Supporting Fault-Tolerant Parallel Programming in Linda[1]

David E. Bakken and Richard D. Schlichting

TR 93-18

Abstract

Linda is a language for programming parallel applications whose most notable feature is a distributed shared memory called tuple space. While suitable for a wide variety of programs, one shortcoming of the language as commonly defined and implemented is a lack of support for writing programs that can tolerate failures in the underlying computing platform. This paper describes FT-Linda, a version of Linda that addresses this problem by providing two major enhancements that facilitate the writing of fault-tolerant applications: stable tuple spaces and atomic execution of 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. The design of these enhancements is presented in detail and illustrated by examples drawn from both the Linda and fault-tolerance domains. An implementation of FT-Linda for a network of workstations is also described. The design is based on replicating the contents of stable tuple spaces to provide failure resilience and then updating the copies using atomic multicast. This strategy allows an efficient implementation in which only a single multicast message is needed for each atomic collection of tuple space operations.

June 8, 1993

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

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

1 Introduction

Parallel computing has long been heralded as the wave of the future, particularly as hardware prices have fallen rapidly in recent years. However, this parallelism has manifested itself in a wide variety of diverse architectures, ranging from shared memory multiprocessors to hypercubes to networks of workstations. This variety of architectures has engendered a corresponding variety of different software paradigms for harnessing parallelism. For example, an application on a shared memory multiprocessor might use shared variables to communicate and synchronize, while one on a hypercube would likely be based on message passing. Having such a myriad of approaches complicates the building of portable parallel programs, a major problem given the high cost of software development and the rapid pace at which new and better hardware becomes available.

To help overcome this obstacle, a number of architecture-independent abstractions and languages have been developed [6]. One such language is Linda [16, 13, 17], a *coordination language* that provides a collection of primitives for process creation and interprocess communication that can be added to existing languages. The main abstraction provided by Linda is *tuple space* (TS), a distributed shared memory [32] that can be used by processes to communicate and synchronize despite the lack of physical shared memory. The abstraction is implemented by the Linda runtime system transparently to user processes.

The specific tuple space primitives provided by Linda allow processes to deposit tuples into TS and to withdraw or read tuples whose contents match a pattern specified in the primitive. Thus, tuple space provides for *associative access*, in which information is retrieved by content rather than address. This property, together with the temporal and spatial decoupling inherent in the abstraction, makes it especially easy for use by application programmers. Linda implementations are available on a number of different architectures [11, 27, 8, 10] and for a number of different languages [27, 22, 10, 20].

Despite these advantages, one significant deficiency of Linda as originally defined and commercially available is that failures in the underlying computing platform have not been considered. 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 that use Linda to write general parallel applications cannot take advantage of fault-tolerance techniques to, for example, recover a long-running scientific application after a failure. Second, 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, despite the language's overall appeal in these situations. The significance of these omissions is only likely to become more serious given the trend towards more applications that fall into these categories.

In this paper, we describe FT-Linda, a version of Linda intended to address these problems by providing support for programming fault-tolerant parallel applications. To do this, FT-Linda includes two major enhancements: the ability to have *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 [26]; the latter allows collections of tuple operations to be executed in an all-or-nothing fashion despite failures and concurrency. 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 results in an efficient implementation. These features help distinguish FT-Linda from other efforts aimed at introducing fault-tolerance into Linda [40, 41, 23, 4, 24, 9, 15, 33].

FT-Linda is being implemented using Consul, a communication substrate for building fault-tolerant systems [30, 29], and the *x*-kernel, an operating system kernel that provides support for composing network protocols [21]. Stable TSs are realized using the *replicated state machine approach* [37], where tuples are replicated on multiple processors to provide failure resilience and then updated using atomic multicast.

Atomic execution of multiple tuple operations is achieved by placing some reasonable limits on the type of operations that can be included, and then using a single multicast message to update replicas. Although the language design is general, the focus in the implementation has been on tolerating so-called processor *crash* (or *fail-silent*) failures; such a failure occurs when a processor halts without undergoing any erroneous state transitions [35]. Finally, an interesting side effect of our implementation strategy is better semantics for structuring certain Linda applications even when fault-tolerance is not a concern.

The remainder of this paper is organized as follows. Section 2 overviews Linda and some of the specific problems that failures can cause. The design of FT-Linda is then described in Section 3, with Section 4 giving examples of its use. Section 5 discusses the implementation design and some initial performance results; all the various parts of the system have been implemented, but final integration awaits the porting of Consul to a new version of the *x*-kernel. Finally, Section 6 discusses possible extensions and related work, while Section 7 offers some concluding remarks.

2 Linda and Failures

2.1 Linda Overview

Linda is a system for constructing parallel programs based on a communication abstraction known as *tuple space* (TS) [16, 3, 13]. 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. In essence, TS is a specialized virtual shared memory, where the illusion of a shared resource in a distributed memory system is provided by the Linda runtime system.

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 into TS. 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. **out** is nonblocking.

The **in** operation withdraws a tuple matching the specified parameters from TS; we call these parameters its *pattern*. To match such 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. Actuals are literal values or variables. To match a value in a tuple, the value of that literal or variable actual must be of the same type and value as the corresponding value in the tuple.

