1998

Implementing a fault-tolerant C-Linda in OS/2

Stephan Bettermann

Edith Cowan University

Recommended Citation


This Thesis is posted at Research Online.
https://ro.ecu.edu.au/theses/1006
Edith Cowan University

Copyright Warning

You may print or download ONE copy of this document for the purpose of your own research or study.

The University does not authorize you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.

You are reminded of the following:

• Copyright owners are entitled to take legal action against persons who infringe their copyright.

• A reproduction of material that is protected by copyright may be a copyright infringement. Where the reproduction of such material is done without attribution of authorship, with false attribution of authorship or the authorship is treated in a derogatory manner, this may be a breach of the author’s moral rights contained in Part IX of the Copyright Act 1968 (Cth).

• Courts have the power to impose a wide range of civil and criminal sanctions for infringement of copyright, infringement of moral rights and other offences under the Copyright Act 1968 (Cth). Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.
Implementing a Fault-Tolerant C-Linda in OS/2

by

Stephan Bettermann, B. App. Sc. (Hons)

A dissertation submitted in partial fulfilment of the requirements for the award of

Master of Science (Computer Science)

at the
School of Computer and Information Science
Edith Cowan University

under the supervision of

Dr Thomas O’Neill and Mr Garry Chaplin

4th December 1998
Abstract

The parallel paradigm is accepted by computer scientists as playing a role of increasing importance in their discipline area. To enable operation in the parallel paradigm, Linda may be used to extend existing and well understood programming languages such as C, Eiffel, Prolog, and Lisp. Linda is not a complete programming language in its own right, but rather a coordination language which is injected into a host programming language thereby creating a parallel dialect of that host language.

The behaviour of every component, whether hardware or software, may be defined by a service specification. Thus, a failure may be defined as occurring when a component behaves in a manner not consistent with its service specification. The ability of a system to tolerate failures of any of its components is called fault-tolerance. The probability of failure occurring is higher in parallel systems and distributed systems than in single-processor systems due to increased hardware and software complexity. Furthermore, as many computations on such systems are long lived, the ability to tolerate failures is particularly important.

Personal Computers represent an abundant computing resource. Connecting multiple Personal Computers via a Local Area Network, a common practice in medium to large scale organisations, results in a computing resource which is typically idle, but which is readily useable in the parallel paradigm. By exploiting this idle resource a fault-tolerant Linda may be implemented. Such a Linda implementation achieves low-cost fault-tolerance by distributing and replicating a tuple space in a ring arrangement over the Local Area Network.

This thesis describes the development of a fault-tolerant Linda injected into the ANSI C programming language running on a Local Area Network of Personal Computers, each one running under OS/2. The Local Area Network is accessed using NetBIOS, as provided by OS/2’s Communication Manager.
DECLARATION

I certify that this thesis does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any institution of higher education and that, to the best of my knowledge and belief, it does not contain any material previously published or written by another person except where due reference is made in the text.

Signature: ____________________________

Date: 26.11.1999
Acknowledgements

Of the three people that have been my supervisors, it is Dr Thomas O’Neill that I wish to thank most profoundly. Your tireless, tenacious, and diligent efforts are something that I will always admire, and never forget. But most of all I thank you for your friendship. I also wish to thank Garry Chaplin, my second supervisor, for his patience, and for taking on the role vacated by Dr Geoff Sutcliffe who left the department soon after I started this research.

I would also like to thank my fellow students Vincent Phillips, Andrew Mehnert, and Yoon Choi for their encouragement, the competitive challenges that they have presented, but mostly for the companionship. All of you have had an immense impact on my life.

Without the continuing support of my parents Karl-Heinz and Petra Bettermann this thesis would not have been possible. I thank you whole-heartedly for everything that you have done, and hope that one day I’ll be able to reciprocate by being to my children what you are to me.

But it is my wife Jin to whom I am most indebted. I thank you for the love, companionship, encouragement and support that you have given so freely. I am eternally grateful for your prayers, for your faith, and for the strength and comfort that you have shared with me, and that have enriched my life. Your contribution has been immeasurable, and I will value it, and you, forever.
# Table of Contents

Abstract .................................................................................................................. iii

1 Introduction ........................................................................................................... 1

   1.1 The Background of the Study ................................................................. 1
       'Distributed' or 'Parallel'? ............................................................... 1
       The Linda Model ............................................................................. 2
       The OS/2 Operating System .......................................................... 3
       Fault-Tolerance .................................................................................. 3

   1.2 The Significance of the Study ............................................................... 4

   1.3 Research Objectives ............................................................................. 4

   1.4 Statement of Study ............................................................................. 4

   1.5 Organisation of Thesis ....................................................................... 5

2 Literature Review ........................................................................................... 6

   2.1 Literature on Linda ............................................................................. 6
       Coordination Language ...................................................................... 8
       Linda Implementations ..................................................................... 9
       Discussion ......................................................................................... 10
       Linda Derivatives ............................................................................ 10

   2.2 Fault-Tolerance .................................................................................... 11

   2.3 Fault-Tolerant Linda Implementations ............................................. 14

3 Theoretical Framework ................................................................................... 16

   3.1 OS/2 Facilities .................................................................................... 16
       Inter-process Communication and Synchronisation ...................... 16
       Event Semaphores ......................................................................... 17
       Mutex Semaphores ......................................................................... 17
       Muxwait Semaphores .................................................................... 18
       Pipes .................................................................................................. 18
       Dynamic Link Libraries ................................................................ 19

   3.2 The Communication Manager's NetBIOS Interface ......................... 20

   3.3 The Linda Paradigm ............................................................................ 25
       The Tuple Space ............................................................................... 25
       The Linda Operators ................................................................. 26
           out .............................................................................................. 26
           in ................................................................................................ 26
           rd ............................................................................................... 27
           eval ............................................................................................ 27
           inp ............................................................................................. 27
           rdp ............................................................................................ 28
1 Introduction

This thesis describes the development of a fault-tolerant C-Linda for a Local Area Network (LAN) of Personal Computers (PCs) running the OS/2 operating system. The first section of this chapter introduces relevant background information; the second places the study into context by observing its significance; the third outlines the research objectives; the fourth defines the statement of study; and, finally, the fifth describes the organisation of the remainder of the thesis.

1.1 The Background of the Study

The background necessarily encompasses the following key components: namely, distributed/parallel computing in general, Linda, the OS/2 operating system, and fault-tolerance. Each of these components is now discussed further.

'Distributed' or 'Parallel'? 

Parallel processing may be defined to occur when a task is divided into multiple subtasks and some of these subtasks execute concurrently. Distributed processing on the other hand may be defined to occur when a task is divided into multiple subtasks and these subtasks are distributed over multiple processors. These processors may or may not execute concurrently the subtasks assigned to them.

A distributed system may consist of a varying number of heterogeneous processors in diverse geographic locations, connected via an unreliable low-speed communications medium. In comparison, a parallel system tends to consist of a fixed number of homogeneous processors in very close geographic proximity connected via a reliable high-speed communications medium. Both systems may be used to perform distributed and parallel processing.

The environment, for which the development described in this thesis is targeted, of a Local Area Network of PCs may readily be defined to be a distributed system as it satisfies the key criteria of a distributed system’s definition, as given above.
Introduction

The Linda Model

The parallel paradigm is accepted by computer scientists as playing a role of increasing importance in their discipline area (Weston, 1990; Jellinghaus, 1990; Borrmann, Herdieckerhoff, and Klein, 1988), within which "many problem domains lend themselves naturally to concurrency" (Sebesta, 1989, p. 345). Sommerville (1989, p. 184) agrees when stating that "there are some applications ... where a parallel approach is a completely natural one." These naturally parallel problems are more easily and appropriately expressed in a parallel programming language than in a sequential one (Bettermann, 1992, p. 19; Carriero & Gelernter, 1989; Borrmann & Herdieckerhoff, 1989, p. 1).

To enable operation in the parallel paradigm, Linda, first proposed by David Gelernter of Yale University in 1983 (Whiteside & Leichter, 1988), may be used to extend existing and well understood programming languages such as C, Eiffel, Prolog, and Lisp. By doing so Linda eliminates the need to create specific programming languages for that paradigm. Linda is not a complete programming language in its own right, but rather a coordination language which may be injected into a programming language, called the host language, thereby creating a parallel dialect of that host language. The injection of Linda is achieved by providing a set of operators that manipulate a common data area called the tuple space. These operators supply the capabilities of inter-process communication, synchronisation, and process creation that are required for parallel processing.

As expressed by Andrews (1975) quoted in Gelernter (1985, p. 80) there are:

Three kinds of mechanisms, and three corresponding models of concurrent programming: monitors (shared variables), message passing, and remote operations.

Gelernter (1985, p. 80) argues that Linda "is sufficiently different from all three to constitute a fourth model." Berndt (1989, p. 1) expresses the advantage of this fourth model by revealing that:

One of the difficulties associated with parallel programming is communication among cooperating processes. Linda simplifies the problem of parallel process communication by supporting an uncoupled programming style. All process communication is via Linda operations on tuple space [sic]. This uncoupled process model frees the programmer from the low-level concerns about explicit synchronisation common in many parallel programming systems.
Introduction

The OS/2 Operating System

Personal computers represent an abundant computing resource, with OS/2 being one operating system available for PCs (Orfali & Harkey, 1992). OS/2 is a pre-emptive multi-tasking operating system, allowing processes to execute in logical concurrency. Processes in OS/2 exist in distinct address spaces: therefore, for processes to communicate, they must do so using an inter-process communication (IPC) facility.

The capabilities of inter-process communication, synchronisation, and process creation, as provided by OS/2's Application Program Interface (API), are limited to processes within a single OS/2 machine (i.e., a PC running OS/2). Processes existing on separate OS/2 machines must use another mechanism, such as the Communication Manager, to communicate and synchronise.

Fault-Tolerance

According to Cristian (1991, p. 57) "computing systems consist of a multitude of hardware and software components that are bound to fail eventually". However, the overheads of making computing systems resilient to failure has always been considered too high, "given the low probability of such events" (Bal, 1992, p. 37).

Distributed systems "are systems based on a set of separate computers [each of which is a computing system in its own right] that are capable of autonomous operation, linked by a computer network" (Coulouris & Dollimore, 1988, p. 2). The probability of failure occurring in distributed systems is higher than in single-processor systems because of the complex nature of such systems (Hariri, Choudhary & Sarikaya, 1992). Consequently, distributed systems must tolerate such failure if they are to remain practical. The redundant nature of distributed systems provides for cost effective application of fault-tolerance techniques (Hariri, Choudhary & Sarikaya, 1992).

Parallel systems may be implemented on both distributed systems and parallel computers (Coulouris & Dollimore, 1988). Fischer (1990) deems as reasonable the assumption that parallel computers are not subject to faults. However, such an assumption should not be made about distributed systems, due to their inherently complex nature (Fischer, 1990; Hariri, Choudhary & Sarikaya, 1992). Yet "a distributed implementation of a parallel system is of interest because it can provide an
Introduction

economical source of concurrency, can be easily scaled to match the needs of particular computations, and can be fault-tolerant” (Xu & Liskov, 1989, p. 199).

1.2 The Significance of the Study

Although C is a common host language for Linda injections, the literature presents a dearth of evidence of an embedding of Linda into ANSI C for a LAN of OS/2 machines. Yet such an implementation is desirable because it may provide us with a low-cost alternative to parallel computers.

Parallel computers are expensive and statically sized in terms of processors and memory capacity. However, we may construct a system capable of parallel and distributed computations from components that already exist in most organisations, namely a LAN of PCs, as demonstrated by the Linda implementation described later. Furthermore, this thesis proposes and demonstrates a method of constructing a distributed system such that it is fault-tolerant.

1.3 Research Objectives

Even though LANs of PCs are substantial and abundant computing resources, and OS/2 is a readily available operating system for such an environment, a fault-tolerant C-Linda for this combination does not currently exist. The goal of this research is to extend knowledge of achieving fault-tolerance in such a commonly used, and therefore highly available, environment.

1.4 Statement of Study

The aim of this thesis is to design, implement and test the means of achieving fault-tolerance in a C-Linda implementation that operates in a LAN of PCs, each one of which is running the OS/2 operating system. The product is referred to as OS/2-C-Linda. The LAN is accessed using NetBIOS, as provided by OS/2’s Communication Manager.
1.5 Organisation of Thesis

Chapter two contains a literature review that explores relevant work by others in the areas of Linda and fault-tolerance. The exploration of Linda-related literature adopts the general direction of Linda’s evolution. The review of literature related to fault-tolerance is restricted to literature relevant to the goals of this thesis.

Chapter three discusses the theoretical framework that, together with the literature review, forms the foundation for the main body of this thesis. The theoretical framework contains an overview of facilities (including NetBIOS) that OS/2 provides to applications, and offers a precise definition of the Linda paradigm, followed by an in-depth coverage of fault-tolerance techniques and a description of a failure taxonomy.

Chapter four closely follows the software development life cycle. As such, it contains an analysis section, a design section, an implementation section, and a testing section. The analysis section defines the problem, identifies a set of requirements, and describes which of the failures introduced in the theoretical framework are tolerated by OS/2-C-Linda. The design section then describes the chosen solution that satisfies the defined requirements. The implementation section details the system architecture, the protocols, and relevant implementation criteria. The concluding testing section outlines proof of functionality.

Chapter five concludes by summarising achievements and indicating future directions for pursuit.

The thesis incorporates two appendices: the first one of these contains the reference list, and the second offers a bibliography of further readings.
2 Literature Review

This chapter contains a literature review that explores relevant work by various authors in the areas of Linda and fault-tolerance. The first section discusses literature relating to Linda, and the second fault-tolerance. Any theoretical basis is revealed only when the review makes this necessary; the next chapter describes the theoretical basis for the work described in this thesis.

2.1 Literature on Linda

Linda was first described in 1983 by David Gelernter from Yale University (Whiteside & Leichter, 1988, p. 192). Since this seminal work, a research group called the Linda Group has come into existence at that institution. Therefore, it comes as no surprise that the largest body of Linda-related literature is attributable to that research group. Moreover, when focusing on the group’s publications, especially literature authored, or co-authored, by David Gelernter, an evolution of Linda-applicable research foci, and of Linda itself, becomes clear.

Early literature, such as Gelernter (1985) concentrates on Linda and its supporting concepts. In that paper Gelernter describes generative communication, “the basis for a new distributed programming language” (Gelernter, 1985, p. 80). Arguing for its acceptance as a distributed programming language, Carriero and Gelernter (1986) “briefly rehash” arguments that Linda as a parallel programming language is “in most cases more powerful and expressive than comparable ones” (p. 111). However, later literature, i.e., Carriero and Gelernter (1986), defines Linda to be a parallel programming language, as opposed to a distributed language. Even later literature, such as Gelernter and Carriero (1992), categorically refutes this by stating that “Linda is not a programming language” (Gelernter & Carriero, 1992, p. 97), but rather a coordination language.

The Linda Group’s research focus then shifts to examine how applications may be constructed using Linda, and for what types of problem Linda solutions may be employable. Specifically, support for a programming style called replicated worker,
also called the *bag-of-tasks* programming paradigm (Bakken & Schlichting, 1991), that uses the *distributed data structures* communication paradigm (Kaashoek, Bal, & Tanenbaum, 1989, p. 175) is identified to be one of Linda's strengths by Ahuja, Carriero, and Gelernter (1986). A replicated worker may be defined to be one of multiple instances of a process, digesting a bag of subtasks due to a partitioning of a task (Bakken & Schlichting, 1991, p. 249). Unlike "most models of parallelism ... [which] assume that a program will be parallelized by partitioning it into a large number of simultaneous activities", Linda's support for distributed data structures permits the parallelism to be achieved "by replication as well as partitioning" (Ahuja, Carriero, & Gelernter, 1986, p. 27).

Since it is deemed simpler to create multiple instances of "one process than to create the same number of distinct processes" (Ahuja, Carriero, & Gelernter, 1986, p. 27), it is claimed by Linda's designers that "parallel programming in Linda is conceptually not harder than sequential programming" (Kaashoek, Bal, & Tanenbaum, 1989, p. 175). However, Kaashoek, Bal, and Tanenbaum (1989) argue that "Linda's support of distributed data structures is at too low a level" (p. 175), and while "Linda is an improvement over more traditional parallel languages, ... it could be improved on some major points" (p. 179). Nevertheless, a number of programming experiments (Ahuja, Carriero, & Gelernter, 1986; Carriero, Gelernter, & Leichter, 1986; Carriero & Gelernter, 1988; Gelernter, 1989) show that Linda may be used effectively for a wide range of problem domains.

Linda research has led to the development of Piranha, a system that harnesses CPU "cycles routinely wasted in local area networks" (Gelernter & Kaminsky, 1992, p. 1). The "Piranha system is, in practice, an execution model and support system for a certain class of Linda programs" (Gelernter & Kaminsky, 1992, p. 1). Carriero, Gelernter, Kaminsky, and Westbrook (1993, p. 1) further classify Piranha as "a general-purpose adaptive parallelism environment"; that is, an environment that supports "parallel computations on a dynamically changing set of processors." Adaptive parallelism has been suggested by Carriero, Freeman, and Gelernter (1993) as an alternative to "typical approaches" for sharing multiprocessors among its users, such as "simple space-sharing and inefficient, restricted forms of time-sharing" (Carriero, Freeman, & Gelernter, 1993, p. 1).

Independently to the development by the Linda Group at Yale, a research group in Italy started on the development of Shared Prolog around 1987 (Ciancarini, 1992,
A Shared Prolog program consists of a set of "Prolog programs that communicate associatively via a shared workspace called [a] blackboard" (Ambriola, Ciancarini, & Danelutto, 1990, p. 40). Upon realising "the similarities between the two languages [Shared Prolog and Linda]", the Italian research group resolved "to close the gap" (Ciancarini, 1992, p. 112) and, subsequently, the Italian researcher, Paolo Ciancarini, was seconded to the Linda Group at Yale. This close collaboration resulted in the creation of PoliS, "an extension of Linda with multiple tuple spaces" (Ciancarini, 1992, p. 110). PoliS, in turn, forms the basis of a new parallel logic language called PoliS Prolog (Ciancarini, 1992, p. 110).

Coordination Language

Gelernter originally classifies Linda as a distributed programming language (Gelernter, 1985, p. 80) and as a parallel programming language (Carriero & Gelernter, 1986, p. 110). Since then, the notion of *programming language* has been refined into the taxonomy proposed by Gelernter and Carriero (1992). They state that programming languages consist of two distinct components, computation language and coordination language. Any programming language that does not contain both components are deemed *incomplete* by them. Within this taxonomy they classify Linda as a coordination language.

The usefulness of separating these two components to the point of having distinct languages that may then be combined to form *complete* programming languages is discussed by Gelernter and Carriero (1992). Butcher and Zedan (1991) believe that the process of combining these two components results in programming languages that violate some of the "points [i.e., simplicity, regularity, orthogonality, abstraction, clarity, information hiding, explicit interfaces, safety, expressivity, and efficiency] which are desirable in any programming language" (Butcher & Zedan, 1991, p. 90). They substantiate this claim by examining how C-Linda "contradicts a number of the principles outlined" (p. 91). However, irrespective of any theoretical correctness, "the idea of coordination language is useful" when designing new parallel languages "because it allows ... [the designer] to neatly separate [sic] the issues pertinent to concurrency and communication from the issues pertinent to data structures and sequential control flow" (Ciancarini, 1992, p. 112).
Linda Implementations

Linda has been injected into a variety of programming languages, in a number of operating environments. This section enumerates some of these injections, of which C is the most common host language.


Besides C, other reported host languages for Linda injections cover the full spectrum of programming languages; i.e., imperative, logic, and object-oriented programming languages. Some examples are the following:

• Modula-2 (Borrmann & Herdieckerhoff, 1989),
• Fortran (Carriero & Gelernter, 1986, p. 236),
• C++, PostScript, Scheme (Carriero & Gelernter, 1989, p. 445),
• Joyce (Pinakis & McDonald, 1991),
• Common Lisp (Gelernter, 1989, p. 34),
• Lisp (Yuen & Wong, 1990),
• muProlog (Sutcliffe & Pinakis, 1990),
• SICStus Prolog (Sutcliffe & Pinakis, 1991; Sutcliffe, 1993),
• Smalltalk (Matsuoka & Kawai, 1988), and
• Eiffel (Jellinghaus, 1990).

As can be seen, Linda injections encompass a broad spectrum of programming languages.

A number of proprietary Linda implementations are also available, such as a "commercial version of TSNet, the first Linda system over a network ...[which runs on]"
a network of SUN workstations under UNIX SUNOS" (Ciancarini & Guerrini, 1993, p. 81). Scientific Computing Associates offer Hypercomputer Linda, an implementation that "runs on LANs and wide-area networks (WANs) and can encompass all sorts of machine architectures" (Gelernter & Philbin, 1990, p. 214).

The environments in which these implementations operate are equally as diverse as the host languages. Linda implementations exist for:

- shared memory multi-computers such as the Encore Multimax, Sequent Balance and Symmetry and Alliant FX/8 (Carriere & Gelernter, 1989, p. 445),
- distributed-memory multi-computers such as the Intel iPSC/2, S/Net (Carriere & Gelernter, 1989, p. 445), Parwell-1 (Bormann & Herdieckerhoff, 1989, p. 5), and
- LANs such as Vax/VMS-based local area nets (Carriere & Gelernter, 1989, p. 445), those of DOS machines (Bettermann, 1992), and "a collection of Sun workstations connected in a local area network" (Pinakis, 1991, p. 1).

**Discussion**

Every known Linda injection into a non-imperative host language has modified Linda's semantics. For example, Matsuoka and Kawai (1988) adapted the Linda paradigm for the object-oriented programming language Smalltalk. They "transformed the Tuple Space Communication Model for better affinity with Object-Oriented computation, and integrated it as an alternative method of communication among ... distributed objects" (Matsuoka & Kawai, 1988, p. 276). Sutcliffe and Pinakis (1990, 1991), and Sutcliffe (1993) also adapted the Linda paradigm in their implementations of Linda injections into the non-imperative host language Prolog. The tuple space in those implementations, for example, has capabilities in addition to those of standard tuple spaces, allowing "unification and Prolog style deductions in the tuple space" (Sutcliffe & Pinakis, 1990, p. 1). Ciancarini (1992, pp. 110-111) points out that although Linda has aspects that are similar to those of logic languages like Prolog, it has fundamental differences, a number of which are enumerated by Ciancarini (1992).

**Linda Derivatives**

Butcher and Zedan (1991) believe that there are "problems with ... [viewing] Linda as an inter-process communication (IPC) mechanism ... [that] is independent of the host language" (p. 91) as proposed by Gelernter and Carriero (1989). Butcher and Zedan
Literature Review

(1991) propose a new language called Lucinda which is a merger of Linda and Russell, "a computation language" (Butcher & Zedan, 1991, p. 90). Because of the problems they believe exist, they do not inject Linda into Russell, but rather combine the two languages to form Lucinda. However, Gelernter and Carriero (1992) address some of the problems described by Butcher and Zedan (1991).

The concepts introduced and utilised by Linda have been adopted for purposes other than as a coordination language. By way of example, Laura (Tolksdorf, 1992) is the application of Linda concepts to an open systems problem.

2.2 Fault-Tolerance

The study of fault-tolerance is too large to include herein a complete review of its literature. Rather, literature relating solely to fault-tolerance in the context of this thesis is discussed.

The following sections first concern themselves with literature that best discuss the concepts of fault-tolerance. The focus of discussion then shifts to programming paradigms and supporting abstractions that may be used to guide the construction of fault-tolerant systems. Lastly, for completeness, literature treating adequate programming support, i.e., the provision for fault-tolerance at the programming language level, is covered.

