A Design for a Fault-Tolerant, Distributed Implementation of Linda

Andrew Xu
Oracle Corporation
Belmont, CA 94002

Barbara Liskov
MIT Laboratory for Computer Science
Cambridge, MA 02139

Abstract

A distributed implementation of a parallel system is of interest because it can provide an economical source of concurrency, can be easily scaled to match the needs of particular computations, and can be fault-tolerant. This paper describes a design for such an implementation for the Linda parallel system, in which processes share a memory called the tuple space. Fault-tolerance is achieved by replication: by having more than one copy of the tuple space, some replicas can provide information when others are not accessible due to failures. Our replication technique takes advantage of the semantics of Linda so that processes encounter little delay in accessing the tuple space. In addition to providing an efficient implementation for Linda, the paper extends work on replication techniques by showing what can be done when semantics are taken into account.

1. Introduction

A distributed implementation of a parallel system is attractive for several reasons:

1. Existing uniprocessor computers, possibly heterogeneous, can be used to run large jobs in parallel instead of acquiring expensive multi-processor machines.
2. The number of computers used to perform a task can be scaled to the size of the task.
3. With a proper fault-tolerant mechanism, failures of individual computing nodes, possibly caused by loss of power or hardware mulfunction, will not disrupt program execution.

However, distributed systems also give rise to some problems. The communication overhead may be substantially higher than in a multi-processor system. In addition, networks are susceptible to failures: messages may be lost, duplicated, or delivered out of order, the network may fail in a way that causes the system to be partitioned into subparts that cannot communicate, or the computing nodes may crash. Unless care is taken, the failure of a single component may cause the entire parallel computation to fail.

This paper describes a design for a fault-tolerant, distributed implementation of the Linda parallel programming system [1, 10]. A Linda system consists of processes that share a memory known as the *tuple space*. Our implementation makes the tuple space highly available. High availability is achieved by redundancy: the tuple space is replicated on several nodes so that some replicas can provide information when others become inaccessible due to failures.

Replication provides high availability of data, but may cause data inconsistency among replicas. Failure to deliver messages or network partitions may cause some replicas not to receive needed information; duplicate messages may cause some replicas to receive extra information; a replica may have contain old information after recovery from a failure. We present a method that solves these problems. Our main contribution is a way of organizing the tuple space and carrying out the operations on it that results in low delay for Linda programs. We refer to this part of our mechanism as the *operations protocol*. In addition we also present a *view change algorithm* that is used to reconfigure the system when failures occur; this algorithm is based on earlier work [6, 7, 18], but is tailored to the needs of the operations protocol.

Our method has some attractive properties. First, replication is completely hidden from the user program; the replicated tuple space appears to be a single entity. Second, the tuple space can tolerate simultaneous failures, and progress can be made as long as a majority of the replicas can communicate with one another. Third, little delay is imposed on the user programs. These properties make a distributed implementation of Linda a viable alternative to an implementation on a multi-processor machine.

Several implementations have been provided for Linda. Two of these are designed for networks [1, 4], but neither provides a highly-available tuple space in a general communications networks. Compared with these implementations, our protocol tolerates failures that are common in general networks and provides good performance.

We begin in Section 2 by briefly describing Linda. Section 3 gives an overview of our scheme, which is then described in more detail in Sections 4 and 5. The following sections discuss related work and describe some extensions to our system. We conclude with a summary of what we have accomplished.

2. Linda

A Linda system consists of several processes, which we will refer to as *workers*, and a *tuple space* memory that is shared by the workers. The workers cooperate by communicating through the tuple space. Tuple space operations permit workers to exchange information and also to synchronize.

The tuple space consists of a collection of *tuples*. Each tuple contain a *logical name* followed by one or more data elements, which can be either data values such as "1" and "true", or *formals*, which are typed variables. Examples of valid tuples are ("X", 1, **true**) and ("A", "John", **formal** score).

Workers interact with the tuple space via three operations: **out**, **In**, and **rd**. **Out**(*t*) adds tuple *t* to the tuple space. **In**(*s*) and **read**(*s*) are used extract information from a *matching* tuple in the tuple space. The argument tuple *s* is refered to as a *template*. A template matches a tuple if both have the same logical names, the same number of fields, corresponding fields are type consonant, corresponding data items are equal, and there are no corresponding formals. For example, ("X", **formal** i, 3, 4, 5) matches ("X", 2, 3, **formal** j, 5), but ("X", **formal** i) does not match ("X", **formal** j).

