SUPPORTING FAULT-TOLERANT PARALLEL PROGRAMMING IN LINDA

(Ph.D. Dissertation)

David Edward Bakken

TR 94-23

August 8, 1994

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

This work was supported in part by the National Science Foundation under grant CCR-9003161 and the Office of Naval Research under grant N00014-91-J-1015.

SUPPORTING FAULT-TOLERANT PARALLEL PROGRAMMING IN LINDA

by

David Edward Bakken

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

1994

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under the rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of the manuscript in whole or in part may be granted by the copyright holder.

SIGNED: ______________________________

ACKNOWLEDGMENTS

I wish to express sincere, profound, and profuse thanks to my advisor, Rick Schlichting. His encouragement, guidance, friendship, and humor have been a delight to experience. He has greatly honed my research and writing skills and has been a researcher to venerate and emulate. I thank Greg Andrews for all he taught me in so many different ways about concurrent programming, research, and writing. I thank him and Larry Peterson for their contributions to my research and for serving on my committee. And I thank Mary Bailey for serving as a substitute on my final committee and for reviewing this dissertation.

Others have contributed directly to this research. I thank Vic Thomas and Shivakant Mishra for their many discussions regarding this research, and Shivakant for developing Consul. I thank Jerry Leichter, LRW Systems, and GTE Laboratories for providing the Linda precompiler that was used as a starting point for the FT-Linda precompiler. I thank Rob Rubin, Rick Snodgrass, Dennis Shasha, and Tom Wilkes for their useful comments on FT-Linda. I thank Dhananjay Mahajan and Sandy Miller for their work on Consul. I thank Ron Riter and Frank McCormick for teaching me so much about software and systems during my time at Boeing; these lessons were invaluable in graduate school and will no doubt continue to be so during the rest of my career. I thank Brad Glade for suggesting read-only tuple spaces. I thank the National Science Foundation for supporting this work through grant CCR-9003161 and the Office of Naval Research through grant N00014-91-J-1015.

Many people have made my stay in Tucson delightful. I thank the department's system administrators and office staff for keeping things running so very smoothly. I thank my fellow graduate students Doug Arnett, Nina Bhatti, Peter Bigot, Wanda Chiu, Peter Druschel, Curtis Dyreson, Vincent Freeh, Patrick Homer, Clint Jeffery, Nick Kline, David Lowenthal, Vic Thomas, Beth Weiss, and Andrey Yeatts for their friendship. I thank Anwar Abdulla, Yousef Akeel, Harvey Bostrom, Nancy Nelson, and Nayla Yateem for making my stay in Tucson more pleasant, each in their own ways. I am thankful for the sweet fellowship of the members of the prayer group, especially Fred and Ruth Fox, Derek and Shannon Parks, and Craig and Mary Bell.

I thank my parents for their love and support over the years, and I thank my brother as well as Gram and Joanne. And I can never hope to thank my dear wife, Beth, or my precious children, Abby and Adam, enough for their love and encouragement during my studies. Enduring my grad school habit was difficult for them all.

Finally, and foremost, I thank God for providing me a way to heaven through His Son, Jesus Christ.

Dedicated to the memory of my twin brother
Gregory Harold Bakken
January 2, 1961 – August 16, 1987

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

As people are becoming increasingly dependent on computerized systems, the need for these systems to be dependable is also increasing. However, programming dependable systems is diffcult, especially when parallelism is involved. This is due in part to the fact that very few high-level programming languages support both fault-tolerance and parallel programming.

This dissertation addresses this problem by presenting FT-Linda, a high-level language for programming fault-tolerant parallel programs. FT-Linda is based on Linda, a language for programming parallel applications whose most notable feature is a distributed shared memory called tuple space. FT-Linda extends Linda by providing support to allow a program to tolerate failures in the underlying computing platform. The distinguishing features of FT-Linda are stable tuple spaces and atomic execution of multiple tuple space operations. The former is a type of stable storage in which tuple values are guaranteed to persist across failures, while the latter allows collections of tuple operations to be executed in an all-or-nothing fashion despite failures and concurrency. Example FT-Linda programs are given for both dependable systems and parallel applications.

The design and implementation of FT-Linda are presented in detail. The key technique used is the replicated state machine approach to constructing fault-tolerant distributed programs. Here, tuple space is replicated to provide failure resilience, and the replicas are sent a message describing the atomic sequence of tuple space operations to perform. This strategy allows an efficient implementation in which only a single multicast message is needed for each atomic sequence of tuple space operations.

An implementation of FT-Linda for a network of workstations is also described. FT-Linda is being implemented using Consul, a communication substrate that supports fault-tolerant distributed programming. Consul is built in turn with the *x*-kernel, an operating system kernel that provides support for composing network protocols. Each of the components of the implementation has been built and tested.

CHAPTER 1

INTRODUCTION

Computers are being relied upon to assist humans in an ever-increasing number and variety of ways. Further, these applications have become increasingly sophisticated and powerful. To keep pace with the corresponding increased demand for faster computers, computer manufacturers have built parallel machines that allow subproblems to be solved simultaneously. Parallel programs running on these computers are now used to solve a wide variety of programs, including scientific computations, engineering design programs, financial transaction systems, and much more [Akl89, And91].

Parallel programming is much harder than sequential programming for a number of reasons. Coordinating the processors is difficult and error-prone. Also, there are different architectures that run parallel programs, and programs written on one cannot generally be run on another. Finally, one of the processors involved in the parallel computation could fail, leaving the others in an erroneous state.

Different abstractions and programming languages have been developed to help programmers cope with the difficulties of parallel programming. One such language is Linda, which provides a powerful abstraction for communication and synchronization called *tuple space*. However, Linda does not allow programmers to compensate for the failures of a subset of the computers involved in a given computation. This limits its usefulness for many long-running scientific computations and for *dependable systems*, which are ones that are relied upon to function properly with a very high probability. This dissertation addresses Linda's lack of provisions for dealing with failures by analyzing the failure modes of typical Linda applications and by proposing extensions that allow Linda programs to cope with such failures.

1.1 Motivation for Parallel Programming

Parallel programming involves writing programs that consist of multiple processes executing simultaneously. This activity includes subdividing the problem to be solved into subproblems, and then programming the sequential solution to each subproblem. The latter requires programming the ways in which the processes that solve the subproblems communicate and coordinate. This communication and coordination can be through physically shared memory or by message passing.

There are three main motivations for parallel programming: performance, economics, and dependability. First, while computers have been increasing in speed, the demands of computer users have kept up with these increases, and in some cases surpassed them.

In fact, for many important problems, even today's fastest uniprocessor supercomputer cannot execute as fast as its users would like and could profitably use. Examples of such problems include environmental and economic modelling, real-time speech and sight recognition, weather forecasting, molecular modelling, and aircraft testing [Akl89]. Parallel programming enables these problems to be solved more efficiently by performing multiple tasks simultaneously.

The second reason for parallel programming is economics: It is generally cheaper to build a parallel computer with the same or even more aggregate computational power than a fast uniprocessor. The former can be constructed of relatively inexpensive, off-the-shelf components, because each processor can be much slower than the uniprocessor's sole processor. The latter, however, must use low-volume components constructed from more exotic and expensive materials to achieve significantly better performance than can be achieved with a single off-the-shelf component [HP90].

The third reason for parallel programming involves dependability. A system or component is said to be *dependable* if reliance can justifiably be placed on the service it delivers [Lap91]. Computers are increasingly relied upon in situations where their failures could result in economic loss or even loss of life. For example, the 1990 failure of an AT&T long distance network was caused by the malfunction of a single computer [Neu92, Jac90]. In such applications, the dependability of computers is a vital concern. Architectures on which parallel programs are executed have redundancy inherent in their multiple processors, and thus they offer at least the potential to tolerate some failures while still providing the service or finishing the computation that has been assigned to them. Uniprocessors, on the other hand, have only one processor and thus suffer from a single point of failure, i.e., the failure of the processor can cause the failure of the entire machine.

1.2 Architectures for Parallel Programming

A number of different architectures for parallel programming have been developed. They vary mainly in the number of processors and the ways in which the processors can communicate with each other. We examine three categories in the spectrum of parallel computer architectures that together comprise the vast majority of parallel computers in use today: shared memory multiprocessors, distributed memory computers, and distributed systems.

The first architecture for parallel programming is the *shared memory multiprocessor*, which are sometimes called *closely coupled machines* or simply *multiprocessors*. These are parallel computers with multiple processors that share the same memory unit. The processes in a parallel program running on such a computer communicate by reading from and writing to a shared memory. This memory is fairly quick to access. Since processes on different CPUs communicate with each other with such low latency, *interprocess communication* (IPC) is very quick with these computers. However, the path to the shared memory, its bus, is a bottleneck if too many processors are added, and thus multiprocessors do not scale well.

The next architecture is the *distributed memory computer*, also known as the *multicomputer*. Here, each processor has fast, private memory. However, a processor may not access memory on another processor. Rather, processors communicate by sending messages over high-speed message passing hardware. This message passing hardware does not have a single bottleneck as a multiprocessor does, and thus multicomputers scale better than multiprocessors in the number of processors. However, a multicomputer's IPC latency is higher than a multiprocessor's.

The third architecture for parallel programming is the distributed system. A *distributed system* consists of multiple computers connected by a network. However, unlike the multiprocessor or the multicomputer, these computers are physically separated. Processes on different computers typically communicate with latencies on the order of a few milliseconds for computers in the same building and on the order of a few seconds or more for computers thousands of miles away. The latency for this IPC is much slower than a multicomputer's, but and the physical separation of the processors in a distributed system offers advantages for dependable computing. Examples of distributed systems include workstation clusters connected by a local area network (LAN), machines on the Internet cooperating to solve a problem, and the computers on an airplane used for controlling its various functions such as adjusting its control surfaces and plotting its course.

1.3 Simplifying Parallel Programming

Parallel programming is increasing in both popularity and diversity, but writing parallel programs is difficult for a number of reasons.First, parallel programs are simply more complicated; they have to deal with concurrency, something that sequential programs do not have to do [And91]. For example, a procedure performing an operation on a complex data structure that operates correctly on a uniprocessor will generally not function properly if multiple processes simultaneously call that procedure. A second reason that parallel programming is more difficult concerns coping with *partial failures*, i.e., the failure of part of the underlying computing platform. Some applications have to be able to tolerate partial failures and keep executing correctly. This makes programming more difficult; for example, the uniprocessor procedure mentioned above that operates on a complex data structure would likely leave that data structure in an erroneous state if it failed while executing in the middle of the procedure body. A final reason why writing parallel programs is more difficult regards *portability*. Programs that are written for one kind of processor or one kind of parallel architecture will not generally run on another. Portability is a serious problem given the high cost of developing software and the frequency at which new parallel computers become available.

Abstractions to simplify the task of the programmer fall in two main categories: fault tolerance and process coordination. Fault tolerance abstractions provide powerful models and mechanisms for the programmer to deal with failures in the underlying computing platform [MS92, Jal94, Cri91]. Process coordination mechanisms allow multiple processes to communicate and coordinate [And91]. These two categories overlap, but they differ so are described separately below.

1.3.1 Fault Tolerance Abstractions

Fault-tolerant abstractions allow a programmer to construct *fault-tolerant software*, i.e., software that can continue to provide service despite failures. These abstractions come in many forms and are structured so that each abstraction is built on lower-level ones. *Failure models* specify a component's acceptable behavior, especially the ways in which it may fail, i.e., deviate from its specified behavior. All other abstractions described below assume a given failure model; i.e., they will tolerate specific failures. *Fault-tolerant services* fall under two categories. One kind provides functionality similar to that which standard hardware or operating systems provide but with improved semantics in the presence of failures. The other kind of fault-tolerant service provides consistent information to all processes involved in a parallel computation. Finally, *fault tolerance structuring paradigms* are canonical program structuring techniques to help simplify the construction of fault tolerant software.

Failure Models

Failure models provide a way for reasoning about the behavior of components in the presence of failures by specifying assumptions about the effects of such failures. Failure models form a hierarchy, from weak to strong [MS92]: a given failure model permits all failures in all stronger models plus some additional ones. A weak failure assumption assumes little about the behavior of the components and is thus closer to the real situation with standard, off-the-shelf components. However, it is much more difficult for the programmer to use, because there are many ways (and combinations thereof) in which the components may fail. Indeed, one of the major challenges is to develop failure models and other abstractions that are powerful enough to be useful yet simple enough to allow an efficient implementation.

The weakest failure model is *Byzantine* or *arbitrary* [LSP82]. Here components may fail in arbitrary ways. The *timing* failure model assumes a component will respond to an input with the correct value, but not necessarily within a given timing specification [CASD85]. The *omission* failure model assumes that a component may fail to respond to an input [CASD85]. A stronger model yet is *crash* or *fail-silent*, where a component fails without making any incorrect state transitions [PSB$^+$88]. The strongest model is *fail-stop*, which adds to the crash model the assumption that the component fails in a way that is detectable by other components [SS83].

A system with a given failure model can be constructed from components with a weaker failure model. For example, [Cri91] gives an example of constructing a storage service that will tolerate one read omission failure from two elementary storage servers with read omission failure semantics. Also, the TCP/IP protocol is built on top of unreliable protocols, yet it provides reliable, ordered delivery of a sequence of bytes [Com88].

Fault-Tolerant Services

One category of fault-tolerant service provides functionality similar to standard hardware or operating systems, but with improved semantics in the presence of failures. This category includes stable storage, atomic actions, and resilient processes. The contents of a *stable storage* are guaranteed to survive failures [Lam81]. *Atomic actions* allow a number of computations to be seen as an indivisible unit by other processes despite concurrency and failure [Lam81]. A *resilient process* can be restarted and then continue to correctly execute even if the processor on which it is executing fails; techniques to construct resilient processes — for example, checkpointing — are discussed in [MS92].

The other category of fault-tolerant service provides consistent information to all processes involved in a parallel program, which is especially necessary in a distributed system with no memory or clock shared among the processors. The abstractions that these services offer are generally concerned with determining and preserving the *causal relation* among various events that occur on different processors in a distributed system [Lam78]. That is, if events a and b occur at times T_a and T_b, respectively, could a have affected the execution of b? This is simple to ascertain if both a and b occur on the same processor, because T_a and T_b were obtained from the same local clock. However, if they occurred on different machines in a distributed system, we cannot naively assume that the clocks are synchronized to reflect potential causality.

The fault-tolerant services that provide consistent information includes common global time, multicast, and membership services. *Common global time* provides a distributed clock service that maintains potential causality among events despite failures [RSB90]. For example, in the scenario above, if $T_a \geq T_b$, and both were read from the common global time service, then event a could not have affected the execution of b, because a did not happen before b. A *multicast* service delivers a message to each process in a group of processes such that the message delivered to each process in a consistent order relative to all other messages despite failures and concurrency [BJ87, CASD85, PBS89, GMS91]. This is very important in many different kinds of fault-tolerant programs, especially those constructed using the state machine approach described below. Finally, a *membership* service provides processes in a group consistent information about the set of functioning processors at any given time [VM90].

Fault Tolerance Structuring Paradigms

Fault tolerance structuring paradigms are canonical program structuring techniques that have been developed in conjunction with the above abstractions and services to help programmers structure fault-tolerant distributed programs. As such, each paradigm applies to a number and variety of different applications. This greatly reduces the complexity of developing such programs.

Three major paradigms are object/action, primary/backup, and state machines. The object/action paradigm involves passive objects that export actions, which are operations to modify the long-lived state of the object [Gra86]. These actions are *serializable*, which means that the effect of any concurrent execution of actions on the same object is equivalent to some serial sequence. They are also *recoverable*, i.e. executed completely or not at all. These actions are called *transactions* in the database context.

The *primary/backup* paradigm features a service implemented by multiple processes; the primary process is active, while the backup processes are passive [AD76, BMST92]. Only the active process responds to requests for the service; in the case of the failure of the primary process, one backup process will become the primary, starting from a checkpointed state.

A *replicated state machine* consists of a collection of servers [Sch90]. Each client request is multicast to each server replica, where it is processed deterministically. Because each replica processes each request, the state machine approach is sometimes called *active replication*, while the primary/backup paradigm is called *passive replication*. The state machine approach is described in more detail in Section 2.4.

1.3.2 Process Coordination Mechanisms

Fault-tolerance abstractions help a programmer write parallel programs that can cope with failures; process coordination mechanisms provide the programmer with techniques that allow different processes to communicate and synchronize. Process coordination mechanisms include remote procedure call, message passing, shared memories, and coordination languages. These mechanisms may be provided to the programmer either through an explicitly parallel language or through libraries of procedures for a sequential language. Parallel programming languages provide high-level abstractions for programmers to use. These generally permit a cleaner integration of the sequential and concurrent features of the language than do libraries of procedures. Examples of such languages include Ada [DoD83], Orca [Bal90], and SR [AO93]. Libraries of procedures allow the programmer to leverage experience with an existing language and reuse existing code. They can also support many languages, allowing processes written with different languages to communicate.

Remote procedure call (RPC) [Nel81, BN84] is like a normal procedure call, except the invocation statement and the procedure body are executed by two different processes, potentially on different machines. RPC is a fundamental building block for distributed systems today, including client/server interactions and many distributed operating systems such as Amoeba [TM81]. It is also supported in many distributed programming languages, including Aeolus [WL86], Argus [LS83], Avalon [HW87], and Emerald [BHJL86]. *Message passing* allows processes on different computers to exchange messages to communicate and synchronize. PVM [Sun90] and MPI [For93b, For93a] are two of the more well-known libraries of message passing libraries; many parallel programming languages explicitly provide message passing as well [And91].

A *shared memory* is memory that can be accessed by more than one process. It can be implemented in a multiprocessor by simply providing a shared memory bus that connects all processors to the memory. A *distributed shared memory* (DSM) provides the illusion of a physically shared memory that is available to all processes in a distributed system [NL91]. The regions of the DSM are either replicated on all computers or are passed among them as needed; this is done by the language's implementation and is transparent to the programmer. DSMs thus allow distributed systems and multicomputers to be programmed much more like a multiprocessor. Munin is an example of a system that implements a DSM [CBZ91].

Coordination languages augment an existing computational language such as FORTRAN or C to allow different processes to communicate in a uniform fashion with each other and with their environment [GC92]. A coordination language is completely orthogonal to the computational language. Coordination languages thus provide in a uniform fashion many of the services such as process-process and process-environment communication typically provided by an operating system. A coordination language can thus be used by a process to perform I/O, communicate with a user, or communicate directly with another process, regardless of the location and lifetime of that process.

1.4 Linda

Linda is a coordination language that provides a collection of primitives for process creation and interprocess communication that can be added to existing languages [Gel85, CG89]. The main abstraction provided by Linda is *tuple space* (TS), an associative shared memory. The abstraction is implemented by the Linda implementation transparently to user processes.

The tuple space primitives provided by Linda allow processes to deposit tuples into TS and to withdraw or read tuples whose contents match a pattern. Thus, tuple space provides for *associative access*, in which information is retrieved by content rather than address. Tuple space also provides *temporal decoupling*, because communicating processes do not have to exist at the same time, and *spatial decoupling*, because processes do not have to know each other's identity. These properties make Linda especially easy for use by application programmers [CS93, ACG86, CG88, CG90]. Linda implementations are available on a number of different architectures [CG86, Lei89, Bjo92, SBA93, CG93, Zen90] and for a number of different languages [Lei89, Jel90, Has92, Cia93, SC91]. Linda has been used in many real-world applications, including VLSI design, oil exploration, pharmaceutical research, fluid-flow systems, and stock purchase and analysis programs [CS93].

Despite these advantages, one significant deficiency of Linda is failures can cause Linda programs to fail. For example, the semantics of tuple space primitives are not well-defined should a processor crash, nor are features provided that allow programmers to deal with the effect of such a failure. The impact of these omissions has been twofold. First, programmers who use Linda to write general parallel applications cannot take advantage of fault-tolerance techniques, for example, to recover a long-running scientific application after a failure. Second, despite the language's overall appeal, it has generally been impossible to use Linda to write critical applications such as process control or telephone switching in which dealing with failures is crucial.

1.5 FT-Linda

In this dissertation, we describe FT-Linda, a version of Linda that provides support for programming fault-tolerant parallel applications. To do this, FT-Linda includes two major enhancements: *stable tuple spaces* and support for *atomic execution* of TS operations. The former is a type of stable storage in which the contents are guaranteed to persist across failures; the latter allows collections of tuple operations to be executed in an all-or-nothing fashion despite failures and concurrency. Also, we provide additional Linda primitives to allow tuples to be transferred between tuple spaces atomically. The specific design of these enhancements has been based on examining common program structuring paradigms — both those used for Linda and those found in fault-tolerant applications — to determine what features are needed in each situation. In addition, the design is in keeping with the "minimalist" philosophy of Linda and permits an efficient implementation. These features help distinguish FT-Linda from other efforts aimed at introducing fault-tolerance into Linda [Xu88, XL89, Kam90, AS91, Kam91, CD94, CKM92, PTHR93, JS94].

FT-Linda is being implemented using Consul, a communication substrate for building fault-tolerant systems [Mis92, MPS93b, MPS93a], and the *x*-kernel, an operating system kernel that provides support for composing network protocols [HP91]. Stable TSs are realized using the replicated state machine approach, where tuples are replicated on multiple processors to provide failure resilience and then updated using atomic multicast. Atomic execution of multiple tuple space operations is achieved by using a single multicast message to update replicas. The focus in the implementation has been on tolerating processor crash failures, though the language design is general and could be implemented assuming other failure models. Further, the current implementation focuses on distributed systems, because their inherent redundancy and physical separation make them attractive candidates for dependable computing.

The major contributions of this dissertation are:

- An analysis of the failure modes of typical Linda programs and of Linda's *atomicity requirements* — at what level its operations should be realized in an "all or nothing" fashion despite failures and concurrency.

- Extensions to Linda to help tolerate failures, namely stable tuple spaces and atomic execution of multiple tuple space operations.

- An efficient implementation design for the resulting language, FT-Linda. An atomic sequence of tuple space operations only requires one multicast message, and FT-

> Linda's costs of processing an operation is comparable to an optimized Linda implementation.

Other contributions include better semantics for Linda, even in the absence of failures; multiple tuple spaces with attributes that allow different kinds of tuple spaces to be created; primitives to allow the transfer of tuples between different tuple spaces; and disjunction, which allows one from a number of matching tuples to be selected.

1.6 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes Linda in detail and discusses its failure modes. Alternatives for extending Linda to deal with failures are described; this also serves as a motivation for the next chapter and further delineates the contributions of this dissertation.

Chapter 3 describes FT-Linda, our version of Linda to help it tolerate failures. It describes in detail FT-Linda's stable tuple spaces and atomic execution of multiple tuple space operations. The way these features can be used to help tolerate failures is illustrated using a series of examples.

Chapter 4 gives more examples of how both applications and system programs written in FT-Linda can tolerate failures. It then shows how FT-Linda programs can handle the failure of the main or master process.

Chapter 5 describes the implementation of FT-Linda. First it describes the compiler, and then how the runtime system is implemented on top of Consul and the *x*-kernel. It then describes initial performance results, optimizations, and future extensions to the implementation.

Chapter 6 offers some concluding remarks and future research directions for FT-Linda.

CHAPTER 2
LINDA AND FAILURES

Linda is a coordination language that is simple to use yet powerful. However, problems with its semantics and its lack of facilities to handle failures makes it inappropriate for long-running scientific applications and dependable systems, domains for which it would otherwise be suitable [WL88, CGM92, PTHR93, BLL94].

This chapter provides the background information and motivation for our extensions to Linda. First, it gives an overview of Linda. Next, it discusses semantic limitations of Linda that are a problem even in the absence of failures. It then examines the problems Linda programs have in the presence of failures, followed by design alternatives for providing fault-tolerance to Linda; this motivates our particular design choices. These design choices are tightly intertwined with the particular language extensions we offer and with the way in which we implement them, topics for Chapters 3 and 5, respectively.

2.1 Linda Overview

Linda is a parallel programming language based on a communication abstraction known as *tuple space* (TS) [Gel85, ACG86, CG89, GC92, CG90]. Tuple space is an associative (i.e., content-addressable) unordered bag of data elements called *tuples*. Processes are created in the context of a given TS, which they use as a means for communicating and synchronizing. A tuple consists of a *logical name* and zero or more values. An example of a tuple is (“N”, 100,**true**). Here, the tuple consists of the logical name N and the data values 100 and **true**. Tuples are immutable, i.e., they cannot be changed once placed in the TS by a process.

The basic operations defined on a TS are the deposit and withdrawal of tuples. The **out** operation deposits a tuple. For example, executing

out (“N”, 100,**true**)

causes the tuple (“N”, 100,**true**) to be deposited. As another example, if i equals 100 and $boolvar$ equals **true**, then the operation

out (“N”, $i, boolvar$)

will also cause the same tuple, (“N”, 100,**true**), to be deposited. An **out** operation is $asynchronous$, i.e. the process performing the **out** need only wait for the arguments to be evaluated and marshalled; it does not have to block until the corresponding tuple is actually deposited into TS.

The **in** operation withdraws a tuple matching the specified parameters, which collectively are known as the *template* of the operation. To match a tuple, the **in** must have the same number of parameters, and each parameter must match the corresponding value in the tuple. The parameters to **in** can either be actuals or formals. An actual is a value that can be specified as either a literal value or with a variable's name; in the latter the value of the variable will be used as the actual. To match a value in a tuple, the value of an actual in a template must be of the same type and value as the corresponding value in the tuple.

A formal is either a variable or type name, and is distinguished from an actual by the use of a question mark as the first character. Formals automatically match any value of the same type in a tuple. If the formal is a variable name, then the operation assigns the corresponding value from the tuple to the variable. This is the mechanism through which different processes in a Linda program receive data from other processes. For example, if i is declared as an integer variable and b as a boolean, then executing

in(“N”, $?i, ?b$)

will withdraw a tuple named N that has an integer as its first argument and a boolean as its second (and last) one, and will assign to i and b the respective values from the tuple. If there is no such tuple, then the process will block until one is present. As another example, executing

in(“N”, 100,**true**)

will withdraw a tuple named N whose second argument is an integer with value 100 and whose third argument is a boolean with value **true**. Here, the parameters of **in** are all actuals, so executing this operation does not give the program new data, like it would if it had formal variables. Such an operation may, however, be useful for synchronization.

Formals are legal in **out** operations, but such usage is rare in practice. In this case, because an **out** does not withdraw or inspect a tuple, the formal variable is not assigned a value. Rather, a valueless type is placed in the appropriate field in the tuple; it must be matched by an actual, not a formal, in the template for the corresponding Linda operation that withdraws or inspects the tuple.

The Linda **rd** operation is like **in** except it does not withdraw the tuple. That is, it waits for a matching tuple and assigns values from the tuple to formal variables in its template. Unlike **in**, however, it leaves the matching tuple in TS.

Operations **rdp** and **inp** are non-blocking versions of **rd** and **in**, respectively. If an appropriate tuple is found when one of these operations is executed, **true** is returned as the functional value and and the operation behaves like its non-blocking counterpart; if no such tuple is found, **false** is returned and the values of any formal parameters remain unchanged.

The final Linda primitive is **eval**, which is used for process creation. In particular, invoking

$$\texttt{eval}\,(\text{“}func\text{”}, func(args))$$

will create a process to execute $func$ with $args$ as parameters, and also deposit an *active tuple* in TS. This tuple may not be matched by any TS operation, but when $func$ returns the tuple will become a normal “inactive” tuple containing the return value of $func$.

Few applications seem to need the coupling of process creation and tuple depositing that **eval** offers, and it has a number of semantic and implementation problems [Lei89]. As a result, it is not examined further in this dissertation; FT-Linda provides an alternative mechanism for process creation.

Finally, TS may outlive any process in the program and may even persist after termination of the program that created it. Thus, Linda allows *temporal uncoupling* because two processes that communicate do not have to have overlapping lifetimes. Linda also allows *spatial uncoupling* because processes need not know the identity of processes with which they communicate.[1] These properties allow Linda programs to be powerful, yet simple and flexible.

2.2 Problems with Linda Semantics

While Linda has many advantages, it is not without its deficiencies. Consider the boolean operations **inp** and **rdp**. When they return **false**, they are making a very strong statement about the global state of TS: they are indicating that no matching tuple exists anywhere in TS. This has historically been considered too expensive to implement for a distributed system or on a multicomputer [Lei89, Bjo92]. Thus, implementations of Linda for such systems generally do not offer **inp** and **rdp** or they offer a weaker *best effort* semantics. Here, **inp** and **rdp** are not guaranteed to find a matching tuple; they may return **false** even if a matching tuple exists in TS.

Also, because **out** is asynchronous, there is no guarantee as to when the tuple it specifies will be deposited into TS. This, coupled with a best effort semantics for **inp** and **rdp**, makes some Linda code deceptive. Consider the following example from [Lei89, Bjo92], which is used to argue that **inp** and **rdp** should not be supported:

[1] If they do need to know, a process identifier or handle can be included in the tuple. See Section 4.1.2 for an example.

```
process P1     process P2
  out("A")       in("B")
  out("B")       if (inp("A")) then
                   print (``Must succeed!'')
```

The problem is a naive programmer may believe that this code is synchronized properly so that the **inp** must succeed. However, this is not the case with either asynchronous **out**s or with best effort semantics for **inp**, or both. With asynchronous **out**s, the tuple "B" may be deposited in TS before "A", so "A" is not yet present in TS when the **inp** checks for it. And with best effort semantics for **inp**, **inp** is not guaranteed to find the tuple "A", even if it is present in TS.

Note that these semantic problems occur even in the absence of failures. They are discussed further in Section 3.3, where alternatives are provided.

2.3 Problems with Failures

Linda programs also have problems in the presence of failures. In particular, as noted in Chapter 1, the effects of processor failures, i. e., crash failures, on TS is not considered in standard definitions of the language or most implementations. In examining how Linda is currently used to program parallel applications and would likely be used for fault-tolerant applications, two fundamental deficiencies are apparent. The first is lack of *tuple stability*. That is, the language contains no provisions for guaranteeing that tuples will remain available following a processor failure. Given that tuple space is the means by which processes communicate and synchronize, it is easy to imagine the problems that would be caused if certain key tuples are lost or corrupted by failure. Moreover, a stable storage facility is a key requirement for the use of many fault-tolerance techniques. For example, *checkpoint and recovery* is a technique based on saving key values in stable storage so that an application process can recover to some intermediate state following a failure [KT87]. This technique cannot be implemented with Linda in the absence of tuple stability, given that TS is the only means for interprocess communication in a Linda program.

The second deficiency can be characterized as *lack of sufficient atomicity*. Informally, a computation that modifies shared state is atomic if, from the perspective of other computations, all its modifications appear to take place instantaneously despite concurrent access and failures. In Linda, of course, the shared state is the TS and the computations in question are TS operations. The key here is that Linda provides only *single-op atomicity*, i.e., atomic execution for only a single TS operation. Thus, the intermediate states resulting from a series of TS operations may be visible to other processes.

Providing *multi-op atomicity*, a means to execute multiple TS operations atomically, is important for using Linda to program fault-tolerant applications. For example, *distributed consensus*, in which multiple processes in a distributed system reach agreement on some common value, is an important building block for many fault-tolerant systems [TS92].

```
Initialization
    out("count", value)

Inspection
    rd("count", ?value)

Updating
    in("count", ?oldvalue)
    out("count", newvalue)
```

Figure 2.1: Distributed Variables with Linda

However, Linda with single-op atomicity has been shown to be insufficient to reach distributed consensus with more than two processes in the presence of failures or with arbitrarily slow (or busy) processors [Seg93]. The key is lack of sufficient (multi-op) atomicity.

Even typical Linda programs cannot be structured to handle failures with only single-op atomicity. To illustrate this, we consider specific problems that arise in two common Linda programming paradigms: the distributed variable and the bag-of-tasks. Both these paradigms are used to solve a wide variety of problems, meaning that deficiencies in these two paradigms are applicable to a large class of Linda programs.

Distributed Variable

The simplest paradigm is the distributed variable. Here, one tuple containing the name and value of the variable is kept in TS. Figure 2.1 shows how typical operations on such a variable named $count$ might be implemented. The first operation initializes the variable $count$ to $value$. To inspect the value of $count$, **rd** is used as shown. Finally, updating the value involves withdrawing the tuple and depositing a new tuple with the new value. Note that the tuple must be withdrawn with **in** and not just read with **rd** to guarantee mutually exclusive access to the variable and uniqueness of the resulting tuple.