Cristian (1991) provides a good introduction to the field of fault-tolerance; the paper being a "snapshot of ... [his] understanding of basic concepts and issues in fault-tolerant distributed systems" (Cristian, 1991, p. 76). He introduces a standard terminology, such as service, server, the depends relation, and service specification, and defines a failure taxonomy that hardware and software systems may suffer. Furthermore, he identifies a means of tolerating failures and several approaches to achieve fault-tolerance.

As suggested by Cristian (1991, p. 76), the identification and elaboration of the fundamental concepts of fault-tolerant distributed computing is an essential requirement in order to understand it. Some of the concepts recounted by Cristian (1991, p. 76):

are fundamental to any kind of system ... [whereas others] are specific to fault-tolerant systems. The latter concepts capture the goals of fault-tolerant computing as well as its trade-offs: mask component failure when possible,
and when masking is not possible or economical, ensure that the system has some clearly specified failure semantics.

Heimerdinger and Weinstock (1992) complement Cristian's efforts by presenting a conceptual framework for fault-tolerance that includes discussions of the concepts of fault-tolerance and how systems can fail. Also included is a description of mechanisms typically used to tolerate such failures and guidelines for developing fault-tolerant systems (Heimerdinger & Weinstock, 1992, p. 1). Although most of their description of fault-tolerance concepts coincides with that of Cristian (1991), they refine the definition of the term *failure* into *fault* and *failure* (see section 3.4 for details).

Mishra and Schlichting (1992) provide a more detailed overview of the field of fault-tolerance; indeed, their paper is based in part on the work of Cristian (1991). Their overview consists of an examination of the following four programming paradigms that may be used to construct fault-tolerant distributed systems:

- the object/action model, in which applications are structured as objects, as prescribed by the Object Oriented programming paradigm. Each object has state that is typically "stored on stable storage to survive failures" (p. 4), and a set of operations (actions) that may be performed on it. Each object's "action has two properties that guarantee the atomicity of its execution with respect to both failures and the concurrent execution of other actions" (p. 4). Operations are recoverable as they are "either executed completely or not at all, despite failures", also called totality and the unitary property (p. 4), and serialisable, also called indivisibility, as the concurrent execution of multiple actions has same results as executing them serially (p. 4).

- the primary/backup approach, which requires applications to be organised as collections "of services, each of which is implemented by multiple processes to provide fault-tolerance" (p. 4).

- the state machine approach, where applications are structured as in the primary/backup approach. However, rather than replicating the results of any state changing operation, the state machine approach requires that any state changing operation is also executed on any replicas (p. 4).

- the conversation model, that structures applications as collections "of concurrent processes that communicate by exchanging messages" (p. 4).

Such programming paradigms are necessary because of the recognised difficulty in developing fault-tolerant systems (Bakken & Schlichting, 1991, p. 248; Cristian, 1991,
Without them, "building distributed fault-tolerant systems ... [would] remain an art" (Cristian, 1991, p. 76). Moreover, these programming paradigms use well recognised abstractions, called fault-tolerant services, to realise their goals. Mishra and Schlichting (1992, p. 3) categorise these fault-tolerant services as follows: those that provide functionality "similar to features found in standard systems, but with improved failures semantics .... [and those that] provide consistent information to processes executing on different machines in a distributed system." The first category includes abstractions of:

- stable storage, defined by Mishra and Schlichting (1992, p. 3) as failure free data storage, i.e., fault-tolerant memory or disk storage.
- atomic actions, defined to be multiple computational steps that are seen as a single atomic step, i.e., "an indivisible state transformation" (Mishra & Schlichting, 1992, p. 3). Each of the atomic action's computational steps may be executed on different processors, and at different times. Furthermore, the property of atomicity must hold despite possible failures.
- resilient processes, defined to be processes "that can continue to execute correctly even if interrupted by a failure and then restarted" (Mishra & Schlichting, 1992, p. 22).
- certain types of remote procedure call (RPC), which is defined to be a call to a procedure that resides on a remote processor. Just like an ordinary procedure call, a remote procedure call allows the caller to pass parameters. While the remote procedure executes, the caller is blocked until the remote procedure completes. Upon that completion results may be returned to the caller. The caller becomes unblocked and continues execution.

The second category of fault-tolerant services described by Mishra and Schlichting (1992) includes:

- common global time, a service that facilitates the determination of the causal order of events on different processors. Given two events $\alpha$ and $\beta$, if the execution of $\alpha$ affects the execution of $\beta$, then $\alpha$ and $\beta$ are causally related (p. 7). Typically, the causal relation of events $\alpha$ and $\beta$ executing on the same processor is determined using the local clock, that is, the time at which $\alpha$ executes in relation to the time at which $\beta$ executes. However, the causal relation of $\alpha$ and $\beta$ executing on different processors cannot be determined by the clocks local to their processor because "local clocks can drift relative to one another at a variable and unpredictable rate" (p. 7).
Literature Review

- multicast, a service that provides the ability to send messages reliably to all processes within a group of processes.
- membership, a service that provides, to a system’s processes, a consistent view of processes functioning within the system. The membership service has proven itself “to be one of the most fundamental services in fault-tolerant distributed systems, simplifying many problems” (Mishra & Schlichting, 1992, p. 22), such as the multicast service.

Programming languages such as FT-SR and Argus provide support for the construction of fault-tolerant software at the programming language level (Schlichting & Thomas, 1992; Schlichting & Thomas, 1995; Bal, 1992). FT-SR, based on the programming language SR, supports the construction of fail-stop atomic objects, which Schlichting and Thomas (1992) argue is “a ‘lowest common denominator’ for the various programming paradigms” (p. 1). The other example, Argus, uses the object/action model. An example of a system that may be used in the development of fault-tolerant software is described by Zhou (1993). His SRPC system is “a simple and fault-tolerant remote procedure call system” (p. 2).

Although operating systems can transparently make applications fault-tolerant, Bal (1992), through experience gained by work described in his paper, concludes that implicitly implementing fault-tolerance “is potentially more efficient, but also requires more work from the programmer” (p. 37). This is due to the fact that every program solution is unique and thus the manner in which it can be made fault-tolerant varies between problems. This implies that software required to be fault-tolerant must be designed as such from the start; this is demonstrated by the fourth experiment described in his paper.

2.3 Fault-Tolerant Linda Implementations

There are several known fault-tolerant Linda implementations. The first implementation, due to Xu and described in Xu (1988) and Xu and Liskov (1989), limits the scope of fault-tolerance to the tuple space only; no provision is made to extend fault-tolerance to any Linda clients that are using the tuple space. Although a second implementation, due to Patterson, Turner, Hyatt, and Reilly (1993), addresses that issue, their approach requires that all processes of a Linda client have overlapping lifetimes. This requirement violates the property of temporal uncoupling, a property
fundamental to Linda. Furthermore, their description suggests that all processes of a Linda client must have overlapping lifetimes in order for their implementation to provide fault-tolerance to the Linda clients.

Two other known implementations provide fault-tolerance to Linda clients by providing a fault-tolerant tuple space and a set of operators that may be used by Linda clients to aid fault-tolerance. The two implementations differ in their approach. The implementation by Bakken and Schlichting (1993, 1995) uses replication, as opposed to the implementation, described in Silva, Veer, and Silva (1995), which uses checkpointing to achieve fault-tolerance.

The implementation described in this thesis is a vehicle to prove that a particular architecture of structuring the tuple space satisfies the requirements of a fault-tolerant tuple space. While Bakken and Schlichting (1993) define a method of extending fault-tolerance to Linda clients (i.e., atomic execution of tuple space operations), this issue is not paramount to the goals defined for this thesis.
3 Theoretical Framework

This chapter defines the theoretical framework within which OS/2-C-Linda operates. The first section in this chapter covers the facilities used by OS/2-C-Linda that are not provided by the ANSI C libraries, but by OS/2. Section two describes the NetBIOS interface that OS/2-C-Linda uses to communicate over a LAN. Section three describes the Linda paradigm that OS/2-C-Linda implements. The last section, section four, concerns itself with fault-tolerance.

3.1 OS/2 Facilities

OS/2-C-Linda uses two OS/2 facilities: Inter-process Communication and Synchronisation, and Dynamic Link Libraries. Although these facilities exist in other operating systems, their implementation in OS/2 is described briefly.

Inter-process Communication and Synchronisation

The smallest unit of execution in OS/2 is a thread. A process, then, consists of one or more threads. An additional abstraction in OS/2 is a session, which is a collection of processes, just as a process is a collection of threads.

Processes in OS/2 exist in distinct address spaces. Therefore, they must use an inter-process communications facility to communicate and synchronise. Furthermore, the threads of a process may need to communicate with, synchronise with, and create, other threads. To that end OS/2 provides:

- semaphores: namely, event semaphores, mutual exclusion semaphores (mutex semaphores) and multiple wait semaphores (muxwait semaphores). Each of these may be either named or anonymous, both of which may be shared by multiple processes,
- pipes: either named pipes (of which multiple instances may exist) or unnamed pipes. Named pipes may be shared by multiple processes, whereas unnamed ones may be shared by related processes only, i.e., a process may share an ancestor created named pipe with the creator or any of the creator’s descendant processes,
- queues, which are always named,
Theoretical Framework

- shared memory. Each allocated shared memory block may be a named shared memory block or an unnamed shared memory block.

Of the facilities enumerated above, OS/2-C-Linda uses the first two only for communication and synchronisation. Thus, only these are described herein.

Event Semaphores

An event semaphore signals an event, and this semaphore is closest to Dijkstra's semaphore semantics (Sebesta, 1989). Event semaphores may be in one of two states: posted or reset. When in a reset state, any thread of execution waiting on the event semaphore (i.e., waiting for an event to occur) is blocked until the event semaphore becomes posted. Each event semaphore maintains a post count; when in a reset state the post count is zero. Every event semaphore may be posted any number of times, each time incrementing the post count by one. Unlike Dijkstra's semaphore semantics, when a thread waiting on an event semaphore is unblocked, because the semaphore is posted, the post count is not decremented. Rather, the event semaphore must be reset. When doing so, the event semaphore is placed in a reset state, the post count is reported and then set to zero.

The reported post count may be used to emulate Dijkstra's semaphore semantics. Dijkstra's V operator may be implemented by posting the event semaphore. Dijkstra's P operator requires a private post count, initially zero. When P is called and the private post count is zero, P waits on the event semaphore, blocking until the event semaphore is posted, possibly with a call to V. Upon unblocking, P resets the event semaphore and sets the private post count to the reported post count. When P is called and the private post count is greater than zero, P decrements it and returns without waiting or resetting.

Mutex Semaphores

A mutex semaphore is used to guard access to a shared resource. To obtain access to the resource, ownership of the guarding mutex semaphore should be requested. Multiple requests to obtain ownership of a mutex semaphore are queued by OS/2 and satisfied in first-in-first-out (FIFO) order. Upon completing access to the shared resource, the caller must release ownership of the mutex semaphore so that other threads may obtain it.
Theoretical Framework

Muxwait Semaphores

A muxwait semaphore contains a collection of mutex or event semaphores. However, a muxwait semaphore may contain only one type of semaphore at any particular instant in time. At the time of creating a muxwait semaphore, the caller must specify whether to impose either *any semantics* or *all semantics* on the new muxwait semaphore. The any semantics imposition implies that OS/2 posts the muxwait semaphore whenever any contained semaphore is posted or released. Conversely, the all semantics imposition implies that OS/2 posts the muxwait semaphore only when all contained semaphores are posted or released.

When a muxwait semaphore becomes posted, those mutex semaphores that caused the posting become owned by the thread that is waiting, or waits next, on the muxwait semaphore. The table below shows when a muxwait semaphore, in its varying forms, is in a posted state, and which mutex semaphores become owned. At all other times it is in a reset state.

Table 3.1: Muxwait Semaphore Semantics.

<table>
<thead>
<tr>
<th>Event Semaphores</th>
<th>Mutex Semaphore</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Any</strong></td>
<td>The muxwait semaphore is posted when any event semaphore is in a posted state. The muxwait semaphore is posted when any mutex semaphore is in a released state. The caller owns the mutex semaphore that caused the muxwait semaphore to be posted.</td>
</tr>
<tr>
<td><strong>All</strong></td>
<td>The muxwait semaphore is posted when all event semaphores are in a posted state. The muxwait semaphore is posted when all mutex semaphores are in a released state. The caller owns all mutex semaphores.</td>
</tr>
</tbody>
</table>

Pipes

The semantics for named and unnamed pipes differ substantially. Named pipes may be inbound (the pipe may be read from), outbound (the pipe may be written to), or duplex (effectively two pipes, one inbound, the other outbound). Furthermore, named pipes may be shared by unrelated processes, and any number of instances of the named pipe may be created. Conversely, unnamed pipes may be read from and written to from either end. When a thread attempts to read from a pipe, named or otherwise, that currently contains no data, the thread is blocked until some other thread writes data into the pipe.

Both named and unnamed pipes have buffers that hold the current contents of the pipe. In OS/2 2.x the size of these buffers is restricted to $2^{16}-1$ bytes. Unnamed pipes
Theoretical Framework

have a single buffer only, as do inbound and outbound named pipes. However, duplex named pipes have two buffers.

An event semaphore may be attached to either end, or both ends, of any number of named pipes. When any activity occurs on a named pipe's end, to which the semaphore is attached, OS/2 posts that event semaphore. Any number of event semaphores may be attached to any number of pipes. However, a single event semaphore only may be attached to a particular named pipe's end.

Dynamic Link Libraries

Typically, the executable code of an application exists as a single file. The source code for the application is compiled into object code and, subsequently, linked to the underlying operating system or hardware to create the single executable image. The advantage of such a model is that all the executable code is readily available in memory for execution. The disadvantages are that:

• changes to one part of the system require re-linking the entire application, making such a model an inflexible one, and
• multiple instances of an application result in multiple copies of its executable image in memory.

By using Dynamic Link Libraries (DLLs), the executable code of an application may be distributed over several DLLs. The DLLs required by an application are either dynamically or statically linked to the executable image. If dynamic linkage is employed, the application itself is responsible for loading the DLL into memory and for obtaining the address of each routine that is to be called. If static linkage is employed, the operating system is responsible for loading the DLL into memory; each DLL may be loaded when the application starts or only when routines within the DLL are called. This model has the following advantages:

• changes to parts of the application do not require re-linkage of the entire application,
• multiple instances of the same application may share DLLs. When another instance of an already executing application starts, the DLLs may already exist in memory. DLLs must specify whether each instance operates within its own address space, or whether all instances share the same address space,
• the operating system may swap out infrequently used DLLs, thereby freeing memory,
Theoretical Framework

• applications may be modified at run-time by updating DLLs currently not loaded,
• applications may adapt themselves to the system; facilities not available may easily be detected and an alternative course of action may be adopted.

The disadvantages are:

• the added overhead of dynamically linking the application at load-time or run-time,
• the increased possibility of errors. An application may attempt to load a DLL at run-time that does not exist on the system,
• applications no longer consist of a single executable image, but an executable image and a collection of supporting DLLs.

OS/2 allows each DLL to define an initialisation routine and a termination routine. A DLL must specify whether OS/2 executes the initialisation routine when it creates the first instance of the DLL, or every time it creates an instance of the DLL. It must also specify whether OS/2 executes the termination routine when all DLL instances have terminated, or every time a single DLL instance terminates.

3.2 The Communication Manager's NetBIOS Interface

NetBIOS provides interfaces that applications may use to access a communications medium such as a Token Ring LAN. Two sufficiently high-level interfaces were considered, namely the session support interface and the datagram support interface (IBM, 1990, p. 1-10). Session support is “provided by the session layer” (IBM, 1990, p. 4-2) of the ISO Reference Model, and provides point-to-point connections, called sessions, which applications may use to exchange information reliably. Datagram support “goes directly to the link layer” (IBM, 1990, p. 4-2) of the ISO Reference Model, and provides “a simple but unreliable transmission service with powerful broadcast capabilities” (Orfali & Harkey, 1992, p. 405) called datagrams.

The Communications Manager provides NetBIOS via a DLL called ACSNETB.DLL which exports a single Application Program Interface (API) called (not unexpectedly) NETBIOS. The C prototype for this API is:

extern unsigned NETBIOS( char * );
#pragma linkage( NETBIOS, far16 pascal )
Theoretical Framework

NETBIOS expects as its only parameter the address of a Network Control Block (NCB), and returns a 32-bit unsigned integer. The NCB, a 68 byte block of memory that contains the fields presented in Table 3.2 below, is used to specify the action requested when calling NETBIOS. An NCB that has been passed to NetBIOS, and is in the process of completing, is said to be posted. Similarly, when the request that was expressed by the formulated NCB completes, the NCB is said to be completed.

Table 3.2: The Network Control Block.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bytes</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCB_COMMAND</td>
<td>1</td>
<td>Byte</td>
<td>NetBIOS command</td>
</tr>
<tr>
<td>NCB_RETCODE</td>
<td>1</td>
<td>Byte</td>
<td>NetBIOS return code</td>
</tr>
<tr>
<td>NCB_LSN</td>
<td>1</td>
<td>Byte</td>
<td>Local Session Number field</td>
</tr>
<tr>
<td>NCB_NUM</td>
<td>1</td>
<td>Byte</td>
<td>Local Name Number field</td>
</tr>
<tr>
<td>NCB_BUFFER</td>
<td>4</td>
<td>Address</td>
<td>Address of message buffer in segment:offset format</td>
</tr>
<tr>
<td>NCB_LENGTH</td>
<td>2</td>
<td>Word</td>
<td>Length of message buffer</td>
</tr>
<tr>
<td>NCB_CALLNAME</td>
<td>16</td>
<td>Bytes</td>
<td>Name on local or remote NetBIOS</td>
</tr>
<tr>
<td>NCB_NAME</td>
<td>16</td>
<td>Bytes</td>
<td>Local name</td>
</tr>
<tr>
<td>NCB_RTO</td>
<td>1</td>
<td>Byte</td>
<td>Receive timeout</td>
</tr>
<tr>
<td>NCB_STO</td>
<td>1</td>
<td>Byte</td>
<td>Send timeout</td>
</tr>
<tr>
<td>NCB_POST</td>
<td>4</td>
<td>Address</td>
<td>Address of post routine in segment:offset format</td>
</tr>
<tr>
<td>NCB_ADPTR_NUM</td>
<td>1</td>
<td>Byte</td>
<td>Adaptor number</td>
</tr>
<tr>
<td>NCB_CMD_CMPL</td>
<td>1</td>
<td>Byte</td>
<td>Command status</td>
</tr>
<tr>
<td>NCB_RESERVE</td>
<td>14</td>
<td>Bytes</td>
<td>Reserved area for all commands except NCB.RESET</td>
</tr>
</tbody>
</table>

(IBM, 1990, p. 4-4)

OS/2 2.x operates within a flat 32-bit address space. The Communications Manager, being a 16-bit application, operates within a segmented address space. A segmented address space is accessed using pairs of 16-bit addresses, typically denoted as 16:16 addresses or segment:offset addresses. When an application operating within
Theoretical Framework

A flat address space calls an API operating within a segmented address space, *thunking* must take place. During this process:

- flat addresses are converted to segmented addresses,
- parameters are copied,
- pointers are converted,
- segment boundary crossings are checked,
- the stack is converted,
- the API is called,
- the context is restored, and
- control is returned to the caller.

All of these steps must be taken when calling the NetBIOS API because it is provided by a DLL existing within a segmented address space, and NETBIOS accepts as its only parameter the address of an NCB. Compilers that recognise the need to thunk and which automatically insert the required code into the executable image simplify the task of thunking. However, the contents of the NCB cannot be checked by the compiler; hence, the caller must ensure the following requirements pertain:

- any addresses placed into the NCB are segmented addresses, and
- the memory blocks pointed to by any addresses in the NCB are segment aligned, i.e., they do not cross segment boundaries.

The former requirement may be satisfied by coercing flat addresses into segmented addresses. The latter requirement may be satisfied by using *tiled memory* that OS/2 guarantees not to cross segment boundaries.

Each request may be posted in one of two modes: NO_WAIT mode and WAIT mode. A request posted in WAIT mode will cause the calling thread to block until the request completes, either by succeeding or by failing. Upon completing, the NCB’s NCB_RETCODE field contains a completion code which either:

- indicates that the request completed successfully, or
- identifies the cause of the request’s failure.

Contrariwise, posting a request in NO_WAIT mode causes the NetBIOS API to return immediately, thus not blocking the calling thread. The NCBs of requests posted in this manner must exist undisturbed until the command completes, i.e., its contents may be queried but not changed. Initially, the NCB’s NCB_RETCODE field contains a status code that indicates the NCB has not yet completed. When the request eventually completes, the NCB’s NCB_RETCODE field contains the final completion code of the
Theoretical Framework

NetBIOS request. Calling threads may poll this field, waiting for the request to complete. However, this is not a suitable strategy in a multi-threaded operating system as the CPU would be exploited for polling, rather than being available for other tasks.

NetBIOS provides an alternative, although the exact mechanism is unique for each NetBIOS implementation. Nevertheless, typically the caller may store the address of a function, called a callback, in the NCB's NCB_POST field. When the NCB completes, NetBIOS executes the function referred to by the NCB's NCB_POST field. Context is provided to the callback via registers: the implementation available to this research passes the address of the NCB to the callback via the DS:EX register pair. Furthermore, the NetBIOS implementation available to this research exists within a 16-bit DLL. Thus, the callback function must be a 16-bit function as NetBIOS does not provide a mechanism for specifying that the callback function is a 32-bit function and, thus, that thunking must occur when calling the 32-bit function. Since no 16-bit compilers are available to this research, and therefore 16-bit functions cannot be generated, this is not possible. Therefore, this research may not post requests in NO_WAIT mode.

Session support provides sessions which are point-to-point connections between processes identified by NetBIOS names. NetBIOS provides two types of such names: unique names and group names. Processes executing on a NetBIOS station (a PC running NetBIOS) may add any number of unique and group names to a data structure called the local name table that each NetBIOS station maintains. However, the NetBIOS implementation available to this research restricts the total number of names that may be defined in all local name tables to 255.

NetBIOS guarantees the distinctiveness of its unique names. Furthermore, an established unique name may not be defined as a group name and visa versa. However, a group name may be shared by multiple processes and may thus appear in multiple name tables. When a process attempts to add a name (referred to as a candidate name), the local station asks all other stations whether the candidate name appears in their local name table as either a unique name or a group name. Since stations are not aware of each other's existence, a protocol involving considerable timeout and retry values is used: this causes the long execution delay that makes the adding of names an expensive operation in NetBIOS. Only upon receiving no positive response after a defined number of timeout/retry broadcasts does the local station add the candidate name to the
local name table and return success to the calling process. A name is removed from the local name table only when the process that defined the name deletes the name.

A session between two processes (called *session partners*) is established by mutual consent: i.e., both processes must indicate the desire to establish a session. They do so in an ordered manner. One process (the receiver) indicates that it is willing to accept calls (either from any name, or a name that it specifies) to a name that it has defined. The second process (the caller) calls the name that the first process has indicated willingness to accept calls on. When the name of the caller matches the name specified by the receiver, or when the receiver has indicated willingness to accept a call from any name, a session is established.

Each session is identified by a *session number* that is added to the *local session tables* of the caller and the receiver; the NetBIOS implementation available to this research restricts the total number of concurrent session to 255. A session is terminated when either process terminates the session. Although the session number is removed from the local session table when a session partner terminates the session, the session number is not removed from the local session table of the session partner’s station. Thus, both session partners need to terminate the session in order to remove the session number from the local session tables of both stations.

Sessions facilitate the reliable exchange of data as *messages*. A process may send data to another process through a session by packaging the data into a single contiguous block of memory and sending it as a message. The receiving process must provide to NetBIOS a single contiguous block of memory into which to receive the message. This block of memory should be large enough to accommodate the received message. Should the message received be larger than the block of memory provided, the block of memory is filled with as much of the message as possible, and the remainder of the message must be received using further NetBIOS commands. Note that the NetBIOS implementation available to this research limits the amount of data that may be sent as a single message through a session to $2^{16} - 1$ bytes, i.e., 65,535 bytes. Any messages larger than this must be fragmented into blocks smaller than $2^8$ bytes before sending, and defragmented upon receipt.