In(*s*) removes some tuple *t* that matches *s* from the tuple space and assigns the values of the actuals in *t* to the formals in *s*. If no matching tuple is available, the executing worker waits until one is; if many matching tuples are available, one is chosen arbitrarily. **Rd**(*s*) is the same as **In**(*s*) except that the matching tuple remains in the tuple space.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125, and in part by the National Science Foundation under grant DCR-8503662.

In addition to these operations, three others, **eval**(*p*), **Inp**(*s*), and **rdp**(*s*), have been proposed [3]. **Eval**(*p*) starts a process to execute procedure *p*; it has little to do with the tuple space, and hence will be ignored. **Inp**(*s*) and **rdp**(*s*) are similar to **In**(*s*) and **rd**(*s*), except they are non-blocking. If there is a matching tuple to *s*, **Inp**(*s*) and **rdp**(*s*) behave exactly the same as **In**(*s*) and **rd**(*s*), respectively. Otherwise, "no_match_found" is signalled. **Inp**(*s*) and **rdp**(*s*) are not included in our protocol; we discuss them in Section 5.2.

3. Overview

Replication is the standard technique for increasing data availability. By replication, we mean maintaining several *physical copies*, usually distributed over a set of nodes at distinct locations, of each *logical tuple*. When a copy of a logical tuple becomes unavailable due to node or network failures, the rest of the copies can still provide information. For simplicity, we assume that the tuple space is uniformly replicated, that is, each replica contains an entire copy of the tuple space. In Section 7, we will see that this constraint can be relaxed so that each tuple can be stored on a subset of the replicas.

Our system consists of a set of tuple space replicas and a set of workers. Each tuple space replica or worker resides on some physical node. A physical node can contain any number of replicas or any number of workers or both. We expect that the number of replicas is relatively small, e.g., 3 or 5. Each replica is identified by its unique *replica id*; the replica ids are totally ordered.

All physical nodes are connected by a communications network. The network may be either a local area net or geographically distributed. Nodes are independent computers that communicate with each other only by sending messages over the network. Nodes can crash, but we assume they are failstop processors [19], i.e., they fail by halting. The network may lose, delay, and duplicate messages, or deliver messages out of order; link failures may cause the network to partition into subnetworks that are unable to communicate with each other. We assume that nodes eventually recover from crashes and partitions are eventually repaired. We rule out byzantine failures [16], in which system components may act in arbitrary, even malicious, ways, because they are rare, and because the algorithms needed to cope with them are expensive.

In general, it is impossible for a node to tell whether the inability to receive a message from some other node is due to a crash of that other node or a network partition. The effect of either failure, as the node perceives it, is the same — no message is received. Therefore, our scheme will not rely on distinguishing crashes from partitions.

As mentioned, our plan is to mask failures by maintaining several copies of the tuple space. Our design is driven by the following goals:

1. Consistency: The replicated tuple space must present a consistent state to the workers. The results of operations must be the same as if there were only one copy of the tuple space. For instance, concurrent **In** operations must not extract the same tuple from different replicas.
2. Availability: The tuple space should have a high probability of being available despite failures. Our goal is that as long as the majority of replicas (e.g., 3 out of 5) can communicate with each other, the tuple space is available.
3. Performance: Operations should perform efficiently to support the requirements of the parallel programming paradigm. In particular, operations should delay workers as little as possible.

Our scheme consists of two parts: the *operations protocol* and the *view change algorithm*. The operations protocol is used to carry out tuple space operations on behalf of workers. Our plan is to store each tuple at every replica. **Out** operations store the new tuple at all replicas, and **In** operations remove the selected tuple from all replicas. A **rd** operation can read from any replica.

Our method causes little delay to workers. An **out** operation is performed in the background and does not delay a worker at all. A worker doing a **rd** need wait only for a response from a single replica. A worker doing an **In** must wait until all replicas are checked for matches. At this point a single matching tuple is selected and the worker is allowed to proceed. The tuple is removed from the replicas in the background.

The view change algorithm is adopted from the virtual partitions protocol described in [6, 7]. It allows the tuple space to continue to be available in spite of failures by reorganizing the replicas into a new *view*. A view is a subset of replicas that are supposed to be able to communicate; it always contains a majority of the replicas. Associated with each view is an unique *viewid*. Later views have larger viewids.

Each worker runs in a view and executes its operations at the replicas in this view. Thus, when we talked about an **out** or **In** occurring at all replicas, we meant that they occurred at all replicas in the worker's current view; similarly, a **rd** operation reads from a replica in the current view. As part of a view change, a new state for the tuple space is selected; this state is guaranteed to contain the effects of **out** and **In** operations that occurred at all replicas in the previous view.