Unfortunately, if the possibility of a processor crash is considered, the protocol has a window of vulnerability. Specifically, if the processor executing the process in question fails after withdrawing the old tuple but before replacing it with the new one, that tuple will be lost because it is present only in the volatile memory of the failed processor. We call this problem the *lost tuple* problem. The result is that processes attempting to inspect or modify the distributed variable will block forever. The problem is due to the inability to execute the **in** and subsequent **out** as an atomic unit with respect to failures.

```
process worker
   while true do
      in("$subtask$", ?$subtask\_args$)
      calc($subtask\_args$, var $result\_args$)
      for ($all\ new\ subtasks\ created\ by\ this\ subtask$)
         out("$subtask$", $new\_subtask\_args$)
      out("$result$", $result\_args$)
   end while
end proc
```

Figure 2.2: Bag-of-Tasks Worker

Bag-of-Tasks

Linda lends itself nicely to a method of parallel programming called the bag-of-tasks or replicated worker programming paradigm [ACG86, CG88]. In this paradigm, the task to be solved is partitioned into independent subtasks. These subtasks are placed in a shared data structure called a *bag*, and each process in a pool of identical workers then repeatedly retrieves a subtask description from the bag, solves it, and outputs the solution. In solving it, the process may use only the subtask arguments and possibly non-varying global data, which means that the same answer will be computed regardless of the processor that computes it and the time at which it is computed. Among the advantages of this programming approach are transparent scalability, automatic load balancing, and ease of utilizing idle workstation cycles [GK92, CGKW93, Kam94]. And, as we show in Chapter 3, it can easily be extended to tolerate failures.

Realizing this approach in Linda is done by having the TS function as the bag. The TS is seeded with subtask tuples, where each such tuple contains arguments that describe the given subtask to be solved. The collection of subtask tuples can thus be viewed as describing the entire problem.

The actions taken by a generic worker are shown in Figure 2.2. The initial step is to withdraw a tuple describing the subtask to be performed; the logical name "$subtask$" is used as a distinguishing label to identify tuples containing subtask arguments. The worker computes the results, which are subsequently output to TS with the identifying label "$result$". Also, any new subtask tuples that this subtask generates are placed into TS (this would actually be done in the procedure *calc*, but is shown outside the procedure for clarity). If the computation portion of any worker in the program generates new subtask tuples, then we say that the solution uses a *dynamic* bag-of-tasks structure. If no new subtask tuples are generated, then we call the solution a *static* bag-of-tasks structure; in this case, a master process is assumed to subdivide the problem and seed the TS with all appropriate subtask tuples.

Termination detection—ascertaining that all subtasks have been computed—can be accomplished by any one of a number of techniques, including worker deadlock or by keeping a count of the number of subtasks that have been computed. The actual way in which the program is terminated is irrelevant for our purposes, and so is ignored hereafter.

The bag-of-tasks paradigm suffers from two problems when failures are considered. The first is again the lost tuple problem. Specifically, if the processor fails after a worker has withdrawn the subtask tuple but before depositing the result tuple, that result will never be computed. This is because, during this period, the only representation of that tuple is in the (volatile) memory of the processor. When the processor fails, then, that subtask tuple is irretrievably lost.

The second problem is a somewhat different problem, which we call the *duplicate tuple* problem. This problem occurs if the processor fails after the worker has generated some new subtasks but before it has deposited the result tuple. In this case, assuming the lost tuple problem is solved, another worker will later process the subtask, generating the same new subtasks that are already in TS. Such an occurrence can lead to the program producing incorrect results—for example, the process that consumes the result tuples may expect a fixed number of result tuples—in addition to wasted computation. The cause of the problem is again lack of sufficient atomicity. What is needed in this case is some way to deposit all the new subtask tuples and the result tuple into TS in one atomic step.

2.4 Implementing Stability and Atomicity

Given the problems identified above, the challenge is to develop reasonable approaches to implementing stable TSs and atomic execution within Linda. For stable TSs, choices range from using hardware devices that approximate the failure-free characteristics of stable storage (e.g., disks) to replicating the values in the volatile memory of multiple processors so that failure of (some number of) processors can be tolerated without losing values. In situations where stable values must also be shared among multiple processors as is the case here, replication is a more appropriate choice.

To realize a replicated TS, we use the replicated state machine approach (SMA) introduced in Chapter 1. In this technique, an application is implemented as a state machine that maintains *state variables*, which encode the application's state. These variables are modified by *commands* that the state machine exports; a client of the state machine sends it a *request* to execute a command. To provide resilience to failures, the state machine is replicated on multiple independent processors, and an ordered *atomic multicast*, also mentioned in Chapter 1, is used to deliver commands to all replicas reliably and in the same order.[2] For commands that require a reply, one or more state machine replicas will send a reply message to the client. If the commands are deterministic and executed atomically with respect to concurrent access, then the state variables of each replica will remain consistent. The SMA is the basis for a large number of fault-tolerant distributed systems [BSS91, MPS93a, Pow91].

[2] This ordering can be relaxed in some cases; see [Sch90].

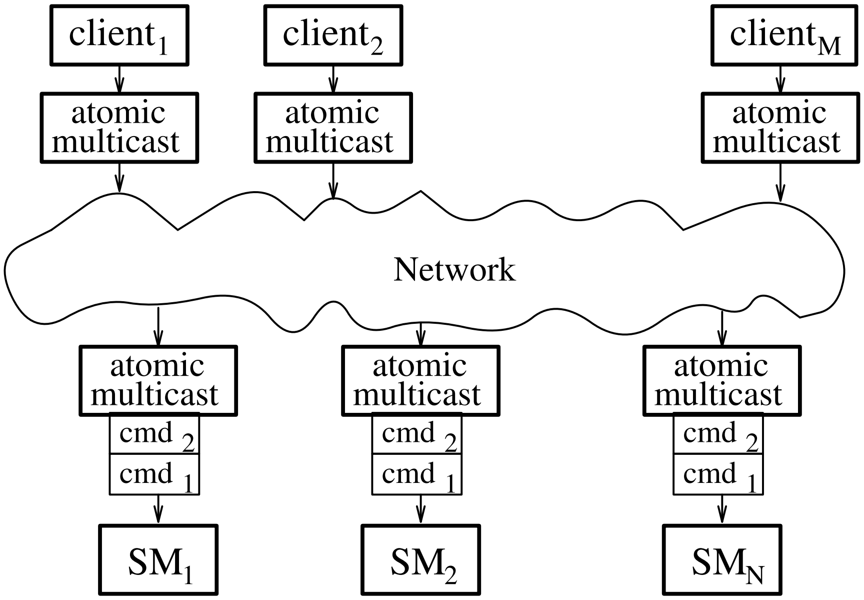

Figure 2.3: State Machine Approach

The SMA is illustrated in Figure 2.3. Here the M clients use atomic multicast to submit commands to the N replicas of the state machine. This ensures that the state machines receive the same sequence of command messages, even in the presence of failures and concurrency.

To use the SMA to provide tuple stability for Linda, then, each state machine replica maintains an identical copy of TS. These copies, which we call *tuple space replicas*, will be kept consistent with each other, because each replica receives the same sequence of TS operations and processes them in the same, deterministic manner. Having multiple identical copies of TS thus allows the failure of some of these copies to be tolerated while still preserving the TS abstraction at higher levels.

Given the use of replication to realize stable TSs, the next step is to consider schemes for implementing atomic execution of multiple tuple operations that use this TS. Such a scheme must guarantee, in effect, that either all or none of the TS operations are executed at either all or none of the functioning processors hosting copies of the tuple space. Additionally, other processes must not be allowed concurrent access to TS while an update is in progress.

A number of schemes would satisfy these requirements. For example, techniques based on the two-phase commit protocol for implementing general database transactions could be used [Gra78, Lam81]. While sufficient, these techniques are expensive, requiring multiple rounds of message passing between the processors hosting replicas. At least part of the reason for the heavyweight nature of the technique is that it supports atomic execution of essentially arbitrary computations. While important in a database context, such a facility is stronger than necessary in our situation where only simple sequences of tuple operations require atomicity. Accordingly, a simpler scheme is desired even if it provides a less general solution.

We believe a good compromise is to exploit the characteristics of the replicated state machine approach to implement atomicity. Recall that each command in this scheme is executed atomically; that is, a command is considered a single unit that is either applied as a whole or not at all, is applied at either all functioning processors or none (as guaranteed by the atomic multicast), and is executed without interleaving by each replica. Given this, a simple scheme for atomically executing multiple TS operations is to treat the entire sequence as, in essence, a single state machine command. The operations are disseminated together to all replicas in a single multicast message. This is then executed at each TS replica by its *TS manager*, the process that operates on the TS replica, as dictated by the serial order in which commands are processed by each TS manager. This technique has the virtue of being simple to implement and requires fewer messages than the transactional approach,[3] while still supporting the level of atomicity needed to realize fault tolerance in many Linda applications.

This broad outline of an implementation strategy serves not only to describe alternatives, but also to motivate the specific design of our extensions to Linda for achieving fault tolerance. The extensions and implementation are perhaps more sophisticated than implied by this discussion—for example, provisions for synchronization and a limited form of more general atomic execution are also provided—yet the overall design philosophy follows the two precepts inherent in the above discussion. First, the extensions must provide enough functionality to allow convenient programming of fault-tolerant applications in Linda. Second, the execution cost must be kept to a minimum. The trick has been to balance the often conflicting demands of these two goals, while still providing mechanisms that preserve the design philosophy and semantic integrity established by the original designers of Linda.

2.5 Summary

This chapter describes Linda, a coordination language that offers both simplicity and power. Its features are described, and problems with its semantics and with failures are then discussed. Alternatives for providing fault-tolernace to Linda are outlined, and our design based on the replicated state machine approach is described.

Linda's main abstraction is tuple space, an associative, unordered bag of tuples. Linda provides primitives to deposit tuples into tuple space as well as to withdraw and inspect tuples that match a specified template. These primitives can be used to allow the processes in a parallel program to communicate and synchronize in a simple yet powerful manner.

[3] This comparison assumes that the transaction's data or logs are replicated. If not, then the transactional approach may require fewer message (zero to FT-Linda's one). However, in this case the fault-tolerance guarantees of the transaction will be far weaker than FT-Linda's, because the failure of a single device or computer can halt the entire transactional system.

Linda's semantics are lacking, however, even in the absence of failures. Its best effort semantics means that the boolean primitives are not guaranteed to find a matching tuple even though one exists. And Linda's asynchronous **out**s mean that tuples generated by a single process may appear in tuple space in a different order than specified by the process.

Linda programs also have problems in the presence of failures. It lacks tuple stability, meaning that tuples are not guaranteed to survive a processor failure. Linda's single-op atomicity is also inadequate for handling failures. With many common Linda paradigms, a tuple is withdrawn and represented only in the volatile memory of a process. If this process fails, the tuple will be irretrievably lost.

There are a number of ways in which stability and multi-op atomicity could be provided to Linda. Stability could be provided by using a stable storage or by replicating the values on multiple processors. For FT-Linda, we use the latter approach, the replicated state machine approach. Multi-op atomicity could be provided by either adding transactions to Linda or by allowing a (less general) sequence of TS operations to be executed by replicated TS managers. We use the latter approach, which is also the replicated state machine approach.

CHAPTER 3
FT-LINDA

FT-Linda is a variant of Linda designed to facilitate the construction of fault-tolerant applications by including features for tuple stability, multi-op atomicity, and strong semantics [BS91, BS94, SBT94]. The system model assumed by the language consists of a collection of processors connected by a network with no physically shared memory. For example, it could be a distributed system in which processors are connected by a local-area network, or a multicomputer such as a hypercube with a faster and more sophisticated interconnect. FT-Linda could be implemented on a multiprocessor, but the single points of failure that multiprocessors have would make it largely pointless.

Processors are assumed to suffer only fail-silent failures, in which execution halts without undergoing any incorrect state transitions or generating spurious messages. The FT-Linda runtime system, in turn, converts such failures into *fail-stop failures* [SS83] by providing failure notification in the form of a distinguished *failure tuple* that gets deposited into TS. We also assume that processors remain failed for the duration of the computation and are not reintegrated back into the system;[1] extensions to allow such reintegration are considered in Section 5.8.1.

To support the main additions to the languages, FT-Linda has provisions for creating processes and for allocating unique system-wide process identifiers. A process is created using the $create$ function, which takes as arguments the name of the function to be started as a process, the host machine on which to start it, a *logical process ID* (LPID), and initialization arguments. There must be a 1:1 correspondence between processes and LPIDs at any given time, but different processes may assume the same LPID at different times. A unique LPID is typically generated prior to invoking $create$ by using the new_lpid routine. A process can determine its LPID using the routine my_lpid.

The remainder of this chapter is organized as follows. First, it discusses FT-Linda's provisions for stable tuple spaces and atomic execution of multiple tuple space operations. It then gives the semantics of FT-Linda's tuple space are then given. Finally, it describes possible extensions to FT-Linda.

3.1 Stable Tuple Spaces

To address the problem of data stability, FT-Linda includes the ability to define a *stable tuple space*. However, not wanting to mandate that every application use such a TS

[1] Note that the *computation* that was running on the failed processor can—and often will—be recovered using another physical processor.

given its inherent implementation overhead, this abstraction is included as part of more encompassing provisions. Specifically, FT-Linda allows the programmer to create and use an arbitrary number of TSs with varying attributes. The programmer specifies these attributes when creating the TSs, and a TS *handle* is returned to facilitate access to that TS. This TS handle is a first-class object that is subsequently passed as the first argument to other TS primitives such as **in** and **out**, placed into tuples, etc.

FT-Linda currently supports two attributes for tuple spaces: *resilience* and *scope*.[2] The resilience attribute, either *stable* or *volatile*, specifies the behavior of the TS in the presence of failures. In particular, tuples in a stable TS will survive processor failures, while those in a volatile TS have no such guarantee. The number of processor failures that can be tolerated by a stable TS without loss or corruption of tuples depends on the number of copies maintained by the implementation, a parameter specified at system configuration time. Given N such copies, tuples will survive given no more than $N-1$ failures, assuming no network partitions.[3]

The scope attribute, either *shared* or *private*, indicates which processes may access a given TS. A shared TS can be used by any process; such a TS is analogous to the single TS in current versions of Linda. A private TS, on the other hand, may be used only by the single logical process whose LPID is specified as an argument in the TS creation primitive (described below). As already noted, a process can only have a single LPID at a time, and only one process in the system at a time can have a given LPID.

Allowing access to private TSs based on the notion of a logical process allows the work of a process that has failed to be taken over by another newly-created process. To do this, the failure is first detected by waiting for the failure tuple associated with the processor on which it was executing. At this point, a new process is created with the same LPID. Once this is done, the new process can use any of the private TSs that were being used by the failed process, assuming, of course, that they were also declared to be stable. Such a scenario is demonstrated in Section 4.1.2.

A single stable shared TS is created when the program is started, and can be accessed using the handle $TSmain$. Other tuple spaces are created using the FT-Linda primitive ts_create. This function takes the resilience and scope attributes as arguments, and returns a TS handle, A third argument, the LPID of the logical process that can access the TS, is required in the case of private TSs. To destroy a TS, the primitive $ts\_destroy$ is called with the appropriate handle as argument. Subsequent attempts to use the handle result in an exceptional condition.

As noted in Chapter 2, stability is implemented by replicating tuples on multiple machines. As a result, a TS created with the stable attribute, whether shared or private, is also called a *replicated tuple space*. Conversely, a TS created with attributes volatile and private is called a *local tuple space*, because its tuples are only stored on the processor on which the TS was created, or it is called a *scratch tuple space*, because it is often used to

[2] Other possible attributes are discussed in Section 3.5.1

[3] Section 5.8.3 discusses handling network partitions.

hold intermediate results that will either be discarded or merged with a shared TS using primitives described below.

Different types of TSs have different uses. For example, a stable and private TS can be used by a process such as a server that must have some of its state survive failure so that it can be reincarnated. Replication is necessary for this type of TS, even though it is not shared, because it must be stable. An example of this use is given in Section 4.1.2. A scratch TS, on the other hand, need not be replicated and thus can have very fast access times. Additionally such a TS can be used in conjunction with the provisions for atomic execution of TS operations introduced shortly to provide duplicate atomicity.

3.2 Features for Atomic Execution

Two features are provided in FT-Linda to support atomic execution: *atomic guarded statements* and *atomic tuple transfer primitives*. Atomic guarded statements are used to execute sequences of TS operations atomically, potentially after blocking; the atomic tuple transfer primitives **move** and **copy** allow collections of tuples to be moved or copied between tuple spaces atomically. Each is addressed in turn below.

3.2.1 Atomic Guarded Statement

An atomic guarded statement (AGS) provides all-or-nothing execution of multiple tuple operations despite failures or concurrent access to TS by other processes.

Simple Case

The simplest case of the AGS is

$$\langle \; guard \;\Rightarrow\; body \; \rangle$$

where the angle brackets are used to denote atomic execution. The $guard$ can be any of **in**, **inp**, **rd**, **rdp**, or **true**, while the $body$ is a series of **in**, **rd**, **out**, **move** and **copy** operations, or a null body denoted by **skip**. The process executing an AGS is blocked until the guard either *succeeds* or *fails*, as defined below. If it succeeds, the body is then executed in such a way that the guard and body are an atomic unit; if it fails, the body is not executed. In either case, execution continues with the next statement after the AGS.

Informally, a guard succeeds if either a matching tuple is found or the value **true** is returned. The specifics are as follows. A **true** guard succeeds immediately. A guard of **in** or **rd** succeeds once there is a matching tuple in the named TS, which may happen immediately, at some time in the future, or never. A guard of **inp** or **rdp** succeeds if there is a matching tuple in TS when execution of the AGS begins. Conversely, a guard fails if the guard is an **inp** or **rdp** and there is no matching tuple in TS when the AGS is executed. A boolean operation used as a guard may be preceded by **not**, which inverts the success

Linda op	FT-Linda equivalent
out$(\ldots)$	$\langle$ **true** $\Rightarrow$ **out**$(\ldots)$ $\rangle$
$other\_op(\ldots)$	$\langle$ $other\_op(\ldots) \Rightarrow$ **skip** $\rangle$

Table 3.1: Linda Ops and their FT-Linda Equivalents

semantics for the guard in the expected way. Note that in this case, execution of an AGS may have an effect even though the guard fails and the body is not executed; for example, if the failing guard is "**not inp**$(\ldots)$", a matching tuple still gets withdrawn from TS and any formals assigned their corresponding values. However, the body will not be executed because the guard failed.

An atomic guarded statement with boolean guards can also be used within an expression. The value of the statement in this case is taken to be **true** if the guard succeeds and **false** otherwise. This facility can be used, for example, within the boolean of a loop or conditional statement to control execution flow.

Only one operation—the guard—is allowed to block in an AGS. Thus, if an **in** or **rd** in the body does not find a matching tuple in TS, an exceptional condition is declared and the program is aborted. The implementation strategy also leads to a few other restrictions on what can be done in the body, most of which involve data flow between local and replicated TSs. These restrictions are explained further in Section 5.5.1.

Finally, our implementation strategy dictates that Linda TS operations not appear outside of an AGS. Table 3.1 gives the FT-Linda equivalent of standard Linda TS operations; in it, $other\_op$ may be **in**, **inp**, **rd**, or **rdp**. It would be easy to implement a preprocessor to translate the standard Linda operations into these equivalents if desired. For convenience, we use the standard Linda notation for single TS operations below.

Using Atomic Guarded Statements

The AGS can be used to solve atomicity problems of the sort demonstrated earlier in Chapter 2. For example, consider the lost tuple problem that occurs in the bag-of-tasks paradigm when a failure interrupts execution after a subtask tuple has been withdrawn but before the result tuple is deposited. Recall that the essence of the problem was that there was a window of vulnerability where the subtask was not represented in some form in TS. Specifically, it was not in TS in some form after the worker withdrew the subtask tuple and before it deposited the corresponding result tuple. We remove this window of vulnerability by maintaining a version of the subtask tuple in TS while the subtask is being solved. This will ensure that the subtask is represented in TS in some form at all times, and thus it can be recovered if the worker process fails.

To implement this solution in FT-Linda, then, an $in\_progress$ tuple is deposited into TS atomically when the subtask tuple is withdrawn, and then removed atomically when the result tuple is deposited. This $in\_progress$ tuple completely describes the subtask tuple. It also indicates the host on which the worker is executing, so if that host fails it can

```
# TSmain is {stable, shared}
process worker()
   while true do
      ⟨ in (TSmain, "subtask", ?subtask_args) ⇒
         out (TSmain, "in_progress", my_hostid, subtask_args) ⟩
      calc (subtask_args, var result_args)
      ⟨ in(TSmain, "in_progress", my_hostid, subtask_args) ⇒
         out (TSmain, "result", result_args) ⟩

   end while
end worker
```

Figure 3.1: Lost Tuple Solution for (Static) Bag-of-Tasks Worker

be ascertained which subtasks need to be recovered from their $in\_progress$ counterparts.

The code for the static version of the bag-of-task worker demonstrating this technique is shown in Figure 3.1. Here, the worker deposits the $in\_progress$ tuple atomically with withdrawing the $subtask$ tuple. It later withdraws this $in\_progress$ tuple atomically with depositing the $result$ tuple. This scheme ensures that the subtask is represented in exactly one form in $TSmain$ at all times, either as a $subtask$ tuple, an $in\_progress$ tuple, or a $result$ tuple.

To complete this example, we also now consider the problem of regenerating the lost subtask tuple from the $in\_progress$ tuple. This job is performed by a *monitor process*. In general, applications need a set of monitor processes for each window of vulnerability, a region of code where a failure would require some recovery, e.g., the regeneration of the $subtask$ tuples from $in\_progress$ ones. One monitor process is created on each machine hosting a TS replica. This ensures that the failure guarantees for monitor processes are exactly as strong as those for stable TSs: there is at least one monitor process from a given set that has not failed if and only if there is at least one TS replica that has not failed. Having some TS replica hosts without a monitor process is possible but provides weaker failure guarantees. In particular, a stable TS might still be available in this case yet essentially inaccessible because all monitor processes had failed.

The monitor process for bag-of-tasks worker is given in Figure 3.2. The monitor waits for a failure tuple indicating the failure of a host and then attempts to recover subtask tuples for all $in\_progress$ tuples associated with the failed host. Note that, even though all monitor processes execute this code, each subtask tuple will be recovered only once due to the atomic semantics of the AGS. The *failure identifier* passed to the monitor process upon initialization is used to match particular failure tuples with monitor processes. This identifier is generated by a call to an FT-Linda routine, which also registers it with the

```
process monitor(failure_id)
   while true do
      in(TSmain,"failure", failure_id, ?host)
      # recover subtask tuples from the failed host
      while ⟨ inp(TSmain, "in_progress", host, ?subtask_args) ⇒
                 out (TSmain, "subtask", subtask_args) ⟩ do
         noop
      end while
   end while
end monitor
```

Figure 3.2: Bag-of-Tasks Monitor Process

language runtime system. When a host failure occurs, one failure tuple is deposited into $TSmain$ for each registered failure identifier.

Note that to guarantee recovery from worker failures, the failure IDs of monitor processes must be registered before any workers are created. Otherwise, a window of vulnerability would exist between the creation of the worker and registration of the failure ID. Should the worker fail, in this case no monitor process would receive notification of this failure, and any subtask the worker was executing while it failed would thus not be recovered.

Note also that each monitor process is guaranteed to get exactly one failure tuple for each failure. This property is guaranteed by Consul's membership service, which generates one failure notification per failure event. This allows the monitor processes to keep consistent information about failures, e.g. the crash counts for each host.

Further ways in which the atomic guarded statement can be used are demonstrated in Chapter 4.

Disjunctive Case

The AGS has a disjunctive case that allows more than one guard/body pair, as shown in Figure 3.3. A process executing this statement blocks until at least one guard succeeds, or all guards fail. To simplify the semantics, and to fit into a programming language's type system, the guards in a given statement must all be the same type of operation, i.e., all **true**, all blocking operations (**in** or **rd**), or all boolean operations (**inp** or **rdp**). In addition, if the guards are boolean, either all or none of the guards may be negated. If the guards are all **true**, then all guards succeed immediately; in this case, the first is chosen and the corresponding body executed. If the guards are all blocking, then the set of guards that would succeed if executed immediately—that is, those for which there is a matching tuple

```
⟨
    $guard_1 \Rightarrow body_1$
or
    $guard_2 \Rightarrow body_2$
or
    ...
or
    $guard_n \Rightarrow body_n$
⟩
```

Figure 3.3: AGS Disjunction

in the named TS when the statement starts—is determined. If the size of that set is at least one, then one is selected deterministically by the implementation, and the corresponding guard and body executed. Otherwise, the process executing the AGS blocks until a guard succeeds. If the guards are all boolean, then the set of guards that would succeed at the time execution is commenced is again determined. If the size of the set is at least one, then the selection and execution is done as before. If, however, the set is empty, the AGS will immediately return false and no body will be executed.

An example using disjunction is given in Section 4.1.1.

3.2.2 Atomic Tuple Transfer

FT-Linda provides primitives that allow tuples to be moved or copied atomically between TSs. The two primitives are of the form

$transfer\_op$($TSfrom$, $TSto$ [, $template$])

Here, $transfer\_op$ is either **move** or **copy**, $TSfrom$ is the source TS , and $TSto$ is the destination TS. The $template$ is optional and consists of a logical name and zero or more arguments, i.e., exactly what would follow the TS handle in a regular FT-Linda TS operation. If the template is present, only matching tuples are moved or copied; otherwise, the operation is applied to all the tuples in the source TS. Also, because the template in a transfer command may match more than one tuple, any formal variables in $template$ are used only for their type, i.e., they are not assigned a new value by the operation.

Although similar to a series of **in** (or **rd**) and **out** operations, these two primitives provide useful functionality, even independent of their atomic aspect. To see this, note that a

```
move($TSfrom$, $TSto$)
```

would move the same tuples as executing

```
in($TSfrom$,$?t$)
out($TSto$, $t$)
```

for each tuple t in $TSfrom$, assuming no other processes accesses $TSfrom$ while these **in-out** pairs are in progress.[4] Moreover, even if the tuples in $TSfrom$ are all of the same tuple *signature*, an ordered list of types, the above **move** is likely to be much more efficient than its Linda equivalent:

```
while inp($TSfrom$,$?t$) do
  out($TSto$, $t$)
```

as we shall see in Chapter 5, Of course, **move** and **copy** are also atomic with respect to both failures and concurrency, while the equivalent sequence of **inp**s and **out**s would not be, even if combined in a series of AGSs:

```
while  ( $\langle$ inp($TSfrom$,$?t$) $\Rightarrow$ out($TSto$, $t$) $\rangle$ ) do
  noop
```

In this case, other processes could observe the intermediate steps here. That is, they could have access to $TSto$ when the **move** operation being simulated was only partially finished, i.e. when some but not all of the tuples had been moved.

As an example of how a **move** primitive might be used in practice, consider the dynamic bag-of-tasks application from Chapter 2. Recall that this particular paradigm suffered from both the lost tuple and the duplicate tuple problems. A way to solve these problems is shown in Figure 3.4. This is similar to the static case shown in Figure 3.1 except that here, $TSscratch$ is a scratch TS that the worker uses to prevent duplicate tuples. To do this, all new subtask tuples as well as the result tuple are first deposited into this TS . Then, the $in\_progress$ tuple is removed atomically with the moving of all the tuples from $TSscratch$ to $TSmain$. If the worker fails before executing the final AGS that performs this task, the subtask will be recovered as before, and another worker will compute the same result and generate the same new subtasks. In this case, any result and subtask tuples in $TSscratch$ will be lost, of course, which is desirable. However, if the worker fails after the final AGS , then the new subtask tuples are already in a stable TS.

Finally, note that a monitor process similar to that for the static worker case, given in Figure 3.2, would be needed here as well. The complete source code for this example is in Appendix E.

[4] This example is not legal Linda because it treats tuples as first-class objects. However, it serves to make the point.

```
process worker()
    TSscratch := ts_create(volatile, private, my_lpid() )
    while true do
        ⟨ in (TSmain, "subtask", ?subtask_args) ⇒
            out (TSmain, "in_progress", my_hostid, subtask_args) ⟩
        calc (subtask_args, var res_args)
        for (all new subtasks created by this subtask)
            out(TSscratch, "subtask", new_subtask_args)
        out(TSscratch, "result", res_args)
        ⟨ in(TSmain, "in_progress", my_hostid, subtask_args) ⇒
            move (TSscratch, TSmain) ⟩
    end while
end worker
```

Figure 3.4: Fault-Tolerant (Dynamic) Bag-of-Tasks Worker

3.3 Tuple Space Semantics

FT-Linda offers semantics that rectify the shortcomings discussed in Section 2.2. These semantics are simple to provide, given that the SMA guarantees that each TS replica receives the same sequence of TS operations, and by the way each replica processes these operations.

First, **inp** and **rdp** in our scheme provide absolute guarantees as to whether there is a matching tuple in TS, a property that we call *strong inp/rdp semantics*. That is, if **inp** or **rdp** returns **false**, FT-Linda guarantees that there is no matching tuple in TS. Of all other distributed Linda implementations of which we are aware, only PLinda [AS91, JS94] and MOM [CD94] offer similar semantics. Strong inp/rdp semantics can be very useful because they make a strong statement about the global state of the TS and hence of the parallel application or system built using FT-Linda.

FT-Linda also provides *oldest matching semantics*, meaning that **in**, **inp**, **rd**, and **rdp** always return the oldest matching tuple if one exists. These semantics are exploited in the disjunctive version of an AGS as well to select the guard and body to be executed if more than one guard succeeds. Oldest matching semantics can be very useful for some applications, as shown in Section 4.1.2.

Finally, unlike standard versions of Linda, **out** operations in FT-Linda are not completely asynchronous. In particular, the order in which **out** operations in a process are applied in their respective TSs is guaranteed to be identical to the order in which they are initiated, which is not the case when **out** operations are asynchronous. This *sequential ordering property* reduces the number of possible outcomes that can result from executing a collection of **out** operations, thereby simplifying the programming process.

As an example of the differences in TS semantics between Linda and FT-Linda, consider the Linda fragment from Section 2.2:

```
process P1      process P2
  out("A")        in("B")
  out("B")        if (inp("A")) then
                    print (``Must succeed!'')
```

Recall that the problem here is that a programmer may assume that the **inp** must always succeed, which is not true without both the strong inp/rdp semantics and the sequential ordering properties. To implement the above code fragment in FT-Linda, we could group each process's operations in a single AGS or implement each op as its own AGS. For example, the following implements the semantics likely intended by the programmer:

```
process P1                    process P2
  ⟨ true ⇒                    ⟨ in("B") ⇒ skip ⟩
    out("A")                  if (⟨ inp("A") ⇒ skip ⟩ ) then
    out("B")                    print (``Must succeed!'')
  ⟩
```

The other grouping alternatives work in the same way; in all cases the **inp** will succeed.

3.4 Related Work

A number of other efforts have addressed the problem of providing support for fault-tolerance in Linda. These include enhanced runtime systems, resilient processes and data, and transactions. We discuss each in turn, as well as other research projects with (non-fault-tolerant) features related to FT-Linda's. We then conclude this section by summarizing the unique language features of our extensions.

Enhanced Linda Runtime Systems

One class of fault-tolerant versions of Linda does not extend Linda per se, but rather focuses on adding functionality to the implementation. [XL89, Xu88] give a design for making the standard Linda TS stable. The design is based on replicating tuples on a subset of the nodes, and then using locks and a general commit protocol to perform updates. This replication technique takes advantage of Linda's semantics so workers suffer little delay when performing TS operations. Specifically, **out** and **rd** are unaffected by these locks; a worker performing an **out** need not wait until the tuple is deposited into TS, and a worker performing a **rd** need only wait until one of the replicas has responded with a matching tuple.

This technique works as follows. An **out** stores a tuple at all TS replicas, and an **in** removes one from all replicas; a **rd** can read from any replica. The **out** is performed in the background, so the worker is not delayed. A **rd** broadcasts a request to all replicas; the worker can proceed when the first reply, which contains all matches for the template at that replica, arrives (assuming there are any matches). An **in** must acquire locks for the signature it wishes to match from all replicas. It broadcasts its request to all replicas. The replicas each send a reply message, indicating whether the lock was granted and, if so, including a copy of all matching tuples. If the worker receives locks from all replicas, it ascertains if there is a matching tuple in all the matching sets it received. If so, the worker's node chooses one and lets the worker proceed, while in the background it sends a message to the other replicas informing them of the tuple selected. However, if not all locks were acquired, or there was no matching tuple in the intersection of the tuples sent by the replicas, the worker sends messages to all replicas releasing the locks, and then starts over.

Processor failures and recoveries and network partitions are handled in [XL89, Xu88] using a view change algorithm based on the virtual partitions protocol in [ASC85]. This allows all workers that are in a majority partition to continue to use TS despite network partitions.