NetBIOS guarantees the integrity of messages: messages are guaranteed to be delivered either with the same content and in same order that they were sent, or not at all. When a message cannot be delivered such that the above axiom holds, NetBIOS notifies the sender by returning an error code. Furthermore, the session through which
the message could not be delivered is terminated (dropped), thus also notifying the session partner, if still alive, that a failure has occurred.

3.3 The Linda Paradigm

Linda is a coordination language that is injected into host languages to create parallel dialects of those languages (Whiteside & Leichter, 1988, p. 192). It consists of a set of Linda operators that manipulate a common data area called the tuple space. Linda is injected into a host language by making the Linda operators available in that host language, typically in the form of subroutine calls.

The Tuple Space

The tuple space is a collection of tuples (see below). Processes communicate and synchronise by placing tuples into, and reading (and possibly removing) tuples from, the tuple space. Any number of identical tuples may exist in the tuple space. Tuples placed into the tuple space remain there until removed. The tuple space is an associative memory in that tuple access is not by use of an address but by type and content. Processes may be created by placing a live tuple into the tuple space.

A tuple is an ordered collection of fields. The number of fields determines a tuple’s arity. Fields are either actual or formal. Actual fields have both a type and a value. Formal fields have a type but no value. The types that fields may be, and thus the values that each field may hold, are inherited from the host language. The type of a tuple is the cross-product of the types of the tuple’s fields.

Syntactically a field is a variable or a value. Assume the following C variable definition exists:

\[
\text{int answer} = 42;
\]

The following is a tuple containing one actual field of type int with value 42:

\[
(\text{answer})
\]

Therefore, the type of the tuple is int. Preceding a field with a ‘?’ identifies it as a formal field. Thus, the following is a tuple containing one formal field of type int:

\[
(\text{?answer})
\]
Theoretical Framework

Similarly, the example below is a tuple containing two fields. The first field is actual of type char[] with value "The answer is", and the second is formal of type int.

( "The answer is", ?answer )

Therefore, the type of the tuple is char[] x int:

The Linda Operators

Linda provides four operators (out, in, rd, and eval) and two predicates (inp, and rdp). The semantics are discussed in the following sections:

out

The out operator places a tuple into the tuple space. The following tuple of type char[] x int:

( "The answer is", 42 )

is placed into the tuple space by the statement:

out( "The answer is", 42 )

in

The argument of the in operator defines a tuple template. The in operator finds a tuple in the tuple space that matches the tuple template, and extracts it from the tuple space. A tuple and a tuple template match if they are of the same arity and the corresponding fields of the two tuples match. Actual fields match if their types and their values correspond, whereas actual fields match formal fields if their types correspond only. Formal fields never match formal fields.

If more than one matching tuple exists in the tuple space, one tuple is chosen nondeterministically. If no matching tuple exists, the process that called the in operator blocks until a matching tuple is placed into the tuple space. If at a later point in time a matching tuple is placed into the tuple space, outstanding requests for that tuple are serviced in a nondeterministic order. Thus, the process that first requested the tuple is not necessarily serviced first.
Actual fields of the extracted tuple are assigned to corresponding formal fields of the
tuple template. Assume the following tuple of type char[] × int exists in the tuple
space:

( "The answer is", 42 )

The following in operation removes this tuple from the tuple space and assigns 42 to
the formal field ?answer:

in( "The answer is", ?answer )

**rd**

The rd operator only differs from the in operator in that rd does not remove a
matching tuple from the tuple space.

**eval**

The eval operator places a live tuple into the tuple space. For each field in a live
tuple a separate process is created to evaluate it. These processes run in parallel with
the process that called the eval operator. Upon completion of all created processes,
the live tuple turns into an ordinary tuple.

Assuming a function square_root exists that returns the square root of its
non-negative integer parameter, the following statement places a live tuple containing
two fields into the tuple space:

```
 eval( square_root( 16 ), "square root of 16" )
```

The second field trivially evaluates to itself. For the first field the process
square_root is created and run in parallel with the process that called eval. When
square_root has evaluated the first field to be of value 4, the live tuple turns into
the following tuple:

( 4, "square root of 16" )

**inp**

The inp operator is the predicate form of the in operator. It is similar to the in
operator except that it does not block if the tuple space does not contain a matching
tuple. Instead, the inp operator indicates failure by returning FALSE.
The rdp operator is the predicate form of the rd operator. If no matching tuple exists in the tuple space rdp indicates failure by returning FALSE.

### 3.4 Fault-Tolerance

Fault-tolerance is a concept associated with hardware and software systems (henceforth, referred to as systems). This section defines what a failure is, presents a taxonomy of failures that may be suffered, outlines the usage of redundancy to tolerate failures, and discusses checkpointing.

A system may be viewed as consisting of multiple components, each being a server that provides services to other servers. A server may implement its services by using services provided by other servers (Cristian, 1991, p. 57).

Every server is subject to a service specification that defines the behaviour the server may exhibit (Cristian, 1991, p. 58; Mishra & Schlichting, 1992, p. 5). A component/server has suffered a failure if it exhibits behaviour not defined within its service specification; Table 3.3 below presents a taxonomy of failures that components may suffer — this table is drawn from Cristian (1991, pp. 58-59), Turek and Shasha (1992, p. 9), and Mishra and Schlichting (1992, p. 5). Fault-tolerant systems extend the service specifications of their components to include failure semantics, that is, the behaviour a component is likely to exhibit when suffering a failure (Cristian, 1991, p. 59). Failure semantics are also called failure models (Schlichting & Thomas, 1991, p. 1; Mishra & Schlichting, 1992, p. 5).
### Table 3.3: Failure Taxonomy.

<table>
<thead>
<tr>
<th>Failure</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Omission Failure</td>
<td>A component omits to respond to an input.</td>
</tr>
<tr>
<td>Timing Failure</td>
<td>A component’s response is functionally correct but untimely.</td>
</tr>
<tr>
<td>Early Timing Failure</td>
<td>A component’s response is functionally correct but early.</td>
</tr>
<tr>
<td>Late Timing Failure</td>
<td>A component’s response is functionally correct but late.</td>
</tr>
<tr>
<td>Response Failure</td>
<td>A component’s response is incorrect.</td>
</tr>
<tr>
<td>Value Failure</td>
<td>The value of a component’s output is incorrect.</td>
</tr>
<tr>
<td>State Transition Failure</td>
<td>A component’s state transition is incorrect.</td>
</tr>
<tr>
<td>Crash Failure</td>
<td>A component that suffers one or more omission failures until it is restarted.</td>
</tr>
<tr>
<td>Amnesia Failure</td>
<td>Upon restarting a component after a crash failure, the component has a predefined initial state.</td>
</tr>
<tr>
<td>Partial-Amnesia Failure</td>
<td>Upon restarting a component after a crash failure, the component partially has the state before the crash failure, and partially a predefined initial state.</td>
</tr>
<tr>
<td>Pause-Crash</td>
<td>Upon restarting a component after a crash failure, the component has the state that it had before the crash failure.</td>
</tr>
<tr>
<td>Halting-Crash</td>
<td>A component that suffered a crash failure and never restarts.</td>
</tr>
<tr>
<td>Byzantine Failure</td>
<td>A component behaves in a completely unspecified manner, and “may make unknown, inconsistent, or even malicious actions” (Mishra &amp; Schlichting, 1992, p. 5).</td>
</tr>
</tbody>
</table>

A component that may exhibit failure semantics α only, e.g. omission failures, is said to have *stronger* failure semantics than a component that may suffer failure semantics α and β, e.g. omission failures and performance failures (Cristian, 1991, p. 60). Furthermore, a component that may suffer all failure semantics, i.e., the weakest possible failure semantics, is said to have *arbitrary* or Byzantine failure semantics (Cristian, 1991, p. 60; Mishra & Schlichting, 1992, p. 5; Turek & Shasha, 1992, p. 15).

The servers of a system form a *dependency hierarchy* based on the *depends relation*. The depends relation exists between two servers α and β, if the correctness of α’s behaviour depends on the correctness of β’s behaviour (Cristian, 1991, p. 57; Heimerdinger & Weinstock, 1992, p. 12).

A component is fault-tolerant if it either masks failures, i.e., continues to operate correctly despite failures (Suzuki, Katayama & Schlichting, 1993, p. 1; Harari, Choudhary & Sarikaya, 1992, p. 51; Cristian, 1991, p. 57; Heimerdinger & Weinstock, 1992, p. 10), or fails detectably by exhibiting clearly defined failure semantics

---

1 Known also as a *performance* failure.
2 Mishra and Schlichting (1992, p. 5) state that Byzantine failures are also known as *arbitrary* failures.
(Cristian, 1991, p. 57). Masking of failures may be achieved using several failure-masking techniques, the one chosen depends on the failures which are to be masked. In practice failure-masking techniques typically fall somewhere in a continuum whose end points are hierarchical masking and group masking (Cristian, 1991, p. 61). These categories are discussed in more detail below.

Heimerdinger and Weinstock (1992) distinguish between faults and failures based on the notion of a system boundary, called the boundary of observation (p. 10). A fault occurs when, in concurrence with Cristian's definition of failures, any component behaves in a manner not consistent with its service specification. The fault propagates up the dependency hierarchy, the path taken being known as the fault trajectory (Heimerdinger & Weinstock, 1992, p. 20), until it is either tolerated or it reaches the boundary of observation (Heimerdinger & Weinstock, 1992, p. 20).

For example, assume that component α consists of component β, and that component β consists of component γ. Assume also that the correctness of α's behaviour depends on the correctness of component β's behaviour. Similarly, the correctness of β's behaviour depends on the correctness of component γ's behaviour. When component γ suffers a fault that is not tolerated, the fault propagates and causes a fault at β. Assuming that the current focus of observation is component β, and thus the boundary of observation is the boundary of component β, a failure occurs when the fault at component β is not tolerated, allowing the boundary of observation to be reached. As α is outside the current scope of observation it is not considered. Observable effects of faults at the system boundary are called symptoms, a failure being the most extreme symptom of a fault (Heimerdinger & Weinstock, 1992, p. 20). However, if the focus of observation should shift to component α, the fault at β that is not tolerated and propagated to α does not breach the system boundary. Thus, a fault only has occurred.

**Masking Techniques**

The most common technique of masking a component's failure is to replicate that component (Cristian, 1991, p. 57, Xu, 1988, p. 19). Should the component fail, the failure can be masked by using replicas of the failed component; the number of replicas being referred to as the resilience degree. However, replicas are not immune to failure and, given an untimely sequence of failures they may fail also (Schlichting & Thomas,
Given a resilience degree of ρ, ρ concurrent failures may be tolerated\(^{(3)}\). Should all ρ+1 instances of the component fail, i.e., the component itself and all replicas, resilience has been exhausted and the failure cannot be masked. Therefore, well defined failure semantics are exhibited, and the failure is propagated. If no component tolerates the failure as it propagates up the dependency hierarchy, the system as a whole fails by exhibiting well defined failure semantics. Thus, the failure is propagated to the external entity which consumes the output from the system, possibly a human user who would view this "as the catastrophic failure of ... the system" (Schlichting & Thomas, 1992, p. 3).

By increasing the number of replicas the probability of resilience exhaustion decreases and, therefore, the probability of successful masking increases. Consequently, the selection of a resilience degree is a trade-off entailing resource limitations, the probability of failures occurring, and "the cost of failures ... [such as] costs incurred by incomplete or incorrect computations" (Hariry, Choudhary & Sarikaya, 1992, p. 50). However, a system may deliberately not be endowed with the ability to tolerate particular failures because these failures are unlikely to occur, or toleration of them are impractical or too difficult, such as Byzantine failures. Thus, fault-tolerance can be "guaranteed relative to some set of assumptions concerning the number and type of failures" (Schlichting & Thomas, 1992, p. 2) only.

Replication may be accomplished \textit{actively or passively} (Mishra & Schlichting, 1992, p. 5). Active replication occurs when any state changing operations performed on a server is performed on all replicas also. Conversely, passive replication requires only that any replicas that intend to provide the service of the failed server obtain the state of the failed server before providing the service. This may be done actively at runtime, or immediately before providing the service. At runtime a server may actively propagate its state to its replicas so that should it fail the replicas have its state. Alternatively, a server may store its state in stable storage so that upon failure any replicas may obtain the state from stable storage.

Another form of replication is \textit{checkpointing}. A component that uses checkpointing to attain fault-tolerance saves its state onto stable storage (stable storage thus holding a replica of the checkpointed component). Upon recovering from a failure, the component restores the state last checkpointed, thereby restoring the state that existed immediately before the failure occurred. A variation of this technique is to checkpoint\(^{(3)}\) for Byzantine failures, \(2p+1\) instances are needed. Given \(p\) failures, the majority of components remain correct (Schneider, 1990, p. 303).
Theoretical Framework

the state periodically only, and then logging all state changing operations performed on
the state since it was last checkpointed in a transaction log. Upon restarting, the last
checkpointed state is restored and all operations performed on the state are replayed
using the transaction log, thereby restoring the state that existed immediately before the
failure occurred. It should be clear that this technique is useful for tolerating Amnesia
and Partial-Amnesia failures only.

Group Masking and Hierarchical Masking

There are two categories of masking techniques: group masking, and hierarchical
masking. What distinguishes these two categories is who tolerates any failures. Given
two servers α and β, where α depends on β, upon β failing a group masking technique
prescribes that β’s replicas tolerate the failure, masking it from α. Conversely, a
hierarchical masking technique prescribes that α tolerates β’s failure, possibly by
requesting the service provided by β’s replicas. However, as Cristian (1991, p. 61)
points out, in practice failure-masking techniques typically fall somewhere in the
continuum of which these two categories are two end points.
This chapter is structured as follows: the problem is defined and requirements are identified; a design is presented that satisfies the identified requirements; and salient elements of the implementation are discussed. The chapter concludes by describing the scheme used to prove that the identified requirements are met.

4.1 Problem Definition

The goal of this thesis is to implement a fault-tolerant Linda for the C programming language. This implementation is to operate in a LAN of PCs, each PC running the OS/2 operating system. Given these components a Linda system, or Virtual Linda Machine, is to be constructed. This Linda system, thus, exists within the LAN of PCs, spanning any number of PCs in the LAN and it provides a tuple space to any Linda client executing on any PC participating in the system.

The requirements that have been set for this implementation may be categorised into the following:

- requirements that must be met and whose forms are rigid,
- requirements that must be met and whose forms are negotiable, and
- requirements that are desirable and whose forms are negotiable.

**Necessary Requirements**

As discussed previously, Linda’s primary function is to provide a common storage facility, called the tuple space, that may be manipulated by processes called Linda clients using a set of Linda operators. Furthermore, the literature review and the theoretical framework reveal that existing fault-tolerant Linda implementations provide, at the least, a fault-tolerant tuple space. All but one of the known fault-tolerant Linda implementations, the one due to Xu, extend fault-tolerance to the Linda clients; that is, by using Linda these clients become fault-tolerant. This thesis restricts itself to providing a fault-tolerant tuple space only, as Xu does.

The resilience degree of the tuple space must be configurable, rather than being hard-coded: thus, it is a parameter to the implementation. Furthermore, the implementation must cater for the dynamic nature that is typical of distributed systems,
such as a LAN of PCs. New PCs may become available on the LAN, and PCs currently available on the LAN may become unavailable, either due to failure or by operator choice.

Failures shall be tolerated once they are detected. The effect of this tolerance is that, given repair time $\Delta$, $p$ failures occurring within time $\Delta$ are guaranteed to be tolerated. However, the number of failures tolerated over lifetime $\lambda$, where $\lambda >> \Delta$, is likely to exceed $p$.

Resilience is to be achieved using replication: that is, the tuple space is replicated over a number of PCs in order to make it resilient to failures. Lastly, the tuple space is to be replicated only to the degree necessary to guarantee the specified resilience degree. It is this requirement that results in this implementation contributing a novel notion: all previous implementations uniformly replicate the tuple space onto all nodes, providing fault-tolerance at the expense of space.

**Negotiable Requirements**

The NetBIOS interface that allows access to the LAN is made available by OS/2’s Communication Manager. Each PC has its own local memory, but there is no shared memory. The only method of communication between the PCs is by use of the NetBIOS interface. Of the two interfaces that NetBIOS provides, session support is to be used. While datagram support provides the facility of broadcasting (a facility necessary for managing the redundancy by which fault-tolerance is achieved), datagram support has weaker failure semantics than session support. Datagram support may suffer omission failures, timing failures, response failures, and all crash failures. Session support is guaranteed to suffer halting-crash failures only.

Session support provides point-to-point communication in the form of sessions. However, a broadcast may be emulated using a number of sessions. Since the goal of this thesis is not to implement yet another reliable broadcast facility, but rather to implement a fault-tolerant tuple space, this emulation suffices. Furthermore, the assumption is made that sessions are assumed not to suffer failures while a broadcast is being made, which uses a number of point-to-point message exchanges via sessions.
**Enhancing Requirements**

The Linda operators required to manipulate the tuple space must be implemented for subsequent use by Linda clients. However, since the goal of this implementation is to demonstrate a means of providing fault-tolerance for a Linda tuple space, and not to be a working system, this requirement is regarded as less crucial. Nevertheless, in order to be able to compose tuple spaces, a minimal implementation of Linda operators is required. Furthermore, because fault-tolerance is to be provided for the tuple space only, and is not to be extended to Linda clients, the implementation of the eval operator (see section 3.3) is not considered relevant because the remaining Linda operators suffice for the demonstration.

A self-evident improvement would be to allow the resilience degree to change at run-time. This might be effected by the software itself upon the realisation that the current resilience degree is inappropriate. Upon determining that the system is not likely to exhaust resilience, it might lower the resilience degree. Similarly, upon realising that the system is likely to exhaust resilience, it might raise the resilience degree. Alternatively, the operator might alter the resilience degree based on observation of system performance, or perception of impending performance coupled with prospective needs.

**The Assumed Failure Model**

Only a subset of the failures contained in the failure taxonomy (see table 3.3) are relevant. For example, NetBIOS masks omission, late timing, and response failures, failures that may occur while engaging in a session, into halting-crash failures. Early timing, amnesia and partial-amnesia failures are not applicable to NetBIOS sessions, whereas byzantine failures are not tolerated. The PCs, as managed by the OS/2 operating system, may suffer late timing, crash (persistent memory, i.e., disk storage, making partial-amnesia and pause-crash failures possible) and byzantine failures. Early timing and omission failures are not applicable to PCs, whereas response failures are assumed not to occur.

Due to the complexity of tolerating byzantine failures, OS/2-C-Linda will not tolerate such failures. Furthermore, as OS/2-C-Linda does not employ any persistent memory, partial-amnesia and pause-crash failures may not be suffered. Thus, as OS/2-C-Linda servers mask omission failures and pause-crash failures into
halting-crash failures, and late timing failures into omission failures (and, therefore, into halting-crash failures), OS/2-C-Linda may suffer halting-crash failures only. Consequently, it is halting-crash failures that are tolerated by OS/2-C-Linda.

4.2 Design Solution

A LAN of PCs is a set of PCs connected via some communications medium. Each PC has a single CPU and some local memory. There exists no shared memory between PCs; the only means of communication between PCs is via the LAN.

The Linda system consists of a collection of Linda servers, each running on a participating PC in the LAN. Not every PC in the LAN may choose to be a part of a Linda system, but those that do may execute multiple instances of a server. Thus, multiple Linda systems may exist concurrently in the LAN, possibly sharing PCs.

The Linda servers collectively maintain the tuple space, and provide access via Linda operators to the tuple space for processes called Linda clients. The operators are made available to clients as a DLL which exports the operators as C functions. When a client starts execution, the DLL connects to a server on its machine through a named duplex pipe that has been created by the server. This connection is realised by the DLL’s initialisation routine opening a named duplex pipe whose name is specified by the environment variable Client_Pipe.

Each server ensures that there exists a named duplex pipe instance with a name specified by the environment variable Client_Pipe, to which clients may connect. Consequently, given \( \chi \) clients connected to server \( \sigma \), there exist \( \chi + 1 \) instances of a named duplex pipe to server \( \sigma \), which means that while a client is connecting to a server any further clients will fail to open a pipe to it. Upon failing to open a pipe to a server, clients suspend for one second before re-attempting to open the pipe. When a client terminates, the DLL’s termination routine causes the server to disconnect and close the pipe.

Distributing and Replicating the Tuple Space

The tuple space is fault-tolerant to a degree of resilience specified as a configuration parameter. Since no shared memory exists between the PCs participating in the LAN, the tuple space must be partitioned amongst the Linda servers. As any Linda server
may suffer any of the defined failures, the tuple space must be replicated as well as being distributed.

Given three Linda servers the tuple space, shown in figure 4.1, is partitioned amongst them. The number of servers is denoted as $v$ — in this example, $v$ is 3.

![Figure 4.1: The tuple space.](image)

The tuple space $T$ is partitioned into three tuple space portions (labelled $T_1$, $T_2$ and $T_3$) and distributed over the Linda servers as shown in figure 4.2:

![Figure 4.2: Distribution of tuple space portions.](image)

Each tuple space portion is then replicated onto $p$ other Linda servers. The diagram in figure 4.3 illustrates the distribution and replication given a resilience degree of 1:

![Figure 4.3: Distribution and replication of tuple space portions.](image)

Primary and replica tuple space portions are mapped onto servers in a strict ring arrangement. Each server maintains the tuple space portion assigned to it (i.e., its primary tuple space portion), as well as replicas of the primary tuple space portions of
up to $\rho$ previous servers. When the number of servers, $v$, is less than or equal to the resilience degree, $\rho$, each server maintains $v-1$ replicas only and resilience cannot be guaranteed. The tuple space is *blocked*, i.e., its state may not change, until enough servers join the system for $v$ to exceed $\rho$.

Therefore, a ring exists of $v$ interlocked groups, where the members of each group embody both the server maintaining the primary of a tuple space portion, and all the servers maintaining replicas of the tuple space portion. Figure 4.4 illustrates a system consisting of 5 servers and a resilience degree of 1. Server 1 maintains the primary of tuple space portion 1, and a replica of server 5's primary tuple space portion. Similarly, server 2 maintains the primary of tuple space portion 2, and a replica of server 1's primary tuple space portion.

OS/2-C-Linda employs active replication (see section 3.4). When the state of any primary tuple space portion changes, the server maintaining that primary also updates the state of all replicas.

\[\text{Figure 4.4: Structuring of tuple space portions.}\]
Due to the strict ring structuring, the distance between the server that maintains the primary of a portion and the server that maintains a replica may be referred to as the order. Consequently, given a system consisting of 7 servers and a resilience degree of 2 as illustrated in figure 4.5, the replica of server 7's primary tuple space portion maintained by server 1 is of order 1. Similarly, the replica of server 6's primary tuple space portion maintained by server 1 is of order 2. Clearly, the primary of a tuple space portion must be of order 0. Thus, for denotational purposes, tuple space portions may be uniquely denoted as $T_{portion,order}$; i.e., server 1 maintains $T_{(1,0)}$, $T_{(7,1)}$, and $T_{(6,2)}$ and server 2 maintains $T_{(2,0)}$, $T_{(1,1)}$, and $T_{(7,2)}$.