The next sections describes our method in more detail. The operations protocol is described in Section 4; the view change algorithm is described in Section 5.

4. The Operations Protocol

The execution of the operations protocol requires the cooperation of both the workers and the replicas. When a worker issues a tuple space operation, a request for the operation is formed at the worker. Periodically, the requests are sent to each replica in the worker's view, and are executed by the replica. After execution, the replica sends back either a result (if there is one) or a completion acknowledgment.

At each replica, the tuple space is organized as a set of *tuple sets*. Each tuple set contains all tuples with the same logical name. There is a lock associated with each tuple set. When a tuple set is locked by a worker, further **In** operations of all other workers involving that tuple set are blocked until the lock is released by the locking worker.[1] However, **read** and **out** operations are not affected by these locks.

This section describes the execution of operations. We begin by describing how each operation is carried out and the constraints that operation execution must satisfy. Then we discuss processing at workers, and finally processing at replicas. We assume that workers do not fail; we discuss a method to cope with workers' failures in Section 7.

4.1. Executing Operations

Let W be a worker executing an operation. The three operations on a replicated tuple space are implemented as follows.

The **out**(*t*) operation writes to all replicas:

1. The request to execute the operation is broadcast to all replicas in W's view, and W waits for acknowledgments from the replicas.
2. At each replica, *t* is stored into the local copy of the tuple space, and an acknowledgment is sent to W.
3. If W does not receive acknowledgments from all replicas in its view, it repeats the request until all acknowledgments have been received. It is the replicas' responsibility to discard redundant requests for the same **out**.

The **In**(*s*) operation removes the same tuple from each replica. It is done in two phases. First it acquires the locks and reads from the replicas; this is the **In1** phase:

[1] We could use a finer grain of locking in which we lock just the tuples that might match the template; such locks are known as predicate locks [8].

1. W sends template *s* to all replicas in its view.
2. Each replica locks the tuple set with *s*'s logical name and returns a set containing all matching tuples to W; if there are no matching tuples, an empty set is returned. If the tuple set is already locked by another worker, W's request is refused.
3. If all replicas accept the request and there is a non-empty intersection of the tuple sets W received, an arbitrary tuple in the intersection is selected, the actuals of the selected tuple are assigned to the formals of *s*, and phase two starts.
4. If not all replicas have responded within a reasonable time, or if all responded but only a majority of replicas in W's view accept the request, or if all accept but the intersection is empty, W repeats the first phase.
5. Otherwise, all replicas have responded but only a minority have accepted the request. In this case W instructs the replicas to release the locks, and phase one will be repeated after the locks have been released.

In the **In2** phase the tuple is removed from the tuple space:

1. W informs all replicas in the view about the selection in phase one. The replicas remove the selected tuple from their copies of the tuple space, release the lock set during the first phase, and send an acknowledgment to W. The phase is finished only when all replicas have replied. Otherwise, it is repeated until they have. Duplicate requests for the same **In2** are discarded by the replicas.

It would be a violation of our consistency goal for an **In** to delete a different matching tuple from each replica. Instead, the *same* tuple must be removed by all replicas in the view. **In1**'s mission is to ensure that this constraint is met. A selection can be made only when the executing worker has a lock on the same tuple at every replica in its view; a non-empty intersection guarantees this condition. No selection can be made if the intersection is empty; the worker must be blocked until all replicas have replied to the **In1** request and a selection is made.

The locks keep the tuples under consideration from being removed by other concurrent **In** operations. If there are concurrent **In1**'s concerning the same tuple set, each might acquire locks at some replicas, and none would be able to complete. In other words, there would be a deadlock. To resolve such a situation, we release locks when the worker has acquired them only at a minority of replicas; this will enable a worker with a majority to succeed in acquiring locks at all replicas. The probability of several competing workers who repeatedly acquire only a minority of locks can be reduced by introducing a random delay, so that workers make their next attempts to set the lock at different times.

Finally, **Rd**(*s*) selects a matching tuple from any replica:

1. Template *s* is broadcast to all replicas in W's view. Each replica searches for a matching tuple in its local copy of the tuple space. If a matching tuple is found, a copy of it is sent back to W. Otherwise, the replica informs W that no matching tuple is found.
2. If W receives a tuple from some replica, it assigns the actuals of the returned tuple to the formals of *s*, and the operation is complete. Responses from the rest of the replicas are ignored.
3. If no tuple is received within a reasonable time, **rd** is repeated.