Formals are a question mark followed by the name of a variable or type. Formals automatically match any value of the same type. If the formal has a variable name then **in** assigns to the variable the corresponding value from the tuple. For example, if i is declared as an integer variable and b as a boolean, 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. Such an operation may, however, be useful for synchronization.

The Linda **rd** operator is like **in** except it does not withdraw the tuple. Similarly, operators **rdp** and **inp** are exactly like **rd** and **in**, respectively, except that they are nonblocking. Rather, they return a boolean result that indicates if an appropriate tuple was found.

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

`eval`($func(args)$)

will create a process to execute $func$ with $args$ as parameters, and later deposit a tuple with $func$'s return value in TS.

A 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* since two processes that communicate do not have to have overlapping lifetimes. Linda also allows *spatial uncoupling* since 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 Failures

As noted in the Introduction, the effects of processor failures on the execution of Linda programs are 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 should certain key tuples become lost or corrupted by failure. Moreover, such a stable storage facility is a key requirement for 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 [25].

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, as currently defined, the intermediate states resulting from a series of TS operations may be visible to other processes.

Providing 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 [39]. However, Linda with single-op atomicity has been shown to be insufficient to reach distributed

[1] If they do need to know, a process identifier or handle can be included in the tuple. See Section 4.2 for an example.

```
Initialization
    out($“count”, value$)

Inspection
    rd($“count”, ?value$)

Updating
    in($“count”, ?oldvalue$)
    out($“count”, newvalue$)
```

Figure 1: Distributed Variables with Linda

consensus with more than two processes in the presence of failures or with arbitrarily slow (or busy) processors [38]. The key is lack of sufficient 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 of these paradigms are used to solve a wide variety of problems, meaning that deficiencies here can be viewed as applying 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 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; if the old value is not of interest, then the parameter $?oldvalue$ could be simply ?**int**. 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 since its only representation is in the local 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.

Bag-of-Tasks. Linda lends itself nicely to a method of parallel programming called the bag-of-tasks or replicated worker programming paradigm [3, 12]. 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 which processor computes it and at what time. Among the advantages of this programming approach are transparent scalability, automatic load balancing, ease of utilizing idle workstation cycles [18, 14], and, as discussed below, easy extension to fault-tolerant operation.

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. The initial step is to withdraw a tuple describing the subtask to be performed; the label “work” is used as a distinguishing mark to identify tuples

```
process worker
    while true do
        in("work", ?subtask_args)
        calc(subtask_args, var result_args)
        for (all new subtasks created by this subtask)
            out("work", new_subtask_args)
        out("result", result_args)
    end while
end proc
```

Figure 2: Bag-of-Tasks Worker

containing subtask arguments. The worker computes the results, which are subsequently output to TS with an identifying label. Also, any new subtasks 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.

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. The second 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.

2.3 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 a general technique called the *replicated state machine approach* (SMA) [37]. In this technique, an application is represented as a state machine that maintains state variables and makes modifications in response to commands from other state machines or the environment. To provide resilience to failures, the state machine is replicated on multiple independent processors, and an *ordered atomic multicast* is used to deliver commands to all replicas reliably and in the same total order.[2] If the commands are deterministic and executed atomically with respect to concurrent access, then the state variables of each replica with remain consistent. The SMA is the basis for a large number of fault-tolerant distributed systems [7, 30, 34].

[2] This ordering can be relaxed in some cases; see [37].

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 [19, 26]. 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, in some sense, "less" atomicity.

What we believe is a good compromise is based on the specifics of the SMA. In that scheme, each individual command to a replicated state machine meets the kind of atomicity requirements that are our goal. 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 performed by serial processing that does not allow concurrent access. 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, and then executed in sequence as dictated by the serial order in which commands are processed at each replica. This technique has the virtue of being simple to implement and requires less message passing than transactions, 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. 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.

3 FT-Linda

FT-Linda is a variant of Linda designed to facilitate the construction of fault-tolerant applications by including features for data stability and atomic execution. 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 hypercube with a faster and more sophisticated interconnect. 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* [36] 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;[3] extensions to allow such reintegration are considered in Section 6.1.

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

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 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.

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

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 *logical process identifier* (LPID) is specified as an argument in the TS creation primitive (see below). This LPID can be assigned either by the application program or FT-Linda, and must be specified before a process can execute any operations involving a private TS. 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 4.3.

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 required arguments, and returns a TS *handle* that is subsequently passed as the first argument to other TS primitives such as **in** and **out**. An optional third argument used in the case of private TSs is the LPID of the logical process that can access the TS. If no such argument is given and the private attribute is specified, the LPID of the creating process is used. 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 above, 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 also called a *local tuple space*, since its tuples are only stored on the processor on which the TS was created.

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.

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

Table 1: Linda Ops and their FT-Linda Equivalents

3.2.1 Atomic Guarded Statement

An atomic guarded statement (AGS) provides all-or-none 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**, **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 proceeds with the next statement.

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 be 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 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**(. . .)", a matching tuple still gets withdrawn from TS and any formals assigned their corresponding values.

An atomic guarded statement 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 involving data flow between local and replicated TSs. These restrictions are explained further in Section 5.3.2.