Figure 4.5: Tuple space distributed over 7 nodes with a resilience degree of 2.
The corresponding tuple space distribution matrix is given in table 4.1:

**Table 4.1: Tuple Space Portion Distribution Matrix.**

<table>
<thead>
<tr>
<th>Server</th>
<th>Primary</th>
<th>1st Order</th>
<th>2nd Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 1</td>
<td>$T_{(1,0)}$</td>
<td>$T_{(7,1)}$</td>
<td>$T_{(6,2)}$</td>
</tr>
<tr>
<td>Server 2</td>
<td>$T_{(2,0)}$</td>
<td>$T_{(3,1)}$</td>
<td>$T_{(7,2)}$</td>
</tr>
<tr>
<td>Server 3</td>
<td>$T_{(3,0)}$</td>
<td>$T_{(2,1)}$</td>
<td>$T_{(1,2)}$</td>
</tr>
<tr>
<td>Server 4</td>
<td>$T_{(4,0)}$</td>
<td>$T_{(3,1)}$</td>
<td>$T_{(2,2)}$</td>
</tr>
<tr>
<td>Server 5</td>
<td>$T_{(5,0)}$</td>
<td>$T_{(4,1)}$</td>
<td>$T_{(3,2)}$</td>
</tr>
<tr>
<td>Server 6</td>
<td>$T_{(6,0)}$</td>
<td>$T_{(5,1)}$</td>
<td>$T_{(4,2)}$</td>
</tr>
<tr>
<td>Server 7</td>
<td>$T_{(7,0)}$</td>
<td>$T_{(6,1)}$</td>
<td>$T_{(5,2)}$</td>
</tr>
</tbody>
</table>

**Server IDs**

Each server is uniquely identified by a *Server ID* (SID). Every server in the system knows the system’s current topology by maintaining information regarding:

- what servers exist in the system,
- the SID assigned to each server, and
- their position within the system.

The position, sometimes called the *logical SID*, is defined to be its position within the ring in relation to the *oldest* server: i.e., the server that has existed in the system the longest time. Thus, logical SIDs reflect the temporal ordering of assignments of existing SIDs.

**Joining and Leaving the System**

As an ordering exists among the servers, the server with the lowest *logical* SID is called the *elder*. When a new server wishes to join the system, it contacts the elder, which:

- assigns an SID to the new server,
- informs the new server of the current system configuration, including the resilience degree, and
- informs all servers in the system of the new server and the SID assigned to it.

Every server updates its *system map* with the details of the new server (see section 4.3). The new server’s position is by definition the successor of the highest known position,
and the new server's position is therefore determinable by all servers when receiving notification of a new server.

Upon receiving the aforementioned information from the elder, the new server

- initiates a reassignment of tuple space portions, and
- assigns replica management of its, still empty, new tuple space portion to other servers.

Given a resilience degree of $\rho$, the new server:

- requests replicas of the primaries of the $\rho$ previously youngest servers, and
- assigns responsibility to manage replicas of its primary to the $\rho$ oldest servers.

Since the new server now manages a replica of each of the $\rho$ youngest server's primaries, the $\rho$ oldest servers are now each managing a replica of order $\rho+1$. As this is not necessary to guarantee a resilience degree of $\rho$, the $\rho$ oldest servers may drop that replica upon being assigned responsibility for the management of a replica of the new server's primary. The system configuration and entire distribution matrix of the tuple space after another server has joined the system described by figure 4.5 and the distribution matrix described in table 4.1 is given in figure 4.6 and table 4.2 respectively.
Figure 4.6: Tuple space distributed over 8 nodes with a resilience degree of 2.
Table 4.2: Tuple Space Portion Distribution Matrix after Joining.

<table>
<thead>
<tr>
<th>Server</th>
<th>Primary</th>
<th>1st Order</th>
<th>2nd Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 1</td>
<td>$T_{1.0}$</td>
<td>$T_{1.1}$</td>
<td>$T_{1.2}$</td>
</tr>
<tr>
<td>Server 2</td>
<td>$T_{2.0}$</td>
<td>$T_{2.1}$</td>
<td>$T_{2.2}$</td>
</tr>
<tr>
<td>Server 3</td>
<td>$T_{3.0}$</td>
<td>$T_{3.1}$</td>
<td>$T_{3.2}$</td>
</tr>
<tr>
<td>Server 4</td>
<td>$T_{4.0}$</td>
<td>$T_{4.1}$</td>
<td>$T_{4.2}$</td>
</tr>
<tr>
<td>Server 5</td>
<td>$T_{5.0}$</td>
<td>$T_{5.1}$</td>
<td>$T_{5.2}$</td>
</tr>
<tr>
<td>Server 6</td>
<td>$T_{6.0}$</td>
<td>$T_{6.1}$</td>
<td>$T_{6.2}$</td>
</tr>
<tr>
<td>Server 7</td>
<td>$T_{7.0}$</td>
<td>$T_{7.1}$</td>
<td>$T_{7.2}$</td>
</tr>
<tr>
<td>Server 8</td>
<td>$T_{8.0}$</td>
<td>$T_{8.1}$</td>
<td>$T_{8.2}$</td>
</tr>
</tbody>
</table>

Should the joining server fail to complete any of these tasks, and therefore fail the joining protocol, it cannot join the system. This is regarded as a failure, as some servers may have already dropped replicas upon being assigned a replica of the new server’s primary. The protocol for dealing with failures is described later.

The reason why new servers join behind the eldest server and not the youngest server is that older servers are likely to have larger tuple space portions than younger servers. A new server that exists in the system for a short time only would cause the $\rho$th replica of the elder’s large tuple space portion to be dropped and copied again over a short period of time. Furthermore, by joining behind the eldest server, the new server assigns replicas of its initially empty tuple space portion to the $\rho$ oldest servers, which is computationally inexpensive, and accepts responsibility of the replicas of the $\rho$ youngest servers. Since the tuple space portions of the $\rho$ youngest servers are likely to be smaller than the tuple space portions of the $\rho$ oldest servers, the amount of data copied across the LAN, in a system comprising more than $\rho$ servers, is less than would have been copied if the larger tuple space portions of the $\rho$ oldest servers had been copied.

In a system comprising less than, or exactly, $2\rho$ servers (i.e., $\nu \leq 2\rho$), and more than $\rho$ servers (i.e., $\nu > \rho$), the set of the $\rho+1$ oldest servers intersects the set of the $\rho+1$ youngest servers, as shown in table 4.3. The $2(\rho+1)-\nu$ youngest of the $\rho+1$ oldest servers are also the $2(\rho+1)-\nu$ oldest of the $\rho+1$ youngest servers. In the system described by the distribution matrix given in table 4.3, one server, server 3, is the youngest of the three ($\rho+1$) oldest servers, as well as the oldest of the three ($\rho+1$) youngest servers.
Table 4.3: Distribution Matrix of a system of 5 servers where $p=2$, i.e., $\nu \leq 2p$.

<table>
<thead>
<tr>
<th>Server</th>
<th>Primary</th>
<th>1st Order</th>
<th>2nd Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 1</td>
<td>$T_{(1,2)}$</td>
<td>$T_{(5,1)}$</td>
<td>$T_{(4,2)}$</td>
</tr>
<tr>
<td>Server 2</td>
<td>$T_{(2,3)}$</td>
<td>$T_{(1,1)}$</td>
<td>$T_{(1,2)}$</td>
</tr>
<tr>
<td>Server 3</td>
<td>$T_{(3,3)}$</td>
<td>$T_{(2,1)}$</td>
<td>$T_{(1,2)}$</td>
</tr>
<tr>
<td>Server 4</td>
<td>$T_{(4,3)}$</td>
<td>$T_{(3,1)}$</td>
<td>$T_{(2,2)}$</td>
</tr>
<tr>
<td>Server 5</td>
<td>$T_{(5,3)}$</td>
<td>$T_{(4,1)}$</td>
<td>$T_{(3,2)}$</td>
</tr>
</tbody>
</table>

In a system comprising less than, or exactly, $p$ servers (i.e., $\nu \leq p$), the $p$ oldest servers are also the $p$ youngest servers, as shown in table 4.4. The set of the $p$ oldest servers fully intersects the set of the $p$ youngest servers: i.e., the two sets are equivalent. In the example given in table 4.4, servers 1 and 2 are the $p$ oldest, as well as the $p$ youngest servers. Note that in such a system fault-tolerance cannot be guaranteed.

Table 4.4: Distribution Matrix of a system comprising less than, or exactly, $p$ servers.

<table>
<thead>
<tr>
<th>Server</th>
<th>Primary</th>
<th>1st Order</th>
<th>2nd Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 1</td>
<td>$T_{(1,2)}$</td>
<td>$T_{(2,1)}$</td>
<td>unassigned</td>
</tr>
<tr>
<td>Server 2</td>
<td>$T_{(2,2)}$</td>
<td>$T_{(1,1)}$</td>
<td>unassigned</td>
</tr>
</tbody>
</table>

Servers leave the system by effectively simulating server failures. The result is that no more than $p$ servers may leave the system at the same time. One benefit of this approach is that the set of protocols is simplified as only two protocols are required: these being the joining protocol and the failure toleration protocol (described below). However, rather than simply terminating operation and leaving the detection of the simulated server failure to the affected servers, the leaving server informs all servers that will be affected that it is leaving, and then terminates in a controlled manner. This information greatly aids detection of the simulated server failure, and allows for speedy toleration. The development of a leaving protocol that overcomes the limitation that no more than $p$ servers may leave simultaneously is left to further research.
Inter-Server Communication

Servers communicate through NetBIOS sessions. Each server, as shown in figure 4.7, has a single session with:

- every other server that maintains a replica of this server’s primary tuple space portion,
- every other server for which this server maintains a replica, and
- the elder.

Figure 4.7: The 16 sessions \((pv+(v-(2p+1)))\) connecting the servers in a system comprising 7 nodes with a resilience degree of 2.

However, between two given servers a single session only exists. Thus, when the elder is one of the servers that maintains a replica for a given server \(\chi\), a single session
only exists between server \( \chi \) and the elder. Similarly, recall that when the number of servers does not exceed \( 2p \) (i.e., \( v \leq 2p \)), a set of servers exists (see previous section) that:

- each maintain a replica for server \( \chi \), and
- have a replica maintained for them by server \( \chi \).

In that case, a single session only exists between server \( \chi \) and each server contained in the set. Figure 4.7 shows that server 2 is connected to server 1, the elder, via a single session only. Because server 2 maintains a replica for server 1, it already has a session to server 1. Therefore, it does not maintain another session to server 1 in order to have a session to the elder. Similarly, server 3 maintains a replica of order 2 for server 1 and, thus, has a session (indicated as a replica session in figure 4.7) to server 1. Consequently, it does not maintain another session to server 1 in order to have a session to the elder.

There is a need for sessions between any server \( \chi \) and the servers that maintain a replica for server \( \chi \), because a change in the state of the primary tuple space portion must be actively propagated to the replicas. Since each server maintains replicas of other servers, each server also maintains a session to the server for which it maintains replicas. The need for a session to the elder arises out of the need to search all tuple space portions, as will become apparent in the description of the tuple space operators given later.

NetBIOS limits the number of sessions that may exist concurrently to 254 sessions. Furthermore, the number of names that may exist concurrently are limited to 255 names (see section 3.2). However, OS/2's Communication Manager may be configured to lower these limits even further. Assuming that a LAN's NetBIOS protocol is available exclusively for an OS/2-C-Linda system, and that the NetBIOS protocol has been configured with the maximum number of names and sessions, the resilience degree and the number of nodes participating in that system are limited by the relation \( pv^+(v-(2p+1)) \leq 254 \).

**Detecting and Tolerating Server Failures**

This section first discusses how server failures are detected, and then how detected server failures are tolerated. Moreover, the discussion on the latter initially concentrates on a high-level description, and then on a low-level description.
Detecting Server Failures

Before any server failure may be tolerated it must be detected. The primary means of detection is NetBIOS: every command executed by NetBIOS yields a return code. This return code indicates either successful completion of the command, or provides a reason as to why the command could not be completed. Because server failures and NetBIOS session failures are indistinguishable, server failures are assumed when a NetBIOS session fails. This assumption, as proven later in the implementation discussion, does not weaken data integrity.

However, the NetBIOS implementation available to OS/2-C-Linda provides good failure semantics only when failures occur while NetBIOS is actively processing a command. Failures that occur while NetBIOS is idle (i.e., NetBIOS is not processing a command, or the command involves waiting for a remote NetBIOS to send some data) are detected once NetBIOS becomes active. Furthermore, some failures, such as physically unplugging the network cable, are detected solely by use of a timeout: NetBIOS fails the command because it was unable to complete it within the given amount of time. Consequently, should an indefinite timeout be specified, some failures may never be detected. For example, assume that processes $P_1$ and $P_2$ are running on machines $M_1$ and $M_2$ respectively. Furthermore, assume that process $P_2$ is waiting indefinitely on the completion of a NetBIOS command on a session to process $P_1$, and that the network cable is then physically unplugged from machine $M_1$. Experiments have shown that process $P_2$ will not detect the failure, unless it attempts to send some data to, or to receive some data from, process $P_1$, specifying a time period within which the operation must complete.

The secondary means of detecting server failures is a pinging mechanism (also referred to as beaconing). This mechanism facilitates the detection and toleration of failures while none of the tuple space portions change state. The pinging mechanism ensures that activity occurs on each session at least every $\Delta$ milliseconds. When a server realises that no message has been received from a session partner for more than $\Delta$ milliseconds, it sends a ping packet to that session partner. The sending of a ping packet gives NetBIOS the opportunity to report any failures that are immediately detectable. Upon receiving a ping packet, a server responds by sending a pong packet. The session partner is deemed to have failed after $\lambda$ attempts to elicit a pong packet fail.
Tolerating Server Failures

Given the topology of figure 4.5, up to $p=2$ concurrent failures may be tolerated. Furthermore, the tuple space is not fully replicated, but rather sufficiently enough to guarantee that up to $p=2$ concurrent failures may be tolerated.

Should any server fail, the server that maintains the replica of the highest order tolerates the failure by integrating the replica into its primary. For example, assuming server 4 fails, server 5 integrates $T_{(4,1)}$ into its primary $T_{(5,0)}$. Clearly, server 5 must ensure that all replicas of its primary are updated. Server 6 may simply integrate $T_{(4,p=2)}$ into $T_{(5,1)}$. However, because $p-1=1$, servers maintaining replicas of $T_{(5,0)}$ also maintain a replica of $T_{(4,0)}$, so server 5 needs to ensure only that the lowest order replica, $T_{(5,p=2)}$, is updated. Consequently, server 5 must copy $T_{(4,1)}$ to server 7, so that server 7 may integrate it into $T_{(5,p=2)}$. This observation is true for all possible values of $p$: i.e., for any failure a replica of the primary tuple portion assigned to the failed server must be copied:

- from the server maintaining the highest order replica, and
- to the server maintaining the lowest order replica of the server that is integrating the replica into its primary.

The OS/2-C-Linda Operators

This section describes the design of OS/2-C-Linda's implementation of the Linda operators out, in, rd, inp and rdp. The syntax of the OS/2-C-Linda implementations of these operators is based on the DOS-C-Linda implementation described in Bettermann (1992). Since this research does not consider Linda's eval operator, it is not implemented and not discussed here.

In order to eliminate possible conflicts with existing functions, the names of the OS/2-C-Linda operators (OS/2-C-Linda implementations of the Linda operators) are prefixed with "linda_". Thus, the OS/2-C-Linda operators are:

- linda_out
- linda_in
- linda_rd
- linda_inp
- linda_rdp
The `out` operator places the tuple defined in its parameter list into the tuple space; the
parameter lists of the `in`, `rd`, `inp` and `rdp` operators define tuple templates for which
matching tuples are to be found in the tuple space. A side-effect of calling any of the
`in`, `rd`, `inp` and `rdp` operators is that the values of all actual fields of a matching
tuple is assigned to the formal fields of the tuple template. In OS/2-C-Linda this
side-effect is implemented by assigning the values of all actual fields in the matching
tuple to the actual parameters corresponding to formal fields in the tuple template.

Because tuples and tuple templates may have any number of fields, the formal
parameter list of these operators must be of variable length. The host language, C,
provides a standard mechanism for dealing with variable length parameter lists. This
mechanism requires that the programmer, and not the programming language, handles
the assignment of actual parameters to formal parameters: i.e., the programmer must
communicate, to the function, the number and type of each of the actual parameters,
and receive the actual parameters within the function.

C's `printf()` function, defined in `stdio.h`, has a variable length parameter list
that adheres to an accepted technique of communicating this information. The first
parameter is a `format string` that defines the number and type of any subsequent
parameters. A format string consists of characters and `format specifications`: the latter
define the number and type of any actual parameters that follows the format string.

Thus, the OS/2-C-Linda operators require, as their first parameter, a format string
that defines the number of fields and their types of the tuple, or tuple template, defined
in its actual parameter list. This approach, as outlined by Bettermann (1992, p. 29), has
the benefit that the programmer provides information that OS/2-C-Linda cannot derive:
namely,

- how to interpret pointers (as an address, or the value referenced by the pointer),
- the structure of the data that pointers reference, and
- the polarity of tuple fields; i.e., whether a field is actual or formal.

Furthermore, the usage of a format string results in the OS/2-C-Linda operators being
aptly similar to C's `printf()` function. For a detailed description of the format
string's syntax see Bettermann (1992, pp. 30-42).
OS/2-C-Linda provides the Linda operators as C functions. The prototypes of these functions are given below:

```c
LINDA_RETURN_CODE linda_in( char *format, ... ) ;
LINDA_RETURN_CODE linda_out( char *format, ... ) ;
LINDA_RETURN_CODE linda_rd( char *format, ... ) ;
LINDA_RETURN_CODE linda_rdp( char *format, ... ) ;
LINDA_RETURN_CODE linda_inp( char *format, ... ) ;
```

As can be seen from the above, each Linda operator returns a LINDA_RETURN_CODE. This type has been defined as an enumerated data type, with LINDA_SUCCESS indicating successful completion of the Linda operation. All other possible values, such as LINDA_PREDICATE_FAILS, LINDA_OUT_OF_MEMORY, and LINDA_SERVER_DIED, indicate that the Linda operation did not complete successfully, and the cause for the failure.

**Tuple Signatures**

A tuple's tuple signature (hereinafter simply referred to as signature) is derived from the format string that describes the number and type of each field in the tuple or tuple template. For example, assume a tuple is inserted into the tuple space with the following OS/2-C-Linda function call:

```c
linda_out( "%s %d", "The answer is", 42 ) ;
```

The signature of the tuple that is inserted is derived from the format string ("%s %d") by removing all white-space and polarity indicators ('%' and '?'). Thus, the signature of the tuple inserted by the above function call is "sd".

Similarly, assume that the following C declaration exists that defines the number of days in each month of a non-leap year:

```c
```

The format string of the tuple inserted into the tuple space with the following OS/2-C-Linda function call is "s1u20d":

```c
linda_out( "%s %lu %20d", "Days", 12lu, days );
```
Signatures of tuple templates are derived in the same manner. The signature of the tuple template contained in the parameter list of the following OS/2-C-Linda function call is “s d”:

```
   linda_in( "%s ?d", "The answer is", &answer );
```

**Tuple Space Distribution**

The tuple space is partitioned into tuple space portions based on the notion of a tuple signature: each tuple space portion owns a set of signatures. Signatures describe the types of tuples that a tuple space portion holds; a given tuple space portion consists entirely of tuples with signatures that the tuple space portion owns.

A tuple space portion acquires ownership of a signature when a local Linda client performs a Linda operation using a tuple or tuple template with a signature that is not owned by any tuple space portion in the Linda system. Similarly, a tuple space portion relinquishes ownership of a given signature when:

- the last tuple with that signature is removed from it, or
- the last blocked request for a tuple with that signature completes.

Initially, Linda servers create tuple space portions that are empty, and thus own no signatures. When a Linda client inserts a tuple, the local Linda server determines who owns the signature of the new tuple. Upon determining that no tuple space portion owns the signature, it takes ownership of the new tuple’s signature and inserts the tuple into its tuple space portion. Similarly, upon determining that Linda server \( \chi \) owns the new tuple’s signature, the local Linda server hands the new tuple over to Linda server \( \chi \), who inserts it into its tuple space portion.

This method of partitioning is convenient because it is easily accomplished and, consequently, computationally inexpensive. Furthermore, tuples congregate in a tuple space portion of one of the set of the servers that produce or consume tuples with that signature.

As all possible signatures are equally likely to occur (although tuples with a large number of fields are rare), tuple space distribution based on signatures must be uniform. Similarly, because, from a Linda server’s perspective, all Linda clients are equally likely to cause a tuple space portion to acquire ownership of a signature, the distribution of signature ownership among Linda servers is also uniform. A more
exhaustive determination of this method’s statistical properties is left for further research.

4.3 Description of the Implementation

This section describes the implementation of OS/2-C-Linda. The implementation manifests itself as two executable components. These two components, identified in section 4.2 above, are the OS/2-C-Linda server (hereinafter referred to as server), and the Linda library (hereinafter referred to as library). The server takes the form of an executable file, whereas the library takes the form of a DLL. Each will be described in more detail below.

The architecture of the server is presented in figure 4.8. The server is placed into context via the inclusion of external entities in the figure: namely, Linda Servers and Linda Clients.

![Diagram of Server Architecture](image)

**Figure 4.8: Server Architecture.**

As can be seen in figure 4.8, the server is a multi-threaded process (see section 3.1), with the main thread, created by OS/2 when the process is started, being the ancestor of all threads, except itself. The taxonomy below defines which threads (if any) each thread creates. Indentation denotes a parent/child thread relationship, with the parent creating
the child thread upon commencement of execution. Note that the name thread creates a ping thread and a receiver thread for each session with another server.

- Main thread
  - Client Manager thread
    - Client Socket thread
  - Console Manager thread
    - Keyboard thread
  - Network Manager thread
    - Name thread
      - Ping thread
      - Receiver thread

The server is an application that services requests. Requests presented to a server fundamentally originate from a Linda client or a console operator using some server in the system. Furthermore, servers may need to co-operate with other servers in an effort to satisfy requests presented to them. Therefore, client or console operator requests submitted to a server may be redirected to some other server or result in requests being made of other servers. Consequently, each server has three interfaces through which requests may be made: namely,

- the client interface for client requests,
- the console interface that facilitates the expression of requests by console operators, and
- the network interface, via which
  - client or console operator requests, originally presented to some other server, are redirected to this server, and
  - requests may be made of this server by other servers that are servicing requests presented to them.

The architecture of the server mirrors this scenario: each interface is implemented by a manager thread, respectively called client manager, console manager and network manager. All manager threads are fathered by the main thread, whose only other duty is, eventually, to synchronise their termination. These manager threads utilise services
provided by OS/2, C and NetBIOS to interface with the sources that may present requests to them: effectively,

- the client manager uses OS/2’s named pipes to establish communication channels with clients and then to communicate with these clients,
- the console manager uses C’s `getch()` to obtain requests from the console operator, and
- the network manager uses NetBIOS to establish sessions with other servers and then to communicate with these other servers.

Well-designed applications for multi-threaded operating systems, such as OS/2, should utilise the CPU only to complete tasks. Thence, a thread that has not been assigned a task should not utilise the CPU. Rather, the thread should suspend execution, thus preventing itself from using the CPU, until it is assigned a task, typically through the receipt of an event.

As described earlier (see section 3.1), any number of event semaphores may be attached to a named pipe end. OS/2 posts these event semaphores when any activity occurs on the named pipe end to which the event semaphores are attached. Accordingly, a thread may be constructed such that it is suspended until some request arrives through a named pipe, causing activity on that named pipe.

Assume that the thread wishes to read requests from a number of named pipes. Each named pipe has an event semaphore attached to the end from which the thread wishes to read. Assume, also, that each event semaphore is attached to one named pipe end only, and that the event semaphores are added, with any semantics, to a muxwait semaphore.

The thread, then, may loop around the muxwait semaphore. Each time around the loop, the thread blocks while the muxwait semaphore is in a reset state. The muxwait semaphore becomes posted (thus, causing the thread to unblock) only when activity occurs on a named pipe end to which is attached one of the event semaphores contained in the muxwait semaphore. The unblocked thread, before continuing around the loop:

- resets the event semaphore,
- determines which named pipe the activity (which caused the event semaphore to become posted) has occurred on,
- reads the request from the named pipe decided upon previously, and
- services the request.
Of the services used by the manager threads only one (the named pipe service) caters for this event-driven design, and then only for communicating via *established* pipe connections. The process of establishing a named pipe connection between two threads is likely\(^1\) to block the thread that creates the named pipe. This thread remains in a blocked state until another thread opens the named pipe, thereby establishing a named pipe connection between the two threads. The remaining services used by the manager threads do not lend themselves to an event-driven design either, as particularized below:

- C's `getch()`, used by the console manager, blocks the calling thread until the operator presses a key on the keyboard, and
- NetBIOS, used by the network manager, may be used only in a blocking manner, as asserted in section 3.2.