[PTHR93] also implements a stable TS by replication, but uses centralized algorithms to serialize tuple operations and achieve replica consistency for single TS operations. In this scheme, processes attach themselves to the particular subspaces (portions of TS) to which they reference. The degree of replication is dictated by the number of nodes sharing a subspace; all nodes using a particular subspace have a local copy of it. The node with the lowest node ID from among these TS replicas for a given subspace is the *control node*. Requests to delete a tuple (i.e, in the implementation of **in**) are sent to this control node. The other replicas are then each sent a message indicating which tuple to delete from their copy of the subspace.

MTS [CKM92] addresses the issue of relaxing the consistency of the TS replicas to improve performance. With MTS, there is a replica of TS on each node and three different consistencies from which the programmer can choose, depending in the pattern of usage for a given tuple's signature. *Weak consistency* can be used if there are neither simultaneous **in**s nor **rd**s on that signature. To implement this, the **in** routine selects the tuple it deletes, then sends an *erase* message to the other replicas instructing them which tuple to delete from TS. *Non-exclusive consistency* can be used in the absence of simultaneous **in**s on that signature. The node performing the **in** selects the tuple to be deleted, sends a **delete** message to all other replicas to tell them which tuple to delete, and then waits for replies from all the replicas. Finally, *strict consistency* may be used in any situation. It uses a two-phase commit protocol similar to that described above from [XL89, Xu88].

While all these schemes are undoubtedly useful, we note again that adding only this type of functionality without extending the language has been shown to be inadequate for realizing common fault-tolerance paradigms [Seg93]. Also, unlike our approach, most scenarios in the above schemes require multiple messages to update the TS replicas.

Resilient Processes and Data

While the above schemes provided resilient data, another project [Kam90] aims to achieve fault-tolerance by also providing resilient processes. It accomplishes this by checkpointing TS and process states and writing logs of TS operations. Processes unilaterally checkpoint their state to a stable storage as well as the message logs recording what TS operations they have performed. If a process fails, this message log is replayed to reconstruct the process' state.

While the scheme is currently only a design that was never implemented, it appears to have a significant message overhead. It also mandates that all processes be resilient. This overhead is not necessary in many cases, such as the bag-of-tasks paradigm, as shown in Chapter 3. In this paradigm, a given worker process does not have to be resilient, so long as any subtask a failed worker was executing is recovered and executed by another worker. Also, this scheme has no provisions for atomic execution of TS operations. The user can construct them (e.g., with a tuple functioning as a binary semaphore), but this seems less attractive than an explicitly atomic construct.

Transactions and Linda

Two projects have added transactions to Linda. PLinda allows the programmer to combine Linda tuple space operations in a transaction, and provides for resilient processes that are automatically restarted after failure [AS91, JS94]. The programmer can choose one of two modes to ensure the resilience of TS. In the first mode, all updates a transaction makes are logged to disk before the transaction is committed. In the second, the entire TS is periodically checkpointed to disk; the system ensures there are no active transactions during this checkpointing. PLinda handles failed or slow processors, processes, and networks. This design is sufficient for fault tolerance—indeed, it is more general than what FT-Linda provides.

PLinda has its limitations, however. Its TS and checkpoints and are not replicated, so if the disk suffers a loss of data failure (such as a head crash), the program will fail. Also, if the host the disk is attached to fails, then the program cannot progress until that host recovers. Further, as discussed above, many applications do not need the overhead of a resilient process. (This overhead seems to be much less than that in [Kam90, Kam91], however, in part because PLinda provides a mechanism for a process to log its private state.)

MOM provides a kind of lightweight transaction [CD94]. It extends **in** to return a tuple identifier and **out** to include the identifier of its parent tuple (e.g., the subtask it was generated by). It then provides a **done**(id_list) primitive that commits all **in** operations in id_list and all **out** operations whose parents are in id_list plus all I/O performed by the process during the transaction. MOM provides mechanisms to allow the user to construct a limited form of resilient processes. Finally, it also provides oldest-matching semantics.

However, MOM also has limitations, partly due to its intended original domain of long-running, fault-tolerant space systems. One limitation is that its primitives are only designed to support the bag-of-tasks paradigm. Also, it assumes the user can associate a timeout with a subtask tuple, so if the **in** operation does not find a match within this time the transaction will automatically be aborted. Further, MOM also does not replicate its checkpoint information, so it suffers from the same single point of failure that PLinda does.

Other Features Related to FT-Linda

Other features similar to those provided in FT-Linda have also been proposed at various times. [Gel85] briefly introduces *composed statements*, which provide a form of disjunction and conjunction. [Bjo92] includes an optimization that collapses **in-out** pairs on the same tuple pattern; it requires restrictions similar to FT-Linda's opcodes. [Lei89] discusses the idea of multiple tuple spaces, and some of the properties that might be supported in such a scheme. Support for disjunction has also been discussed in [Gel85, Lei89] and in the context of the Linda Program Builder [AG91a, AG91b]. The latter offers the abstraction of disjunction by mapping it onto ordinary Linda operations and hiding the details from the user. None of these efforts consider fault-tolerance.

Summary

FT-Linda has many novel features that distinguish it from other efforts to provide fault-tolerance to Linda. It is unique in that it is the only design that uses the state machine approach to implement this fault tolerance. It is also the only Linda variant to provide multi-op atomicity that is not transaction-based. Its provisions to allow the creation of multiple TSs of different kinds are unique, as are its tuple transfer primitives. FT-Linda is the only Linda version we are aware of to support disjunction, and its collection of strong semantic features is unique.

3.5 Possible Extensions to FT-Linda

A number of features are attractive candidates for addition to FT-Linda: additional attributes for tuple spaces, nested AGSs, AGS branch notification, tuple space clocks, tuple space partitions, guard flags, and allowing TS creation and destruction primitives in the AGS.

3.5.1 Additional Tuple Space Attributes

Two additional attributes for tuple spaces might also be considered. The first is encryption. With this scheme, attribute values can be *unencrypted* or *encrypted*. The latter would imply that all data in the TS are encrypted, while with the former they would not be. This attribute would, of course, be specified when the TS is created. The encrypting of actuals will take place in a part of the FT-Linda runtime system that is in the user process' address space, for security reasons, and the encryption key will be stored there. This way, the key never leaves the user's address space.

Since tuple matching is based only on equality testing, not on other relational operators (e.g., greater than) or ranges, Linda implementations typically implement matching by doing a byte by byte comparison of the actual in the template and the corresponding value in the tuple. As a result, this scheme will still work on encrypted tuple spaces, provided that the actuals in both the **out** that generates tuple and **in** (or other operator) that tries to match it are encrypted with the same key. Thus, the TS managers would not need to be changed to implement this encryption of data.

Note that this encryption scheme can be used in conjunction with a private TS to ensure that only one process can actually access the data. If this is not done, any process could withdraw tuples from the encrypted TS. Also, it could possibly learn something useful from the number and kind of tuples in that TS. Of course, the removal of those tuples could also be harmful in and of itself.

A second possible attribute is to indicate write permissions. A *read-write* TS may be modified at any time, exactly as TSs currently are in FT-Linda. However, *read-only* TSs may not be written after they have been initialized. In such a scheme, the tuple space would be seeded by its creator, which would then call an FT-Linda primitive to announce that the tuple space's initialization is complete. After this point, the TS may not be modified, i.e., it may not be operated on by **out**, **in**, **inp**, **move**, or as a destination for **copy**. Since the TS will no longer change, it could be replicated on each computer and perhaps in each address space. Such a tuple space could be used to disseminate efficiently global data that does not change yet may be accessed often, e.g. the two matrices to be multiplied together.

3.5.2 Nested AGS

The multiple branches of a disjunctive AGS are the only form of conditional execution within an AGS. However, the only form of conditional execution this disjunction allows is to choose which branch is to be executed. Once this decision is made, every operation in the body will be executed. An extension to FT-Linda that would allow another form of conditional execution — within the body — is to allow nested AGSs. One possible syntax is:

```
⟨ $guard_1$ ⇒
    ...
    ⟨ $guard_2$ ⇒ $body_2$ ⟩
    ...
⟩
```

Of course, $body_2$ could, in turn, have one of its elements be another AGS, and so forth.

Recall that to implement the AGS efficiently, we mandate that no TS operation in the body may block. Thus, to allow nested AGSs, **in** and **rd** would not be allowed to block in the guards of an AGS that is not at the outer level, because those guards are part of the body of an enclosing AGS. Thus, in practice their boolean counterparts would most likely be used.

3.5.3 Notification of AGS Branch Execution

One difficulty in using the disjunctional form of the AGS is that often the code following a disjunctive AGS will need to know which branch was executed. This takes extra coding, e.g., either by adding an extra formal variable somewhere in each branch, or by an extra **out** in each branch to deposit a distinguished tuple into a scratch TS. The FT-Linda implementation could directly indicate which branch was executed in a number of ways, for example, with a per-process variable similar to Unix's `errno` variable or with a library routine that returns this information.

3.5.4 Tuple Space Clocks

Another possible extension would be to provide common global time [Lam78]. This has been suggested in a Linda context in the tuple space clock suggested in [Lei89]. Here, the author notes the usefulness of such a clock that preserves potential causality for a distributed program, then states

> The interesting question is whether there is a semantics for weak clocks which allows them to be useful in writing parallel programs, but still admits of an efficient implementation.

Such a clock would be trivial to provide in FT-Linda. Recall that each replicated TS manager receives the same sequence of AGSs containing the same sequence of operations. A running count of this sequence of operations can thus serve as a clock. The value of the clock could be placed in a distinguished tuple (that must not be withdrawn) or made available in some other fashion to the FT-Linda program.

This clock would preserve potential causality, assuming that processes communicate only through TS. For example, suppose process A reads value $time_A$ and then deposits a tuple into TS. If process B withdraws that tuple and then reads $time_B$, $time_A$ must be less than $time_B$. Conversely, if we know process C dealt with event C at $time_C$, process D dealt with event D at $time_D$, and $time_C \geq time_D$, then event C could not have affected event D.

3.5.5 Tuple Space Partitions

There is one set of replicated TS managers in the current FT-Linda implementation (to be described in Chapter 5) that implements all replicated TSs. However, in general, there could be many different unrelated application programs using replicated TSs. It would thus be desirable to allow unrelated applications to have different sets of replicated TS managers, each managing its own set of TSs. This could significantly reduce the workload for each of the replicated TS managers.

To accomplish this, a replicated TS would be created in the context of a given *partition*; a partition could be well-known, dynamically allocated by the runtime system, or both. Each partition would then be managed by a different set of replicated TS managers. An AGS could only contain TSs from one partition, because to allow more would require coordination between different sets of TS managers. Finally, $TSmain$ would be created in its own partition.

3.5.6 Guard Expressions

An AGS's guards are in effect a kind of scheduling expression that dictates which branch will be chosen and when that branch will be executed. This expression can currently be empty (**true**) or a Linda TS operation (**in**, **inp**, **rd**, or **rdp**). This can be extended further to allow a boolean expression to also influence the selection of a guard. Such a boolean expression would be called a *guard expression*. Example guards might appear as follows:

- $expr_1$ **and in**$(\ldots) \Rightarrow$
- $expr_2 \Rightarrow$
- $expr_3$ **and inp**$(\ldots) \Rightarrow$

A guard whose guard expression is false would not be eligible for execution in TS; i.e., if the guard has a TS operation, the TS manager would not even search for a matching tuple for it.

This guard expression would make FT-Linda's guards more similar to constructs in other concurrent languages. The notion of a *guarded communication statement* was introduced in [Dij75]; its guard contained both an optional expression and an optional communication statement. The guarded expression has since been used in various forms in parallel programming languages such as CSP [Hoa78], Ada [DoD83], SR [AO93], and Orca [Bal90].

Recall that in a disjunctive AGS, to simplify the semantics and to fit into a programming language's type system, all guards must be the same: either all absent (**true**), blocking (**in** or **rd**), or boolean (**inp** or **rdp**). This same restriction would hold. for example, because the three guards are blocking, absent, and boolean, respectively, they could not be used in the same AGS. No other restrictions would be required to add guard flags to the language; the guard flag simply narrows the list of eligible guards before the AGS request is sent to the TS managers.

3.5.7 TS Creation in an AGS

In some cases, it is desirable to move tuples to another TS to operate on them, and then either return the tuples or a synthesized result in their place. With this technique, the

TS handle must be left as a placeholder to allow the recovery of those tuples in case the process operating on them fails. For example, a process may use the following:

```
$safe\_ts := ts\_create$(stable, shared)
⟨ true ⇒
   out($TSmain$, "$safe\_ts$", $my\_host, safe\_ts$)
   move($TSmain, safe\_ts$)
   copy($safe\_ts, TSscratch$)
⟩
# Operate somehow on tuples in $TSscratch$
# to produce result tuple
⟨ true ⇒
   in($TSmain$, "$safe\_ts$", $my\_host, safe\_ts$)
   out($TSmain$, "$result$", $result$)
⟩
$ts\_destroy(safe\_ts)$
```

The problem with this scenario is that of the host fails either right before the first AGS or right after the last AGS, then ts_safe will never be destroyed. The accumulation of such orphaned TSs could be a serious problem in a long-running application.

This can be avoided by allowing a TS to be created and destroyed inside an AGS. In the above scenario, the creation and the destruction of ts_safe would be inside the first and last AGS, respectively. Additionally, because variable assignment is not allowed in an AGS, the TS creation primitive would take the TS handle as an additional argument. With this extension, ts_safe could not be orphaned, because its creation and destruction would be atomic with the depositing and withdrawing of the placeholder TS handle, respectively.

This addition would be very simple to implement. Indeed, the commands to create and destroy a replicated TS are already broadcast to the replicated TS managers in messages generated by `ts_create()` and `ts_destroy()`, respectively. It would thus be trivial to allow these operations to also be combined in an AGS with other TS operations.

3.6 Summary

This chapter describes FT-Linda, a version of Linda that allows Linda programs to tolerate the crash failures of some of the computers involved with the computation. FT-Linda supports the creation of multiple tuple spaces with different attributes. It also has provisions for atomic execution, and it also features strong semantics. This chapter also surveys other versions of Linda designed to provide support for fault-tolerance. Finally, this chapter surveys possible extensions to FT-Linda.

FT-Linda supports two attributes for tuple spaces: resilience and scope. The resilience specifies the behavior of the tuple space in the presence of failures; it can be either stable or volatile. The scope attribute — shared or private — indicates which processes may access the tuple space. Both attributes must be specified when the tuple space is created.

FT-Linda has two provisions for atomic execution, the atomic guarded statement and tuple transfer primitives. The atomic guarded statement allows a sequence of tuple space operations to be performed in an all-or-nothing fashion despite concurrency and failures. It can be used to construct a static bag-of-tasks worker, the failure of which can be recovered from by a monitor process. The atomic guarded statement also has a disjunctive form that specifies multiple sequences of tuple space operations, zero or one of which will be executed atomically. FT-Linda's tuple transfer primitives that allow tuples to be moved or copied atomically between tuple spaces. This can be used, in conjunction with the atomic guarded statement and monitor processes, to create a fault-tolerant dynamic bag-of-tasks worker.

FT-Linda offers strong semantics that are useful for fault-tolerant parallel programming. Its Strong **inp/rdp** semantics provide absolute guarantees that the boolean primitives will find a matching tuple if one exists. FT-Linda's oldest matching semantics mean that the oldest tuple that matches the given template will be selected by that operation. Finally, its sequential ordering property ensures that tuple space operations from a given process will be applied in their respective tuple spaces in the order prescribed in that process's code.

Other projects have provided fault-tolerant support for Linda. One class provides resilient tuple spaces but does not extend Linda any; this is not sufficient to solve the distributed consensus problem. Another design provides both resilient process and resilience tuple spaces. Two projects — PLinda and MOM — have added transactional support to Linda.

FT-Linda could be extended in a number of directions. These include additional tuple space attributes, nested atomic guarded statements, tuple space clocks, tuple space partitions, and guard expressions.

CHAPTER 4
PROGRAMMING WITH FT-LINDA

This chapter illustrates FT-Linda's applicability to a wide range of problems. It presents five examples from FT-Linda's two primary domains: highly dependable systems and parallel programming, For simplicity, we only concern ourselves in these examples with the failure of worker or server processes; the final section in this chapter discusses handling the failure of the main/master process.

4.1 Highly Dependable Systems

This section gives FT-Linda implementations of three system-level applications. First, it gives a replicated server example. This is an example of programming replicated state machines in FT-Linda. Next, it presents an FT-Linda recoverable server — an example of the primary/backup approach [AD76, BMST92]. Since the server is not replicated, there are no redundant server computations in the absence of failures. Finally, this section presents an FT-Linda implementation of a transaction facility. This demonstrates the utility of FT-Linda's tuple transfer primitives, and the ability of the language to implement abstractions more powerful than the atomic guarded statement (AGS). Transactions are, of course, an example of the object-action paradigm introduced in Section 1.3.1 [Gra86].

4.1.1 Replicated Server

In this example, a process implements a *service* that is invoked by client processes by issuing *requests* for that service. To provide availability of the service when failures occur, the server is replicated on multiple machines in a distributed system. To maintain consistency between servers, requests must be executed in the same order at every replica.

The key to implementing this approach in FT-Linda is ordering the requests in a failure-resilient way. We accomplish this by using the Linda distributed variable paradigm in the form of a *sequence tuple* for each service. The value of this tuple starts at zero and is incremented each time a client makes a request, meaning that there is exactly one request per sequence number for each service and that a sequence number uniquely identifies a request. This strategy results in a total ordering of requests that also preserves causality between clients, assuming that clients communicate only using TS. This sequence number is also used to ensure that server replicas process each request exactly once.

An FT-Linda implementation of a generic replicated server follows. First, however, for each server (*not* each server replica), the following is performed at initialization time to create the sequence tuple:

```
⟨ in ("sequence", sid, ?sequence) ⇒
    out ("sequence", sid, PLUS(sequence, 1) )
    out ("request", sid, sequence, service_name, service_id, args)
⟩
if (a reply is needed for service)
    in("reply", sid, sequence, ?reply_args)
```

Figure 4.1: Replicated Server Client Request

```
out ("sequence", server_id, 0)
```

The *server_id* uniquely identifies the (logical) server with which the sequence tuple is associated.

Given this tuple, then, a client generates a request by executing the code shown in Figure 4.1. Here, the client does three things: withdraws the sequence tuple, deposits a new sequence tuple, and deposits its request; after this it withdraws the appropriate reply tuple if necessary. These three actions must be performed atomically. To see this, consider what would happen given failures at inopportune times. If the processor on which the client is executing fails between the **in** and the **out**, the sequence tuple would be lost, and all clients and server replicas would block forever. Similarly, if a failure occurs between the two **out**s, there would be no request tuple corresponding with the given sequence number, so the server replicas would block forever.

Two additional aspects of the client code are worth pointing out. First, note that the client includes a *service_name* and *service_id* in the request tuple. This information specifies which of the services by server *sid* is to be invoked. The redundancy is needed for structuring the server, as will be seen below. Second, note the **PLUS** in the TS operation depositing the updated sequence tuple. This *opcode* results in the value of the sequence tuple that was withdrawn in the previous **in** being incremented when it is deposited back into TS . These opcodes, which also include such common operations like **MIN**, **MAX**, and **MINUS**, are intended to allow a limited form of computation within atomic guarded statements. As already mentioned above, general computations—including the use of expressions or user functions in arguments to TS operations—are not allowed due to their implied implementation overhead. For example, among other things, it would mean having to transfer the code for the computation to all processors hosting TS replicas so that it could be executed at the appropriate time. Also, these general computations must be executed, in essence, in a critical section—that is, while the runtime system is processing the AGS in question—which could severely degrade overall performance.

The code for a generic server replica that retrieves and services request tuples is given in Figure 4.2. The server waits for a request with the current sequence number using the disjunctive version of the AGS, one branch (i.e., guard/body pair) for each

```
$seq := 0$
loop forever
    ⟨
        rd($TSmain$, "$request$", $my\_sm\_id, seq, service_1, ?servicenum, ?x$) $\Rightarrow$ skip
    or
        ...
    or
        rd($TSmain$, "$request$", $my\_sm\_id, seq, service_n, ?servicenum, ?a, ?b, ?c$) $\Rightarrow$ skip
    ⟩
    case $servicenum$ of
        # each service does out("$reply$", $my\_sm\_id, seq, reply\_args$) if
        # the service returns an answer to the client
        $1:$ $service_1(x)$
            ...
        $n:$ $service_n(a,\ b,\ c)$
    end case
    $seq := seq + 1$
end loop
```

Figure 4.2: Server Replica

service offered by the server. When a request with the current sequence number arrives, the server uses the assignment of the $service\_id$ in the tuple to variable $servicenum$ to record which service was invoked. This tactic is necessary because normal variable assignment to record the selection is not permitted within an atomic guarded statement. Finally, after withdrawing the request tuple, the server executes a procedure (not shown) that implements the specified service. This procedure performs the computation, and, if necessary, deposits a reply tuple.

An alternate way of ascertaining which branch was executed is to have each branch deposit a distinguished tuple into a scratch TS in which one field indicates which branch was executed. For example, branch 3 would execute:

```
out ($TSscratch$,"$branch$", 3)
```

A third alternative is to extend FT-Linda to indicate this directly, as discussed above in Section 3.5.3.

Note that in the above scheme, some tuples get deposited into TS but not withdrawn. In particular, request tuples are not withdrawn, and neither are reply tuples from all but one of the replicas. If leaving these tuples in TS is undesirable, a garbage collection scheme could be used. To do this for request tuples, the sequence number of the last request processed by each server replica would first need to be maintained. Since no request with an earlier sequence number could still be in the midst of processing, such tuples could be withdrawn. A similar scheme could be used to collect extra reply tuples as well.

The complete source code for this example is in Appendix B.

4.1.2 Recoverable Server

Another strategy for realizing a highly available service is to use a *recoverable server*. In this strategy, only a single server process is used instead of the multiple processes as in the previous section. This saves computational resources if no failure occurs, but also raises the possibility that the server may cease to function should the processor on which it is executing fail. To deal with this situation, the server is constructed to save key parts of its state in stable storage so that it can be recovered on another processor after a failure. The downside of this approach when compared to the replicated server approach is, of course, the unavailability of the service during the time required to recover the server.

In our FT-Linda implementation of a recoverable server, a stable TS functions as a stable storage in which values needed for recovery are stored. Monitor processes on every processor wait for notification of the failure of the processor on which the server is executing; should this occur, each attempts to create a new server. Distributed consensus is used to select the one that actually succeeds.

An FT-Linda implementation of such a recoverable server and its clients follows. For simplicity, we assume that the server only provides one service, and that the service requires a reply; multiple services would be handled with disjunction as in the previous example. We also assume that the following is executed upon initialization to create the server:

```
server_lpid := new_lpid()
server_TS_handle := ts_create(stable, private, server_lpid)
out(TSmain,“server_handle”, server_id, server_TS_handle)
out(TSmain, “server_registry” ,server_id, serve_lpid, host)
create(server, host, server_lpid, server_id)
```

This allocates an LPID for the server, creates a private but stable tuple space for its use, places the handle for this tuple space in the globally-accessible TS $TSmain$, and creates the server. The final **out** operation creates a *registry tuple* that records which processor is executing the server. This tuple is used in failure handling, as described below.

```
process $server(my\_id)$
    # Read in handle of private TS
    rd($TSmain$, "$server\_handle$", $my\_id$, $?TSserver$)
    # Read in state if present in TSserver, otherwise initialize it there
    $state$ := $initial\_state$
    ⟨ not rdp($TSserver$, "$state$", $my\_id$, $?state$) ⇒
        out($TSserver$, "$state$", $my\_id$, $initial\_state$)
    ⟩
    loop forever
        ⟨ in($TSmain$, "$request$", $my\_id$, $?client\_lpid$, $?args$) ⇒
            out($TSmain$, "$in\_progress$", $my\_id$, $client\_lpid$, $args$)
        ⟩
        # calculate service & its reply, change $state$, do output
        ...
        ⟨ in($TSmain$, "$in\_progress$", $my\_id$, $client\_lpid$, $args$) ⇒
            out($TSmain$, "$reply$", $my\_id$, $reply\_args$)
            in($TSserver$, "$state$", $my\_id$, $?old\_state$)
            out($TSserver$, "$state$", $my\_id$, $state$)
        ⟩
    end loop
end $server$
```

Figure 4.3: Recoverable Server

The code used by client processors to request service is:

```
out($TSmain$, "$request$", $server\_id$, $client\_lpid$, $args$)
in($TSmain$, "$reply$", $server\_id$, $client\_lpid$, $?reply\_args$)
```

Note that no sequence tuple is needed for the client of a recoverable server, because there is only one actual server process at any given time. Another function this tuple performs in Section 4.1.1, beyond ordering requests for the server, is to ensure that the client withdraws the reply tuple corresponding to its request. To ensure this without the sequence tuple, the client includes its LPID in the request tuple, the server includes it in the reply tuple, and then the client withdraws the reply tuple with its LPID in it.

The server itself is given in Figure 4.3. The server first reads its initial state and TS handle from $TSmain$, and then enters an infinite loop that withdraws requests and leaves an $in\_progress$ tuple as in previous examples. Finally, it performs the service, updates the state tuple, and outputs a reply.

```
process monitor(failure_id, server_id)
   loop
      ⟨ in(TSmonitor,"failure", failure_id, ?host) ⇒ skip ⟩
      # see if server server_id was running on failed host
      if (⟨ rdp(TSmain, "server_registry", server_id, ?server_lpid, host) ⇒ skip⟩ )
         then
         # Regenerate request tuple if found
         ⟨ inp(TSmain, "in_progress", server_id, ?client_lpid, ?args) ⇒
            out (TSmain, "request", server_id, client_lpid, args) ⟩
         # Attempt to start new incarnation of failed server
         if( ⟨ inp(TSmain, "server_registry", server_id, ?server_lpid, host) ⇒
               out(TSmain, "server_registry", server_id, server_lpid, my_host)⟩ )
            then
            create (server, my_host, server_lpid, server_id)
         end if
      end if
   end loop
end monitor
```

Figure 4.4: Recoverable Server Monitor Process

When a processor fails, two actions must be taken. First, any *in_progress* tuple associated with a failed server must be withdrawn and the corresponding request tuple recovered. Second, the failed server itself must be recreated on a functioning processor. To perform these actions, however, we need to know if the failed processor was in fact executing a server. This information is determined using the registry tuples alluded to above.

A monitor process strategy similar to the bag-of-tasks example is used to implement these two actions. Here, it is structured to monitor a single server, although it could easily be modified to handle the entire set of servers in a system. The code is shown in Figure 4.4. The monitor handles the failure of its server by first dealing with a possible *in_progress* tuple; if present, one such monitor will succeed in regenerating the associated request tuple. The next step is to attempt to create a new incarnation of the server, a task that requires cooperation among the monitor processes on each machine to ensure that only one is created. This is implemented by having the monitor processes synchronize using the registry tuple in a form of distributed consensus to agree on which should start the new incarnation. The selected process then creates the new server, while the others simply continue.

The above scheme features one recoverable server process executing at any given time. However, this scheme could easily be modified for performance reasons by having multiple recoverable servers working on different requests in parallel. This would only require slight modifications to the above scheme. For example, the monitor process could not assume there was at most one $in\_progress$ tuple and thus would have to loop until it found no more. Note that this scheme is different from the replicated server scheme in Section 4.1.1. With this scheme, there are multiple primary/backup servers implementing the same service, but they operate independently on different requests. In the replicated server example, on the other hand, all servers execute each request submitted.

The recovery code for the recoverable server is more complicated than for the replicated server in the previous section. Fortunately, however, failures are usually infrequent relative to client requests, so the additional cost of the recovery code will rarely be paid in practice. Also, thanks to FT-Linda's oldest-matching semantics, a sequence tuple does not have to be maintained, as it does with the replicated server. These benefits accrue for each client request and, for many applications, far outweigh the extra costs incurred for each failure. Those applications that cannot tolerate the fail-over time of the recoverable server may need to use the replicated server instead.

The complete source code for this example is in Appendix C.

4.1.3 General Transaction Facility

A transaction is a sequence of actions that must be performed in an all-or-nothing fashion despite failures and concurrency. Although similar to an AGS, a transaction is more general, because the latter can feature arbitrary computations and, in the Linda context, an indefinite number of TS operations. However, we can construct a transaction from a sequence of AGSs. In this subsection, we give an FT-Linda implementation of a library of procedures that provide transactions to user-level processes. The interface for this library is given in Figure 4.5. For simplicity, we assume that variable identifiers are well-known or ascertainable, and that variables are simply integers.

To initialize the system, *init_transaction_library()* is called once, followed by *create_var()* for every variable to be involved in any transaction. After this point, usage of transactions may begin. To perform a transaction, *start_transaction()* is called with a list of the variables to be involved with the transaction; it returns a unique transaction identifier (TID). After this, *modify_var()* is called each time a variable is to be modified, followed by either *commit()* or *abort()*. Finally, we provide *print_variables()* to print out an atomic snapshot of all variables. Note that the user of this transaction library has only to be aware of this interface, not of the fact that it is implemented with FT-Linda.

To implement this transaction facility, we maintain one lock tuple and one variable tuple for each variable created. Recall that in the fault-tolerant bag-of-tasks solution in Section 3.2.1, a subtask being worked on was kept in an alternate form (an "$in\_progress$" tuple) in $TSmain$. We call such a tuple a *placeholder* or a *placeholder tuple*, and say that the original tuple is in its *placeholder form*. In this transaction manager, then, to implement both stability of and mutual exclusion on a variable's tuples, we will convert

```
# Types
type tid_t is int # Transaction ID
type var_t is int # Variable ID
type val_t is int # Variable Value

# Transaction procedures
procedure init_transaction_library()
procedure create_var(var_t var_id, val_t init_val)
procedure destroy_var(var_t var_id)
procedure start_transaction(var_t var_id_list[], int num_vars) returns tid_t
procedure modify_var(tid_t tid, var_t var_id, val_t new_val)
procedure abort(tid_t tid)
procedure commit(tid_t tid)
procedure print_variables(string message)
```

Figure 4.5: Transaction Facility Interface

these tuples into their placeholder forms when their variable is being used in a transaction. Also, during a transaction, two scratch TSs are kept, one with the starting values of all variables in the transaction, and the other with the current values of all variables in the transaction. Maintaining these two scratch TSs makes it simple to abort or commit all changes to the variables with a single AGS consisting primarily of a few **move** operations. Finally, because the transaction facility is implemented as a library of routines linked into the user's address space, the monitor processes are the only processes created — there are no server or other system processes of any kind. Thus, the only thing the monitor processes have to do is to abort any transactions that were being executed by a user on a host that failed.

In the following figures we describe the FT-Linda implementation of the routines given in Figure 4.5. For simplicity, these implementations do not concern themselves with checking for user errors, e.g. testing if a variable used in a transaction has been created previously.

Figure 4.6 gives the routines to initialize the transaction facility.[1] The two things that *init_transaction_facility()* has to do is to create a tuple storing the current TID and also create the monitor processes. To create a variable, *create_var()* creates two tuples in $TSmain$, one for the variable and the other for the lock. Conversely, to destroy a variable, the lock and variable tuples must be removed. Of course, variables should not

[1] In this chapter, we abbreviate an AGS with a **true** guard:

$\langle$ **true** $\Rightarrow$ $body$ $\rangle$

as

$\langle$ $body$ $\rangle$

for the sake of brevity and clarity.

```
procedure init_transaction_facility()
    # Initialize the transaction IDs
    ⟨ out(TSmain, “tids”, 1) ⟩
    # Create the monitor processes
    for $i := 1$ to num_hosts() do
        lpid := new_lpid()
        f_id := new_failure_id()
        create(monitor_transactions, i, lpid, f_id)
    end for
end init_transaction_facility

procedure create_var(var_t var_id, val_t init_val)
    ⟨
        out(TSmain, “var”, var_id, init_val)
        out(TSmain, “lock”, var_id)
    ⟩
end create_var

procedure destroy_var(var_t var_id)
    # Block until any pending transaction using var_id completes
    ⟨ in(TSmain, “lock”, var_id) ⇒
        in(TSmain, “var”, var_id, ?val_t)
    ⟩
end destroy_var
```

Figure 4.6: Transaction Facility Initialization and Finalization Procedures

be destroyed if they may still be in use.

Figure 4.7 gives the code to start a transaction. First, the scratch TSs for the original and current values of the variables are created. Their TS handles are then stored in $TSmain$ for later retrieval by internal routines *get_orig()* and *get_cur()*, respectively. After this, mutual exclusion is acquired on all variables involved with the transaction; this is performed in a linear order to ensure that deadlock cannot occur. To acquire mutual exclusion on a variable, its lock tuple is withdrawn. Additionally, to facilitate recovery from failures as well as the ability to either commit or abort the transaction later, the lock and variable tuples are moved to their placeholder form in $TSmain$, a copy of the variable tuple is placed into both scratch TSs, and a copy of the lock tuple is placed into $orig\_ts$. Finally, start_transaction returns the TID for this transaction; the user must pass this to *modify_var()*, *commit()*, and *abort()*.

