from the replicas.
      Returns TRUE iff this was the last result needed.
  *)
  BEGIN
    IF self.nReplicas < 1 THEN  (* TRUE before prepare is called *)
```

```
      LOCK self.m DO
        WHILE self.nReplicas < 1 DO
          Thread.Wait(self.m, self.collecting);
        END;
      END; (* lock *)
    END;
    IF self.nResultsAdded = self.nResultsWanted THEN
      RETURN TRUE;
    ELSIF (self.nResultsAdded + self.nFailures + 1) >= self.nReplicas THEN
      (*
        If we can be sure that too many failures have happended
        already so that adding another result doesn't matter.
      *)
      RETURN TRUE;
    ELSE
      self.results := IntList.Cons(e, self.results);
      INC(self.nResultsAdded);
      RETURN TRUE;
    END;
  END Add;

PROCEDURE AddFailure(self : T) : BOOLEAN =
  (*
    Documentation:
      The parallel rpc reports failures to the collator. In case
      the number of failures is too high to allow for normal
      result processing, TRUE is returned.
  *)
  BEGIN
    INC(self.nFailures);
    IF self.nFailures + self.nResultsAdded >= self.nReplicas THEN
      RETURN TRUE;
    ELSE
      RETURN FALSE;
    END;
  END AddFailure;

PROCEDURE GetResult(self : T) : INTEGER RAISES {TooManyFailures} =
  (*
    Documentation:
      The procedure returns the processed result. The exception
      TooManyFailures is raised appropriately.
  *)
  BEGIN
    IF self.nResultsWanted > self.nResultsAdded THEN
      RAISE TooManyFailures;
    END;
    RETURN IntList.Nth(self.results, 0);
  END GetResult;

  BEGIN (* Module initialisation *) END IntCollator.
```

Appendix B

Probability Formalism

A small amount of probability calculation is used throughout this dissertation. This appendix presents the formalism used and contains a discussion of availability in majority locking schemes.

B.1 Probability

The probability that an event A will occur in a certain period is denoted $P(A)$ where $0 \leq P(A) \leq 1$. A value of zero means that the event never occurs, and a value of one means that the event certainly will occur. The probability $P(\neg A)$ that an event A will not occur is found by: $P(\neg A) = 1 - P(A)$.

If A and B are independent events, i.e. the occurrence of A does not affect the probability of B (and vice versa), then the probability of both events occurring is the product of their probabilities:

$$P(A \text{ and } B) = P(A) \cdot P(B) \tag{B.1}$$

The assumption of independent events is in some cases inappropriate (e.g. see §3.2.2 p. 32) and must be considered in each case.

If the probability $P(A)$ of event A per unit of time is much less than one and A is memoryless[1], then the mean time to event A is:

$$MT(A) \approx \frac{1}{P(A)} \tag{B.2}$$

If events A, B, C have mean time $MT(A)$, $MT(B)$, $MT(C)$ then the mean time to the first one of the three events $P(F)$ is (using equation B.2):

$$MT(F) \approx \frac{1}{\left(\frac{1}{MT(A)} + \frac{1}{MT(B)} + \frac{1}{MT(C)}\right)} \tag{B.3}$$

The *MTTF rating* for a component is the mean time to failure, i.e. the predicted time before the first failure. Using equation B.3 above; given n components A, statistically independent and with the same MTTF rating, $MTTF_A$, the mean time to the first failure event $MTTF_{first}$ is:

$$MTTF_{first} \approx \frac{MTTF_A}{n} \tag{B.4}$$

[1] An event A is memoryless if the event is just as likely to occur very shortly as it is to occur in a long time. Reliability models of computing equipment normally makes this assumption.

B.2 Availability of Majority Locking Schemes

Although majority locking may appear rather restrictive, it gives rather good availability in some configurations. The availability A_m of a majority locking scheme can be determined as follows. To be available, the scheme requires cooperation from at least $\lfloor \frac{n}{2} \rfloor + 1$ replicas, or that at most $\lceil \frac{n}{2} \rceil - 1$ replicas are failed (n_{failed} in the formula). If the replicas' failure modes are independent, the probability P of availability can be determined using a binomial distribution function. Summing this function over the range of valid numbers of failures gives the probability that the majority scheme is able to find enough replicas among the n replicas.

$$P = \sum_{i=0}^{n_{failed}} \binom{i}{n} p^i (1-p)^{n-i} \tag{B.5}$$

The formula assumes that each replica fails with a probability p, and that their failures are independent.