Consequently, functionality is required that provides semantics, akin to those described above for named pipe communication, for the services that do not currently provide them. This functionality must:

- implement a buffering mechanism, such as a queue, through which requests are conveyed to the appropriate manager thread as soon as they occur at the source, and
- allow event semaphores to be *attached* to the buffering mechanism such that they are in a posted state when the buffering mechanism contains requests.

Thus, other lower-level threads, hereinafter referred to as *service threads*, are needed for the construction of the server as a well-designed multi-threaded application. This need exists because

- requests must be conveyed to the appropriate manager, via the buffering mechanism as soon as they occur, and
- the means of obtaining the requests from the source are of a blocking nature.

These service threads mask to the manager threads the blocking nature of C’s `getch()`, NetBIOS, and the establishment of OS/2’s named pipes by blocking on calls to these services. When a call completes, and the service thread thereby unblocks, any requests obtained are enqueued onto the buffering mechanism. The service thread ensures that,

\(^1\)The on-line documentation states that `DosConnectNPipe()` will place the named pipe into a *LISTENING* state, and block the calling thread until a client opens the pipe using `DosOpen()`. Since `DosOpen()`, according to the on-line documentation, returns `ERROR_PIPE_BUSY` when the pipe is not in a *LISTENING* state, `DosConnectNPipe()` must be called before `DosOpen()` can succeed. Because it is unlikely that the two threads call `DosConnectNPipe()` and `DosOpen()` simultaneously, the creator is likely to block. Furthermore, in a single-CPU machine the creator will block, as it is impossible for the two threads to call `DosConnectNPipe()` and `DosOpen()` simultaneously.
while the buffering mechanism contains requests, all event semaphores attached to it are in a posted state, so achieving the semantics.

The implementation of each thread, and the manner in which it communicates with other threads, is described next. Note that every thread creates, upon starting, a number of resources, such as event semaphores and mutex semaphores. Some resources are used by multiple threads for communication and synchronisation. Consequently, the description of these threads is not sequential, and threads are sometimes referred to before they are detailed.

**The Main Thread**

The main thread communicates with its child threads (the client manager, console manager and network manager) purely through event semaphores collected in muxwait semaphores, and only when the server is to stop. To that end, the main thread creates four event semaphores: one event semaphore for the main thread to post when all manager threads are to complete execution, and one event semaphore for each manager thread to post upon completing execution. For clarity, the former semaphore is referred to as the *termination* semaphore, and the latter three semaphores are collectively termed *trigger* semaphores. Therefore, these latter event semaphores are respectively called the *client manager trigger*, *console manager trigger*, and *network manager trigger event semaphores*. 

Essentially, the main thread is blocked, for most of its lifetime, on a muxwait semaphore with any semantics that contains the trigger semaphores. Thus, when any manager thread finishes, and so posts its trigger semaphore, the main thread unblocks. The unblocked main thread proceeds by posting the termination semaphore and blocking execution waiting on a muxwait semaphore with all semantics that contains the trigger semaphores. Only when all manager threads complete execution and, hence, all trigger semaphores are posted, does the main thread unblock. Upon unblocking, the main thread may safely terminate, thus ending the process that implements the server.

The architecture outlined above guarantees the clean termination of the server. This ability is necessary because a process, whose main thread terminates execution, does not terminate properly while any one of its threads is blocked on a call to a 16-bit function. Rather, the process is turned into a ‘zombie’ process which will never terminate and release its resources. This does not apply to threads blocked on 32-bit
functions, as these are terminated by OS/2 when the main thread terminates, thus releasing all resources.

Manager Threads

Each manager thread, upon starting, creates a muxwait semaphore with any semantics that contains at least one event semaphore: the main thread’s termination semaphore. Furthermore, the muxwait semaphore contains the event semaphores posted by any of the service threads that the manager thread spawns. Each manager thread loops around a muxwait semaphore that is posted only when

- a service thread indicates, by posting an event semaphore, that some work has arrived for processing and is obtainable from some buffering mechanism, or
- the main thread wishes to terminate.

In the former case, the manager thread retrieves the request from the buffering mechanism, handles the request, and resets the event semaphore (if required) before continuing around the loop. In the latter case, the manager thread closes all service threads and, before terminating, posts the trigger semaphore assigned to it by the main thread. The manager thread is able to determine from which buffering mechanism to retrieve the request by using information provided by the muxwait semaphore: muxwait semaphores report which semaphore causes their posting. Since there is a one-to-one mapping of event semaphores to buffering mechanisms, the buffering mechanism that holds a request to be retrieved may be determined from this information.

Thus, the server threads provide an interface to the manager threads that allow the manager threads to service request from multiple sources when they occur, or suspend execution when no requests exist that require processing.

The Client Manager

The client manager is responsible for accepting new clients, and for servicing their requests: the former task is accomplished with the aid of the client socket thread. As event semaphores may be attached to a named pipe end, the latter task does not require the use of a service thread. The client socket thread, described below, accepts new clients and communicates the new client’s details to the client manager via a single element queue, called the socket queue. This queue need not be any larger than one
element because one client may attach to the server at a time. Attached to this queue are two event semaphores: one to the head of the queue, the other to the tail. Accordingly, the semaphore attached to the head is called the head semaphore, and the semaphore attached to the tail is called the tail semaphore. Furthermore, the head semaphore is posted when the queue contains any elements, and the tail semaphore is posted when the queue contains no element.

It is the head semaphore that the client manager adds to its muxwait semaphore. Thus, when a client connects to the server, the socket thread accepts the connection, and inserts the new client's details into the queue. This insertion causes the head semaphore to become posted, which causes the client manager to unblock. The unblocked client manager retrieves this information from the queue and processes the new client. Part of the information transferred from the socket thread is an event semaphore that is attached to the named pipe that connects the server to the client. Thus, the client manager adds this event semaphore to the muxwait semaphore.

Any client may send a request to the server through the named pipe. The client server unblocks when the request arrives at the server end of the named pipe. This request is read from the named pipe by the client manager upon unblocking as a result of the event semaphore posting. Client requests may be serviced immediately by referring to the local tuple space portions, or by redirecting them to other Linda servers.

The client manager terminates the client socket thread by killing it using OS/2's DosKillThread() API: i.e., the client socket thread does not complete execution by itself. The routine that kills the client socket thread also discards the client socket thread's event semaphores and the named pipe to which no client has yet connected.
The Console Manager

The console thread is responsible for providing an interface to the operator. This interface allows the operator to perform operations on the Linda server: namely,

- shutting the Linda server down,
- showing current system configuration,
- printing the primary tuple space portion,
- listing Linda clients connected to the Linda server,
- killing Linda clients connected to the Linda server,
- saving local tuple space portions, and
- loading local tuple space portions.

As identified earlier, C’s `getch()` function blocks the calling thread until the operator presses a key on the keyboard. Therefore, a service thread called the keyboard thread is used to obtain commands from the operator. This service thread maintains a queue of characters read off the keyboard, to which an event semaphore (the `keyboard semaphore`) is attached. The event semaphore is posted when the queue contains characters, and is in a reset state when the queue is empty.

Subsequently, the `muxwait` semaphore around which the console thread loops contains the main thread’s termination semaphore and the keyboard semaphore. Thus, when the operator presses any key on the keyboard, the character is added to the queue, which causes the keyboard semaphore to become posted, which causes the console thread to unblock.

The console thread reads characters off the queue and accumulates them into commands. These commands are passed to a command processor that implements the operations described above. When the operator enters the command to shut the server down or when the completion semaphore becomes posted, the console thread posts the trigger semaphore assigned to it and terminates.

The Network Manager

The network manager is responsible for accepting connections from other servers and for receiving and servicing requests made or redirected by other servers. However,
the communications facility used for inter-server communication, NetBIOS, suffers a number of restrictions: namely,

- the NetBIOS operations do not cater for an event-driven design as NetBIOS may be used in a blocking fashion only,
- failures may be detected only when a NetBIOS operation is executed, i.e., NetBIOS reports failures only to the caller of any following NetBIOS operations, and
- NetBIOS messages are limited in size to $2^{16}-1$ bytes.

Consequently, a communications protocol has been implemented over NetBIOS that mirrors the operations provided by NetBIOS, but in a manner that overcomes the limitations enumerated above. However, as OS/2-C-Linda does not require the full set of NetBIOS operations, only a subset of the operations provided by NetBIOS is provided by the protocol as a set of APIs.

Essentially, the protocol addresses the first of the restrictions named above by employing service threads — just like the keyboard and client socket threads that mask the blocking nature of the underlying services used by the client manager and the console manager. These service threads are:

- name threads to accept requests for connections to other servers, and
- receiver threads to receive requests through these connections.

These service threads use queues to relay (to the network manager) any requests that are presented to them. Specifically, name threads use queues called name queues to relay to the network manager requests for new connections. Similarly, the receiver threads use queues called session queues to relay to the network manager requests received through these connections.

Threads are also used to address the second restriction: for every session a ping thread is created that executes NetBIOS operations when neither session partner is using NetBIOS, e.g., during times of system inactivity. Any failures occurring during such times may be reported by NetBIOS to the ping thread, who may then report them to the network manager.

The third restriction is overcome by splitting messages into message fragments no larger than $2^{16}-1$ bytes. Message fragments are reassembled by the receiver thread, who enqueues the message only upon receipt of all message fragments.
Recall that NetBIOS sessions are point-to-point connections between NetBIOS names. For every name that the network manager adds to the local name table, a name thread is created. This name thread is responsible for allowing other servers to connect to the name, thus establishing a session to the server.

Every session is abstracted by the network manager into a data structure called a session handle that encapsulates information needed to use and manage the session. When another server calls a NetBIOS name (thus establishing a session) the name thread allocates a session handle, and communicates it to the network manager via a name queue. Attached to this queue is an event semaphore that is posted when the queue contains session handles, and is in a reset state when the queue is empty. As the queue is used by multiple threads (the name thread and the network manager), the functions that change the state of the queue use, in a monitor-like fashion, a mutex semaphore to ensure that only one thread at a time changes the state of the queue. Each function sequentially

- obtains ownership over the mutex semaphore (blocking the calling thread until obtaining ownership),
- performs some, possibly state-changing, operation on the queue (such as adding a session handle to the queue), and
- relinquishes ownership over the mutex semaphore.

The protocol employs NetBIOS messages as packets, each packet having its own packet type and each containing its own set of attributes to define its state, which is determined from the packet’s type — every packet contains its state-defining type as an attribute. The packet types and the state-defining attributes are described in detail in a subsequent section.

A salient feature of the protocol is that each message, upon sending, is fragmented into a header packet followed by multiple fragment packets. The header packet contains attributes that define:

- the length of the entire message,
- the length of the fragment contained in the header packet, and
- the first (and possibly only) message fragment.

Since NetBIOS guarantees that messages are received in the same order that they are sent, packets need not be numbered so that they can be reassembled correctly. Furthermore, NetBIOS guarantees that messages are not lost, and that they are delivered correctly. Therefore, the protocol need not concern itself with requesting
already sent but lost or corrupted messages. Should some condition prevent NetBIOS from delivering a message correctly, NetBIOS drops the session.

Given that the packet type attribute occupies one byte, the message length attribute occupies four bytes, the fragment length attribute occupies two bytes, and that the maximum size of NetBIOS messages is $2^{16} - 1$ bytes, the largest message fragment that the header packet may accommodate is $2^{16} - 8$ bytes. Similarly, fragment packets contain a packet type attribute (one byte), a fragment length attribute (two bytes), and a message fragment. Given the maximum NetBIOS message size of $2^{16} - 1$ bytes, the largest message fragment that a fragment packet may accommodate is $2^{16} - 4$ bytes. Accordingly, any message larger than $2^{16} - 8$ bytes must be fragmented into a header packet and fragment packets: the first $2^{16} - 8$ bytes are placed into the header packet, the remainder of the message is sent via fragment packets, $2^{16} - 4$ bytes at a time. Messages not exceeding $2^{16} - 8$ bytes fit completely into the header packet and so no fragment packets are required.

Each receiver thread communicates with the network manager via a queue called the session queue. When the session partner sends a packet through the session, the receiver thread receives the packet, encapsulates it into a message object, and adds it to the end of the session queue. Message objects are added to the session queue in the same manner that session handles are added to the end of the name queue, i.e., the session queue is protected by a mutex semaphore to guard against concurrent access to it by multiple threads. Attached to each session queue is an event semaphore called the session semaphore. The session semaphore is posted when the session queue contains message objects, and is in a reset state when the queue is empty.

However, not every packet received by the receiver thread results in the formulation of a message object. Messages that are fragmented into multiple packets (header packets and fragment packets) are reassembled into the original message, and a single message object only is added to the session queue. Furthermore, ping and pong packets are consumed by the receiver thread and don't result in the enqueueing of
message objects onto the session queue. Lastly, events other than the receipt of messages may occur on each session: namely,

- a request is received from the session partner to close the session,
- notification that the session has been forcibly closed by the session partner is received,
- NetBIOS drops the session due to some error, and thus returns an error code to the receiver thread or ping thread, and
- the ping thread detects that the session or the session partner has failed.

Hence, message objects describe the nature of the event that has occurred on the session. The receipt of message fragments result in the enqueueing of message objects that contain the de-fragmented messages. Requests by the session partner to close the session result in the enqueueing of message objects informing the network manager of such requests. Notifications that the session partner has forcibly closed the session, or detection of session failure by the receiver thread or ping thread, result in the receiver thread enqueueing message objects onto the session queue notifying the network manager of this event, and then terminating the ping thread and finally itself.

The network manager uses these three service threads in the same manner that the client and console manager use their service threads: it adds (to its muxwait semaphore) the event semaphores attached to the name and session queues. When a session handle arrives through a name queue, the network manager unblocks and adds (to its muxwait semaphore) the event semaphore attached to the session queue of the new session. Similarly, when a packet arrives through a session, the network manager handles the event, which is most likely to be a request from another server.

**Service Threads**

Each service thread is created and terminated by the manager thread that utilises its services. The fashion in which each type of service thread is terminated varies for each thread, as described below.

**Client Socket**

The client socket thread is responsible for accepting connections from new clients. To that end, the client socket thread creates a named pipe and blocks execution waiting for a client to connect to the named pipe. When a client connects to the server by
opening the pipe, the client socket thread is unblocked. Upon becoming unblocked, the
client socket thread creates an event semaphore and attaches it to the named pipe. An
initialisation protocol (described in the ensuing paragraph) is executed, then the client
socket creates another instance of the named pipe, and continues around the loop.

The initialisation protocol requires that information such as process ID and process
name is written by the client to the named pipe. Upon receiving this information, the
client socket thread writes an integer to the named pipe: 1 when the client connection is
successful, or 0 when the connection fails.

Because the OS/2 API that must be called to connect a client to a named pipe blocks
execution of the thread until another thread has opened the named pipe, the client
socket thread cannot be terminated by posting an event semaphore. Instead, the client
socket thread must be terminated using OS/2's DosKillThread() API. Any
resources obtained by the socket thread must be relinquished, although not necessarily
by the thread that obtained them. Furthermore, OS/2 will, on the server's behalf,
relinquish all resources when the server terminates.

For each client, the client socket thread creates a data structure called the \textit{client
record} that contains pertinent information, such as the handle of the named pipe and
the event semaphore attached to the named pipe. This client record is communicated to
the client manager via a single element queue (already described above). The two
event semaphores attached to this queue are used both as an indicator of the queue's
state, as well as a means of guarding access to it.

In order for the client thread to insert client records into the queue, it must first wait
(possibly blocking) on the tail semaphore which, as elucidated previously, is posted
only when the queue contains no elements. Upon unblocking, the thread resets the tail
semaphore, inserts the client record into the queue, and posts the head semaphore. The
client manager may remove the client record from the queue by performing a
complement of the aforementioned scheme. This entails waiting (possibly blocking)
on the head semaphore, resetting the head semaphore, removing the client record from
the queue, and posting the tail semaphore.

\textit{Keyboard Thread}

Upon starting, the keyboard monitoring thread creates a queue, two event
semaphores (called \textit{queue} and \textit{keyboard}) and a mutex semaphore. Client threads may
open the keyboard monitor, thereby receiving the queue semaphore and posting the keyboard semaphore. Also, client threads may close the keyboard monitor (the last thread to close the keyboard monitor resets the keyboard semaphore).

When the keyboard event semaphore is posted, the keyboard thread reads characters from the keyboard using C’s `getch()` function. This function causes the keyboard thread to suspend execution until a key on the keyboard is pressed. When a key is pressed, the character read is added to the queue by firstly obtaining the mutex semaphore that guards access to the queue, secondly adding the character to the end of the queue, and finally releasing the mutex semaphore. The queue event semaphore remains in a posted state while the queue contains characters, and remains in a reset state when the queue is empty.

Any thread that wishes to read characters from the keyboard may use the keyboard monitor. When the queue semaphore is in a posted state the queue contains characters, read from the keyboard, that may be removed from the queue. This removal is effected by firstly obtaining the mutex semaphore that guards access to the queue, secondly removing the character from the head of the queue, and finally releasing the mutex semaphore.

**Name Thread**

For most of its lifetime, the name thread is blocked on a call to NetBIOS, indicating to the latter that it is willing to accept connections to the name associated with the name thread. Upon receiving a call, thus establishing a session, the name thread creates

- a session handle,
- two threads that manage the session for this server, namely a receiver thread and a ping thread,
- a packet queue, and
- an event semaphore that is attached to the packet queue.

This session handle, containing the information needed to use and manage the session, is placed onto, and removed from, the name queue in the manner described above.

The name thread may not be terminated by OS/2’s `DosKillThread()` API as it is likely to be blocked on a call to NetBIOS. As discussed previously, NetBIOS resides within a 16-bit DLL (see section 3.2), and OS/2’s `DosKillThread()` API cannot
terminate a thread that is blocked on a call to a 16-bit function. Consequently, the
name thread must be flagged for termination, and unblocked by calling the name
(thereby establishing a NetBIOS session) for which the name thread is broadcasting
willingness to accept connections to. Upon unblocking, the name thread detects that it
is flagged for termination, closes the newly created session and terminates.

The Ping Thread

Failures may be detected by inspecting the return code from every NetBIOS call. However, relying solely on return codes is insufficient for the immediate detection of failures. Rather, NetBIOS can report the failure only when a call to NetBIOS is made. Consequently, when no activity occurs or no requests are made of NetBIOS the failure cannot be detected. But it is during times of inactivity when it is most appropriate to detect and handle failures as the system's capacity to process tasks is not infringed upon by the process of handling failures.

The ping thread ensures that, within some time $\Delta$ (where $\Delta$ is a system parameter), at least one attempt is made to send a NetBIOS message to the session partner. Should the session fail, the ping thread detects and reports this failure when NetBIOS fails to deliver the message successfully and therefore drops the session. Furthermore, the ping thread ensures that at least one message is received from the session partner within time $\Delta^*\lambda$, where $\lambda$ is also a system parameter. This assertion allows server failures to be detected: when no NetBIOS message is received from the remote server within time $\Delta^*\lambda$ the remote server must have failed as the remote server's ping thread should ensure that at least one NetBIOS message is sent within time $\Delta$.

Furthermore, the NetBIOS message that the local ping thread sends is a ping packet. When a receiver thread notifies its ping thread that a ping packet has been received, the ping thread responds by sending a pong packet to the session partner. Since over time $\Delta^*\lambda$, $\lambda$ ping packets are sent successfully that should result in the receipt of $\lambda$ pong packets, failure to receive a single packet of any type may be taken as an indication that the remote server has failed. Note that the receipt of any message (ping, header, fragment, etc.) indicates that the remote server has not failed.

The ping thread ensures that one message at least is received during time $\Delta^*\lambda$ by using a counter called the retry counter. Whenever the ping thread receives notification that a packet has been received, it resets the retry counter to zero. When determining that no packet has been received within time $\Delta$, the ping thread inspects
the retry counter. When the retry counter has a value lower than $\lambda$, the ping thread sends a ping packet to the session partner, and increments the retry counter. The session partner's ping thread should, upon receiving notification from its receiver thread, respond by sending a pong packet. Should the ping thread, when determining that no packet has been received within time $\Delta$, find that the retry counter has reached the value of $\lambda$, then the ping thread knows that the session partner has failed.

The Receiver Thread

Each receiver thread is responsible for receiving packets (i.e., NetBIOS messages) from its session, and for controlling the ping thread that ensures that any failures are promptly detected. Upon receiving a packet, the receiver thread handles it according to its packet type, possibly accumulating packets into messages that are eventually enqueued onto the session queue.

The complete set of packet types understood by the receiver thread is enumerated in table 4.5 below. As the set of attributes that packets contain depends on the packet type, the description of each packet type in the table below includes a listing of the attributes for given types.
Table 4.5: Packet Types and the Attributes they contain.

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| Header       | The first (and possibly only) fragment of a message. Header packets contain attributes that define:  
• the length of the message for which this packet is the header packet,  
• the length of the message fragment (the first message fragment) contained in the header, and  
• the first message fragment. |
| Fragment     | A fragment of a message, possibly the last. Packets of this type contain attributes that define:  
• the length of the message fragment contained in the fragment, and  
• a message fragment. |
| Acknowledgment | Return a return code indicating success or cause of failure. Each packet of this type contains only one other attribute besides the packet type attribute: the return code. |
| Ping         | Used to test the validity of the session and to elicit activity from the session partner in the form of a pong packet. Ping packets contain no attributes other than the packet type attribute. |
| Pong         | Used to acknowledge receipt of a ping packet, test the validity of the session, and embodies the activity requested by the session partner. Pong packets contain no attributes other than the packet type attribute. |
| Close        | Request session closure by the session partner. Each packet of this type contains only one other attribute besides the packet type attribute: the return code that is to be returned to the network manager. |
| Abort        | Used to inform session partner of impending session closure. Abort packets, just like close packets, contain only one other attribute besides the packet type attribute: the return code that is to be returned to the network manager. |
| Datagram     | May be used for exchanging small messages directly as NetBIOS messages. Packets of this type contain attributes that define the length of the message (which must be smaller than $2^{16}-1$ bytes) and the actual message. |

The following is a description of how the receiver thread responds to receiving packets of each type (except the previously described ping and pong packets, see above). For clarity, when considering an entity in a situation where multiple such entities exist, the specific entity is referred to as the local entity, while any other entity is referred to as a remote entity. However, when referring to two entities in relation to a third entity, local qualifies the entity closer to the third entity, whereas remote qualifies the further one.

Header packets, fragment packets, and acknowledgement packets are used in the exchange of messages. Upon receiving a header packet, the receiver thread allocates the memory necessary to store the entire message, and inserts the first fragment into it. Having successfully allocated the required memory, the receiver thread informs the sender that it is ready to receive the rest of the message by sending an acknowledgement packet. The sender will, upon receiving the acknowledgement packet,
packet from its local receiver thread, send the remainder of the message using fragment packets, which the receiver thread will add to the memory allocated earlier. Only when the entire message has been received does the receiver thread add a message object to the session queue.

Upon receiving a close packet, the receiver thread informs the Network Manager by enqueueing an appropriate message object onto the session queue, and initiates the termination of the session. Session termination requires that the receiver thread closes the NetBIOS session and initiates the termination of the ping thread, before terminating itself. The receipt of an abort packet is handled in an almost identical manner, except that (not unexpectedly) the network manager is informed that the session was aborted rather than closed.