Figure 4.8 gives the code to modify a variable. It updates the value of the variable's tuple in the transaction's scratch tuple space that stores the current values of the transaction's

```
procedure $start\_transaction$(var_t $var\_id\_list$[], int $num\_vars$) returns tid_t
    # Allocate the next transaction ID for this transaction
    $\langle$ in($TSmain$,“$tid$”, $?tid$) $\Rightarrow$
        out($TSmain$,“$tid$”,PLUS($tid, 1$))
    $\rangle$
    # Create scratch TSs for original and current values of
    # vars involved in this transaction
    $cur\_ts := ts\_create$(volatile, private, $my\_lpid$)
    $orig\_ts := ts\_create$(volatile, private, $my\_lpid$)
    # Deposit handles into TS for retrieval by get_cur() and get_orig()
    $\langle$
        out($TSmain$,“$ts\_cur$”, $tid, cur\_ts$)
        out($TSmain$,“$ts\_orig$”, $tid, orig\_ts$)
    $\rangle$
    # Acquire locks for vars in this transaction, in a linear order
    # In doing so, move its lock tuple to a lock_inuse tuple,
    # do similarly for the var tuple, and add a copy of the var to $cur\_ts$
    $sort(var\_id\_list[])$
    for $i := 1$ to $num\_vars$ do
        $\langle$ in($TSmain$, “$lock$”, $var\_id\_list[i]$) $\Rightarrow$
            out($TSmain$, “$lock\_inuse$”, $my\_hist, tid, var\_id\_list[i]$)
            out($orig\_ts$, “$lock$”, $var\_id\_list[i]$)
            in($TSmain$, “$var$”, $var\_id\_list[i], ?val$)
            out($TSmain$, “$var\_inuse$”, $my\_hist, tid, var\_id\_list[i], val$)
            out($orig\_ts$, “$var$”, $var\_id\_list[i], val$)
            out($cur\_ts$, “$var$”, $var\_id\_list[i], val$)
        $\rangle$
    end for
    return $tid$
end $start\_transaction$
```

Figure 4.7: Transaction Initialization

```
procedure modify_var(tid_t tid, var_t var_id, val_t new_val)
    cur_ts := get_cur(tid)
    ⟨
        in(cur_ts, "var", var_id, ?val_t)
        out(cur_ts, "var", var_id, new_val)
    ⟩
end modify_var
```

Figure 4.8: Modifying a Transaction Variable

variables.

Figure 4.9 gives the code to abort and commit a transaction. To abort a transaction, the lock and variable tuples must be restored in $TSmain$ and their placeholders discarded. Also, the tuples storing the TS handles for the transaction's scratch TSs are withdrawn. The code to commit a transaction is identical to the code to abort, except that the variable tuples are moved from the scratch TS holding the current values of the variables, rather than the one holding their original values.

The code to print out an atomic snapshot of all variables is given in Figure 4.10. An AGS copies all variables into a scratch TS, either in their normal or placeholder form. From there, they can be withdrawn one a time and their values printed out.

Finally, Figure 4.11 gives the monitor process for the transaction facility. Recall that, because it has no system processes to recover; it only has to abort any transactions that were being executed by a client on the failed host. Thus, the monitor process only regenerates variable and lock tuples for any variables that were acquired by a transaction on the failed host. These tuples are recovered from their placeholders already in $TSmain$.

The transaction facility given above requires a modest number of AGS requests. To implement a transaction the main cost is one *replicatedAGS*, an AGS involving replicated TSs, per variable in the transaction; this happens when the transaction is started. After this, it only costs one *local AGS*, an AGS involving only local (scratch) TSs, to modify a variable, and then one replicated AGS to either commit or abort the transaction. Also, it only takes one replicated AGS and then one scratch AGS per variable to print out an atomic snapshot of all variables. Finally, note that only one process may participate in a given transaction in the above implementation. However, it would be very simple to extend this example to allow multiple processes to cooperate in realizing a single transaction. The only change would be to create the transaction's TSs ("ts_cur" and "ts_orig") as stable and shared rather than as volatile and private. Of course, this would make this transaction facility more expensive. In particular, *modify_var()* would be much more expensive; it would require a replicated AGS rather than a (much cheaper) local one.

The complete source code for this example is in Appendix D.

```
procedure abort (tid_t tid)
    cur_ts := get_cur(tid)
    orig_ts := get_orig(tid)
    # Put the lock and (old) var tuples involved with this transaction
    # back into TSmain, discard the scratch TS tuples,
    # and discard the lock and variable inuse placeholders
    ⟨
        move (orig_ts, TSmain, “lock”, ?var_t)
        move (orig_ts, TSmain, “var”, ?var_t, ?val_t)
        in (TSmain, “ts_cur”, tid, ?ts_handle_t)
        in (TSmain, “ts_orig”, tid, ?ts_handle_t)
        move (TSmain, cur_ts, “lock_inuse”, my_host, tid, var_t)
        move (TSmain, cur_ts, “var_inuse”, my_host, tid, ?var_t, ?val_t)
    ⟩
    ts_destroy(cur_ts)
    ts_destroy(orig_ts)
end abort

procedure commit (tid_t tid)
    cur_ts := get_cur(tid)
    orig_ts := get_orig(tid)
    # Put the lock and (new) var tuples involved with this transaction
    # back into TSmain, discard the scratch TS tuples,
    # and discard the lock and variable inuse placeholders
    ⟨
        move (orig_ts, TSmain, “lock”, ?var_t)
        move (cur_ts, TSmain, “var”, ?var_t, ?val_t)
        in (TSmain, “ts_cur”, tid, ?ts_handle_t)
        in (TSmain, “ts_orig”, tid, ?ts_handle_t)
        move (TSmain, cur_ts, “lock_inuse”, my_host, tid, var_t)
        move (TSmain, cur_ts, “var_inuse”, my_host, tid, ?var_t, ?val_t)
    ⟩
    ts_destroy(cur_ts)
    ts_destroy(orig_ts)
end commit
```

Figure 4.9: Transaction Abort and Commit

```
procedure print_variables(string message)
   # Copy an atomic snapshot of all variables, whether inuse or not
   scratch_ts := ts_create(volatile, private, my_lpid)
   ⟨
      copy(TSmain, scratch_ts, "var", ?var_t, ?val_t)
      copy(TSmain, scratch_ts, "var_inuse", ?int, ?tid_t, ?var_t, ?val_t)
   ⟩
   print(``Variables at'', message)
   while ( ⟨ inp(scratch_ts, "var", ?var, ?val) ⇒ skip ⟩ ) do
      print(`` var'', var, ``value'', val)
   end while
   while ( ⟨ inp(scratch_ts, "var_inuse", ?host, ?tid, ?var, ?val) ⇒ skip ⟩ ) do
      print(`` var'', var, ``value'', val, ``inuse with tid'', tid,
         ``on host'', host)
   end while
   ts_destroy(scratch_ts)
end print_variables
```

Figure 4.10: Printing an Atomic Snapshot of all Variables

```
procedure monitor_transactions(failure_id)
   loop
      ⟨ in(TSmonitor, "failure", failure_id, ?host) ⇒ skip ⟩ # Wait for a failure
      # Regenerate all lock and variable tuples we find in-progress
      # for any transactions on the failed host.
      while ( ⟨  inp(TSmain, "lock_inuse", failed_host, ?tid_t, ?var) ⇒
                 out(TSmain, "lock", var)
                 in(TSmain, "var_inuse", failed_host, ?tid_t, var, ?val)
                 out(TSmain, "var", var, val)
              ⟩
            ) do
         noop
      end while
      end loop
   end monitor_transactions
```

Figure 4.11: Transaction Monitor Process

```
loop forever
    in("subtask", ?subtas_argsk)
    if(small_enough(subtask_args))
        out("result", result(subtask_args))
    else
        out("subtask", part1(subtask_args))
        out("subtask", part2(subtask_args))
    end if
end loop
```

Figure 4.12: Linda Divide and Conquer Worker

4.2 Parallel Applications

FT-Linda is applicable to a wide variety of parallel applications. We have already seen one user-level FT-Linda application in the bag-of-tasks example given in Chapter 3. This section presents two more. The first, the divide-and-conquer worker, is a generalization of the bag-of-tasks worker. The second one is implementation of barriers with FT-Linda; these barriers and related algorithms are applicable to a large number of scientific problems.

4.2.1 Fault-Tolerant Divide and Conquer

The basic structure of divide and conquer is similar to the bag-of-tasks, where subtask tuples representing work to be performed are retrieved by worker processes [Lei89]. The difference comes in the actions of the worker. Here, upon withdrawing a subtask tuple, the worker first determines if the subtask is "small enough," a notion that is, of course, application dependent. If so, the task is performed and the result tuple deposited. However, if the subtask is too large, the worker divides it into two new subtasks and deposits representative subtask tuples into TS. Such a worker is depicted in Figure 4.12.

An FT-Linda solution that provides tolerance to processor crashes is given in Figure 4.13. Here, the worker leaves an *in_progress* tuple when withdrawing a subtask, as done with the bag-of-tasks. It then decides if the task is small enough. If it is, it calculates the answer and atomically withdraws the *in_progress* tuple while depositing the answer tuple. If not, it divides the task into two subtasks and atomically deposits these new subtask tuples while withdrawing the in_progress tuple. A monitor process similar to the one discussed in Section 3.2.1 would be used to recover lost subtask tuples upon failure.

An alternative strategy involving a scratch TS similar to that used in Section 3.2.2 could also be employed. The subtask tuples are first deposited into the scratch TS, which is then merged atomically with the shared TS upon withdrawal of the *in_progress* tuple. This

```
loop forever
    ⟨ in("$subtask$", ?$subtask\_args$) ⇒ out("$in\_progress$", $subtask\_args, my\_hostid$) ⟩
    if($small\_enough(subtask\_args)$)
        $result\_args := result(subtask\_args)$
        ⟨ in("$in\_progress$", $subtask\_args, my\_hostid$) ⇒ out("$result$", $result\_args$) ⟩
    else
        $subtask1\_args := part1(subtask)$
        $subtask2\_args := part2(subtask)$
        ⟨
            in("$in\_progress$", $subtask\_args, my\_hostid$) ⇒
                out("$subtask$", $subtask1\_args$)
                out("$subtask$", $subtask2\_args$)
        ⟩
    end if
end loop
```

Figure 4.13: FT-Linda Divide and Conquer Worker

strategy would be especially appropriate if a variable number of subtasks are generated depending on the specific characteristics of the subtask being divided.

The complete source code for this example is in Appendix F..

4.2.2 Barriers

Many scientific problems can be solved with iterative algorithms that compute a better approximation to the solution with each iteration, stopping when the solution has converged. Examples of such problems include partial differential equations, region labelling, parallel prefix computation, linked list operations, and grid computations [And91]. Such algorithms typically involve an array, with the same computation being performed on each iteration. Since the computations for a given portion of the array and for a given iteration are independent of the other computations for that iteration, such algorithms are ideal candidates for parallelization. However, the computation for a given iteration depends on the result from the previous iteration. Thus, all processes must synchronize after each iteration to ensure that they all are using the correct iteration's values. This synchronization point is called a *barrier*, because all processes must arrive at this point before any may proceed past it.

While barriers are convenient for parallel programming, they would be even more convenient if an application using barriers could tolerate the failure of some of the processors involved with the computation. This would mean making the worker processes resilient, so that if one failed, a reincarnation would be created to resume execution where the failed worker stopped.

In this subsection, we develop a fault-tolerant barrier with FT-Linda. First, we briefly discuss multiprocessor implementation of barriers, outlining three different implementation techniques: shared counters, coordinator processes, and tree-structured barriers. This serves to survey the problem more concretely. We then discuss a simpler way to implement barriers with Linda, taking advantage of its associative matching and blocking primitives. A fault-tolerant FT-Linda version of this solution is then given. This is followed by an explanation of how the techniques used in making this fault-tolerant barrier can be used to make the other kinds of barriers fault-tolerant, as well as classes of algorithms similar to those involving barrier synchronization.

Multiprocessor Implementations of Barriers

We now describe three techniques for implementing a barrier on a multiprocessor; this material is summarized from [And91], where two other schemes are also given. The first technique is to maintain a *shared counter* that records the number of processes that have reached the barrier. Thus, when it arrives at the barrier, a worker process increments its value and then busy waits (spins) until the counter's value equals the number of workers. While this technique is simple to implement, it suffers from severe memory contention; on each iteration, each worker writes to the counter and then continually reads it. This does not scale well, even with the assistance of multiprocessor cache coherence and atomic machine instructions such as fetch and add.

The second technique is to use a *coordinator process*. With this scheme, when a worker arrives at a barrier, it informs a distinguished coordinator process, and then waits for permission from the coordinator to proceed. For each iteration, then, the coordinator simply waits until all workers have arrived at the barrier, then gives them all permission to proceed.

Using a coordinator process solves most of the memory contention problems associated with the shared counter technique. However, it introduces two new problems that limit its scalability. First, it uses an additional process. Since the size of many problems is a power of two, and the number of processors in many multiprocessors is even (or even a power of two), this means that the coordinator will often have to share a processor with a worker. This will slow down all workers, because none may proceed until all have completed a given iteration. Second, the number of computations the coordinator must perform increases linearly with the number of workers. This also affects scalability, because most or all workers are generally not doing productive work while the coordinator is performing these computations, i.e. actively receiving replies and disseminating permission to pass.

A third technique is to use a tree-structured barrier. Recall that both the shared counter and the coordinator process implementation had scalability and other problems. We can overcome these problems by eliminating the coordinator process and disseminating its logic among the worker processes. A tree-structured barrier organizes the workers in

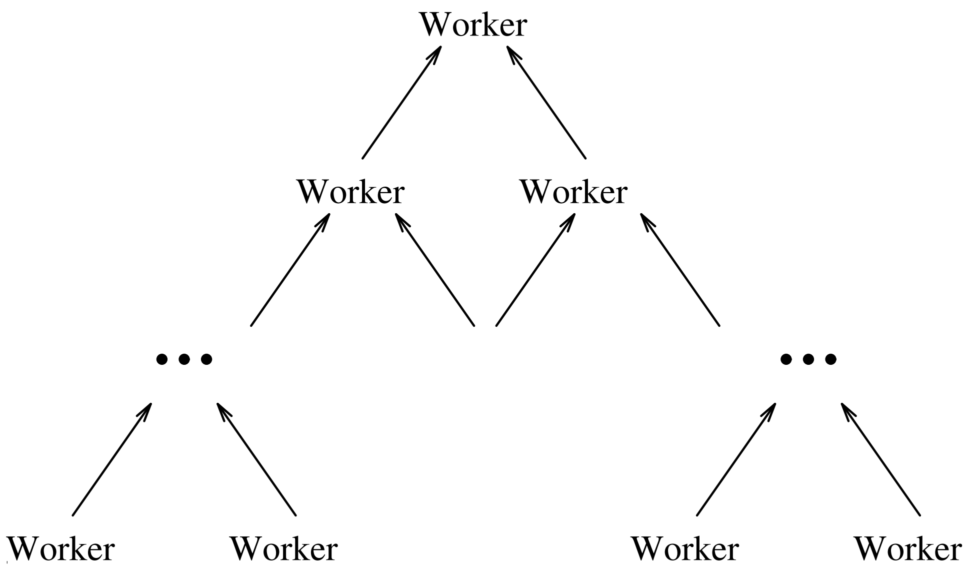

Figure 4.14: Tree-structured barrier

a tree in the manner shown in Figure 4.14.[2] In this scheme, the workers signal their completion up the tree. They then wait for permission to continue; this will be broadcast by the root with some multiprocessor architectures and signalled back down the tree in others. Note that with this barrier there are three different kinds of synchronization logic: in the leaf nodes, which only pass up a signal; in the interior nodes, which both wait for and then pass up signals, and in the root node, which only waits for a signal. Similarly, these different nodes also have different roles regarding broadcasting or passing down the permission to complete. This approach leads to logarithmic execution time and also scales well, because workers are signalling and broadcasting up the tree in parallel.

A final implementation note is that on a multiprocessor, two versions of the array are typically maintained: one with the current values of the array and the other in which the workers store the next iteration's results. At the barrier, the role of each array is switched. Also, we note that different parallel iterative algorithms vary in the way in which they read the last iteration's global data during a given iteration: some barriers have to use all elements of the array on every iteration, while others require access to only a small portion. Thus, to avoid confusion, in the Linda examples in this subsection we add a comment indicating where the data is initialized and where it is read in each iteration. Finally, some barriers also have a reduction step after the barrier and then a second barrier after this. (The reduction step is used, for example, to detect termination.) In some cases this reduction step can also be combined with the synchronization step in the first barrier (and sometimes even also with the reading of the data). We also omit this reduction phase for brevity and clarity. However, it is fairly straightforward to implement these in both Linda and FT-Linda.

[2] This figure is taken from [And91].

```
procedure init_barrier()
    out("barrier_count", 0)
    # ...initialize both copies of the global array ...
    # Create worker(id:1..N) with local Linda's
    # process creation mechanism (omitted)
end init_barrier
```

Figure 4.15: Linda Shared Counter Barrier Initialization

Linda Barriers

A Linda barrier could reasonably be implemented with a shared counter, a coordinator process, or a tree. For example, because of Linda's associativity and blocking operations there would not be memory contention on a shared counter; all workers would simply block until they were allowed to proceed. To demonstrate this, a Linda barrier using a shared counter is given in Figures 4.15 and 4.16; it uses a single counter tuple to record the number of workers that have reached the barrier. A Linda implementation with a coordinator process is similarly easy to envision, although it would still suffer from the same problems as the multiprocessor version, namely the existence and computational delay imposed by the extra process. Finally, a Linda barrier with a tree is likewise simple to realize.

While the shared counter and coordinator process techniques can thus be implemented with Linda, we can take advantage of Linda's associativity and blocking operations to devise a solution with similar performance but one that serves as a cleaner basis for a fault-tolerant barrier. In this scheme, each worker produces a tuple on each iteration that indicates it is ready to proceed to the next iteration; a worker waits for tuples from all workers. The pseudocode for this solution is given in Figure 4.17. The initialization consists of depositing the initial values for the array in TS, as well as creating the worker processes.

Fault-Tolerant Barriers

The solution in Figure 4.17 is simple to use as a base for a fault-tolerant version. Indeed, looking at Figure 4.17, it may not be apparent that the failure of a worker would require any recovery at all. It certainly has nothing like the windows of vulnerability associated with a distributed variable or a bag-of-tasks subtask. However, in this barrier example, the failure of a worker does indeed require recovery because it must be resilient.

To elaborate, recall that with the bag-of-tasks paradigm, a given worker process did not have to be resilient; if it failed, any subtask that was in its placeholder form simply had to be recovered and later re-executed by another worker. The failed worker process

```
process $worker$(id)
    # Initialize iter, current, next
    $iter := 1$
    $current := iter$
    $next := 1 - current$
    # ...read in some or all of global array for iteration $current$ ...
    # Compute next iteration, unless answer has converged
    while not $converged(a, iter, epsilon)$ do
        # ...update my portion of global array for next iteration from $my\_array$ ...
        $compute(my\_array, id)$
        # Wait at the barrier:  get count and see if I'm last
        in(“$barrier\_count$”, $?count$)
        if ($count < (N - 1)$) then
            # Not all workers at barrier yet
            out(“$barrier\_count$”, $count + 1$) # Update count
            in(“$barrier\_proceed$”, $iter + 1$) # Wait until last done
        else
            # I'm the last to arrive at the barrier
            out(“$barrier\_count$”, $0$) # Reset count for next time
            for $i := 1$ to $(N - 1)$ do
                out(“$barrier\_proceed$”, $iter + 1$) # Let other workers proceed
            end for
        end if
        # Update iter, current and next
        $iter := iter + 1$
        $current = next$
        $next = 1 - current$
        # ...read in some or all of global array for iteration $current$ ...
    end while
end $worker$
```

Figure 4.16: Linda Shared Counter Barrier Worker

```
procedure $init\_barrier$()
    # ...initialize the global array ...
    # Create worker(id:1..N) with local Linda's
    # process creation mechanism (omitted)
end $init\_barrier$

process $worker$($id$)
    $iter := 1$
    # ...read in some or all of global array for next iteration ...
    # Compute next iteration, unless answer has converged
    while not $converged(a, iter, epsilon)$ do
        # ...update my portion of global array for next iteration from $my\_array$ ...
        $compute(my\_array, id)$
        out ("$ready$", $iter + 1, id$)
        # Wait at the barrier
        for $i := 1$ to $N$ do
            rd("$ready$", $iter + 1, i, ?a[i]$)
        end for
        # Garbage collection; nobody could need it past the barrier
        if($iter > 1$)
            in("$ready$", $iter - 1, id, ?row\_t$)
        end if
        $iter := iter + 1$
        # ...read in some or all of global array for next iteration ...
    end while
end $worker$
```

Figure 4.17: Linda Barrier

```
procedure $init\_barrier$()
    $iter := 1$ # First iteration for workers
    # ...initialize the global array ...
    # Create the monitor processes
    for $host := 1$ to $num\_hosts()$ do
        $lpid := new\_lpid()$
        $f\_id := new\_failure\_id()$
        $create(monitor, host, lpid, f\_id)$
    end for
    # Create worker(id:1..N) and its registry tuple
    for $id := 1$ to $N$ do
        $lpid := new\_lpid()$
        $host := id \% num\_hosts()$
        out ($TSmain$, "$registry$", $host, id, iter$)
        $create(worker, host, lpid, id)$
    end for
end $init\_barrier$
```

Figure 4.18: FT-Linda Barrier Initialization

did not have to be reincarnated as long as its subtask was recovered, because it was not performing a computation that was assigned specifically to it. This is not the case with the worker in Figure 4.17, however. Here, for each iteration a given worker must update its portion of the global array for the next iteration and produce a "$ready$" tuple. Thus, if a worker fails, it must be reincarnated, i.e. it must continue executing exactly where it failed.

To accomplish this, we maintain a registry tuple similar to the registry tuple in the recoverable server of Section 4.1.2. This is used to record on which host a given worker is executing. This way, if the host fails, it can be ascertained which workers have failed and thus need to be reincarnated. However, in the case of the barrier example, this registry tuple also needs to record on which iteration the worker is currently working, so that if it fails, its reincarnation can do the correct computation and then proceed.

The FT-Linda solution outlined above is given in Figures 4.18, 4.19, and 4.20. Figure 4.18 gives the pseudocode to initialize a barrier. It creates the initial global data, the monitor processes, the registry tuples, and the workers. Figure 4.19 gives the worker. The main difference from the worker in Figure 4.17 is that when it deposits the "$ready$" tuple it also atomically increments the value of its current iteration in the registry tuple. In the event of a worker failure, this action ensures that a given iteration's "$ready$" tuple will be deposited by a worker exactly once. Also, when it begins, the worker does not assume it should start with iteration 1. Rather, it reads in its current iteration — the one it should start with — from its registry tuple. Finally, Figure 4.20 gives the monitor process. It is

```
process $worker(id)$
    # Initialize iter
    rd($TSmain$, "$registry$", $?host\_t, id, ?iter$) # if reincarnation $iter$ may be $> 1$
    # ...read in some or all of global array ...
    # Compute next iteration, unless answer has converged
    while not converged(a, iter, epsilon) do
        compute($my\_array, id$)
        # ...update my portion of global array for next iteration from $my\_array$ ...
        # Atomically deposit ready tuple for next iteration & update my registry
        ⟨
            out($TSmain$, "$ready$", PLUS($iter, 1$), $id$)
            in($TSmain$, "$registry$", $?host, id, iter$)
            out($TSmain$, "$registry$", $host, id$, PLUS($iter, 1$))
        ⟩
        # Barrier: wait until all workers are done with iteration $iter$
        for $i := 1$ to $N$ do
            ⟨ rd($TSmain$,"$ready$", PLUS($iter, 1$), $i, ?a[i]$) ⇒ skip ⟩
        end for
        if ($iter > 1$) then
            # Garbage collection on previous iteration
            ⟨ in($TSmain$,"$ready$", MINUS($iter, 1$), $id$) ⟩
        end if
        $iter := iter + 1$
        # ...read in some or all of global array for next iteration ...
    end while
end $worker$
```

Figure 4.19: FT-Linda Barrier Worker

```
process monitor(f_id)
   loop
      ⟨ in(TSmain, "failure", f_id, ?failed_host) ⇒ skip ⟩
      # Try to reincarnate all failed workers found on this host
      while ( ⟨ inp(TSmain, "registry" ,failed_host, ?id, ?iter) ⇒
                   out(TSmain, "registry", my_host, id, iter)
              ⟩ )   do
         # Create a new worker on this host
         lpid = new_lpid()
         create(worker, my_host, lpid, id)
         nap(some) # crude load balancing
      end while
   end loop
end monitor
```

Figure 4.20: FT-Linda Barrier Monitor

very similar to the monitor process for the recoverable server given in Figure 4.4, except that it must also include the current iteration in the registry tuple. Unlike the recoverable server example, however, this monitor process assigns a new LPID to a reincarnation of a failed worker. It does not need to use the same LPID, while the recoverable server's reincarnation had to be created with the same LPID to be able to access its state stored in a private (and stable) TS.

The same general techniques can also be used with the other kinds of barriers described above. In all cases, when a worker or coordinator process fails, it must be reincarnated so that it starts at the proper location. This is accomplished with the use of a registry tuple. A process must also use a placeholder tuple if it needs to perform a more general atomic action than a single AGS allows. For example, a coordinator process would leave a placeholder tuple for each synchronization tuple it received from a worker, while a process in the tree barrier solution would leave a placeholder when it withdraws the first signal tuple from a child. These placeholder tuples would then be moved or withdrawn from $TSmain$ at the final AGS for a given iteration, which would also increment the registry tuple.

The complete source code for this example is in Appendix G..

Using Fault-Tolerant Barriers for Systolic-Like Algorithms

Barriers are typically implemented on multiprocessors using physically shared arrays, although above we have shown how they can be implemented using Linda's TS, a shared memory that has been implemented on a wide range of architectures. Other kinds of parallel iterative algorithms are similar to barriers in that all workers work on the next iteration using the current values, then wait for all to complete the iteration before proceeding. However, parallel iterative algorithms such as systolic-like algorithms[3] use message passing, not shared memory, to synchronize. Problems that can be solved with such algorithms include matrix multiplication, network topology, parallel sorting, and region labelling [And91]. Fortunately, the FT-Linda barrier solution given above already uses Linda's blocking operations to synchronize, so its basic ideas apply directly to systolic-like and other data flow algorithms that use message passing.

Consider a systolic-like algorithm to multiply Arrays A and B to obtain the product array C, a (non-fault-tolerant) Linda example given in [Bjo92]. In this scheme, each worker is responsible for computing a given part of the result matrix C. Portions of A and B are streamed through the various workers so that, at a given step, each worker has the portions of A and B that are to be multiplied with each other. The worker accumulates its portion of C, and when done, deposits a tuple with this result. Thus, a worker iteration consists of the following steps:

1. Withdraw the next submatrices of A and B from the appropriate workers upstream.

2. Multiply them, accumulating the result in its portion of C (kept in TS).

3. Deposit the submatrices of A and B for consumption by the appropriate workers downstream.

Note that this is very similar to a barrier worker's iteration in Figure 4.19. The worker here uses two inputs rather than one, but this difference is obviously cosmetic. Indeed, the only significant difference is that the systolic-like worker actually consumes the tuples. Thus, the only structural change to use Figure 4.19 with systolic-like multiplication is to generate an $in\_progress$ tuple when withdrawing the submatrices, then atomically do Steps 2 and 3 in one AGS while also updating the registry tuple and withdrawing the $in\_progress$ tuples.

Finally, consider an optimization to this scheme. The key observation is that, in this systolic-like scheme that uses a message-passing paradigm, each worker produces tuples that are sent to a specific worker. The scheme above distinguishes a submatrix intended for a particular worker by placing its identifier in the tuple. However, as shown in Section 5.2, this can lead to severe contention. The problem is that the submatrix tuples being sent to all workers are on the same hash chain, even though a given worker can possibly match only a small fraction of them when withdrawing tuples intended for it. Unfortunately, the TS operations must still check all tuples on those hash chains. This contention can be greatly reduced by creating a stable and shared TS for each worker, where all submatrix tuples intended for that worker will be deposited. This way, the submatrix tuples are spread out

[3] We use the term 'systolic-like' rather than 'systolic' here because the workers in systolic algorithms run in strict lock step, generally enforced by hardware, while the workers here can get up to one iteration out of phase with each other.

among many TSs, and only submatrix tuples to be withdrawn by a given worker will be on the hash chain in that worker's TS. This eliminates the contention described above. Since TSs require on the order of a hundred bytes of memory, also as shown in Section 5.2, this scheme scales to a large number of workers.

4.3 Handling Main Process Failures

We have only been concerned so far in this dissertation with handling the failure of worker and monitor processes. We now consider how to handle the failure of a main process, the initial process in a Linda program that is created when the program is started. While the details are very application-dependent, the same general techniques used in handling the failure of a worker process can be applied to handle failures of a main process.

Main processes generally go through three phases: initialization, synthesis, and finalization. The initialization phase is relatively short, and includes creating monitor and worker processes, creating and seeding tuple spaces, etc. We assume that the entity that started the program will monitor its progress until the initialization phase is over and restart the program if the main process fails before initialization is complete. The synthesis phase involves any necessary reductions on the workers outputs. No synthesis may be necessary; for example, workers performing matrix multiplication deposit their results directly into TS. If synthesis is necessary, then a set of identical synthesis processes (and their monitor processes) can be created to perform it, so the failure of some of these processes can be tolerated. Of course, these processes must not have any windows of vulnerability, just like worker processes must not. An example of synthesizing results in fault tolerant manner is given below. Finally, in the finalization phase, an application generally reports the results of any synthesis to the entity that executed the program. If this can be done through TS, then this reporting step is not needed; it is often done in the synthesis phase. Sometimes, however, this is not sufficient, so the results of the synthesis have to be reported in some other fashion, e.g. writing to a disk or to a console. In either case, as assume that the entity that started the program will periodically ensure that the finalization process has not failed and restart it if it has.

The manner in which a synthesizing process is made fault-tolerant is very similar to that for a worker process. The main concern is not to leave any windows of vulnerability where data that may be needed later is represented only in volatile memory. Sometimes a synthesis operation can be done in a single AGS. In other cases, however, multiple AGSs and a placeholder tuple are required. As an example, consider the problem of finding the minimum cost (or distance) from among a number of searches (or matches). Here, the (bag-of-tasks) workers remove data tuples, calculate the cost associated with the data in the tuple, and deposit a cost tuple. The synthesis consists of finding the minimum cost from among the cost tuples. This can be accomplished in the manner shown in the example in Figure 4.21. There is no reason to leave a placeholder tuple here, because there is no window of vulnerability: this synthesis can be accomplished in a single AGS by using the **MIN** opcode. And, because there are no placeholder tuples, there is no need

```
# Record the lowest cost found among the cost result tuples.
# Initialization phase deposited tuple (“min_cost”,INFINITY).

while not done() do

    ⟨ in(TSmain, “cost”, ?cost) ⇒
        in(TSmain, “min_cost”, ?min_cost)
        out(TSmain, “min_cost”, MIN(cost, min_cost))
    ⟩

end while
```

Figure 4.21: Simple Result Synthesis

for monitor processes to recover from the failure of a synthesis process executing this code. Finally, note that the **in** in the body can never fail, because AGSs are executed atomically and the “min_cost” tuple is always present in TS outside any AGS.

For some synthesizing processes, however, the simple, single-AGS scheme in Figure 4.21 will not be sufficient, because they cannot perform the synthesis in a single AGS. For example, if the synthesis above had to not only record the cost, but also the identity of the data that resulted in that cost, then a single AGS is not powerful enough. In the example in Figure 4.21, the **MIN** opcode was sufficient to allow the recording of the lowest cost. There is no way, however, to record a corresponding identifier along with its cost in a single AGS. The way to do this is to use multiple AGSs and $in\_progress$ tuples, as shown in Figure 4.22. Here, the “$cost$” tuple to be synthesized is withdrawn, as well as the “min_cost” tuple to which it is compared; in both cases $in\_progress$ tuples are left as a placeholder. The “min_cost” tuple is then replaced atomically with removing the $in\_progress$ tuples. Of course, a set of monitor processes would be required to recover the “$cost$” and “min_cost” tuples from their placeholders in the event of a failure.