If calculated for a selection of p and n it becomes clear that the scheme does achieve relatively good availability for quite small values of n, even though the availability is poor for $n < 3$.

$n \setminus p$	0.01	0.05	0.1	0.2	0.3
1	0.99	0.95	0.9	0.8	0.7
2	0.9801	0.9025	0.81	0.64	0.49
3	0.999702	0.99275	0.972	0.896	0.784
4	0.999408	0.985981	0.9477	0.8192	0.6517
5	0.99999	0.998842	0.99144	0.94208	0.83692
10	≈ 1	0.999936	0.998365	0.967207	0.901191
15	≈ 1	≈ 1	0.999966	0.99576	0.937625
20	≈ 1	≈ 1	0.999993	0.999769	0.959723

Figure B.1: Failure resilience of majority voting scheme, calculated with six digits accuracy.

As an example of how this would affect the availability of a real system, consider the following scenario: A distributed group scheduling application running on a workstation depends on a non-replicated service on another workstation to function correctly, for example a mailservice. Assume the workstation running the mailservice is slightly unreliable, perhaps it is also used for software development or other computing intensive tasks and is therefore commonly overloaded, and that it usually runs for 10 days before crashing. This gives an approximate probability of failure on any given day of $\frac{1}{10} = 0.1$ (using eq. B.2).

Such high probability of failure might not be appropriate for such applications. By replicating the mail-service onto e.g. 5 workstations with the same reliability, and using a replication scheme based on majority locking for serialisation, the probability of failure (i.e. unavailability) would be reduced to $1 - 0.99144 \approx 0.009$. The application could now, again ignoring other failures, achieve an MTTF of approximately 111 days.

Bibliography

[1] Russell J. Abbott. Resourceful Systems for Fault Tolerance, Reliability and Safety. *ACM Computing Surveys*, 22(1):35–68, March 1990.

[2] Richard M. Adler. Distributed Coordination Models for Client/Server Computing. *IEEE Computer*, pages 14–22, April 1995.

[3] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronised Clocks. In SIGMOD95 [172], pages 23 34. SIGMOD RECORD 24(2).

[4] A. L. Ananda, B. H. Tay, and E. K. Koh. A Survey of Asynchronous Remote Procedure Calls. *ACM Operating Systems Review*, 26(2):92–109, April 1992.

[5] Birger Andersen, Carlos Baquero, and Rui Oliveira, editors. *Proceedings of ECOOP'95 Workshop on Mobility and Replication*, Aarhus, Denmark, August 1995.

[6] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. In SOSP15 [174], pages 109–126. This paper also appears in ACM TCoS 14(1):41–79, Feb. 1996.

[7] Anish Arora and Mohamed Gouda. Closure and Convergence: A Foundation of Fault-Tolerant Computing. *IEEE Transactions on Software Engineering*, 11:1015–1027, November 1993.

[8] M. Atkinson and A. England. Towards New Architectures for Distributed Autonomous Database Applications. In *Security and Persistence: Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, Bremen, West Germany, 1990.

[9] M. Atkinson and R. Morrison. Orthogonal Persistent Object Systems. *VLDB Journal*, 4(3):319–401, 1995.

[10] A. Avizienis. The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11:1491–1501, December 1985.

[11] Henry G. Baker, editor. *Lecture Notes in Computer Science (Vol. 986): Proceedings International Workshop on Memory Management IWMM 95*, Kinross, UK, September 1995. Springer-Verlag.

[12] Henry G. Baker. Preface. In *Lecture Notes in Computer Science (Vol. 986): Proceedings International Workshop on Memory Management IWMM 95* [11].

[13] Michael Barborak, Miroslaw Malek, and Anton Dahbura. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.

[14] Philip A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, February 1996.

[15] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In SOSP15 [174].

[16] Robert V. Binder. Software Quality and Object Orientation. *IEEE Computer*, 28(10):68–69, October 1995.

[17] Jan C. Bioch and Toshihide Ibaraki. Generating and Approximating Nondominated Coteries. *IEEE Transactions on Parallel and Distributed Systems*, 6(9):905–914, September 1995.

[18] Kenneth P. Birman. The Process Group Approach To Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[19] Kenneth P. Birman and Bradford B. Glade. Reliability Through Consistency. *IEEE Software*, pages 28–41, May 1995.

