
R²PC: Fault-Tolerance Made Easy

Oscar Manso
Submitted to Dublin City University
for the degree of
Doctor of Philosophy (Computer Applications)

This research was carried out under the supervision of
Dr. David Sinclair (Computer Applications)

Dublin City University

This thesis is based on the candidate's own work

August 1999

Declaration

I hereby certify that the material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy (Computer Applications) is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: _____

Oscar Manso

Date: _____

29/9/22

Date: _____

Acknowledgements

I want to express my most deep respect, appreciation and gratitude to Dr. David Sinclair for his constant guidance and support throughout the period of my research.

I would also like to extend my gratitude to the institution of Dublin City University as a whole and to the many individuals that make it such a special place, from the dean to the gardeners that take care of the lovely park that I have used so often as the more direct escape route from the computer room. I would specially like to mention Prof. Michael Ryan and the School of Computer Applications in general for providing me with the opportunity and the necessary funding to undertake my research.

The list of particular names to thank for their collaboration and camaraderie would be too long to include in here, but I would want to mention the names of the friends that proof-read my thesis: Brian Parsons, Carmel Ryan, William Woods, Kevin Hopkins and David Escala.

Thank you all.

Table of Contents

INTRODUCTION.....	1
1.1 MOTIVATION.....	1
1.2 FAULT-TOLERANCE CONCEPTS.....	2
1.2.1 <i>General Concepts</i>	2
1.2.2 <i>Fault-tolerant Computer Concepts</i>	6
1.3 REQUIREMENTS FOR FAULT-TOLERANT SOFTWARE SYSTEM SUPPORT.....	10
1.3.1 <i>Message Communication</i>	11
1.3.2 <i>Remote Procedure Call</i>	14
1.3.3 <i>Co-ordination Languages</i>	16
1.4 NEW MODELS FOR THE CONSTRUCTION OF FAULT-TOLERANT SOFTWARE.....	18
1.5 OUTLINE OF THE DISSERTATION.....	20
RELATED WORK.....	21
2.1 RELIABLE REMOTE PROCEDURE CALLS.....	21
2.1.1 <i>Introduction</i>	21
2.1.2 <i>Troupes</i>	22
2.1.3 <i>Clusters</i>	23
2.1.4 <i>Contexts</i>	24
2.1.5 <i>Hot Replication</i>	25
2.1.6 <i>Piranha</i>	26
2.1.7 <i>Phoenix</i>	27
2.1.8 <i>Conclusions</i>	28
2.2 FAULT-TOLERANT LINDA SYSTEMS.....	30
2.2.1 <i>Introduction</i>	30
2.2.2 <i>Reliable Tuple Space</i>	30
2.2.3 <i>Redefining the Semantics of Linda</i>	32
2.2.4 <i>Conclusions</i>	35
DEFINITION OF A NEW RELIABLE RPC PROTOCOL.....	36
3.1 INTRODUCTION.....	36
3.1.1 <i>Group Addressing</i>	37
3.1.2 <i>Execution Semantics</i>	37
3.2 R ² PC PROTOCOL.....	39

3.2.1	<i>System Overview</i>	39
3.2.2	<i>Nested Calls</i>	41
3.2.3	<i>Fault Recovery</i>	44
3.2.4	<i>Implementation Details</i>	47
3.3	GROUP COMMUNICATION.....	50
3.3.1	<i>Introduction</i>	50
3.3.2	<i>Group Structure</i>	50
3.3.3	<i>Group Identification</i>	51
3.3.4	<i>Intergroup Communication</i>	53
3.3.5	<i>Intragroup Communication</i>	55
3.3.6	<i>Reliability</i>	57
3.3.7	<i>Implementation Details</i>	59
3.4	CONCLUSIONS.....	61
EXAMPLE APPLICATION		63
4.1	INTRODUCTION.....	63
4.2	PCLINDA.....	64
4.2.1	<i>Overview of the System</i>	64
4.2.2	<i>Implementation of PCLinda</i>	65
4.3	R ² PCLINDA.....	67
4.3.1	<i>Group Addressing</i>	67
4.3.2	<i>Determinism</i>	68
4.4	CONCLUSIONS.....	71
SYSTEM EVALUATION		72
5.1	INTRODUCTION.....	72
5.2	EFFICIENCY.....	73
5.2.1	<i>Transmission Cost of R²PC calls</i>	73
5.2.2	<i>Inherent Parallelism of R²PCLinda</i>	75
5.2.3	<i>Parallelism with Communication</i>	78
5.2.4	<i>Cost of Replicas in R²PCLinda</i>	84
5.3	RELIABILITY.....	86
5.3.1	<i>Crash Faults</i>	86
5.3.2	<i>Omission Faults</i>	89
5.3.3	<i>Omission-Crash Faults</i>	90
5.4	EVALUATION OF RESULTS.....	93
5.5	CONCLUSIONS.....	94
CONCLUSION		95

6.1	SUMMARY	95
6.2	FUTURE RESEARCH	96
6.3	CONCLUDING REMARKS	97
APPENDIX A – SYSTEM CONFIGURATION		99
APPENDIX B – SOURCE CODE FOR THE TESTS		103
B.1	R ² PC APPLICATION TEST	103
B.2	EVALUATION OF PI	106
B.3	EVALUATION OF PRIME NUMBERS	108
APPENDIX C – RESULTS		112
C.1	TRANSMISSION COSTS OF R ² PC vs. RPC	112
C.2	EVALUATION OF PI	113
C.3	EVALUATION OF PRIME NUMBERS	115
C.4	PRIME NUMBERS EVALUATION FOR ONE WORKER PROCESS	116
C.5	EVALUATION OF CRASH FAULTS	116
C.6	EVALUATION OF OMISSION FAULTS	117
C.7	EVALUATION OF OMISSION/CRASH FAULTS	118
BIBLIOGRAPHY		120
INDEX		124

List of Figures

Figure 1 Possible set of components of a railway crossing system.....	3
Figure 2 The provision of fault-tolerance implies a cost.....	5
Figure 3 Invocation of a Remote Procedure Call.....	15
Figure 4 Example of Linda operations used to access a shared variable.....	31
Figure 5 Programming the bag-of-tasks paradigm in FT-Linda.....	34
Figure 6 Overview of the R ² PC Protocol.....	41
Figure 7 Possible command sequence generated by nested machine states.....	42
Figure 8 Possible deterministic command sequence generated by a client (the vertical axis represents time).....	44
Figure 9 Information to store for recovery purposes.....	46
Figure 10 Representation of the Group Object.....	48
Figure 11 Refinement of the Group Object.....	49
Figure 12 Scheme representing the use of the Directory Service to send a message from Group k to Group m. ...	54
Figure 13 Thread model used in the R ² PC processes.....	60
Figure 14 Overview of the PCLinda System.....	64
Figure 15 Representation of the Tuple Space Object.....	65
Figure 16 Refinement of the Tuple Space Object.....	66
Figure 17 Response times for the transmission of R ² PC and RPC calls.....	74
Figure 18 Response times for the transmission of RPC calls.....	74
Figure 19 Pi equals the area under the curve $4/(1+x^2)$	75
Figure 20 Evaluation of Pi for Unreliable Evaluation vs. Evaluation with Resilience 1 to 3.....	76
Figure 21 Evaluation of Pi for Unreliable Evaluation vs. Evaluation with Resilience 6, 8 and Total.....	76
Figure 22 Initial Solution Prime Numbers. Code to execute by eval processes.....	79
Figure 23 Initial Solution for Prime Numbers. Main program.....	79
Figure 24 Assigning the tasks to be done by the eval processes.....	80
Figure 25 Evaluation of Prime Numbers for Unreliable Evaluation vs. Evaluation for Resilience 1 and 2.....	82
Figure 26 Unreliable Evaluation of Prime Numbers.....	82
Figure 27 Evaluation of Prime Numbers using only one worker process.....	85
Figure 28 Evaluation of the Prime Numbers problem for one worker in the presence of Crash Faults.....	88
Figure 29 Percentage of System Failures in the presence of Crash Faults.....	88
Figure 30 Evaluation of the Prime Numbers problem for one worker in the presence of Omission Faults.....	89
Figure 31 Evaluation of the Prime Numbers problem for one worker in the presence of Crash/Omission Faults.....	92
Figure 32 Percentage of System Failures in the presence of Crash and Omission Faults.....	92

Abstract

R²PC: Fault-tolerance Made Easy

Fault-tolerance is a concept that is becoming more and more important as computers are increasingly being used in application areas such as process control, air-traffic control and communication systems. However, the construction of fault-tolerant software remains a very difficult task, as it requires extensive knowledge and experience on the part of the designers of the system.

The basics of the Remote Procedure Call (RPC) protocol and its many variants are a fundamental mechanism that provides the adequate level of abstraction for the construction of distributed applications and release the programmers from the burden of dealing with low level networking protocols. However, the standard definition of the protocol does not provide us with semantics that are sufficiently transparent to deal with unexpected hardware and software faults, i.e. the programmer has to deal with possible problems that may occur.

To deal with this problem, different reliable variations of the RPC protocol have been defined. This dissertation introduces a new reliable protocol - R²PC - with the following characteristics.

- Symmetric treatment of client and server processes.
- Use of concurrently processed nested calls in stateful servers.
- The achievement of failure transparency at the application level.

Introduction

Introduction of Concepts and State of Art

1.1 Motivation

Fault-tolerance is a concept that is becoming more and more important as computers are increasingly used in application areas such as process control, air-traffic control and communication systems. However, the construction of fault-tolerant software remains a very difficult task inasmuch as it requires extensive knowledge and experience on the part of the designers of the system. This difficulty is to a large extent due to the lack of powerful generic tools and methods, which would enable the designer to express their algorithms at an abstract level and thus free them from having to concentrate on minor details. The traditional way of constructing fault-tolerant applications can be compared with the expression of algorithms using assembly code - the quantity of details that have to be taken into account makes the programmer's task enormously difficult.

Distributed systems, consisting of computers connected by a network, are an appropriate platform for the construction of fault-tolerant software.

The basics of the Remote Procedure Call (RPC) [Birrell et al. 84] protocol and its many variants (like Method Invocation [OMG 98] or Java RMI [<http://www.javasoft.com>]) are a fundamental mechanism for the construction of distributed applications by providing an appropriate level of abstraction that releases programmers from the burden of dealing with low level networking protocols.

For this reason, it would be logical to use the RPC mechanism for the construction of fault-tolerant software as well. However, the standard definition of the protocol does not provide us with sufficiently transparent semantics to deal with unexpected hardware and software faults [Coulouris et al. 88].

This paper introduces R²PC, a new Reliable RPC protocol, which provides an appropriate framework for the easy construction of fault-tolerant applications.

1.2 Fault-tolerance Concepts

In fault-tolerance terminology there are a number of terms that, when applied to computer systems, have traditionally been used in a very confusing way – due, no doubt, to the lack of consensus on what exactly fault-tolerance is. This lack of common understanding is an issue that has been tackled by different authors ([Cristian 91], [Heimerdinger et al. 92]).

Based on their work, this section introduces a number of basic definitions that will be used to establish a proper reference framework throughout the rest of the thesis.

1.2.1 General Concepts

A *system* is the entire set of components, both computer related and non-computer related, that provides a service to a user.

An example of a system could be a computer-controlled train level-crossing. The system would have to provide the users with safe traffic control for cars and trains at a certain junction.

A possible set of components of such a system would be the barrier itself, sensors to detect the cars and trains, the computer unit that controlled all the components and the operators in charge of the maintenance of the system (see Figure 1). It should be noted that the operators are included as part of the system, but the drivers of the vehicles - our users of the system - would not be.

A system is said to have a *failure* if the service it delivers to the user deviates from the system specification for any period of time.

A *fault* is the failure of a subsystem (a component of the system). A fault may or may not lead to other faults, or to a failure of the system.

A *symptom* is the observable effect of a fault at the system boundary.

A *fault-tolerant system* is a system that may continue providing its service in the presence of faults.

Let us return to our example and try to establish a better understanding of the distinctions between the different concepts involved.

Suppose that a bit in the computer's memory becomes stuck - that is a failure of the memory and a fault of the computer system.

If the memory fault affects the operation of the program to the extent that the computer system believes that there is no train coming when, in actual fact, a train is about to reach the railway crossing, what we have is a computer system failure and a fault in the overall train barrier system.

If the computer system's output indicates that there is no train, that is a symptom of the computer system's fault.

If the operator of the system becomes aware of the problem and lowers the barrier in time, the system did not fail, it tolerated the fault.

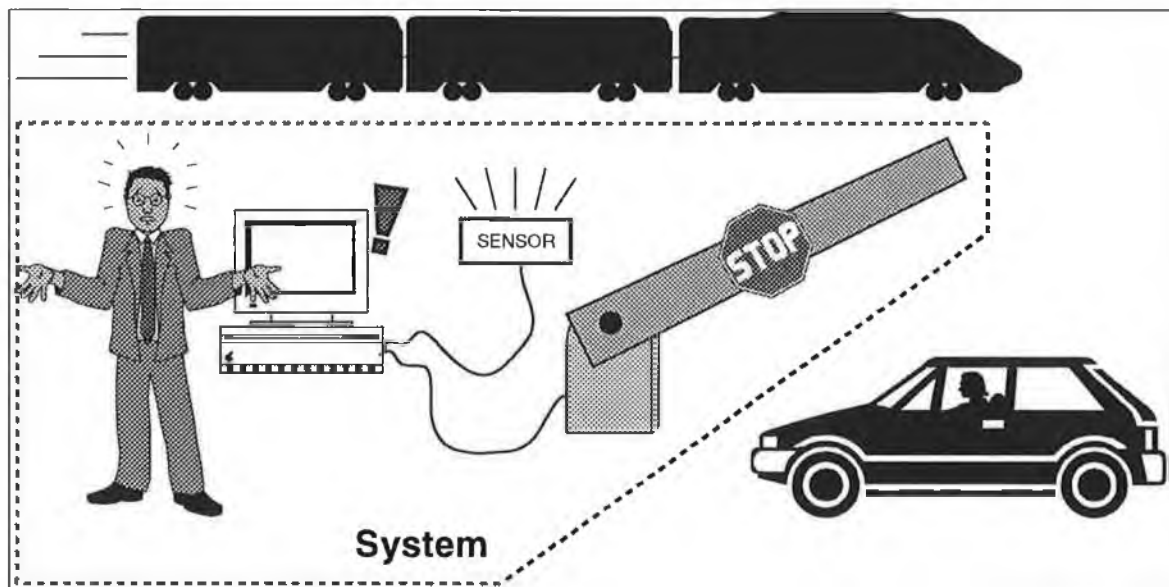


Figure 1 Possible set of components of a railway crossing system

A fault is *observable* if information about its existence is available at the system boundary.

A fault is *detected* if the fault-tolerance mechanism of a system finds it. Otherwise it is *latent*, regardless of whether it is observable or not.

Now, imagine that instead of getting a bit of memory stuck, one of the sensors had failed resulting in the same indication from the computer system. Could we say that the computer system had failed? It depends on the specification given to the computer system. If the computer system had been supposed to detect the problem and solve it by itself, we could then say that, in this case, there had been a failure of the computer system. But, if the computer system was only supposed to inform the operator about the problem, the computer system would have worked in accordance with its specifications. In fact, the computer system could be said to have been working in accordance with its specifications even in the case where the memory got stuck! An example of this type of case would be a situation where the task of the computer system was to communicate any possible problem to the operator, leaving him/her the ultimate responsibility for the proper administration of the barrier.

Therefore, as we can see, it is very important to specify precisely what the system is supposed to do.

The logical or physical blocks that make up a system are called **components**. Each component can also be made up of sub-components.

An **atomic** component is a component that is not divisible or one that we choose not to divide into more sub-components.

Theoretically, any component of the system is liable to fail. However, each component of the system is supposed to accomplish certain specifications. Therefore, for any system, there is a level beyond which the faults are not worth considering. This level is called the **fault-floor** of the system.

The **system boundary** is the set of components that make up the system. Failures occur when faults reach the system boundary.

The **span of concern** is the area that lies between the fault-floor of the system and the system boundary. The span of concern is the area within which faults need to be taken into consideration.

Fault-tolerance can only be achieved through the use of redundancy. *Redundancy* is the provision of functional capabilities which have been designed to deal with possible faults - in other words, capabilities that would be unnecessary in a fault-free environment. The use of redundancy obviously implies a cost. However, it is important to note that the use of redundancy by itself does not solve the problem of fault-tolerance in a system, it just helps to detect the problems.

In our example, we could improve the fault-tolerance of the system by adding extra sensors and including parity bits in our memory chips to help us in the detection of possible faults. In this case, the addition of a signalling system to regulate the passage of the trains would improve substantially the safety of the system (see Figure 2). The design of such a signalling system should ensure that a train is not allowed to go unless the barrier is down.

Unfortunately, as we have said, the use of redundancy does imply a cost. It is natural that a user, not being aware of these extra costs, would demand a completely fault-tolerant system. However, protecting a system from every conceivable fault can be extremely expensive in terms of money, space and time. It is ultimately the responsibility of the users and the builders of the system to strike a balance between the requirements of the system and its cost. Whenever possible, it is better to concentrate on likely faults and ignore the less likely ones – unless, of course, they can be dealt without any additional cost.

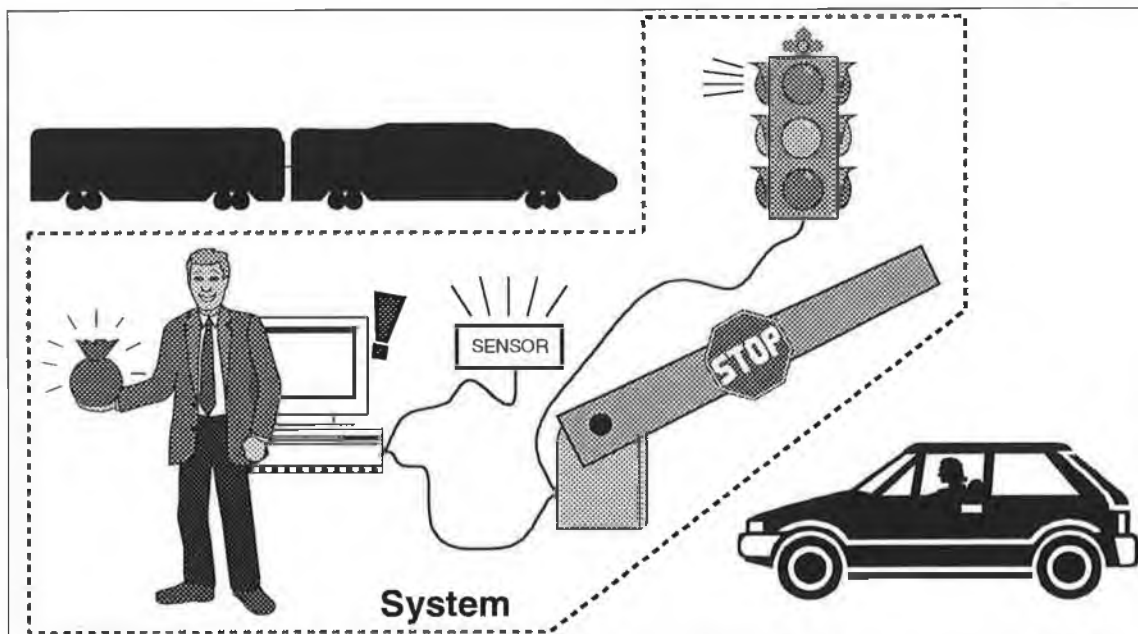


Figure 2 The provision of fault-tolerance implies a cost

1.2.2 Fault-tolerant Computer Concepts

The previous section dealt with general concepts used for any fault-tolerant system. Digital computer systems have special characteristics that determine how systems fail and what fault-tolerance mechanisms are appropriate. This section introduces concepts that are particular for computer systems.

In computer systems, high-availability is a term commonly used when referring to fault-tolerance [Resnick 96].

A *highly available system* is a system designed and implemented with sufficient redundancy in its components to provide its essential services while masking any failure or fault of computer nodes or software objects.

Depending on the degree of transparency in which the replacement of a component occurs, we can have different types of highly available systems:

- **Manual masking:** Following a component fault, some manual action is required to put the redundant component into service, during which time the system is unavailable for use. In actual fact, systems of this type are not normally considered to be highly available.
- **Cold/Warm Standby:** Following a component fault, users of the component are disconnected and lose any work in progress (i.e. they roll back to the last consistent, committed state of their work). An automatic fault detection and recovery mechanism then detects the fault and brings into service the redundant component. Once this is done, users are able to reconnect to it and begin processing again from the point of their rollback.

Recovery time is application dependent but it usually involves cleaning up file systems, databases and persistent resources, all of which can take a considerable amount of time.

The more time a system requires for recovery the colder its “temperature” is.

This type of mechanism is typical of transactional systems that distinguish between two entities: the transactions (or processes) that execute operations and the data accessed by the transactions. The data exists independently of the transactions and the consistency of

the system is defined solely in terms of the state of this external, persistent database. Normally, transactions communicate through the use of this external database.

The System V [Cheriton 88] and Arjuna [Shrivastava 94] are two examples of fault-tolerant services that belong to this category.

- **Hot Standby:** Following a component fault, users of the components are not disconnected and are not aware of the fault whatsoever.

Recovery times in these systems are of very short duration. In fact, the concept of recovery time does not really apply since from the client's perspective there is no recovery.

This type of mechanism is typical for message based applications that manage data as part of the state of a process and define consistency in terms of the joint states of the processes that belong to the system. Normally this model corresponds to the control/communication applications where it is unacceptable to wait indefinitely for a system component to recover.

The sufficient degree of redundancy required is normally expressed in terms of the resilience degree of the system. The *resilience degree* of a highly available system is the maximum number of simultaneous node failures that the system can tolerate without failing.

Computer systems are composed of multiple interrelated components interrelated. Due to the quantity and variety of faults that may arise during the execution of such systems, faulty systems are normally grouped together into fault classes. [Powell 95] describes a very general model to classify different types of computer faults. Here we present a taxonomy that describes the most common system fault-classes used in the literature starting from the most unreliable fault-classes [Barborak et al. 93].

We understand by *Byzantine Fault* any possible fault in the system. Systems in this class [Lamport et al. 82], can produce any type of errors at any time, arbitrarily or maliciously.

In the *Authenticated Byzantine Fault* class we have systems with the same characteristics as in the previous class, but in this case every message can be verified to detect whether it contains erroneous information (arbitrary or malicious).

The systems that belong to the *Incorrect Computation Fault* class produce incorrect results, but they are always produced on time.

The systems that belong to the *Arbitrary Timing Fault* class do not produce incorrect value results, however, the results are delivered outside their specified time constraints (either earlier or later).

The *Omission Fault* class describes systems in which the results are either produced on time or not at all.

The *Crash Fault* class describes the systems in which, once a certain number of results have been omitted, we have the guarantee that any further results will be omitted as well. This class is also known as *Fail-Silent* class.

The systems that belong to the *Fail-Stop* class alert to the other processors when they cease operation.

Finally, the systems that belong to the *No Fault* class do not produce any type of errors.

Theoretically, a fault-tolerant service with a sufficient level of reliability can be achieved in any system independently of the fault-floor that we adopt, but the cost of algorithms and number of necessary components will increase as the fault-floor lowers. For example, if we take the *Byzantine* model, [Dolev et al. 1985] shows that if we have n processors, t of which are faulty, we will need at least $O(nt)$ messages to reach agreement. However, if we consider *Authenticated Byzantine* systems the algorithm becomes much simpler and we will only need $O(n + t^2)$ messages.

In practice, many existing fault-tolerant systems are implemented using the *Fail-Silent* or the *Fail-Stop* model as fault-floor [Schlichting et al. 83]. This is normally considered a sufficiently reliable model for current systems. However, if a more reliable model is needed, [Schneider 84] shows how to implement *fail-stop* processors using *byzantine* systems. Unfortunately, with a finite amount of hardware resources we can only implement an approximation to a completely fault-tolerant system. For this reason, the paper describes an approximation to an *n-fail-stop* processor, a system that behaves as a *fail-stop* processor unless $n+1$ failures occur within its components. The result is that to implement an *n-fail-stop* processor, $2n+1$ processes are required.

As well as dealing with failures at the software execution level we should also deal with failures at the hardware level. A *replaceable hardware component* is a physical component that can be

removed from the system without affecting other hardware components. Ideally, it should be possible to remove a replaceable hardware component from the system (either because of a failure or for preventive maintenance) without affecting the activity of the software components running on the system. As this is often too expensive or impossible to achieve, the next best approach is to ensure that the failure induced into the software components by the replacement of any hardware component has “nice” semantics, such as *crash* or *omission*. In this way, the software level will be able to detect the hardware fault and deal with it appropriately before it propagates outside the boundaries of the system.

Depending on the granularity of the replaceable hardware components we can distinguish between coarse grain architectures and fine grain architectures. The replaceable components of fine grain architectures will be the fundamental hardware components of the system itself. In the coarse grain architectures, the replaceable components will package together different basic hardware components, such as CPU, I/O and memory. At the hardware level, it is important to replicate every component. Furthermore, fault-tolerant hardware systems have to ensure that each component is connected to the rest via more than one disjoint path. Examples of this type of hardware architectures can be found in [Siewiorek 90], [Laprie et al. 90] and [Cristian 91].

Another major source of problems is the occurrence of errors in the software process, resulting in systems that do not behave as expected. The only solution for this type of problem is the use of methodologies, formal verification, and proper fault validation by means of the use of complete tests. In recent times, this issue has become increasingly relevant and different standards are being created to recommend practices for the development of safety critical systems [CEI/IEC 97], [ISO/IEC 98]. An alternative approach to dealing with this type of problem is the implementation of more than one variant of the function to be performed, or so called *program diversity* [Avizienis et al. 84]. In theory, truly diverse designs would eliminate dependencies on design methodologies, software implementations and tests, and this would help in the creation of software that would meet the requirements of the initial specifications.

This issue of errors in the software process takes on particular relevance in the context of our thesis because the creation of fault-tolerant applications is a very complex problem and one of the main focus of this thesis is to simplify the creation of this type of applications.

From now on, when we use the term *fault-tolerant software system* (or *fault-tolerant application*) it will be understood to mean a software system designed and implemented to

provide its essential services while masking the presence of up to r – the resilience degree – simultaneous *crash* failures and providing *hot-standby* transparency.

There is a natural link between the creation of fault-tolerant systems and the idea of constructing distributed applications. On one hand, the potentially large number of machines in a distributed system makes the probability of at least one component failing large, which could lead to the unavailability of the service. On the other hand, distributed systems possess inherent potential for fault-tolerance because the failure of some number of machines can be masked by the components that are available. Although the probability of at least one machine failing can be high, the probability of all the machines failing can be extremely small.

A ***distributed system*** is a system that divides its work and data among different machines.

A fault-tolerant system needs to divide its work among different machines so as to ensure that the failure of any machine in the system does not cause the failure of the fault-tolerant system.

The designer of fault-tolerant software has to deal with many difficult issues such as: fault-detection, distributed consistency, synchronisation and order of messages.

However, traditionally, distributed systems have not been designed with the idea of supporting the development of fault-tolerant applications in mind. Consequently, developers of this type of application have had no assistance in dealing with all these issues.

The next section studies and describes different approaches to supporting the development of distributed systems and their requirements to develop fault-tolerant software.

1.3 Requirements for Fault-Tolerant Software System Support

Fault-tolerant systems are usually modelled as client/server applications [Cristian 91].

A ***computer service*** specifies a collection of operations whose execution can be triggered by users of the service. The execution of the operations may result in outputs and/or service state changes.

A ***server*** implements a service without exposing the internal service state representation and operation implementation details to clients.

A *client* is a user of the service. The accuracy of a client depends on the accuracy of the servers that it uses.

A server may use services from other servers. In other words, a server may behave as a client and as a server.

Different abstract programming mechanisms have been proposed and used to represent the client/server model. In the next section we present the most common ones arranged according to their level of abstraction.

1.3.1 Message Communication

Introduction

In the Message Passing model, processes communicate through the exchange of messages.

In a traditional message passing mechanism (e.g. [Berkeley 86]), the developer of the application decides what type of information is being sent, to which process, if a reply should be awaited, and what should be done if anything goes wrong.

The main advantage of this model is its flexibility. As it is such a low-level construct, the programmer is in control of every communication aspect of the application and consequently, the application can be tuned very efficiently.

Fault-tolerance issues

The construction of a fault-tolerant application using a standard message passing mechanism requires a considerable amount of work as the programmer has to deal with all the fault-tolerant issues involved such as: fault-detection, recoverability, synchronisation, retransmissions, and serializability of messages.

However, the introduction of the *Group Communication* model used in the V distributed kernel [Cheriton 84], marked a first step towards releasing the programmer from many of the above problems.

A *process group* is a set of processes that share a set of common characteristics (internal state) interacting and co-ordinating with each other to provide a common external interface [L. Liang et al. 90].

Groups can be said to be either *open* or *closed*. In a ***closed group***, only its members can send messages to the group while, on the other hand, in an ***open group*** outsiders are permitted to forward messages to the group.

The structure of a group can be either *static* or *dynamic*. In a ***dynamic group***, processes can join or leave the group at runtime. A ***static group*** keeps the same members for its entire lifetime.

Communications between external clients and group members are called ***intergroup communications***, while, internal communications among group members are known as ***intragroup communications***.