Note that, in practice, a synthesizer process would likely be more optimized than the one in Figure 4.22. For example, it would typically record the lowest cost it had processed so far in a local variable. If the cost discovered in the first AGS was not less than this, then most of the rest of the code in Figure 4.22 would not be executed; the “$cost\_in\_progress$” tuple would then simply been removed.

4.4 Summary

This chapter illustrates the flexibility and versatility of FT-Linda in two major domains, highly dependable computing and parallel applications. All examples demonstrate the usefulness of the AGS, in combination with monitor processes, to recover from a failure.

The replicated server example shows how FT-Linda can be used to construct a service with no fail-over time. It also demonstrates how distributed variables (the sequence tuple) can be used to order events in a Linda system. It further illustrates the usefulness of AGS

```
# Record the lowest cost found among the cost result tuples,
# and the ID associated with that cost.
# Initialization phase deposited tuple ("min_cost", INFINITY, NULL_ID).
while not done() do
    ⟨ in(TSmain,"cost", ?cost, ?id)  ⇒
        out (TSmain,"cost_in_progress", host, cost, id)
    ⟩
    ⟨ in(TSmain,"min_cost", ?min_cost, ?min_id)  ⇒
        out (TSmain,"min_cost_in_progress", host, min_cost, min_id)
    ⟩
    if(cost < min_cost) then
        # Update the min tuple with cost and id
        ⟨
            in(TSmain,"cost_in_progress", host, cost, id)
            in(TSmain, "min_cost_in_progress", host, min_cost, min_id)
            out(TSmain, "min_cost", cost, id)
        ⟩
    else
        # Restore the min tuple to its previous form
        ⟨
            in(TSmain,"cost_in_progress", host, cost, id)
            in(TSmain, "min_cost_in_progress", host, min_cost, min_id)
            out(TSmain, "min_cost", min_cost, min_id)
        ⟩
    end if
end while
```

Figure 4.22: Complex Result Synthesis

disjunction. This replicated server is an example of the state machine approach.

The recoverable server example explains how to implement a service with FT-Linda featuring no redundant server computations. It shows the usefulness of a private and stable TS to store key state for later recovery. Also, it displays how FT-Linda's oldest matching semantics can be used to avoid a client's starvation. This recoverable server is an example of the primary/backup technique.

The transaction facility demonstrates how multiple AGSs can be combined to provide a higher level of atomicity than what a single AGS gives the programmer. It requires only one AGS to either commit or abort a transaction, and thus demonstrates the usefulness of the tuple transfer primitives. This transaction facility is an example of the object/action paradigm.

The divide-and-conquer example shows how FT-Linda can be used in a more dynamic environment, where the size of subtasks can vary. It also demonstrates the utility of the tuple transfer primitives.

The barrier example illustrates how Linda's TS can be used to avoid problems endemic in multiprocessor implementations of barriers. It then shows how FT-Linda can implement fault-tolerant barriers in a simple and efficient manner. This technique also extends to systolic and other similar parallel iterative algorithms.

Finally, the recovery from main process failures is discussed. The techniques for achieving this are much like those used for worker processes.

CHAPTER 5

IMPLEMENTATION AND PERFORMANCE

A prototype implementation of FT-Linda has been built. All components have been separately tested, and are awaiting the completion of a newer version of the Consul communication substrate. The precompiler consists of approximately 15,000 lines of C code, of which approximately 3,000 was added to an existing Linda compiler to handle the extra constructs for FT-Linda. The FT-Linda runtime system consists of approximately 10,000 lines of C, not including the Consul communication substrate.

This section is organized as follows. It first gives an overview of the implementation. Next, it discusses the major data structures. After that, it describes FT-LCC, the FT-Linda C compiler, and the processing of atomic guarded statement (AGS) request messages by the TS managers. This section then discusses restrictions on the AGS, followed by initial performance results and optimizations. Finally, it describes future extensions to the implementation.

5.1 Overview

The implementation of FT-Linda consists of four major components. The first is FT-LCC, a *precompiler* that translates a C program with FT-Linda constructs into C generated code. The second is the *FT-Linda library*, which is linked with the object file comprising the user code. This library manages the flow of control associated with processing FT-Linda requests and contains a TS manager for TSs that are local to that process. The third is the *TS state machine*, which is an *x*-kernel protocol that sits beneath the user processes on each machine. This protocol contains the replicated TS managers, which are implemented using the state machine approach. Finally, the fourth part is Consul, which acts as the interface between user processes and the TS state machines, and the network. This collection of *x*-kernel protocols implements the basic functionality of atomic multicast, consistent total ordering, and membership. It also notifies the FT-Linda runtime of processor failures so that failure tuples can be deposited into the TS specified by the user application. While the implementation has been designed with Consul in mind, we note that any system that provides similar functionality (e.g., Isis [BSS91]) could be used instead.

The runtime structure of a system consisting of N host processors is shown in Figure 5.1. At the top are the user processes, which consist of the generated code together with the FT-Linda library. Then comes the TS state machine, Consul, and the interconnect structure. The prototype is based on workstations connected by a local-area network, although the overall design extends without change to many other parallel architectures. The

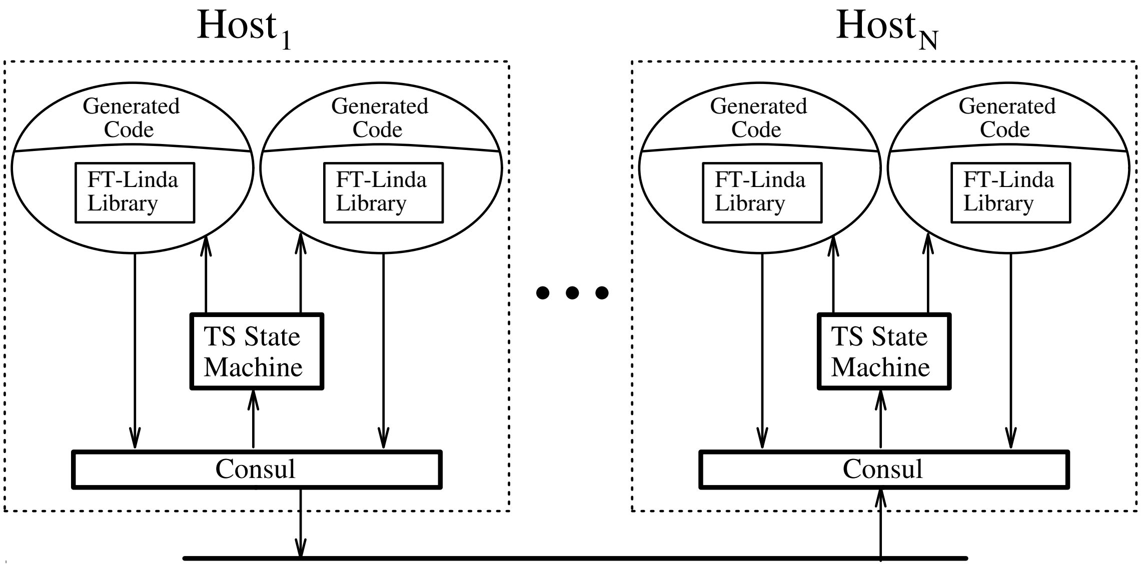

Figure 5.1: Runtime Structure

edges between components represent the path taken by messages to and from the network. Providing this message-passing functionality and the rest of the protocol infrastructure is the role of the *x*-kernel.

As noted, the implementation is currently nearing completion. All four parts have been implemented and tested, with final integration waiting the completion of a port of Consul to version 3.2 of the *x*-kernel. The final version will run on DEC 240, HP Snake, and other workstations under Mach, Unix, and stand-alone with the *x*-kernel.

5.2 Major Data Structures

There are two major categories of data structures in FT-Linda. The first is the single (complicated) data structure used by the user process to communicate a request to the TS managers. The second is the collection of data structures required to implement a TS. These two categories are discussed in turn.

The request data structure contains the information required to process requests such as TS operations in the user's address space, and to execute a state machine command at the appropriate TS managers. The most complex version is, of course, the request data structure associated with an AGS. In addition to general status information, this data structure contains an array of *branch data structures* for the branches of the AGS. This branch data structure contains an *op data structure* for the guard and an array of such data structures for the body. An op data structure contains copies of the TS handle(s) involved with the operation, space for a global timestamp, the operator (e.g., **in**), and its tuple index. It also contains the length of the operation's data and information for each parameter. The information for each parameter includes its polarity (i.e., actual or formal), the offset in

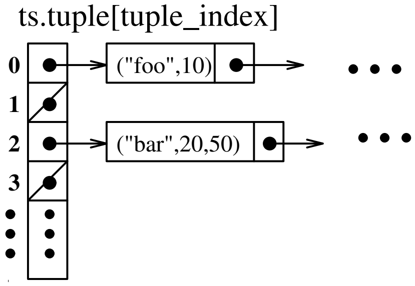

Figure 5.2: Tuple Hash Table

the data area of its actual or formal value (if any), and opcode arguments.

The request data structure also contains space for the value of any formal and actual parameters. Field `r_formal[]` is assigned values by a TS manager whenever an **in**, **inp**, **rd**, or **rdp** operation with a formal variable is executed; the formal values are later copied from `r_formal[]` into the user's variables by the GC. Field `r_actual[]` stores the value of actuals, and is assigned its values by the GC.

Once the invocation from the generated code to the FT-Linda library has been made, the control logic in the library routes the request appropriately. For example, if it involves only a local TS, it is dealt with by the TS management code within the library itself, while if it involves a replicated TS, it is multicast to the TS state machine protocols using Consul. In the latter case, this is the hand-off point between the user process and the control thread that carries the message through the *x*-kernel protocol graph to the network. As such, the user process is blocked until the request has been completed. When the request is completed, the request data structure is returned to the user process, where the generated code copies the values for any formals to the corresponding variables in the user address space.

Tuple spaces are implemented in two places, the FT-Linda library for local TSs and the TS state machine for replicated TSs. The algorithms and data structures used in both places are essentially identical. In each, a table of TS data structures is maintained. When a request to create a new TS is processed, a new table entry is allocated; the index of this entry is known as the *TS index*. The TS handle returned as the functional value of ts_create contains this index, as well as the attributes of the tuple space. A subsequent request to destroy a TS frees the appropriate table entry and increments an associated version number. This number is used to detect future TS operations on the destroyed TS.

A TS itself is represented as a hash table of tuples, as shown in Figure 5.2. The entries in the hash table are op data structures. This op data structure is thus used for both a TS operation in the AGS request data structure and for a tuple in TS. The difference between the two usages is that, for efficiency's sake, the values of all actuals in an AGS are stored in one field, `r_actual[]`. However, the TS manager matching a tuple in TS cannot access this field, because the AGS request will have been recycled.[1] Thus, when

[1] It is highly desirable to recycle the AGS data structure once it has finished execution, not leave it

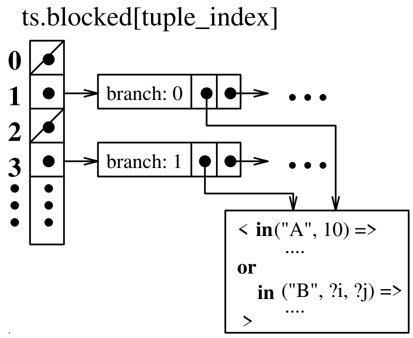

Figure 5.3: Blocked Hash Table

an op data structure is allocated for a tuple, extra room at the end is allocated for field `o_actual[]`. The value of the actuals are then copied from `r_actual[]` in the AGS into `o_actual[]` in the tuple.

The index into the hash table for a given tuple is simply the tuple index assigned by the precompiler modulo the hash table size. Linda implementations (including FT-Linda) generally try to ensure the table size is larger than the number of unique signatures (and hence different values of tuple indices), so a particular hash chain will contain only tuples from one signature. This reduces unnecessary contention, and is easy to accomplish in virtually all cases, because few Linda programs use more than a small number of signatures.

Also associated with each TS is another hash table used to store blocked AGS requests. That is, at any given time, this table contains requests with guards of **in** or **rd** for which no matching tuple exists. An example of such a table is shown in Figure 5.3. Here, the blocked AGS request has two **in** guards: one waiting for a tuple named A with a tuple index of 1 and the other waiting for a tuple named B with a tuple index of 3. Note that the request itself is stored indirectly in the table because, in the general case, such an AGS may have multiple guards and thus may be waiting for tuples with different signatures.

The C definitions of major FT-Linda data structures are given in in Appendix H. A TS data structure should have a hash chain for both tuples and blocked AGSs for each possible tuple signature. (This is required for efficiency, not correctness.) However, because most programs only use a few different TS signatures, and a hash chain takes little space, a TS takes on the order of a hundred bytes of space.

5.3 FT-LCC

The phases involved in building an FT-Linda program are as follows:

1. Run the C preprocessor (cpp) on the C FT-Linda code.

allocated until the last tuple that has data in `r_actual[]` has been removed from TS.

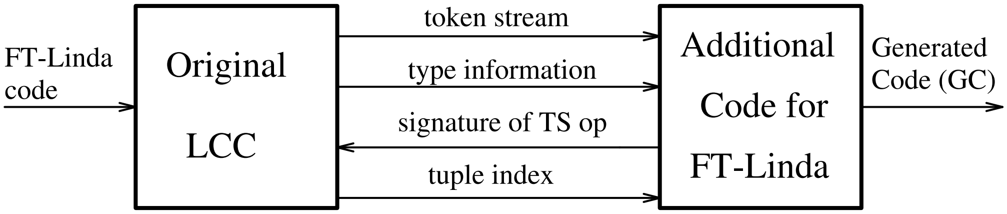

Figure 5.4: FT-LCC Structure

2. Run FT-LCC on this to generate C generated code (GC).
3. Compile the GC with a C preprocessor and compiler.
4. Link the object modules with the FT-Linda library to produce the executable program.

The first step is necessary because FT-LCC must process the code with the macro substitution and file inclusion completed. Next, FT-LCC generates the GC, which is subsequently compiled and linked with the FT-Linda library to produce the executable program. Note that the GC must be preprocessed in step 3, not just compiled, because it uses macros (especially constants) defined in auxiliary FT-Linda include files that the GC specifies.

The remainder of this section on FT-LCC is organized as follows. First, the internal structure of FT-LCC is described. A discussion of why FT-LCC is more complex than LCC is given, followed by a description of the different kinds of parameters FT-LCC must parse. This section concludes with an example of GC for a simple AGS.

FT-LCC Internal Structure

The FT-Linda precompiler, FT-LCC, is a derivative of the LCC precompiler [Lei89, LRW91]. The internal structure of FT-LCC is shown in Figure 5.4. The original LCC code handles everything not involving an FT-Linda construct, with the bulk of the additional code being dedicated to handling the AGS. This code receives from LCC a token stream and type information for those tokens, as shown in Figure 5.4. This stream is generally output unchanged until the opening '<' of an AGS is detected. At this time, the AGS-handling code parses the AGS and generates the code to implement the AGS. This GC marshalls the request data structure, passes it to the FT-Linda library, and then unmarshalls the AGS after its execution is finished. This unmarshalling consists mainly of assigning values to formal variables.

For each TS operation such as **in** that is included in the AGS, FT-LCC passes the signature of the op — its ordered list of types — to the LCC code, which returns the unique tuple index for that signature. This index is used in the GC to calculate the hash table entry for the index by taking its residue modulo the hash table size. This hash index is then stored in the AGS request structure and subsequently used by the TS managers for matching purposes.

FT-LCC Compared to LCC

FT-LCC is more complex than LCC mainly because of the requirement that operations in an AGS be executed atomically. The data in an AGS request structure is produced and consumed in the following phases (which are elaborated further below):

1. When the AGS is executed by the user code, the GC marshalls the request data structure with the information described above. The information placed into the request is the structure of the given AGS (number of branches, information about each op, etc.) and the values of actuals that are known. These include constants, which are known at compile time, and variables that are not formals also found earlier in the same branch. These values are placed into field `r_actual[]` of the request data structure. We call this phase *marshall time*.

2. The local and replicated TS Managers process the request. We call this phase *TS manager time*. Here the values of actuals in `r_actual[]` are used, and the values of formal variables are placed into `r_formal[]` from a matching tuple. If a formal variable appears later in the branch as an actual variable, then its value is read from `r_formal[]`.

3. After the processing of the AGS request by the TS managers is finished, the FT-Linda library returns control to the GC. The GC then assigns the formal variables set by the AGS with the values from `r_formal[]`. We call this phase *formal assignment time* or *unmarshalling time*.

As an example to help motivate why FT-LCC is more complicated than LCC, consider the following Linda sequence for a distributed variable:

```
in("count", ?count)
out("count", count + 1)
```

In a Linda program, the **in** and **out** operations are separate invocations to the Linda runtime system. Thus, even in a distributed implementation of Linda, the value $count + 1$ would be known at marshall time and thus disseminated appropriately along with the other information needed to implement the **out** operation in TS. However, consider the FT-Linda equivalent to the above fragment:

```
⟨ in(TSmain, "count", ?count) ⇒
   out(TSmain,"count", PLUS(count, 1))
⟩
```

Here, $count$ is an actual variable in the **out**. However, its value is not known until TS manager time, when the **in** in the AGS is processed at the TS State Machines and its value

Case	Tokens	param_t value	Polarity
1	typename	P_TYPENAME	1
2	? typename	P_TYPENAME	0
3	? varname	P_FORMAL_VAR	0
4	varname	P_VAL	1
5	varname	P_FORMAL_VAL	1
6	value	P_VAL	1
7	OPCODE$_i$ (args)	P_VAL	1
8	OPCODE$_i$ (args)	P_OPCODE$_i$	1

Table 5.1: FT-Linda Parameter Parsing

is placed into `r_formal[]`. FT-LCC thus must note whether an actual's value is known by marshall time or whether its value is not known until TS manager time, because the GC and TS managers must process these cases differently.

FT-LCC Parsing Cases

FT-LCC can ascertain whether or not the value of an actual will be known at marshall time by the actual's syntax and by recording which variables have been used as formal variables in the current branch. Recall that FT-Linda does not allow function calls or expressions as such arguments; only constants, variables, and opcodes may appear. The specific cases encountered when parsing an FT-Linda operation are given in Table 5.1. FT-LCC has to track whether a variable has been used in a given branch in a number of these cases, as described below. For each parameter, then, it tracks the kind of parameter it is with values of type `param_t`, as well as the polarity, which indicates whether the parameter is a formal (polarity 0) or an actual (polarity 1).

The specific cases in Table 5.1 are as follows:

1. A typename used as an actual.
2. A typename used as a formal.
3. A formal variable. This will be noted so that further usage of this variable in the branch can be diagnosed properly; specifically, to distinguish between Cases 4 and 5 and between Cases 7 and 8. When a matching tuple is found for this operation, the value from the corresponding field in the matching tuple is placed into `r_formal[]` if it is an **in**, **rd**, **inp**, or **rdp** (i.e., not for **out**, **move**, or **copy**). This value is used by the GC at formal assignment time to set the formal variable's value. Also, this value in `r_formal` is used if the variable is used later in the branch as an actual variable (i.e., Case 5).
4. A variable used as an actual that was not a formal previously in the branch. The value of this actual is stored into `r_actual[]` by the GC at marshall time.

5. A variable used as an actual that was a formal previously in the branch. The value for this actual is not known until TS manager time, when the TS operation in which the variable was a formal is performed. Its value is fetched from `r_formal[]` and used for the operation.

6. An actual whose value is specified by a literal value. This value is known at compile time and is stored into `r_actual[]` by the GC when marshalling the request.

7. An opcode (e.g., **PLUS**($count$,1)) whose parameters' values are all either Cases 4 or 6, i.e. not Case 5. This means that the values of these parameters are all known at marshall time. At this time, then, the GC will evaluate the opcode and place its result into r_actual, and indeed it will be treated just like an actual from Cases 4 or 6.

   Each opcode has its own value of type `param_t`. Current opcode values are `P_OP_MIN`, `P_OP_MAX`, `P_OP_MINUS`, and `P_OP_PLUS`, corresponding to opcodes **MIN**, **MAX**, **MINUS**, and **PLUS**, respectively.

8. An opcode with at least one parameter of Case 5. The values of the parameters are thus not all known when the GC is marshalling the AGS request. Thus, the opcode will have to be evaluated by the TS manager that executes this operation.

Cases 4–8 may optionally be preceded with a C type cast, i.e., a type surrounded by parentheses. Also, as denoted by the polarity column in Table 5.1, Cases 2 and 3 are formal parameters while the other are actual parameters.

As an example, the essential portion of the GC produced for the following distributed variable update:

```
⟨ in(TSmain, “count”, ?count) ⇒
    out(TSmain,“count”, PLUS(count, 1))
⟩
```

is given in Figures 5.5 and 5.6. Figure 5.5 gives the skeleton for the GC. It contains four main phases:

1. Initialize fields that only have to be initialized once (this will be explained further below in the section on optimizations). The innermost part of this section has been elited from Figure 5.5, and is given in Figure 5.6.

2. Initialize fields that have to be initialized each time the AGS is executed. In this case, only the address of $count$ need be initialized here.

3. Pass the request data structure to the FT-Linda library, which will pass it to the TS managers.

```
{   /* start of AGS 1 statement starting at count.c:28 */
#define AGS_NUM   1

   register int gc_i; static int gc_init_done = 0;
   /* other variables defined local to this AGS scope omitted for brevity */

   if (!gc_init_done) {    /* Initialize the one-time fields */

      register op_t *gc_op;    /* current op within br */
      gc_init_done = 1;

      strncpy(gc_req→r_filename, "count.c", MAX_FILENAME);
      gc_req→r_filename[MAX_FILENAME] = '\0';
      gc_req→r_starting_line = 28;

      gc_req→r_kind = REQ_AGS;

         /* branch #0 pre-processing */
         /* ....  See Figure 5.6 */
         /* end of branch 0 */

      /* request post-processing */
      gc_req→r_num_branches = 1;
      gc_req→r_next_offset = 8;   /* r_next_offset aligned 4 ==> 8 */

      rts_request_init(gc_req);

   }  /* !gc_init_done – stuff initialized once */

   /* These have to be filled in with each AGS call, not just once. */
   gc_formal_ptr→f_addr[0][0] = (void *) &(count);

   /* Pass the request to the FT-Linda library, which will pass it to the TS managers. */
   ftlinda_library( (void *) gc_req, &gc_reply, AGS_NUM);

   /* Code to fill in formal variables from
    * gc_formal_ptr->f_addr[branch][formalnum] omitted. */

#undef AGS_NUM   /* 1 */

}  /* end of AGS 1 statement starting at count.c:28 */
```

Figure 5.5: Outer AGS GC Fragment for $count$ Update

```
/* branch #0 pre-processing */
gc_br = &(gc_req→r_branch[0]);

  /* guard for branch 0 preprocessing */
  gc_op = &(gc_br→b_guard);
  gc_br→b_guard_present = TRUE;
  gc_op→o_optype = OP_IN;

  memcpy(&(gc_op→o_ts[0]),&(TSmain),sizeof(ts_handle_t));  /* copy TS handle */
    /* Parameter 0, case 1 */
    gc_op→o_param[0] = P_TYPENAME;
    /* Parameter 1, case 3 */
    gc_op→o_param[1] = P_FORMAL_VAR;
    gc_op→o_idx[1] = 0; /* count is formal 0 for this branch (#0) */
    gc_br→b_formal_offset[0] = 0; /* formal #0 (count) is at r_formal[0..3] */

  /* guard post-processing */
  gc_op→o_polarity = 0xfffffffd; gc_op→o_arity = 2;
  gc_op→o_type = 0; gc_op→o_hash = 0;    /* ( 0 & (MAX_HASH-1)) */

  /* body[0] preprocessing */
  gc_op = &(gc_br→b_body[0]);
  gc_op→o_optype = OP_OUT;

  /* read in the 1 TS handle for out */
  memcpy(&(gc_op→o_ts[0]),&(TSmain),sizeof(ts_handle_t));
    /* Parameter 0, case 1 */
    gc_op→o_param[0] = P_TYPENAME;
    /* Parameter 1, case 8 */
    gc_op→o_param[1] = P_OP_PLUS;
    /* Some complicated code to describe  PLUS(count,1) is omitted
     * for clarity. It has to note that opcode parameter count is Case 5
     * and thus not known until TS manager time,
     * while opcode parameter '1' is known at marshall time.  */
  /* body[0]  post-processing */
  gc_op→o_polarity = 0xffffffff; gc_op→o_arity = 2;
  gc_op→o_type = 0; gc_op→o_hash = 0;    /* ( 0 & (MAX_HASH-1)) */

/* branch #0 post-processing */
gc_br→b_body_size = 1;
/* end of branch 0 */
```

Figure 5.6: Inner AGS GC Fragment for $count$ Update

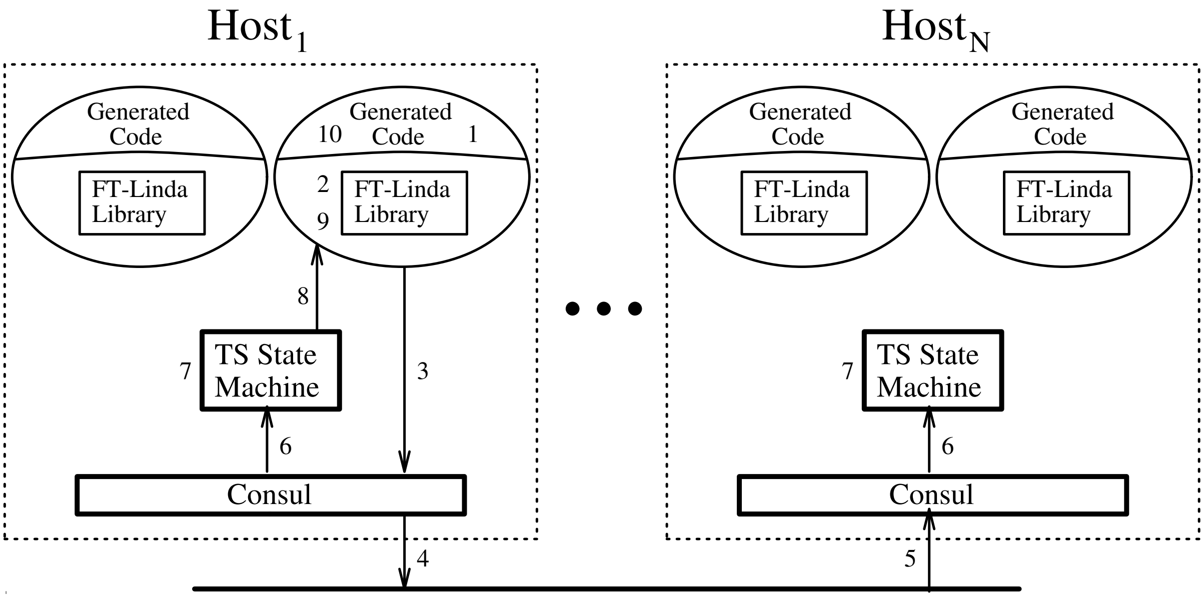

Figure 5.7: AGS Request Message Flow

4. Fill in the formal variables in the code (in this example, only $count$) from field `r_formal[]` in the request data structure.

Finally, Figure 5.6 contains the code to marshall the op data structures involved with this AGS.

5.4 AGS Request Processing

The processing of an AGS request is the most fundamental and important function the runtime system performs. We will discuss it in a general fashion first, then look at specific examples of processing particular AGSs.

5.4.1 General Case

Atomic guarded statements are clearly the most complicated of the extensions that make up FT-Linda. They include provisions for synchronization, guarantee atomicity, and allow TS operations in which multiple TSs are accessed. To demonstrate how these provisions impact the implementation, here we discuss the way in which requests generated by such statements are handled within the implementation. This discussion also serves to highlight how the components of the system interact at runtime.

The steps involved in processing a generic AG statement are illustrated in Figure 5.7. These can be described as follows.

1. The generated code fills in the fields of the request data structure that describes the AGS, and then invokes the FT-Linda library.

2. The code for managing local TS within the library executes as many of the TS ops in the AGS as possible. If such an operation withdraws or reads a tuple, the values for any formals in the operation are placed in the request data structure; this ensures that later operations that access these formals have their values. Processing of this AGS stops if a local TS operation is encountered that depends on data or tuples from a replicated TS operation earlier in the statement; more on this below.

3. The AGS request is submitted to Consul's ordered atomic multicast service.

4. Consul immediately multicasts the message. Lost messages are handled transparently to FT-Linda by the multicast service within Consul.

5. The message arrives at all other hosts.

6. Some time later Consul passes the message up to the TS state machine. The order in which messages are passed up is guaranteed to be identical on all hosts.

7. Each TS state machine executes all TS operations in the AGS involving replicated TSs. As in step 2, if such an operation withdraws or reads a tuple, the values for any formals in the operation are placed into the request data structure. If the request has blocking guards with no matching tuples, then it is stored in the blocked hash table until a matching tuple becomes available.

8. The TS state machine on the host that originated the AGS returns its request data structure to the FT-Linda library code. Note that this step and the remaining ones are only executed on the processor from which the AGS originated, because the replicated TSs are now up to date.

9. The library code managing local TSs executes any remaining TS operations in the AGS request. The original invocation from the generated code to the FT-Linda library then returns with the request data structure as the result.

10. The generated code copies values for formals in the AGS into the corresponding variables in the user process. The process can now execute the next statement after the AGS.

Thus, the processing of an AGS can be viewed as three distinct phases: processing of local TS operations, then dissemination and processing of replicated TS operations, and finally, any additional processing of local TS operations. This paradigm is responsible for the restrictions on the way in which TS operations are used in the body of an AGS that were mentioned in Section 3.2.1. For example, it would be possible to construct an example in which the data flow between TS operations would dictate going between local TSs and replicated TSs multiple times. Our experience is that such situations do not occur in practice, and so such uses have been prohibited. These and other restrictions are discussed further in Section 5.5.

Finally, we note that all the steps above may not be needed for certain AGS requests. For example, if the AGS statement does not involve replicated TSs, then the request will not be multicast to the TS state machines, and therefore steps 3 – 9 above will not be executed. As another example, if the request consists solely of **out** operations, the user process will not wait for a reply.

5.4.2 Examples

To make the processing of AGS requests more concrete, this section examines some specific examples in detail. In the following, let $scratch\_tsidx$ be the index of a local TS $TSscratch$, $main\_tsidx$ be the index of the replicated TS $TSmain$, and $tidx$ be the tuple index of the tuple or template of the operation in question. The tasks performed by the generated code in each case are identical to the above, and so are omitted.

Local Case

First consider an example that involves only a local TS:

```
⟨ true ⇒
   out(TSscratch, “foo”, i)
   out(TSscratch, “foo”, j)
   in(TSscratch, “foo”, ?k)
⟩
```

The generated code passes the request to the FT-Linda library where local TSs are implemented. To perform the **out** operations, this code creates a tuple using the values from the operations. It then attaches the tuple to the appropriate hash chain in the tuple hash table, i.e., $TS[scratch\_tsidx].tuple[tidx]$. The **in** is then executed. In doing so, the oldest matching tuple in TS will be withdrawn; in this case, it would be the tuple deposited by the first **out** in the AGS, assuming no matching tuples existed prior to execution of this statement. After all operations in the body have been executed, the request data structure is returned to the generated code, where the value for k is copied into k's memory location in the user process's address space.

Single Replicated Operation

Consider now a lone TS operation on a replicated TS:

```
⟨ in(TSmain, “foo”, i, ?j) ⇒ skip ⟩
```

After the request data structure is passed to the FT-Linda library code, it is immediately multicast to all TS state machines. Upon receiving the message, each state machine first checks for a matching tuple in the tuple hash table entry $TS[main\_tsidx].tuple[tidx]$. If this entry is not empty, the first matching one on this list—that is, the oldest match—is dequeued. If no such matching tuple is found, then the request is stored on the blocked queue for the guard, $TS[main\_tsidx].blocked[tidx]$. In either case, once a matching tuple arrives, the **in** is executed, and the matching tuple used to fill in the request data structure with the value of j. The state machine on the originating host then returns the data structure back to the user process.

Both Local and Replicated Tuple Spaces

Now consider a case involving both local and replicated TSs:

```
⟨ true ⇒
  in($TSscratch$, “$foo$”, $?i$, 100)
  in($TSmain$, “$bar$”, $i$, $?j$)
  rd($TSscratch$, “$foo$”, $j$, ?k)
⟩
```

The request data structure is first passed to the code implementing local TSs in the FT-Linda library. As many operations as possible are then executed, which in this case is only the first **in**; the subsequent local **rd** cannot be executed because it depends on the value for j, which will not be present in the request data structure until later. In processing this local **in**, the new value for i retrieved from the matching tuple is copied into the request data structure. Of course, neither this **in** nor any other operation in the body may block.

Next, the request is passed to Consul, which transmits it by multicast to all machines hosting copies of the TS. Some time later each TS state machine gets the request message. At this point, each state machine removes the oldest tuple that matches the second **in** and updates the value for j in the request. Note that, to find this match, the value used for i is taken from the request data structure because its value was assigned earlier within the AGS.