[20] Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, pages 47–76, February 1987.

[21] A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25:260–274, 1982.

[22] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network Objects. SRC Research Report 115, DEC, February 1994.

[23] A.D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[24] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed Garbage Collection for Network Objects. SRC Research Report 116, DEC, December 1993.

[25] Andrew D. Birrell. An Introduction to Programming with Threads. SRC Research Report 35, DEC, January 1989.

[26] Andrew P. Black. References; notes from the Store Working Group, Glasgow Research Festival 1995. Unpublished research note. Dept. of Computing Science, University of Glasgow.

[27] Andrew P. Black and Yeshayahu Artsy. Implementing Location Independent Invocation. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):107–119, January 1990.

[28] Andrew P. Black and Mark P. Immel. Encapsulating Plurality. In Guerraoui et al. [87], pages 57–79.

[29] Grady Booch. *Object-Oriented Design with Applications*. Benjamin Cummings, 1991.

[30] Borland International, Inc. *ObjectWindows for C++ Version 2.0: Reference Guide*. Borland International, Inc., 1993.

[31] Georges Brun-Cottan and Mesaac Makpangou. Adaptable Replicated Objects in Distributed Environments. Research Report 2593, Project SOR, INRIA, May 1995.

[32] Clemens H. Cap. Massive Parallelism with Workstation Clusters — Challenge or Nonsense? Technical Report IFI-TR 94.01, Department of Computer Science, University of Zurich, December 1993.

[33] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 Report (revised). SRC Research Report 52, DEC, November 1989.

[34] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[35] Michael J. Carey, Michael J. Franklin, and Markos Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *Proceedings of the 1994 SIGMOD*, Minneapolis, MN, May 1994.

[36] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[37] Thomas L. Casavant and Mukesh Singhal. *Readings in Distributed Computing Systems.* IEEE Computer Society Press, 1994.

[38] Roderic G.G. Cattell. *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, 1991.

[39] CCITT. *Recommendation X.500: The Directory - Overview of Concepts, Models and Service.* International Telecommunications Union, Place des Nations, 1211 Geneva, Switzerland, 1988.

[40] Stefano Ceri and Giuseppe Pelagatti. *Distributed Databases.* McGraw Hill, 1984.

[41] David R. Cheriton and Timothy P. Mann. Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183, May 1989.

[42] David R. Cheriton and Dale Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 44–57, 1993.

[43] K. G. Chin and Q. Zheng. FDDI-M: A Scheme to Double FDDI's Ability of Supporting Synchronous Traffic. *IEEE Transactions on Parallel and Distributed Systems*, 6(11):1125–1131, November 1995.

[44] Flavin Christian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2), 1991.

[45] Fan R. K. Chung. Reliable Software and Communication I: An Overview. *IEEE Journal on Selected Areas in Communications*, 12(1):23–32, January 1994.

[46] Brian A. Coan and Daniel Heyman. Reliable Software and Communication III: Congestion Control and Network Reliability. *IEEE Journal on Selected Areas in Communications*, 12(1):40–45, January 1994.

[47] Douglas E. Comer. *Principles, Protocols and Architecture*, volume 1 of *Internetworking with TCP/IP.* Prentice Hall, 1995.

[48] Eric C. Cooper. Replicated Distributed Programs. In *Proceedings of the Tenth ACM Symphosium on Operating Systems Principles*, pages 63–78, Orcas Island, Washington, U.S.A., December 1985. ACM Operating Systems Review 19(5).

[49] Eric C. Cooper. Replicated Procedure Call. *ACM Operating Systems Review*, 20(1):44–65, January 1986.

[50] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design Inc., and Sunsoft Inc. *The Common Object Request Broker: Architecture and Specification, revision 1.1.* OMG, 1991.

[51] George Couloris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design.* Morgan Kaufmann Publishers, second edition, 1994.

[52] M. Mira da Silva, M. P. Atkinson, and A. P. Black. Semantics for Parameter Passing in a Type-Complete Persistent RPC. In *Proceedings of the 16th International Conference Distributed Computing Systems*, Hong Kong, May 1996.

[53] John Daniels and Steve Cook. Strategies for Object Sharing in Distributed Systems. *Journal of Object Oriented Programming*, pages 27–36, January 1993.

[54] David Garlan and Robert Allen and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, pages 17–26, November 1995.

[55] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, 17(3):341–370, 1985.