Distributed applications can make use of the Group Communication model in different ways:

- **Distribution of workload:** Processor groups can be useful for load balancing, for example, distributing incoming requests among different processors.
- **Distribution of data:** In this case the state of the group is partitioned among its members. This could be the case for a distributed naming service such as the OSI directory service.
- **Replication of data:** Each member of the group holds a replica of the state of the group. This is particularly useful for the development of fault-tolerant applications.

Group communication can support the development of fault-tolerant systems through the use of the following features [L. Liang et al. 90]:

- **Communication transparency:** This feature is comprised of two related aspects:
 - **Atomic message delivery:** An atomic message is either received and processed by all the group members or by none at all. This feature makes it possible to get around the problems associated with a partial communication failure by converting it into a total failure.
 - **Application-level absolute ordering:** The sequence of messages processed by any member of a group has to change its state in accordance with the rest of the members.
- **Naming transparency:** Group members have to be transparently bound to a single group name. This service may be particularly complex to implement in dynamic groups because

the system has to ensure that all the group members have a consistent view of the current members of the group.

- **Failure transparency:** The failure of a group member can be notified to the application so that recovery actions can be taken at the application-level or, alternatively, it can be dealt with transparently. In this case, the system can either try to automatically select other members to take over the failed member's tasks or present the failure as a total group failure.

It has to be said, however, that some of these requirements may be in conflict with the features necessary for the development of applications that do not require fault-tolerance. Indeed, it could be conceived that an application that seeks to distribute its workload among its members using such strong communication semantics might find that the cost of the work-distribution is too expensive.

Several existing systems support different group communication semantics. For example, the V kernel's focus is on efficiency but its communication mechanism does not guarantee atomic order [Cheriton 88], while the Amoeba system does guarantee application-level ordering [Kaashoek 92] and Isis provides a set of primitives to the application level, each of which is equipped with different delivery semantics [Birman 93]. Consequently, the application programmers become tied to the semantics offered by the system they use.

A different approach is taken with the introduction of configurable group communication services (such as the case of Horus [van Renesse et al. 96] or Coyote [Bhatti et al. 98]). In this case, the semantics of the primitives used can be configured and changed at run-time, thus adding another level of flexibility to these systems.

Conclusions

The construction of fault-tolerant applications using traditional message passing mechanisms requires an enormous amount of work at programming level.

The introduction of the Group Communication mechanism has done much to facilitate the tasks of the programmer in terms of the development of fault-tolerant applications.

In the final analysis, however, the use of these low-level primitives alone cannot offer a completely transparent solution. In fact, many of these systems include, or are built with the idea of including a Reliable Remote Procedure Call mechanism.

1.3.2 Remote Procedure Call

Introduction

A *Remote Procedure Call (RPC)* is the execution of a procedure that resides in a foreign node. With the use of this mechanism, the programmer does not have to deal with the transmission or reception of messages. This mechanism is used in a very similar way to a normal procedure call but with the important difference that the procedure is executed in a foreign node.

This mechanism provides an adequate level of abstraction for the construction of distributed applications by virtue of the fact that it releases programmers from the burden of dealing with low level networking protocols.

Since its definition [Birrell et al. 84], this protocol has been very successful and adopted as a fundamental mechanism for the implementation of many distributed systems and standards for the development of distributed applications (see for example Method Invocation [OMG 98] or Java RMI [<http://www.javasoft.com>]).

Fault-tolerance issues

The standard definition of the protocol, unfortunately, does not provide us with sufficiently transparent semantics to deal with unexpected hardware and software faults. During the processing of a procedure call, one of the following events may occur (see Figure 3):

1. The request message may be lost.
2. The reply message may be lost.
3. The server process may crash.
4. The client process may crash.

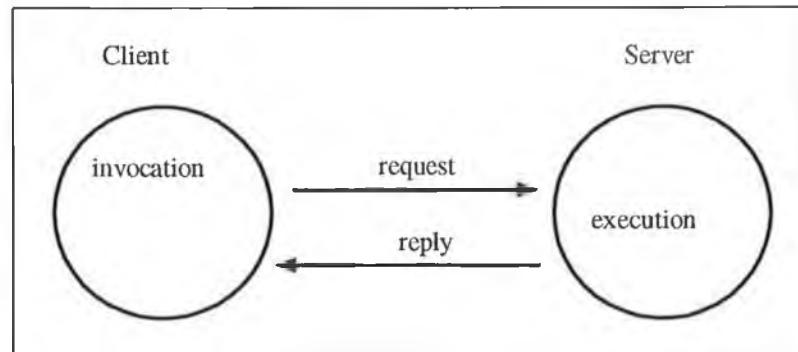


Figure 3 Invocation of a Remote Procedure Call

Traditional RPC packages provide error checking and primitive recovery mechanisms to support the client and server processes. Three basic types of execution semantics have been defined - *at least once*, *at most once*, and *exactly once* semantics [Coulouris et al. 88]. With ***at least once*** semantics, the client process makes repeated attempts to successfully execute the procedure until it either times out or receives an indication of a server failure. In this case, the procedure may have been executed many times before the client acknowledges a successful execution. With ***at most once*** semantics, the steps of RPC execution continue until an unrecoverable error occurs. The client and the server only transmit one message/procedure execution. If one of the steps fails, the client receives an exception and the remote procedure call aborts. The error may occur either before or after the actual procedure execution has occurred. Therefore, if the caller receives an error message, there is no guarantee that the procedure has not executed. If no errors occur, the procedure runs only once. ***Exactly once*** semantics guarantee that in case of failure, the remote procedure will not be executed. Otherwise, the procedure will be executed only once.

The definition and use of these semantics forces the programmer to be aware of a number of possible errors during the execution of the call. This extra responsibility makes programming of remote procedure calls a complex and cumbersome task. If we use *at least once* semantics, the client process has to repeat the call until it is successful, and once it is successful, the client has to take into account the number of times it has been executed (if it is using non-idempotent procedures). When errors appear using *at most once* semantics, the client process does not know whether the procedure has been executed or not. Finally, the "best" semantics to offer the programmer are *exactly once semantics*, because, in case of errors, at least the programmer can try to redirect the call to other servers without having to worry about whether the call was already processed or not.

However, unfortunately, these semantics are very rarely offered to the programmer because in a networked environment they are very difficult to guarantee since errors may occur at any stage of the process.

Conclusions

As we have seen, this situation is far from perfect. The ideal would be to define a mechanism that could recover from errors in a transparent way so that the user of the service would not have to be aware of possible communication problems.

1.3.3 Co-ordination Languages

Introduction

A coordination language provides operations to create computational activities and to support communication among them. Coordination languages hide the direct link between the client and the server of a distributed application. Due to its simplicity and power, we are going to focus on Linda, which is a coordination model for parallel programming developed at Yale University [Gelenster 85].

In Linda, processes exchange information through the use of an abstract communication environment called tuple space. The ***Tuple Space*** (TS) is a content-addressable shared memory system. Processes exchange information in the form of tuples. A ***tuple*** is a series of typed values, for example:

("a string", 10, 4.5)

A tuple in TS is equally accessible to all processes but is bound to none. The communicating processes are uncoupled in space and time (tuples are persistent objects). Another interesting feature of Linda is that processes are created in the form of "live" tuples that are inserted in the tuple space. The "live" tuple carries out some specific computation and then turns into an ordinary data object tuple.

Linda defines the following primitives:

- out(Tuple)

This operation inserts a new tuple in the tuple space.

- rd(Template)

A template specifies a tuple to search. A template can describe the search for a particular tuple by specifying the values of each of the typed fields, or it can search for a tuple within a group by just writing a collection of some of the values and specifying the type of the rest of the fields to search. These types are specified in the form of variables (formals) that are going to be assigned with the values (actuals) of the tuple found in tuple space.

For example, $rd(?a, 1, ?b)$ would search for a tuple which in its second field had the value 1 and the other two fields held an integer value.

- $in(Template)$

The same as before but in this case, when the tuple is found, it is removed from the tuple space.

- $eval(Tuple)$

In this case, at least one of the fields of the tuple has to specify the name of a function to be executed. This sentence creates a “live” tuple that is executed independently of the process that create it. “Live” tuples cannot be accessed until they turn into static tuples.

Fault-tolerance issues

The Linda model facilitates the programming of fault-tolerant applications due to the following characteristics:

- **Space Uncoupling:** The different processes of the system are address-space disjoint. The processes in the system do not need to know the location of the processes to which they are communicating. They communicate through the use of a common framework, the Tuple Space. This feature helps in the implementation of a fault-tolerant mechanism because it facilitates the restart and replication of processes in different nodes.
- **Time Uncoupling:** The co-operating Linda processes do not need to know anything about their relative speed of processing. This means that a process waiting for the result of a failed process will get suspended until the failed process is restarted.
- **Structured Naming:** A receive operation can normally be matched against several different tuples result. This means that, in case of a process failure, the system can still provide result tuples that match the search of the requesting processes without need to block them. Therefore, the system can still function when only partial data is available.

However, although these characteristics facilitate the implementation of a Fault-Tolerant system, they are not enough to guarantee the existence of a Fault-Tolerant system. Any implementation supporting fault-tolerant processes must ensure the following fault-tolerance conditions:

- **Correctness condition:** The result of a computation should be the same as the result of a failure-free execution.
- **Termination condition:** The computation will reach a final state in finite amount of time if at least one node is operational.

Nevertheless, the original definition of Linda does not consider what happens in abnormal conditions, for example, the failure of a “live” tuple.

Conclusions

The Linda model provides a very high level of abstraction that may be very appropriate for the creation of fault-tolerant applications. However, the subject has not been tackled in the original definition of the model. The effects of processor failures on the execution of Linda programs are not considered in the standard definitions of the language.

1.4 New models for the construction of fault-tolerant software

Our main objective in this thesis has been to assist application programmers and minimise their effort in the development of future highly available applications.

With this idea in mind, we have added fault-tolerant mechanisms to the existing higher-level model to construct distributed applications, minimising the semantic and syntactic changes necessary to achieve the requirements of fault-tolerance and transparency.

This thesis presents the design and implementation of R²PC, a new Reliable Remote Procedure Call (RRPC) protocol with the following characteristics:

- The protocol provides mechanisms to guarantee the recovery of client and server processes. Many RRPC systems do not even consider the option of restarting client processes (see [Beedubail et al. 95], [D. Liang et al. 98]). Probably this is due to the fact that many fault-tolerant systems have been designed with the only intention of providing a fault-tolerant service. In fact, the idea of restarting client processes does not seem to make much sense on a broad distributed system (for example, a server in Alabama cannot control whether a client in Dublin is working or not). However, this concept is important

in a restricted environment where there is a need to implement a fully reliable system (for example, a computer's LAN controlling a nuclear reactor). Such type of application may require reliability in all the components and not only in the servers. Client processes should also have the right to survive!

- The protocol permits the use of stateful servers that can act as clients for other servers (nested calls). This is a basic aspect for the creation of many complex distributed applications, for example, the definition of the OMA standard and particularly, one of its key components, CORBA, requests and enhances the use of nested RPC calls [Vinoski 97]. However, we know of very few systems that deal with nested RRPC calls (see [Jalote 89], [Yap et al. 88] and [Beedubail et al. 95]). And from those systems, none permits the concurrent service of different calls.
- The use of the reliable protocol is "fully" application transparent. The desirable property of a fully transparent RRPC mechanism would be that we could transport our traditional unreliable RPC (as defined in [Birrell et al. 84]) programs into this new system obtaining reliable applications automatically. However, such an automatic adaptation is not possible due to syntactic and semantic factors. One of the main aims of R²PC has been to create a fault-tolerant system semantically and syntactically as close to the original definition of the RPC mechanism as possible, facilitating the transition from an unreliable environment to a reliable one. We have been successful in doing so by introducing only two main changes in the original definition:
 - Syntactically: Traditional RPC mechanisms address a particular process in a specific node. If this node fails, our application will have to change the address to which the call is being directed. In order to have a reliably transparent mechanism, the R²PC mechanism has to identify entities independently from where are located.
 - Semantically: The provision of fault-tolerance will be guaranteed if the behaviour of each client process can be determined from the results obtained in the execution of the R²PC calls evaluated.

In fact, existing RRPC systems include stronger semantic preconditions on their reliable protocol such as the non-provision for client recovery ([Beedubail et al. 95],

[Yap et al. 88], [D. Liang et al. 98]), or the non-provision for nested calls ([Wood 93], [D. Liang et al. 98], [Maffeis 96]).

- The protocol facilitates an efficient implementation that provides concurrent service of different calls and minimises the number of messages exchanged through the use of multicast mechanisms.

As a testbed for our design, we have used R^2PC to implement $R^2PCLinda$, a new fault-tolerant Linda system that incorporates reliability to the model without requiring any change in its original syntax or semantics. As far as we know, this is the first existing implementation of Linda with such characteristics.

1.5 Outline of the Dissertation

The remainder of this dissertation is organised as follows. Chapter 2 describes related work in the areas of Reliable Remote Procedure Calls systems and fault-tolerant Linda systems. It compares the approaches that have been used to construct fault-tolerant applications at a medium and high level of abstraction and serves as a foundation to compare our approach to others.

Chapter 3 presents our approach to constructing fault-tolerant distributed applications. A new Reliable Remote Procedure Call system is introduced preserving the semantics of the original model and providing a framework for the development of transparent and efficient fault-tolerant systems.

Chapter 4 describes an example of the application of our approach. Using our system, we provide a fault-tolerant implementation of a Linda system.

Chapter 5 evaluates the efficiency and reliability of our resulting system. It presents an analysis of the system using different examples.

Chapter 6 summarises the dissertation and offers future research directions.

Related Work

High-Level Constructs for Fault-Tolerance

2.1 Reliable Remote Procedure Calls

2.1.1 Introduction

This chapter presents different existing Reliable Remote Procedure Call (RRPC) models in chronological order. The systems are described and compared by studying their efficiency and transparency semantics.

The following definitions are introduced to provide a consistent study of the systems to be compared:

We define that the processes of a system are *deterministic* if we can guarantee the following condition – from a given a state, the invocation of an operation on different copies of a process makes the same sequence of requests and the states of each process will be the same after the operation is completed.

This precondition is assumed by all of the fault-tolerant systems described.

Fault-tolerant systems use three different replication models:

- In the *active replication* approach requests are sent to and performed by all the replicated elements.
 - In the *passive replication* approach one copy of the replicated processes is declared as primary and the others as backups. All requests are sent to the primary which is responsible for providing the service, the backups are passive but they get updated information from the primary so that they are capable of assuming the role of the primary if this process fails. This mechanism is also known as the *primary-backup* approach.
-

- The *coordinator/cohort* approach combines the two previous methods by declaring a primary and backups but, in this case, the backups are active, internally updating its state with the information transferred from the primary. This allows for a faster recovery.

2.1.2 Troupes

[Cooper 85] describes the RPC mechanism implemented for the Circus system, one of the first Reliable RPC mechanisms created. The implementation of the system only uses intergroup communication (a process group is named *troupe* in the paper). Each client sends a 1-to-many call to each member of the server group, and each server replies to the client group with another 1-to-many call. There is no communication among the members of a group and each member works independently of the others. Inconsistencies can be detected at the client side and appropriate measures can be taken according to the voting policy adopted.

Application Transparency

Inconsistencies could also arise in the case of including client thread execution or using nested calls.

The system includes a commit protocol to detect any inconsistencies and transforms such attempts into a deadlock. Deadlock detection is then used to abort and retry one or more of the offending transactions.

However, the lack of a mechanism to identify nested calls globally among the system means that inconsistencies could remain undetected (see section 3.2.2). This means that the system can not deal with the use of nested calls.

The system provides support for the recovery of client and server processes. However, this mechanism has to be triggered at the application level on reception of errors from the primitives.

Efficiency

This protocol uses the active replication approach because each group member executes every call. The implementation of the system only uses point to point communication (the multicast mechanism was not available at the moment of its implementation). The protocol is quite expensive requiring $N*M$ point-to-point messages for the processing of each call (N and M being the number of clients and servers respectively). However, the introduction of a multicast mechanism would reduce the number of multicast messages transmitted to $N+M$.

Although the protocol supports the use of threads, the calls from different client groups are evaluated serially on the server side.

2.1.3 Clusters

[Yap et al. 88] presents a protocol based on the establishment of a hierarchy defined among the members of the group (named *clusters* in this case) to establish a linearly ordered sequence of calls to execute. The member with the highest rank is called the primary while the rest are the subordinates.

The primary client process sends a message to the primary server. The call is transferred from one subordinate of the group to the other until it reaches the last member, from this point an *ack* is transferred linearly back to the primary which then, executes the call and sends the result to the primary client. The chain of calls with the result is transferred among the client members until the primary client receives an *ack* from its immediate subordinate. Finally, a *done* message is sent to the primary process of the server group, which transfers it linearly to its subordinates. When that process receives the *ack* from its immediate subordinate, the call has finished its execution.

Application Transparency

This protocol is based on the passive replication approach.

In this protocol, nested calls can be supported by introducing their service within the linear sequence of messages generated during the service of another call.

Only server processes are replicated (they can work as clients during the service of a nested call). Client processes are not replicated.

The use of threads is not supported.

Efficiency

This system only uses point-to-point communication. With this mechanism, the number of point-to-point messages required for the normal processing of a call is reduced to $N+(2*M)$. However, the efficiency of the protocol would not improve with the use of a multicast mechanism due to its linear transfer nature.

Each call has to be processed sequentially and moreover, the execution of each call has to wait for its linearly ordered execution among all the members of a cluster.

This situation results in a very poor scalability. The processing cost of a call increases almost linearly with the number of replicas of a cluster.

2.1.4 Contexts

Using the group primitives provided by Amoeba [Kaashoek 92], Wood describes the implementation of a more efficient RRPC system [Wood 93]. Its scheme is based on the coordinator-cohort model. The coordinator of the server group (groups are called *contexts* in the paper) transfers the client call to its cohorts using the reliable multicast service provided by Amoeba. All the members execute the call, but only the coordinator sends the result to the coordinator of the client group, which forwards it to its cohorts through the use of another multicast.

The protocol was tuned and adapted to make the most efficient use of the Amoeba group primitives by exploiting knowledge of the internal implementation of these primitives. For example, Amoeba's groups are closed, permitting only intragroup communication. Its groups implement total order through the designation of a *sequencer* process that serializes the transmission of messages among the group members.

Client/server communication is achieved through the use of point-to-point calls among the coordinators of each group. The protocol elects the sequencers of each group as coordinators in order to improve its performance.

Application Transparency

The use of the protocol is syntactically quite transparent to its user. However, the user has to define *receive_state* and *transfer_state* routines to implement an application-level state transfer among the coordinator and a new cohort that joins the group.

Semantically, the protocol does not permit the use of nested calls and no provision for thread support is included. However, [Wood 93] points to some of the problems and solutions that could be introduced to achieve that support.

The protocol permits the replication and availability of client and server processes.

Efficiency

The execution of a RRPC requires the use of 2 point-to-point + 2 reliable multicast messages. The implementation of the reliable multicast service through the use of a hardware multicast mechanism improves the efficiency and scalability of the system considerably.

However, the protocol structure does not permit the concurrent service of different calls simultaneously. This affects the resulting efficiency of the system because incoming calls from different clients can only be served in sequential order.

2.1.5 Hot Replication

The scheme presented by [Beedubail et al. 95] uses the coordinator-cohort model. This permits a very quick recovery time (*hot replication*). The general algorithm for the processing of a RRPC is very similar to the one presented previously in the context groups. However client user processes are not replicated in this algorithm.

Nested calls are taken into consideration by identifying each call globally in the system. The identification scheme used is based on the one introduced in [Jalote 89]. However, its identification scheme presents the problem that the number of identification fields needed to represent a call grows linearly with the number of levels that the call is nested in – in other words, to identify a nested call of N levels, N different numeric fields are required.

Application Transparency

The use of Hot Replication is syntactically transparent to its user. However, as the user does not define any primitives to transfer the state and the system does not provide any automatic mechanism to do so, the system cannot permit new replicas to join once the state of the coordinator has changed its state and flushed old messages.

As mentioned previously, user clients are not replicated and the system supports the use of nested calls. However, the use of threads is not considered.

Efficiency

As in the previous protocol, a nested RRPC requires the use of 2 point-to-point + 2 reliable multicast messages. However, as the system does not provide support for threads, it cannot provide concurrent service for different simultaneous calls.

2.1.6 Piranha

Piranha is an availability management and monitoring tool for CORBA applications [Maffeis 96].

The CORBA standard does not include any group communication features to support the construction of fault-tolerant applications [Maffeis et al. 97]. For this reason, Piranha was implemented on top of Electra [Landis et al. 97], an extended CORBA architecture that provides the Isis Virtual Synchrony model.

Object references are persistent. They are still valid after the referenced objects fail. As soon as an instance of the object is restarted, client applications can make progress without having to contact the naming service. This mechanism is achieved by viewing object references as multicast addresses.

Failure detection is provided by a separate service that makes suggestions about failed objects using timeout mechanisms. Another service, the *availability manager*, relies on the information provided by the *failure detection* service to restart objects that may have failed. The *availability manager* is implemented in the form of an object group, with a group member per machine. The service is not scaleable because each member has to maintain information on all the application objects running on the other machines.

The system provides a very complete monitor service that provides system administrators with a complete view of the status of the objects running in the system. If an object has failed, the system administrator can detect whether or not the object has been restarted automatically, or can decide on the node where to restart it. This tool also permits the manual migration of objects for load balancing purposes.

From the user's point of view, RRPC's are activated as method invocations to remote objects.

The issues involved in developing fault-tolerant object-based applications differ from those of process-based applications in several aspects. First of all, a server process often contains many objects. The detection mechanism for *process crash* is not sufficient to detect the *object crash*. As many object servers are implemented as multithreaded servers with each object running in a separate thread, the detection mechanism for *process failure* may not detect an *object failure* (or a *thread failure*). On the other hand, the use of objects and inheritance provide a natural framework for the inclusion of methods that facilitate the implementation of fault-tolerant services, such as methods

to transfer the current status of an object or methods that permit the use of different reliability semantics.

Unfortunately, the protocol does not seem to deal with nested call invocation, which in the context of CORBA, is a considerable drawback due to the fact that the definition of the OMA standard requests and enhances the use of nested RPC calls [Vinoski 97].

2.1.7 Phoinix

Phoinix provides the development of fault-tolerant CORBA applications by extending the CORBA service without modifying the original standard definitions [D. Liang et al. 98].

Instead of including the use of group communication primitives and replicated objects, the system relies on the exception mechanisms provided by the standard. In the case of an object failure, the client receives an exceptional signal from the ORB. However, instead of raising this exception signal to the client directly, a Phoinix reliable module (linked with the client's code) intercepts the exception signal and activates another service object via the ORB, binding the client to the new restarted service.

As service objects cannot be replicated, a special service, called the *log manager*, is in charge of keeping track of the necessary state information to recover any failing object. To ensure the fault-tolerance of this service, the *log manager* is replicated as two CORBA objects, each being the hot standby of the other.

Application Transparency

Although the Phoinix mechanism is syntactically transparent, semantically it does not support either the use of nested calls or the recovery of client processes. However, the introduction of support for nested calls could be achieved with the inclusion of global identifiers and appropriate logging of requests and responses.

Efficiency

The scalability of the Phoinix system is one of its biggest problems because the log manager has to hold state information for all the existing reliable objects.

The resilience degree of the system is limited. The simultaneous failure of a reliable object and the log manager could be disastrous, but this scenario is not considered to be likely to occur.

The performance of the system under normal conditions is very good. The transmission of a RRPC only involves the use of 2 RPC's, one to the server object and another to the *log manager*. Concurrent service of different calls is tolerated.

The recovery process will have to create an object and restart it from scratch. The time for recovery will depend on the checkpoint frequency used.

2.1.8 Conclusions

Ideally, we would like to be able to transform our currently unreliable applications into efficiently fault-tolerant ones automatically.

However, the idea of developing a general method to construct fault-tolerant applications in a completely transparent manner is fallacious. Programmers have to be aware that they are developing fault-tolerant applications, at least to ensure that their systems are deterministic.

Whichever application is being implemented, programmers should be provided with powerful and simple tools that could be used comfortably to obtain reliable systems at reasonable levels of efficiency.

As we can see from the tables below, the current situation is far from ideal. The existing systems are semantically too restrictive, either excluding the use of nested calls, or not permitting the recovery of client processes. The systems that do permit nested calls are too inefficient, particularly because they do not permit the concurrent service of different calls, a very important feature for a service that may have to provide support to large numbers of clients.

For this reason, the ability to support threads is an important characteristic on the server side. However, on the client side it is not that important, particularly considering that the resulting systems have to be deterministic. The programmer would have to be very careful with the use of threads on the client side.

Transparent recovery is an important feature because the programmer should not have to deal with node failures. However, the requirement of having to include state transfer routines in the code should not be a problem from the programmer's point of view, particularly when using an object oriented language.

Turning to the issue of efficiency, it is important to take into consideration the scalability of the system, particularly in systems that have to deal with large numbers of nodes and processes (or

objects). Current systems, where failures are considered to be rare, will not need a high degree of resilience.

Finally, the time needed for recovery is a factor whose importance varies according to the type of applications being dealt with. Control applications where the response time is a very important factor may need very fast recovery methods. Normally, the recovery time depends on the system model used by the system, whether it uses active, passive or coordinator/cohort replication. However, the normal response time of the system will also depend on the model used. Passive systems will not require as many resources as active ones. If possible, the programmer of the system should be able to choose what model suits his/her needs better.

	<i>Troupes</i>	<i>Clusters</i>	<i>Contexts</i>	<i>Hot Replic.</i>	<i>Phoenix</i>
Nested Calls	No	Yes	No	Yes	No
Thread Support	Yes	No	No	No	Yes
User Client Recovery	Yes	No	Yes	No	No
Transparent Recovery	No	Total	Transfer State Routines	Total	Transfer State Routines

Table 1 Summary of the properties needed to achieve reliable semantic transparency

	<i>Troupes</i>	<i>Clusters</i>	<i>Contexts</i>	<i>Hot Replic.</i>	<i>Phoenix</i>
# Msg/Call	$N * M$ ptop	$N + 2 * M$ ptop	2 ptop + 2 mcast	2 ptop + 2 mcast	2 RPC
Concurrent Service	No	No	No	No	Yes
Scalability	Poor	Poor	Good	Good	Poor
Recovery Time	Instant	Slow	Fast	Fast	Slow

Table 2 Summary of the efficiency costs for fault-tolerance

2.2 Fault-Tolerant Linda Systems

2.2.1 Introduction

This section introduces the issues involved and solutions taken in introducing fault-tolerance to the Linda model.

Two basic approaches are taken:

- Systems that implement fault-tolerance trying to preserve the original semantics of the model.
- Systems that introduce changes in the original model in order to achieve more efficient implementations.

2.2.2 Reliable Tuple Space

In Linda, processes communicate via use of the shared structure called *tuple space* (TS). Consequently, the failure survival of TS is one of the most important aspects in any fault-tolerant Linda system. This reliability requires some form of redundancy.

[Xu et al. 89] describes a design in which accesses are based on a read-any-write-all protocol. There is an incarnation number associated with each access to the TS (called a *view*). The failure of any node changes the global incarnation number using a two-phase commit protocol. The algorithm replicates all the tuples in each TS replica. However, [Xu et al. 89] discusses a method to partition the TS among different sets of replicas.

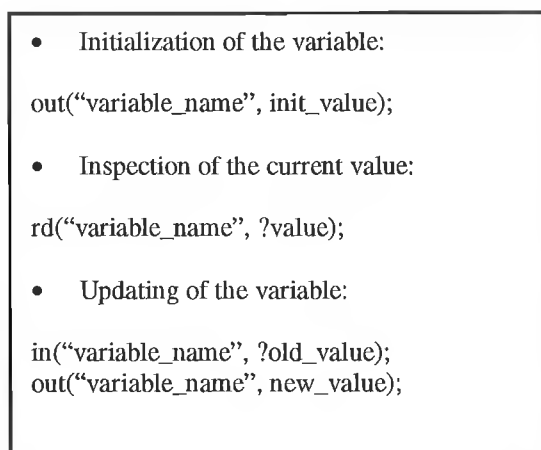
[Bakken et al. 95] uses the replicated state machine approach where the tuples are replicated in each processor using an atomic multicast protocol. This means that each tuple operation performed on stable TS generates an atomic multicast message in order to replicate TS in each tuple server of the system. In this case, each tuple server holds all the tuples present in TS.

Plinda [Jeong et al. 94] replicates TS in disks using a checkpoint mechanism. In this mechanism, TS will not be available until the restart of the failed processors.

However, the existence of reliable TS is not sufficient in itself to obtain a fault-tolerant Linda system. One of the main problems with fault-tolerant Linda is the lack of sufficient atomicity in its operators. If we consider the initial semantics of Linda, each operation modifies the shared tuple

space independently, i.e., Linda provides only single-operation atomicity. Linda programs cannot be structured to handle failures with these semantics. [Bakken et al. 95] presents a simple example to illustrate some of the problems involved with the use of these semantics.

In the example we want to use a shared variable between our processes. In TS we will keep a tuple containing the name and value of the shared variable. The typical operations used to access such variable are represented in Figure 4.



- Initialization of the variable:
`out("variable_name", init_value);`
- Inspection of the current value:
`rd("variable_name", ?value);`
- Updating of the variable:
`in("variable_name", ?old_value);`
`out("variable_name", new_value);`

Figure 4 Example of Linda operations used to access a shared variable

Unfortunately, if the process that is updating the value of the variable fails just after withdrawing the tuple that contains the old value of a variable, the rest on the processes of the system will reach deadlock waiting for the value of a variable that was never replaced. The problem is due to the inability to execute atomically all of the operations that update the value of a variable.