Following execution of replicated TS operations, the remaining local TS operation is performed on the host on which the AGS originated. To do this, the request data structure is first passed back to the user process. The **rd** operation is then executed; again, the value of j used for matching is taken from the request data structure. The value of the matching tuple is used to fill in the value for k before the request data structure is returned to the generated code. There the new values of i, j, and k are copied from the request data structure into their respective variables in the user process's address space.

Move Operations

The **move** operation is treated as a series of **in** and **out** operations, as illustrated by the following example:

```
⟨ true ⇒
   move(TSscratch, TSmain)
   in(TSmain, "foo", ?i)
⟩
```

In this example, the generated code invokes the entry point in the FT-Linda library, which in turn invokes the local TS code. There, all tuples in $TSscratch$ are removed, and the **move** replaced in the request data structure by an **out**($TSmain, t$) for each such tuple t in $TSscratch$. When the TS state machines receive this request, they execute these **out** operations and then the final **in**.

If the **move** had been a **copy** instead, the only difference would be that the tuples are copied from $TSscratch$ rather than removed. Templates in such tuple transfer operations are handled using the same matching mechanism as for normal TS operations.

AGS Disjunction

Consider an AGS involving disjunction, as in the following:

```
⟨
   in(TSmain, "ping", 1) ⇒
      ...
or
      ...
   in(TSmain, "ping", n) ⇒
⟩
```

The actions taken here are similar to earlier examples until processing reaches the TS state machines. When the state machines receive this request, they find the oldest matching tuple for each guard. If no such tuple exists, a stub for each branch is enqueued on $TSmain$'s blocked hash table. If there are matching tuples, the oldest among them is selected and the corresponding branch processed as described previously. Note that the oldest matching semantics implemented by FT-Linda are important here because it is this property that guarantees all state machines choose the same branch.

Unblocking Requests

An **out** operation may generate a tuple that matches the guards for one or more blocked requests. Consider the following.

```
⟨ true ⇒ out(TSmain, “foo”, 100, 200) ⟩
```

First, the tuple generated by the **out** is placed at the end of the appropriate hash chain in the TS data structure, i.e., the chain starting at $TS[main\_tsidx].tuple[tidx]$. Then, the TS state machines determine if there are matching guards stored on the analogous hash chain in the blocked hash table, i.e., $TS[main\_tsidx].blocked[tidx]$. If so, they take them in chronological order and schedule any number of **rd** guards and up to one **in** guard to be considered for execution, along with their bodies, after the current AGS is completed.

5.5 Rationale for AGS Restrictions

Now that FT-Linda's language features, usage, and implementation have been examined, we can better motivate the design decisions leading to the AGS restrictions mentioned above. These restrictions involve dataflow, blocking in the body, expressions and function calls as parameters, and conditional execution in the body.

5.5.1 Dataflow Restrictions

As noted in Section 5.4.1, the three phase process (Steps 2, 7, and 9) of executing TS operations in atomic guarded statements leads to restrictions based on data flow between formals and actuals in the statement. Informally, any AGS that cannot be processed in these three steps is not allowed. Note that these steps involve local, replicated, and local TSs, respectively.

Following is an example of an AGS that does not meet these criteria and therefore, is disallowed:

```
⟨ true ⇒
   in(TSscratch, “foo”, ?i, 100)
   in(TSmain, “bar”, i, ?j)
   rd(TSscratch, “foo”, j, ?k)
   out(TSmain, “bar”, k, ?l)
⟩
```

The data flow — local to replicated to local to replicated — violates the three phase processing. If this were permitted, efficiency would suffer, and handling failures and concurrency would be more complicated. Efficiency would suffer because it is not possible to process this AGS with only one multicast message to the replicated TS managers. Also, handling failures is more complicated. In this case, the host with $TSscratch$ could fail between the **in** and **rd** involving $TSscratch$. Since $TSscratch$ would thus no longer be available, the replicated TS managers would have to be able to undo the effects of **in** on $TSmain$ to cancel the AGS. Further, to implement this more general AGS would add complexity to the replicated TS managers. They would have to take additional measures too ensure that this AGS appears to be atomic to other processes, i.e. so no other AGS could access *TSmain* between the **in** and the **out** operations.

The exact nature of the dataflow restrictions depend on the particular operation, but are based on whether the compiler and runtime system can somehow implement the AGS in the three phases. For example, the following code from Figure 3.4 is permissible:

```
⟨ in(TSmain, “in_progress”, my_hostid, subtask_args) ⇒
    move (TSscratch, TSmain)
⟩
```

In this case, the **move** is converted in step 2 into a series of **in** operations that read from *TSscratch* and corresponding **out** operations that deposit into *TSmain*. Matching tuples will therefore be removed from *TSscratch* in step 2 and then added to *TSmain* in step 7 after the guard is executed.

5.5.2 Blocking Operations in the AGS Body

No operation in the body of an AGS is permitted to block, as discussed in Section 3.2.1. With this restriction it is much simpler to implement the AGS's atomicity in the presence of both failures and concurrency. Once a guard has a matching tuple in TS, then both the guard and the body can be executed without the need to process any other AGS. This is a simple and efficient way to provide atomicity with respect to concurrency.

Allowing the body to block causes many of the same sorts of difficulties described above with regard to the three-phase rule. Indeed, the code fragment given above that violated this rule did block between the second **in** and the **rd**, as far as the replicated TS managers are concerned. That is, they had to stop processing after the second **in** until the **rd** had been performed in *TSscratch*. The difficulties this caused are very similar to the difficulties causes by allowing blocking in the body. In both cases, more messages are required, and the TS managers have to do more work to be able to handle failures and concurrency in the middle of an AGS.

5.5.3 Function Calls, Expressions, and Conditional Execution

FT-Linda does not allow a function call or an expression to be a parameter to a TS operation, as mentioned in Chapter 3 and described further in Section 5.3. It also does not allow any sort of conditional execution within an AGS, apart from which guard is chosen in an AGS. The reasons for these restrictions will now be described in turn. A common denominator in these restrictions is that their absence would make FT-Linda harder to understand, program, and implement.

A parameter to an FT-Linda operation may not contain a function call or an expression, as shown in Table 5.1 in Section 5.3; the only form of computation allowed in an AGS is the opcode. Allowing a function call in an AGS would be allowing an arbitrary computation inside a critical section. This would degrade performance, because each TS manager could not process other AGSs while this computation was taking place. Also, any such functions would have to have restrictions on them; for example, they would have to be free of side-effects, and they could not reference pointers. This is required to maintain reasonable semantics, because the functions would be executed on all the machines hosting the TS replicas, rather than just on the user's host. Similarly, any expressions would have to have restrictions on them, because some expressions would have to be evaluated in a distributed fashion with values obtained from the replicated TS managers. While there may be a way to define a reasonable usage of expressions and functions in FT-Linda parameters, we feel it would be difficult to explain cleanly. It would also be difficult to rationalize why a slightly more general form should not be permitted.

Similarly, one could imagine permitting some form of loops or conditional statement in an AGS. To be useful, however, many ways one could envision using these would require some form of variable assignment. For example, variable assignment would be used to store a return value from **inp**, if it were allowed in the body, something that makes sense if one is to allow varaible assignment. It thus would become quite difficult to add such conditional statements in ways that could be cleanly described, efficiently implemented, and yet did not beg for more functionality.

5.5.4 Restrictions in Similar Languages

Other researchers have found it necessary or at least desireable to impose restrictions similar to those discussed above. As mentioned in Section 3.4, one optimization in the Linda implementation described in [Bjo92] collapses an **in** and an **out** into one operation at the TS manager. However, in order for the compiler to be able to apply this optimization, the **in** and **out** have to use only simple calculations very similar to FT-Linda's opcodes. Of course, this is applicable to many common Linda usages, most notably a distributed variable.

A second example is Orca, a language that is useful for many of the same kinds of applications [BKT92, Bal90, TKB92, KMBT92]. The language is based on the shared-object model and has been implemented on both multiprocessors and distributed systems. An Orca object consists of private data and external operations that it exports. Operations consist of a guard (a boolean expression) and a series of statements that may modify the object's private data. An operation is both atomic and serializable, as is FT-Linda's AGS.

To achive these semantics, Orca's designers have placed some restrictions similar to thost described above for FT-Linda. For example, only the guard may block. Also, the guard must be free of side effects. These restrictions are crucial in allowing the reasonable implementation of the atomicity properties of Orca's operations.

5.5.5 Summary

The above restrictions allow the AGS to be implemented with reasonable semantics and performance in the presence of failures and concurrency. And, as we have demonstrated in the examples in Chapters 3 and 4, these restrictions do not appear to adversely affect FT-Linda's usage; they still allow FT-Linda to be useful for a variety of applications. Finally, other researchers have also found it necessary to impose similar restrictions to achieve similar goals.

5.6 Initial Performance Results

Some initial performance studies have been done on the FT-Linda implementation. As noted, the runtime system has not yet been merged with Consul, so the measurements capture only the cost of marshalling, the AGS, performing its TS operations at the TSs involved, and then unmarshalling the AGS. In the tested version, the control flow from the library to the state machine was implemented by procedure call rather than the *x*-kernel. As such, only one replica of the TS state machine was used.

Table 5.2 gives timings figures for a number of different machines. The first result column is for an empty AGS, while the next give the cost of incrementing a distributed variable. Subsequent columns gives the marginal cost of including different types of **in** or **out** operations in the body. We note that the i386 figures are comparable to results reported elsewhere [Bjo92]. This is encouraging for two reasons. First, FT-Linda is largely unoptimized, while the work in [Bjo92] is based on a highly optimized implementation. Second, we have augmented the functionality of Linda, not just reimplemented existing functionality.

These figures can be used to derive at least a rough estimate of the total latency of an AGS by adding the time required by Consul to disseminate and totally order the multicast message before passing it up to the TS state machine. For three replicas executing on Sun-3 workstations connected by a 10 Mb Ethernet, this dissemination and ordering time has been measured as approximately 4.0 msec [MPS93a]. We expect this number to improve once the port of Consul to a faster processor is completed.

We note that even these relatively low latency numbers overstate the cost involved in some ways. A key property of our design is that TS operations from an AGS in one user process on a given processor can be executed by the TS state machine while those from other processes on the same processor are being disseminated by Consul. This concurrent processing means that, although the latency reflects the cost to an individual process, the overall computational throughput of the system is higher because other processes can continue to make progress. In other words, the latency does not necessarily represent wasted time, because the processor can be performing user-level computations or disseminating other AGSs during this period. To our knowledge, this ability to process TS operations concurrently within a distributed Linda implementation is unique to FT-Linda.

The above performance figures support the contention that the state machine approach

	empty		cost per body op						
Machine	AGS	**in-out**$_0$	**out**$_0$	**out**$_1$	**out**$_2$	**in**$_0$	**in**$_1$	**in**$_2$	**in**$_3$
SparcStation 10	4	31	25	28	28	8	9	20	19
HP Snake	4	33	23	26	26	10	11	23	23
SparcStation IPC	14	123	77	88	90	22	25	61	64
i386 (Sequent)	30	300	147	176	184	91	120	270	264

empty AGS $\equiv \langle$ **true** $\Rightarrow$ **skip** $\rangle$
in-out$_0$ $\equiv \langle$ **in**($TSmain$,“$TEST$”, $?i$) $\Rightarrow$ **out**($TSmain$,“$TEST$”, **PLUS**(i,1)); $\rangle$
out$_0$ $\equiv$ **out**($TSmain$,“$TEST$”)
out$_1$ $\equiv$ **out**($TSmain$,“$TEST$”, $1, 2, 3, 4, 5, 6, 7, 8$)
out$_2$ $\equiv$ **out**($TSmain$,“$TEST$”, a, b, c, d, e, f, g, h)
in$_0$ $\equiv$ **in**($TSmain$,“$TEST$”)
in$_1$ $\equiv$ **in**($TSmain$,“$TEST$”, $?int, ?int, ?int, ?int, ?int, ?int, ?int, ?int$)
in$_2$ $\equiv$ **in**($TSmain$,“$TEST$”, $1, 2, 3, 4, 5, 6, 7, 8$)
in$_3$ $\equiv$ **out**($TSmain$,“$TEST$”, $?a, ?b, ?c, ?d, ?e, ?f, ?g, ?h$)

Table 5.2: FT-Linda Operations on Various Architectures (μsec)

is a reasonable way to implement fault tolerance in Linda. Although a more thorough analysis is permature at this point, our speculation is that this approach will prove competitive with transactions, a common way to achieve fault tolerance with dependable systems, and checkpoints, the technique most widely used to achieve fault tolerance in scientific programs. None of these three fault-tolerance paradigms have been developed and deployed fully enough with Linda to make quantitative comparisons about their strengths and weaknesses. Fortunately, active research is being performed in all three areas, so hopefully in the near future we will be able understand better the performance and tradeoffs of the different approaches.

5.7 Optimizations

As already noted, the TS managers and GC have been only slightly optimized. However, one optimization has provided great benefits, and two other planned optimizations would also be beneficial.

The key observation for the first optimization is that most of the information in an AGS request data structure does not change between different executions of the AGS. Examples of items that do not change include the number of branches and the signature of each operation. In fact, only two items can vary between different executions of the same AGS by the same process. The first is the address of a stack variable used as a formal parameter (addresses of formal variables are used in the GC at formal assignment time).

This corresponds to Case 3 in Table 5.1, if the variable in question is a stack variable (an “automatic” variable in C and some other languages). The second is the value of a variable used as an actual or as an opcode parameter, and then only if this variable was not a formal earlier in the branch. This is Case 4 in in Table 5.2. Thus, with this optimization, the AGS request data structure is allocated and fully initialized in the generated code the first time it is executed for a given process. When the same AGS is then executed later only the values and addresses of the aforementioned variables are copied.

This optimization dramatically reduces the execution loads associated with the AGS command. For example, the times in Table 5.2 for the Sparcstation IPC were 1200 – 1500 μseconds higher—more than an order of magnitude—without this optimization. And it is useful not only for timing tests but also for real-world applications, because many AGS statements are in loops.

The second optimization is the network analogy of the first one. Since most of the information in an AGS request data structure does not change between invocations, the information that does not change only needs to be sent to its TS manager(s) once, and stored for future use. This greatly reduces the size of the AGS request that has to be sent over the network each time, and thus reduces the latency. It also eliminates the CPU time to copy the unchanging information. As noted previously, this optimization has not yet been implemented.

The third and final optimization is to make AGSs with all **out**s, **move**s, and **copy**s — a *write-only AGS* — not to delay the user any more than necessary. Currently, the user is blocked until the AGS has been executed by all pertinent TS managers. However, the AGS need only block the user until the GC has marshalled the its request data structure and submitted it to the FT-Linda library, because the user code receives no information from the AGS’s execution. Thus, it need not wait for this execution to occur.

5.8 Future Extensions

We hope to extend FT-Linda’s implementation to provide greater functionality and performance in a number of ways. These are discussed in the following sections. First, the reintegration of failed TS managers is discussed. Next, the replicating of TS managers to a subset of the hosts in an FT-Linda system is described. We conclude with a discussion of how network partitions can be handled.

5.8.1 Reintegration of Failed Hosts

The major problem in reintegrating a failed processor upon recovery is restoring the states of replicated TSs that were on that machine. Although there are several possible strategies, a common one used in such situations is to obtain the data from another functioning machine. To do this, however, requires not only copying the actual data, but also ensuring that it is installed at exactly the correct time relative to the stream of tuple operations coming over the network. That is, if the state of the TSs given to the recovering

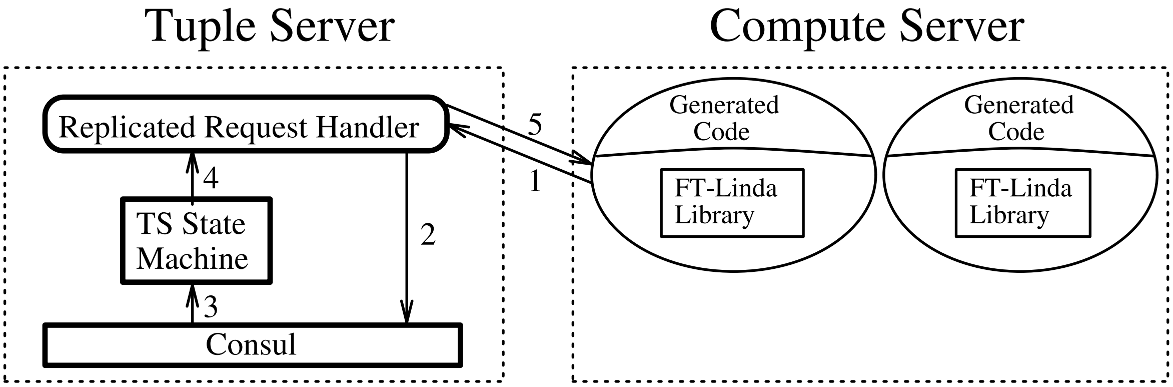

Figure 5.8: Non-Full Replication

processor P_i is a snapshot taken after AGS S_1 has been executed but before the next AGS, S_2, then P_i must know to ignore S_1 but execute S_2 when they arrive.

Fortunately, Consul's membership service provides exactly the functionality required [MPS93a]. When a processor P_i recovers, a *restart message* is multicast to the other processors, which then execute a protocol to add P_i back into the group. The key property enforced by the protocol is that all processors—including P_i—add P_i to the group at exactly the same point in the total order of multicast messages. This point could easily be passed up to the TS state machine and used as the point to take a snapshot of all replicated TSs to pass to P_i. Note that, to reintegrate a TS state machine, the hash tables for both tuples and blocked requests would need to be transferred. This general scheme for reintegrating failed hosts could also be used to incorporate new tuple servers into the system during execution.

5.8.2 Non-Full Replication

The FT-Linda implementation currently keeps copies of all replicated TSs on all processors involved in the computation. Using all processors is, however, unnecessary. We can designate a small number to be *tuple servers*, and use only these processors to manage replicated TSs. Each tuple server would maintain copies of all replicated TSs and would either be well-known or ascertainable from a name server. [CP89] User processes would execute on separate *compute servers*.

An organization along these lines would necessitate some changes in the way user processes interact with the rest of the FT-Linda runtime system. For example, Figure 5.8 demonstrates the differences in the processing of an AGS. Rather than requests being submitted to Consul directly from the FT-Linda library, a remote procedure call (RPC) [Nel81, BN84] would be used to forward the request to a *request handler process* on a tuple server. This handler immediately submits it to Consul's multicast service as before. Later, after the AGS has been processed by the TS state machines, the request handler on the tuple server that originally received the request sends the request back to the user process.

Failure of a compute server causes no additional problems beyond those present in complete replication, but the same is not true for tuple servers. In particular, if such a failure occurs, then any user process awaiting for a reply from that processor could potentially block forever. To solve this problem, the user process would time out on its reply port and resubmit the request to another tuple server. To prevent the request from being inadvertently processed multiple times, the user process would attach a unique identifier to the request, and the local TS logic and the TS state machines only process one request with a given identifier.

Note that tuple servers in this scenario are very much analogous to the file servers found in a typical workstation environment. Indeed, the way in which they would be used would likely be similar as well, with a few tuple servers and many compute servers. Of course, the tuple servers could have faster CPUs and more memory than the compute servers, as file servers typically do when compared to the client machines they serve.

This relatively small degree of replication for stable TSs that non-full replication allows should be sufficient for many fault-tolerant applications. It should also scale much better than full replication.

5.8.3 Network Partitions

A network partition occurs when hosts on a network that can normally communicate with each other cannot do so, typically due to the failure of a communication link or of a network gateway. The typical way to handle such a situation is to allow only those hosts that are in a majority partition to continue [ASC85]. This ensures that there will not be multiple, divergent versions of the data, because at most one partition contains a majority of the hosts. Hosts that are in the minority must wait until they are in the majority, update their copy of the replicated data, and then proceed.

The current implementation of Consul does not handle network partitions. However, it would be very easy to extend Consul to do so, as described in [MPS93a]. This change would be completely transparent to the FT-Linda implementation, assuming the scheme for reintegrating hosts discussed above had been implemented. Consul's membership protocol on a given host already keeps track of which hosts it can currently communicate with. Thus, it knows the size and membership of its partition. Additionally, this information is kept consistent with all other members in its partition. It is thus simple to extend this membership protocol to check whether or not its host is in the majority partition. If it is, it proceeds as normal. If not, it simulates a crash and then reintegrates upon rejoining the majority partition.

5.9 Summary

This chapter describes the details of the implementation of FT-Linda. An overview is given of the major components, followed by a description of the major data structures were then given; The FT-LCC precompiler is presented, followed by the details regarding the processing of an atomic guarded statement. The restrictions on the AGS are described next. Initial performance results and optimizations are then given, followed by a discussion of future planned extensions to the implementation.

The two major categories of data structures in FT-Linda's implementation are for request messages sent to the tuple space managers and for tuple spaces. The request messages specify operations that the tuple space managers perform, including creating and destroying tuple spaces and also the atomic guarded statement. These messages also include space for the values of actuals and formals to be recorded. A tuple space is simply two hash tables, one for tuples and the other for blocked atomic guarded statements.

The FT-LCC precompiler generates C code that includes generated code (GC) to marshall an AGS request data structure and pass this request to the FT-Linda library. It is more complex than its LCC (Linda) predecessor because it must deal with the atomic guarded statement's atomic properties, and some of the values involved in an atomic guarded statement are not known until the request is processed at the replicated tuple space managers.

The processing of an AGS takes place at a number of stages. It is marshalled by the GC, then operated on by the local TS manager. After this, it is multicast to all replicated TS managers, where it is executed at the same logical time. The AGS is then processed further at the local TS manager, and then the AGS returns control to the next statement in the program. The actual logic to process an atomic guarded statement at a given tuple space manager is virtually identical for local and replicated tuple spaces.

The atomic guarded statement has restrictions involving dataflow, blocking in the body, function calls, expressions; it also allows no conditional execution inside the body. The dataflow restrictions are necessary to allow efficient processing and multicasting of the atomic guarded statement. An operation in the body of an atomic guarded statement is not allowed to block for similar reasons. Function calls and expressions are not permitted in an atomic guarded statement because they would allow an arbitrary computation inside an atomic guarded statement. Also, the functions and expressions would have to have limitations on them that would be hard to explain cleanly and to justify why more general forms are not permitted. Finally, we note that another Linda implementation and a similar language, Orca, place restrictions similar to many of those that FT-Linda mandates.

Initial performance results are given in this chapter. The costs in processing tuple space operations are comparable to another Linda implementation. This is encouraging, because FT-Linda is largely unoptimized and it also has augmented the functionality of Linda.

The implementation has been optimized some, and future optimizations are possible. The information in an atomic guarded statement that does not change is only marshalled once, and a planned optimization will also only transmit this unchanging information over the network once. Also, write-only atomic guarded statements can be executed in the background while the user's code proceeds.

There are three ways in which we plan to extend the FT-Linda implementation. Support for reintegration of failed or new hosts can be provided directly by Consul’s membership service. Non-full replication can be achieved by having tuple servers on a subset of the machines. Finally, Consul can be extended in proven ways to allow replicated TS managers that can still communicate with the majority of their peers to continue operating in the face of network partitions.

CHAPTER 6
CONCLUSIONS

6.1 Summary

In this dissertation we have addressed the problem of providing high-level language support for fault-tolerant parallel programming. We have created a version of Linda, which we call FT-Linda, to permit the construction of fault-tolerant parallel programs. The distinguishing features of FT-Linda are its stable tuple spaces and atomic execution of multiple tuple space operations.

We surveyed Linda in Chapter 2. Linda has semantic deficiencies even in the absence of failures, including weak **inp/rdp** semantics and asynchronous **out**s. Linda applications also have problems in the presence of failures, most notably Linda's lack of tuple space stability and its single-operation atomicity. Common Linda paradigms such as the distributed variable and the bag-of-tasks paradigm also have problems in the presence of failures. Finally, we concluded this chapter by outlining alternative ways for implementing tuple space stability and multi-operation atomicity.

In Chapter 3 we presented FT-Linda. It provides mechanisms for creating multiple tuple spaces with different attributes. These attributes are resilience, which specifies a tuple space's failure behavior, and scope, which designates which processes may access a tuple space. FT-Linda has two provisions for atomic execution. The first is the atomic guarded statement (AGS), which allows a sequence of tuple space operations to be executed in an all-or-nothing fashion despite failures and concurrency. The AGS also has a disjunctive form that allows the programmer to specify multiple sequences of tuple space operations from which zero or one sequence is selected for atomic execution. The second provision for atomic execution is FT-Linda's tuple transfer primitives, which allow tuples to be moved or copied atomically between tuple spaces. FT-Linda features improved semantics: strong **inp/rdp** semantics, oldest matching semantics, and the sequential ordering property. In this chapter we also surveyed other efforts to allow Linda programs to tolerate failures. Finally, in Chapter 3 we discussed possible extensions to FT-Linda; these include additional tuple space attributes, nested AGSs, notification of which sequence of tuple space operations was executed (if a disjunctional AGS is used), tuple space clocks, tuple space partitions, guard expressions, and allowing the creation and destruction of tuple spaces to be included in AGSs.

In Chapter 4 we demonstrated FT-Linda's usefulness in constructing highly dependable systems and parallel applications. We gave three examples of dependable systems constructed with FT-Linda: a replicated server, a recoverable server, and a transaction facility. The replicated server features a server with no fail-over time — there is no recover phase to delay a client of a failed server — that uses a distributed variable tuple to maintain replica consistency. The recoverable server performs no redundant computations in the absence of failures, and saves its private state in a stable tuple space for recovery purposes. The transaction facility is a library of user-level procedures implemented with FT-Linda. This shows how AGSs can be used to construct a higher level abstraction. In particular, it demonstrates the power of FT-Linda's tuple transfer primitives.

Chapter 4 also presented two examples of using FT-Linda to construct fault-tolerant parallel applications: a divide-and conquer worker and a fault-tolerant barrier. The divide-and-conquer worker is a generalization of the bag-of-tasks example from Chapter 3. The fault-tolerant barrier solution also applies to another class of algorithms, systolic-like algorithms. We concluded Chapter 4 with a discussion of how to tolerate the failure of a main process.

We discussed the implementation and performance of FT-Linda in Chapter 5. The major data structures are for the AGS and the tuple spaces. The FT-LCC precompiler is a derivative of the Linda LCC precompiler; its main differences from LCC are due to the atomic nature in which multiple tuple space operations are combined in the AGS. An AGS is executed by marshalling the AGS data structure, sending it to the various tuple space managers for processing, and then unmarshalling the data structure (mainly copying the values of formal variables into their respective variables). The AGS has restrictions similar to those in similar parallel languages. The cost per processing an FT-Linda operation by the tuple space managers is competitive with other Linda implementations. Optimizations to FT-Linda's implementation include marshalling and transmitting only once the unchanging information in an atomic guarded statement. Future extensions to FT-Linda include reintegration of failed hosts, non-full replication, and tolerating some network partitions.

6.2 Future Work

This work can be expanded in many different directions. These include completing and extending FT-Linda's implementation, porting it to other environments, extending the language, investigating FT-Linda's use as a back-end language, and considering a real-time variant of FT-Linda.

One major area for future is to complete and extend the implementation. The first item is to complete the integration with Consul. After that, a port of FT-Linda to Isis would be interesting. The extensions to the implementation discussed in Section 5.8 — reintegration of failed hosts, allowing non-full replication, and tolerating network partitions — should be completed. Finally, the implementation could be optimized further. One optimization would be to transmit only unchanging information in an AGS request message, as was discussed in Section 5.7. Another optimization to investigate is whether it is beneficial to use a semantic dependent ordering in delivering the AGS requests to the replicated TS managers than the more restrictive total ordering. For example, AGSs with all **rd**s are commutative with each other, and thus a collection of such read-only AGSs could be executed in different orders at different replicas. Unfortunately, a semantic dependent ordering can feature higher latency than a total ordering if only a few of the operations are commutative. It is an open question as to whether enough Linda applications would contain a sufficient number of read-only AGSs so as to warrant using a semantic dependent ordering. However, note that this choice could be made by the user when starting the FT-Linda program.

The FT-Linda language will continue to mature. One possibility in this area is to accommodate many or all of the possible extensions listed in Section 3.5. Another is to support more opcodes. A final possibility is to remove or lessen some of the restrictions, most notably to allow expressions in an AGS.

FT-Linda also seems a good candidate as a back-end language for indirect usage. One example of this was given in the general transaction facility in Section 4.1.3. Another possibility is to build on Linda tools or other additions to Linda. The Linda Program Builder (LPB) is a high-level Computer Aided Software Engineering (CASE) tool that is used to make programming in Linda simpler. For example, it presents the programmer with menu choices for common Linda paradigms such as the bag-of-tasks and the distributed variable paradigms. In doing so, it ensures that the programmer supplies information about the actual usage of each tuple signature. This allows a higher level of optimization than just static analysis of the code would permit. It would be instructive to explore FT-Linda's usage as a back-end to the LPB; indeed, the programmer might not even need to know about FT-Linda, only Linda. Another interesting Linda project is Piranha, which allows idle workstations to be used transparently. FT-Linda's integration with Piranha would be of great interest to many, especially if it were also integrated with a CASE tool such as the LPB, Another back-end usage of FT-Linda is with finite state automatas (FSAs), which are very useful for specifying features in telecommunications systems [Zav93]. FT-Linda could be used to implement FSAs. For example, events and states could be represented by tuples, and a state change — the consumption and production of event and state tuples — could be represented by a single AGS.

Finally, there is a great need for more high-level language support for fault-tolerant, real-time programming. If Linda's simple tuple space model could be used for this it would be both edifying and useful. Real time systems need predictability, which can be met in part by ensuring that all timing constraints are met. There are a number of possible ways that FT-Linda could be extended to facilitate this, a thorough discussion of which is beyond the scope of this dissertation. However, any such extensions would have to answer at least the following questions:

- Will hard real time or soft real time or both be supported?

- Will the predictability be achieved by priorities or deadlines or both?

- Are these priorities or deadlines to be associated with a TS, a TS operation, or an entire AGS? Further, if they are associated with a TS or a TS operation, what is the meaning of the deadline/priority in the face of an AGS that has multiple priorities represented in the TSs involved with it.

Further, it would be interesting if facilities to allow the programmer to deal with a timing failure could be provided in a similar fashion to how FT-Linda provides failure notifications upon the failure of a host. Additionally, the state machine approach used to implement FT-Linda seems well-suited to provide real-time support, in addition to fault-tolerance. Indeed, one such project is already underway to provide real-time communication support for replicated state machine in the Corto project [AGMvR93], a real-time successor to Isis. This could serve as an excellent basis for a real-time versions of FT-Linda.

APPENDIX A

FT-LINDA IMPLEMENTATION NOTES

The C FT-Linda code in the appendices differs from the pseudocode given in previous chapters in a number of ways.

First, the code to parse array subscripts in an AGS has not yet been implemented. This is only encountered in Appendix G. The workaround used in the following appendices is to use a non-subscripted variable in the AGS and copy to and from it. This variable can be an array or structure or anything that is not subscripted and that the C `sizeof()` facility will give the true size of the data in question (i.e., not pointers, either directly or in structures).

Second, **inp** and **rdp** have not yet been implemented in expressions. In this case, these operations are boolean guards and the entire AGS is a boolean expression. The workaround is to set a variable with a sentinel value that cannot occur in a tuple and then use this variable as a formal in an AGS. The value of this variable can be compared to the illegal value after the AGS to ascertain whether or not the AGS guard and body was executed.

Third, the implementation provides a hook, `ftl_fail_host()`, to simulate the failure of a host.