[56] Wim De Pauw, Doug Kimelman, and John Vlissides. Modeling Object-Oriented Program Execution. In *Lecture Notes in Computer Science: Proceedings ECOOP'94 (Vol. 821): European Conference on Object Oriented Programming*. Springer-Verlag, 1994.

[57] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms For Replicated Database Maintainance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada, August 1987.

[58] Peter Dickman. *Distributed Object Management in a Non-Small Graph of Autonomous Networks with Few Failures*. PhD dissertation, Darwin College, University of Cambridge, 1991.

[59] Peter Dickman. Some Limitations for Operating Systems Support for Object Replication. Unpublished research note, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK, 1994.

[60] Peter Dickman, Mesaac Makpangue, and Marc Shapiro. Contrasting Fragmented Objects with Uniform Transparent Object References for Distributed Programming. In *SIGOPS 1992 European Workshop, on Models and Paradigms for Distributed Systems Structuring*, Project SOR, INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France, September 1992.

[61] Danny Dolev and Dalia Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[62] Kemal Efe and Venkatesh Krishnamoorthy. Optimal Scheduling of Compute-Intensive Tasks on a Network of Workstations. *IEEE Transactions on Parallel and Distributed Systems*, 6(6):668–673, June 1995.

[63] Hans Eriksson. MBONE: The Multicast Backbone. *Communications of the ACM*, 37(8):54–60, August 1994.

[64] P.D. Ezhilchelvan, I. Mitrani, and S.K. Shrivastava. An empirical study of the performance of distributed replicated systems. Technical Report Series 278, Computing Laboratory, University of Newcastle upon Tyne, January 1989.

[65] Jean-Charles Fabre, Vincent Nicomette, Tanguy Pérennou, Robert Stroud, and Zhixue Wu. Implementing Fault-Tolerant Applications Using Reflective Object-Oriented Programming. In *Predictably Dependable Computing Systems* [149], chapter III, pages 189–208.

[66] Tor E. Fægri. Investigation of distributed consensus algorithms. Senior honours project, Department of Computing Science, University of Glasgow, June 1994.

[67] Tor E. Fægri. Limitations for inconsistency in support layers for reliable distributed object systems. In Andersen et al. [5].

[68] Edward W. Felten and Dylan McNamee. Improving the Performance of Message-Passing Applications by Multithreading. In *Proceedings of the Scalable High Performance Computing Conference*, pages 84–89, April 1992.

[69] M. Fisher, N. Lynch, and M. Merritt. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[70] Robert Fowler. Architectural Convergence and The Granularity of Objects in Distributed Systems. In Guerraoui et al. [87], pages 33–46.

[71] Robert J. Fowler. The complexity of using forward addresses for decentralised object finding. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing (PODC'5)*, pages 108–120, August 1986.

[72] William B. Frakes and Christopher J. Fox. Sixteen Questions About Software Reuse. *Communications of the ACM*, 38(6):75–87, June 1995.

[73] Steven Fraser. Patterns: From Cult to Culture. In *OOPSLA'95 Proceedings Addendum: Symposia Summaries*, pages 85–88, 1995.

[74] Jeffrey Fritz. Video Connections. *Byte*, pages 113–116, May 1995.

[75] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[76] Hector Garcia-Molina and Daniel Barbara. How to Assign Votes in a Distributed System. *Journal of the Association for Computing Machinery*, 32(4):841–860, October 1985.

[77] Kurt Geihs, Birgitte Bär, and Arno Puder. Toward Open Service Environments. In Guerraoui et al. [88], pages 153–163.

[78] Kurt Geihs, Reinhard Heite, and Ulf H. Hollberg. Protected Object References in Heterogeneous Distributed Systems. *IEEE Transactions on Computers*, 42(7):809–815, July 1993.

[79] David K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the Seventh ACM Symphosium on Operating Systems Principles*, pages 150–162, December 1979.

[80] Steve Gillmor. Notes 4.0: Now It's Webware. *Byte*, 21(4):133–136, April 1996.

[81] Richard A. Golding and Darrell D. E. Long. The Performance of Weak-Consistency Replication Protocols. Technical Report UCSC-CRL-92-30, Concurrent Systems Laboratory, Computer and Information Sciences, University of California, Santa Cruz, July 1992.

[82] James Gosling, Bill Joy, and Guy Steele. Java language specification version 1.0. Available from the Java WWW server at http://java.sun.com:80/doc/language-specification.html.