[Kambhatla 90] discusses a method to implement fault-tolerance while at the same time keeping the initial semantics of Linda. The method is based on checkpointing process states and logging messages. The main idea is to keep a *log space* (LS) for recovery purposes. LS is similar to TS but with an important difference: the tuples in LS have a total order based on the process identifier and the time of insertion. In LS we keep the last checkpoint state of TS and all the tuples that have been *outed* and *ined* since the last checkpoint. When we have to restart execution we have to go through the following steps:

- Load the checkpointed state.

- For each *out* operation, first check if this tuple is already in LS. If it is, then the operation logically becomes a null operation (this mechanism should also be used for each *eval* operation).
- For each *in* operation we first check LS to verify if the tuple had been received before. If it had, the operation is satisfied with the tuple present in LS. Otherwise, we access the tuple from TS.

[Kambhatla 90] discusses an algorithm to ensure that the states of LS, TS and processes are consistent. The estimation overheads of this method are predicted to be linear with respect to the number of tuple operations performed by the program. However, the method has not been implemented. We note that the method only works for processes whose execution is deterministic upon receiving the same replies from TS.

2.2.3 Redefining the Semantics of Linda

The languages that redefine the semantics of Linda introduce operators to group different Linda operations into one atomic action. The larger the number of operations grouped into one single action, the lower the number of opportunities for parallelism that can be obtained.

MOM

The MOM system [Cannon et al. 94] redefines the semantics and syntax of Linda. It introduces a *done* call that is equivalent to the *commit* calls typical of database transactions. The system implements fault-tolerance by not committing any partial result until *done*. The system forces the programmers to use the bag-of-tasks paradigm [Carriero et al. 90]. A process first requests a task in tuple space (TS), performs the indicated processing and finally returns the results to TS. At the completion of the task a *done* call is issued to indicate that all the task tuples retrieved have been successfully processed. This mechanism simulates the existence of a two-phase commit protocol. Under these semantics it is much easier to implement a fault-tolerant mechanism. The system locks all the *ined* tuples and stores all the *outed* tuples in temporary storage until the issue of the *done* call, at which point all the partial results generated by the process are reflected in tuple space.

The MOM model only allows for algorithms that can be expressed using the producer/consumer paradigm. Another problem is that the model imposes a certain degree of serialisation since intermediate results are not reported until *done*. Consequently, MOM influences the resulting efficiency of our program even in the case of normal execution with no failures.

Linda Piranha

The Piranha system [Carriero et al. 93] is another variant designed to utilise idle workstations effectively for parallel computation. This model also structures the parallel programs using the producer/consumer paradigm. But in this case the system is designed to take advantage of new resources as they become available, and also reduce the number of usable resources without aborting. The Piranha system may cancel the execution of a process when a workstation needs to be used for other purposes. If this happens, the process in execution terminates its local computation and restores any intermediate results using a retreat operation. Therefore the programmers need to write the code to restore any intermediate result which makes programming more difficult than with the MOM model.

FT-Linda

FT-Linda [Bakken et al. 95] does not restrict the programmer using the producer/consumer paradigm. It introduces two new constructs to the syntax of Linda: the Atomic Guarded Statement and the Atomic Tuple Transfer.

The Atomic Guarded Statement provides all-or-none execution of a group of tuple operations. The execution of this grouped statement is conditioned to the execution of a *guard* statement. It has the form:

$$(\text{guard} \Rightarrow \text{body})$$

The process is blocked until the *guard* either succeeds or fails. If it succeeds, then the *guard* and the *body* are executed as an atomic unit; if it fails, the *body* part is not executed.

FT-Linda permits the creation of multiple tuple spaces with different attributes. For example, the user can define a *stable* or a *volatile* tuple space. Only the *stable* tuple spaces will survive processor failures. A tuple space can also be *shared* or *private* to indicate which processes may access it. The Atomic Tuple Transfer primitive permits the movement and copying of tuples atomically between different tuple spaces. With this primitive the programmer can evaluate all the partial results in a local tuple space and, at the end, transfer all the tuples from the local to the global tuple space in a single action.

The introduction of these primitives facilitates an efficient implementation of the model by relaxing the implementation of the two-phase commit protocol. However, the programmer needs to be involved heavily in the reliable implementation of the program. Each FT-Linda program requires the implementation of a *monitor* process which, in case of failure, will re-establish the system to a consistent state.

```

process worker()
  while true do
    ( in (TS, "work", ?task_args) =>
      out (TS, "in_progress", my_id, task_args) )
    calc (task_args, &result)
    ( in(TS, "in_progress", my_id, task_args) =>
      out(TS, "result", result) )
  end while
end worker

process monitor(failure_id)
  while true do
    in(TS, "failure", failure_id, ?host)
    while ( in(TS, "in_progress", host, ?task_args) =>
      out (TS, "work", task_args) do
      nooperation
    end while
  end while
end monitor

```

Figure 5 Programming the bag-of-tasks paradigm in FT-Linda

For example, Figure 5 shows the structure of a program which represents the bag-of-tasks paradigm using FT-Linda. In this case, when a task is withdrawn from tuple space, a new tuple *in_progress* describing the current task in execution is deposited atomically in tuple space. When a host failure occurs, one failure tuple is deposited into TS indicating the host that failed. The monitor process then detects the failure and can restore all the *in_progress* tuples from the failed host.

One aspect not mentioned in [Bakken et al. 95] is the explanation of what happens in the case of the monitor process failing after detecting a failure. As we can see, the programming of a complex application can be very cumbersome and difficult to verify. Although the resulting system may be efficient, the programmer has a large responsibility in the final efficiency of the program. FT-Linda encourages the programmer to keep all the temporal results in local tuple space and, at the end of the processing, transfer the final results to global tuple space. Consequently we also end up with a serialised program.

PLinda

PLinda [Jeong et al. 94] provides a mechanism to checkpoint the current state of a process. In this way, processes are allowed to save only minimal information required for recovery at checkpoint. This makes the mechanism cheaper than transparent checkpoints that have to save the entire process image in stable storage. The mechanism is used in a similar form as access to tuple space but it is designed to save the state of a process. This separate tuple space is accessed sequentially and it associates each tuple with a particular process.

Plinda also provides two operators (*xstart* and *xcommit*) to group the statements to be executed atomically. The programmer is not forced to use the producer/consumer paradigm. A task can have different groups of atomic statements to execute, preserving the current state of the task between these atomic statements. In this way, a task can recover to the last checkpointed state in case of failure.

2.2.4 Conclusions

This section has presented two different approaches for the introduction of fault-tolerance in Linda systems.

The first option is the automatic provision of fault-tolerance without introducing any change in the original semantics of the model. This is the ideal approach from the programmer's point of view because the use of the original model provides the more natural solutions and, at the same time, permits the automatic conversion of the available deterministic applications into fault-tolerance ones. However, from the system designer's point of view, the provision of such a fault-tolerant system represents a complex problem that may offer very poor levels of response. Currently, we do not know of any existing implementation with such characteristics.

On the other hand, the introduction of changes in the original semantics of the model theoretically facilitates the implementation of fault-tolerant Linda systems, permitting better levels of system's efficiency. However, programmers are interested in the efficiency of their resulting applications more than the efficiency of the system. Using these models, the programmer is forced to serialise the original applications, reducing the opportunities for parallelism. It is difficult to assess the approach that can provide better response times at the application level. Furthermore, the use of these models may obscure the semantics of the original model, requiring more effort on the part of the application programmer.

Definition of a New Reliable RPC Protocol

R²PC: Transparent Fault-Tolerance

3.1 Introduction

The main purpose of this thesis is the design of an efficient and scalable mechanism that provides simple, clear and transparent semantics for the development of fault-tolerant applications.

The Remote Procedure Call (RPC) mechanism provides an appropriate framework to construct distributed applications. The programmer does not have to deal with the transmission or reception of messages. This mechanism can be used in a very similar way to a normal procedure call but with the important difference that the procedure is executed in a foreign host.

Since the definition of RPC [Birrell et al. 84], this mechanism has been very successful and has been the basis for the implementation of many existing distributed systems and for the specification of standards for development of future systems (like Method Invocation [OMG 98] or Java RMI [<http://www.javasoft.com>]).

Section 1.3.2 has shown that the original definition of the protocol forces the programmer to be aware of a number of possible errors during the execution of a call. For this reason, different variations of the original protocol have been introduced (called Reliable Remote Procedure Call - RRPC - systems). Nevertheless, section 2.1 has shown that existing systems are semantically too restrictive and/or inefficient.

This thesis introduces R²PC, a new RRPC system that adds reliability to the original RPC protocol while retaining the main semantics of the model at a reasonable cost.

In order to guarantee fault-tolerance, R²PC introduces a few syntactic and semantic changes to the original definition of RPC.

3.1.1 Group Addressing

One of the main problems involved with normal RPC protocol is that client processes specify a particular node where the remote procedure is to be executed. This ensures that if the server process fails, the mechanism has to return an error. There is nothing else that can be done.

Consequently, in order to have a transparent and reliable mechanism it is necessary to address process groups. Every process of the system will form part of a process group. The group members exchange information through the use of group communication primitives.

The current implementation of the system follows the primary-backup approach whereby the primary process carries out the main computation of the group and transfers information about its status to the replicas.

3.1.2 Execution Semantics

From a programmer's point of view, the execution semantics offered by the R²PC protocol are *Exactly Once* semantics with no possibility of failure. Any failure will be dealt automatically by the system. If the system can not deal with all the failures, the result will be the failure of the system.

With these semantics, programmers can concentrate in the development of their distributed system achieving automatic fault-tolerance without having to worry and deal with all the possible fault scenarios that may occur.

In order to achieve this target, each R²PC call is treated as a transaction, keeping the properties of Atomicity, Isolation, Definitiveness and Consistence [Elmasiri et al. 94].

The implementation of the transaction mechanism can be quite simple in the case of using idempotent procedures. In this case, the result of the call will not depend on the status of the server but on the parameters from the client. Taking this precondition into consideration, in the case of a server failure, the call can be simply redirected to another available server.

The main problems become apparent in the case of using servers that need to preserve their status according to the previous calls made from the clients which, in practice, is the most usual case.

- Atomicity: The R²PC has to be either completely executed or not executed at all. In case of acceptance, the call has to be processed exactly once, and once it has been processed, there is no way back. To accomplish these requirements we have to consider for example,

the situation where a client does not receive a reply message. How do we establish whether the call has been processed or not?

To address this requirement each R²PC call has been associated with a unique sequence number. If a reply message is lost, the server will now be able to distinguish whether the call had already been processed. If this is the case, the server process will only have to return the same results evaluated previously. The form and use given to these sequence numbers will be presented in more detail in the description of the protocol.

- Consistence: If a transaction starts in a consistent state, after completion of the transaction, the resulting state has to be consistent as well.

Every process is presumed to be deterministic upon reception of the same results from the execution of the same sequence of R²PC calls.

The protocol does not define an ordering on the results of the original sequence of R²PC calls. However, in the case of a failure, the system will generate the same sequence of results from the re-executed R²PC calls. This order will be preserved as well for the case of nested calls.

- Isolation: The transaction has to be executed independently from other transactions that are executing concurrently. In effect, the transaction is executed as if it was isolated.

In the new system, all the R²PC calls are sent to and processed by a single process, which is the current primary process. This process orders the execution of the calls. The least the primary process must do is to sequence the transmission and acceptance of results. Different calls may be processed simultaneously using different threads, but the programmer has to establish the appropriate critical sections in order to prevent calls interfering with each other.

- Definitiveness: Once the transaction has been accepted, its results are permanent.

For example, what happens when the server process that attended the request crashes?

The results and calls evaluated by each process are kept in the replicas until a checkpoint is made. Each process in the system forms part of a group.

However, the provision of these requirements may not be sufficient. It may be necessary to define a transaction as a much bigger unit than a R²PC. For example, what happens when a client

crashes? Maybe this client had some pending R²PC calls to execute. Should the R²PC protocol guarantee the restart and recovery of the client processes? This idea does not seem to have much sense in a broad distributed system - a server in Alabama cannot control whether a client in Dublin is working or not. However, in a restricted environment where there is a need to implement a reliable system, for example, a computer's LAN controlling a nuclear power plant, such a guarantee would be sensible.

For this reason, the property of Consistence is treated at two different levels:

- A single R²PC - keeping the consistence of the execution of this R²PC among all the servers of the group.
- The complete execution of any process - if any primary process crashes, the group will elect one of the replica processes as the primary process and will restart it from the last checkpoint before the crash.

The system manager can decide what groups need replication and in what numbers by specifying the resilience degree for each group.

The next sections describe the theoretical and technical details of our implementation of R²PC.

3.2 R²PC Protocol

3.2.1 System Overview

In order to guarantee the recovery of client and server process, in the R²PC system there are no single processes. Any process forms part of a group.

Each group of the system is an independent unit organised using the primary/backup approach. In this case, each group elects a particular process, called *primary*, that carries the computation of the group. The *primary* sends recovery information to the rest of the members of the group, called *replicas*. In the following description, when we refer to the evaluation of a call by a group, it is the primary process of the group that carries out the evaluation unless otherwise specified.

When a client group executes an R²PC call, a message (identified with a unique call identifier) is sent to the corresponding server group, which evaluates the call and sends a reliable multicast message with the results to all its replicas. Only after a successful multicast to the replicas in the server group does the server send the results back to the client's primary process. Finally, the

primary client process sends a reliable multicast to its own replicas with the result of the call before returning control to the application.

Each process group can act as client, server or both. Each group holds two history buffers: one for the client part and the other for the server part of each process. This information is stored for recovery purposes. In this way, every process can be restored to its previous state by re-evaluating all the local calculations made from the last checkpoint. All this information can be stored in a machine independent format, avoiding the dependence on a particular machine configuration.

The client side of the history buffer stores the results returned from each call so that, when a client crashes and restarts, it does not have to ask the server processes to re-evaluate the calls that had already been made. The client uses the information stored in its internal tables to re-establish the state of the group. The server side needs to store information about the input calls that had been processed from the last checkpoint. At restart, a primary server will re-evaluate all these calls from the last checkpoint.

In the case of a primary server crash, a new server is elected. As the failed server could have been processing the call from a client, the client may have remained blocked waiting for the result. For this reason, when a client does not get any result back from the server, after a certain period of time, it will repeat the call. The *directory service* - a service in charge of translating any group address into the network address that identifies the current primary process of the group - will re-direct the call to the new primary server of the group. Once the new server receives the call, it will check the call identifier. If the call had already been processed before the crash it will just return the results, otherwise it will evaluate the call.

If the crash is on the client side, the results returned by the server will be lost. However, this is not a problem. When the client restarts execution again, the call will be requested again and the server will not re-evaluate the call, it will just send the same results once more.

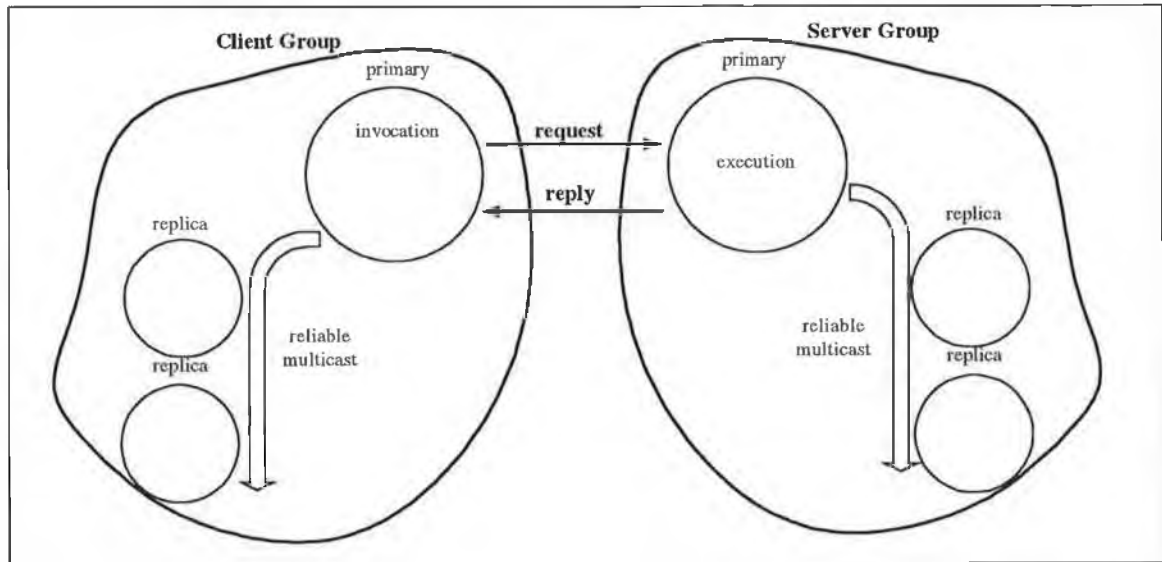


Figure 6 Overview of the R2PC Protocol

3.2.2 Nested Calls

This section is going to compare our protocol with the *State Machine Approach* [Schneider 90] in order to verify the correctness of our model.

State machines are used to model services and servers. [Schneider 90] defines a *state machine* as a set of state variables, which encode its *state* and *commands*, which transform its state. A deterministic program implements each command. Execution of the command is atomic with respect to other commands and modifies the state variables and/or produces some output. A client of the state machine makes a request to execute a command. The request names a *state machine*, the command to be performed and contains any information needed by the command. *Output* from request processing is sent to the client that is awaiting response from its prior request. Outputs from a state machine are completely determined by the sequence of requests it processes, independent of time and any other activity in a system.

In case of a nested call, output from a state machine can also be used as a request to other state machines. This case is not considered in the original model of a state machine. In fact, the introduction of a nested call could introduce non-determinism in the execution of the state machine. Why?

Let us consider the case of Figure 7, where P_i are processes and M_j refers to the j th call requested by the client process P_i . The execution of a command will involve a return of information for the executing process.

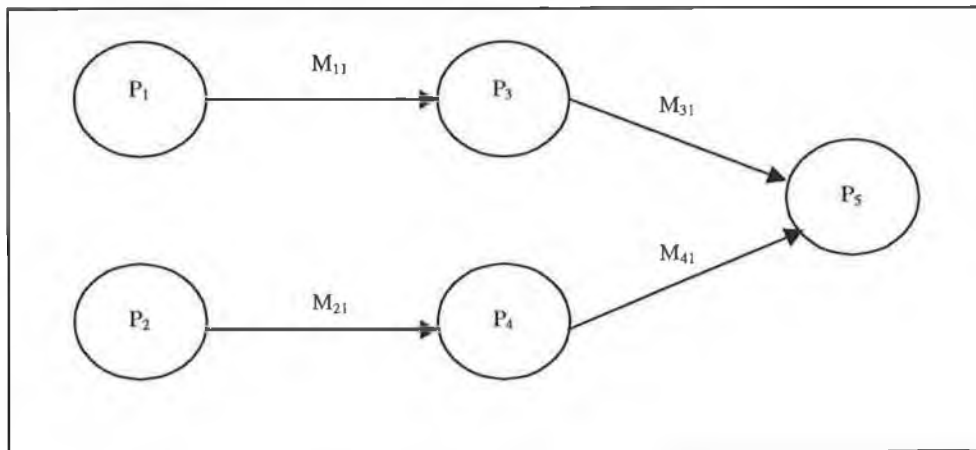


Figure 7 Possible command sequence generated by nested machine states

The execution of the command M_{11} in P_3 generates the execution of the command M_{31} in P_5 . At the same time, the execution of M_{21} in P_4 generates the execution of the command M_{41} in P_5 . Now, if M_{31} and M_{41} are not commutative commands over P_5 , neither P_3 nor P_4 behave as a deterministic state machine anymore. Their results will depend on the results that they receive.

How can this problem be solved for this example? In fact, with the use of nested calls, the outputs from a state machine are not completely determined by the sequence of requests it processes. For example, in our case, the results from P_3 depend on whether P_5 has processed the command M_{31} before M_{41} or not. However, once this sequence of commands has been executed, its order is not going to change anymore - i.e. its execution order will be determined.

Using R²PC, it does not matter whether P_5 processes M_{31} first or M_{41} first. However, once P_5 has taken a decision and executed these commands in a particular order, this order does not change anymore. If any state machine is restarted again, it has to execute the same sequence of commands that had been executed in first instance.

The restart of a state machine by itself is not a problem. The only task to be done is the execution of the same sequence of commands that had been invoked in the first instance. However, the system needs to identify uniquely each message generated by any process, in such a way that in

case of a repeated invocation, the server can return the results already evaluated without the need of re-execution. How is this going to be achieved?

The message identifier cannot be a simple sequence number per connection because, although this technique works for client-to-server connections due to the fact that client processes are deterministic upon reception of the same results, server-to-server connections (nested RPC calls) are not. As we have seen, the sequence of messages that a server generates is not deterministic upon reception of the same results. This sequence also depends on the order of calls received from its client processes.

In the first instance, this factor should not be such a problem because, in the case of restart of a server process, re-evaluating the calls that the server had received in the same order as the calls arrived from the clients could regenerate the same sequence of calls. However, this would force the server to process each client call in a sequential way. Serialisation of calls would be needed to ensure that the sequence order of the calls generated at the restart of a server is exactly the same as the order generated in the first instance. Otherwise, in the case of service calls that implied the use of two or more nested calls, the system could not guarantee that the sequence of calls generated by the server would be the same because different server threads could interact between the service of different calls.

Therefore, instead of using simple sequence numbers, each command to execute in the system has to be identified precisely, not only among the requests that have been received in our state machine, but also among all the requests generated in the system.

In the R²PC system, each client process is a deterministic process upon reception of the same results from the same sequence of calls. Therefore, the execution of a client process will precisely determine a unique sequence of calls. Consequently, to identify each message completely, the group *id* of the client will be used together with the sequence number of this message within the sequence of messages originated by this client¹.

The Figure 8 presents a case where the invocation of M_{11} in C_1 determines the execution of M_{12} followed by M_{13} and M_{14} . The results from these operations will determine the execution of other

¹ [Jalote 89] presents another scheme for identifying each call globally in the system. However, its format presents the problem that to identify a nested call of N levels, we need to use N different numeric fields.

commands (M_{15}) until the formation of unique chain of commands formed by the execution of C_1 .

As each call can now be uniquely identified in the system, we can ensure that the same results are returned for the evaluation of any previously executed command.

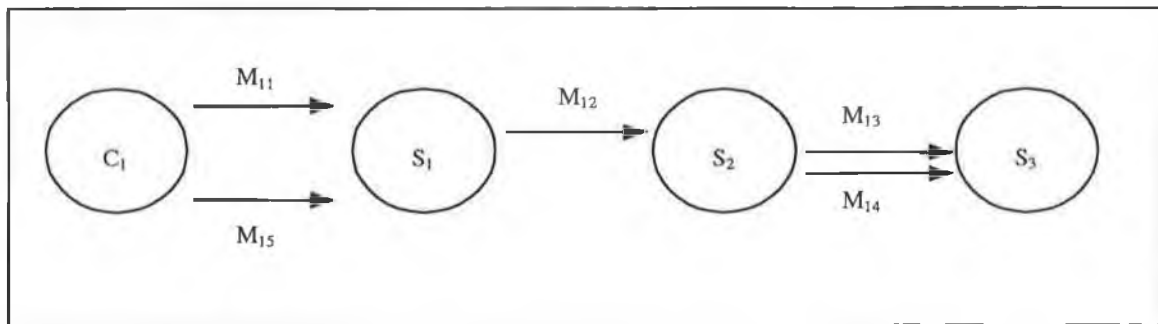


Figure 8 Possible deterministic command sequence generated by a client (the vertical axis represents time)

The use of the global identifiers solves the problem of recovering client and server processes with nested R²PC calls. As a result of this mechanism of global identification, it is not necessary to impose an order of execution in the processing of the calls. In consequence, server processes can use concurrent threads to process different calls. However, it is task of the programmer to define the proper critical sections to ensure that the execution of concurrent calls in a server do not interfere with each other.

On the other hand, programmers need to ensure that their applications are deterministic, i.e. the behaviour of each client process (or server in case of a nested call) can be determined from the results obtained in the execution of the R²PC calls evaluated. This is due to the fact that the client processes are used to fix the order of execution of the calls in the system.

3.2.3 Fault Recovery

During normal operation, the primary process will send enough information to the replicas to permit them to resume operation of the group in case of failure.

On the client side, replicas will store the results returned for each call number. In this way, when a client crashes and restarts, it will not ask the server processes to re-evaluate the calls that had

already been made. It will merely need to retrieve the results from these calls to return to its previous local state.

On the server side, replicas will have to know what calls had been processed by the primary process from the last checkpoint. The new server will have to re-execute all these calls again to return to its previous local state.

After the failure of a primary process, the system will elect a new one. Then, this new primary process will execute a recovery algorithm. During recovery, the system ensures that the calls are re-evaluated in the same order that had been determined by the evaluation of the primary process. It does this by using only one thread to re-execute all the calls previously evaluated. This order will be fixed by the time a call finishes its evaluation and not when the call starts being evaluated - i. e. If two calls were being served simultaneously by two different threads, the call that had finished its execution first will be the one evaluated first during the recovery process. Otherwise, the recovery phase could finish in a deadlock. For example, when the first call being evaluated had to wait for a result provided by a second call. During normal execution, the second call to arrive could finish its evaluation first and then, the first call received would finish afterwards with no problem. However, during recovery, if the first call received was evaluated first the system would reach a deadlock because, at this time, only one thread was being used and the system would wait indefinitely for the first call to finish before processing the second one.

It is important to note that, in order to guarantee a proper recovery, programmers have to ensure that R²PC services include a single critical section that only releases shared resources at the end of the service. Unfortunately, the inclusion of this requirement reduces considerably the potential for concurrency. However, this condition is necessary to guarantee serializable and recoverable schedules [Ullman 88].

Figure 9 presents the information that the system stores for recovery purposes.

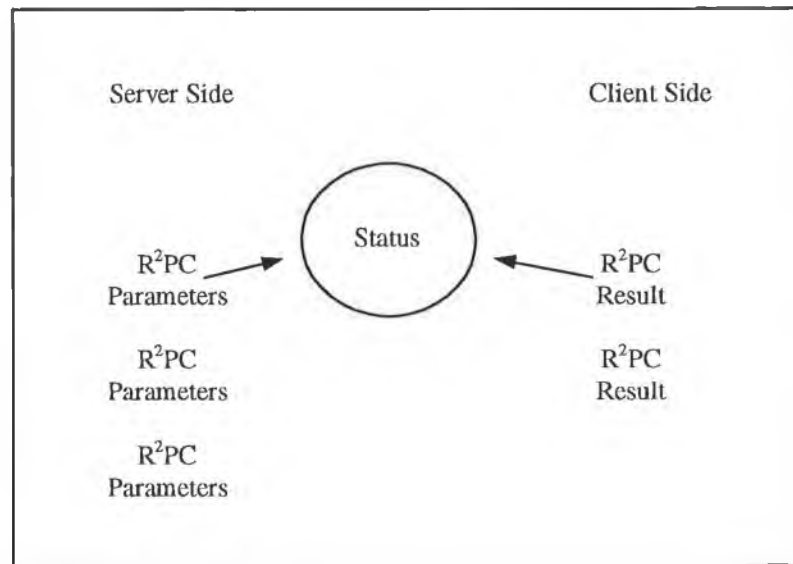


Figure 9 Information to store for recovery purposes

The main problem with the scheme presented is that the information required to keep the status of the system grows linearly with the number of calls evaluated by the processes. This is intolerable, particularly for processes that are supposed to be active for long periods of time.

One first approach to limit the amount of information required would be to use a coordinator-cohort scheme in which replicas were actively re-evaluating the calls received from the primary process. In this way, replicas could flush the RPC calls as long as they were evaluating them. The system should ensure that replicas only use the information stored in their cache without interfering with the work processed by the primary process. This approach would be very similar to the one taken by [Beedubail et al. 95]. This would permit very quick recoveries because the state of the replicas would be nearly the same as the one of the primary process. The main drawback of using this method on its own is that the system would be unable to permit the reestablishment of new replicas into the group. Once the recovery information has been flushed, a new replica process will not be able to join the group as it cannot get the recovery information anymore. This is unacceptable because, in the long term, the process group would eventually fail as it would not be able to re-establish the components that had failed.

Consequently, in both cases the use of checkpoints is required. In the R²PC system, a *checkpoint* is the action of transferring all the information about the current state of the primary process to the replicas so that if the primary process is interrupted, a replica can be restarted at the point at

which the last checkpoint occurred. Therefore, once a checkpoint has been executed, all the processes can flush the information about the R²PC calls processed in the group up to this last checkpoint.

The quantity of information required to be stored in a checkpoint is normally quite constant during the lifetime of an application, particularly in the case of control/communication applications in which we are interested. Checkpoints in transactional systems may require the transference of enormous quantities of information because they store all the data in file systems.

The use of shadow copies is an interesting approach for the provision of checkpoints in transactional systems [Parrington et al. 94] avoiding the transfer of great quantities of information during a checkpoint. However, the use of this mechanism can delay the recovery phase because it forces the system to wait for crashed processes to be restarted (cold standby).

Control/communication applications do not usually require holding great amounts of state information [Birman 91]. This fact facilitates the use of hot standby techniques for such systems as it can be assumed that it is possible to transfer their state information in reasonable amounts of time.