Fourth, the logical name of a tuple is not a character string like "$name$" but rather is a void type, which is stylistically represented in uppercase (e.g., NAME). This void type is declared with the `newtype` operator, which is the C-Linda analogy to the C `typedef` operator. For further information, see [Lei89],

APPENDIX B

FT-LINDA REPLICATED SERVER

```
/* Replicated server example.  There is no monitor process since the failure
 * of a server does not need to be cleaned up after. */

#ttcontext replicated_server
#include <stdio.h>
#include "ftlinda.h"

#define THIS_SERVICE    1
#define NUM_CLIENTS     5
#define CMD1        1   /* SQR */
#define CMD2        2   /* SUM */
#define CLIENT_LOOPS    2
#define SQR_ANS(x)    ((x) * (x))
#define SUM_ANS(a,b,c)    ((a) + (b) + (c))

newtype void SERVER_TIME;
newtype void REQUEST;
newtype void REPLY;
newtype void SQR_CMD;   /* CMD1 */
newtype void SUM_CMD;   /* CMD2 */

void server(void);
void client(int);
```

```
LindaMain (argc, argv)
int argc;
char* argv [];
{
   int host, i, lpid, num_hosts = ftl_num_hosts();

   /* initialize the sequence tuple */
   < true => out(TSmain, SERVER_TIME, THIS_SERVICE, (int) 0); >

   /* Create one server replica on each host */
   for (host=0;  host<num_hosts;  host++) {
   lpid = new_lpid();
   ftl_create_user_thread(server, "server", host, lpid, 0, 0, 0, 0);
   }

   /* create some clients */
   for (i=0; i<NUM_CLIENTS; i++) {
   lpid = new_lpid();
   host = i % num_hosts;
   ftl_create_user_thread(client, "client", host, lpid, i, 0, 0, 0);
   }
   /* The LindaMain thread goes away here, but the program won't be
    * finished until all living clients are through */
}
```

```
/* The client invokes both services CLIENT_LOOPS times.  It also
 * tests the answers it gets, something of course a real client
 * generally would not (and often could not) do.
 */
void
client (int client_id)
{
   int time, i, x, a, b, c, answer;

   printf("Client %d on host %d here\n", client_id, ftl_my_host() );

   for (i=0; i<CLIENT_LOOPS; i++) {

   /* invoke the first command */
   x=i+10;
   < in(TSmain, SERVER_TIME, THIS_SERVICE, ?time) =>
      out(TSmain, SERVER_TIME, THIS_SERVICE, PLUS(time,1) );
      out(TSmain, REQUEST, THIS_SERVICE, time, SQR_CMD, CMD1, x);
   >

   /* wait for the first reply to this command */
   < in(TSmain, REPLY, THIS_SERVICE, time, ?answer) => skip >
   if (answer ≠ SQR_ANS(x))
      ftl_exit("Client got bad sqr answer.", 1);

   /* invoke the second command */
   a=i*100; b=i*200; c=i*300;
   < in(TSmain, SERVER_TIME, THIS_SERVICE, ?time) =>
      out(TSmain, SERVER_TIME, THIS_SERVICE, PLUS(time,1) );
      out(TSmain, REQUEST, THIS_SERVICE, time, SUM_CMD, CMD2, a, b, c);
   >

   /* wait for the first reply to this command */
   < in(TSmain, REPLY, THIS_SERVICE, time, ?answer) => skip >
   if (answer ≠ SUM_ANS(a, b, c))
      ftl_exit("Client got bad answer.", 1);

   }

   printf("Client %d on host %d done\n", client_id, ftl_my_host() );

}
```

```
/* The server implements two different commands, both of which return
 * a simple answer.
 */
void
server()
{
    int time, x, a, b, c, answer, cmd;

    /* loop forever over all times */
    for (time=0; ; time++) {

    /* read the next request tuple */
    < rd(TSmain, REQUEST, THIS_SERVICE, time, SQR_CMD, ?cmd, ?x) => skip
    or
      rd(TSmain, REQUEST, THIS_SERVICE, time, SUM_CMD, ?cmd, ?a, ?b, ?c) => skip
    >

    /* compute the answer for the request */
    switch(cmd) {
        case CMD1:
            answer = SQR_ANS(x);
            break;
        case CMD2:
            answer = SUM_ANS(a,b,c);
            break;
        default:
            ftl_exit("Server error", 1);
            /*NOTREACHED*/
    }

    /* send the reply */
    < true => out(TSmain, REPLY, THIS_SERVICE, time, answer); >

    }
    /*NOTREACHED*/
}
```

APPENDIX C

FT-LINDA RECOVERABLE SERVER EXAMPLE

```
/* Recoverable server example.  This shows how to implement two commands,
 * instead of the one shown earlier in the paper.  */

#ttcontext recoverable_server
#include <stdio.h>
#include "ftlinda.h"
#include "assert.h"

/* Anything that can be an actual in a tuple had better be cast to a type
 * so the signatures are guaranteed to match */
#define MY_SERVICE        (int) 1
#define NUM_CLIENTS        (int) 5
#define CMD1            (int) 1    /* SQR */
#define CMD2            (int) 2    /* SUM */
#define CLIENT_LOOPS        2
#define SQR_ANS(x)        ((x) * (x))
#define SUM_ANS(a,b,c)        ((a) + (b) + (c))
#define INITIAL_SERVER_HOST    (int) 1
#define ILLEGAL_HOST        −1
#define ILLEGAL_SQR        −1
#define INIT_SUM        (int) 0
#define INIT_SQR        (int) 0

newtype void REQUEST;
newtype void IN_PROGRESS;
newtype void REPLY;
newtype void SERVER_REGISTRY;
newtype void SERVER_STATE;
newtype void SERVER_HANDLE;
newtype void SQR_CMD;    /* CMD1 */
newtype void SUM_CMD;    /* CMD2 */
newtype void REINCARNATE;

void server(void);
void client(int);
void monitor(int);
```

```
LindaMain (argc, argv)
int argc;
char* argv [];
{
   int f_id, host, i, server_lpid, lpid, num_hosts = ftl_num_hosts();
   ts_handle_t server_handle;

   /* For now we use a {Stable,Shared} TS for the server, instead
    * of a {Stable,Private} one as one would normally do, since
    * passing an LPID to the create primitive isn't implemented yet. */
   server_lpid = new_lpid();
   create_TS(Stable, Shared, &server_handle);

   /* Initialize the TS handle and registry for the server.
    * We will not initialize the state, since if server_handle were
    * Private, as our solution would normally have, we could not do this.
    */
   < true =>    out(TSmain, SERVER_HANDLE, MY_SERVICE, server_handle);
      out(TSmain, SERVER_REGISTRY, MY_SERVICE, server_lpid, INITIAL_SERVER_HOST);
   >

   /* create a monitor process on each host */
   for (host=0;  host<num_hosts;  host++) {
   lpid = new_lpid();
   f_id = new_failure_id();
   ftl_create_user_thread(monitor, "monitor", host, lpid, f_id, 0, 0, 0);
   }

   /* Create one server on host INITIAL_SERVER_HOST */
   assert(ftl_num_hosts() ≥ INITIAL_SERVER_HOST);
   lpid = new_lpid();
   ftl_create_user_thread(server, "server", INITIAL_SERVER_HOST, lpid, 0, 0, 0, 0);


   /* create some clients */
   for (i=0; i<NUM_CLIENTS; i++) {
   lpid = new_lpid();
   host = i % num_hosts;
   ftl_create_user_thread(client, "client", host, lpid, i, 0, 0, 0);
   }
   /* The LindaMain thread goes away here, but the program won't be
    * finished until all living clients are through */
}
```

```
/* The client invokes both services CLIENT_LOOPS times.  It also
 * tests the answers it gets, something of course a real client
 * generally would not (and often could not) do.
 */
void
client (int client_id)
{
   int i, x, a, b, c, answer, my_lpid = ftl_my_lpid();

   printf("Client %d on host %d here\n", client_id, ftl_my_host() );

   for (i=1; i≤CLIENT_LOOPS; i++) {

   /* invoke the first command */
   x=i+10;
   < true => out(TSmain, REQUEST, MY_SERVICE, my_lpid, SQR_CMD, CMD1, x); >

   /* wait for the first reply to this command */
   < in(TSmain, REPLY, MY_SERVICE, my_lpid, ?answer) => skip >
   if (answer ≠ SQR_ANS(x))
      ftl_exit("Client got bad sqr answer.", 1);

   /* invoke the second command */
   a=i*100; b=i*200; c=i*300;
   < true => out(TSmain, REQUEST, MY_SERVICE, my_lpid, SUM_CMD, CMD2, a, b, c); >

   /* wait for the first reply to this command */
   < in(TSmain, REPLY, MY_SERVICE, my_lpid, ?answer) => skip >
   if (answer ≠ SUM_ANS(a, b, c))
      ftl_exit("Client got bad answer.", 1);

   }

   printf("Client %d on host %d done\n", client_id, ftl_my_host() );

}
```

```
/* The server implements two different commands, both of which return
 * a simple answer.
 */
void
server()
{
   int x, a, b, c, answer, cmd, best_sqr, best_sum, client_lpid;
   ts_handle_t my_ts;

   printf("Server here on host %d\n", ftl_my_host() );

   /* read in server's TS handle */
   < rd(TSmain, SERVER_HANDLE, MY_SERVICE, ?my_ts) => skip >

   /* read in server's state.  Note that we would normally initialize
    * it like this:
    *       if ( < not rdp(my_ts, SERVER_STATE, ... ) =>
    *           out(my_ts, SERVER_STATE, ... initial values ...); >
    * However, we cannot do this since the AGS has not yet been implemented
    * as part of an expression.  Thus, we will simulate this.
    */
   best_sqr = ILLEGAL_SQR;
   < rdp(my_ts, SERVER_STATE, MY_SERVICE, ?best_sqr, ?best_sum) => skip >

   if (best_sqr == ILLEGAL_SQR) {

   /* The rdp did not find a state tuple, so we will create one.
    *
    * Note that having these two AGSs would not work in general,
    * i.e. it won't have the same semantics viz. failures and concurrency
    * as the AGS expression we would normally implement it with.
    * We know here that it works, however, since there can be only this
    * server executing MY_SERVICE at once.
    */

   best_sqr = INIT_SQR;
   best_sum = INIT_SUM;
   < true => out(my_ts, SERVER_STATE, MY_SERVICE, best_sqr, best_sum); >

   }
```

```
  /* loop forever */
  for (;;) {
  /* read the next request tuple */
  < in(TSmain, REQUEST, MY_SERVICE, ?client_lpid, SQR_CMD, ?cmd, ?x) =>
     out(TSmain, IN_PROGRESS, MY_SERVICE, client_lpid, SQR_CMD, cmd, x);
  or
    in(TSmain, REQUEST, MY_SERVICE, ?client_lpid, SUM_CMD, ?cmd, ?a, ?b, ?c) =>
     out(TSmain, IN_PROGRESS, MY_SERVICE, client_lpid, SUM_CMD, cmd, a, b, c);
  >

  /* compute the answer for the request */
  switch(cmd) {
     case CMD1:
        answer = SQR_ANS(x);
        best_sqr = (answer > best_sqr ? answer : best_sqr);
        break;
     case CMD2:
        answer = SUM_ANS(a,b,c);
        best_sum = (answer > best_sum ? answer : best_sum);
        break;
     default:
        ftl_exit("Server error", 1);
  }

  /* send the reply and update state */
  switch(cmd) {
     case CMD1:
        < in(TSmain, IN_PROGRESS, MY_SERVICE, ?int, SQR_CMD, cmd, ?int) =>
           out(TSmain, REPLY, MY_SERVICE, client_lpid, answer);
              in(my_ts, SERVER_STATE, MY_SERVICE, ?int, ?int);
              out(my_ts, SERVER_STATE, MY_SERVICE, best_sqr, best_sum);
        >
        break;
     case CMD2:
        < in(TSmain, IN_PROGRESS, MY_SERVICE, ?int, SUM_CMD, cmd, ?int, ?int, ?int) =>
           out(TSmain, REPLY, MY_SERVICE, client_lpid, answer);
              in(my_ts, SERVER_STATE, MY_SERVICE, ?int, ?int);
              out(my_ts, SERVER_STATE, MY_SERVICE, best_sqr, best_sum);
        >
        break;
     default:
        ftl_exit("Server error", 1);
  }
  }
  /*NOTREACHED*/
}
```

```
void
monitor(int failure_id)
{
  int failed_host, host, server_lpid, client_lpid, my_host = ftl_my_host(), reincarnate, x, a, b, c, cmd;
  ts_handle_t scratch_ts;

  create_TS(Volatile, Private, &scratch_ts);

  for (;;) {
  < in(TSmain, FAILURE, failure_id, ?failed_host) => skip > /* Wait for a failure */

  /* See if server MY_SERVICE was executing on the failed host.
   * Again, we simulate an AGS in an expression here. */
  host = ILLEGAL_HOST;
  < rdp(TSmain, SERVER_REGISTRY, MY_SERVICE, ?server_lpid, ?host) => skip >
  if (host == failed_host) {

    /* Service MY_SERVICE, which we are monitoring, has failed.
     * Regenerate any request tuples found for MY_SERVICE.  Note
     * that since there is only one server replica there can be
     * at most one IN_PROGRESS tuple. */
    < inp(TSmain, IN_PROGRESS, MY_SERVICE, ?client_lpid, SQR_CMD, ?cmd, ?x) =>
      out(TSmain, REQUEST, MY_SERVICE, client_lpid, SQR_CMD, cmd, x);
    or
     inp(TSmain, IN_PROGRESS, MY_SERVICE, ?client_lpid, SUM_CMD, ?cmd, ?a, ?b, ?c) =>
      out(TSmain, REQUEST, MY_SERVICE, client_lpid, SUM_CMD, cmd, a, b, c);
    >

    /* Attempt to start a new incarnation of the failed server.  Again,
     * we simulate the effect of an AGS expression with the REINCARNATE tuple. */
    < inp(TSmain, SERVER_REGISTRY, MY_SERVICE, server_lpid, failed_host) =>
    out(TSmain, SERVER_REGISTRY, MY_SERVICE, server_lpid, my_host);
    out(scratch_ts, REINCARNATE, (int) 1);
    >

    /* See if we did change the registry; in this case create a new * server on this host. */
    reincarnate = 0;
    < inp(scratch_ts, REINCARNATE, ?reincarnate) => skip >
    if (reincarnate)
    ftl_create_user_thread(server, "server", my_host, server_lpid, 0, 0, 0, 0);
  }
  }
}
```

APPENDIX D

FT-LINDA GENERAL TRANSACTION MANAGER EXAMPLE

D.1 Specification

```
/* FT-Linda specification for a general transaction manager. It does not
 * generally verify that variable IDs are correct or do other error checking. */

newtype int val_t;    /* values for variables used in a transaction */
newtype int var_t;    /* variable handles */
newtype int tid_t;    /* transaction IDs */

#define ILLEGAL_VAR     −1

/* init_transaction_mgr() must be called exactly once before any transaction
 * routine below is used */
void init_transaction_mgr(void);

/* create_var creates a variable with ID var and an initial value of val. */
void create_var(var_t var, val_t val);

/* destroy variable var */
void destroy_var(var_t var);

/* start_transaction begins a transaction involving the num_vars variables
 * in var_list. It returns the transaction ID for this transaction. */
tid_t start_transaction(var_t var_list[], int num_vars);

/* modify_var modifies var to have the value new_val. This is assumed
 * to be called after a transaction was started with this variable. */
void modify_var(tid_t tid, var_t var, val_t new_val);

/* abort aborts transaction tid */
void abort(tid);

/* commit commits transaction tid */
void commit(tid);

/* print out the varaibles and their values, in one atomic snapshot */
void print_variables(char *msg);
```

D.2 Manager

```
#include <malloc.h>

newtype void TIDS;
newtype void LOCK;
newtype void LOCK_INUSE;
newtype void VAR;
newtype void VAR_INUSE;
newtype void TS_CUR;
newtype void TS_ORIG;

/* For each transaction we keep two scratch TSs: cur_ts keeps
 * the current values of the variables, and orig_ts keeps the
 * original values of the variables plus VAR and LOCK tuples.
 * get_cur and get_orig are utility routines that fetch the
 * handles for these two TSs for a transaction.
 */
static void get_cur(tid_t, ts_handle_t *);
static void get_orig(tid_t, ts_handle_t *);

static void monitor_transactions(int failure_id);


/* init_transaction_mgr() must be called exactly once before any transaction
 * routine below is used */
void init_transaction_mgr()
{
   int lpid, f_id;
   /* create one monitor process on each host */
   for (host=0;  host<ftl_num_hosts();  host++) {
      lpid = new_lpid();
      f_id = new_failure_id();
      ftl_create_user_thread(monitor_transactions, "monitor_transactions",
            host, lpid, f_id, 0, 0, 0);
   }

   < true => out(TSmain, TIDS, (tid_t) 1); >
}
```

```
/* create_var creates a variable with ID var and an initial value of val.
 * No validity check is done on val.
 */
void create_var(var_t var, val_t val)
{

   if (var == (var_t) ILLEGAL_VAR)
   ftl_exit("create_var has a conflict with ILLEGAL_VAR", 1);

   < true =>
   out(TSmain, VAR, var, val);
   out(TSmain, LOCK, var);
   >
}

/* destroy variable var */
void destroy_var(var_t var)
{
   /* this blocks until any current transaction with var completes */
   < in(TSmain, LOCK, var) => in(TSmain, VAR, var, ?val_t); >
}
```

```
/* start_transaction begins a transaction involving the num_vars variables in var_list.
 * It returns the transaction ID for this transaction or ILLEGAL_TID if the transaction
 * could not be started (either because there were too many outstanding transactions
 * or because a variable in var_list[]).  */
tid_t
start_transaction(var_t var_list[], int num_vars)
{
   tid_t tid; var_t *vars, var; val_t val;
   int i, my_host = ftl_my_host();
   static  int var_compare(var_t *i, var_t *j);
   ts_handle_t cur_ts, orig_ts;
   char buf[100];

   < in(TSmain, TIDS, ?tid) => out(TSmain, TIDS, PLUS(tid,1) ); >

   /* Create scratch TSs to keep a pristine copy of the variables involved
    * in this transaction as well as their current uncommittted values */
   create_TS(Volatile, Private, &cur_ts); create_TS(Volatile, Private, &orig_ts);
   < true => out(TSmain, TS_CUR, tid, cur_ts);    out(TSmain, TS_ORIG, tid, orig_ts); >

   /* Create a safe copy of var_list and then sort it */
   vars = (var_t *) calloc(num_vars, sizeof(var_t) );
   assert(vars ≠ (var_t) NULL);
   for (i=0; i<num_vars; i++)
   vars[i] = var_list[i];
   qsort(vars, num_vars, sizeof(var_t), var_compare);

   /* acquire all the locks for these variables in order */
   for (i=0; i < num_vars; i++) {
   var = vars[i];
   /* Move LOCK from TSmain to orig_ts and leave LOCK_INUSE in TSmain
    * (it is used only for recovery).  Similarly for VAR; also add
    * a copy of VAR to cur_ts.  */
   < in(TSmain, LOCK, var) =>
      out(TSmain, LOCK_INUSE, my_host, tid, var);
      out(orig_ts, LOCK, var);
      in(TSmain, VAR, var, ?val);
      out(TSmain, VAR_INUSE, my_host, tid, var, val);
      out(orig_ts, VAR, var, val);
      out(cur_ts, VAR, var, val);
   >
   }
   cfree(vars);
   return tid;
}
```

```
/* modify_var modifies var to have the value new_val. This is assumed
 * to be called after a transaction was started with this variable.
 */
void modify_var(tid_t tid, var_t var, val_t new_val)
{
   ts_handle_t cur_ts;
   char buf[100];

   get_cur(tid, &cur_ts);

   sprintf(buf,"cur_ts for modify var V%d val %d tid %d", var, new_val, tid);

   < in(cur_ts, VAR, var, ?val_t) => out(cur_ts, VAR, var, new_val); >

}

/* abort aborts transaction tid */
void
abort(tid_t tid)
{
   ts_handle_t cur_ts, orig_ts;
   int my_host = ftl_my_host();
   char buf[100];

   /* Regenerate LOCK and VAR from TSmain, discard their INUSE
    * placeholders there, and remove the scratch TS handles from TS. */
   < true =>
   move(orig_ts, TSmain, LOCK, ?var_t);
   move(orig_ts, TSmain, VAR, ?var_t, ?val_t);
   in(TSmain, TS_CUR, tid, ?ts_handle_t);
   in(TSmain, TS_ORIG, tid, ?ts_handle_t);
   move(TSmain, cur_ts, VAR_INUSE, my_host, tid, ?var_t, ?val_t);
   move(TSmain, orig_ts, LOCK_INUSE, my_host, tid, ?var_t);
   >

   destroy_TS(cur_ts); destroy_TS(orig_ts);
}
```

```
/* commit commits transaction tid */
void
commit(tid_t tid)
{
   ts_handle_t cur_ts, orig_ts;
   int host = ftl_my_host();
   char buf[100];

   get_cur(tid, &cur_ts);
   get_orig(tid, &orig_ts);

   /* Restore the LOCKs from this transaction from orig_ts, and move the
    *  current values of all variables involved in this transaction from
    * cur_ts to TSmain.  Discard the VAR_INUSE and LOCK_INUSE placeholders,
    * and remove the scratch TS handles from TS.
    */

   < true =>
   move(orig_ts, TSmain, LOCK, ?var_t);
   move(cur_ts, TSmain, VAR, ?var_t, ?val_t);
   in(TSmain, TS_CUR, tid, ?ts_handle_t);
   in(TSmain, TS_ORIG, tid, ?ts_handle_t);
   move(TSmain, cur_ts, VAR_INUSE, host, tid, ?var_t, ?val_t);
   move(TSmain, cur_ts, LOCK_INUSE, host, tid, ?var_t);
   >

   destroy_TS(cur_ts);
   destroy_TS(orig_ts);

   printf("Aborted transaction %d for LPID %d\n", tid, ftl_my_lpid() );
}
```

```
static void
monitor_transactions(int failure_id)
{
    int failed_host;
    val_t val;
    var_t var;

    for (;;) {
    /* wait for a failure */
    < in(TSmain, FAILURE, failure_id, ?failed_host) => skip >

    /* regenerate all LOCKs and VARs we find for any transactions
     * on failed_host.
     */
    do {
        var = ILLEGAL_VAR;
        < inp(TSmain, LOCK_INUSE, failed_host, ?tid_t, ?var) =>
        out(TSmain, LOCK, var);
        in(TSmain, VAR_INUSE, failed_host, ?tid_t, var, ?val)
        out(TSmain, VAR, var, val);
        >
    } while (var ≠ ILLEGAL_VAR);


    }
}
```

```
/* print_variables will store the values into a buffer and then print, since
 * it could be scheduled out at each AGS.  That would almost certainly make
 * the output interlaced with other output, which is not very useful. */
void
print_variables(char *msg)
{
   ts_handle_t scratch_ts;
   val_t val; tid_t tid; var_t var; int host;
   char buf[1000], buf2[100];    /* Big enough ... */

   create_TS(Volatile, Private, &scratch_ts);

   /* Grab an atomic snapshot of all variables, whether in use or not. */
  < true =>
   copy(TSmain, scratch_ts, VAR, ?var_t, ?val_t);
   copy(TSmain, scratch_ts, VAR_INUSE, ?int, ?tid_t, ?var_t, ?val_t);
  >

  sprintf(buf, "Variables at %s\n", msg);
  /* Format out that snapshot, again simulating an AGS expression. */
  do {
   var = ILLEGAL_VAR;
   < inp(scratch_ts, VAR, ?var, ?val) => skip >
   sprintf(buf2, "\tV%d=0x%x\n", var, val);
   strcat(buf, buf2);
   } while (var ≠ ILLEGAL_VAR);
  do {
   var = ILLEGAL_VAR;
   < inp(scratch_ts, VAR_INUSE, ?host, ?tid, ?var, ?val) => skip >
   sprintf(buf2,"\tV%d=0x%x\t(INUSE with tid %d on host %d)\n",
      var, val, tid, host);
   strcat(buf, buf2);
   } while (var ≠ ILLEGAL_VAR);

   destroy_TS(scratch_ts);
   printf(buf);
}
```

```
static  int
var_compare(var_t ∗i, var_t ∗j)
{
        return(∗i − ∗j);
}


static void
get_cur(tid_t tid, ts_handle_t ∗handle)
{
   ts_handle_t temp;
   unsigned int old = rts_debug_value();

   < true => rd(TSmain, TS_CUR, tid, ?temp); >

   ∗handle = temp;
}


static void
get_orig(tid_t tid, ts_handle_t ∗handle)
{
   ts_handle_t temp;

   < true => rd(TSmain, TS_ORIG, tid, ?temp); >

   ∗handle = temp;
}
```

D.3 Sample User

```
/* transaction_user.c is a torture test for the transaction manager.
 * srandom() initializes things for random() outside of this file.
 */

#ttcontext transaction_user
#include <stdio.h>
#include "ftlinda.h"
#include "assert.h"
#include "transaction_mgr.h"
#include "transaction_mgr.c"

/* We define symbols to use as handles for the variables, as well
 * as their initial values */
#define A_VAR   1
#define B_VAR   2
#define C_VAR   3
#define D_VAR   4
#define E_VAR   5
#define F_VAR   6
#define G_VAR   7
#define H_VAR   8

#define A_INIT  0x100
#define B_INIT  0x200
#define C_INIT  0x300
#define D_INIT  0x400
#define E_INIT  0x500
#define F_INIT  0x600
#define G_INIT  0x700
#define H_INIT  0x800

#define CLIENT_LOOPS 10
#define ABORT_ROLL  (random() % 6)
#define FAIL_ROLL   (random() % 30)

static void client1(), client2(), client3(), client4(), client5();
static void shuffle(var_t[],int);
static void maybe_fail(void);
```

```
LindaMain (argc, argv)
int argc;
char* argv [];
{
  int lpid, num_hosts = ftl_num_hosts();

  printf("LindaMain here\n");

  init_transaction_mgr();

  /* Create the variables */
  create_var(A_VAR, A_INIT);
  create_var(B_VAR, B_INIT);
  create_var(C_VAR, C_INIT);
  create_var(D_VAR, D_INIT);
  create_var(E_VAR, E_INIT);
  create_var(F_VAR, F_INIT);
  create_var(G_VAR, G_INIT);
  create_var(H_VAR, H_INIT);

  /* Create the clients */
  lpid = new_lpid();
  ftl_create_user_thread(client1, "client1", 1 % num_hosts, lpid, 0, 0, 0, 0);
  lpid = new_lpid();
  ftl_create_user_thread(client1, "client2", 2 % num_hosts, lpid, 0, 0, 0, 0);
  lpid = new_lpid();
  ftl_create_user_thread(client1, "client3", 3 % num_hosts, lpid, 0, 0, 0, 0);
  lpid = new_lpid();
  ftl_create_user_thread(client1, "client4", 4 % num_hosts, lpid, 0, 0, 0, 0);
  lpid = new_lpid();
  ftl_create_user_thread(client1, "client5", 5 % num_hosts, lpid, 0, 0, 0, 0);

}
```

```
/* The clients test the transaction manager */
/* client1 */
#define CLIENT       1
#define CLIENT_NAME    client1
#define VARS_USED    {A_VAR, C_VAR, G_VAR}
#include "transaction_client.c"       /* defines client1 */
#undef CLIENT
#undef CLIENT_NAME
#undef VARS_USED

/* client2 */
#define CLIENT       2
#define CLIENT_NAME    client2
#define VARS_USED    {B_VAR, D_VAR, F_VAR, G_VAR}
#include "transaction_client.c"       /* defines client2 */
#undef CLIENT
#undef CLIENT_NAME
#undef VARS_USED

/* client3 */
#define CLIENT       3
#define CLIENT_NAME    client3
#define VARS_USED    {A_VAR}
#include "transaction_client.c"       /* defines client3 */
#undef CLIENT
#undef CLIENT_NAME
#undef VARS_USED

/* client4 */
#define CLIENT       4
#define CLIENT_NAME    client4
#define VARS_USED    {C_VAR, G_VAR}
#include "transaction_client.c"       /* defines client4 */
#undef CLIENT
#undef CLIENT_NAME
#undef CLIENT_NAME
#undef VARS_USED

/* client5 */
#define CLIENT       5
#define CLIENT_NAME    client5
#define VARS_USED    {A_VAR, B_VAR, C_VAR, D_VAR, E_VAR, F_VAR, G_VAR}
#include "transaction_client.c"       /* defines client5 */
#undef CLIENT
#undef CLIENT_NAME
#undef VARS_USED
```

```
/∗ Shuffle the variable array ∗/
static void
shuffle(var_t vars[], int num_vars)
{
    int i, slot;
    var_t item;

    for (i=num_vars−1; i > 0; i−−) {
    /∗ swap vars[i] with vars[slot] for some slot in [0,i) ∗/
    slot = random() % i;
    item = vars[i];
    vars[i] = vars[slot];
    vars[slot] = item;
    }
}


static void
maybe_fail()
{
    int host = ftl_my_host();
    /∗ see again if we should fail our host ∗/
    if ( (host ≠ 0) && (FAIL_ROLL == 0) ) {
    printf("Failing host %d\n", host);
    ftl_fail_host(host);
    }

}
```

D.4 User Template (transaction_client.c)

```
/* This file holds the template for each client; they are all
 * instantiated with different macros for the function name,
 * variables used, etc.
 */
void
CLIENT_NAME ()
{
   static var_t vars_used[] = VARS_USED;
#define NUM_VARS_USED (sizeof(vars_used) / sizeof(var_t))
   int i, j, my_host = ftl_my_host(), abort_it;
   tid_t tid;
   char buf[200], buf2[50];
   char name[20];

   sprintf(name, "client%d", CLIENT);
   sprintf(buf,"%s here on host %d with %d variables: ",
   name, ftl_my_host(), NUM_VARS_USED);

   for (i=0; i<NUM_VARS_USED; i++) {
   sprintf(buf2, "V%d ", vars_used[i]);
   strcat(buf, buf2);
   }

   printf("%s\n", buf);

   for (i=1; i ≤ CLIENT_LOOPS; i++) {
   val_t vals_this_time[NUM_VARS_USED];

   /* Shuffle the order of the variables used */
   shuffle(vars_used, NUM_VARS_USED);

   tid = start_transaction(vars_used, NUM_VARS_USED);
   maybe_fail();

   /* Generate some random values for these variables */
   for (j=0; j<NUM_VARS_USED; j++)   {
      vals_this_time[j] = (int) (random() % 0x10000);
      modify_var(tid, vars_used[j], vals_this_time[j]);
   }
```

```
    maybe_fail();
    /* See if we should abort */
    abort_it = (ABORT_ROLL == 0);
        if (abort_it)
        abort(tid);
    else
        commit(tid);

    sprintf(buf, "after %s %s changes: ",
        name, (abort_it ? "aborted" : "committed") );
    for (j=0; j<NUM_VARS_USED; j++)  {
        sprintf(buf2, "V%d=0x%x", vars_used[j], vals_this_time[j]);
        strcat(buf, buf2);
    }
    print_variables(buf);
    }
    printf("%s on %d all done\n", name, my_host);
#undef NUM_VARS_USED
}
```

APPENDIX E

FT-LINDA BAG-OF-TASKS EXAMPLE

```
/* Bag of Tasks example program. */
#ttcontext bag_of_tasks
#include <stdio.h>
#include "ftlinda.h"

newtype void SUBTASK;
newtype void RESULT;
newtype void INPROGRESS;

#define ILLEGAL_VAL -1    /* illegal value for subtask */
#define NUM_WORKERS 10
#define NUM_SUBTASKS 20

char *progname;

static void worker(int);
static void calc(int, int *);
static void get_input(int *, int *);
static void monitor();
```

```
LindaMain (argc, argv)
int argc;
char* argv [];
{
  int i, val, f_id, lpid, host, num_hosts = ftl_num_hosts();
  progname = argv[0];

  printf("%s here with %d hosts in [0..%d)\n", progname, num_hosts,
      num_hosts);

  /* Create a monitor thread on each host */
  for (host=0; host < num_hosts; host++) {
  lpid = new_lpid();
  f_id = new_failure_id();
  ftl_create_user_thread(monitor, "monitor", host, lpid, f_id, 0, 0, 0);
  }

  /* Create some workers */
  for (i=0; i<NUM_WORKERS; i++) {
  lpid = new_lpid();
  ftl_create_user_thread(worker, "worker", i % num_hosts, lpid, i, 0, 0, 0);
  }

  /* Create some subtasks */
  for (i=0; i<NUM_SUBTASKS; i++) {
  < true => out(TSmain, SUBTASK, i, i); >
  }

  /* Wait for those subtasks to be completed. */
  for (i=0; i<NUM_SUBTASKS; i++) {
  < in(TSmain, RESULT, i, ?val) => skip >
  }

  printf("%s done\n", progname);

  ftl_exit(NULL, 0);   /* exit normally */

}
```

```
static void
monitor(failure_id)
int failure_id;
{
   int num, val, failed_host, id;

   while(1) {

   failed_host = −1;      /∗ sanity check ∗/

   /∗ wait for a failure like a vulture ∗/
   < in(TSmain, FAILURE, failure_id, ?failed_host) => skip >

   /∗ try to regenerate any INPROGRESS tuple for a failed worker
    ∗ on the host that failed ∗/
   do {

      val = ILLEGAL_VAL;    /∗ illegal subtask value ∗/

      < inp(TSmain, INPROGRESS, failed_host, ?id, ?num, ?val) =>
               out(TSmain, SUBTASK, num, val); >

   }
   while (val ≠ ILLEGAL_VAL);
   }
}
```

```
static void
worker(id)
int id;
{
   int num, val, result, host=ftl_my_host();

   ts_handle_t TSscratch;

   create_TS(Volatile, Private, &TSscratch);

   printf("worker(%d) here on host %d\n", id, host);

   while (1) {

   /* the worker ID in the INPROGRESS tuple is not needed, but is
    * useful for debugging */
   < in(TSmain, SUBTASK, ?num, ?val) =>
            out(TSmain, INPROGRESS, host, id, num, val); >

   calc(val, &result);

   < true => out(TSscratch, RESULT, num, result); >

   < in(TSmain, INPROGRESS, host, id, num, val) =>
      move(TSscratch, TSmain); >

   }
}


static void
calc(val, result_ptr)
int val;
int *result_ptr;
{
   *result_ptr = 2*val;    /* really simple ... */
}
```