[83] Yvon Gourhant. An Object-Oriented Approach for Replication Management. In Pâris and Garcia-Molina [139], pages 74–77.

[84] Jim Gray. The cost of messages. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 1–7, Toronto,Ontario,Canada, August 1988. ACM, ACM Press.

[85] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[86] Irene Greif, Robert Seliger, and William Weihl. A Case Study of CES: A Distributed Collaborative Editing System Implemented In Argus. *IEEE Transactions on Software Engineering*, 18(9):827–839, September 1992.

[87] Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors. *Lecture Notes in Computer Science (Vol. 791): Proceedings ECOOP'93 Workshop on Object Based Distributed Programming*. Kaiserslautern, Germany, July 1993. Springer-Verlag.

[88] Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors. *Lecture Notes in Computer Science (Vol. 938): Proceedings International Workshop on Theory and Practice in Distributed Systems*, Dagstuhl Castle, Germany, September 1994. Springer-Verlag.

[89] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems* [130], chapter 5, pages 97–146.

[90] S. E. Hardcastle-Kille. Replication and Distributed Operations extensions to provide an Internet Directory using X.500. Network Working Group, Request For Comments: RFC 1276, November 1991.

[91] J. S. Heidemann, T. W. Page, R. G. Guy, and G. J. Popek. Primarily Disconnected Operation: Experiences with Ficus. In Pâris and Garcia-Molina [139], pages 2–5.

[92] Maurice Herlihy. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[93] Stefan G. Hild. Disconnected Operation for Wireless Nodes. In Andersen et al. [5].

[94] Andy Hisgen, Andrew Birrell, Charles Jerian, Timothy Mann, and Garret Swart. Some Consequences of Excess Load on the Echo Replicated File System. In Pâris and Garcia-Molina [139], pages 92–95.

[95] Jun ichiro Itoh, Yasuhiko Yokote, and Mario Tokoro. SCONE: Using Concurrent Objects for Low-level Operating System Programming. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 385–398, Austin, Texas, USA, October 1995. Published in ACM SIGPLAN Notices 30(10).

[96] JavaSoft Inc. The java language: An overview. Available from the Java WWW server at http://java.sun.com:80/doc/Overviews/java/.

[97] J.J.Kistler and M.Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), 1992.

[98] Alan Joch. How Software Doesn't Work. *Byte*, 20(12):48–58, December 1995.

[99] Gail E. Kaiser, Wenwey Hseush, Steven S. Popovich, and Shyhtsun F. Wu. Multiple Concurrency Control Policies in an Object-Oriented Programming System. In *Research Directions in Concurrent Object-Oriented Programming*, chapter 7, pages 195–210. MIT Press, 1993.

[100] Frank Kappe. A Scalable Architecture for Maintaining Referential Integrity in Distributed Information Systems. Authors email address: fkappe@iicm.tu-graz.ac.at.

[101] Thilo Kielmann. Object-Oriented Distributed Programming with Objective Linda. In *Proceedings First International Workshop on High Speed Networks and Open Distributed Platforms*, St. Petersburg, Russia, June 1995.

[102] Rivka Ladin, Murray S. Mazer, and Alec Wolman. Replicating the Procedure Call Abstraction. In Pâris and Garcia-Molina [139], pages 86–89.

[103] B. W. Lampson. Designing a Global Name Service. In *Proceedings of the Fifth ACM Annual Symposium on Principles of Distributed Computing*, pages 1–10, Calgary, Canada, 1986.

[104] Jean-Claude Laprie. Dependability — Its Attributes, Impairments and Means. In *Predictably Dependable Computing Systems* [149], chapter I, pages 3–18.

[105] Jean-Claude Laprie and Karama Kanoun. X-Ware Reliability and Availability Modeling. *IEEE Transactions on Software Engineering*, 18(2):130–147, February 1992.

[106] Ted G. Lewis. Where Is Client/Server Software Headed? *IEEE Computer*, pages 49–55, April 1995.

[107] Jochen Liedke. On μ-Kernel Construction. In SOSP15 [174].

[108] Mengjou Lin, Jenwei Hsieh, David H. C. Du, Joseph P. Thomas, and James A. MacDonald. Distributed Network Computing Over Local ATM Networks. *IEEE Journal on Selected Areas in Communications*, 13(4):733–748, May 1995.

[109] David S. Linthicum. Network Management in the Distributed Enterprise. *Open Computing*, 12(9), September 1995.