Another problem usually associated with the use of checkpoints is that the information required to save the state of a process is normally machine dependent. Consequently, this information can only be exchanged among compatible processors. This problem can be solved using an externalised format that can be exchanged among different processor models.

Another interesting approach to consider is the introduction of transfer state routines by the user. This facilitates the implementation of the checkpoints and also reduces the information needed to transfer the state of a process. However, this forces the programmer to implement routines to transfer and receive the current state of the application, which in some cases may not be that easy, particularly in applications that need to be reused. However, such a mechanism can be incorporated quite naturally in object oriented languages.

3.2.4 Implementation Details

The protocol has been implemented on a network of Personal Computers using the Windows Sockets 2 API as network communications platform.

The Group Object offers the main functionality required for the implementation of the reliable transaction system. The notation used to describe the design of the components of the system is based on the notation presented in [Sully 93]. An object is represented as a rectangle with the name of the object at the top and a series of “buttons”. These “buttons” are labelled with the name of the services that the object offers to the rest of the application.

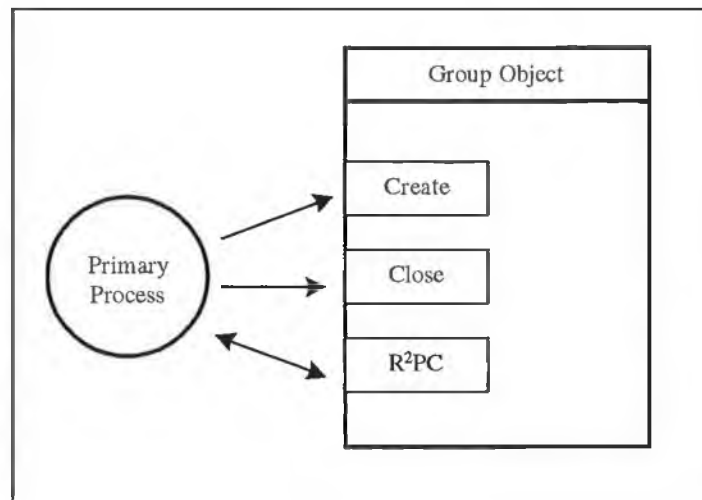


Figure 10 Representation of the Group Object

The creation of a Group Object marks the start of the monitoring of this group (start of the transaction at the level of consistency of the group). This monitoring process will not finish until the execution of the Close method.

The execution of each R²PC is treated as the execution of a complete transaction in itself.

The refinement for the Group Object is shown in Figure 11. This basic scheme represents the interactions among this object and the Directory Service (DS) for the creation and termination of replica processes.

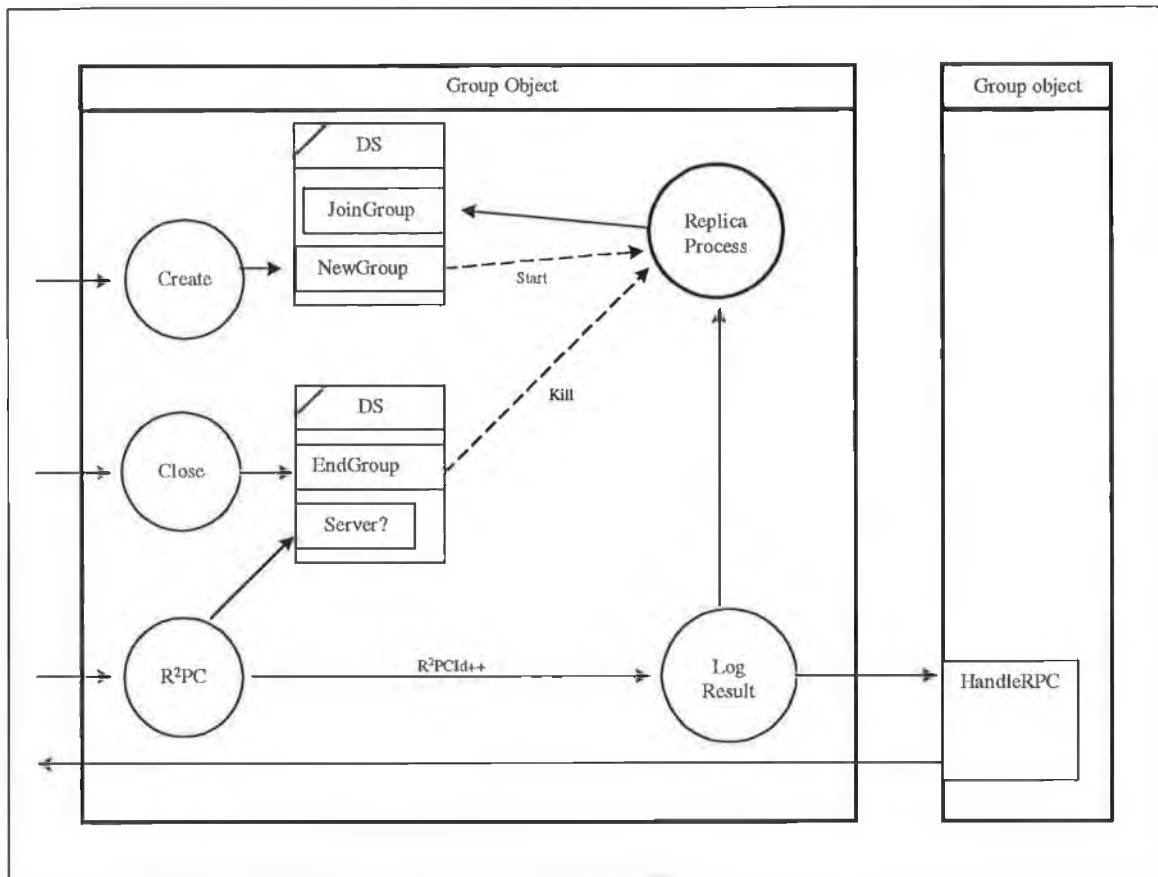


Figure 11 Refinement of the Group Object

The code for a R²PC process group is included in a single program. On the server side, the programmer has to specify the handlers that will serve each procedure call. On the client side, the programmer has to define a main part that will contain the code that the group will execute at start time.

Normally, R²PC applications will require the use of common services, such as a Directory Service (DS) that can be used to provide a mapping between logical names and group identifiers. The DS assigns these services permanent group identifiers that are known to the rest of the groups (see section 3.3.3).

The current implementation does not provide any stub compiler because R²PCLinda (see section 4.3) has been used as our main user-programming interface. However, the implementation of such a compiler should be a simple exercise due to the regular structure of the final code required for a R²PC application (see Appendix A).

At run time, the system will automatically create the threads necessary to execute the main code of the application, to serve the incoming calls and to provide the mechanisms for fault detection and recovery. The replicas for each group will also be created automatically when a group is started.

3.3 Group Communication

3.3.1 Introduction

This section describes the characteristics required and provided for the group communication system designed to support our implementation of R²PC. The resulting mechanism is based on the group communication system adopted in Amoeba [Kaashoek 92]. However, due to the fact that our protocol has been specifically designed to be used as a platform for the development of R²PC, we have constructed a theoretically more efficient implementation of reliable group communication. The main reason for this achievement is that Amoeba's groups elect a process to sequence all the communication among group members. Therefore, any message sent to the group has to be redirected first to this sequencer process. This is not necessary in our case because the primary process of a group automatically takes the role of this sequencer process.

3.3.2 Group Structure

Our process groups follow the *primary-backup* approach - the primary process carries out the main computation of the group and transfers information about its status to the replicas.

Groups are dynamic, permitting the incorporation of new members to the group at any time of the process.

The first group to exist in the system is the Directory Service (DS). The DS is a special group that translates any group address into the particular address that identifies the current primary process of each other group. Another important task of the DS is the distribution of replica processes among the different processors of the system according to the information provided by the primary process of a group.

The DS is composed of a number of Directory Service Providers (DSP). Each active processor holds an active DSP. For efficiency reasons, each DSP may hold information from other DSPs cached in its local memory. But there is no guarantee that this replicated information will be kept consistently. A DSP is uniquely responsible to guarantee the consistency of information that refers to the groups whose primary process resides in the same processor as the DSP itself.

3.3.3 Group Identification

Group Identifiers (*groupId*) are used to provide a unique mapping for intergroup and intragroup communication.

A new R²PC process notifies its existence to the system by sending a *new_group* message to its local DSP. At that time, the DS generates a unique *groupId* for the new process and starts the execution of replica processes according to the reliability required for that group. Once this is done, the *groupId* is sent back to the process that initiated the *new_group* call and this one becomes the primary process for the new group. From that moment, the primary process can start its reliable execution.

Generation of Group Identifiers

The DS reserves an address space for the allocation of permanent *groupId* used by group services. In the rest of the cases (i.e. normal application groups), *groupIds* are generated dynamically by the DS.

The current implementation allocates these identifiers sequentially. The DS ensures that each group has a unique *groupId* by using a voting protocol based on the two-phase commit protocol (see [Davidson et al. 85] and [Babaoğlu et al. 93] for a description of the two-phase commit protocol).

During initialisation, a DSP sends a *start_DSP* message to the rest of the members of the DS (using a multicast message, see section 3.3.5). Then, it waits for *vote_start_DSP* messages from other existing member of the group. Each *vote_start_DSP* message contains the last identifier that is believed to be assigned by the DS. After waiting for a sufficient period of time, the DSP processes all the *vote_start_DSP* messages and initialises its own state with the highest identifier received. This decision is then transferred to the rest of the group members with a *set_status_DSP* message.

At reception of a *new_group* message, the DSP creates a new identifier and sends a *new_group_prepare* message to the rest of the group members. The members will reply with a *new_group_vote* message to notify whether they are ready to accept the new identifier or not. If the rest of the group members accept the identifier, the DSP sends them a *set_status_DSP* message

and then, it assigns that identifier to the new group. If there is no agreement, the DSP will keep suggesting other identifiers until it succeeds. If there are no replies, the DSP will assume that the rest of DSP's have crashed and it will accept its initial suggestion.

In order to ensure the uniqueness property of new identifiers, the protocol guarantees that once a member has notified that it is ready to accept an identifier with a *new_group_vote* message, it will never suggest that accepted value as identifier for a new group.

During the execution of the protocol, different DSP's can act as co-ordinators trying to create new identifiers. This fact requires careful examination to avoid the possibility of deadlocks. A deadlock could be reached in the case when given the same initial value, the different co-ordinators suggested the same chain of values. In this case, all the co-ordinators would suggest the same identifier and none would get the value accepted, then they would all try again, suggesting the same next identifier that would be rejected as well, and they would keep suggesting values forever. In order to avoid such a condition, each DSP will suggest a new identifier by increasing the last suggested value by a random quantity within a safe interval.

The protocol just described guarantees the condition that every new identifier accepted is unique using *crash-fault* processors. This model ensures that if processes crash we are at least guaranteed that communication is reliable and each message is received inside their specific time constraints.

However, with only a few changes, the protocol could increase its probability of working properly in less reliable environments. In the case where the communication layer is likely to lose messages, a DSP that received no replies after sending a *new_group* message could not have the guarantee that it is the only DSP alive. In this case, if the DSP was suspicious that other DSP's could be alive, it could repeat sending *new_group* messages for a while. Another option would be to use a reliable multicast mechanism such as the one described later on.

However, none of these options would guarantee the uniqueness of identifiers in case of temporarily partitioned networks. To deal with such a problem, we could use very long identifiers randomly generated as suggested in the implementation of the FLIP protocol [Kaashoek et al. 93]. The use of this technique on its own does not completely guarantee the creation of unique identifiers, however, it leads to very scalable implementations while reducing considerably the likelihood of an identifier clash.

3.3.4 Intergroup Communication

Any group contacts the DS by communicating directly with the local DSP at a specific network port. Intergroup communication to a group other than the DS is achieved by sending a message to the primary process of the destination group. The sender process has to contact the DS to locate where the destination primary process is.

Each DSP keeps a routing table with the information of the groups that it owns – the groups whose primary process is located in the same node as the DSP - and another with information cached from other DSP's.

At reception of an address enquiry, the local DSP checks for the required information in its tables. If the information is not available, the DSP sends a *locate* message asking which DSP holds the information for that particular group.

Only the DSP that currently owns that group will reply with the information of the current address for the primary process. Once this information is received by the enquiring DSP, it will transfer it to the primary process that originated the enquiry. Both processes, the local DSP and the primary process, will hold this information in their cache for future reference.

From that moment, the sending group will be able to communicate directly with the primary process of the destination group.

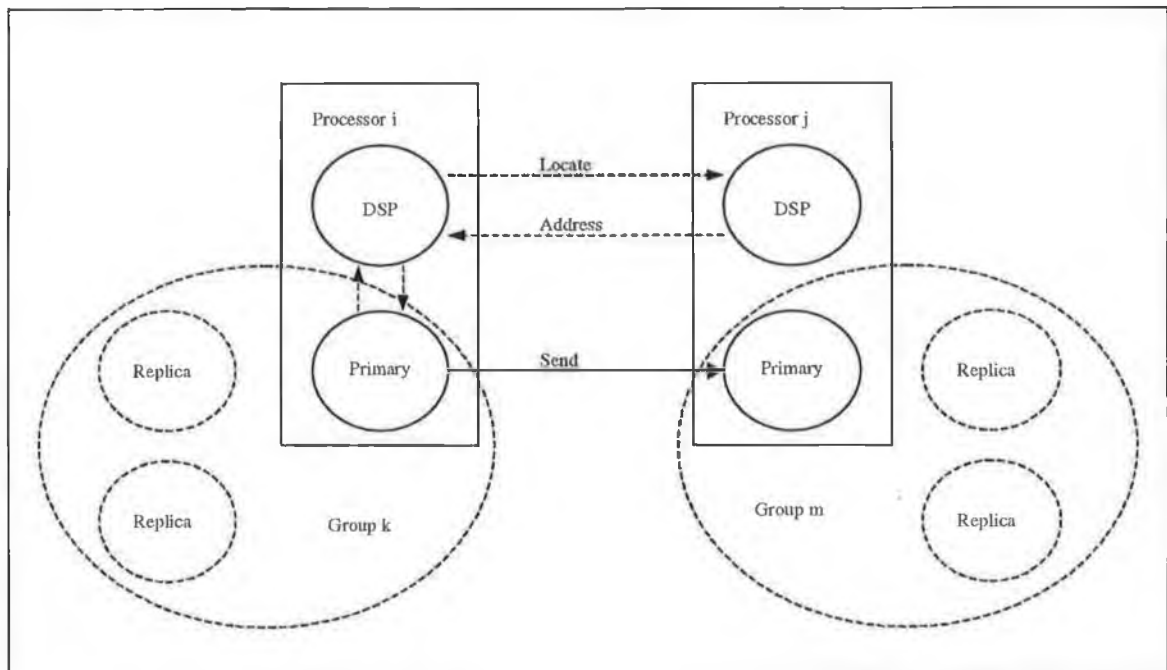


Figure 12 Scheme representing the use of the Directory Service to send a message from Group k to Group m.

This intergroup communication mechanism is used to implement a R^2PC call.

At the time when the destination primary process fails, it will be restarted in another node and the new primary process will inform of the new changes to its local DSP.

From that moment, the sender process will not receive any reply from its R^2PC requests. After a certain period of time, it will automatically flush the contents of its cache and it will enquire the local DSP again. Then, the local DSP will also flush the contents of its cache and it will send a new *locate* message.

If the failed nodes had been owned by the local DSP, this information would not be removed from the local routing table yet, however, the local DSP would send the *locate* message anyway. If, as reply of the *locate* message, the DSP realises that this information is already owned by another DSP, then the local DSP will remove the information from its local table.

The failure of a DSP could represent the disconnection of a group from the rest of the system. In order to account for this eventuality, the primary process of each group will periodically check whether the local DSP is still alive. If the failure of the local DSP is detected, the primary process will kill itself, hoping that the rest of the replicas will re-establish the state of the group.

This algorithm operates fine for reliable communication environments. In cases where the communication layer may omit messages, the primary process of each group should ensure that the DS holds the right information by transferring the current state of the group to the local DSP at regular intervals of time. As a matter of fact, this information is transferred every time that the primary process checks for the survival of the local DSP.

3.3.5 Intragroup Communication

In the case of the DS group, members communicate among themselves using normal multicast messages.

In the case of communication among members of groups other than the DS, the primary process of the group communicates with its members using a reliable multicast protocol.

Reliable Multicast

The design of R²PC relies on the use of a Reliable Multicast mechanism with the following semantics.

- Communication transparency:
 - Atomic message delivery: Any reliable message has to be received and processed either by all the members of the group or by none of them.
 - Ordering: Application-level ordering is enough to guarantee the proper evaluation of the algorithm.
- Naming Transparency: Group members have to be bind to a single name.
- Failure Transparency: After a processor failure, the protocol has to go through a recovery phase in which the group is rebuilt from the processors that are still alive. The protocol has to guarantee (1) that all the members in the rebuilt group receive all the messages successfully sent by the primary process of the original group before the failure and (2) that surviving members of the rebuilt group will receive all messages successfully sent by the primary process of the new group after the failure.
- Group Structure: Groups can be either open or closed. What is important is to have dynamic groups where processes can join or leave the group at runtime.

The Multicast Protocol implemented and used is based on the group communication system adopted in Amoeba [Kaashoek 92] with the following characteristics.

- Communication transparency:
 - Atomic message delivery: All or none.
 - Ordering: Total ordering per group.
- Naming Transparency: The *groupId* is used to identify a reliable multicast address to which all the group members are subscribed.
- Failure Transparency: As required.
- Group Structure: Closed and dynamic groups.

A reliable multicast message guarantees the delivery of the message to at least r members, r being the resilience degree of the message.

When a primary process has to send a reliable multicast message, it first sends a normal multicast message and waits for acknowledgement messages from the replicas. If r replicas have replied to the message, the reliable multicast message is completed, otherwise, the normal multicast is sent again.

This mechanism requires the use of message identifiers to distinguish among duplicated and new messages. As this mechanism has been designed to implement R²PC, we decided to adopt the identifiers described in section 3.2.2.

In our implementation another factor has been introduced, c , the **confidence degree** of a multicast call. This factor indicates the maximum number of times that a multicast call will be re-evaluated in case of not receiving acknowledgement from all the replicas required by the *resilience* degree. Once the message has been retransmitted c times, the transmission of the message will be considered complete, independently of the number of acknowledgements received from the replicas.

In cases where the communication layer is very reliable - such as current local area networks - it may make sense to define a low confidence degree. Section 3.3.6 will show that during the execution of a reliable multicast mechanism, the primary process tries to restart any replicas that may have failed. Therefore, if the primary process has been trying to restart replicas for a while

and is not getting enough replies, it may assume that the node has crashed completely and that it is not going to restart anyway. In such a case, it may be sensible to continue with the execution of the group, even if the resources available are below the minimum established by the resilience degree defined.

On the other hand, in cases where the communication layer is very unreliable or where there is a guarantee that any crashed nodes will be replaced, the system manager may require a very high confidence degree from the multicast mechanism.

3.3.6 Reliability

One method used to control the reliability of the system is the provision of a separate monitor service in charge of visioning what process groups may be in trouble. The main problem associated with the use of this method is its scalability. The more process groups that exist in the system, the more difficult it is to keep track of all of them.

R²PC does not require such a separate service because each group is responsible to guarantee its own survival. This solution is scalable because the number of members for each group is fixed by its resilience degree.

Fault Detection

The most common mechanism used to detect the failure of a process is the use of timers. The monitor service sends a message to the process that is suspected to have failed. If this process does not reply after a certain interval of time, the monitor assumes that its target process has failed.

This method can work fine for crash fault processors. However, in systems where messages may be delayed or omitted by the network communication layer, this method on its own would not work because the monitor can make mistakes by assuming that a slow process is faulty.

This type of mistake may be unavoidable. However, what is important is that once a process is considered to have failed, this information is propagated consistently among the rest of the processes of the system.

In order to provide such a consistent view, each group has associated with its state an *incarnation* number. The *incarnation* number of a group is incremented after recovery from a primary process failure. Each message sent is stamped with the *incarnation* number. Only messages

equal to the current *incarnation* number are processed. Messages from an old primary process are discarded and a kill message is sent back to the old primary process. When a primary process receives a message with a higher *incarnation* number, it realises that a more recent primary process exists and it kills itself.

As we have already mentioned in the introduction of this section, group members monitor each other to detect the failure of a component of the group.

The primary process checks the status of the replicas every time that it invokes the reliable multicast protocol (see section 3.3.5). The primary process uses a timer that triggers an automatic invocation of a reliable multicast NULL call after a certain period of inactivity.

Each replica uses a timer to control the activity of the primary process. If the replica has not received any message from the primary process after the limit period, it sends an *are_you_alive?* message to the primary process. If the primary process does not reply at all, the replica will activate the restart protocol to elect a new primary process.

The failure of the local DSP is considered to be equivalent to the failure of the node. In this case, each group periodically updates the local information of the DSP. In the case of detecting the failure of the DSP, the primary process terminates execution expecting that the group will be re-established somewhere else. Instead of committing suicide, the primary process could try to restart the DSP process. However, the current implementation of the protocol uses the first option for simulation purposes, in this way, the failure of a DSP can be used to simulate the failure of a node.

Restart & Recovery

On detection of the failure of a replica process, the primary process communicates with the DS to request the creation of another replica process in the system. Once the DS has created this new replica (possibly in a different node), the replica contacts the primary process to ask for the current status of the group and this information is transferred to the replica process.

The case of failure of the primary process is more complex because replicas have to reach an agreement to elect the new primary process of the group. The election protocol used in this case is based on the invitation protocol presented in [Garcia-Molina 82].

This protocol runs in two phases. The first phase is used to determine how many members are alive and to elect a coordinator for the second phase. A replica that triggers the restart algorithm sends a *restart* message to the rest of the members and takes the initial role of the coordinator. On reception of the *restart* message, replicas reply with a *vote* message indicating what is their current state and wait for a *decision* message from the coordinator. If a coordinator receives a restart message from another coordinator, the one with the lowest network address becomes the coordinator of both. At the end of this invitation phase, there is only one coordinator left that knows exactly how many members are alive and their current state.

In the second phase, the coordinator elects as new primary process the one whose state is more up to date - if the coordinator notices that the previous primary process is still alive, that one will be elected as primary, otherwise the candidate will be chosen randomly among the more up to date ones. If one of the members does not have the complete status of the group, the coordinator sends it the missing messages. On reception of the *decision* message, the process that has been elected primary starts the recovery phase (see section 3.2.3) while the rest of the members remain replica process.

This algorithm operates correctly for *crash fault* processors. In environments with *omission* failures, replicas that may not have been acknowledged during the first phase of the protocol may restart the execution of the first phase of the protocol again, delaying the final result. However, in case of temporary network partitions where the multicast messages may not reach all the replicas, two different members may become primary processes. Once the partition problem is solved, the primary processes will detect the existence of each other with the same incarnation number. The one with the lowest network address remains as the primary process and the other is killed. Unfortunately, such a situation may lead to inconsistencies when the primary process that is killed has already processed calls from other groups.

3.3.7 Implementation Details

The run-time part of R²PC provides each process with a number of threads to carry out its work. For the client part, the system sets up a worker thread that carries out the computation of the main part of the program. For the server part, a different worker thread is set up to service the calls from each different group that is communicating with the current process. There is also one thread in charge of the *intragroup* communications with other members of the same group as the

current process. All these worker threads actually communicate with the rest of world via the sender and receive threads (see Figure 13).

Primary and replica processes share the same source code. When a replica process takes the role of the primary, the active thread in charge of the *intragroup* communications takes care of the recovery phase and, once the recovery is over, it starts creating the necessary threads to provide service to the groups that communicate with it.

The timers used to monitor the survival of each process base their functionality on the current *networkLatency*. This parameter is automatically adjusted by the system in order to adapt to traffic variations. The messages used by the monitor part are assigned with the highest priority in order to avoid any extra delays.

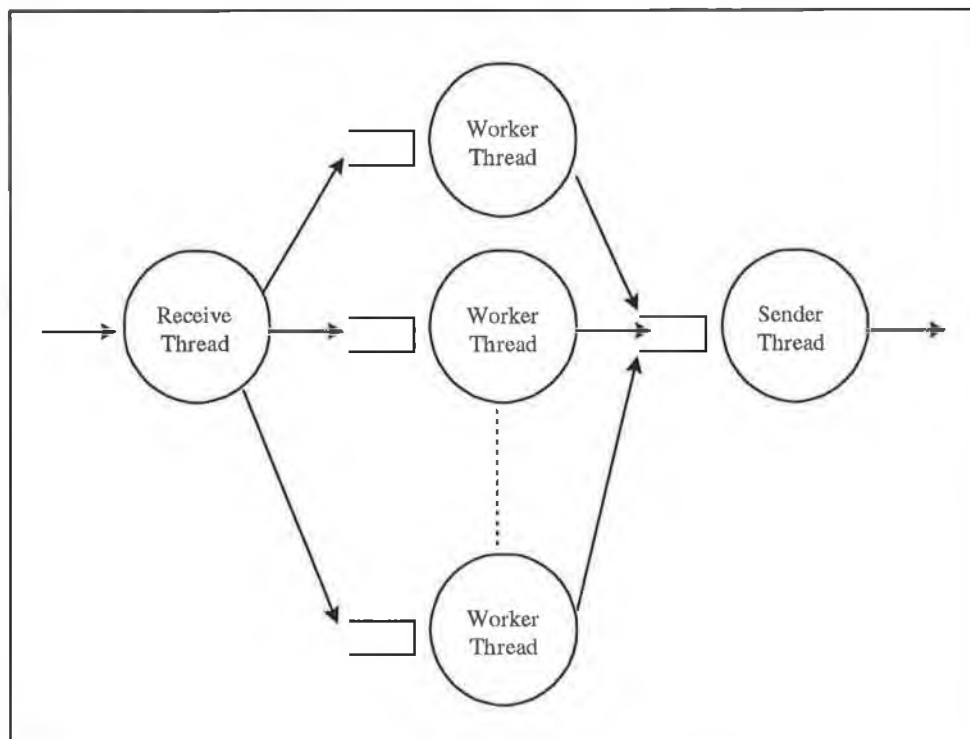


Figure 13 Thread model used in the R²PC processes

The system manager can set up the initial value assigned to the *networkLatency* parameter and to other parameters of the system through the use of the configuration files described in the Appendix B.

3.4 Conclusions

This chapter has outlined our approach to construction of a Reliable Remote Procedure Call (RRPC) service. It also presented the implementation details of the group communication service used to implement our model. The rest of the dissertation is devoted to applications of this model and examples that illustrate how these tools can be applied.

Our approach differs from existing approaches to the provision of RRPC services described in section 2.1 The main differences being the transparency of the resulting system and the creation of a scalable and efficient system that permits the reliable service of concurrent calls.

In terms of transparency, our system supports:

1. Use of nested calls (a recognised factor for the creation of complex distributed systems).
2. Use of threads.
3. Provision for recovery of any process of the system.
4. Use of transparent recovery mechanisms (although the use of checkpoints has not been covered in the current implementation).

The system can be implemented efficiently in terms of the following:

1. Use of a fast reliable communication protocol (the number of messages required for each RRPC is 2 point to point + 2 reliable multicast).
2. Concurrent service of different calls.
3. Provision of a scalable solution (permitting the creation of many different groups with different degrees of resilience).
4. Provision of fast recovery times (replicas are active and ready with all the information necessary to restart).

The main characteristics of the resulting system can be summarised as follows:

- There are no single processes. Any process forms part of a group.
- A new process group is automatically created when a new process is executed. The process is automatically replicated in the available platforms according to the

configuration parameters set for the system. Each process group is uniquely identified in the system with a single address.

- A group has two member types, primary and replica processes. The primary process is the entity that performs all the computation of the group. The replica processes are passive entities that receive information about the computation being evaluated in the primary process of the group.
- Each group has one and only one primary process active at any time. The members of each group ensure automatically that this condition is accomplished. The primary process verifies that the replicas are alive and the replicas check that the primary process is alive. We assume that the components of the system can only fail by crashing (fail-crash processors), although this requirement could be relaxed as shown in [Schneider 84]. Any failing component of the group is discarded and, if possible, restarted elsewhere. A component failure is detected using a timeout mechanism following the principles of the Virtual Synchrony model developed for Isis [Birman et al. 94]. A group fails only when all its members fail simultaneously.
- Any group can act as client, server or both (permitting the use of nested calls). Our protocol ensures that the restart of a primary process will evaluate all the calls executed from the last checkpoint in the same order and obtaining the same results that had been previously obtained in the execution of the previous primary process up to the moment of the crash. In our system, the provision of fault-tolerance will be guaranteed if any process in the system is deterministic upon reception of the same results from the execution of the same sequence of R^2PC calls. (Note that although the sequence of results to receive does not have to be deterministic beforehand, the system will guarantee that the execution of a R^2PC is considered a transaction, once a result has been obtained for a particular call, this result is not going to change anymore).

Example Application

R²PCLinda: Adding fault-tolerance to an existing distributed system

4.1 Introduction

This chapter presents the design and implementation of R²PCLinda, a fault-tolerant version of PCLinda.

PCLinda [Manso 96] is a tool designed to develop parallel and distributed applications over a network of Personal Computers based on the Linda coordination language [Gelernter 85]. Linda is a powerful set of operations that can be included into a computational language to incorporate time and space distribution.

Taking PCLinda as application example of our model, we decided to tackle the fault-tolerant implementation of a generic tool to create distributed programs. In doing so, we also proved the broad spectrum of fault-tolerant distributed applications that can be constructed with our R²PC protocol.