Datagram packets may be used to exchange small messages. Upon receiving a datagram, the receiver thread allocates the memory necessary to store the datagram, inserts the datagram contents into it, and adds a message object containing the datagram onto the session queue. Similar to the receipt of a header packet, the receipt of a datagram packet causes the receiver thread to send an acknowledgement packet to the sender, informing the sender that the datagram has been received successfully.

**Inter-Server Communication**

The focus of this section is to describe how OS/2-C-Linda servers cooperate as an OS/2-C-Linda system according to the design outlined earlier (see section 4.2). Such a description must, necessarily, define how OS/2-C-Linda systems are formed, and how servers both join and leave existing server cooperatives. Furthermore, as the primary theme of this thesis is fault-tolerance, a description of precisely how server failures are tolerated is essential.

**Framework**

Inter-server communication occurs strictly via sessions using the communications protocol implemented by the network manager and outlined earlier. However, the distribution and replication of each server's pivotal state, the tuple space, is paramount to the ability to tolerate server failures. Consequently, two areas of inter-server
OS/2-C-Linda

communication must be described: communication required to establish and maintain the ring structure of the servers in the system, and communication required to:

- distribute and replicate the tuple space,
- perform state changing operations on the tuple space, and
- tolerate server failures.

The servers use a protocol called the ring layer, implemented using the communications protocol described above, for inter-server communication. This ring layer provides two primary functions: namely,

- the provision of remote procedure calls (RPCs), and
- the propagation of system maps.

Remote Procedure Calls

Each server provides a number of procedures, called remote procedures, that may be called remotely by any server in the system. Like ordinary procedures (i.e., those known from the imperative programming paradigm) each remote procedure may receive multiple parameters, and return multiple parameters. Furthermore, all procedures of this RPC implementation send to the caller a return code, indicating either successful completion of the remote procedure, or the cause of its failure.

Upon de-queueing a message from one of its session queues, the network manager interprets it as one of the following:

- a remote call to a procedure,
- a remote response to a procedure call, or
- a failure notification.
Every message that embodies a remote procedure call contains a header, a selector, and an actual parameter list. Similarly, the message that embodies the return from the remote procedure contains a header, the return code and any parameters that are passed back to the caller. The header defines:

- the destination of the remote procedure call or response, i.e., the server (to which the calling server may not necessarily be connected directly) that is to receive the call,
- the source of the remote procedure call or response, i.e., the server that is to receive the return code, as well as any returned parameters,
- the request ID that uniquely identifies each RPC, and which may be used to match a response to a call,
- the system map of the caller, and
- the priority of the remote procedure call, i.e., whether the RPC is being used to tolerate a failure.

An RPC message's remaining contents (i.e., the selector and the actual parameter list) respectively define the procedure that is to be called, and the actual parameter list that is to be passed to the procedure. In order to avoid deadlocks, every procedure call to a local procedure results in the creation of a thread that executes the called procedure and returns the results produced by the procedure to the caller. For simplicity, each parameter passed to a remote procedure must be provided as a pointer to a block of memory, and the size of that block of memory. This requires the caller and the remote procedure to serialise and de-serialise any parameters themselves. No type checking of parameters is performed by this simple RPC implementation.

As the elder is connected to all servers in the system, a server that needs to communicate with another server (with whom it has no direct session) may do so via the elder. Thus, the message embodying the remote procedure call is sent to the elder who routes the message to the target server. Similarly, the elder routes the response to the remote procedure call back to the caller.

**Blocking and Non-Blocking Calls**

A thread that wishes to call a remote procedure serialises the parameters that it wishes to pass to the remote procedure into memory, and calls the remote procedure. This call may be made in a blocking or a non-blocking manner.
When calling a remote procedure in a blocking fashion, the calling thread blocks on an event semaphore until the network manager de-queues a response to the remote procedure call from one of its session queues. Upon de-queueing such a response, the network manager places the results (the return code and any returned parameters) into a location known to the calling thread, and unblocks the calling thread by posting the event semaphore that the calling thread is blocked on.

However, a calling thread may not wish to receive any results from a remote procedure. Furthermore, a thread may need to engage in some other processing while the remote procedure executes. Therefore, a thread may provide a function that is to be executed by the network manager upon receiving a response from the called procedure. By providing a null address, the calling thread indicates that it is not interested in any results from the remote procedure, and the network manager does not act upon receiving any results.

Alternatively, the thread may provide an event semaphore that is to be posted by the network manager upon receiving a response from the called procedure. This event semaphore may be different to the event semaphore that the calling thread blocks on when calling the remote procedure in a blocking fashion.

**Multicast RPCs**

A call to a remote procedure may be directed at a single server only, or at a set of servers. The latter form is a super-set of the former form in that the call is sent to each server in a set. However, when remotely calling a procedure on a set of servers, the calling server handles the responses differently than it would when remotely calling a procedure on a single server.

This difference arises from the fact that multiple responses will be received: one from each server in the set. There are two ways in which multiple responses may be handled:

- wait for the first *successful* response (i.e., the first response with a return code of LINDA_SUCCESS) from any server in the set, and pass it to the caller as already described above for calling a procedure remotely on a single server, or

- collect a response from every server in the set, and pass the set of responses to the caller when every server in the set has responded.
Therefore, the caller must specify which of the two semantics are to be employed when receiving responses to the RPC.

Just like an RPC directed at a single server, an RPC directed at multiple servers that cannot be completed locally (i.e., the requesting server is not connected to all destination servers) is forwarded to the elder, who is connected to every server in the system. The elder also handles the responses to the RPC, returning the responses back to the calling server, where the network manager passes the results back to the calling thread in the manner already described above.

**Mapping of a Response to a Call**

When calling a remote procedure, the network manager allocates a unique request ID, and places the request ID into the header of the remote procedure call before sending the request to the remote server. The network manager then stores the request ID and the function address (or event semaphore) in a table. Any response to the remote procedure call will contain the request ID in the header of the response. Upon receiving the response to the call, the network manager retrieves the information previously placed into the table, deposits the results into a known location and calls the function (or posts the event semaphore).

The network manager must respond to the receipt of remote procedure calls to local procedures in an order that may not necessarily reflect the order in which they arrive. For example, given multiple unhandled messages (which may have arrived through the same session) the network manager must handle failure notifications before handling remote procedure calls. Moreover, to tolerate failures (the required response to receiving a failure notification) the network manager may need to call procedures residing on others servers. Consequently, the network manager must:

- handle remote procedure calls and responses that are necessary to tolerate failures before handling failure notifications, and
- handle failure notifications before handling remote procedure calls and responses that are not necessary for the toleration of failures.

An elegant but simple solution is to use a prioritised queue; each network manager maintains exactly one such queue (called the network queue). Similar to session queues, the network queue contains an event semaphore that is posted when the network queue contains events to be processed, and that is in a reset state when it
OS/2-C-Linda

contains no events for processing. The priorities of events added to the network queue are as follows:

- remote procedure calls and responses that are necessary to tolerate failures are of the highest priority,
- failure notifications are of a lower priority, and
- remote procedure calls and responses that are not necessary to tolerate failures are of the lowest priority.

The prioritised nature of the network queue ensures that the network manager always handles failure notifications before remote procedure calls and responses that are not necessary to tolerate failures. However, as the network manager may not, while tolerating a failure, respond to failure notifications or remote procedure calls and responses that are not necessary to tolerate failures, an event semaphore (called the *toleration semaphore*) is used to signify that a failure is currently being tolerated. This toleration semaphore is in a reset state while a failure is being tolerated; at all other times it is in a posted state.

Using the termination semaphore the network manager may respond to events on the network queue that represent remote procedure calls and responses that are necessary to tolerate failures, but ignore events on the network queue that represent failure notifications and remote procedure calls and responses that are not necessary to tolerate failures. While the toleration semaphore is posted, the network queue semaphore is posted only when the network queue contains remote procedure calls and responses that are necessary to tolerate failures. However, while the toleration semaphore is reset the network manager is posted when the network queue contains any event.

Rather than making use of another thread, it is the network manager that de-queues messages off the session queues and, for each one adds appropriately prioritised events onto the network queue. Although the use of a thread is conceptually cleaner as the network manager then needs to respond to events only (leaving the lower level messages to another thread), assigning this responsibility to the network manager results in a simple preferable solution.

**System Maps**

Every remote procedure call carries with it the system map of the caller. Similarly, every remote procedure response carries with it the system map of the callee. Since new servers are added to every server’s system map by the elder (see 4.2), when a
server $\alpha$ receives a remote procedure call or response from server $\beta$, and $\beta$’s system map does not contain the SID of server $\gamma$ whose SID is contained in $\alpha$’s system map, some assumptions may be formed. Either $\alpha$ may assume that $\gamma$ has failed and that $\beta$ has detected that failure, or that $\gamma$ is a new server and that $\alpha$’s system map contains the SID whereas $\beta$’s system map does not. In the latter case, $\beta$ may add $\gamma$’s SID to its system map. However, in the former case, $\alpha$ may remove $\gamma$ from its system map and initiate the failuretoleration protocol if server $\alpha$ maintains a replica for server $\gamma$, or server $\gamma$ maintains a replica for server $\alpha$.

Server failures detected by inspecting the system maps of other servers result in the creation of a failure notification for the failed server. As a result, two events are enqueued onto the network queue: the created failure notification, and the received procedure call or response. In the unlikely case that a remote server has incorrectly deemed another server as failed (this is detected when a received system map does not contain a server with whom the local server may still communicate), the deemed server failure must be honoured as some servers may already have taken action.

Ideally, the SIDs of new servers should be propagated by elder to the servers using a broadcast. However, as NetBIOS does not provide a reliable broadcast, a set of point-to-point message exchanges, i.e., a unicast, must be used as shown in figure 4.11. Yet, due to the non-atomic nature of the multicast $\gamma$ may receive message $m_3$ from $\beta$ before receiving message $m_2$ from $\alpha$ because $\alpha$ may send message $m_2$ to $\gamma$ after $\beta$ has, in response to receiving $m_1$ from $\alpha$, sent message $m_3$.

![Image](https://example.com/image.png)

**Figure 4.11:** NetBIOS delivers messages in order.

Therefore, each system map is time-stamped by the elder using a wrapping logical clock. A logical clock may be used to represent the temporal ordering of a set of events. Unlike ordinary clocks, logical clocks are not absolute: any given logical clock
only defines the temporal ordering of events using that same logical clock. Consequently, logical clocks typically increment at a non-constant real-time frequency, i.e., the real time between two events $e_n$ and $e_m$ is completely arbitrary.

The implementation of a logical clock is efficiently achieved using an $n$-bit unsigned number. Furthermore, assuming that no two events (using the same logical clock) occur more than $2^{n-1}-1$ logical clock ticks apart, the finite nature of $n$-bit unsigned numbers may be overcome by allowing logical clock values to wrap. Given a logical clock value domain of $0..2^n-2$ (an odd domain), wrapping determines that the successor of $2^n-2$ is 0 and the predecessor of 0 is $2^{n-2}$. As logical clock values may wrap, they may not reflect their temporal order. However, by exploiting the assumption stated above, their temporal order may be derived.

Table 4.6 presents the two possible distances between any two points of a wrapping domain of size $n=3$. There exist two distances between any two points because the domain wraps: if the calculated distance is negative, $2^n-1$ is added to it. Any distance exceeding $2^{n-1}-1$ is clearly marked by a darkened background.

Table 4.6: All distances between $t_a$ and $t_p$ in a wrapping domain of size $n=3$.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Since $\text{MAX}(t_a, t_p) - \text{MIN}(t_a, t_p)$ evaluates to a distance between $t_a$ and $t_p$, when:

$\text{MAX}(t_a, t_p) - \text{MIN}(t_a, t_p) \leq 2^{n-2}-1$

then

$t_a > t_p$. 
However, when:

$$\text{MAX}( t_a, t_b ) - \text{MIN}( t_a, t_b ) > 2^{n-2} - 1$$

then

$$t_a < t_b.$$ 

Since the domain $0..2^n-2$ contains an odd $(2^n-1)$ number of values, $\text{MAX}( t_a, t_b ) - \text{MIN}( t_a, t_b )$ can never be equal to $\text{MIN}( t_a, t_b ) - \text{MAX}( t_a, t_b )$. Therefore, there exists no ambiguity as to whether $t_a > t_b$ or $t_a < t_b$, despite the fact that either value may have wrapped multiple times.

Presented below as table 4.7 is a summary of the relationships that exist in a wrapping domain of size $n=3$. For means of comparison, the cells that were marked with a darkened background in table 4.6 are also marked in the same manner in table 4.7. Observe that the relationship in those cells is $<$. 

**Table 4.7: Relationships between $t_a$ and $t_b$ in a wrapping domain of size $n=3$.**

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>=</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>1</td>
<td>&gt;</td>
<td>=</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&gt;</td>
<td>&gt;</td>
<td>=</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>3</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>=</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>4</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>=</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>5</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>=</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>6</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>=</td>
<td>=</td>
</tr>
</tbody>
</table>

**Discourse**

Because the domain wraps, there exist two distances between any two points: one distance that traverses 0, and one distance that does not. By restricting the set of possible domains to domains of odd sizes, the set of distances between any two points in that domain must contain a distance that is shorter than the other.

The assumption that no two points are further apart than $2^{n-1} - 1$ allows comparison between values that may have wrapped a different number of times: when the shortest distance between the two points is $\leq 2^{n-1} - 1$ then both points must have wrapped the same number of times. Consequently, the smaller of the two points is temporally
smaller. However, when the shortest distance between the two points is $> 2^{n-1} - 1$ then two points must have wrapped a different number of times; therefore, the larger of the two points is temporally smaller.

**Application of Logical Clocks to System Map Management**

Due to the absence of a reliable broadcast facility, each system map is identified with a logical clock value. When comparing two system maps, their contents and their logical clock values are used in the manner defined in table 4.8. In the context of table 4.8, $\alpha$ refers to the local server, and $\beta$ refers to the remote server, i.e., $\alpha$ is performing the comparison of a system map received from $\beta$.

**Table 4.8: Use of logical clock values when comparing system maps.**

<table>
<thead>
<tr>
<th>Comparison of Map Contents</th>
<th>contents match</th>
<th>$\alpha$ is missing entries</th>
<th>$\beta$ is missing entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_\alpha &lt; t_\beta$</td>
<td>$\alpha$ gets $\beta$'s logical clock value</td>
<td>add missing entries to $\alpha$, and $\alpha$ gets $\beta$'s logical clock value</td>
<td>$\alpha$ has entries that $\beta$ doesn't because $\alpha$ hasn't detected failures; $\alpha$ detects the failures now</td>
</tr>
<tr>
<td>$t_\alpha = t_\beta$</td>
<td>$\alpha$ and $\beta$ are equal: no action necessary</td>
<td>owner of $\beta$ has not detected failures, but will do so now</td>
<td>$\alpha$ has entries that $\beta$ doesn't because $\alpha$ hasn't detected failures; $\alpha$ detects the failures now</td>
</tr>
<tr>
<td>$t_\alpha &gt; t_\beta$</td>
<td>$\beta$ gets $\alpha$'s logical clock value</td>
<td>owner of $\beta$ has not detected failures, but will do so now</td>
<td>$\alpha$ knows of an entry that $\beta$ doesn't know of yet</td>
</tr>
</tbody>
</table>

**Toleration of Failures**

The network manager initiates the failure toleration protocol upon receiving a failure notification from the network queue. A failure notification, itself, is added to the network queue by the network manager itself in response to:

- receiving notification that a server has failed, or
- detecting a failure while inspecting the system map of another server.

Note that failure to send a message, e.g., failure to send a request to call a remote procedure, will cause the underlying protocol to fail the session and enqueue an appropriate message onto the appropriate session queue. The network manager will add a failure notification event to the network queue upon receiving notification from the session queue.
Failures are tolerated immediately upon being detected, but only when there are no other failures currently being tolerated. Any failures detected during the toleration of a failure are scheduled for toleration; once a failure has been tolerated, the next one is tolerated. Thus, failures are tolerated sequentially in the order of detection.

An example application of the described scheme is presented next. Given servers $S_1$, $S_2$ and $S_4$, of logical SIDs 2, 1 and 3 respectively (as shown in figure 4.9), assume that server $S_4$ fails and that $S_1$ detects and integrates $T_{(4,2)}$ into $T_{(2,1)}$, but that $S_2$ doesn't detect the failure (and therefore does not integrate $T_{(4,1)}$ into $T_{(2,0)}$) before $S_3$ (not shown, but replaces $S_4$) begins to join the system. $S_2$, the elder, accepts $T_{(3,1)}$ and moves $T_{(4,1)}$ up to $T_{(4,2)}$. $S_3$ detects $S_4$'s failure when finding itself unable to establish a session with $S_4$ while completing the joining process, and flushes its tuple space portion into $S_2$'s tuple space portion by remotely calling a procedure on $S_2$. Flushing a tuple space portion is a process whereby ownership over the contents of a tuple space portion is relinquished to another server (the contents then exist in the tuple space portion of the owning server and all its replicas).

$S_3$'s flushing of its tuple space portion causes $S_2$ to detect $S_4$'s failure upon inspecting the system map carried along with the remote call from $S_3$. Hence, $S_2$ immediately handles the failure (i.e., before executing the flushing procedure called by $S_3$) by integrating $T_{(4,2)}$ into $T_{(3,1)}$. Having tolerated the failure, $S_2$ executes the remote procedure called by $S_3$, flushing $T_{(3,1)}$ (now containing $T_{(4,2)}$) into its portion, $T_{(2,0)}$.

![Figure 4.9: Section of a system that suffers a failure.](image)
Given servers $S_1$, $S_2$, $S_3$ and $S_4$ contained in the system partially depicted in figure 4.10, assume that server $S_2$ fails, that server $S_3$ detects the failure, but that server $S_4$ does not detect the failure. Server $S_1$ then removes tuple $\tau$, contained in $T_{(2,0)}$, the tuple space portion originally owned by server $S_2$. Upon detecting $S_2$'s failure, $S_3$ inherits that tuple space portion and integrates it into its primary tuple portion (i.e., $T_{(3,0)}$). Server $S_3$ removes tuple $\tau$ and attempts to update its replicas, one of which is maintained by server $S_4$. However, server $S_4$ cannot remove $\tau$ because its replica of $S_3$'s tuple space portion (i.e., $T_{(3,1)}$) is not equivalent to $S_3$'s tuple space portion (i.e., $T_{(3,0)}$) because it hasn't integrated its replica of $S_3$'s tuple space portion (i.e., $T_{(2,2)}$) into its replica of $S_3$'s tuple space portion (i.e., $T_{(3,1)}$).Factually, its replica of $S_3$'s tuple space portion does not contain tuple $\tau$.

![Figure 4.10: Section of a system suffering a failure into which a tuple is inserted.](image)

However, as every remote procedure call contains the system map of the caller, $S_1$'s call to $S_4$ to remove $\tau$ from $T_{(3,1)}$ results in $S_4$ detecting $S_2$'s failure as $S_2$ is no longer contained in $S_3$'s system map. As failures are tolerated before remote procedure calls not necessary for the toleration of failures, $S_4$ integrates $T_{(2,2)}$ into $T_{(3,1)}$ before satisfying the request to remove $\tau$ from $T_{(3,1)}$.

**Meta-Language**

Before the formal description of the protocols is presented, the notation used to express the algorithms is described. For clarity, the algorithms are shown in their
completeness, i.e., as if all contained RPCs are blocking. Wherever a non-blocking
RPC is used, the ellipses indicate that the functions that implement the given algorithm
are split at that point; the RPC is given a function pointer that implements the
algorithm fragment that follows the ellipses. Recall that when a function pointer is
provided to an RPC, the function being pointed to is executed when the RPC
completes.

Servers are referred to either by their SID, or by their status. For example, \( S_{\text{SID}} \)
refers to the server with the given SID. Furthermore, given that SID\text{'elder} evaluates to
the SID of the elder, \( S_{\text{SID\text{'elder}}} \) refers to the server that is currently the elder. For
convenience, \( S_{\text{SID\text{'elder}}} \) may be written as \( S_{\text{elder}} \).

Servers may also be referred to by their position in the system in relation to some
defined SID. Given that SID\text{+}\( \phi \) evaluates to the SID of the server whose logical SID is
\( \phi \) positions greater than the given SID, \( S_{\text{SID\text{+}\phi}} \) denotes the actual relative server. For
example, given a system with servers \( S_2, S_4, S_9, S_5, \) and \( S_1 \) whose logical SIDs are 1, 2,
3, 4 and 5 respectively, \( S_{4+2} \) refers to \( S_5 \) and \( S_{9+3} \) refers to \( S_2 \). Similarly, \( S_{\text{SID\text{-}\phi}} \) denotes
the server whose logical SID precedes the SID of the given server by \( \phi \) positions, e.g.,
in the system defined previously, \( S_{1.2} \) refers to \( S_9 \) and \( S_{2.2} \) refers to \( S_5 \).

The order of a server in relation to another server may be determined by calculating
the distance between the logical SIDs of the two servers. Because the servers are
joined in a ring, there are always two distances between any two servers. For example,
in the given system above, the two distances between servers \( S_5 \) and \( S_4 \) are \( S_5-S_4=2 \) and
\( S_4-S_5=3 \). The first distance is the \textit{successive order}, i.e., \( S_{4+2} \) evaluates to \( S_5 \), and the
second distance is the \textit{predecessive order}, i.e., \( S_{4.3} \) evaluates to \( S_5 \).

Formally then, the expression \( S_{a-S_P} \) yields the successive order, \( \lambda_a \), of \( S_a \) in relation
to \( S_P \), such that \( S_{P+\lambda_a} \) evaluates to \( S_a \). Similarly, the expression \( S_{P-S_a} \) yields the
predecessive order, \( \lambda_a \), of \( S_a \) in relation to \( S_P \), such that \( S_{P-\lambda_a} \) evaluates to \( S_a \).

The '\_' is used to denote syntactic elements that are not of concern in the current
scope. For example, \( T(\_\text{order}) \) denotes a tuple space portion of the stated \textit{order},
irrespective of which server owns that tuple space portion. The expression
\( T(\_\text{order}) \Rightarrow T(\_\text{order+1}) \) then denotes that the tuple space portion of the stated \textit{order}
becomes the tuple space portion of the stated \textit{order+1}, i.e., the order is incremented by
one.
To denote a parameter that is received, as opposed to one which is provided, the ‘&’ is used. For example, &p means that p is bound to the value being provided.

Requests made of, and received from other servers are expressed in terms of sending messages. For example, Session.Write( S_{SID}, ACCEPT_REPLICA( order ) ) denotes a call to S_{SID} for remote procedure ACCEPT_REPLICA. A single parameter, order, is passed to this procedure. Similarly, Session.Read( S_{&xi}, ACCEPT_REPLICA( &order ) ) denotes that a server receives a call to ACCEPT_REPLICA from some server, where &xi is bound to the SID of the server making the call. A single parameter, order, is received by this procedure from the caller.

Information other than remote procedure calls may be sent and received via sessions. For example, Session.Read( S_{elder}, CONFIGURATION => ( &p, &v, &SID ) ) denotes that a message labelled CONFIGURATION is received from the elder (the label is only included in the algorithm for clarity and is not used in the implementation). This message contains three parameters which, upon receipt, are bound to p, v and SID respectively. Similarly, Session.Write( S_{xi}, CONFIGURATION => ( p, v, SID ) ) denotes that a message labelled CONFIGURATION is written to the server whose SID is &xi.

Server Protocols

This section describes how servers join into OS/2-C-Linda systems, and how the integrity of the system is maintained despite servers joining and leaving the server cooperative. The protocol that ensures the integrity of the system despite server failures is also presented in this section.
Joining the Server Collective

The process of joining a system is formally described in algorithm 1 below. However, three salient issues should be discussed before presenting the algorithm: namely,

- the manner in which a new system is established,
- how a unique NetBIOS name is used as a token by servers attempting to become a system's elder, and
- the usage of a unique NetBIOS name by a joining server as a token to guarantee that it is the only server joining a given system.