[110] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, 1983.

[111] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In M.Tamer Özsu, Umeshwar Dayal and Patrick Valduries, editor, *Distributed Object Management*, pages 79–91. Morgan Kaufmann, 1994.

[112] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Liuba Shrira. Efficient Recovery in Harp. In Pâris and Garcia-Molina [139], pages 104–106.

[113] Mark C. Little and Santosh K. Shrivastava. Object Replication in Arjuna. BROADCAST Project Deliverable Report, Vol. 2, October 1994. Available from Department of Computing Science, University of Newcastle, Newcastle upon Tyne, NE1 7RU UK.

[114] M.C. Little and D.L. McCue. The Replica Management System: a Scheme for Flexible and Dynamic Replication. In *Proceedings of the Second Workshop on Configurable Distributed Systems*, Pittsburg, March 1994.

[115] Roy Longbottom. *Computer System Reliability.* John Wiley & Sons, 1980.

[116] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Narzul Le, and Marc Shapiro. Structuring Distributed Applications as Fragmented Objects. Research Report 1404, INRIA,France, January 1991.

[117] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented Objects for Distributed Abstractions. In *Readings in Distributed Computing Systems* [37], chapter 4, pages 170–186.

[118] Frank Manola. Interoperability Issues in Large-Scale Distributed Object Systems. *ACM Computing Surveys*, 27(2):268–270, June 1995.

[119] Bruce E. Martin, Claus H. Pedersen, and James Bedford-Roberts. An Object-Based Taxanomy for Distributed Computing Systems. In *Readings in Distributed Computing Systems* [37], chapter 4, pages 152–169.

[120] Margaret Martonosi, Anoop Gupta, and Thomas E. Anderson. Tuning Memory Performance of Sequential and Parallel Programs. *IEEE Computer*, 28(4):32–40, April 1995.

[121] Karim R. Mazouni, Benoit Garbinato, and Rachid Guerraoui. Filtering Duplicated Invocations Using Symmetric Proxies. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 118–126, Lund, Sweeden, August 1995.

[122] Derek McAuley. Private communications. Dept. of Computing Science, University of Glasgow.

[123] R. McConnell. The European Meta Computing Utilising Integrated Broadband Communications (E=MC2) Project. In Bob Hertzberger and Guiseppe Serazzi, editors, *Lecture Notes in Computer Science (Vol. 919): Proceedings International Conference and Exhibition on High-Performance Computing and Networking*, pages 54–59, Milan, Italy, May 1995. Springer-Verlag.

[124] D. L. McCue and M. C. Little. Computing Replica Placement in Distributed Systems. In Pâris and Garcia-Molina [139], pages 74–77.

[125] James G. Mitchell, Jonathan J. Gibbons, Graham Hamilton, Peter B. Kessler, Yousef A. Khalidi, Panos Kougiouris, Peter W. Madany, Michael N. Nelson, Michal L. Powell, and Sanjay R. Radia. An overview of the spring system. Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View Ca 94043. The document is available from the Sun WWW server at http://www.sun.com/.

[126] P. Mockapetris. Domain Names — Concepts and Facilities. Network Working Group, Request For Comments: RFC 1034, November 1987.

[127] P. Mockapetris. Domain Names — Implementation and Specification. Network Working Group, Request For Comments: RFC 1035, November 1987.

[128] Kenneth More. The Lotus Notes Storage System. In SIGMOD95 [172], pages 427–428. SIGMOD RECORD 24(2).

[129] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulis. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.

[130] Sape Mullender. *Distributed Systems*. ACM Press Frontier. Addison-Wesley, second edition, 1993.

[131] Sape Mullender. Kernel Support for Distributed Systems. In *Distributed Systems* [130], chapter 15, pages 385–409.

[132] Sape J. Mullender. Interprocess Communication. In *Distributed Systems* [130], chapter 9, pages 217–250.

[133] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert var Renesse, and Hans van Staveren. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, pages 44–53, May 1990.

[134] Roger M. Needham. Names. In *Distributed Systems* [130], chapter 12, pages 315–327.

[135] John R. Nicol, C. Thomas Wilkes, and Frank A. Manola. Object Orientation in Heterogeneous Distributed Computing Systems. *IEEE Computer*, pages 57–67, June 1993.