The implementation of R²PCLinda shows the flexibility of the R²PC system designed by having facilitated the creation of a fault-tolerant version of an existing complex distributed system with practically no extra effort.

The next section presents a brief description of the implementation of PCLinda but for exact details and further explanations see [Manso 96]. Then, we present the main changes required to adapt the initial system and obtain a fault-tolerant version, R²PCLinda. Finally, we present a summary of the tasks to do when including fault-tolerance into an existing distributed system using R²PC.

4.2 PCLinda

4.2.1 Overview of the System

The implementation of PCLinda is based on the use of the RPC model. The system represents the existence of Tuple Space as a collection of *local tuple space* processes that interact among themselves to achieve the properties of the model. Figure 14 shows the basic components of a PCLinda application.

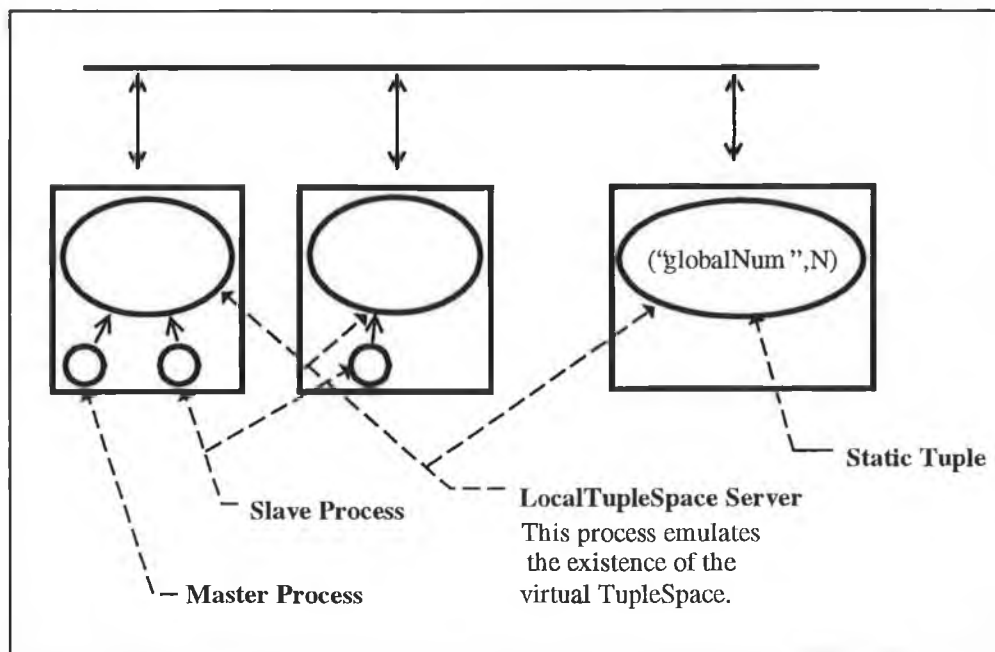


Figure 14 Overview of the PCLinda System

The execution of a user program is represented as the execution of a collection of different processes, the *master* process runs the *main* part of the program and the *slave* processes execute the *eval* calls of the program.

The Tuple Space is represented by the simultaneous execution of *local tuple space* servers. Each server is a *daemon* process executing on each processor of the system. When a user process invokes a Linda operation, it invokes a library call that communicates with the *local tuple space* server executing in its machine. This server evaluates the required function by exchanging

information with the other servers in the system and finally, the result is returned to the user process (if any result was required).

4.2.2 Implementation of PCLinda

A Linda program interacts with the Tuple Space through the use of the operations: *out*, *eval*, *rd* and *in*. These last two return the required tuple to the Linda program (a description of the semantics defined for these operations can be found in section 1.3.3).

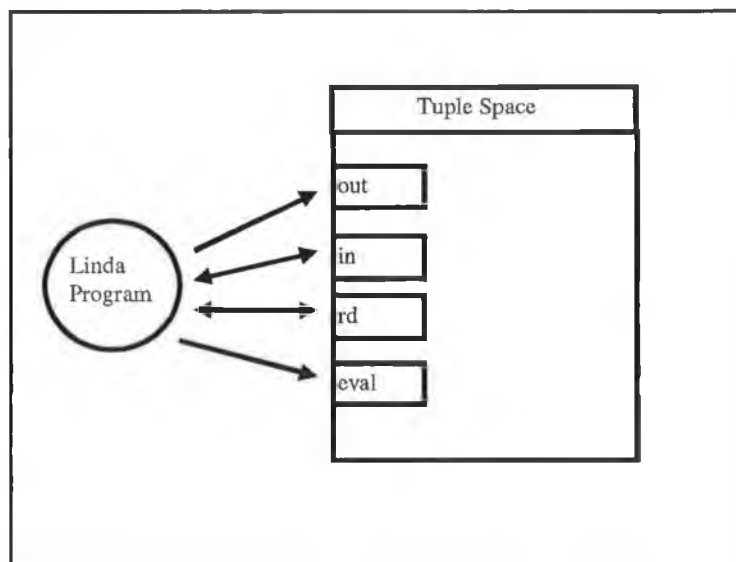


Figure 15 Representation of the Tuple Space Object

The refinement of the context diagram of the *tuple space object* is shown in Figure 16. This scheme represents the Tuple Space as a distributed object formed by the interaction of different *local tuple space* objects.

The diagram describes the implementation of each of the internal functions of the *tuple space* object. For example, the invocation of an *eval* function leads to the following succession of events:

- Firstly, the user program invokes an RPC to call the *eval* function in its *local tuple space* server.
- Then, the *eval* function calls the *live tuple decisor*. This object decides where to evaluate the live tuple.

- The *live tuple decissor* may decide to evaluate the tuple locally or in another *local tuple space*, in which case it communicates its decision to the other *local tuple space*.
- The *local tuple space* creates a live tuple. At this point the RPC finishes its execution and the control is brought back to the user program.
- Once the live tuple finishes its evaluation, the *localEval* function will invoke an *out* to insert the resulting tuple as a static tuple.

The implementation of the *eval* function requires the use of a nested RPC call when the *live tuple decissor* decides to invoke the *localEval* function on a different *local tuple space* server.

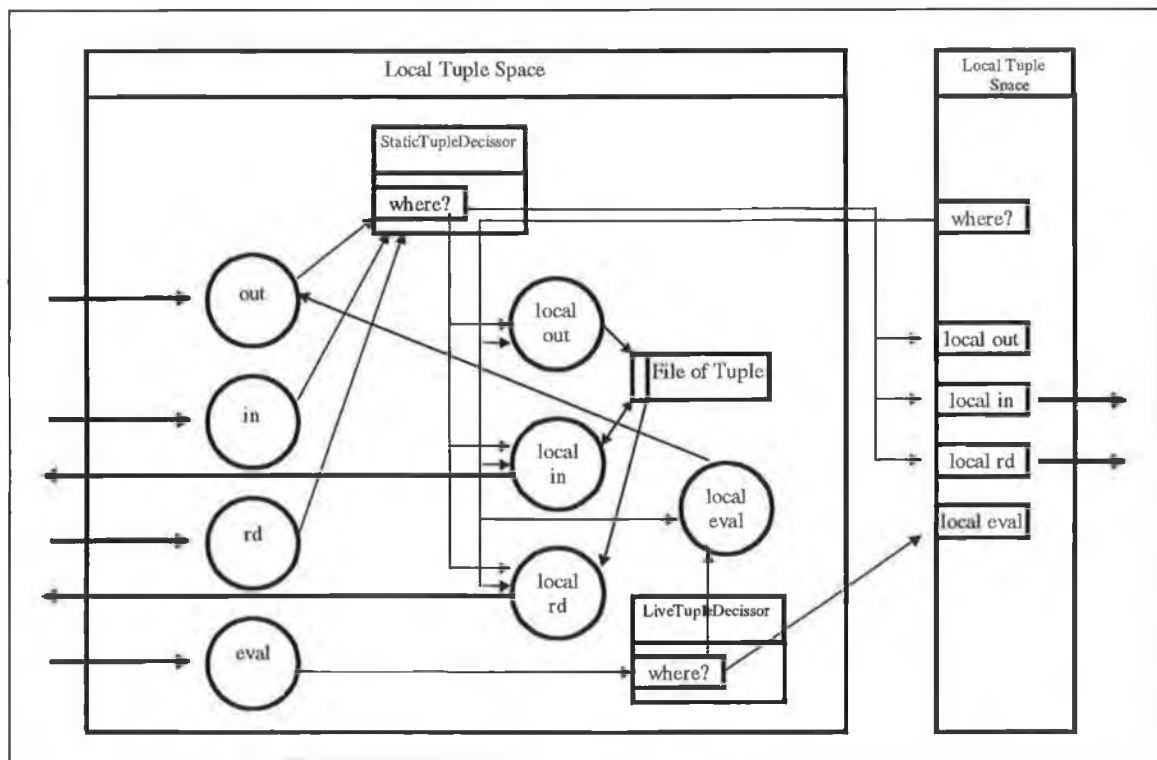


Figure 16 Refinement of the Tuple Space Object

The diagram shows graphically the implementation of the services offered by the *tuple space* object.

- *Out*: This method firstly demands the *static tuple decissor* (see section 4.3.2) *where* does it have to insert this tuple. If it has to be inserted in the same processor, it just executes a *localOut*

call, otherwise, it issues a remote *localOut* call in the processor where the tuple has to be inserted.

- *Rd*: This method also asks the *static tuple decisor* where it has to look for the tuple that matches with the template executing the *localRd* in the appropriate processor.
- *In*: This method is practically the same as *Rd* with the difference that it will issue a *localIn*. The *localIn* method will remove the tuple from *tuple space*.
- *Eval*: This method addresses the *live tuple decisor* (see section 4.3.2) to ask *where* does it have to evaluate the tuple and then, it executes a *localEval* in the designated processor.
- *localOut*, *localRd*, *localIn*: These methods do not use the *static tuple decisor*. They insert or remove tuples from the list of tuples that make up the *Local Tuple Space*.
- *localEval*: This method creates a new process that will execute the function to evaluate.

What is important to note in the diagram is the fact that any communication among different *local tuple space* servers requires the use of nested RPC calls.

The implementation of PCLinda also included the creation of a precompiler that translates a PCLinda C++ program into two C++ modules including calls to the methods defined in the PCLinda library. The first module created contains the code for the *main* part of the program and the second one is the code for the *eval manager* program (see [Manso 96]).

4.3 R²PCLinda

Fault-tolerance is introduced in PCLinda by substituting the original RPC protocol with the new R²PC model. In order to use R²PC, a distributed system has to be adapted syntactically and semantically.

4.3.1 Group Addressing

R²PC calls have to be directed to process group identifiers. As *groupIds* are normally created dynamically, there is no way that an application can learn what is the *groupId* of a group at compilation time. In order to solve this problem, R²PCLinda introduces the implementation of a Name Directory Service (NDS). This service can be accessed at a static *groupId* address known to all the process groups of the system.

The NDS provides a mapping between logical names and *groupIds*. At start time, once the DS (see section 3.3.2) assigned a *groupId* for an application, this application publishes its address through the NDS service.

In R²PCLinda, each local tuple space server publishes its group address under the logical name “localTS”. The first time that a Linda application requests the *groupId* for its “localTS”, the DNS tries to provide the application with the *groupId* of the “localTS” service whose network address is the same as the process making the request. However, once a binding has been established it will not be changed.

For example, suppose that we have two processors, processor *i* and processor *j*.

At start time two process groups will be created, *localTS_i*, whose primary process runs in processor *i* and *localTS_j*, with primary process running in processor *j*. When *client_i* is created in processor *i*, it is binded to *localTS_i*. Now, if the primary process of *client_i* crashes and resumes execution in processor *j*, *client_i* will not change the binding to *localTS_i*, it will keep using *localTS_i* as its “localTS”.

4.3.2 Determinism

The behaviour of fault-tolerant programs created with R²PC has to be determined by the results obtained from the evaluation of the Reliable Remote Procedure Calls executed. This condition is necessary and sufficient to guarantee the recovery of any failing process (see section 3.2.2).

The decisor objects are the main sources of non-determinism for the PCLinda system.

Static Tuple Decisor

The *static tuple decisor* influences directly in the Tuple Space distribution. The amount of communication exchanged among *local tuple spaces* will depend on this distribution, which implies that the implementation of the *tuple decisor* directly influences the efficiency of the resulting system.

The problem can be presented using a mathematical notation [Shekhar 93]. Let:

- $in_set(P)$: set of all tuple types that the procedure *P* can refer/remove.
- $out_set(P)$: set of all tuple types that the procedure *P* can place in Tuple Space.
- $access_set(P) = in_set(P) \cup out_set(P)$. These are all the tuples related to procedure *P*.

If we consider that each process *P_i* is executing in a different node, the most efficient distribution corresponds to the case where for each process *P_i*, the $access_set(P_i)$ is placed on the same node

where P_i is executing. Unfortunately, this is possible only if there is no other process P_j such that $\text{access_set}(P_i) \cap \text{access_set}(P_j)$ is not a null set. In practice, such a distribution is not possible as most of the applications do not fall into the above category. Consequently, other distribution schemes have to be used:

- **Random Selection of a Node:** A first possibility is to decide randomly where to place each tuple belonging to the $\text{out_set}(P)$. This scheme may result in a large amount of communication, because the process P may have to send a tuple corresponding to the $\text{out_set}(P)$ to another node and then, may have to check all the nodes to find a tuple belonging to the $\text{in_set}(P)$.
- **Operator Partitioned Scheme:** In this scheme the tuples are distributed according to the operator used. For example, all the tuples belonging to the $\text{out_set}(P)$ can be mapped to the node i and all the requests belonging to the $\text{in_set}(P_i)$ to all the nodes. Another possibility is to reverse the previous scheme. That is the requests belonging to $\text{in_set}(P)$ are diverted only to node i and all the tuples belonging to $\text{out_set}(P)$ are sent to every node. There are also many variations of these two schemes.
- **Hash Based Schemes:** In this case, each tuple type is mapped onto a node according to a predefined scheme. This method reduces communication because it fixes each tuple to a node at compilation time. Usually, the most efficient systems are the ones that define the mapping by analysing the dependencies of the application.

For the implementation of R²PCLinda, we need to define a system whose behaviour is determined by the results of the R²PC calls received. For this reason, the use of a random scheme would have been totally inappropriate.

The *static tuple decisor* finally adopted uses a simple hash-based scheme. The location for a tuple is determined from the value of the first parameter of the tuple. The node where a request tuple can be found will be determined in all the cases but one, when the first parameter of the tuple is a template. In this case the system will have to check for the requested tuple in all the nodes. However, once a tuple has been transferred to one process, this decision will be determined by the system. In case of re-execution, the same collection of tuples will be transferred to the same process.

Live Tuple Decisor

The *live tuple decisor* has to decide where to evaluate new processes. This object also has an important influence in the efficiency of the resulting system. Two main factors must be considered when making the decision:

- **Minimise the communications:** Following the mathematical descriptions presented in the previous section, we have found that the most efficient distribution corresponds to the case where for each process P_j such that $\text{access_set}(P_j) \cap \text{access_set}(P_i)$ is not a null set, then P_j is placed in the same node as P_i . Unfortunately, for most of the applications this will imply execution of all the processes P_i together on the same node. In other words, the execution of our program will not be parallel, the program will merely execute as a set of processes running concurrently on the same machine.
- **Maximise the use of the computer resources:** The load of the system has to be kept as uniform as possible. In this way all the nodes will have to do the same amount of work and the use of the system will be maximised.

The problem is that these two factors are normally divergent. In general, the more we distribute the work to do, the more communications we get. Therefore, a good option is to try to get the middle point by distributing the processes as much as possible but using a scheme related to the one used to distribute the static tuples. Some implementations use dynamic load balancing algorithms that distribute the processes according to the current load of the system. However, this solution is not deterministic.

In R²PCLinda, the *live tuple decisor* uses a deterministic solution that focuses on load distribution by placing each *eval* process cyclically around the different nodes of the system.

The invocation of an *eval* procedure causes the creation of new process groups, one for each function to evaluate.

If the process that invoked the *eval* procedure fails and is restarted in another node, the recovery process will re-execute all the calls that this process had executed since the last checkpoint in order to re-establish the state of the group. However, it is important to note that the re-evaluation of the *eval* invocations should not create any new process groups because they had already been created the first time that the *eval* procedure was invoked.

For this reason, the implementation of an *eval* procedure is implemented as the successive invocations of different *execute* procedures, one invocation for each function to evaluate. Each *execute* procedure is defined as a system persistent call. The implementation of R²PC had to be modified to guarantee that persistent calls are evaluated only once in the system's lifetime. To do so, the original implementation was modified to guarantee that only system's non-persistent calls would be re-executed during the recovery process.

4.4 Conclusions

In this chapter we have taken an existing distributed system of high complexity and easily added fault-tolerance properties into it by substituting the original RPC calls of the application by the R²PC calls defined in our model.

The process required to adapt an existing system has been quite simple.

First, we had to analyse the functionality of the existing system and try to eliminate any source of non-determinism not coming from the result of Remote Procedure Calls. The introduction of such a condition is necessary to guarantee the consistent recovery of crashed processes.

Once eliminated any undesirable sources of non-determinism, we transformed the RPC invocations into R²PC invocations addressed to process groups. To do so, the design and implementation of a Directory Service was included to map any logical name into a group identifier.

This chapter has demonstrated the flexibility and power of the resulting R²PC protocol as a mechanism to develop fault-tolerant applications.

System Evaluation

Study of the Efficiency and Reliability of the New System

5.1 Introduction

This chapter presents the results from the evaluation of different problems, each one specifically designed to test a particular characteristic of the system.

In order to evaluate the efficiency of the resulting system at the application level, different parallel paradigms have been implemented. These paradigms have been implemented in R²PCLinda because this environment simplifies the creation of parallel applications. Furthermore, the use of R²PCLinda has permitted the evaluation of reliable distributed applications that include the use of nested calls.

The source code used to create the examples presented in this chapter is provided in Appendix B. The R²PC protocol has been implemented on Windows'32 using Windows Sockets 2 as network programming interface. However, any interface that provided support for threads and a multicast service could had been used.

The evaluation of the system has been performed on a self-contained LAN of Pentium-Pro 180 processors with 32 Mbytes of RAM memory running under Windows NT and connected by a 10 Mbps Ethernet.

Each value given in this section is the average resulting from the evaluation of five different runs. Appendix C presents the complete list of results obtained.

5.2 Efficiency

5.2.1 Transmission Cost of R²PC calls

The first test is designed to study the cost of the transmission of a R²PC call.

In this test, a distributed application consisting of two process groups was designed, one acting as client and the other as server. The client group evaluated 200 R²PC null calls on the server group. The server group did not do any processing, it merely received the parameters. In one case, there were no parameters and, in the other, the client sent approximately 2.5 Kbytes of data that were transmitted in five message blocks of 512 bytes.

In order to study the scalability of the reliable system we compared the results of evaluating the null calls using different number of replicas.

The results presented on page 74 show that the current implementation of the reliable multicast mechanism is quite inefficient. In theory, results obtained from similar implementations of such protocol give more promising results (see [Kaashoek 92]). Unfortunately, our current implementation did not manage to get a more efficient implementation without compromising the reliability of the algorithm. We believe that one of the main problems for our result arises from the slow speed of the thread switching mechanism available in the environment that we are working with. However, we expect to get better performances in future versions of our system.

On the other hand, the implementation of our reliable multicast shows good scalability, providing a regular execution independent of the number of replicas being used.

Note that the evaluation of the RPC protocol is constant due to the fact that this mechanism does not use any replicas.

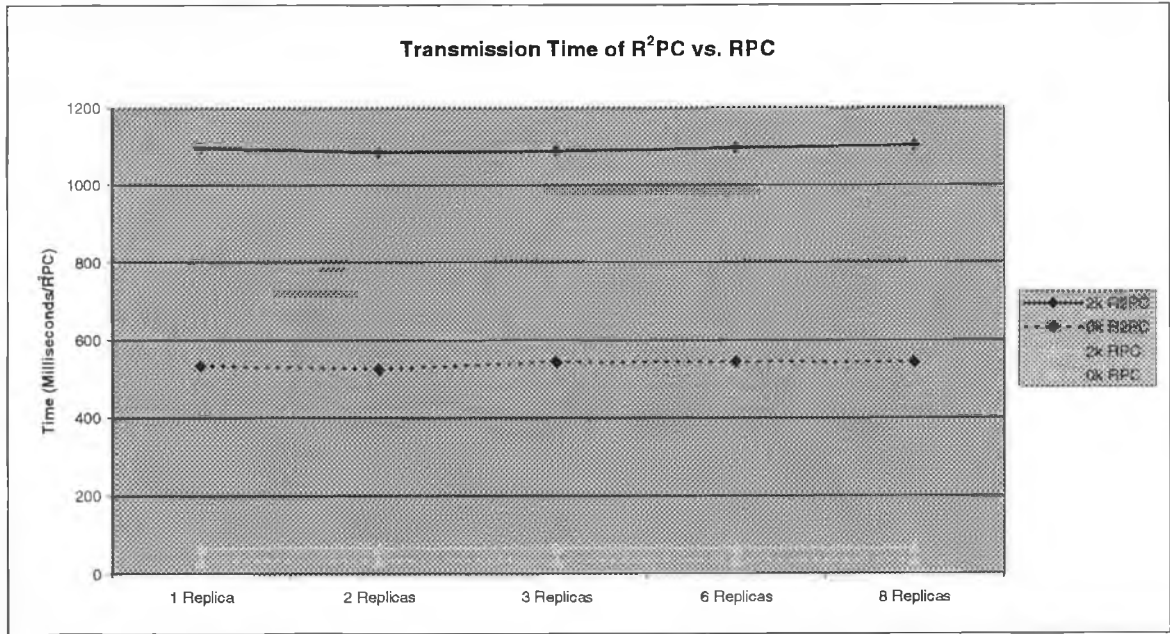


Figure 17 Response times for the transmission of R²PC and RPC calls

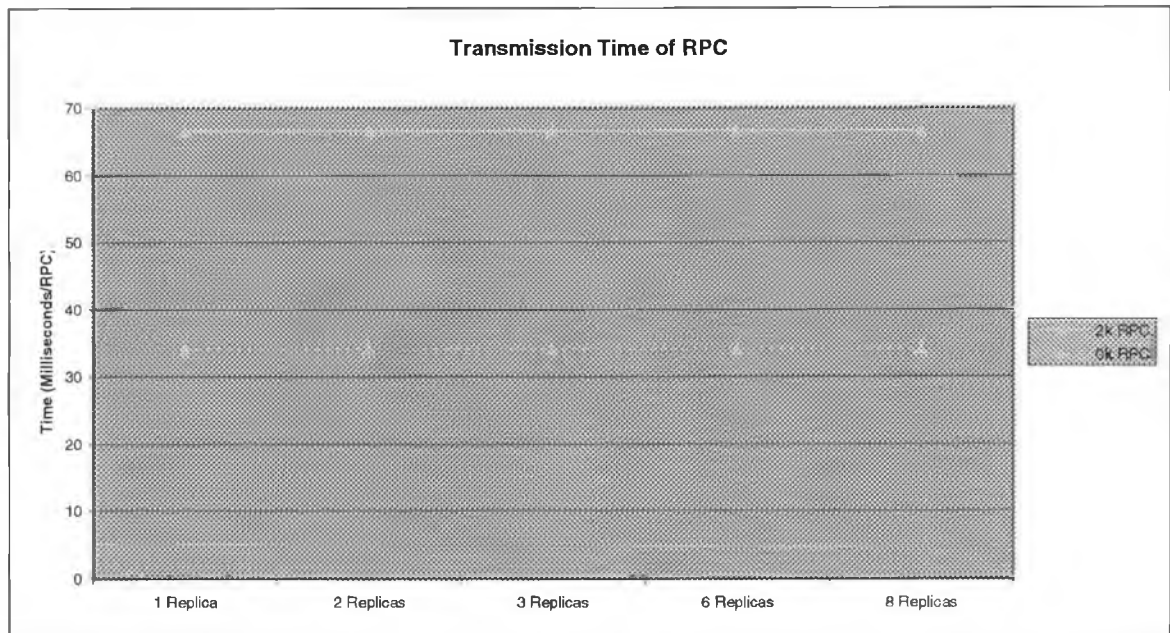


Figure 18 Response times for the transmission of RPC calls

5.2.2 Inherent Parallelism of R²PCLinda

The second test is designed to study the inherent parallelism of the resulting R²PC system at the application level. This test computes numerically the value of π by calculating the area under the curve represented in Figure 19.

This is done by splitting the area to evaluate into thousands of vertical strips and then, summing up the areas of each strip to obtain a numerical approximation to the value of π .

The code for this application is based on the algorithm presented in [Lewis et al. 92]. The main program of the R²PCLinda application creates as many *eval* process workers as active processes in the system and then, it splits the area to evaluate among the *eval* workers. Each *eval* process receives the area to evaluate through the “start” tuple, calculates the area and returns the result through the “pi” tuple. The main program collects all the result tuples and adds all the results to obtain the final value of pi (Appendix B.2 lists the source code used for this application).

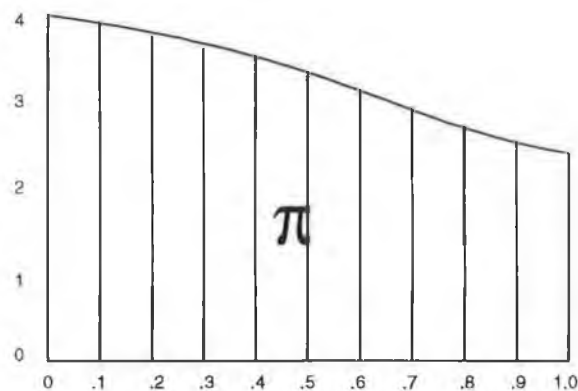


Figure 19 Pi equals the area under the curve $4/(1+x^2)$

As shown on page 76, the evaluation of the unreliable version of Pi presents a linear speedup. This is logical due to the completely parallel nature of the program. The results show that the speedup is remarkable only from the moment that there are more than two nodes in the system. This is due to the fact that the main process of the application is executed on a node on its own

for all the evaluations except one, the case when there is only one node. With only one node, the main process of the application is executed in the same node as the *eval* process.

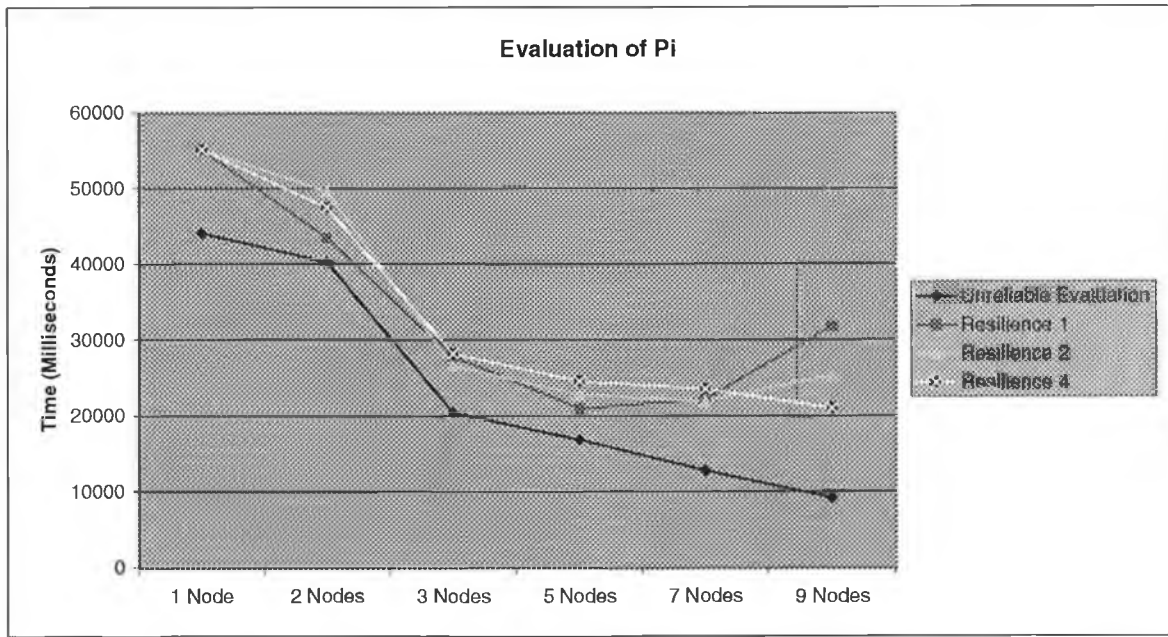


Figure 20 Evaluation of Pi for Unreliable Evaluation vs. Evaluation with Resilience 1 to 3

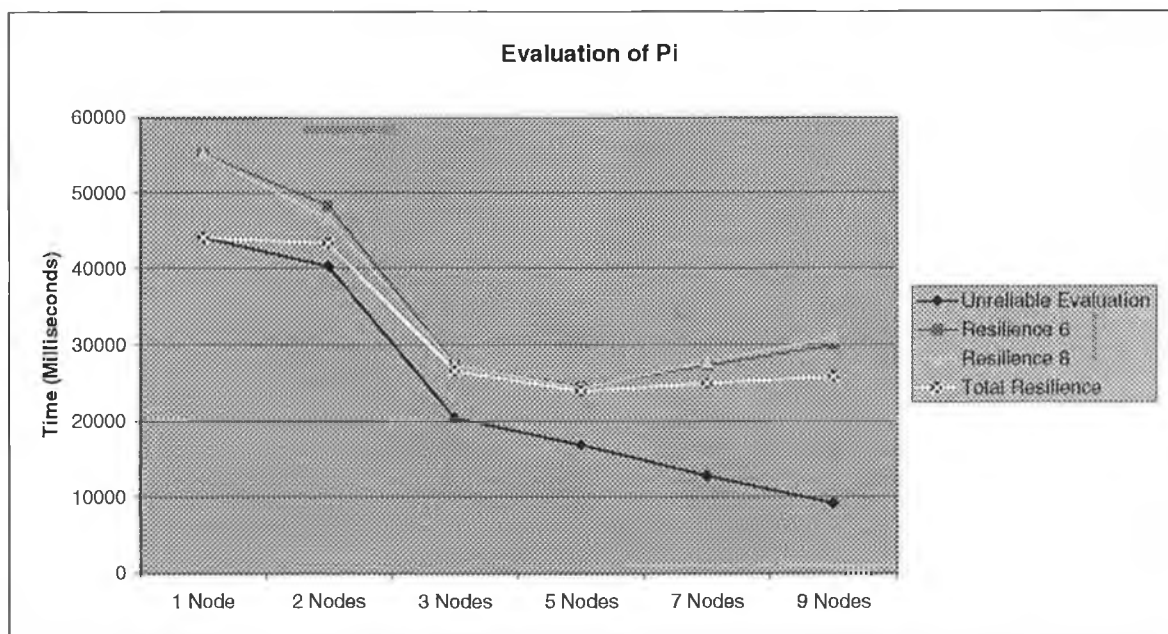


Figure 21 Evaluation of Pi for Unreliable Evaluation vs. Evaluation with Resilience 6, 8 and Total

The cost for the evaluation of all the reliable versions of the algorithm is greater than the unreliable version except for one case – when the system uses *Total Resilience* for only one node. In this situation, the system behaves exactly the same as for the unreliable case. The reason being that when the system uses *Total Resilience*, it tries to start replica processes in all the nodes existing in the system. However, if only one node is available, no replica processes can be started because the primary processes are already using the network ports assigned for the group.