A new system is established by the first server that attempts to join it. When a server wishes to join a system, it must contact the elder of that system. However, there will be no elder to contact when the system does not exist. Therefore, servers that wish to join a system first attempt to become the elder of that system: when the system exists there will be an elder and the new server's attempt to become the elder will fail. Conversely, when the system does not exist the new server's attempt to become the elder will succeed: having succeeded in becoming the elder (possible only when the system does not exist) the server creates the system.

Unique NetBIOS names are a convenient instrument for ensuring that only one server considers itself to be the elder. This instrument is used in the following manner: the elder is the server that succeeds in adding the elder name to the local name table. As NetBIOS guarantees that only a single instance of a unique name exists, at any one time, in all local name tables, only one server may add the elder name to its local name table. Because the elder name identifies the system that is created or joined, all servers that are to belong to a given system must be configured to use the same elder name. Two servers with differently configured elder names will join or create different systems.

In order to prevent multiple servers from joining an existing system simultaneously, and to be able to establish an initial session to the elder, another unique NetBIOS name is used: the joining name. Just like the elder name, the joining name is defined in the server configuration and must be defined to be same for all servers of a given system. Upon commencing the joining process the server obtains the joining name, relinquishing it only when it has completed the process.
OS/2-C-Linda

if not( Name.Add( elder name ) ) then
  loop
    if ( Name.Add( joining name ) ) then
      Session.Open( S_elder ) // the elder has the elder name
      Session.Write( S_elder, JOIN_SYSTEM )
      Session.Read( S_elder, CONFIGURATION => (&p, &v, &SID) ) // read configuration from elder
      Session.Close( S_elder ) // close the session with the elder
      Name.Add( SID ) // add my name
      accept_replicas_from_servers( SID, p, v ) // accept replicas from others
      assign_replicas_to_servers( SID, p, v ) // assign replicas to other servers
      if ( v ≤ p ) then
        block tuple space portions until v > p // cannot guarantee fault-tolerance
      end if
      Name.Delete( joining name ) // delete the joining name
      exit loop
    else
      delay by some defined time period // try again in a little while
    end if
  end loop
else
  assign first SID to myself // become elder
  Name.Add( SID ) // add my name as well
end if

Algorithm 1: Server joins an existing system, or creates a new one if none is found.

Once the elder has assigned an SID to the joining server, the joining server derives a unique name from that SID using a name template defined as a system configuration parameter. For example, given a name template of “LindaServer %03x” and an SID of 7, the unique NetBIOS name derived is “LindaServer 007”.

As shown in algorithm 2, accepting responsibility for the maintenance of the primaries of up to p other servers is achieved by opening a session to the server, by sending a request for a replica of the appropriate order, and by accepting the replica that the server sends in response to receiving the request. Although the server may derive the order of the replica requested from the SID of the server that sends the request, for simplicity the order of the requested server is passed as a parameter.
### Algorithm 2: Joining Server accepts replicas.

Algorithms 3 shows that the assignment of a replica to a server is achieved by opening a session to the server and sending a request to maintain a replica through the session. Provided as a parameter to the replica maintenance request is the replica that is to be maintained.

```prolog
accept_replicas_from_server(SID, p, v).:-
for order in 1..MINIMUM(p, v-1) loop
    Session.Open(S_{SID-order}) // open session to server
    Session.Write(S_{SID-order},
    ACCEPT_REPLICA(order)) // request the orderth replica
    Session.Read(S_{SID-order}, T_{(SID-order, order)}) // read the replica
end loop
```

### Algorithm 3: Server assigns a replica.

The algorithm given below, algorithm 4, shows that the assignment of replicas to no more than $p$ other servers is achieved by calling algorithm 3 for each replica assignment.

```prolog
assign_replica(SID, order, p, v).:-
if (order < (v-p)) then // only if not already connected
    Session.Open(S_{SID+order}) // establish session
end if
Session.Write(S_{SID+order},
    ASSIGN_REPLICA(T_{(SID, order)})) // assign the orderth replica
```

### Algorithm 4: Joining Server assigns replicas.

The next three algorithms (algorithms 5, 6 and 7) describe how servers respond when receiving requests. In order to describe any parameters passed as part of the
request, the statement that reads the request is included as the first step of each algorithm.

Upon receiving an ACCEPT_REPLICA request, a server writes the requested replica to the session with the requesting server. Should ρ servers already maintain a replica of the requested tuple space portion, the server initiates the dropping of the replica whose order now exceeds ρ. This will occur whenever a server joins a system where ν is greater than ρ, i.e., after the server has joined more than ρ+1 servers exist in the system.

Algorithm 5: Server response to joining server request to accept a replica.

In response to receiving a DROP_REPLICA request, servers drop the identified replica, and close the session with the server whose replica has just been dropped. However, as all servers must maintain a session to the elder for system-wide broadcasts, servers do not close sessions to elders when dropping replicas of the elder’s primary tuple space portion.

Algorithm 6: Server response to request to drop a replica.

Servers that receive an ASSIGN_REPLICA request respond by ‘rastering up’ the order of all replicas whose order is higher than the order of the replica received. Note that replicas whose order exceeds ρ are not dropped implicitly. Rather, such replicas
must be dropped explicitly by the owning server, i.e., the server that maintains the primary.

Algorithm 7: Server response to joining server request to assign a replica.

The only server that may receive JOIN_SYSTEM calls is the elder. When receiving such calls, elders respond in the manner outlined in algorithm 8. Notably, the elder assigns a new SID to the joining server and informs every server already in the system of the new server's SID before returning configuration information to the joining server.

Algorithm 8: Elder's response to joining server's request to join the system.

As a result of the broadcast, every server but the elder and the joining server receive a NEW_SERVER call. In response to these calls, each server updates its system map to include the new server's SID as shown in algorithm 9. When a SID is added to a system map, v is incremented and the tuple space becomes unblocked when v=p+1. Similarly, when a SID is removed from a system map, v is decremented and the tuple space becomes blocked when v=p.

Algorithm 9: Server response to elder's notification of a new server.
**Leaving the Server Collective**

A server that wishes to leave a system does so by simulating a server failure, as described by algorithm 10. However, to facilitate prompt detection and handling of the leaving server, the leaving server notifies the elder that it is leaving. In turn the elder notifies each server in the system which server has left. The simulation of a server failure is easily achieved by:

- informing the elder that it is leaving the system,
- dropping its primary tuple space portion,
- dropping all replicas that it maintains for other servers,
- aborting all open sessions, and
- ceasing execution.

```plaintext
Session.Write(S elder, LEAVING_SYSTEM(SID))
drop T(SID, o)
for order in 1..MINIMUM(ρ, v-1) loop
    drop T(SID+1, order)
end loop
```

**Algorithm 10**: Server simulates a failure in a readily detectable manner.

In response to receiving a LEAVING_SYSTEM call, the elder, as shown in algorithm 11, removes the SID of the leaving server from the system map and informs all other servers that the server has left. The servers that maintain a replica for the leaving server will tolerate the (simulated) failure upon receiving notification from a receiver thread that the session to the leaving server has been aborted. Similarly, the servers that maintained a replica for the leaving server will react to the (simulated) failure by assigning a replica of their primary tuple space portion to another server. Thus, informing each server in the system is done only to update every server's map of the system, and not to initiate any action to tolerate the failure.

Should the number of servers left in the system be insufficient to guarantee fault-tolerance to the configured resilience degree, the tuple space becomes blocked, i.e., no state changes may occur in any tuple space portion. The tuple space remains blocked until the number of servers in the system exceeds the resilience degree, i.e., until there are enough servers in the system to guarantee resilience to ρ concurrent failures.
OS/2-C-Linda

Algorithm 11: Elder's response to server's notification that it is leaving.

When a server receives a SERVER_LEFT call it removes the SID of the leaving server from its system map. The algorithm is formally presented in algorithm 12. No other action is initiated; any action required is activated when the session to the leaving server is aborted.

Algorithm 12: Server's response to elder's notification that a server is leaving.

Tolerating a Server Failure

When a server detects a failure it handles the failure by ensuring that its state changes accordingly, and relies on other servers to do the same. The failure of some server, for which a server maintains a replica, is handled by integrating the replica into the tuple space portion of the inheriting server (possibly itself). Should a failure result in the existence of less than \( \rho \) replicas of a server's primary tuple space portion, the server assigns a replica of its tuple space portion to another server, thus ensuring that at all times \( \rho \) replicas of its primary tuple space portion exist. Should the system contain less than \( \rho + 1 \) servers, all state changes to the tuple space must be suspended, as resilience can no longer be guaranteed.

Since the state of a tuple space portion may be altered only by the owning server, who must propagate any state changes to the replicas, any integration inconsistencies may be dealt with during that state change. The action taken upon detecting an inconsistency while inspecting the system map of the server requesting the state change is described in table 4.8.
Algorithm 13: Server's response to detecting a server failure.

Since the server (ψ) that maintains the ρth replica for an inheriting server (ι) may not
detect the failure, the inheriting server (ψ) must initiate the integration of a replica of
the failed server’s (φ) tuple space portion into its replica. Furthermore, as ψ does not
maintain a replica for φ (it would be of order ρ+1), ι must provide to ψ the tuple space
portion to integrate into its ρth replica. It may do so using the algorithm described in
algorithm 14.

Algorithm 14: Server integrates a tuple space portion.

Tuple Space Operators

The tuple space operators facilitate the manipulation of the tuple space, while
maintaining the tuple space’s structure that is necessary to guarantee its resilience to
failures. Briefly, for the purpose of providing an initial overview of scope, when a Linda client calls a Linda operation, the dynamically linked Linda library requests a Linda operation off the local Linda server through a named duplex pipe. The request is received by the server's client manager, who co-ordinates with the network managers of the local server, as well as those of other servers, to effect the operation on a tuple space portion and its replicas. Non-blocking RPCs are used to execute procedures (implementing fragments of Linda operator algorithms) on remote servers in order to operate on the tuple space portions maintained by those remote servers.

It should be noted at this point that the absence of an atomic multicast implementation is overcome by assuming that no failures occur during the set of point-to-point message exchanges simulating a multicast. Since the focus of this thesis is to construct a fault-tolerant Linda implementation, and not to implement yet another atomic multicast protocol, this assumption is deemed reasonable.

Just like the tuple space structuring protocols described earlier, the algorithms for the Linda operators are shown, for clarity, in their completeness, i.e., as if all RPCs are blocking. Below, all algorithm fragments following the ellipsis symbols are contained in functions called by the network manager upon receiving the RPC response, unless explicitly shown to be otherwise.

The Linda operators may be put into three categories: write, read and get. Linda's out and eval operators may be placed into the write category, the rd and rdp operators into the read category, and the in and inp operators into the get category.

**Tuple Signature Ownership**

Recall that every tuple or tuple template has a signature, that signatures are owned by tuple space portions, that every operator defines either a tuple or a tuple template, and that tuples and tuple templates have a tuple signature. All operators require that the owner of the signature of the tuple or tuple template being operated upon be located. Furthermore, if the owner of a signature cannot be found for the non-predicate operators (out, rd and in) in any tuple space portion in the system, the local tuple space portion must become the owner of the signature, because either a tuple or a request for a tuple will be added into the tuple space. In either case, the existence of the tuple or tuple template's signature in the tuple space is required. Note that if the owner of a signature cannot be found for a predicate operator (rpd and inp), the operator completes immediately, as no matching tuple exists in the tuple space.
As multiple servers may simultaneously request ownership over a given tuple signature, ownership is assigned by a single entity. As the elder is the only entity that is connected to all servers in the system, it is the elder that assigns signature ownership. Therefore, the elder maintains in its tuple space portion (the primary and its replicas) the signature of every tuple that exists in the tuple space. For each signature, the elder's portion either contains the set of tuples of that signature, or the SID of the server that owns the signature.

The Linda Operators

In order to operate on a tuple space portion using a tuple or tuple template, the signature of the given tuple or tuple template must be locked by obtaining a signature lock. Signature locks are used to serialise access to the tuple and tuple templates of a given signature, preventing concurrently occurring operations from interfering with another. As all operations must obtain a lock over a signature before being able to operate on the tuple space, and signature lock requests are serviced in order, a signature lock is obtained only after all operations currently requested for any tuple or tuple template of the given signature in the local tuple space portion complete. Furthermore, when locking a signature, all operations on any tuple of that signature are essentially blocked until the lock is released.

If the signature is owned by the primary tuple space portion of the local server, the lock is obtained, the operation is completed on the local tuple space portion (and its replicas), and a completion code is returned to the calling client. Should the signature not exist in the local tuple space portion, any attempts to lock it will result in a return code of LINDA_NOT_FOUND. If the local tuple space portion does not own the given signature, the server that maintains the tuple space portion that does own the given signature must be found, and the operation must be executed on the owning tuple space portion.

To reduce communication overheads, rather than engaging in a 2-phase protocol, where the first phase consists of finding the owner and the second phase consists of forwarding the operation to the owning server, the server called upon by the client forwards the operation to the elder. The elder then searches its tuple space portion for the SID of the owning server, and forwards the operation to that server. Should no server own the given signature, the elder inserts the signature into its tuple space portion, along with the SID of the calling server, and informs the calling server that it is
now the owner of the signature. The calling server, upon receiving such a notification, becomes the owner of the signature, and performs the operation on its tuple space portion.

Before attempting to operate on a local tuple space portion, every server must obtain a signature lock that is relinquished only when the operation has completed. Ownership over a signature is relinquished if, upon unlocking a signature:

• the local tuple space portion contains no tuples, or requests for tuples, of that signature, and

• there are no local operations currently blocked on requests to obtain the signature lock that is being unlocked.

Any server that is called upon by another server to operate on its tuple space portion will return, through the elder, any results to the calling server. The calling server, upon receipt of the results, acknowledges receipt, allowing the called upon server to discard the operation's results. Upon acquiring the results, either from another server or by operating on its own tuple space portion, the server called on by the client returns the results to the calling client.

Only the primary is locked, as any server failure will result in the re-submission, by the elder or calling server, of all incomplete operations for execution by the inheriting server. If the server that has failed is not the elder, the elder re-submits all incomplete operations to the server that inherits the tuple space portion of the failed server. However, if the server that has failed is the elder, the calling server re-submits all incomplete operations to the new elder. It is the responsibility of every server to re-submit incomplete operations to the new elder upon detecting the elder's failure. To prevent the multiple execution of the same operation as a consequence of re-submission following the detection of a server failure, each operation is tagged with an identifier unique to the calling server. The application of a wrapping logical clock (see above) is used, in conjunction with the SID of each server, to identify uniquely each operation by each server.

Contained in every tuple space portion, and its replicas, is the ID of the last operation performed by each server on the given tuple space portion, as well as the ID and results of every operation that has been completed, but for whom the receipt of results has not yet been acknowledged by the client's server. When receiving an operation, the server executes the operation only if it has not been completed before. If it has been completed before, and it has not yet been acknowledged, the server does not
execute the operation, but rather only returns the results again. Operations received for
whom a result receipt has been acknowledged (i.e., their ID is lower than the ID of the
last operation received from the given server) are ignored.

Servers discard the results of operations initiated by clients connected to servers that
have failed, as those results can never be acknowledged, nor can the operations be
re-submitted. Similarly, the ID of the last operation completed for a failed server is
discarded by every server that retained that ID.

Maintaining the Tuple Space Portion Replicas

Any operation performed on a tuple space portion is actively replicated on all
replicas of the given tuple space portion. A two-phase commit protocol is used: the
operation is performed on the local tuple space portion, and then its replicas,
committing the operation only when consensus has been reached that the operation will
complete successfully on all instances of the tuple space portion, i.e., the primary and
all replicas. Until consensus is reached, the operation may be complete, but its effects
are not visible until the operation is committed.

```
out( tuple ) :-

rc = TupleSpace.LockSignature( tuple.GetSignature() )
if (rc = LINDA_SUCCESS ) then
    rc = TupleSpace.Insert( tuple )
    TupleSpace.UnlockSignature( tuple.GetSignature() )
elsif ( rc = LINDA_NOT_FOUND ) then
    rc = Session.Write( Session, INSERT_TUPLE( tuple ) )
...
if (rc = LINDA_NOT_FOUND ) then
    rc = TupleSpace.CreateAndLockSignature( tuple.GetSignature() )
    if ( rc = LINDA_SUCCESS ) then
        rc = TupleSpace.Insert( tuple )
        TupleSpace.UnlockSignature( tuple.GetSignature() )
    end if
end if
end if
return (rc)
```

**Algorithm 15:** The *out* operator.
The `out` operator will attempt to operate on the local tuple space portion — termed `TupleSpace` in algorithm 15 above — or call upon the elder to forward the operation to the owning server. As the operators that operate on the local tuple space portion actively replicate the operations on the replicas of the local tuple space portion, these operators will be subject to suspension until the response from the called-upon server is received.

Should the elder determine that no server currently owns the given signature, it assigns ownership to the calling server and responds with a `LINDA_NOT_FOUND` code. When receiving such a response, a calling server inserts the given signature into its tuple space portion, thereby formally accepting ownership over the signature. Having done so, the calling server may insert the tuple into its tuple space portion.
Algorithm 16: Inserting a tuple into the owning tuple space portion.

Algorithm 16 above describes what any server, including the elder, does when asked to insert a tuple. If the server is the current elder, the server searches for the tuple’s signature: if the signature is not found, the elder creates it and returns LINDA_NOT_FOUND, causing the caller to assume ownership and to insert the tuple into its own tuple space portion. Conversely, if the signature is found, the elder, depending on whether it is the owner of the tuple’s signature, inserts the tuple in its own tuple space portion, or forwards the operation to the owning server. When forwarding an operation, as all operations of a given signature are serialised by the elder through its locking of the signature, and as the out operator does not result in the removal of a signature, the elder may unlock the signature before forwarding the
operation to the owning server. This brings about a slight performance improvement, as the elder is then free to process other operations of the same signature without having to wait for the forwarding RPC to complete.

```lisp
insert(t) :-
rc = TupleSpace.LockSignature(t.GetSignature())
if (rc = LINDA_SUCCESS) then
  // satisfy any pending operations, i.e., in and rd
rc2 = TupleSpace.SatisfyPendingOperations(t)
if (rc2 = LINDA_SUCCESS) then
  rc = TupleSpace.AddTuple(t)
elsif (rc2 = LINDA_TUPLE_CONSUMED) then
  rc = LINDA_SUCCESS
end if
end if
return (rc)
```

**Algorithm 17:** Inserting a tuple, first satisfying pending operations.

When inserting a tuple, the server first satisfies any pending operations, as shown by algorithm 17. A pending operation is any in and rd for which, at the time that they were first received by the owning server, a matching tuple could not be found. When an in or rd operation is received for which a matching tuple cannot be found, the owning server blocks the operation by placing it into a pending state. No response to such operations will be sent to their caller, until a matching tuple is placed into the tuple space. When a matching tuple is eventually placed into the tuple space, the server satisfies all pending operations until the tuple is consumed by an in operation.

As Linda does not define an order in which blocked requests are to be satisfied, any order is by definition ‘correct’. This implementation uses a first-in-first-out (FIFO) ordering: blocked operations are satisfied in the order in which they arrive. The server does not increase the number of pending operations that are satisfied, by satisfying all non-consuming (i.e., rd operators) first.
Algorithm 18: Locking a signature.

Signatures are locked, as shown in algorithm 18, in the primary tuple space portions only, as the replicas are updated only once the signature lock is obtained in the primary tuple space portion. Furthermore, signature locks do not have to survive server failures: any operation currently incomplete will be re-presented to the owning server by the elder, or the calling server in case of elder failure.

Within each tuple space portion, the signatures are contained in a signature table. The signature table is a monitor-like data-structure, i.e., all access to it is guarded by a mutex semaphore. Consequently, a single thread only may lock a signature at a time. To conserve operating system resources, rather than allocating a semaphore to each signature, a simple boolean is used to indicate that a given signature is locked. Therefore, locking a signature involves locking the entire signature table, setting the signature’s lock boolean to true, and unlocking the signature table.
CreateSignature( signature ) :-
if ( ( rc = Signatures.ObtainMutex() ) = LINDA_SUCCESS ) then
  // If the signature definitely doesn't exist — but if it does then that is OK too
  if ( ( entry = Signatures.Find( signature ) ) = NULL ) then
    rc = Signatures.AddSignature( signature )
  end if
  Signatures.ReleaseMutex()
end if
return ( rc )

Algorithm 19: Creating a signature in the local tuple space portion.

Signatures are created in the primary tuple space portion and all replicas using
algorithm 19. Because these protocols are never called directly, but rather only as a
result of higher-level operations by other servers, that will get re-executed when a
failure occurs, they do not need to be explicitly fault-tolerant.

AddSignature( signature ) :-
rc = Signatures.Add1( signature ) // Add locally, non-committed
if ( rc = LINDA_SUCCESS ) then
  for order in 1..p loop
    rc = Session.Write( S_sid,order, ADD_SIGNATURE( signature ) )
    if ( rc ≠ LINDA_SUCCESS ) then
      for rollback in 1..order loop
        Session.Write( S_sid,rollback, ROLLBACK_SIGNATURE( signature ) )
      end loop
      Signatures.AddRollback( signature ) // roll back local addition
      return ( rc ) // exit immediately
    end if
  end loop
  Signatures.Add2( signature ) // commit signature added earlier
end if
return ( rc )

Algorithm 20: Adding a signature into a tuple space portion.
Signatures are added into a tuple space portion by the owning server, referred to as S_a later, using algorithm 20. Algorithm 20 uses two-phase commit protocols to add the signature into the primary tuple space portion, as well as its replicas. First, the signature is added, using Add1, into the local tuple space portion, but the addition is not committed. Only when the signature is also added to the replicas of the tuple space portion is the signature committed by a call to Add2. Until the signature is committed or rolled back, any attempt to lock the signature will block. The attempt unblocks and completes when the signature is committed, or fails when the signature is rolled back.

The replicas are updated using a set of remote procedure calls: the first call causes the receiving server to add, tentatively, the signature to the replica of the S_a's primary using Add1. Should any attempt by S_a result in a non-successful return code, S_a returns the non-successful return code to its caller, but not before dropping the tentative signature created in its primary, as well as directing all servers that have already added the tentative signature to drop the signature.