APPENDIX F

FT-LINDA DIVIDE AND CONQUER EXAMPLE

```
/* Fault-tolerant divide and conquer worker. Here we sum up
 * the elements of a vector to demonstrate the technique. */

#ttcontext divide
#include <stdio.h>
#include "ftlinda.h"

#define MAX_SIZE        256     /* biggest vector size */
#define MAX_ELEM        50      /* biggest element */
#define MIN_ELEM    10    /* smallest element */
#define SIZE_CUTOFF     16
#define SMALL_ENOUGH(task)   (task.size ≤ SIZE_CUTOFF ? 1 : 0)
#define ILLEGAL_SIZE    −1
#define WORKERS_PER_HOST 4    /* number of workers to create on each host */

/* types */
typedef struct {
   int size;
   int elem[MAX_SIZE];
} vec;
newtype void SUBTASK;
newtype void RESULT;
newtype void INPROGRESS;
newtype int  SUM_T;
newtype int  SIZE_T;

/* function declarations */
void worker();
void monitor();
void init(vec *);
void part1(vec*, vec*);
void part2(vec*, vec*);
SUM_T sumvec(vec);
void switch_some();
```

```
LindaMain (argc, argv)
int argc;
char* argv [];
{
  int w, host, lpid, f_id;
  SUM_T sum, total_sum, correct_sum;
  SIZE_T size, total_size;
  vec task;

  /* create one monitor process on each host */
  for (host=0;  host<ftl_num_hosts();  host++) {
    lpid = new_lpid();
    f_id = new_failure_id();
    ftl_create_user_thread(monitor, "monitor", host, lpid, f_id, 0, 0, 0);
  }

  /* create WORKERS_PER_HOST workers on each host */
  for (host=0;  host<ftl_num_hosts();  host++) {
  for (w=0; w < WORKERS_PER_HOST; w++) {
      lpid = new_lpid();
      ftl_create_user_thread(worker, "worker", host, lpid, 0, 0, 0, 0);
  }
  }

  /* initalize the vector */
  init(&task);
  correct_sum = sumvec(task);        /* check answer later with this */

  /* deposit the task into TS */
  < true => out(TSmain, SUBTASK, task); >

  /* wait until the sums have come in from all subtasks */
  do {

    < in(TSmain, RESULT, ?sum, ?size) => skip >
    total_sum += sum;
    total_size += size;

  } while(total_size < MAX_SIZE);

  printf("The sum of the %d elements is %d\n", MAX_SIZE, total_sum);
  if (total_sum ≠ correct_sum)
    ftl_exit("incorrect sum", 1);
  else
    ftl_exit(NULL, 0);      /* halt the worker threads */
}
```

```
void
worker()
{
   int r, host = ftl_my_host(), lpid = ftl_my_lpid();
   vec task, task1, task2;
   SUM_T sum;
   SIZE_T size;

   /* here we will put an extra LPID field in the INPROGRESS
    * tuple to ensure we withdraw our INPROGRESS tuple, not
    * another worker's from this host.
    */

   for (;;) {

      < in(TSmain, SUBTASK, ?task) =>
         out(TSmain, INPROGRESS, task, lpid, host); >

      if (SMALL_ENOUGH(task)) {

         sum = sumvec(task);
         size = task.size;

         < in(TSmain, INPROGRESS, ?vec, lpid, host) =>
            out(TSmain, RESULT, sum, size); >
      }
      else {
         part1(&task,&task1);
         part2(&task,&task2);

         < in(TSmain, INPROGRESS, ?vec, lpid, host) =>
               out(TSmain, SUBTASK, task1);
               out(TSmain, SUBTASK, task2);
         >

      }
   }
}
```

```
void
monitor(int failure_id)
{
   int lpid=ftl_my_lpid(), failed_host, my_host=ftl_my_host();
   vec task;
   SUM_T sum;

   for (;;) {

      /* wait for a failure */
      < in(TSmain, FAILURE, failure_id, ?failed_host) => skip >

      /* Regenerate all subtasks that were inprogress on the failed
       * host.  Note that since the AGS is not yet implemented in
       * expressions we have to test to see if the formal in the
       * inp was set.  To do this, we set task.size to an illegal
       * value; if it is still this after the inp then we know it failed.
       */

      do {

         task.size = ILLEGAL_SIZE;

         < inp(TSmain, INPROGRESS, ?task, ?lpid, failed_host) =>
            out(TSmain, SUBTASK, task); >

      } while (task.size ≠ ILLEGAL_SIZE);
   }
}

/* Initialize the vector randomly */
void
init(vec *task)
{
   int i, count=0;

   task→size = MAX_SIZE;
   /* seed with elements in [MIN_ELEM,MAX_ELEM) */
   for (i=0; i<MAX_SIZE; i++)
      task→elem[i] = MIN_ELEM + (random() % (MAX_ELEM−MIN_ELEM) );
}
```

```
/∗ Fill the first half of t into t1 ∗/
void
part1(vec ∗t, vec ∗t1)
{
   int i, mid = t→size / 2;

   t1→size = mid;

   for (i=0; i<mid; i++)
      t1→elem[i] = t→elem[i];
}

/∗ Fill the second half of t into t2 ∗/
void
part2(vec ∗t, vec ∗t2)
{
   int i, mid = t→size / 2;

   t2→size = t→size − mid;

   for (i=mid; i<t→size; i++)
      t2→elem[i−mid] = t→elem[i];
}

/∗ Sum up the elements in task ∗/
SUM_T
sumvec(vec task)
{
   SUM_T sum=0; int i;

   for (i=0; i<task.size; i++)
      sum += task.elem[i];

   return sum;
}
```

APPENDIX G

FT-LINDA BARRIER EXAMPLE

```
/* Fault-tolerant barrier. */

#ttcontext barrier
#include <stdio.h>
#include "ftlinda.h"

#define    NUM_COLS        64
#define NUM_ROWS        8
#define FAIL_ROLL       (random() % 8)

/* types */
newtype int ROW_T[NUM_COLS];
newtype ROW_T ARRAY_T[NUM_ROWS];
newtype void ARRAY;
newtype void REGISTRY;
newtype void WORKER_DONE;

/* function declarations */
void worker();
void monitor();
void init(ARRAY_T);
int converged(ARRAY_T, int);
int compute(ARRAY_T, int);
static void maybe_fail(int);

/* Note: since array subscripts have not yet been implememted
 * in the AGS parsing code, we have to maintain an extra variable
 * to use in TS operations and then copy to and from outside the AGS.
 */
```

```
LindaMain (argc, argv)
int argc;
char* argv [];
{
  int i, w, host, lpid, f_id, iter=1;
  ARRAY_T a;
  ROW_T r;

  /* initalize the array and place it in TS */
  init(a);
  for (i=0; i<NUM_ROWS; i++) {
  (void) memcpy(r, a[i], sizeof(r));   /* copy a[i] for use in AGS */
  < true => out(TSmain, ARRAY, iter, i, r); >
  }

  /* create one monitor on each host */
  for (host=0;  host<ftl_num_hosts();  host++) {
    lpid = new_lpid();
    f_id = new_failure_id();
    ftl_create_user_thread(monitor, "monitor", host, lpid, f_id, 0, 0, 0);
  }

  /* create NUM_ROWS workers and their registry tuples */
  for (w=0; w<NUM_ROWS; w++)  {
    host = w % ftl_num_hosts();

    /* must create registry tuple before worker! */
    < true => out(TSmain, REGISTRY, host, w, iter); >

      lpid = new_lpid();
      ftl_create_user_thread(worker, "worker", host, lpid, w, 0, 0, 0);
  }

  /* wait until all workers are done */
  for (w=0; w<NUM_ROWS; w++)  {
    < in(TSmain, WORKER_DONE, ?w) => skip >
  }

  printf("Program %s is all done\n", argv[0]);
}
```

```
/* worker(id) updates a[id]) */
void
worker(int id)
{
   ARRAY_T a;
   ROW_T r;
   int i, iter, host=ftl_my_host();

   /* initialize iter and a */
   < rd(TSmain, REGISTRY, host, id,  ?iter) => skip >

   for (i=0; i<NUM_ROWS; i++) {
   < rd(TSmain, ARRAY, iter, i, ?r) => skip > /* read a[i] from TS into r */
   memcpy(a[i], r, sizeof(r));    /* copy into a[i] in memory */
   }

   while ( !converged(a, iter) ) {

   maybe_fail(id);

   compute(a, id);         /* update a[id] in local memory */
   memcpy(r, a[i], sizeof(r));

   /* atomically deposit my row for next iteration & update my registry */
   < true =>
      out(TSmain, ARRAY, PLUS(iter,1), id, r);
      in(TSmain, REGISTRY, ?host, id, iter);
      out(TSmain, REGISTRY, host, id, PLUS(iter,1));
   >

   /* Barrier: wait until all workers are done with iteration iter */
   for (i=0; i<NUM_ROWS; i++) {
      < rd(TSmain, ARRAY, PLUS(iter,1), i, ?r) => skip >
      memcpy(a[i], r, sizeof(r));
   }

   /* Garbage collection on last iteration */
   if (iter > 1) {
      < true => in(TSmain, ARRAY, MINUS(iter,1), id, ?ROW_T); >
   }
   iter++;
   }

   < true => out(TSmain, WORKER_DONE, id); >
}
```

```
void
monitor(int failure_id)
{

#define ILLEGAL_WORKER        −1

    int failed_host, lpid, w, iter, my_host=ftl_my_host();

    for (;;) {

        /* wait for a failure */
        < in(TSmain, FAILURE, failure_id, ?failed_host) => skip >

    /* try to recreate all failed workers found on this host */
        do {

        w = ILLEGAL_WORKER;

        < inp(TSmain, REGISTRY, failed_host, ?w, ?iter) =>
        out(TSmain, REGISTRY, my_host, w, iter);
        >

        if (w ≠ ILLEGAL_WORKER) {
        lpid = new_lpid();
        ftl_create_user_thread(worker, "worker", my_host, lpid, w,
                            0, 0, 0);
        }

        } while (w ≠ ILLEGAL_WORKER);
    }
#undef ILLEGAL_WORKER

}
```

```
/* Initialize the array somehow */
void
init(ARRAY_T a)
{
   int i,j;

   for (i=0; i<NUM_ROWS; i++)
   for (j=0; j<NUM_COLS; j++)
      a[i][j] = 0x1000*i + j;
}

/* compute the next iteration of a[id]. Just a toy example computation ... */
int
compute(ARRAY_T a, int id)
{
  int j;
  int above, below;

  for (j=0; j<NUM_COLS; j++) {

   above = (id == 0 ? 0 : a[id−1][j]);
   below = (id == (NUM_ROWS−1) ? 0 : a[id+1][j]);

   a[id][j] += (above + below);
  }
}

/* Since this uses a toy example with no real meaning, we will simply
 * converge after a few iterations */
int
converged(ARRAY_T a, int iterations)
{
   return (iterations ≤ 3 ? 0 : 1);
}

static void
maybe_fail(int id)
{
   int host = ftl_my_host();
   /* see again if we should fail our host */
   if ( (host ≠ 0) && (FAIL_ROLL == 0) ) {
      ftl_fail_host(host);
   }
}
```

APPENDIX H

MAJOR DATA STRUCTURES

```
/* Major data structures in the FT-Linda TS managers.  Some of the minor data
 * structures and unimportant fields in the following data structures have been
 * ommitted for brevity and clarity.  They have also been reordered for clarity.
 */

/* request_kind_t tracks the kind of request request_t deals with. */
typedef enum {
   REQ_AGS,          /* a normal < ... => ... > command */
   REQ_NEW_LPID,         /* a request for a logical PID */
   REQ_TS_CREATE,         /* create a replicated TS */
   REQ_TS_DESTROY,         /* destroy a replicated TS */
   REQ_FAIL_NOTIFY,    /* notify TS replicas of a host failure */
   REQ_NEW_FAILURE_ID     /* allocate a new failure ID */
} request_kind_t;
```

```
/* request_t is what is passed from the FT-Linda application to the FT-Linda RTS. */
typedef struct request_t {
   request_kind_t    r_kind;          /* what kind of request */
   guard_kind_t     r_guards_kind;        /* absent, blocking, boolean */
   resilience_t     r_guards_resilience;    /* resilience of guards */
   scope_t          r_guards_scope;        /* scope of guards */
   char             r_filename[MAX_FILENAME+1]; /* filename of request*/
   int              r_starting_line;    /* line <...> started on */
   int              r_num_branches;             /* number of branches */
   branch_t         r_branch[MAX_BRANCHES]; /* each guard => body */
   int          r_branch_chosen;    /* which branch did we do? */
   TS_HANDLE_T       r_ts_handle;        /* which TS to destroy */
   UCHAR          r_guard_return_val;    /* return val for => */
   /* r_id is used by non-AGS requests depending on what its r_kind is:
    *
    *    REQ_TS_ID:       TS id allocated
    *     REQ_LPID:       LPID allocated
    *  REQ_CREATE:        TS index of created TS
    *  REQ_DESTROY:        TS index of destroyed TS
    *  REQ_FAIL_NOTIFY:    host that failed
    *  REQ_NEW_FAILURE_ID:    failure ID allocated
    *
    */
   int          r_id;
   int          r_lpid;            /* LPID of request originator */
   int          r_rep_seqn;        /* sequence number for id */
   int          r_next_offset;        /* next offset into r_actual[]*/
   /* unresolved opcode args, i.e. one where at least one argument is
    * P_FORMAL_VAL so the GC couldn't evaluate. */
   opcode_args_t       r_opcode_args [MAX_OPCODES][MAX_OPCODE_ARGS];
   int              r_cookie;              /* magic cookie to try to detect
                    * corruption of request
                    * structure. */
   int              r_times_called;        /* times the AGS has been
                                   * executed */
   int              r_host;                /* host the request sent from*/
   double              r_pad1;                 /* ensure following aligned*/
   UCHAR                r_formal[FORMALSIZE];   /* area to store all the
                                        * formals for the
                                        * chosen branch. */
   double              r_pad2;                 /* ensure following aligned*/
   UCHAR         r_actual[MAX_ACTUALS_SIZE];
} request_t;
```

```
/*
 * a branch is one guard => body
 */
typedef struct branch_t {
   UCHAR    b_guard_present;          /* is there a guard? */
   UCHAR      b_guard_negated;          /* is guard negated with "not" ??*/
   op_t    b_guard;               /* guard of the branch */
   op_t    b_body[MAX_BODY];         /* body of the branch */
   int       b_body_size;              /* no of ops in body  */
   int       b_body_next_idx[MAX_BODY];/* ordering of body ops; 0, then
                   * next_idx[0], ...  Need cause a
                   * move must generate outs before
                   * next op */
   int        b_formal_offset[MAX_FORMALS]; /* offset for each formal
                                   * into r_formal[]. */
   stub_t    *b_stubptr;              /* RTS ptr to branch stub */
} branch_t;


/* param_t tracks what type of parameters each TS op have.  The values of some parameters
 * (P_FORMAL_VAL below) will not have their values known until the request has been received
 * at each replica, since they are a reference to the value of a variable that was a formal
 * in an earlier op in the same <...>.  These can occur either as
 * parameters or as arguments to an opcode parameter.  For example, in
 *
 *    < in(FOO, ?x) => out(FEE, x, MAX(X,1) ) >
 *
 * x is P_FORMAL in the in(FOO...) but in out(FEE... is P_FORMAL_VAL
 * both as a parameter by itsself and as an argument to opcode MAX.
 * The order of the literals used here is important.  Opcodes must come last,
 * and the first opcode must be P_OP_MIN.  This is so the RTS can quickly test
 * whether or not a param is an opcode. */

typedef enum {
   P_TYPENAME,              /* Linda type, actual or formal */
   P_FORMAL_VAR,           /* ?var */
   P_VAL,              /* constant (later expr?) */
   P_FORMAL_VAL,           /* val of var that was a formal
                * earlier in the same branch */
   P_OP_MIN, P_OP_MAX, P_OP_MINUS, P_OP_PLUS, /* P_OP_xxx is opcode xxx */
   P_OP_LOOKUP1, P_OP_LOOKUP2,    P_OP_LOOKUP3
} param_t;
```

```
/* optype_t denotes Linda primitives; op_t stores needed info for them. */
typedef enum {   OP_IN, OP_INP, OP_RD, OP_RDP, OP_MOVE, OP_COPY, OP_OUT} optype_t;

/* guard_kind_t tells what kind the guards are (they all must be the same) */
typedef enum {g_absent, g_blocking, g_boolean} guard_kind_t;

/* we track the arguments for opcode calls.  If none of the agruments
 * are P_FORMAL_VAL then they are all known while the request is being
 * filled in by the GC.  In this case the opcode will be evaluated
 * and the param listed as P_VAL.  Thus, P_OP... params only occur
 * when arg(s) are P_FORMAL_VAL.  (MAY CHANGE FOR SIMJPLICITY) */
typedef struct {
   int oa_formal_index;    /* index into request.bindings if param is FORMAL_VAL
                 * or -1 if arg val is in op_arg_value.*/
   int    oa_op_arg_value;    /* LIMITATION: only ints for opcode args
              * for now.  Could later make this a union. */
} opcode_args_t;

/*  A stub_t variable represents one branch of a request in the RTS.  It is
 *  enqueued on a queue based on the hash value of the branch's guard.  */
typedef struct stub_t {
   struct request_t  *st_request;    /* request for the given stub */
   int        st_branch_index;    /* branch # of corresponding branch for this stub */
   BOOL    is_blocked;        /* is this blocked? else on candidateQ*/
} stub_t;

/* ts_t is a tuple space.  */
typedef struct ts {
   Q_t    ts_blocked[MAX_HASH];    /* stubs for blocked guards */
   Q_t    ts_tuples[MAX_HASH];    /* tuples in the TS */
} ts_t;
```

```
/* op_t is the data structure for both ops in an AGS and also tuples in TS.  If the op is
 * in an AGS then the actuals' data will be stored in the request's r_actual[] area, otherwise
 * the tuple's actuals will be stored in op.o_actual[].  When the TS managers create a tuple from
 * an out op they allocate an op with enough room at the end for o_acutal[] to fit all the actual data.
*/
typedef struct op_t {
   Q_t         o_links;            /* RTS links + key; MUST BE FIRST */
   TIME_T   o_time;              /* RTS time stamp; MUST FOLLOW LINKS */
   TS_HANDLE_T o_ts[2];         /* TS or TSs involved in this op */
   optype_t o_optype;            /*  operator */
   param_t  o_param[NARITY];         /* kind of parameter. ?? */
   /* o_idx[i] is used in different ways, as an index into another array.
    * The way is it used is a function of o_param[i] :
    *      case P_FORMAL_VAR:
    *      case P_FORMAL_VAL:
    *         Here o_idx[i] tells which formal # that parameter i is
    *         for the branch.  This can be used as follows to find
    *         where to store the formal (P_FORMAL_VAR) or where to
    *         retrieve its value from (P_FORMAL_VAL):
    *             formal_idx = tuple.o_idx[i]
    *             offset = b_formal_offset[formal_idx]
    *             formal_address = &(r_formal[offset])
    *      case P_OP_xxx:
    *         Here o_idx[i] tells which unresolved opcode # that parameter
    *         i is for this request.  (Unresolved opcodes are where at
    *         least one of the arguments is P_FORMAL_VAL and thus the
    *         GC can't evaluate it and convert it to P_VAL.)  The info
    *         for this opcode is stored in r_opcode_args[o_idx[i]].
    */
   UCHAR    o_idx[NARITY];
   /* Let start = o_data_start[i] and stop = o_data_stop[i].  Then parameter i's data is in
    * locations [start..stop) of either tuple.o_actual[] or request.r_actual[], depending
    * on which case the parameter is.  */
   UWORD    o_data_start[NARITY];
   UWORD    o_data_stop[NARITY];
   UWORD    o_arity;             /* number of params MAY GO AWAY */
   long     o_polarity;          /* actual/formal; assumes NARITY ≤ 32 */
   int         o_linenum;            /* starting line of op */
   int      o_type;            /* tuple type, aka the tuple's index. */
   int         o_hash;             /*  hash value;f(type,param1) */
   double   o_pad1;                /* ensure o_actual[] aligned */
   UCHAR    o_actual[1];          /* area for actual (P_VAL) data if this op is a tuple. */
} op_t;
```

REFERENCES

[ACG86] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[AD76] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, pages 627–644, October 1976.

[AG91a] Shakil Ahmed and David Gelernter. A higher-level environment for parallel programming. Technical Report YALEDU/DCS/RR-877, Yale University Department of Computer Science, November 1991.

[AG91b] Shakil Ahmed and David Gelernter. Program builders as alternatives to high-level languages. Technical Report YALEDU/DCS/RR-887, Yale University Department of Computer Science, November 1991.

[AGMvR93] Carlos Almedia, Brad Glade, Keith Marzullo, and Robbert van Renesse. High availability in a real-time system. *ACM Operating Systems Review*, 27(2):82–87, April 1993.

[Akl89] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice Hall, 1989.

[And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, California, 1991.

[AO93] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings, Redwood City, California, 1993.

[AS91] Brian G. Anderson and Dennis Shasha. Persistent Linda: Linda + transactions + query processing. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, number 574 in LNCS, pages 93–109. Springer, 1991.

[ASC85] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the 4th ACM SIGACT/SIGMOD Conference on Principles of Database Systems*, 1985.

[Bal90] Henri E. Bal. *Programming Distributed Systems*. Silicon Press, Summit, New Jersey, 1990.

[BHJL86] Andrew P. Black, Norman Hutchinson, Eric Jul, and Henry M. Levy. Object structure in the emerald system. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, Portland, Oregon, September 1986.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[Bjo92] Robert D. Bjornson. *Linda on Distributed Memory Multiprocessors*. PhD thesis, Department of Computer Science, Yale University, November 1992.

[BKT92] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[BLL94] Ralph M. Butler, Alan L. Leveton, and Ewing L. Lusk. p4-linda: A portable implementation of linda. In *Proceedings of the Second International Symposium on High Performance Distributed Computing*, pages 50–58, Spokane, Washington, July 1994.

[BMST92] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary-backup protocols: Lower bounds and optimal implementations. In *Proceedings of the Third IFIP Working Conference on Dependable Computing for Critical Applications*, pages 187–198, Mondello, Italy, 1992.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BS91] David E. Bakken and Richard D. Schlichting. Tolerating failures in the bag-of-tasks programming paradigm. In *Proceedings of the Twenty-First International Symposium on Fault-Tolerant Computing*, pages 248–255, June 1991.

[BS94] David E. Bakken and Richard D. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 1994. To appear.

[BSS91] Kenneth Birman, Andrè Schiper, and Pat Stepheson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[CASD85] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206. IEEE Computer Society Press, June 1985.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 1991.

[CD94] Scott R. Cannon and David Dunn. Adding fault-tolerant transaction processing to Linda. *Software—Practice and Experience*, 24(5):449–466, May 1994.

[CG86] Nicholas Carriero and David Gelernter. The S/Net's Linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[CG88] Nicholas Carriero and David Gelernter. Applications experience with Linda. *ACM SIGPLAN Notices (Proc. ACM SIGPLAN PPEALS)*, 23(9):173–187, September 1988.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CG90] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

[CG93] P. Ciancarini and N. Guerrini. Linda meets Minix. *ACM SIGOPS Operating Systems Review*, 27(4):76–92, October 1993.

[CGKW93] Nicholas Carriero, David Gelernter, David Kaminsky, and Jeffery Westbrook. Adaptive parallelism with Piranha. Technical Report YALE/DCS/RR-954, Yale University Department of Computer Science, February 1993.

[CGM92] Nicholas Carriero, David Gelernter, and Timothy G. Mattson. Linda in heterogenous computing environments. In *Proceedings of the Workshop on Heterogenous Processing*. IEEE, March 1992.

[Cia93] Paolo Ciancarini. Distributed programming with logic tuple spaces. Technical Report UBLCS-93-7, Laboratory for Computer Science, University of Bologna, April 1993.

[CKM92] Shigeru Chiba, Kazuhiko Kato, and Takishi Masuda. Exploiting a weak consistency to implement distributed tuple space. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 416–423, June 1992.

[Com88] Douglas Comer. *Internetworking with TCP/IP*. Prentice-Hall, 1988.

[CP89] Douglas E. Comer and Larry L. Peterson. Understanding naming in distributed systems. *Distributed Computing*, 3(2):51–60, 1989.

[Cri91] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):57–78, February 1991.

[CS93] Leigh Cagan and Andrew H. Sherman. Linda unites network systems. *IEEE Spectrum*, 30(12):31–35, December 1993.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[DoD83] U.S. Department of Defense. *Reference Manual for the Ada Programming Language*. Washington D.C., 1983.

[For93a] Message Passing Interface Forum. Document for a standard messge-passing interface, October 1993. (available from netlib).

[For93b] The MPI Forum. MPI: A message passing interface. In *Proceedings of Supercomputing '93*, pages 878–883, Los Alamitos, California, November 1993. IEEE Computer Society Press.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GK92] David Gelernter and David Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *Proceedings of the Sixth ACM International Conference on Supercomputing*, Washington, D.C., July 1992.

[GMS91] Hector Garcia-Molina and Annemarie Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991.

[Gra78] Jim Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1978.

[Gra86] James N. Gray. An approach to decentralized computer systems. *IEEE Transactions on Software Engineering*, SE-12(6):684–692, June 1986.

[Has92] Willi Hasselbring. A formal z specification of proset-Linda. Technical Report 04–92, University of Essen Department of Computer Science, 1992.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann (Palo Alto, California), 1990.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[HW87] Maurice P. Herlihy and Jeannette M. Wing. Avalon: Language support for reliable distributed systems. In *Digest of Papers, The Seventeenth International Symposium on Fault-Tolerant Computing*, pages 89–94. IEEE Computer Society, IEEE Computer Society Press, July 1987.

[Jac90] Jonathan Jacky. Inside risks: Risks in medical electronics. *Communications of the ACM*, 33(12):138, December 1990.

[Jal94] Pankaj Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

[Jel90] Robert Jellinghaus. Eiffel Linda: An object-oriented Linda dialect. *ACM SIGPLAN Notices*, 25(12):70–84, December 1990.

[JS94] Karpjoo Jeong and Dennis Shasha. PLinda 2.0: A transactional/checkpointing approach to fault tolerant Linda. In *Proceedings of the Thirteenth Symposium on Reliable Distributed Systems*, Dana Point, California, October 1994. To appear.

[Kam90] Srikanth Kambhatla. Recovery with limited replay: Fault-tolerant processes in Linda. Technical Report CS/E 90-019, Department of Computer Science, Oregon Graduate Institute, 1990.

[Kam91] Srikanth Kambhatla. Replication issues for a distributed and highly available Linda tuple space. Master's thesis, Department of Computer Science, Oregon Graduate Institute, 1991.

[Kam94] David Kaminsty. *Adaptive Parallelism with Piranha*. PhD thesis, Department of Computer Science, Yale University, May 1994.

[KMBT92] M. Frans Kaashoek, Raymond Michiels, Henri E. Bal, and Andrew S. Tannenbaum. Transparent fault-tolerance in parallel orca programs. In *Proceedings of the Third Symposium on Experiences with Distributed and*

Multiprocessor Systems, pages 297–311, Newport Beach, California, March 1992.

[KT87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, January 1987.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam81] Butler Lampson. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, pages 246–265. Springer-Verlag, Berlin, 1981.

[Lap91] Jean-Claude Laprie. *Dependability: Basic Concepts and Terminology*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1991.

[Lei89] Jerrold Leichter. *Shared Tuple Memories, Shared Memories, Buses and LAN's—Linda Implementation Across the Spectrum of Connectivity*. PhD thesis, Department of Computer Science, Yale University, July 1989.

[LRW91] LRW Systems. *LRW*^{TM} LINDA-C *for VAX User's Guide*, 1991. Order number VLN-UG-102.

[LS83] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Mis92] Shivakant Mishra. *Consul: A Communication Substrate for Fault-Tolerant Distributed Programs*. PhD thesis, Department of Computer Science, The University of Arizona, February 1992.

[MPS93a] Shivikant Mishra, Larry L. Peterson, and Richard D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1:87–103, 1993.

[MPS93b] Shivikant Mishra, Larry L. Peterson, and Richard D. Schlichting. Experience with modularity in Consul. *Software — Practice and Experience*, 23(10):1059–1075, October 1993.

[MS92] Shivikant Mishra and Richard D. Schlichting. Abstractions for constructing dependable distributed systems. Technical Report 92-19, Department of Computer Science, The University of Arizona, August 1992.

[Nel81] Bruce J. Nelson. *Remote Procedure Call*. PhD thesis, Computer Science Department, Carnegie-Mellon University, 1981.

[Neu92] Peter G. Neumann. Inside risks: Avoiding weak links. *Communications of the ACM*, 35(12):146, December 1992.

[NL91] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, August 1991.

[PBS89] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[Pow91] David Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[PSB$^+$88] David Powell, Douglass Seaton, Gottfried Bonn, Paulo Verissimo, and F. Waeselynk. The Delta-4 approach to dependability in open distributed computing systems. In *Proceedings of the Eighteenth Symposium on Fault-Tolerant Computing*, Tokyo, June 1988.

[PTHR93] Lewis I. Patterson, Richard S. Turner, Robert M. Hyatt, and Kevin D. Reilly. Construction of a fault-tolerant distributed tuple-space. In *Proceedings of the 1993 Symposium on Applied Computing*, pages 279–285. ACM/SIGAPP, February 1993.

[RSB90] Parameswaran Ramanathan, Kang G. Shin, and Ricky W. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer*, 23(10):33–42, October 1990.

[SBA93] Benjamin R. Seyfarth, Jerry L. Bickham, and Mangaiarkarasi Arumughum. Glenda installation and use. University of Southern Mississippi, November 1993.

[SBT94] Richard D. Schlichting, David E. Bakken, and Vicraj T. Thomas. Language support for fault-tolerant parallel and distributed programming. In *Foundations of Ultradependable Computing*. Kluwer Academic Publishers, 1994. To appear.

[SC91] Ellen H. Siegel and Eric C. Cooper. Implementing distributed Linda in standard ML. Technical Report CMU-CS-91-151, School of Computer Science, Carnegie Mellon University, 1991.

[Sch90] Fred Schneider. Implementing fault-tolerant services using the state machine approach. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Seg93] Edward Segall. *Tuple Space Operations: Multiple-Key Search, On-line Matching, and Wait-free Synchronization*. PhD thesis, Department of Electrical Engineering, Rutgers University, 1993.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[TKB92] Andrew S. Tannenbaum, M. Frans Kaashoek, and Henri E. Bal. Parallel programming using shared objects and broadcasting. *IEEE Computer*, 25(8):10–19, August 1992.

[TM81] Andrew S. Tannenbaum and Sape J. Mullender. An overview of the Amoeba distributed operating system. *ACM Operating Systems Review*, 15(3):51–64, July 1981.

[TS92] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *Computer*, 25(6):8–17, June 1992.

[VM90] Paulo Verissimo and José Alves Marques. Reliable broadcast for fault-tolerance on local computer networks. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, pages 54–63, Huntsville, AL, October 1990.

[WL86] C. Thomas Wilkes and Richard J. LeBlanc. Rationale for the design of Aoelus: a systems programming language for the action/object system. In *Proceedings of the 1986 IEEE International Conference on Computer Languages*, pages 107–122, October 1986.

[WL88] Robert Whiteside and Jerrold Leichter. Using Linda for supercomputing on a local area network. In *Proceedings of Supercomputing 88*, 1988.

[XL89] Andrew Xu and Barbara Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing*, pages 199–206, June 1989.

[Xu88] Andrew Xu. A fault-tolerant network kernel for Linda. Master's thesis, MIT Laboratory for Computer Science, August 1988.

[Zav93] Pamela Zave. Feature interaction and formal specifications in telecommunications. *IEEE Computer*, 26(8):20–29, August 1993.

[Zen90] Steven Ericsson Zenith. Linda coordination language; subsystem kernel architecture (on transputers). Technical Report YALEU/DCS/RR-794, Department of Computer Science, Yale University, May 1990.