Finally, our implementation strategy dictates that Linda TS operations not appear outside of an AGS. Table 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 Section 2.2. 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. To solve this problem, 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 and can be used to regenerate the lost subtask tuple should a failure occur. The code for the static version of the bag-of-task worker demonstrating this technique is shown in Figure 3.

```
# $TSmain$ is $\{global, stable\}$
process worker()
    while true do
        $\langle$ in ($TSmain$, "$work$", $?subtask\_args$)
            $\Rightarrow$ out ($TSmain$, "$in\_progress$", $my\_hostid$, $subtask\_args$) $\rangle$
        calc ($subtask\_args$, var $result\_args$)
        $\langle$ in($TSmain$, "$in\_progress$", $my\_hostid$, $subtask\_args$)
            $\Rightarrow$ out ($TSmain$, "$result$", $result\_args$) $\rangle$

    end while
end worker
```

Figure 3: Lost Tuple Solution for (Static) Bag-of-Tasks Worker

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* that executes on each machine hosting a TS replica. The code for this process is shown in Figure 4. The invocation $failure\_ts$ is a function in the runtime system that specifies that failure tuples be deposited into $TSmonitor$.

```
process monitor()
    $my\_lpid = decl\_lpid()$
    $TSmonitor$ = $ts\_create$(volatile,private)
    $failure\_ts(TSmonitor)$
    while true do
        in($TSmonitor$,"$failure$",$?host$)
        # regenerate all $in\_progress$ tuples we find from the failed host
        while $\langle$ inp($TSmain$, "$in\_progress$", $host$, $?subtask\_args$)
                     $\Rightarrow$ out ($TSmain$, "$work$", $subtask\_args$) $\rangle$ do
            noop
        end while
    end while
end monitor
```

Figure 4: Monitor Process

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

Disjunctive Case. The AGS has a disjunctive case that allows more than one guard/tuple pair, as shown in Figure 5. A process executing this statement blocks until at least one guard succeeds, or all guards fail. To simplify the semantics, 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**). If the guards are all **true**,

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

Figure 5: AGS Disjunction

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 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, 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 4.2.

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

$$transfer\_op\texttt{(}TSfrom\texttt{,}\ \ TSto\ \ \texttt{[,}\ \ in\_pattern\texttt{]}\ \ \texttt{)}$$

Here, $transfer\_op$ is either **move** or **copy**, $TSfrom$ is the source TS , and $TSto$ is the destination TS. The $in\_pattern$ 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 TS operation. If the pattern is present, only matching tuples are moved or copied; otherwise, the operation is applied to all the tuples in the source TS. Since the pattern in a transfer command may match more than one tuple, any formal variables in the $in\_pattern$ 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. For example, a single **move** (or **copy**) operation can transfer an arbitrary number of tuples with an arbitrary number of patterns. Indeed, the process that performs the **move** does not have to be aware of the different kinds of patterns currently present in $TSfrom$.

As an example of how a **move** primitive might be used in practice, consider the dynamic bag-of-tasks application from Section 2.2. Recall that this particular paradigm suffered from both the lost tuple and the duplicate tuple problems. A way to solve these problems are shown in Figure 6. This is similar to the static case shown in Figure 3 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 the final AGS that performs this task, the subtask will be regenerated as before, and another worker will compute the same result and generate the same new subtasks. However, if the worker fails after the final AGS , then the new subtask tuples are already in a stable TS.

```
process worker()
    $my\_lpid = decl\_lpid()$
    $TSscratch$ = $ts\_create$(volatile,private)
    while true do
        ⟨ in ($TSmain$, “$work$”, ?$subtask\_args$)
            ⇒ out ($TSmain$, “$in\_progress$”, $my\_hostid$, $subtask\_args$) ⟩
        calc ($subtask\_args$, var $res\_args$)
        for ($all\ new\ subtasks\ created\ by\ this\ subtask$)
            out($TSscratch$, “$work$”, $new\_subtask\_args$)
        out($TSscratch$, “$result$”,$result\_args$)
        ⟨ in($TSmain$, “$in\_progress$”, $my\_hostid$, $subtask\_args$)
            ⇒ move ($TSscratch$, $TSmain$) ⟩

    end while
end worker
```

Figure 6: Fault-Tolerant (Dynamic) Bag-of-Tasks Worker

Finally, note that a monitor process similar to that used with the static worker case would be needed here as well.

3.3 Other Features

FT-Linda has a number of other features, most of which are derived from the way the implementation totally orders TS operations at each replicated TS. First, **inp** and **rdp** in our scheme provide absolute guarantees as to whether there is a matching tuple, a property that we call *strong inp/rdp semantics*. Of all other distributed Linda implementations of which we are aware, only [4] offers similar semantics. Other implementations either do not provide **inp** and **rdp** or provide only a “best effort” attempt to find a matching tuple. (In the latter case, a return value of **false** does not guarantee the lack of a matching tuple at the time the operation was invoked.) Semantics of the type provided by FT-Linda can be very useful since 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 below in Section 4.3.