However, following this reasoning the rest of the reliable cases for only one node should also take the same time as the unreliable case. Why do they take longer? They take longer because in the cases where the resilience degree is greater than zero (and not *Total*), the reliable multicast mechanism tries to restart replica processes and then waits for replies from a number of replicas defined by the *resilience* degree for a certain period of time according to the *confidence* degree defined (section 3.3.5).

Unless specified to the contrary, the tests presented have a default *confidence* degree value of 2, meaning that the reliable multicast algorithm will try to restart replicas twice before returning control to the application.

In theory, according to the results obtained in section 5.2.1, the evaluation of the reliable versions should follow a parallel line to the result obtained from evaluation of the unreliable version. This should be expected due to the fact that the cost of a R²PC call is linear, independent of the number of replicas being used.

This is the case for resilience 4. However, the rest of the evaluations tend to take longer as the number of nodes of the system increases. This is not surprising for evaluations with a high number of replicas, such as the evaluation for resilience degrees 6, 8 and *Total*. In these evaluations, the number of processes executing in each node is quite substantial, consuming a great amount of resources from the system – especially virtual memory. However, this factor does not justify the highest evaluation time required for 9 nodes and resilience 1. The only plausible explanation for this result is to consider this value as a statistical outlier, which came up as result of the relatively high variance obtained among the different sets of results obtained. In order to minimise the effects of such variance, each value was taken as the mean of the values obtained in five different runs (see Appendix C).

In order to devise the reason why evaluations with high number of replicas and nodes require the use of many resources, we can consider the case of using 9 nodes with resilience 8. In this case we would have 20 processes executing in each node. Each node executes a *DSP* provider process (section 3.3.2) and the replica/primary processes for each active group in the system. In this example there are 19 active groups. There is 1 *nameServer* group (section 4.3.1), 9 *localTupleSpace* groups (section 4.2.1), 1 *mainProcess* group and 8 *evalProcess* groups (section 4.2.1).

5.2.3 Parallelism with Communication

The previous example showed the potential parallelism of the R²PCLinda system. The original problem was split into N different tasks that could be solved completely independent from each other. The *eval* processes did not need to exchange information among themselves to evaluate their task. The only communication involved was the transmission of the tasks and results among the main program and the *eval* processes.

However, this type of situation is an ideal one. Very few parallel programs can be structured in such a neat manner. Most parallel programs require an active exchange of information among the entities that perform the parallel processing (named *eval* processes in R²PCLinda).

This section introduces an example in which the *eval* processes require the exchange of information in order to solve a problem.

The problem is the development of an algorithm to count all the prime numbers between 1 and N . The solution is based on the work presented in [Carriero et al. 90].

The program derives parallelism from the fact that, once we know that the number k is prime, we can find the prime numbers between $k+1$ and k^2 .

A simple solution to the problem could create as many *eval* processes as numbers to evaluate, each *eval* process could determine whether its index is prime using the algorithm described in Figure 22.

```

is_prime(index)
{
  limit=sqrt(index)+1;
  for (i=2; i<limit; i++)
  {
    rd("primes", i, ?ok);
    if (ok && index%i==0)
      return 0;
  }
  return 1;
}

```

Figure 22 Initial Solution Prime Numbers. Code to execute by *eval* processes

The main program would be in charge of creating the *eval* processes and counting the number of primes found and stored in Tuple Space (TS). Its code is presented in Figure 23.

```

main()
{
  for (i=2; i<N; i++)
    eval("primes", i, is_prime(i));
  count=0;
  for (i=2; i<N; i++)
  {
    rd("primes", i, ?ok);
    if (ok)
      count++;
  }
  printf("result is %d", count);
}

```

Figure 23 Initial Solution for Prime Numbers. Main program

Although this initial solution evaluates the prime numbers correctly, it is extremely inefficient. It creates too many *eval* processes, each one in charge of producing very little processing.

A more efficient solution can be obtained by creating only one *eval* process in each node available in the system. This is the approach taken for the algorithm to approximate the value of π .

The number of prime numbers to evaluate has been divided into blocks of a fixed size to be processed by the *eval* processes. Each block to evaluate is a different task to do. For this algorithm, tasks are assigned in order. The lowest block is assigned first, then the next-lowest block and so forth. Tasks are assigned using the “next task” tuple as shown in Figure 24.

```
for (;;)
{
in("next task", ? start);
out("next task", start + GRAIN;

// find all the primes from start to start+GRAIN
}
```

Figure 24 Assigning the tasks to be done by the *eval* processes

However, at the moment our algorithm is still very inefficient because it requires a large quantity of communication. This is because every access to TS may require communication among different nodes. To achieve better efficiency, the number of accesses to TS should be minimised.

An initial refinement to reduce the communications is to decrease the number of elements stored in TS. Instead of storing the complete range of numbers from 1 to N specifying whether they are primes or not, only the prime numbers that are found need to be stored in TS. Each entry takes the following form:

(“primes”, i , <ith prime>, <ith prime squared>)

The square of the i th prime is stored along with the prime itself so that workers can simply read instead of having to compute each entry’s square.

After introducing these last modifications the algorithm specifies that worker processes grab tasks to evaluate all the primes within their assigned interval. To test for the primality of k , they divide k by all primes through the square root of k . Worker processes find these primes in TS.

Another easy refinement that can be introduced to our algorithm is that, once a worker process refers to TS to find the value of a prime number that has been found, it copies this value into its local memory to avoid referring to TS in future evaluations. In the algorithm presented in the Appendix B.3 workers use two local arrays, *primes* and *p2* to store the values of the prime numbers found and their square value respectively.

Another optimisation introduced in the algorithm presented in Appendix B.3 is the fact that only the prime numbers from 1 to \sqrt{N} are stored in TS. Process workers only need to return the number of prime numbers found in their interval using the tuple:

("result", task, numberPrimesFound)

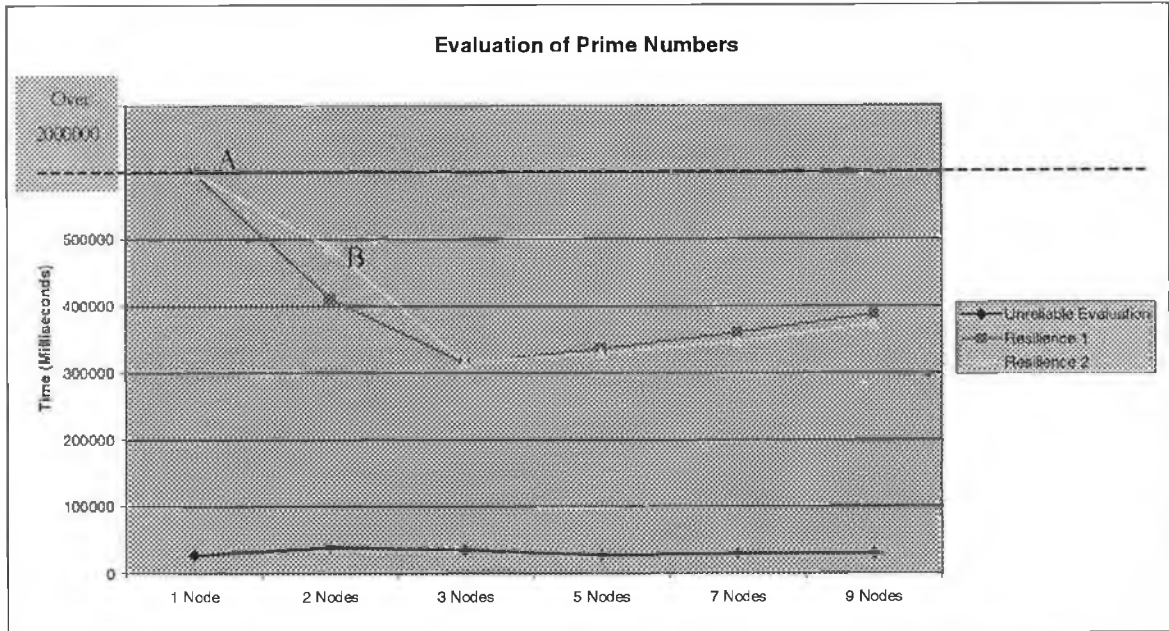


Figure 25 Evaluation of Prime Numbers for Unreliable Evaluation vs. Evaluation for Resilience 1 and 2

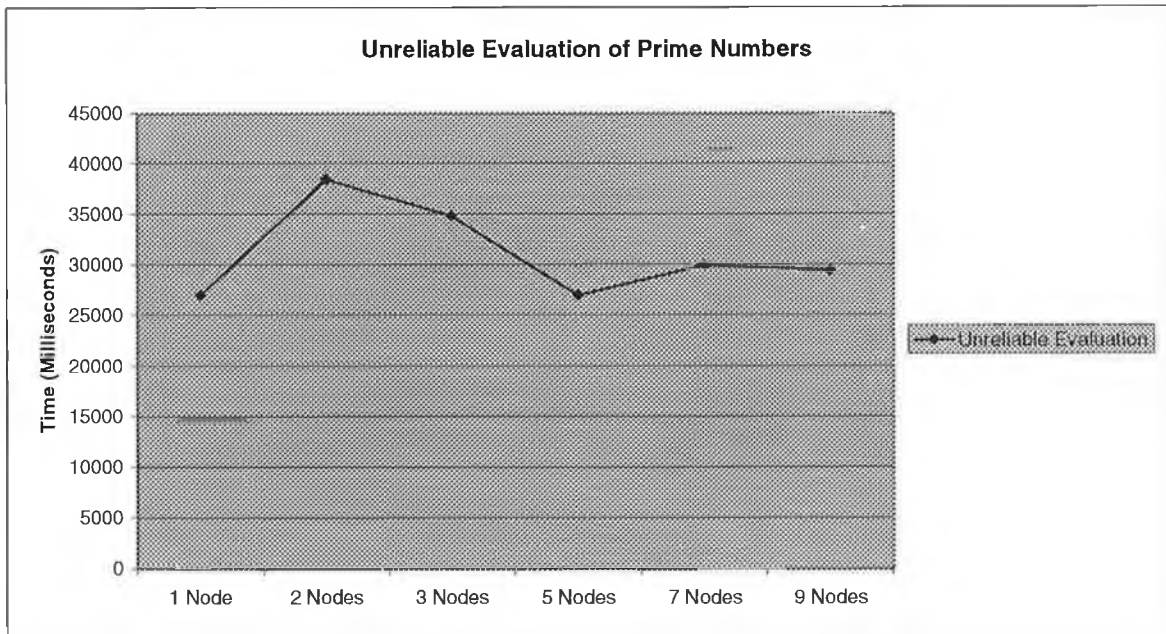


Figure 26 Unreliable Evaluation of Prime Numbers

The resulting algorithm is designed to be adaptable for different environments. In environments with very fast communication links, the communication delays among nodes will be lower and the evaluation of the result can be split into a large number of small tasks. Environments with slow communications will create a small number of bigger tasks.

To adapt the algorithm to different environments, all that is necessary is to adjust the interval to be searched in each task by modifying the value of the parameter *GRAIN* (in the tests the default value was 20000).

The figures presented on page 82 show the results obtained from the evaluation of the algorithm to find the quantity of prime numbers between 1 and 1000000.

Firstly we will analyse the results obtained for the unreliable evaluation of the algorithm (Figure 26).

The first point to mention is that the execution time required for the evaluation for two nodes is much higher than the evaluation for one node. In both cases, only one *eval* process was used. However, in the case of using only one node all the communications among the different processes are transmitted within the same node. Using two nodes, the main program and the *eval* process exchange messages through different nodes and the transmission times take longer.

Between 2 and 5 nodes, the speedup of the algorithm is linear with respect to the number of nodes used. From that point, the speedup of the algorithm reaches a limit because, although the use of more nodes increases the parallelism of the algorithm, it also increases the traffic in the network. At this point, the time lost with communication delays is balanced with the time gained from the parallelism of the algorithm.

The case for reliable evaluations requires a more careful analysis (Figure 25).

The evaluation for one node takes an excessive amount of time due to the lack of enough replica processes (point A in Figure 25). This fact causes the primary process of a group to try to restart replicas after each R²PC call (section 3.3.6) waiting for replies from the replicas according to the *confidence* degree established (section 3.3.5). However, this is also the case for the evaluation with *resilience* degree 2 using 2 nodes (point B in Figure 25). As there are not enough replicas, the system is constantly trying to restart replicas. Why is there such a gap between the execution for 1 node and the execution for 2 nodes?

The explanation for such a result lies in the fact that the system is designed to adapt dynamically to different network traffic conditions. The system uses a parameter - called *networkLatency* - that defines the time required to send a message from one node to another. This parameter is adjusted dynamically as messages are exchanged among different nodes. However, if there is only one node available in the system, the processes in execution will not receive any message from a foreign node. For this reason, the value of the *networkLatency* cannot be adjusted dynamically and the system keeps the initial default value set by the system manager (in the tests a default value of 50 milliseconds was used).

From 2 nodes onwards, the execution of the algorithm follows the same logic than the unreliable case. However, as the transmission costs of a R²PC call are much higher than a normal RPC (section 5.2.1), the algorithm reaches its limit speedup much earlier (at 3 nodes). From that point, the communication costs increase faster than the benefits obtained from the inherent parallelism of the algorithm.

5.2.4 Cost of Replicas in R²PCLinda

The last examples compared the inherent parallelism of the R²PCLinda system versus its reliability. This example is focused on the study the reliability of the system at the application level independent from its inherent parallelism.

This section reused the algorithm presented in section 5.2.3 but using only one worker process, independently of the number of nodes available.

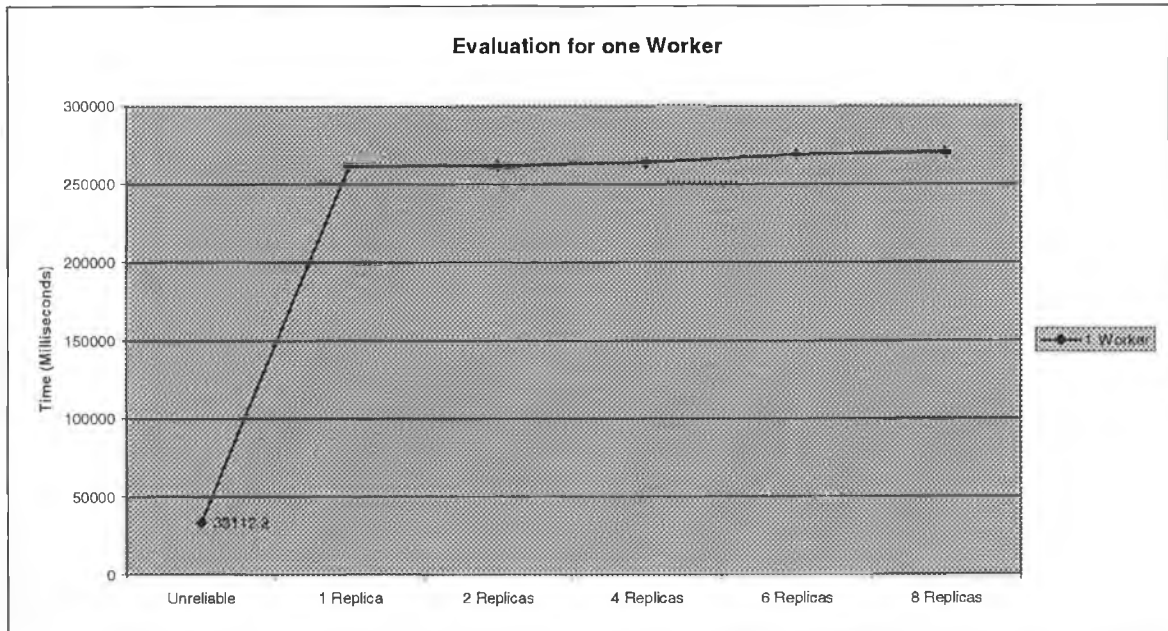


Figure 27 Evaluation of Prime Numbers using only one worker process

The results presented in Figure 27 confirm the results already obtained in section 5.2.1 regarding the transmission costs of the R²PC protocol. The reliable protocol has good scalability, performing equally well independently of the number of replica processes. As seen in section 5.2.1, the unreliable case is much more efficient than the reliable ones.

The next section takes this example as a model to study the reliability of the system.

5.3 Reliability

5.3.1 Crash Faults

The time needed to recover a crashed primary process depends on two factors. The time needed to detect the crash of a node and the time needed to restart the new primary process.

- The time needed to detect the crash of a node depends on the parameters set in the system. The more often the system checks, the more traffic overload will be on the network and the more likely it will be to think that a node has crashed when it has not. On the other hand, the longer the time left before checking, the longer it will take the system to detect a failure. Therefore, this is a parameter that has to be set very carefully.

In R²PC, the system manager can set the value of this parameter by adjusting the *primaryIsAlivePeriod* entry in the configuration file (see Appendix A). In the example case, this parameter was adjusted to 8000 milliseconds.

- The time needed to restart a primary process can vary considerably depending on the moment when the failure occurred. The longer the interval between checkpoints, the longer the restart of the primary process will take to be processed. On the other hand, very frequent checkpoints will cause delays in the normal processing of the algorithm. Therefore, when setting this parameter, the system manager will have to take into account how critical the quick recovery of a primary process is versus the efficiency of the system when no failures occur.

The current implementation of R²PC does not provide a checkpoint mechanism. Therefore, the time needed to recover a process group will depend on the number of calls that was being processed by the group before of the crash.

In this section, a crash routine was introduced in all the processes of the system (primary and replica processes) except for the DSP (the failure of the DSP is equivalent to the complete crash of a node in the current implementation – section 3.3.6).

The crash routine terminates the execution of the current process according to the probability specified by the system manager.

A timer activates this routine at an interval of time defined by the value of the *primaryIsAlivePeriod* system parameter. By activating the crash routine at such interval of time, a crash probability of 1.0 will ensure the failure of the system because all the processes will fail and none will have enough time to recover.

Figure 29 presents the percentage of System Failures in relation to the Crash Probability and number of replica processes used in each group. A System Failure occurs when all the components of a group have crashed simultaneously and the execution of the application cannot reach termination. We can see that the probability of System Failure decreases exponentially in relation to the Process Failure probability defined. What is more interesting to note is that the probability of System Failure tends to decrease in inverse relation to the number of replicas set for each group. Therefore, the existence of a larger number of replicas increases the likelihood of survival of the system.

It is interesting to note the case for Crash-Fault Probability 0.1. In this situation, the only time when the System finished its computation was for the case of 4 replicas. That case was very fortunate. Some of the evaluations for 6 and 8 replicas lasted considerably and were very close to complete the evaluation for all the prime numbers but finally crashed before completing its evaluation. Why was the execution of the system more successful for a lower number of replicas? An explanation for this phenomenon can be found by studying the results shown in Figure 28.

The diagram in Figure 28 presents the average times for the successful evaluations of the system - note that only the situations in which the system failed in all the executions are represented as System Failures in the diagram.

These results show that in case of high probability of process crash, the evaluations with larger number of replicas last much longer than the ones with a low number of replicas. This is due to the fact that the larger the number of processes in the system, the more likely is that one process will crash and will have to be restarted again. In situations where there is a large number of replica processes, each one with a high probability of failure, the system takes longer to complete its evaluation because most of the time is spent in the recovery of failed processes.

The longer is the evaluation time, the larger is the number of test points. This fact increases the probability of system failure, i.e. the probability that all the replicas in some group crash simultaneously.

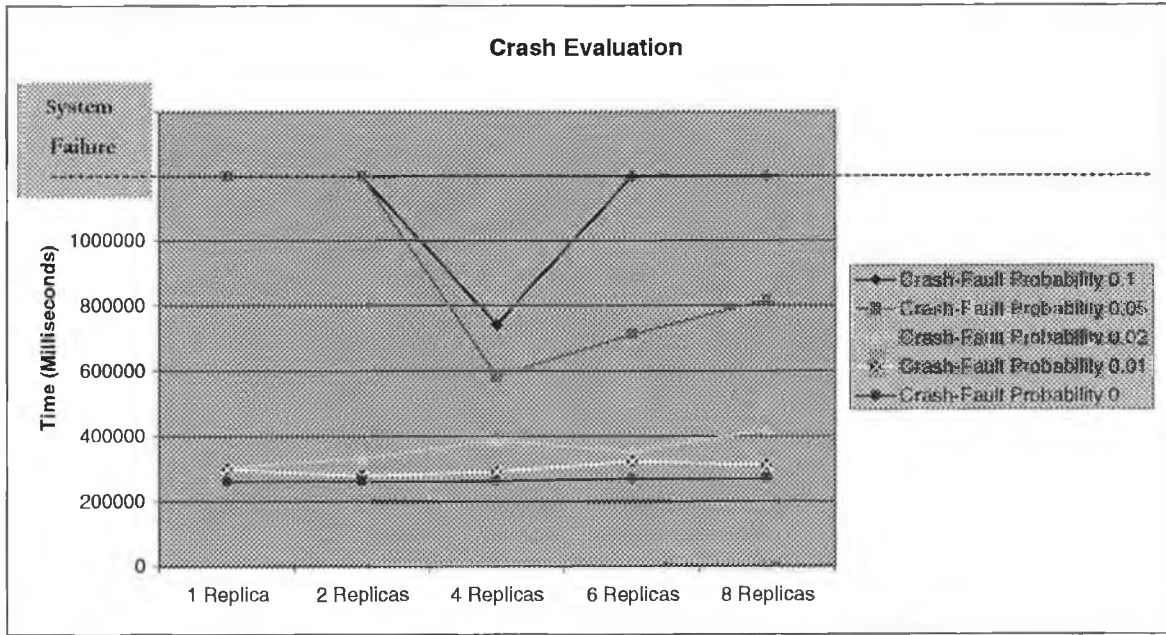


Figure 28 Evaluation of the Prime Numbers problem for one worker in the presence of Crash Faults.

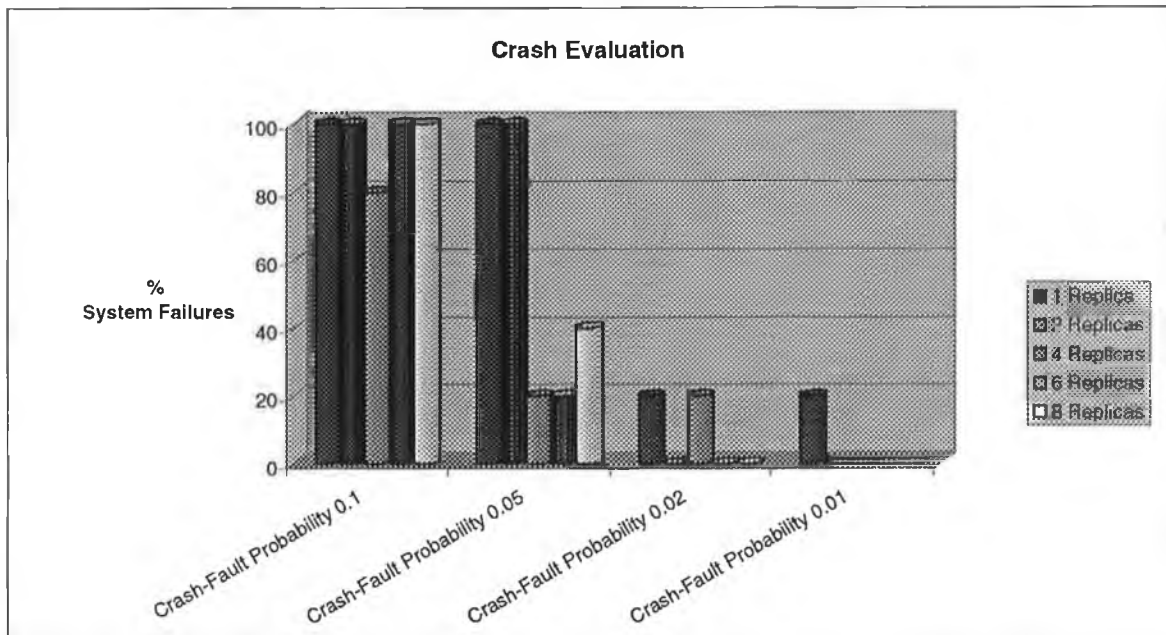


Figure 29 Percentage of System Failures in the presence of Crash Faults

5.3.2 Omission Faults

Omission Faults have been introduced in all the processes of the system (including the DSP processes). The sender thread omits the transmission of a sending block (a message may consist of a large number of blocks) according to the probability set in the tests. An omission probability of 1.0 means that no messages can be exchanged among processes.

The evaluation of the algorithm presented no problems for the cases shown in Figure 30. When a process detects that a message may not have been received, it is retransmitted until it is acknowledged.

Logically, the evaluation of the algorithm takes longer as more blocks are lost and the number of retransmissions grows.

Figure 30 presents only one case (1 replica) due to the fact that in this example there are no process crashes. Therefore, if the system works properly for one replica it will also work for other number of replicas.

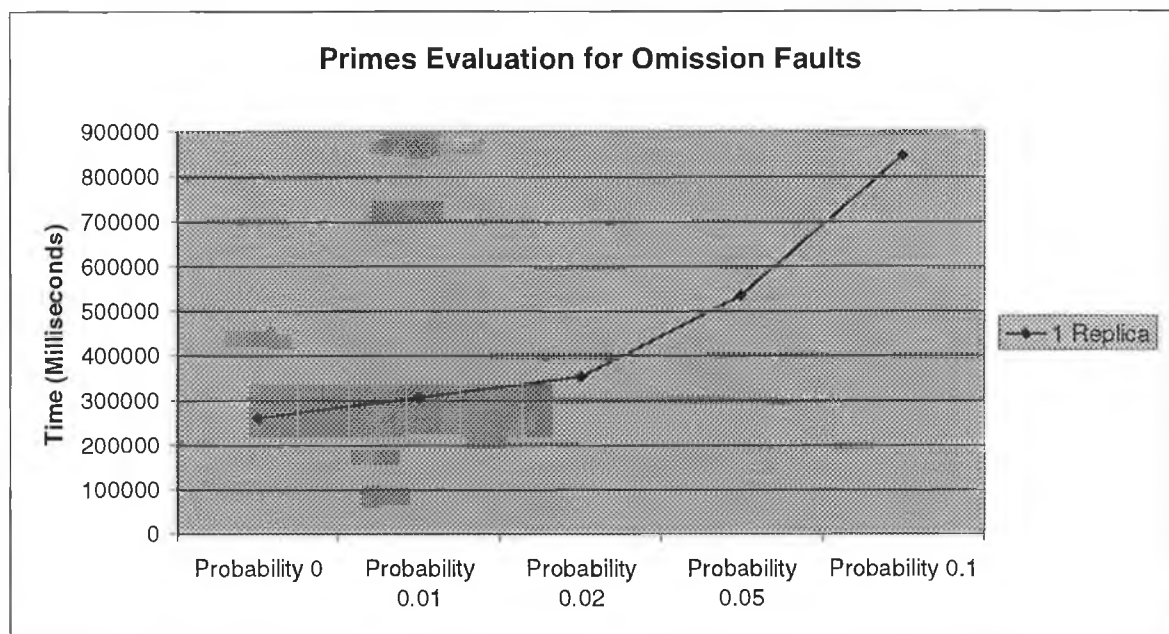


Figure 30 Evaluation of the Prime Numbers problem for one worker in the presence of Omission Faults

5.3.3 Omission-Crash Faults

This last section presents a combination of the last two cases.