However, should all servers that maintain a replica respond with a positive return code, S_a commits the signature in the replicas and in the primary of its tuple space portion. Note that due to the absence of a reliable multicast, the assumption is made that no failure occurs while committing the tentative signature addition to the replicas.

```
Session.Read(S30+order, ADD_SIGNATURE( signature)): -
    ...
    return (Signatures.Add1( signature )) // Add locally, non-committed
```

**Algorithm 21:** Phase one of adding a signature — this may fail.

When a server wishes to update the replicas of its tuple space portion it remotely calls on the servers that maintain the replicas. These servers behave as described in algorithm 21: they apply the same function to the replica that the calling server has already applied to its primary.

```
Session.Read(S30+order, COMMIT_SIGNATURE( signature)): -
    ...
    Signatures.Add2( signature ) // commit earlier added signature
    return (LINDA_SUCCESS )      // its a commit: it must succeed
```

**Algorithm 22:** Phase two of adding a signature — this cannot fail.
Having determined that the signature will successfully be added to all replicas, the server will commit that signature addition. When a server wishes to commit a signature addition to its replicas it calls on the remote servers, as described in algorithm 22. As the committal cannot fail, this RPC will always return a successful return code.

```
Session.Read(S_id+order, DROP_SIGNATURE(signature)) :-
    ... return (Signatures.Delete(signature))
```

**Algorithm 23:** Delete a signature from the replica of the caller’s tuple space portion.

A server that relinquishes ownership over a signature deletes the signature from the tuple space portion’s replicas by remotely calling upon each server. The behaviour of a server upon receipt of such a call is described by algorithm 23.

```
Session.Read(S_id+order, ROLLBACK_SIGNATURE(signature)) :-
    ... return (Signatures.AddRollback(signature))
```

**Algorithm 24:** Rollback a signature addition from a tuple space portion replica.

A server that cannot successfully add an uncommitted signature into every replica of its tuple space portion must delete the signature from those replicas that contain the uncommitted signature. Since the signature has not yet been committed, attempts to delete it must fail. Therefore, `AddRollback()` is used to roll the signature addition back.
CreateAndLockSignature( signature, &owner):-

if ( (rc = Signatures.ObtainMutex() ) = LINDA_SUCCESS ) then
  if ( ( entry = Signatures.Find( signature ) ) ≠ NULL ) then
    entry.lock = TRUE
    owner = entry.SID
  else
    rc = LINDA_INTERNAL_ERROR
    // already locked and it shouldn’t be
  end if
else
  rc = LINDA_NOT_FOUND
end if
Signatures.ReleaseMutex()
end if
return ( rc )

Algorithm 26: Finding the owner of a signature, locking the signature.
Algorithm 26 is a specialisation of algorithm 18. However, this form of the algorithm returns to the caller the SID of the server that owns the given signature.

```
UnlockSignature(signature) :-
    if ((rc = Signatures.ObtainMutex()) = LINDA_SUCCESS) then
        if ((entry = Signatures.Find(signature)) ≠ NULL) then
            if (entry.lock) then
                entry.lock = FALSE
            else
                rc = LINDA_INTERNAL_ERROR // it's not locked, so cannot unlock
            end if
        end if
        Signatures.ReleaseMutex()
    end if
    return (rc)
```

**Algorithm 27: Unlocking a signature.**

Signatures that are locked must be unlocked when the operation being performed is complete. The unlocking of signatures is described by algorithm 27.

```
SatisfyPendingOperations(tuple) :-
    if ((rc = Signatures.ObtainMutex()) = LINDA_SUCCESS) then
        if ((entry = Signatures.Find(signature)) ≠ NULL) then
            while (operation = entry.GetOperation(tuple)) loop
                rc = operation.Unblock(tuple) // unblock the pending operation
                if ((rc = LINDA_SUCCESS) && (operation.GetSelector = in)) then
                    rc = LINDA_TUPLE_CONSUMED // the in operator consumes tuples
                end if
                if (rc ≠ LINDA_SUCCESS) then
                    break; // exit the while loop
                end if
            end loop
            Signatures.ReleaseMutex()
        end if
    end if
    return (rc)
```

**Algorithm 28: Satisfy pending operations with the given tuple.**
When inserting a tuple into the tuple space, operations that are currently blocked (because no matching tuple exists in the tuple space) may be unblocked and completed. The blocked operations are attached to the signature in a queue according to the order in which the operations were blocked. Recall that although the signatures of the tuple and the tuple template of every blocked operation match, the given tuple may not match the tuple template of any blocked operation.

Therefore, as shown in algorithm 28, the queue is traversed using the given tuple, unblocking and executing only those operations whose tuple template matches the tuple. The traversal stops when there are no more blocked operations, or until the tuple is consumed by an in operator. The call to the blocked operation (shown as operation.Unblock()) returns a code to the Linda client before completing, returning the same code to the caller, i.e., the server does not have to return the code to the client as this has already been done.

```
AddTuple( tuple ) :-
rc = TupleSpace.Add1( tuple )    // Add locally, non-committed
if ( rc = LINDA_SUCCESS ) then
  for order in 1..p loop
    rc = Session.Write( S碇order, ADD_TUPLE( tuple ) )
    if ( rc ≠ LINDA_SUCCESS ) then
      for rollback in 1..order loop
        Session.Write( S碇rollback, ROLLBACK_TUPLE( tuple ) )
      end loop
    TupleSpace.AddRollback( tuple )    // roll back local addition
    return ( rc )    // exit immediately
  end loop
end if
for order in 1..p loop
  // Note: we assume no failure occurs in this loop!
  Session.Write( S碇order, COMMIT_TUPLE( tuple ) )
end loop
TupleSpace.Add2( tuple )    // commit tuple added earlier
end if
return ( rc )
```

**Algorithm 29:** Adding a tuple into a tuple space portion.
The addition of a tuple into a tuple space portion, shown by algorithm 29, involves the same steps as the addition of a tuple signature (shown by algorithm 20). First, the tuple is added tentatively into the local tuple space portion. Then, the tuple space portion’s replicas are updated tentatively by directing each server that maintains a replica to update it accordingly. Should any server updating a replica respond negatively, all servers that have already updated the replica are directed to discard the tentative update, i.e., to perform a rollback.

When all replicas have responded positively to the request to update the replica of the tuple space portion, they are directed to commit the tentative operation. Finally, the tuple space addition is committed to the local tuple space portion, and a return code is returned to the caller.

```
Session.Read(Siid,order•ADD_TUPLE(tuple)):­
    ... return (TupleSpace.Add1(tuple)) // Add locally, non-committed
```

**Algorithm 30: Phase one of adding a tuple — this may fail.**

When a server wishes to add a tuple to the replicas of its tuple space portion, it remotely calls on the servers that maintain the replicas, just as it does when adding a signature. These servers behave as described in algorithm 30: they apply the same function to the replica that the calling server has already applied to its primary.

```
Session.Read(Siid,order•COMMIT_TUPLE(tuple)):-
    ... TupleSpace.Add2(tuple) // commit earlier added tuple
    return (LINDA_SUCCESS) // its a commit: it must succeed
```

**Algorithm 31: Phase two of adding a tuple — this cannot fail.**

When a server wishes to commit a tuple addition to a replica it calls on the remote server to behave in the manner described by algorithm 31. As the committal cannot fail, this RPC will always return a successful return code.
Algorithm 32: Rollback a tuple addition from a tuple space portion replica.

A server that cannot successfully add a tuple into every replica of its tuple space portion must (using algorithm 32) delete the tuple from those replicas that contain the tuple tentatively. Note that AddRollback() is used to roll the tuple addition back, as the tuple has not yet been completely added and may, therefore, not be deleted.

Algorithm 33: The read operator, a generalisation of in, inp, rd and rdp.

The process of reading a tuple from the tuple space is described by algorithm 33 above. Recall that the in, rd, inp and rdp operators define a tuple template for which a matching tuple is to be found in the tuple space. The process, thus, involves first locking the signature of the tuple template for which a matching tuple is to be found. The signature lock is obtained only when the local tuple space portion owns the signature. Therefore, either a matching tuple is found in the local tuple space portion, or a matching tuple may be placed into the local tuple space portion at a later time. In
either case, the local tuple space portion is the tuple space portion that must be operated on for the given tuple request.

Should the attempt to lock the given tuple template’s signature succeed, the operation is performed on the local tuple space portion, and its replicas, and the results are returned to the calling client. However, should the attempt to lock the given tuple template’s signature fail (because the local tuple space portion does not own the signature), the elder is asked to find and forward the request to the owner. Should the elder fail to find the requested signature, the elder assigns ownership over the given tuple template’s signature to the calling server if and only if the operation is a blocking operation, i.e., in or rd. Recall that the predicate forms of those operators (inp and rdp) return to the calling client with an appropriate return code, without blocking should no matching tuple exist in the tuple space.

When the elder informs the calling server that no owner is found for its blocking operation, the calling server deduces that it is now the owner and creates, and locks, the signature in its tuple space portion. The server has also established that no matching tuple exists in the tuple space and it must, therefore, block the operation by adding the necessary details into the tuple space portion. These details include the tuple template, whether the operation is consuming, and the ID of the client. Recall that client IDs are assembled from the SID of the server to whom they are connected, and an ID that uniquely identifies each client connected to a server.

Note that, although clients are considered to have failed should the server to whom they are connected fails, a tuple space portion may contain the pending operations of clients of other servers. Therefore, pending operations must be inserted into the primary, as well as all replicas of a given tuple space portion. However, upon detecting a server failure, all pending operations of clients connected to the failed server must be discarded.

```
TupleSpace.Read( tuple, template, consume ):-
rc = Signature.Find( tuple, template )  // find tuple matching template
if ((rc = LINDA_SUCCESS) && (consume)) then
    Signature.Delete( tuple )          // remove matching tuple
end if
return( rc )
```

**Algorithm 34:** Obtaining a tuple from a tuple space portion.
When a server, having established that it owns the signature of an operation's tuple template, wishes to apply the operation to its tuple space portion, it behaves as described by algorithm 34. Should any search for a matching tuple prove successful, the server returns the tuple to the caller. If the operation being performed is consuming, i.e., a rd or rdp operation, the tuple is removed from the tuple space before being returned to the caller.

```
Session.Read(S_caller, READ_TUPLE( CID, tuple, template, consume, block ) ) :-
  ...
  if ( SID = SID'elder ) then // if I am the elder
    rc = TupleSpace.FindAndLockSignature( template.GetSignature(), owner )
    if ( rc = LINDA_SUCCESS ) then
      rc = Session.Write( S_owner, READ_TUPLE( CID, tuple, template, consume, block ) )
      ...
      TupleSpace.UnlockSignature( template.GetSignature() )
    end if
  else // I'm not the elder, I must be owner
    rc = read( CID, template, consume, block, tuple )
  end if
  return ( rc )
```

**Algorithm 35: Read the tuple from a tuple space portion.**

Should a server not own the signature, it must forward the operation to another server using algorithm 35. The server called upon by the Linda client forwards the operation to the elder, who determines the owner of the given signature. If the elder owns the given signature, it executes the operation on its tuple space portion, otherwise it forwards the operation to the owning server who executes the operation on its tuple space portion. Should there be no owner, the elder assigns ownership to the calling server and returns an appropriate code to the calling server. The calling server, upon receiving a code that indicates that the signature was not owned and that it is now the owner, performs the operation (that caused the elder to assign ownership to the calling server) on its tuple space portion.
DropOwnership(signature):­

TupleSpace.DeleteSignature(signature)

// if the elder fails to drop the signature, the consequences are negligible
Session.Write(S_elder, DROP_SIGNATURE(signature))

**Algorithm 36: Drop ownership over a signature.**

When a server wishes to relinquish ownership over a signature, it must, as shown by algorithm 36, delete the signature from the primary of its tuple space portion, as well as all replicas. The server deletes the signature from the primary before directing all servers that maintain a replica of its tuple space portion to delete the signature from the replica. Note that only after having deleted the signature from the local tuple space portion, does the server direct the elder to drop the given signature. Should the elder direct any operation to the server after the signature has been deleted from the tuple space portion, but before the elder has deleted the signature out of its tuple space portion, the server's tuple space portion becomes the owner once again, i.e., the direction of the elder is final and is obeyed.

The elder's decision as to who owns a given signature is final because deleting the signature from the elder before deleting the signature from the local tuple space portion could compromise integrity, as demonstrated in the following scenario. A server, S_p, is assigned ownership by the elder before server S_o, the server relinquishing ownership, has updated its replicas. Any failure at that point would result in the inheriting tuple space portion, the primary of S_p, considering itself to be the owner of the signature whose ownership S_p was in the process of relinquishing.

**Primitive Operations of the Tuple Space**

Each tuple space portion is implemented by a distinct instance of the tuple space ADT. The tuple space is an abstract data type (ADT) that provides a set of operations that may be performed on it. Figure 4.11 shows the structure of that ADT.
The tuple space itself contains other ADTs: the main ADT being a signature table. Each signature in the table contains a set of tuples and a set of pending operations, each of the latter itself contains a tuple template. The set of operations that may be performed on each of the ADTs contained in the tuple space ADT is enumerated below:

**Table 4.9: The operations that may be performed on a tuple space.**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add1</td>
<td>First phase of tuple addition. The tuple thus added must be committed using Add2(), or rolled back using AddRollback().</td>
</tr>
<tr>
<td>Add2</td>
<td>The second phase of tuple addition. The tuple must have been added using Add1().</td>
</tr>
<tr>
<td>AddRollback</td>
<td>Tuple Rollback. The tuple thus rolled back must have been added using Add1(), and not have been committed using Add2().</td>
</tr>
<tr>
<td>Delete</td>
<td>Delete a tuple from the tuple space.</td>
</tr>
</tbody>
</table>
For clarity, the operations that may be performed on the tuple space have been divided into groups that reflect the ADTs comprising the tuple space. The following table enumerates the operations that may be performed on the signature table component of the tuple space.

**Table 4.10: The operations that may be performed on a signature table.**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add1</td>
<td>First phase of signature addition. The signature thus added must be committed using Add2(), or rolled back using AddRollback().</td>
</tr>
<tr>
<td>Add2</td>
<td>The second phase of signature addition. The signature must have been added using Add1().</td>
</tr>
<tr>
<td>AddRollback</td>
<td>Signature Rollback. The signature thus rolled back must have been added using Add1(), and not have been committed using Add2().</td>
</tr>
<tr>
<td>Delete</td>
<td>Delete a signature from the local signature table.</td>
</tr>
<tr>
<td>Find</td>
<td>Find the signature record for the given tuple's signature.</td>
</tr>
</tbody>
</table>

Similarly, the table below enumerates the operators that may be performed on a single tuple signature record of a tuple space. Recall that each signature may contain any number of tuples of the given signature, and any multiple tuple requests for tuples of that tuple signature.

**Table 4.11: The operations that may be performed on a signature.**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetOperation</td>
<td>Get the oldest blocked operation that matches the given tuple.</td>
</tr>
<tr>
<td>Find</td>
<td>Find a tuple that matches the given tuple template</td>
</tr>
</tbody>
</table>

The only operation that may be performed on a tuple, or tuple template, contained in a tuple space is to obtain its signature.

**Table 4.12: The operations that may be performed on a tuple.**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetSignature</td>
<td>Returns the signature of the given tuple or tuple template.</td>
</tr>
</tbody>
</table>

Blocked tuple operations, i.e., operations for tuples for which no matching tuple is contained in the tuple space, may be operated upon using the following two functions. Both functions are necessary to unblock the blocked operation.
Table 4.13: The operations that may be performed on a tuple operation.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unblock</td>
<td>Complete the operation blocked earlier by applying the operation to the</td>
</tr>
<tr>
<td></td>
<td>local tuple space portion (i.e., the primary and its replicas), and returning</td>
</tr>
<tr>
<td></td>
<td>the results to the client.</td>
</tr>
<tr>
<td>GetSelector</td>
<td>Returns the operation of the blocked operation, i.e., in, rd, inp or rdp.</td>
</tr>
</tbody>
</table>

4.4 Proof of Tuple Space Fault-Tolerance

The main goal of this thesis, the implementation of a fault-tolerant tuple space, has been reached and a fault-tolerant tuple space is demonstratable. Although the implementation of the Linda operators to manipulate the fault-tolerant tuple space are necessary for this Linda implementation to be of any practical use, they are not paramount for the main goal of this thesis. Consequently, the implemented Linda operators operate on local tuple space portions only, but they are sufficient to demonstrate that the tuple space is fault-tolerant.

The development and test environment consists of two PCs running the OS/2 operating system, connected via a Token Ring LAN. IBM’s Communications Manager is installed on both OS/2 machines, providing an implementation of NetBIOS. As only two OS/2 machines are available, for systems consisting of more than two nodes multiple servers (each with their own configuration) are executed per OS/2 machine.

To prove that the tuple space provided by a set of Linda servers tolerates the failure of participating Linda servers, a set of tests was executed. The set of tests contained a range of system configurations and a cross-section of possible failures (see table 3.3). System configurations varied in the number of servers contained in each system, and in
the resilience degree of each system: the number of servers ranged from two through to five, whereas the resilience degrees ranged from one through to three. Each test consisted of:

- configuring the system to the desired resilience degree; as the resilience degree in effect for a system is determined by the first elder (i.e., the first server to start in the life-time of a system), the configuration for one server only must be changed,

- starting the servers (giving each server a distinct identity by naming a unique configuration file as a command-line argument) that are to participate in the system,

- populating the primary tuple space portion of a server by using the implemented Linda operators,

- writing the primary tuple space portion onto disk by using the console,

- restoring the stored tuple space portion by using the console,

- failing the desired number of servers in a manner that appropriately simulates the failure to be tested, and

- displaying, for each server that was failed, the contents of the primary tuple space portion of the server that inherits the tuple space portion of the failed server: the displayed tuple space portion should contain the tuples that the tuple space portion of the failed server contained.

Although the implemented Linda operators do not actively replicate any state changes onto the replicas of the primary tuple space portion being operated upon, restoring a tuple space portion does appropriately update the replica tuple space portions. Thus, the functionality of detecting and tolerating a server failure may be tested.

A crash failure (e.g., a pause or halting-crash failure) is simulated by rebooting the OS/2 machine that the server is running on for which a crash failure is to be simulated. An omission failure is simulated by generating a timing failure, which gets masked into an omission failure. A timing failure is generated by starting a process that sets its priority to the highest possible priority, and then consumes CPU cycles, effectively starving all other processes running on the OS/2 machine, including the server, of CPU cycles. Because only two OS/2 machines were available, and both the crash and omission failure simulations affect every process on the OS/2 machine that the failure simulation is being performed on, multiple concurrent failures may only be simulated by failing processes on the same OS/2 machine.
Only two of the three possible failures are simulated, as all failures are masked into crash failures: crash failures (which are not masked) test that such failures are detected and tolerated, whereas timing failures (which are masked into omission failures, thence are masked into crash failures) test that masking does occur and that the resulting failure is detected and tolerated.

**Table 4.14:** *The set of system configurations subjected to failures of varying type and concurrent quantity.*

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>single crash failure</td>
<td>single crash failure</td>
<td>single timing failure</td>
<td>single timing failure</td>
</tr>
<tr>
<td>2</td>
<td>N/A</td>
<td>dual crash failure</td>
<td>single timing failure</td>
<td>triple timing failure</td>
</tr>
<tr>
<td>3</td>
<td>N/A</td>
<td>N/A</td>
<td>single timing failure</td>
<td>triple crash failure</td>
</tr>
</tbody>
</table>

Each configuration, as shown in table 4.14, was subjected to one or more concurrent failures from the range of possible failures, where the number of concurrent failures ranges from one through to the current $p + 1$. Three of the configurations, as shown in table 4.14, are not available for testing as $v \leq p$, i.e., there are not enough servers to guarantee the given resilience degree (see section 4.2). Furthermore, three concurrent failures, as caused for the five-server system configured for dual redundancy, will exhaust redundancy. When the number of concurrent failures exceeds the resilience degree, data is lost, i.e., the failures cannot be tolerated.

In all cases the failure was detected and tolerated by the remaining servers. The procedure described above demonstrated that no information was lost from the tuple space, i.e., that the tuple space did tolerate the failure. As a fault-tolerant tuple space has been demonstrated, the fundamental research goal of this thesis has been met.
5 Conclusion

The goal set for this research, to develop a fault-tolerant tuple space, has been reached. In doing so, this research expounds a manner in which a tuple space may be made fault-tolerant in a particular environment. Before completing this thesis by recommending future research directions, a summary of the study that has lead to these results is offered.

5.1 Summary

Chapter one introduces the thesis by relating relevant background information, placing the study into context by observing its significance, and outlining the research objectives, before formally defining the statement of study and describing the structure of the thesis.

The second chapter concerns itself with a review of the literature. It opens with a survey of Linda's evolution that is presented, roughly, in chronological order, and follows into an enumeration of known Linda implementations and Linda derivatives. An explanatory overview of the field of fault-tolerance is included next, continuing into an examination of suggested mechanisms and methodologies that may be used to construct fault-tolerant software. The literature review concludes with a list of known fault-tolerant Linda implementations, other than OS/2-C-Linda.

Chapter three contains the theoretical framework that, along with the literature review, establishes the foundation for the study. Beginning with a description of components provided by the underlying operating system that may be used to construct software, the theoretical framework progresses into a synopsis of the Linda paradigm. A failure-taxonomy drawn from a number of sources is presented, before the theoretical framework closes with descriptions of methods that may be used to tolerate failures.

The study's product, OS/2-C-Linda, is used as a vehicle in the development of a fault-tolerant tuple space, with chapter four detailing its software development. Following the typical software development life-cycle, the chapter commences with defining the problem and identifying the requirements. Each requirement is categorised as necessary, negotiable, or enhancing: the last categorisation delineating the scope of the study.
Conclusion

From the identified requirements a solution is offered that calls for the distribution and replication of the tuple space over a dynamic collection of OS/2-C-Linda servers: each server runs on an OS/2 machine, with each OS/2 machine (possibly) accommodating multiple servers. By structuring the tuple space in this fashion, the amount of replication and the resilience degree are configurable.

The design is bound to the underlying operating system in the implementation section, where the structuring of the server as a collection of threads is elaborated. However, the bulk of the work pertains to the interaction in which servers must engage in order to realise collectively a tuple space. To that end, NetBIOS is abstracted by an RPC layer that not only provides a convenient manner of interacting with other servers, but also augments it by maintaining a map of the servers participating in the manifestation of the tuple space, and by providing the ability to detect server failures.

The protocols that are used by the servers to provide the structure needed to maintain the tuple space are presented, along with those protocols that may be used by the Linda operators to manipulate the tuple space. For completeness, the data structure used by each server to represent a tuple space portion, and manipulated by the latter set of protocols, is described. Chapter four concludes with a description of the test scheme which demonstrates that the tuple space does tolerate failures.

5.2 Further Research

As the Linda operators are implemented to manipulate the local primary tuple space portion only, they do not effect the blocking of the tuple space when the number of servers currently contained in the system is insufficient to guarantee the specified resilience degree. Consequently, when a server leaves the system and the number of servers no longer exceeds the resilience degree, the tuple space cannot become blocked (see section 4.2) until another server joins the system, i.e., until \( v > p \).

Similarly, when the number of concurrent failures exceeds the resilience degree, integrity of the tuple space is destroyed as the failures cannot, therefore, be tolerated. OS/2-C-Linda servers do detect this failure, and propagate it to the user by terminating themselves. Eventually all servers in the system will detect the failure and will terminate, thereby closing down the entire system (i.e., the only appropriate action).

The implementation of Linda operators that manipulate the fault-tolerant tuple space (a design for all operators but \textit{eval} is presented in this thesis) is left as a further
Conclusion

research topic. By implementing these Linda operators a parallel tool would be made available for use in a (relatively) cost-effective and readily available environment.

Another topic of further research is overcoming the limitation, suffered by the current implementation, that no more than \( p \) servers may leave the system simultaneously. This may be achieved by retaining the departing servers in the system until their portions of the tuple space have been transferred to other servers.

The algorithm employed by OS/2-C-Linda to distribute the tuple space may lead to an unevenly loaded system: the worst case scenario being one server’s tuple space portion owning all signatures. Consequently, the algorithm used to distribute the tuple space requires further refinement. Although the current algorithm assigns ownership to the first server that presents a given signature, the initial assignment may eventually become inappropriate. For example: client \( C_1 \) inserts a tuple of signature \( \alpha \) into the tuple space, thereby causing ownership over signature \( \alpha \) to be assigned to \( S_1 \). Client \( C_2 \), attached to server \( S_2 \), then starts to utilise that signature heavily, making \( S_1 \)’s ownership over signature \( \alpha \) inappropriate as \( S_2 \) needs to perform the operations remotely on \( S_1 \)’s tuple space portion.

OS/2-C-Linda’s use of NetBIOS for inter-server communication via the LAN brings about a number of limitations that may be overcome by further research. As NetBIOS limits the number of sessions that may exist concurrently to 254 sessions, the resilience degree and the number of nodes participating in that system are limited by the relation \( pv+(v-(2p+1)) \leq 254 \). Furthermore, as the number of names that may exist concurrently is limited to 255, \( v \) is limited to 254 (the elder having two names). By employing a multi-cast capable communications protocol rather than NetBIOS’s session support, these limitations may be overcome. Furthermore, the use of such a protocol may guarantee that no failures occur while engaging in a simulated multi-cast.
Appendix 1: References


References


References


References


References


Appendix 2: A Bibliography


Bibliography


Bibliography


Pinakis, J. (1993?). *Remote Thread Execution*. Perth, Australia: The University of Western Australia, Department of Computer Science.