[136] Oscar Nierstrasz and Theo Dirk Meijler. Research Directions in Software Composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[137] Derek C. Oppen and Yogen K. Dalal. The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment. *ACM Transactions on Office Information Systems*, 1(3), July 1983.

[138] M.Tamer Özsu, Umeshwar Dayal, and Patrick Valduries. *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.

[139] Jehan-François Pâris and Hector Garcia-Molina, editors. *Proceedings of the Second Workshop on Management of Replicated Data*, Monterey, California, November 1992. IEEE Computer Society Press.

[140] Craig Partridge. *Gigabit Networking*. Addison-Wesley, 1994.

[141] Michael Perloff and Kurt Reiss. Improvements to TCP Performance in High-Speed ATM Networks. *Communications of the ACM*, pages 90–100, February 1995.

[142] Evaggelia Pitoura, Omran Bukhres, and Ahmed Elmagarmid. Object Orientation in Multidatabase Systems. *ACM Computing Surveys*, 27(2):141–196, June 1995.

[143] David Plainfossé and Marc Shapiro. A Survey of Distributed Garbage Collection Techniques. In Baker [11], pages 211–249.

[144] Gerald J. Popek, Richard G. Guy, Thomas W. Page Jr, and John S. Heidemann. Replication in Ficus Distributed File Systems. In Luis-Felipe Cabrera and Jehan-François Pâris, editors, *Proceedings of the Workshop on Management of Replicated Data*, pages 5–10, Houston, November 1990. IEEE Computer Society Press.

[145] David Powell. Distributed Fault Tolerance: Lessons from Delta-4. *IEEE Micro*, pages 36–47, February 1994.

[146] *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*. ACM Press, August 1995.

[147] Larry Press. Resources for Networks in Less-Industrialized Nations. *IEEE Computer*, 28(6):66–71, June 1995.

[148] C. Pu, J. Noe, and A. Proudfoot. Regeneration of Replicated Objects: A Technique and its Eden Implementation. *IEEE Transactions on Software Engineering*, 14(7):936–945, July 1988.

[149] B. Randell, Jean-Claude Laprie, H. Kopetz, and B. Littlewood. *Predictably Dependable Computing Systems*. ESPRIT Basic Research Series. Springer Verlag, 1995.

[150] Andrew L. Reibman and Malathi Veeraraghavan. Reliability Modelling: An Overview for System Designers. *IEEE Computer*, pages 49–57, April 1991.

[151] Andy Reinhardt. Your Next Mainframe. *Byte*, 20(5):48–58, May 1995.

[152] R.M.Needham and A.J.Herbert. *The Cambridge Distributed Computing System*. Addison-Wesley, Reading, Mass., 1982.

[153] Marshall T. Rose. *The Open Book: A Practical Perspective on OSI*. Prentice-Hall, 1990.

[154] R.S.Chin and S.T.Chanson. Distributed, Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[155] John R. Rymer. The Muddle in the Middle. *Byte*, 21(4):67–70, April 1996.

[156] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[157] Mahadev Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, pages 9–21, May 1990.

[158] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[159] Mahadev Satyanarayanan and Ellen H. Siegel. Parallel Communication in a Large Distributed Environment. *IEEE Transactions on Computers*, 39(3):328–348, March 1990.

[160] Jeremy Schlosberg. Where is the elusive groupware payoff? *Open Computing*, 12(9), September 1995.

[161] Dough Schmith and Paul Stephenson. Experiences Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms. In Walter Olthoff, editor, *Lecture Notes in Computer Science (Vol. 952): ECOOP'95 - European Conference on Object Oriented Programming*, pages 399–423, Århus, Denmark, August 1995. Springer-Verlag.

[162] Fred B. Schneider. Replication Management using the State-Machine Approach. In *Distributed Systems* [130], chapter 7, pages 169–197.

[163] Fred B. Schneider. What Good are Models and What Models are Good? In *Distributed Systems* [130], chapter 2, pages 17–26.

[164] James A. Schnepf, David H. C. Du, E. Russel Ritenour, and Aaron J. Fahrmann. Building Future Medical Education Environments Over ATM Networks. *Communications of the ACM*, pages 55–69, February 1995.

[165] M. Schroeder, A. Birrell, and R. Needham. Experience with Grapevine: The Growth of a Distributed System. *ACM Transactions On Computer Systems*, pages 3–23, February 1984.

[166] Michael D. Schroeder. A State-of-the-Art Distributed System: Computing with BOB. In *Distributed Systems* [130], chapter 1, pages 1–16.