4 Examples

This section presents additional examples of how FT-Linda can be used for fault-tolerant parallel programming. First, a fault-tolerant version of a generic divide and conquer algorithm is given; like the bag-of-tasks, this example serves to demonstrate how these extensions can be used for common Linda programming paradigms. Then, we examine an FT-Linda implementation of a replicated server, a common way to structure servers in a distributed system to achieve fault-tolerance. The final example is a recoverable server. It uses distributed consensus, as well as oldest matching semantics to preserve causality.

4.1 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 [27]. 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 7.

```
loop forever
    in(“$task$”, $?task$)
    if($small\_enough(task)$)
        out(“$answer$”, $result(task)$)
    else
        out(“$task$”, $part1(task)$)
        out(“$task$”, $part2(task)$)
    end if
end loop
```

Figure 7: Linda Divide and Conquer Worker

An FT-Linda solution that provides tolerance to processor crashes is given in Figure 8. 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 above would be used to regenerate lost subtask tuples upon failure.

```
loop forever
    $\langle$ in(“$task$”, $?task$) $\Rightarrow$ out(“$in\_progress$”, $task, my\_hostid$) $\rangle$
    if($small\_enough(task)$)
        $answer = result(task)$
        $\langle$ in(“$in\_progress$”, $task, my\_hostid$) $\Rightarrow$ out(“$answer$”, $answer$) $\rangle$
    else
        $task1 = part1(task)$
        $task2 = part2(task)$
        $\langle$
            in(“$in\_progress$”, $task, my\_hostid$) $\Rightarrow$
                out(“$task$”, $task1$)
                out(“$task$”, $task2$)
        $\rangle$
    end if
end loop
```

Figure 8: FT-Linda Divide and Conquer Worker

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

4.2 Replicated Server

In this example, a process implements a *service* that is invoked by client processes by issuing *commands*. 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, commands must be executed in the same order at every replica.[4]

The key to implementing this approach in FT-Linda is ordering the commands in a failure-resilient way. We accomplish this by using the Linda distributed variable paradigm in the form of a *sequence tuple*. The value of this tuple starts at zero and is incremented each time a client makes a request. Thus, there is exactly one request per sequence number, and a sequence number uniquely identifies a request. Given the oldest-matching semantics implemented by FT-Linda, 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 replicas process each command 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:

```
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 9. 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 done 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 replicas would block forever.

```
⟨ in ("sequence", sid, ?seq) ⇒
      out ("sequence", sid, PLUS(seq, 1) )
      out ("request", sid, seq, cmd_name, cmd_id, args)
⟩
if (a reply is needed for cmd)
      in("reply", sid, seq, ?reply_args)
```

Figure 9: Client Request

Two additional aspects of the client code are worth pointing out. First, note that the client includes a cmd_name and cmd_id in the request tuple. This information specifies which of the commands implemented 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

[4] This is, of course, just another instance of the state machine approach, but one that is implemented *using* FT-Linda rather than as part of the language implementation itself.

in the value of the sequence tuple that was withdrawn in the previous **in** being incremented prior to being 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 10. The server waits for a command with the current sequence number using the disjunctive version of the AGS, one branch (i.e., guard/body pair) for each command implemented by the server. When a command with the current sequence number arrives, the server uses the assignment of the cmd_id in the tuple to variable $cmdnum$ to record which command was invoked. This tactic is necessary since 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 command. This procedure performs the computation, and, if necessary, deposits a reply tuple.

```
$time = 0$
loop forever
    $\langle$
        rd($TSmain$, $“request”$, $my\_sm\_id, time, cmd_1, ?cmdnum, ?x$)
    or
        ...
    or
        rd($TSmain$, $“request”$, $my\_sm\_id, time, cmd_n, ?cmdnum, ?a, ?b, ?c$)
    $\rangle$
    case $cmdnum$ of
        # each cmd does out($“reply”$, $my\_sm\_id, time, reply\_args$) if that cmd
        # returns an answer to the client
        $1:$  $cmd_1(x)$
              ...
        $n:$  $cmd_n(a,\ b,\ c)$
    end case
    $time = time + 1$
end loop
```

Figure 10: Server Replica

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 works for the extra reply tuples.

4.3 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 services one command, and that the command requires a reply; multiple commands 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_TS_handle = ts_create(stable, private, server_id)
out(TSmain,“server_handle”, server_id, server_TS_handle)
out(server_TS_handle,“state”, server_id, initial_state)
out(TSmain, “server_registry” ,server_id, host)
eval(server, server_id) on host
```

This creates a private but stable tuple space for use by the server, places the handle for this tuple space and the initial state of the server in a globally-accessible TS, and then creates the server. Note that **eval** has been extended to include the host on which the process is to be started.

The code used by client processors to request service is:

```
out(TSmain, “request”, server_id, args)
in(TSmain, “reply”, server_id, ?reply_args)
```

Note that no sequence tuple is needed for the client of a recoverable server since there is only one actual server process at any given time.

The server itself is given in Figure 11. The server first declares the identifier passed as an argument as its LPID to the runtime system. This step is needed to allow subsequent access to the private TS created upon initialization. The process then 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 command, updates the state tuple, and outputs a reply.

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 regenerated. 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 is accomplished by maintaining a *registry tuple* for each server. This tuple contains the identifiers for both the server and the processor on which it is executing.