This example takes the case for Crash-Fault probability of 0.02 and includes an Omission-Fault probability of 0.02. In this case we are going to study the evolution of the algorithm for different values of *confidence* degree, i.e. the level of trust defined for the reliable multicast mechanism (see section 3.3.5). In the previous examples the *confidence* degree of the system had been set to 2.

As expected from the definition of *confidence* degree, the percentage of System Failures presented in Figure 32 shows that the number of System Failures decreases as the *confidence* of the system increases.

However, for systems belonging to the Omission-Crash fault class, the ratio of System Failures does not seem to be related with the number of replicas being used anymore.

The main reason for this fact is that now there are two different types of System Failures.

- System Crash failures, where all the processes of a group have failed.
- Omission Crash failures, where process groups could not recover properly from the failure of a member due to the fact that the reliable multicast mechanism did not manage to transmit all the information of the group properly.

As it can be seen in the Appendix C.7 most of the System Failures where Omission Crash failures (although possibly they would have evolved into System Crash failures if the evaluation of the processes had continued).

The system is designed to increase its capability to recover from Omission-Crash faults as the number of replica processes increases. The presence of a higher number of replicas will increase the probability that at least one will keep the complete state of the group. This replica will be elected as new primary process of the group (see the election protocol described in section 3.3.6).

However, the results obtained from the example show that the capability of the system to recover from an Omission Failure is not directly related to the number of replicas being used. This is due to the fact that in this example only omissions from sender processes were considered. In the test, when a message is lost, no replica receives it. Therefore, once a reliable multicast message has been omitted, all the replica processes have lost it and from that moment, no replica holds the complete state of the group - unless a replica fails and is restarted again by the primary process. In

such a situation, the primary process would transfer the complete information about its current state to the restarted replica and that replica would be the candidate to be elected primary in case of crash.

The results presented in Figure 31 show that the evaluation for a *confidence* degree of 0 is extremely fast. This is due to the fact that the reliable multicast protocol does not wait for replies from the replica processes. However, Figure 32 shows that in case of failure from a primary process, this protocol is very unlikely to recover properly. In fact, the results presented in the Appendix C.7 show that most of the successful executions in the use of the protocol for *confidence* 0, were the ones in which there was no failure from the primary process of any group.

In fact, as the evaluation of the algorithm was so fast, in the case of *confidence* 0 the crash probability was increased to 0.05. Otherwise most of the evaluations would have finished successfully because the primary processes would finish the evaluation before crashing at least once.

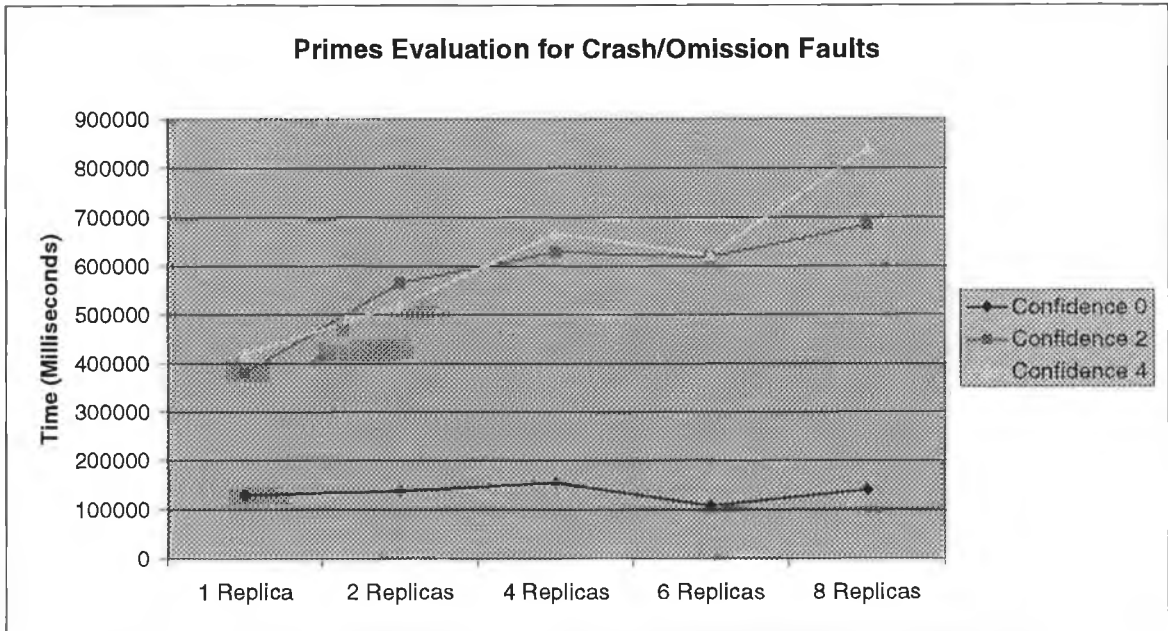


Figure 31 Evaluation of the Prime Numbers problem for one worker in the presence of Crash/Omission Faults

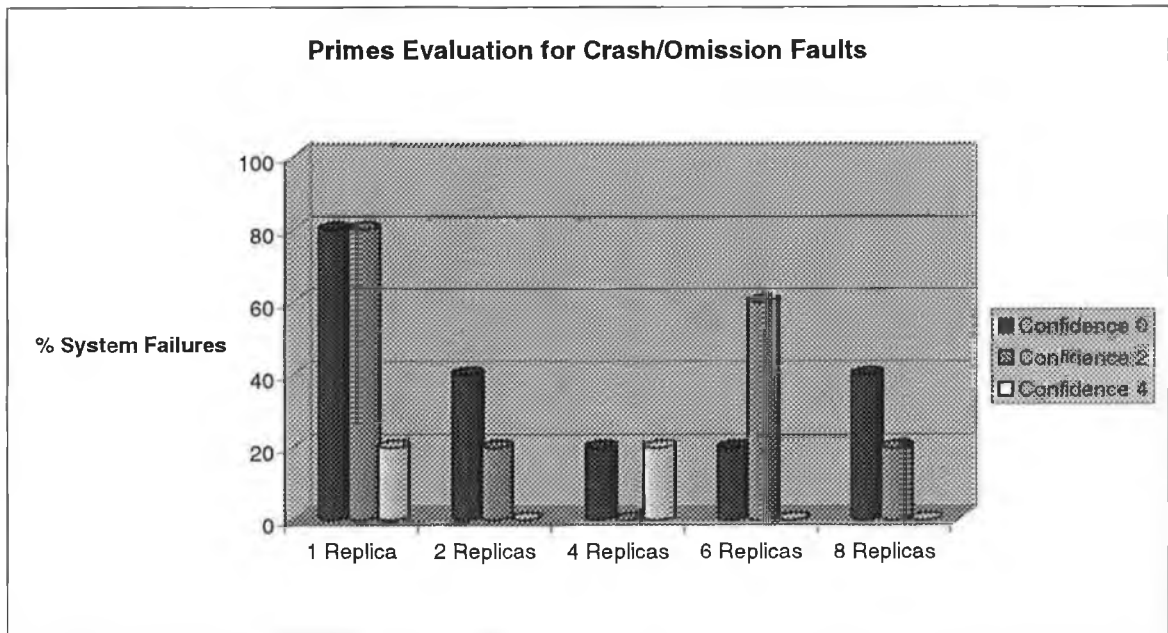


Figure 32 Percentage of System Failures in the presence of Crash and Omission Faults

5.4 Evaluation of results

The results show that R²PC is an appropriate mechanism for the construction of fault-tolerant applications that maintain a distributed volatile state in terms of transparency, efficiency and reliability.

Transparency

Chapter 4 had already presented the power of the R²PC mechanism to facilitate the creation of complex fault-tolerant applications. The reader can confirm the simplicity of using R²PC and R²PCLinda by reviewing the code presented in Appendix B. This appendix presents the source code used to perform the evaluation of the results of this chapter.

Efficiency

Initially, the efficiency of our current implementation may seem slow when compared to other existing systems. However, the efficiency obtained with different systems depends highly on the environment and conditions used for the tests. Unfortunately, we could not provide any empirical comparison of the results of our system with any other existing system because none of the existing systems has been implemented over a Windows NT environment. Nevertheless, section 5.2.1 has identified and isolated the main cause for delays in the current system – the implementation of reliable multicast. This mechanism has been constructed according to the work presented in [Kaashoek 92]. The results obtained for the evaluation of the reliable multicast mechanism running on a collection of 30 MC68030s are very promising [Kaashoek 92]. Therefore, it is expected that the implementation of R²PC on similar environments would boost the efficiency results of the protocol accordingly.

On the other hand, the results presented in sections 5.2.2 and 5.2.3 have shown the capacity of the protocol to perform the concurrent service of different calls by speeding up the evaluation of parallel applications.

Reliability

Surprisingly enough, most of the papers reviewed lack of any results to analyse the reliability of the systems described.

[Yap et al. 88] studies the failure of a node at very specific moments during the execution of a RRPC. It compares the response time of the mechanism in the scenarios of no failure, failure of the primary process after sending an acknowledgement message and failure of the replica process after sending an acknowledgement message. The results show that the event of failure of a primary process is the more expensive scenario that may happen.

[Cannon et al. 94] present the evaluation of a fault-tolerant Linda system in the presence of *crash-faults*. The results evaluate the overhead of the system in the presence of node failures.

Our results identified the *fault-floor* of the resulting system for different failure classes: *crash-fault*, *omission-fault* and *omission/crash-fault*. This information is vital to understand the behaviour of a reliable system and to determine the failure margins of the fault-tolerant applications developed with such system.

5.5 Conclusions

This chapter has tested the efficiency and reliability of R²PC and R²PCLinda using a diverse number of programming paradigms.

The implementation for the cases described resulted in simple but powerful fault-tolerant distributed applications (see Appendix B).

The results have shown that the new system can provide support for the development of parallel fault-tolerant applications. We have also identified areas to improve the efficiency of the resulting system.

R²PC and R²PCLinda have been tested thoroughly in terms of reliability. In this area, the results have surpassed our initial expectations, showing that the resulting systems could survive under very extreme conditions such as Omission/Crash environments with high probability of failure.

In summary, this chapter has shown very promising results for the development of transparent and efficient fault-tolerant systems using R²PC and R²PCLinda.

Conclusion

Achievements, Future Work and Concluding Remarks

6.1 Summary

This thesis addresses the problem of providing a transparent and efficient mechanism for the development of fault-tolerant distributed applications.

The provision of a transparent and abstract mechanism at the programming level is an element of great importance to facilitate the creation of the next generation of fault-tolerant systems. As user demands increase, future fault-tolerant systems will impose new challenges on the application programmers in terms of reliability, complexity and efficiency.

Prior work on the development of middle-ware for the design and implementation fault-tolerant software was described in chapter 2. A number of systems were reviewed showing that the approaches taken by the existing systems either do not offer an adequate level of abstraction or do not provide an adequate framework for the development of efficient fault-tolerant systems.

Chapter 3 presented my new approach to constructing fault-tolerant distributed applications. Its main goal has been the provision of support to the application programmer in terms of abstraction and transparency. A mechanism based on the Remote Procedure Call model was introduced whose only restriction to the semantics of the original RPC protocol is the introduction of determinism. The determinism required to create fault-tolerant applications in the new system is relaxed to the extent that the only requirement is the guarantee that the behaviour of the fault-tolerant programs created is determined by the results obtained from the evaluation of the Remote Procedure Calls executed. This condition is sufficient and necessary to guarantee the recovery of any failing process.

Chapter 4 illustrated the application of the approach to the development of a complex distributed system. The protocol was used to implement a system to develop parallel applications based on

the Linda co-ordination model. This example was used to illustrate the steps required to convert a distributed system based on the RPC protocol into a fault-tolerant distributed system.

The evaluation of the resulting system was given in chapter 5. The chapter provided very promising results in terms of transparency, efficiency and reliability.

- **Transparency:** The R²PC mechanism simplified extremely the creation of complex fault-tolerant distributed systems.
- **Efficiency:** The mechanism permits the concurrent service of different calls and minimises the number of messages exchanged through the use of multicast mechanisms. Both characteristics together have the potential to boost enormously the efficiency of the system.
- **Reliability:** The mechanism showed an outstanding capacity of survival under very extreme conditions, such as Omission/Crash environments with high probability of failure.

6.2 Future Research

The work introduced in this thesis can be expanded in many different ways. These include adapting our system to different environments, expanding our approach to deal with distributed object-oriented systems, studying methods to improve the efficiency of the resulting system, and applying our approach to other fault-tolerant distributed services.

Although the approach has been successfully implemented in a Windows NT environment, the work could be expanded to many other environments. In this direction we would like to study the incorporation of the protocol in standard tools for the development of distributed applications, such as CORBA [OMG 98] or Java RMI [<http://www.javasoft.com>]). At the moment, none of these standards provides an appropriate framework for the development of fault-tolerant distributed systems [Maffeis et al. 97]. Therefore, extensions to these systems are already being studied ([D. Liang et al. 98], [Maffeis 96]).

The R²PC protocol could be introduced in CORBA or Java RMI resulting in the development of a standard and generic environment that would benefit from all the properties of the new approach. However, the creation of such system requires a detailed analysis to study how to adapt the R²PC protocol in an object-oriented language. [D. Liang et al. 98] presents some of the issues involved when objects are introduced in distributed systems.

A careful analysis of the current implementation of the reliable multicast mechanism should provide more efficient results. Further work should be introduced in the current implementation to include a checkpoint mechanism. Another interesting area of research would be the study and introduction of different algorithms to increase the efficiency of the system. For example, it would be interesting to consider different implementations of reliable multicast that could take into account the high reliability of current networks to obtain more efficient protocols.

Although the resulting system has been applied successfully to implement a generic tool to develop fault-tolerant and parallel applications, the protocol should now be applied on existing fault-tolerant environments and compare it with the results obtained for other commercial applications.

6.3 Concluding Remarks

The increased use of computers in critical environments has led to an urgent need to increase the reliability of computer systems. Highly available computer systems provide fault-tolerance by replicating their tasks and data among different processing nodes. However, the construction of such systems leads to the development of very complex distributed applications that are required to deal with many fault-tolerant problems such as synchronisation, fault-detection or fault-recovery.

The Remote Procedure Call (RPC) mechanism simplifies the tasks of the programmer in the construction of distributed applications. However, the original definition of the protocol does not provide a transparent mechanism to create fault-tolerant distributed applications.

For this reason, different variations of the original protocol (RRPC mechanisms) have been created. Nevertheless, the existing RRPC systems are semantically too restrictive and/or inefficient.

- Most of the existing RRPC systems are semantically too restrictive, either excluding the use of nested calls or not permitting the recovery of client processes. These factors limit programmers of new reliable distributed systems and forces them to restructure completely the design of existing distributed applications to adapt them to the semantics required to achieve reliability.
- The existing RRPC systems that do permit the use of nested calls are too inefficient, particularly because they do not permit the concurrent service of different calls.

This thesis presented the definition and implementation of R²PC, a novel RRPC mechanism with important features for the development of fault-tolerant applications.

- R²PC facilitates enormously the introduction of fault-tolerance into new and existing distributed systems thanks to the fact that the new system preserves the semantics of the original RPC protocol avoiding most of the restrictions presented in existing RRPC systems. R²PC allows for the use of nested calls and permits the recovery of client and server processes. R²PC also relaxes the determinism required to create fault-tolerant applications to the extent that the only requirement is the guarantee that the behaviour of the fault-tolerant program created is determined by the results obtained from the evaluation of the Remote Procedure Calls executed.
- R²PC allows for efficient scaling of fault-tolerant applications by including the provision for concurrent service of different calls. This feature is provided automatically by the new system, programmers only have to establish the appropriate critical sections in their code.
- R²PC showed an outstanding capacity of survival under very extreme conditions, such as Omission/Crash environments with high probability of failure.

In summary, this thesis has shown that the definition and creation of R²PC brings a new dimension to the creation of fault-tolerant applications.

Appendix A – System Configuration

The system manager can adjust the behaviour of the R²PC and R²PCLinda systems through the edition of two .ini files.

pclinda.ini

This file contains information that is particular to the processor in use. This file must be included in the Windows or Temp directory.

The main parameters included in this file are:

[IniFiles]

- GlobalIniFile: Describes the name of the .ini file with common information for all the nodes.

[ServerInfo]

- NameHost: Describes the logical name assigned to the node.

[TupleSpace]

- TupleDirectory: Describes the local directory where to save the information of the Local Tuple Space when the user wants to use Persistent Tuple Spaces (this information is included in the global .ini file).
-

pclindag.ini

The name and location of this file is specified in the local .ini file. This file describes the parameters shared between all the processors.

[ServerInfo]

- NumServers: Number of nodes to be used by the system.
- NameServer1: Logical name of the first node.
- AddrServer1: TCP address of the first node.
- NameServer2, NameServer3, ..., NameServerN: The other nodes info.
- AddrServer2, AddrServer3, ..., AddrServerN

- BaseTcpPort: Winsock specific port number used by members of the DS group (the first group of the system).
- BaseMCastAddress: The multicast address set for the DS group.

The rest of the groups of the system obtain their addresses from the addition of their *groupId* with the base address set for the DS group.

- ResilienceDegree: Resilience degree established by the groups of the system.
A value of -1 is interpreted as Total Resilience.
- ConfidenceDegree: A value of -1 is interpreted as maximum confidence.

- NetworkLatency: Initial value used by the groups of the system. It is recommended to set a high value. After initialisation, the system will adjust this parameter automatically. Given in milliseconds. Default value - 50.

- **PrimaryIsAlivePeriod:** This value describes the maximum number of milliseconds that may pass between the failure of a node and the detection of this failure by the system. The minimum value set for this parameter to be equivalent to the time needed for the transmission of recovery information among nodes. Default value – 8000.
- **TimetoProcessRPCCall:** This value corresponds to the average time taken to process a RPC call. A client process will wait the time specified for the arrival of results from the server. After such interval, the call will be invoked once again. Default value – 4000.

[ErrorSimulation]

- **CrashProbability:** This value determines the probability that any process of the system crashes after an interval of *PrimaryIsAlivePeriod* time. Default value – 0.
- **OmissionProbability:** This value specifies the probability that any process of the system omits the transmission of a block. Default value – 0.

[Debugger]

- **InternalLogFile:** Boolean value used to specify whether the system should create a log file including information about all the messages exchanged by each process. This information is used mostly for debugging purposes.
- **InternalLogFileDirectory:** Directory where the logs are being generated.
- **DebugCommandLine:** Name of the debugger program to execute when an *eval* call is traced.

[TupleSpace]

- SaveFile: If TRUE it uses a persistent Tuple Space, preserving the contents of the Tuple Space permanently.
- TupleFile: FileName to use to save the contents of the Tuple Space.

[EvalParameters]

- EvalMode: Mode of execution of the eval programs. It uses the same parameters as defined by Windows, for example if we define SW_SHOWMINIMIZED, then all the eval programs will run in a minimised Window.

Appendix B – Source Code for the Tests

B.1 R²PC Application Test

Application used to evaluate the transmission cost of R²PC.

The current implementation of R²PC does not include any stub compiler to simplify the task of the application programmer. This tool could have been easily added, however it was not included because R²PCLinda was our main developing environment. R²PCLinda includes the existence of a precompiler, what simplifies the syntax and use of the environment (see Appendix B.2 and B.3)

```
//////////////////////////////////// Transmission Cost //////////////////////////////////////

#include <owl/owlpch.h>
#include <owl/applicat.h>
#include <owl/framewin.h>
#include <owl/dc.h>
#include <owl/inputdia.h>
#include "lcliwport.h"

#define GROUP_TEST 1
#define NUMBER_CALLS_TEST 200
#define STRING_TEST "012345678901234567890123456789012345678901234567890123456789"

int globalArgc;
char** globalArgv;
TFrameWindow *mainWindow;

class TDSPWindow : public LCLiWindowPort
{
public:
    TDSPWindow(LCLiWindowPort* parent = 0);
    virtual void SetupWindow();
};

TDSPWindow *windowManager;

/* Functions and variables to use as handlers for RPC */

void nullCall(RRPCParam &par)
{
    int i;
    char message[80];
    par.seqOut++;
}
```

```

/* Main program is activated in the primary thread using the definition of this function */
void mainProgramRRPC(RRPCParam &)
{
    RRPCParam *par, *counterPar;
    LCliWindowPort *cliWindowManager;
    int i, j;
    char message[80];
    long totalMillisec, startMillisec, endMillisec;
    double millisecRPC;

    cliWindowManager=(LCliWindowPort*)windowManager;
    counterPar=new RRPCParam;
    if (cliWindowManager->port->groupId.id != GROUP_TEST)
    {
        for (i=0; i<NUMBER_CALLS_TEST; i++)
        {
            par=new RRPCParam;
            strcpy(par->commandName, "nullCall");

            // Information used to transfer around 2k of info, 50 messages
            for (j=0; j<49; j++)
            {
                strcpy((*par->argvIn)[j+1].s, STRING_TEST);
                par->argcIn++;
            }
            cliWindowManager->callReliableRPC(*par, GROUP_TEST, *counterPar);
            delete par;
        }
    }
}

```

```

TDSPWindow::TDSPWindow(LCliWindowPort* parent)
{
    Init(parent, 0, 0);
}

```

```

void TDSPWindow::SetupWindow()
{
    windowManager=this;
    try
    {
        LCliWindowPort::SetupWindow(globalArgc, globalArgv, mainWindow, mainProgramRRPC);
        addHandlerRPC("nullCall", nullCall);
    }
    catch (LPortError *err)
    {
        err->defaultAction("TDSPWindow::SetupWindow()");
    }
}

```



```
class TDSPApp : public TApplication
{
public:
    TDSPApp() : TApplication() {}
    void InitMainWindow();
};

void TDSPApp::InitMainWindow()
{
    mainWindow=new TFrameWindow(0, "Test R2PC for 200 2k calls", new TDSPWindow);
    SetMainWindow(mainWindow);
}

int OwlMain(int argc, char* argv[])
{
    globalArgc=argc;
    globalArgv=argv;
    return TDSPApp().Run();
}
```

B.2 Evaluation of Pi

```
////////////////////////////////// EVALUATION OF PI //////////////////////////////////
```

```
#include <stdio.h>
#include <malloc.h>
#include <values.h>

#define NUM_ITERATIONS 90000000
#define PRECISSION_FACTOR 10000000000.0

evalFunct int pi()
{
    int start, stop, i;
    double interval, x, area;
    char message[80];

    in("start", &start, &stop, &interval);
    for (i=start; i<=stop; i++)
        {
            x=(i-0.5)*interval;
            x/=PRECISSION_FACTOR;
            area+=4.0/(1.0+x*x);
        }
    out("pi", area);
    sprintf(message, "Interval: %d -> %d, My area is: %20.15lf", start, stop, area);
    writeln(message);
    return 0;
}
```

```

main(int argc, char *argv[])
{
int num_workers, grain;
int i, temp;
double tempArea, area;
double interval=PRECISION_FACTOR/NUM_ITERATIONS;
long totalMillisec, startMillisec, endMillisec;
char message[81];

// Attention! the first parameter is given in argv[0]!
if (argc>0)
    num_workers=atoi(argv[0]);
else
    num_workers=1;
writeln("Parallel Evaluation of number PI");
sprintf(message, "Number of workers: %d, Number of iterations: %d", num_workers,
    NUM_ITERATIONS);
writeln(message);
writeln("Press any key when ready to begin.");
readln(message, sizeof(message));
start=new SYSTEMTIME;
end=new SYSTEMTIME;
for (i=0; i<num_workers; i++)
    eval("worker",pi());
grain=NUM_ITERATIONS/num_workers;
for (i=0; i<num_workers; i++)
    out("start", (i*grain)+1, (i+1)*grain, interval);
area=0;
for (i=0; i<num_workers; i++)
    {
        in("pi", &tempArea);
        area+=tempArea;
    }
area=(area*interval)/PRECISION_FACTOR;
sprintf(message, "Approximation to PI: %20.15lf", area);
writeln(message);
// I clean up tuple space
for (i=0; i<num_workers; i++)
    in("worker",&temp);
}

```

C.3 Evaluation of Prime Numbers

```
////////// PRIMES EVALUATION //////////
////////// AGENDA PARALLELISM //////////

#include <stdio.h>
#include <malloc.h>

#define N 1000000
// The value given to MAX has to be set according to the value of N and GRAIN
#define MAX N/1000
// GRAIN has to be an even number
#define GRAIN 20000
#define FALSE 0
#define TRUE 1
#define POISON_PILL -1

int primes[MAX], p2[MAX];
int new_primes[GRAIN], my_primes[GRAIN];

int init_primes(int p[],int p2[], bool do_out)
/* This task has to fill the first batch of primes, and out them in tuple space */
{
    int count, ok, i, num;
    char primesId[80];

    p[0]=2;
    p2[0]=4;
    p[1]=3;
    p2[1]=9;
    if (do_out)
    {
        out("primes",0, 2, 4);
        out("primes",1, 3, 9);
    }
    count=2;
    // We have to initialize enough prime numbers so that GRAIN+lastprime<lastprime*lastprime
    for (num=5; p2[count-1] <= (GRAIN+p[count-1]); num +=2)
    {
        for (i=1, ok=1; i<count; ++i)
        {
            if (!(num%p[i]))
            {
                ok=0;
                break;
            }
            if (num<p2[i])
                break;
        }
        if (ok)
        {
            p[count]=num;
            p2[count]=num*num;
        }
    }
}
```

```

        if (do_out)
        {
            out("primes",count, num, num*num);
        }
        ++count;
    }
    return count;
}

evalFunc int worker()
{
int count, eot, i, limit, num, num_primes, ok, start;
double start_square, my_primes_square;
char resultId[80], primesId[80];

num_primes=init_primes(primes, p2, FALSE);
eot=FALSE;
while (TRUE)
    {
    in("next task", &num);
    sprintf(resultId, "Next task: %d", num);
    writeln(resultId);
    if (num==POISON_PILL)
        {
        out("next task", POISON_PILL);
        return 0;
        }
    limit=num+GRAIN;
    if (limit>N)
        {
        out("next task", POISON_PILL);
        limit=N;
        }
    else
        out("next task", limit);
    start=num;
    for (count=0; num<limit; num += 2)
        {
        while (!eot && num>p2[num_primes-1])
            {
            rd("primes", num_primes, &(primes[num_primes]), &(p2[num_primes]));
            if (p2[num_primes]<0)
                eot=TRUE;
            else
                ++num_primes;
            }
        for (i=1, ok=1; i<num_primes; ++i)
            {
            if (!(num%primes[i]))
                {
                ok=0;
                break;
                }
            if (num<p2[i])
                break;
            }
        }
    }
}

```

```

        }
    if (ok)
    {
        my_primes[count]=num;
        ++count;
    }
}
/* Send the control process any primes found */
sprintf(resultId, "result_count_%d", start);
out(resultId, count);
sprintf(resultId, "result_%d", start);
i=0;
my_primes_square=my_primes[i]*my_primes[i];
while ( (i<count) &&
        (my_primes_square<N) )
    {
        out(resultId, my_primes[i], i);
        i++;
        my_primes_square=my_primes[i]*my_primes[i];
        if (my_primes_square<my_primes[i])
            my_primes_square=N;
    }
start_square=start*start;
if (start_square<start)
    start_square=N;
if ( (i<count) && (start_square<N) ) // I transmit the last result
    out(resultId, my_primes[i], i);
}
}

main(int argc, char *argv[])
{
    int eot, first_num, i, length, np2;
    int num, num_primes, num_workers, rubbish;
    int total_primes, eot_result;
    long totalMillisec, startMillisec, endMillisec;
    char message[50], resultId[80], primesId[80];

    // Attention! the first parameter is given in argv[0]!
    if (argc>0)
        num_workers=atoi(argv[0]);
    else
        num_workers=1;
    writeln("Primes evaluation with Agenda Paralelism");
    sprintf(message, "Number of workers: %d, Counting number of primes upto: %d, Grain: %d",
        num_workers, N, GRAIN);
    writeln(message);
    writeln("Press any key when ready to begin.");
    readln(message, sizeof(message));
    for (i=0; i<num_workers; ++i)
        eval("worker", worker());
    num_primes=init_primes(primes, p2, TRUE);
    total_primes=num_primes;
    first_num=primes[num_primes-1]+2;
    out("next task", first_num);
    eot=FALSE; // Becomes TRUE at end of table

```

```

for (num=first_num; num<N; num += GRAIN)
{
    sprintf(resultId, "result_count_%d", num);
    in(resultId, &length);
    total_primes+=length;
    sprintf(resultId, "result_%d", num);
    eot_result=FALSE;
    // last two conditions only added for the count case
    for (i=0; ((i<length) && !eot_result) && !eot ); i++)
        {
            in(resultId, &(new_primes[i]), i);
            if (new_primes[i]*new_primes[i]>N)
                eot_result=TRUE;
        }
    length=i;
    for (i=0; i<length; ++i, ++num_primes)
        {
            primes[num_primes]=new_primes[i];
            if (!eot)
                {
                    np2=new_primes[i]*new_primes[i];
                    if (np2>N) // This is the last task to do
                        {
                            eot=TRUE;
                            np2=POISON_PILL;
                        }
                    out("primes", num_primes, new_primes[i], np2);
                }
        }
}
for (i=0; i<num_workers; ++i)
    in("worker", &rubish);
sprintf(message, "There are %d primes less than %d", total_primes, N);
writeln(message);
writeln("Press a key to finish.");
readln(message, sizeof(message));
}

```

Appendix C – Results

Evaluation of the protocols

C.1 Transmission Costs of R²PC vs. RPC

The following tables show the transmission results obtained from the evaluation of the R²PC and RPC mechanism as described in section 5.2.1.