[167] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computer Systems*, pages 198–204, 1986.

[168] Oliver Sharp. The Grand Challenges. *Byte*, 20(2):65 72, February 1995.

[169] Mukesh Singhal Niranjan G. Shivarati. *Advanced Concepts in Operating Systems: Distributed, database, and multiprocessor operating systems.* McGraw Hill, 1994.

[170] Santosh K. Shrivastava. Lessons Learned from Building and Using the Arjuna Distributed Programming System. In Guerraoui et al. [88], pages 17–32.

[171] Santosh K. Shrivastava and Daniel L. McCue. Structuring Fault-Tolerant Object Systems for Modularity in a Distributed Environment. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):421–432, April 1994.

[172] *Proceedings of the 1995 ACM SIGMOD: International Conference on the Management of Data*, San Jose, California, May 1995. ACM SIGMOD. SIGMOD RECORD 24(2).

[173] Ian Sommerville. *Software Engineering.* Addison Wesley, fourth edition, 1992.

[174] *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, December 1995. ACM SIGOPS.

[175] Mirjana Spasojevic and Piotr Berman. Voting as the Optimal Static Pessimistic Scheme for Managing Replicated Data. *IEEE Transactions on Parallel and Distributed Systems*, pages 64–73, January 1994.

[176] K. B. Sriram. A study of the reliability of hosts on the Internet. M.Sc. thesis, University of California Santa Cruz, June 1993.

[177] Doug Stacey. Replication: DB2, Oracle, or Sybase? *SIGMOD RECORD*, 24(4):95–101, December 1995.

[178] Bjarne Steensgaard and Eric Jul. Object and Native Code Thread Mobility Among Heterogeneous Computers. In SOSP15 [174].

[179] R. J. Stroud and Z. Wu. Using metaobject protocols to implement atomic data types. In *Lecture Notes in Computer Science (Vol. 952): Proceedings ECOOP'95 9th European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995. Springer Verlag.

[180] Sun Microsystems, ONC Technologies, 2250 Garcia Avenue, Mountain View, CA 94043 USA. *NFS: Network File System Version 3 Protocol Specification*, June 1993.

[181] Garret Swart, Andrew Birrell, Andy Hisgen, and Timothy Mann. Availability in the Echo File System. Technical Report 112, Digital Systems Research Center, September 1993.

[182] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, second edition, 1989.

[183] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[184] Gerald Tel. *Introduction To Distributed Algorithms.* Cambridge University Press, 1994.

[185] Dave Thomas. Experiences on The Road to Object Utopia. Keynote address, ECOOP'95, August 1995, Aarhus, Denmark.

[186] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, pages 8–17, June 1992.

[187] Anneliese van Mayrhauser and A. Marie Vans. Program Comprehension During Software Maintainance and Evolution. *IEEE Computer*, 28(8):44–55, August 1995.

[188] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[189] Thomas P. Vayda. Lessons From the Battlefild. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Austin, Texas, USA, October 1995. Published in ACM SIGPLAN Notices 30(10).

[190] Ronald J. Vetter. ATM Concepts, Architectures, and Protocols. *Communications of the ACM*, pages 31–38, February 1995.

[191] Jeffrey M. Voas and Keith W. Miller. Software Testability: The New Verification. *IEEE Software*, 12(3), May 1995.

[192] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In SOSP15 [174].

[193] John von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*. Princeton University Press, 1956.

[194] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report TR-94-29, Sun Microsystems Laboraties, Inc., 2550 Garcia Avenue, Mountain View Ca 94043, November 1994. Available from http://www.sun.com/.

[195] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In Baker [11], pages 1–116.

[196] Alan Wood. Predicting Client/Server Availability. *IEEE Computer*, 28(4):41–48, April 1995.

[197] J. Xu, B. Randell, C. M. F. Rubira-Calsavara, and R. J. Stroud. Software Fault Tolerance: Towards an Object-Oriented Approach. Technical Report Series 498, Dept. of Computing Science, University of Newcastle upon Tyne, Department of Computing Science, University of Newcastle, Newcastle upon Tyne, NE1 7RU UK, 1994.

[198] Zhonghua Yang and Keith Duddy. CORBA: A Platform for Distributed Object Computing. *ACM Operating Systems Review*, 30(2):4 31, April 1996.

GLASGOW UNIVERSITY LIBRARY