A monitor process strategy similar to the bag-of-tasks example is used to implement these two actions. There is one such process on each processor hosting a copy of the replicated TS; 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.

```
process server(my_id)
    # Declare LPID to runtime
    decl_lpid(my_id)
    # Read in handle of private TS
    rd(TSmain, "server_handle", my_id, ?TSserver)
    # Read in state
    rd(TSserver, "state", my_id, ?state)
    loop forever
        ⟨ in(TSmain, "request", my_id, ?args) ⇒
            out(TSmain, "in_progress", my_id, args)
        ⟩
        # calculate command & its reply, change state, do output
        ...
        ⟨ in(TSmain, "in_progress", my_id, cmd, args) ⇒
            out(TSmain, "reply", my_id, reply_args)
            in(TSserver, "state", my_id, ?old_state)
            out(TSserver, "state", my_id, state)
        ⟩
    end loop
end process
```

Figure 11: Recoverable Server

The code is shown in Figure 12. 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.

5 Implementation Design

5.1 Overview

The implementation of FT-Linda consists of four major components. The first is a *precompiler*, which 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 implements 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 manages the copies of replicated TSs on this host using the state machine approach explained above in Section 2.3. Finally, the fourth part is Consul, which acts as the interface between user processes and the TS state machine, 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

```
process monitor($server\_id$)
    $my\_id$ = $decl\_lpid()$
    $TSmonitor$ = $ts\_create$(volatile, private)
    $failure\_ts(TSmonitor)$ # tell RTS to send failure tuples here
    while true do
        in($TSmonitor$,"$failure$",$?host$)
        # see if server $server\_id$ was running on failed host
        if ( ⟨ rdp($TSmain$, "$server\_registry$", $server\_id, host$) ⇒ skip ⟩ )then
            # Regenerate request tuple if found
            ⟨ inp($TSmain$, "$in\_progress$", $server\_id$, $?command\_args$) ⇒
                out ($TSmain$, "$request$", $server\_id$, $command\_args$) ⟩
            # Attempt to start new incarnation of failed server
            if(⟨ inp($TSmain$, "$registry$", $server\_id, host$) ⇒
                    out($TSmain$, "$registry$", $server\_id$, $my\_host$) ⟩ ) then
                eval($server(server\_id)$) on $host$
            endif
        endif
    end while
end monitor
```

Figure 12: Monitor process

processor failures so that failure tuples can be deposited into the TS specified by the user application.

The runtime structure of a system consisting of N host processors is shown in Figure 13. 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 other parallel architectures. The 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.

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 and HP Snake workstations using the Mach microkernel.

The remainder of this section is organized as follows. First, more detail is provided on the precompiler, FT-Linda library, and the TS state machine; in doing so, the internal representations of tuple spaces and runtime requests resulting from FT-Linda extensions are also discussed. Then, the focus shifts to perhaps the most interesting aspect of the implementation, the way in which atomic guarded statements are processed. This is followed by some initial results on the performance of TS operations; given the current status of the implementation, these numbers measure only the overhead of tuple processing on a single processor, yet are instructive nevertheless.

5.2 Implementation Components

The FT-Linda precompiler, FT-lcc, is a derivative of the lcc precompiler [27, 28]. FT-lcc performs two tasks in particular. First, it analyzes and catalogs the *signatures* of all patterns used in TS operations within

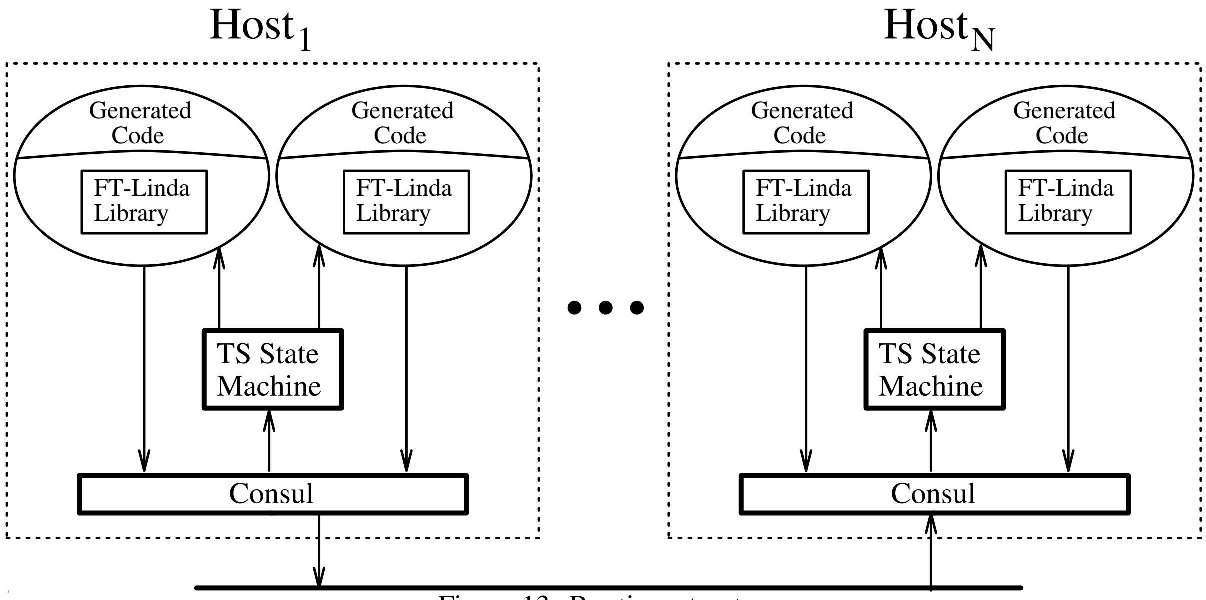

Figure 13: Runtime structure

the program. This information consists of an ordered list of the types for each distinct pattern, and is used primarily for matching purposes. Each signature is assigned a unique ID, called a *tuple index*, which is included in all runtime data structures that involve a tuple or pattern.