In the RPC case there should be no difference among the evaluations using different replicas. To represent this fact graphically, we only obtained the results for one replica and replicated the value for the rest of the replica cases.

Results for 2k R²PC

	<i>1 Replica</i>	<i>2 Replicas</i>	<i>3 Replicas</i>	<i>6 Replicas</i>	<i>8 Replicas</i>
<i>Test 1</i>	1094.92	1085	1084.685	1096.25	1107.89
<i>Test 2</i>	1094.61	1085.155	1088.83	1089.375	1099.455
<i>Test 3</i>	1107.425	1088.28	1094.455	1099.845	1096.875
<i>Test 4</i>	1089.14	1081.405	1086.56	1108.205	1102.11
<i>Test 5</i>	1089.3	1086.64	1085.031	1086.8	1110.15
2k R²PC	1095.079	1085.296	1087.9122	1096.095	1103.296

Results for 0k R²PC

	<i>1 Replica</i>	<i>2 Replicas</i>	<i>3 Replicas</i>	<i>6 Replicas</i>	<i>8 Replicas</i>
<i>Test 1</i>	536.795	523.125	536.875	544.145	539.295
<i>Test 2</i>	535.39	524.45	539.92	535.78	545.625
<i>Test 3</i>	531.25	522.735	540.78	539.925	547.035
<i>Test 4</i>	537.265	527.185	547.81	547.345	538.332
<i>Test 5</i>	528.91	523.75	551.095	553.125	545.074
0k R²PC	533.922	524.249	543.296	544.064	543.0722

Results for 2k RPC

	<i>1 Replica</i>	<i>2 Replicas</i>	<i>3 Replicas</i>	<i>6 Replicas</i>	<i>8 Replicas</i>
<i>Test 1</i>	67.81	67.81	67.81	67.81	67.81
<i>Test 2</i>	67.11	67.11	67.11	67.11	67.11
<i>Test 3</i>	66.95	66.95	66.95	66.95	66.95
<i>Test 4</i>	64.375	64.375	64.375	64.375	64.375
<i>Test 5</i>	66.72	66.72	66.72	66.72	66.72
2k RPC	66.593	66.593	66.593	66.593	66.593

Results for 0k RPC

	<i>1 Replica</i>	<i>2 Replicas</i>	<i>3 Replicas</i>	<i>6 Replicas</i>	<i>8 Replicas</i>
<i>Test 1</i>	33.05	33.05	33.05	33.05	33.05
<i>Test 2</i>	33.83	33.83	33.83	33.83	33.83
<i>Test 3</i>	34.61	34.61	34.61	34.61	34.61
<i>Test 4</i>	35.94	35.94	35.94	35.94	35.94
<i>Test 5</i>	33.05	33.05	33.05	33.05	33.05
0k RPC	34.096	34.096	34.096	34.096	34.096

C.2 Evaluation of Pi

These results correspond to the evaluation of the example described in section 5.2.2.

Results for Unreliable Evaluation

	<i>1 Node</i>	<i>2 Nodes</i>	<i>3 Nodes</i>	<i>5 Nodes</i>	<i>7 Nodes</i>	<i>9 Nodes</i>
<i>Test 1</i>	44094	40250	20344	16875	12860	9360
<i>Test 2</i>	44094	40625	20359	16969	12704	9047
<i>Test 3</i>	44141	40234	20375	16844	12844	9063
<i>Test 4</i>	44141	40281	20359	16766	12813	8938
<i>Test 5</i>	44094	40281	20422	16813	12828	9328
Unreliable Evaluation	44112.8	40334.2	20371.8	16853.4	12809.8	9147.2

Results for Evaluation with Resilience 1

	<i>1 Node</i>	<i>2 Nodes</i>	<i>3 Nodes</i>	<i>5 Nodes</i>	<i>7 Nodes</i>	<i>9 Nodes</i>
<i>Test 1</i>	55125	43469	30719	23125	23578	29609
<i>Test 2</i>	55344	43812	26735	18891	20172	32657
<i>Test 3</i>	55218	43297	27603	20297	23157	30937
<i>Test 4</i>	55188	43469	27203	21234	19000	32656
<i>Test 5</i>	55188	43453	26859	21391	24672	33156
Resilience 1	55212.6	43500	27823.8	20987.6	22115.8	31803

Results for Evaluation with Resilience 2

	<i>1 Node</i>	<i>2 Nodes</i>	<i>3 Nodes</i>	<i>5 Nodes</i>	<i>7 Nodes</i>	<i>9 Nodes</i>
<i>Test 1</i>	55343	49937	28219	25172	22437	28203
<i>Test 2</i>	55187	49219	27547	21781	19906	24578
<i>Test 3</i>	55234	49125	26734	22484	20359	20625
<i>Test 4</i>	55219	49875	24594	21969	25000	26547
<i>Test 5</i>	55188	49844	25688	22047	21984	26531
Resilience 2	55234.2	49600	26556.4	22690.6	21937.2	25296.8

Results for Evaluation with Resilience 4

	<i>1 Node</i>	<i>2 Nodes</i>	<i>3 Nodes</i>	<i>5 Nodes</i>	<i>7 Nodes</i>	<i>9 Nodes</i>
<i>Test 1</i>	55203	44703	29110	23907	24687	22985
<i>Test 2</i>	55203	44516	29375	24325	19094	21469
<i>Test 3</i>	55203	49469	28859	24969	24656	18859
<i>Test 4</i>	55203	49375	27844	25484	20031	19406
<i>Test 5</i>	55218	49359	25781	24156	29094	22375
Resilience 4	55206	47484.4	28193.8	24568.2	23512.4	21018.8

Results for Evaluation with Resilience 6

	<i>1 Node</i>	<i>2 Nodes</i>	<i>3 Nodes</i>	<i>5 Nodes</i>	<i>7 Nodes</i>	<i>9 Nodes</i>
<i>Test 1</i>	55219	49922	25140	24797	24313	23375
<i>Test 2</i>	55187	47531	28360	23656	29313	37640
<i>Test 3</i>	55234	47547	27906	27532	27375	29187
<i>Test 4</i>	55141	50109	27281	23312	31453	27359
<i>Test 5</i>	55203	46250	28109	23469	24032	31985
Resilience 6	55196.8	48271.8	27359.2	24553.2	27297.2	29909.2

Results for Evaluation with Resilience 8

	<i>1 Node</i>	<i>2 Nodes</i>	<i>3 Nodes</i>	<i>5 Nodes</i>	<i>7 Nodes</i>	<i>9 Nodes</i>
<i>Test 1</i>	55172	45547	25968	20610	24579	32297
<i>Test 2</i>	55203	49344	27687	22516	34062	28484
<i>Test 3</i>	55157	44132	26171	25234	31640	28000
<i>Test 4</i>	55219	43844	30500	28203	24360	31344
<i>Test 5</i>	55234	49969	26688	25000	23844	34500
Resilience 8	55197	46567.2	27402.8	24312.6	27697	30925

Results for Completely Reliable Evaluation

	<i>1 Node</i>	<i>2 Nodes</i>	<i>3 Nodes</i>	<i>5 Nodes</i>	<i>7 Nodes</i>	<i>9 Nodes</i>
<i>Test 1</i>	44141	43547	28328	24859	26219	27422
<i>Test 2</i>	44094	43406	26297	23750	25219	24547
<i>Test 3</i>	44093	42891	26969	23937	22500	28125
<i>Test 4</i>	44094	42843	25875	25047	25954	24125
<i>Test 5</i>	44094	44016	25204	22140	24797	24922
Total Resilience	44103.2	43340.6	26534.6	23946.6	24937.8	25828.2

C.3 Evaluation of Prime Numbers

These results correspond to the evaluation of the example described in section 5.2.3.

The average values for the reliable cases with one node have not been depicted in the graphs represented in page 82 due to its disproportion with the rest of the results obtained.

Results for Unreliable Evaluation

	<i>1 Node</i>	<i>2 Nodes</i>	<i>3 Nodes</i>	<i>5 Nodes</i>	<i>7 Nodes</i>	<i>9 Nodes</i>
<i>Test 1</i>	30610	39609	32719	23657	28125	29031
<i>Test 2</i>	26968	36750	36719	28031	31828	33000
<i>Test 3</i>	26750	38078	35656	30984	32390	27985
<i>Test 4</i>	24187	39094	33672	24594	28609	24968
<i>Test 5</i>	26469	38594	35281	27844	28843	32203
Unreliable Evaluation	26996.8	38425	34809.4	27022	29959	29437.4

Results for Evaluation with Resilience 1

	<i>1 Node</i>	<i>2 Nodes</i>	<i>3 Nodes</i>	<i>5 Nodes</i>	<i>7 Nodes</i>	<i>9 Nodes</i>
<i>Test 1</i>	2304015	415219	309406	330031	356328	374015
<i>Test 2</i>	2308172	410625	301531	342704	345453	432218
<i>Test 3</i>	2308297	411875	298078	340563	382141	371859
<i>Test 4</i>	2306781	409156	315609	347984	341469	360265
<i>Test 5</i>	2308094	409125	353062	318188	374344	403094
Resilience 1	2307071.8	411200	315537.2	335894	359947	388290.2

Results for Evaluation with Resilience 2

	<i>1 Node</i>	<i>2 Nodes</i>	<i>3 Nodes</i>	<i>5 Nodes</i>	<i>7 Nodes</i>	<i>9 Nodes</i>
<i>Test 1</i>	2308781	485094	319125	337640	351422	357703
<i>Test 2</i>	2309003	484266	314609	341015	341609	350907
<i>Test 3</i>	2305142	492203	308453	323547	349890	352579
<i>Test 4</i>	2308121	481342	314078	325937	353828	379203
<i>Test 5</i>	2305277	492003	325281	318875	338359	421235
Resilience 2	2307264.8	486981.6	316309.2	329402.8	347021.6	372325.4

C.4 Prime Numbers Evaluation for one Worker Process

These results correspond to the evaluation of the example presented in section 5.2.4.

	Unreliable	1 Replica	2 Replicas	4 Replicas	6 Replicas	8 Replicas
	1 Node	2 Nodes	3 Nodes	5 Nodes	7 Nodes	9 Nodes
Test 1	36640	272031	262593	272375	276750	271890
Test 2	31218	258000	266687	261438	265141	263406
Test 3	32250	261937	267096	257656	273219	278437
Test 4	32906	258437	266125	262109	263937	240110
Test 5	32547	256125	247141	264547	265328	299454
1 Worker	33112.2	261306	261928.4	263625	268875	270659.4

C.5 Evaluation of Crash Faults

These results correspond to the evaluation presented in section 5.3.1.

The results that start with a C followed by a number indicate that the evaluation of the system crashed after the number of milliseconds specified.

The average time taken by the evaluation of the system is not taken when all the evaluations of the system had crashed.

Crash Probability 0.1 (1/10)

	1 Replica	2 Replicas	4 Replicas	6 Replicas	8 Replicas
Test 1	C 96000	C 43000	740359	C 242000	C 115000
Test 2	C 391000	C 221000	C 299000	C 187000	C 1016000
Test 3	C 30000	C 372000	C 570000	C 777000	C 690000
Test 4	C 42000	C 26000	C 516000	C 740000	C 931000
Test 5	C 149000	C 203000	C 311000	C 602000	C 1813000
Crash-Fault Probability 0.1			740359		

Crash Probability 0.05 (1/20)

	1 Replica	2 Replicas	4 Replicas	6 Replicas	8 Replicas
Test 1	C 281000	C 302000	503485	793610	C 910000
Test 2	C 330000	C 610000	425953	C 424000	651891
Test 3	C 21000	C 396000	C 756000	575156	C 923000
Test 4	C 375000	C 30000	965390	772578	643531
Test 5	C 302000	C 23000	443968	707110	1160172
Crash-Fault Probability 0.05			584699	712113.5	818531.33

Crash Probability 0.02 (1/50)

	<i>1 Replica</i>	<i>2 Replicas</i>	<i>4 Replicas</i>	<i>6 Replicas</i>	<i>8 Replicas</i>
<i>Test 1</i>	313984	339625	331203	314234	474937
<i>Test 2</i>	308157	376547	529266	421578	597828
<i>Test 3</i>	C 60000	300547	312234	327969	324719
<i>Test 4</i>	266282	321563	367782	389516	427735
<i>Test 5</i>	330844	318437	C 425000	311594	273016
Crash-Fault Probability 0.02	304816.75	331343.8	385121.25	352978.2	419647

Crash Probability 0.01 (1/100)

	<i>1 Replica</i>	<i>2 Replicas</i>	<i>4 Replicas</i>	<i>6 Replicas</i>	<i>8 Replicas</i>
<i>Test 1</i>	281718	269485	299907	395609	275360
<i>Test 2</i>	C 145000	318938	279704	273329	309438
<i>Test 3</i>	265531	264234	261719	323687	338906
<i>Test 4</i>	312046	265203	258781	291297	284500
<i>Test 5</i>	333219	271203	349031	322375	336110
Crash-Fault Probability 0.01	298128.5	277812.6	289828.4	321259.4	308862.8

Crash Probability 0

	<i>1 Replica</i>	<i>2 Replicas</i>	<i>4 Replicas</i>	<i>6 Replicas</i>	<i>8 Replicas</i>
<i>Test 1</i>	272031	262593	272375	276750	271890
<i>Test 2</i>	258000	266687	261438	265141	263406
<i>Test 3</i>	261937	267096	257656	273219	278437
<i>Test 4</i>	258437	266125	262109	263937	240110
<i>Test 5</i>	256125	247141	264547	265328	299454
Crash-Fault Probability 0	261306	261928.4	263625	268875	270659.4

C.6 Evaluation of Omission Faults

The results presented correspond to the evaluation of the system described in section 5.3.2.

1 Replica

	<i>Probability 0</i>	<i>Probability 0.01</i>	<i>Probability 0.02</i>	<i>Probability 0.05</i>	<i>Probability 0.1</i>
<i>Test 1</i>	272031	299532	349328	485640	892312
<i>Test 2</i>	258000	305344	364500	537765	806953
<i>Test 3</i>	261937	312860	379218	578844	870141
<i>Test 4</i>	258437	308860	328531	586828	853985
<i>Test 5</i>	256125	310156	348562	483110	816188
1 Replica	261306	307350.4	354027.8	534437.4	847915.8

C.7 Evaluation of Omission/Crash Faults

These results correspond to the evaluation presented in section 5.3.3.

The results that start with a C followed by a number indicate that the System Crashed after the number of milliseconds specified.

The results that start with an O followed by a number indicate the time when the System Failed with an Omission error.

The squares in white colour identify executions where the primary processes did not crash at all. The execution times for confidence degree 0 were very fast, for this reason the probability of a crash was increased to 0.05.

The rest of the cases use crash and omission probability equal to 0.02.

The average time taken by the evaluation of the system is not taken when all the evaluations of the system had crashed.

Confidence Degree 0 (Crash Prob: 0.05)

	1 Replica	2 Replicas	4 Replicas	6 Replicas	8 Replicas
Test 1	O 156000	129281	214375	75656	125094
Test 2	128438	O 212000	151922	148672	158407
Test 3	O 133000	155141	O 76000	O 170000	140594
Test 4	C 151000	O 59000	143109	111672	O 172000
Test 5	O 145000	131875	110234	96422	O 228000
Confidence 0	128438	138765.6667	154910	108105.5	141365

Confidence Degree 2

	1 Replica	2 Replicas	4 Replicas	6 Replicas	8 Replicas
Test 1	380906	504359	492610	478547	942297
Test 2	O 465000	507969	588547	O 456000	543453
Test 3	C 43000	634844	444875	O 556000	726922
Test 4	C 202000	615969	862094	758250	524765
Test 5	O 318000	O 311000	755734	O 490000	O 931000
Confidence 2	380906	565785.25	628772	618398.5	684359.25

Confidence Degree 4

	<i>1 Replica</i>	<i>2 Replicas</i>	<i>4 Replicas</i>	<i>6 Replicas</i>	<i>8 Replicas</i>
<i>Test 1</i>	535328	628734	815688	590062	814968
<i>Test 2</i>	368719	480672	○ 347000	547343	601609
<i>Test 3</i>	363094	500703	616563	678984	899594
<i>Test 4</i>	C 99000	444781	631218	541454	808829
<i>Test 5</i>	414141	547312	556563	755015	1066703
Confidence 4	420320.5	520440.4	667508	622571.6	838340.6

Bibliography

[Avizienis et al. 84] A. Avizienis and J. Kelly, *Fault Tolerance by Design Diversity: Concepts and Experiments*, IEEE Computer, vol. 17, no. 8, pp. 67-80, Aug. 1984.

[Babaoğlu et al. 93] Ö. Babaoğlu and S. Toueg, *Understanding Non-blocking Atomic Commitment*, Tech. Report UBLCS-93-2, Laboratory for Computer Science, University of Bologna, Italy, Jan. 1993.

[Bakken et al. 95] D. Bakken and R. Schlichting, *Supporting Fault-Tolerant Parallel Programming in Linda*, IEEE Trans. on Parallel and Distrib. Systems, vol. 6, no. 3, pp. 287-302, March 1995.

[Barborak et al. 93] M. Barborak, M. Malek and A. Dahbura, *The Consensus Problem in Fault-Tolerant Computing*, ACM Computing Surveys, vol. 25, no. 2, pp. 171-220, June 1993.

[Beedubail et al. 95] G. Beedubail G, A. Karmarkar, A. Grujjala, W. Marti and U. Pooch, *Fault Tolerant Objects in Distributed Systems using Hot Replication*, Technical Report TR-95-023, Department of Computer Science, Texas A&M University, USA, April 1995.

[Berkeley 86] *UNIX Reference Manual*, 4.3 Berkeley Software Distribution, Computer Systems Research Group, Computer Science Division, Univ. of California, Berkeley, USA, Apr. 1986.

[Bhatti et al. 98] N. Bhatti, M. Hiltunen, R. Schlichting and W. Chiu, *Covote: A System for Constructing Fine-Grain Configurable Communication Services*, ACM Trans. Comp. Syst., vol. 16, no. 4, pp. 321-366, Nov. 1988.

[Birman 91] K. Birman, *Maintaining Consistency in Distributed Systems*, Tech. Report TR 91-1240, Dept. of Computer Science, Cornell University, USA, Nov. 1991.

[Birman 93] K. Birman, *The Process Group Approach to Reliable Distributed Computing*, Communications of the ACM, vol. 36, no. 12, Dec. 1993.

[Birman et al. 94] K. Birman and R. van Renesse eds., *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, 1994.

[Birrell et al. 84] A. Birrell and B. Nelson, *Implementing Remote Procedure Call*, ACM Trans. Comp. Syst., Vol. 2, No. 1, pp. 39-59, Feb. 1984.

[Cannon et al. 94] S. Cannon and D. Dunn, *Adding Fault-Tolerant Transaction Processing to Linda*, Software - Practice and Experience, vol. 24, no. 5, pp. 449-466, May 1994.

[Carriero et al. 90] N. Carriero and D. Gelernter, *How to Write Parallel Programs: A First Course*, The MIT Press, Cambridge, MA, USA, 1990.

[Carriero et al. 93] N. Carriero, D. Gelernter, D. Kaminsky and J. Westbrook, *Adaptive Parallelism with Piranha*, Tech. Report 954, Yale University, February 1993.

[CEI/IEC 97] CEI/IEC 61508, *Functional safety of electrical/electronic/programmable electronic safety-related systems*, Parts 1-7, 1997.

[Cheriton 84] D. Cheriton, *The V Kernel: A Software Base for Distributed Systems*, IEEE Software, vol. 1, no. 12, pp. 19-43, 1984.

- [Cheriton 88] D. Cheriton, *The V Distributed System*, Communications of the ACM, vol. 31, no. 3, pp. 314-333, March 1988.
- [Cooper 85] E. Cooper, *Replicated Distributed Programs*. Proceedings of the 10th Symposium on Operating System Principles, ACM, pp. 63-78, 1985.
- [Coulouris et al. 88] G. Coulouris and J. Dollimore, *Distributed Systems: Concepts and Design*. Addison-Wesley, California 1988.
- [Cristian 91] F. Cristian, *Understanding Fault-Tolerant Distributed Systems*, Communications of the ACM, pp. 56-78, vol. 34, no. 2, Feb. 1991.
- [D. Liang et al. 98] D. Liang, S. Chou and S. Yuan, *A fault-tolerant object service in the OMG's object management architecture*, Information and Software Technology, vol. 39, pp. 965-973, 1998.
- [Davidson et al. 85] S. Davidson, H. Garcia-Molina and O. Skeen, *Consistency in Partitioned Networks*, ACM Comp. Surveys, vol. 17, no. 3, pp. 341-370, Sept. 1985.
- [Dolev et al. 1985] D. Dolev and R. Reischuk, *Bounds on Information Exchange for Byzantine Agreement*, Journal of the ACM, vol. 32, no. 1, pp. 191-204, Jan. 1985.
- [Elmasiri et al. 94] R. Elmasiri and S. Navathe, *Fundamentals of Database Systems*, Addison-Wesley, 1994.
- [Garcia-Molina 82] H. Garcia-Molina, *Elections in a Distributed Computing System*, IEEE Trans. on Computers, vol. 31, no. 1, pp. 48-59, Jan. 1982.
- [Gelernter 85] D. Gelernter, *Generative Communication in Linda*, ACM Trans. Program. Lang. Syst., vol. 7, no.1, Jan. 1985.
- [Heimerdinger et al. 92] W. Heimerdinger and C. Weinstock, *A Conceptual Framework for System Fault Tolerance*. Tech. Report CMU/SEI-92-TR-33, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, Oct. 1992.
- [ISO/IEC 98] ISO/IEC TR 15504-9, *Information technology - Software process assessment*, Parts 1-9. (SPICE), 1998.
- [Jalote 89] P. Jalote, *Resilient Objects in Broadcast Networks*, IEEE Trans. Soft. Eng., vol. 15, no.1, pp. 68-72, Jan. 1989.
- [Jeong et al. 94] K. Jeong and D. Shasha, *Plinda 2.0: A Transactional/Checkpointing Approach to Fault Tolerant Linda*, in Proc. of the Thirteenth Symp. on Fault-Tolerant Distributed Systems, Oct. 1994.
- [Kaashoek 92] F. Kaashoek, *Group Communication for Distributed Systems*, PhD Thesis, Vrije Universiteit, Amsterdam, 1992.
- [Kaashoek et al. 93] M. Kaashoek, R. van Renesse, H. van Staveren and A. Tanenbaum, *FLIP: an Internetwork Protocol for Supporting Distributed Systems*, ACM Trans. Comp. Syst., vol. 11, no. 1, pp. 73-106, Feb. 1993.
- [Kambhatla 90] S. Kambhatla, *Recovery with Limited Replay: Fault-Tolerant Processes in Linda*, Tech. Report CS/E 90-019, Dept. of Comp. Sci., Oregon Graduate Institute, USA, 1990.
- [L. Liang et al. 90] L. Liang, S. Chanson and G. Neufeld, *Process Groups and Group Communications: Classifications and Requirements*, IEEE Computer, vol. 23, no. 2, pp. 56-66, Feb. 1990.
- [Lamport et al. 82] L. Lamport, R. Shostak and M. Pease, *The Byzantine Generals Problem*, ACM Trans. Program. Lang. Syst., vol. 4, no. 3, pp. 382-401, July 1982.
- [Landis et al. 97] S. Landis and S. Maffei, *Buiding Reliable Distributed Systems with CORBA*, Theory and Practice of Object Systems, ed: John Wiley, New York, Apr. 1997.

- [Laprie et al. 90] J. Laprie, J. Arlat, C. Beounes and K. Kanoun, Definition and Analysis of Hardware and Software Fault-Tolerant Architectures. IEEE Computer, vol. 23, no. 7, July 1990.
- [Lewis et al. 92] T. Lewis and H. El-Rewini, Introduction to Parallel Computing. Prentice-Hall, 1992.
- [Maffeis 96] S. Maffeis, PIRANHA – A hunter of crashed CORBA Objects, Technical Report TR96-1569, Department of Computer Science, Cornell University, Jan. 1996.
- [Maffeis et al. 97] S. Maffeis and D. Schmidt, Constructing Reliable Distributed Communication Systems with CORBA. IEEE Communications, pp. 56-60, Feb. 1997.
- [Manso 96] O. Manso, PCLinda: A Tool for Parallel Programming. MSc Thesis, Dublin City University, Ireland, Oct. 1996.
- [OMG 98] Object Management Group, The Common Object Request Broker Architecture and Specification, MA, USA 1998.
- [Parrington et al. 94] G. Parrington, S. Shrivastava, S. Wheeler and M. Little, The Design and Implementation of Arjuna, BROADCAST Project deliverable report, vol. 4, Oct. 1994.
- [Powell 95] D. Powell, Failure Mode Assumptions and Assumption Coverage. Tech. Report 91462, LAAS-CNRS, 31077 Toulouse, France, March 1995.
- [Resnick 96] R. Resnick, A Modern Taxonomy of High Availability, <http://www.interlog.com/~resnick/HA.htm>, 1996.
- [Schlichting et al. 83] R. Schlichting and F. Schneider, Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems, ACM Trans. Comp. Syst., vol. 1, no. 3, pp. 222-238, Aug. 1983.
- [Schneider 84] F. Schneider, Byzantine Generals in Action: Implementing Fail-Stop Processors. ACM Trans. Comput. Syst., vol. 2, no. 2, pp. 145-154, May 1984.
- [Schneider 90] F. Schneider, Implementing fault-tolerant services using the State Machine approach: A tutorial. ACM Computing Surveys, vol. 22, no. 4, pp. 299-319, Dec. 1990.
- [Shekhar 93] K. Shekar and Y. Srikant, Linda Sub System on Transputers. Computer Languages, Vol. 18, No. 2, 1993.
- [Shrivastava 94] S. Shrivastava, Lessons Learned from Building Arjuna Distributed Programming System. Dagstuhl Seminar on Distributed Systems, pp. 17-32, 1994.
- [Siewiorek 90] D. Siewiorek, Fault-Tolerance in Commercial Computers. IEEE Computer, vol. 23, no. 7, July 1990.
- [Sully 93] P. Sully, Modelling the World with Objects. Prentice Hall, 1993.
- [Ullman 88] J. D. Ullman, Principles of Database and Knowledge-Base Systems, vol. 1, Computer Science Press, 1988.
- [van Renesse et al. 96] R. van Renesse, K. Birman and S. Maffeis, Horus. a Flexible Group Communication System. Communications of the ACM, vol. 39, no. 4, pp. 76-83, Apr. 1996.
- [Vinoski 97] S. Vinoski, CORBA: Integrating diverse applications within distributed heterogeneous environments. IEEE Communications, vol. 14, no. 2, Feb. 1997.
- [Wood 93] M. Wood, Replicated RPC using Amoeba Closed Group Communication. 13th Conference on Distributed Computer Systems, IEEE Computer Society, pp. 499-507, Pittsburgh, PA, USA, 1993.

[Xu et al. 89] A. Xu and B. Liskov, *A design for a fault-tolerant, distributed implementation of Linda*, in Proc. Nineteenth Int. Symp. Fault-Tolerant Comput., pp. 199-206, June 1989.

[Yap et al. 88] K. Yap, P. Jalote and S. Tripathi, *Fault Tolerant Remote Procedure Call*, 8th International Conference on Distributed Computing Systems, IEEE Computer Society, pp. 48-54, 1988.

Index

A

- active replication* 21
- arbitrary timing fault*..... 8
- authenticated byzantine fault*..... 7

B

- byzantine fault*..... 7

C

- checkpoint* 46
- client*..... 11
- cold standby*..... 6
- component*..... 4
 - atomic* 4
 - replaceable hardware* 8
- computer service* 10
- confidence degree*..... 56
- coordinator/cohort* 22
- crash fault* 8

D

- deterministic execution* 21
- distributed system* 10

F

- fail-silent*..... 8
- fail-stop*..... 8
- failure* 2
- fault* 2
 - detected*..... 3
 - latent*..... 3
 - observable*..... 3
- fault-floor* 4
- fault-tolerant application* 9

- fault-tolerant software system*..... 9
- fault-tolerant system* 2

G

- group communication* 11
 - closed group*..... 12
 - dynamic group*..... 12
 - intergroup communication* 12
 - intragroup communication* 12
 - open group* 12
 - process group*..... 11
 - static group*..... 12

H

- highly available system* 6
- hot standby* 7

I

- incarnation number*..... 57
- incorrect computation fault* 8

M

- manual masking*..... 6

N

- no fault*..... 8

O

- omission fault* 8

P

- passive replication*..... 21
- primary-backup* 21

R

- R²PC* 36

<i>redundancy</i>	5	<i>state machine</i>	41
<i>reliable remote procedure call</i>	21	<i>symptom</i>	2
<i>remote procedure call</i>	14	<i>system</i>	2
<i>at least once semantics</i>	15	<i>system boundary</i>	4
<i>at most once semantics</i>	15		
<i>Exactly once semantics</i>	15	T	
<i>resilience degree</i>	7	TS	<i>See tuple space</i>
<i>RPC</i>	<i>See remote procedure call</i>	<i>tuple</i>	16
<i>RRPC</i>	<i>See reliable remote procedure call</i>	<i>tuple space</i>	16
S		W	
<i>server</i>	10	<i>warm standby</i>	6
<i>span of concern</i>	4		