Second, as noted above, FT-lcc takes FT-Linda C source code and transforms it into C generated code. Code not related to FT-Linda is generally unchanged. FT-Linda extensions—AG statements, TS creation and deletion primitives, and requests for logical process identifiers—are replaced in the output by generated code that constructs a *request data structure* and then invokes an entry point procedure in the FT-Linda library.

The request data structure contains enough information to process the primitive being executed. The most complex version is, of course, the request data structure associated with an AGS. In addition to general status information, the data structure contains an entry for each branch in the AGS. This entry 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, the global timestamp of the operation, the operator (e.g., **in**), and its tuple index. It also stores the length of the operation and information for each parameter, as well as a data area to store the value of actuals. The information for each parameter includes its polarity (i.e., actual or formal), the offset in the data area of its actual value (if any), opcode arguments, and an identifying integer that is unique within the branch. The request data structure also contains space for the value of any formal parameters; these are filled in by the code that manages the TS being referenced by the operation in question.

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 blocks on a *port* until the request has been completed; this port is allocated by the library code, and included within the request data structure. 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 TSs 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 14. The index into this table for a given tuple is simply the tuple index assigned by the precompiler, while the entries are op data structures. Both have been described above.

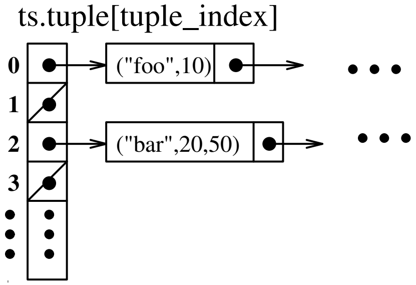

Figure 14: Tuple Hash Table

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 15. 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 since, in the general case, such an AGS may be waiting for any number of matching tuples.

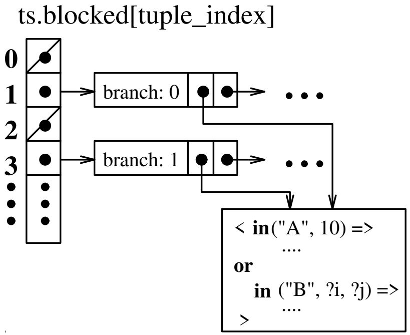

Figure 15: Blocked Hash Table

5.3 AGS Request Processing

5.3.1 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 16. These can be described as follows.

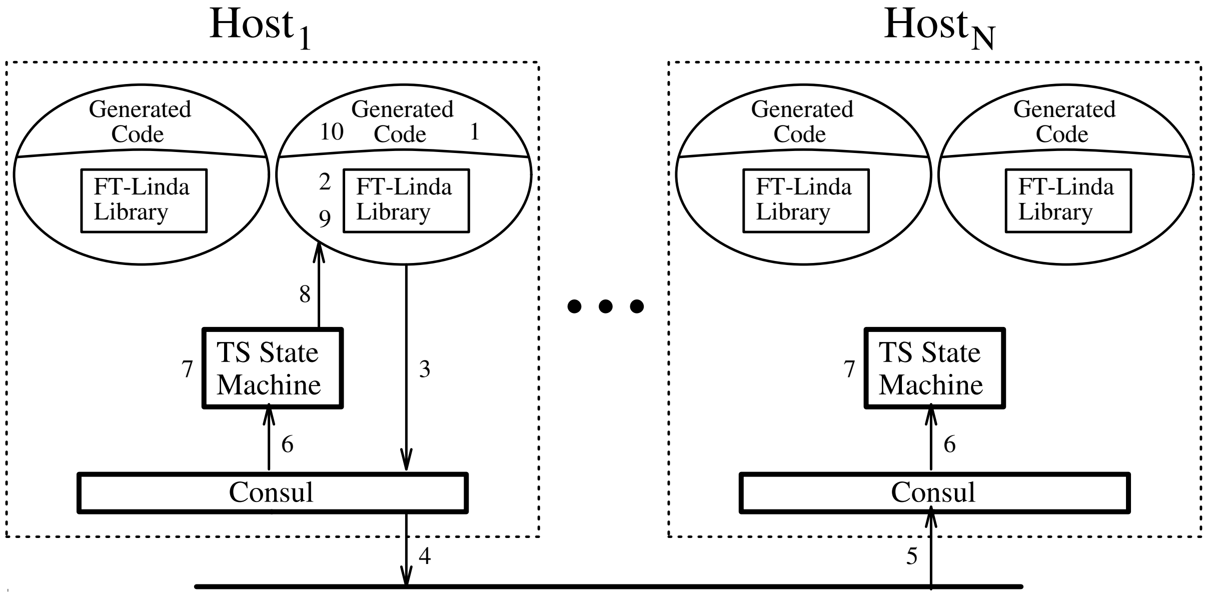

Figure 16: AGS Request Message Flow

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. If the request has blocking guards with no matching tuples, the request is stored in the blocked hash table until a matching tuple becomes available.[5] 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.

[5] Note that this only makes sense as a way to wait for failure tuples since no other user process can access a local TS.

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 request is returned to the FT-Linda library code on the machine from which the request originated. Note that this step and the remaining ones are only executed on the processor from which the AGS originated, since 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 the origin of the restrictions on the way in which TS operations are used in the body 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.

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.

5.3.2 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 TS index of $TSscratch$, $main\_tsidx$ be the TS index of (replicated) $TSmain$, and $tidx$ be the tuple index of the tuple or pattern of the operation in question. The tasks performed by the generated code in each case is identical to the above, and so is 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 hands 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 since 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 can 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 since its value was assigned 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.

As noted above, this 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. Following is an example of an AGS that does not meet this 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)
⟩
```

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. Patterns 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 since 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) ⟩
```

	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 2: FT-Linda Operations on Various Architectures (μsec)

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.

5.4 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 constructing the request data structure in the generated code, traversing the FT-Linda library, performing tuple operations in the TS state machine, and then returning back to the user code. In the tested version, the control flow from the library to the state machine was implemented by procedure call rather than the *x*-kernel.

Table 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 compare favorably with results reported elsewhere [8].

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 [29]. We expect this number to improve once the port of Consul to a faster processor is completed.

Finally, 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 another process 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 since other processes can continue to make progress. To our knowledge, this ability to process TS operations concurrently is unique to FT-Linda.

6 Discussion

FT-Linda and its implementation could be extended in a number of ways to enhance functionality or performance. Here, we discuss two: support for reintegrating failed processors and a technique that allows replicated TSs to be located on only a subset of the processors involved in a computation. In addition, related work is discussed in more detail.

6.1 Reintegration of Failed Processors

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 processor P_i is a snapshot taken after AGS S_1 has been executed but before 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 [29]. 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 the blocked hash table would need to be transferred as well as the actual tuple values. This general scheme for reintegrating failed hosts could also be used to incorporate new tuple servers into the system during execution.

6.2 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. 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 17 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) [31] 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.

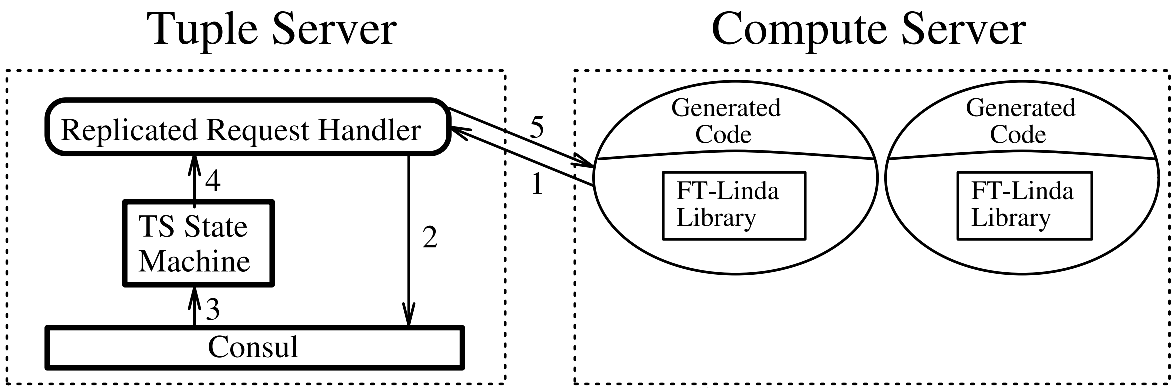

Figure 17: Non-Full Replication

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 attaches a unique ID to the request, and the local TS logic and the TS state machines only process one request with a given ID.

Note that tuple servers in this scenario are very much analogous to the file servers found in a typical workstation environment. The way in which they would be used would likely be similar as well, with a few tuple servers and many compute servers. This relatively small degree of replication for stable TSs should be sufficient for many fault-tolerant applications.

6.3 Related Work

A number of other efforts have addressed the problem of providing support for fault-tolerance in Linda. One class does not extend Linda per se, but rather focuses on adding functionality to the implementation. [41, 40] give a design for making the standard Linda TS stable. The design is based on replicating tuples, and then using locks and a general commit protocol to perform updates. [33] also implements a stable TS by replication, but uses centralized algorithms to serialize tuple operations and achieve replica consistency for single TS operations. [15] addresses the issue of relaxing the consistency of the TS replicas to improve performance. Another project [23, 24] aims to achieve fault-tolerance by checkpointing TS and process states; the scheme is currently only a design and has not been implemented. 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 [38]. Also, unlike our approach, all the designs discussed in this section require multiple messages to update the TS replicas; the sole exception is one special case in [15].

A few efforts have also extended Linda to provide additional atomicity for fault-tolerance purposes. Our initial report [5] introduced an early version of the atomicity that FT-Linda provides; the construct was similar to an AGS , but allowed only a single operation in the body. Plinda [4] allows the programmer to combine Linda tuple space operations in a transaction, and also provides combination commands (e.g., **in-out**) that allow multiple operations to be done atomically. This design is sufficient for fault tolerance—indeed, it is more general than what FT-Linda provides—but the implementation overhead is significant. MOM [9] provides a kind of lightweight transaction. 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.

Other features similar to those provided in FT-Linda have also been proposed at various times. [27] discusses the idea of multiple tuple spaces, and some of the properties that might be supported in such a scheme. [16] briefly introduces *composed statements*, which provide a form of disjunction and conjunction. Support for disjunction has also been discussed in [16, 27] and in the context of the Linda Program Builder [1, 2]. The Program Builder 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.

7 Conclusions

The ability to include fault-tolerance features in Linda applications has the potential to benefit programmers working in a number of areas. One is scientific computing, where an inopportune failure can result in a significant amount of lost computational effort. Another is highly dependable systems, where the otherwise attractive aspects of Linda cannot currently be exploited since fault-tolerance is almost always a requirement in such programs. These and similar examples from other areas argue for a language that combines the virtues of Linda with support for fault-tolerance.

FT-Linda is a version of Linda that attempts to address this need. It does so by including features that are in keeping with Linda's general design philosophy and are efficiently implementable, yet still powerful enough to solve the common problems that arise when processors can fail. The two main features of FT-Linda are stable tuple spaces and atomic guarded statements. The former provide protection against data loss, while the latter support synchronization and the execution of multiple TS operations without having to worry about the effect of failures or concurrency. These features and others are realized by an implementation designed to minimize the number of messages and allow concurrent execution of TS operations on a single processor.

Once the implementation of FT-Linda is complete, our work will concentrate on experimenting with the language to determine the types of applications for which it is best suited. One obvious possibility is the set of parallel applications that are well-adapted to standard Linda, such as those that use the bag-of-tasks paradigm or branch-and-bound algorithms. Another promising area is telecommunications. For example, based on consultation with researchers at GTE Laboratories, we believe that our new language may be appropriate for building fault-tolerant "enhanced calling services" such as call waiting, call forwarding, etc. Our intent is to develop a prototype of such a system as our first major application.

Acknowledgments

Rob Rubin, Rick Snodgrass, and Tom Wilkes provided useful comments on this work. GTE Laboratories and Jerry Leichter provided Linda software used as the basis for FT-Linda.

References

[1] 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.

[2] 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.

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

[4] 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 57 in LNCS, pages 93–109. Springer, 1991.

[5] 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.

[6] H. E. Bal, J. G. Steiner, and A. G. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

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

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

[9] Scott Cannon and David Dunn. A high-level model for the development of fault-tolerant parallel and distributed systems. Technical Report A0192, Department of Computer Science, Utah State University, August 1992.

[10] N. Carriero, D. Gelernter, and T.G. Mattson. Linda in heterogenous computing environments. In *Proceedings of the Workshop on Heterogenous Processing*. IEEE, March 1992.

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

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

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

[14] 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.

[15] 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.

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

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

[18] 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.

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

[20] W. Hasselbring. A formal z specification of proset-Linda. Technical Report 04–92, University of Essen Department of Computer Science, 1992.

[21] 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.

[22] Robert Jellinghaus. Eiffel Linda: An object-oriented Linda dialect. *ACM SIGPLAN Notices*, 25(12):70–84, December 1990.

[23] 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.

[24] Srikanth Kambhatla. Replication issues for a distributed and highly available Linda tuple space. Master's thesis, Department of Computer Science, Oregon Graduate Institute, 1991.

[25] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, January 1987.

[26] B. Lampson. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, pages 246–265. Springer-Verlag, Berlin, 1981.

[27] Jerrold Leichter. *Shared Tuple Memories, Shared Memories, Buses and LAN's—Linda Implementation Across the Spectrum of Connectivity*. PhD thesis, Yale University, Department of Computer Science, July 1989.

[28] LRW Systems. *LRW*^{TM} LINDA-C *for VAX User's Guide*, 1991. Order number VLN-UG-102.

[29] Shivakant Mishra. *Consul: A Communication Substrate for Fault-Tolerant Distributed Programs*. PhD thesis, Department of Computer Science, The University of Arizona, February 1992.

[30] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. Technical Report 91-32, Department of Computer Science, The University of Arizona, 1991.

[31] B. J. Nelson. *Remote Procedure Call*. PhD thesis, Computer Science Department, Carnegie-Mellon University, 1981.

[32] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, August 1991.

[33] 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.

[34] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[35] D. Powell, D Seaton, G. Bonn, P. 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.

[36] 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.

[37] Fred Schneider. Implementing fault-tolerant services using the state machine approach. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[38] Edward Segall. *Tuple Space Operations: Multiple-Key Search, On-line Matching, and Wait-free Synchronization*. PhD thesis, Department of Electrical Engineering, Rutgers University, 1993.

[39] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *Computer*, 25(6):8–17, June 1992.

[40] Andrew Xu. A fault-tolerant network kernel for Linda. Master's thesis, MIT Laboratory for Computer Science, August 1988.

[41] 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.