4.2. Minimizing Delay

To satisfy our minimal delay requirement, we do processing in the background whenever possible. A worker executing a **rd** must be blocked until the first matching tuple is returned from a replica; a worker executing an **In1** must be blocked until the tuple to be removed is selected. However, a worker issuing an **out** operation need not wait; instead, the operation can be performed in the background while program execution continues. Similarly, there is no need for the executing worker to be blocked while an **In2** is in process.

The background processing of **out** and **In2** introduces concurrency between running a worker and its use of the tuple space. This concurrency must be constrained to preserve Linda semantics. For example, if we do not control concurrent execution, a **rd** operation may read a tuple that was supposed to be removed by a previous **In** operation issued by the same worker because the **In2** has not completed by the time the **rd** is executed.

The correctness condition that our implementation must satisfy is known as *linearizability* [13].[2] We require that each operation be atomic and that there be a serial order for all operations that agrees with the causal order constraints [15] of the workers' programs.

For a single worker, the operations must appear to occur in the order that the worker issues them. We satisfy this constraint in a simple way: we execute the operations of each worker at each replica in the same order as they were issued by the worker.

The inter-worker causal order constraint is the following: Suppose by doing an operation p a worker W_2 observes the effects of some operation o performed by another worker W_1. Note that this can happen only when p is a **rd** or **In** operation and o is an out operation; the observation consists of the **rd** or **In** returning information about the tuple that the **out** added to the tuple space. Now suppose that r is any operation that is causally before o; for example, r might be sequentially before o at W_1. Similarly, suppose q is some operation that is causally after p. Then q must be after r.

The interesting case is when p is an **In** operation. Note that the effect of an **In** operation cannot be observed, but it is possible to observe that the **In** has not yet happened, by reading information from the tuple that the **In** removes. Therefore we must be certain that operations known to be after an **out** cannot observe the non-effect of an **In** that happened before the **out**.

For example, suppose there are two workers using the replicated tuple space and that there is at most one tuple ("x", *), where * is an integer, in the tuple space at any time. The tuple space contains tuple ("x", 1) initially. Workers W_1 and W_2 are the only workers in the system, and are running in parallel. Consider the following program fragments:

W_1	W_2
In("x", **formal** x)	**rd**("x", **formal** u)
out("x", x + 1)	**rd**("x", **formal** v) % expect v ≥ u

where *x*, *u*, and *v* are previously declared integer variables in the workers' program. In this example, the integer value in tuples with logical name "x" increases with time. W_1 modifies *x* in a way that satisfies this constraint; W_2 reads *x* and should not observe a violation of the constraint.

Forcing operations to be executed in order at *each* replica is not sufficient to enforce the above constraint because **rd** can return a value from *any* replica. For example, suppose the operations of W_1 and W_2 are executed as follows: W_1's **In**("x", **formal** x) and **out**("x", x + 1) are executed at R_1, W_2's **rd**("x", **formal** u) is executed at R_1 and returns ("x", 2), and finally, W_2's **rd**("x", **formal** v) is executed at R_2 and returns ("x", 1), which is incorrect.

To prevent this problem, we require that requests for an **out** operation not be sent to any replica until previous **In** operations issued by the same worker have been performed at *all* replicas in the current view. Thus, in the above example tuple ("x", 2) cannot exist at R_2 until ("x", 1) has been removed from both R_1 and R_2. So when **rd**("x", **formal** u) returns with ("x", 2), ("x", 1) has already been removed from every replica and therfore cannot be observed by any later operation.

Finally, we note that **out**s and **rd**s need not be delayed by a lock held by an **In**. For example, suppose a **rd** (at W_1) is happening at the same time as an **In** (at W_2) and that both match the same tuple. If the **rd** obeyed the **In**'s lock, it would be serialized after the **In**.

[2] We consider only worker communication via operators on the tuple space.

Without the lock, the **rd** might be serialized before the **in** (if it returns information based on the tuple that the **in** removes). However, since the **rd** and **in** are concurrent, either outcome is permissible, so there is no need for the **rd** to obey the lock. Basically, there is no causal order between a concurrent **in** and **rd** or between a concurrent **in** and **out**, so either serialization is permitted. Similarly, there is no causal order between a concurrent **rd** and **out**; this is why synchronization is not needed for them.

4.2.1. Summary

The sequential and inter-worker constraints are:

1. The operations of each worker must be executed at each replica in the same order as they were issued by the worker;
2. An **out** operation must not be executed at any replica until all previous **in** operations issued by the same worker have completed at all replicas in the worker's view.

The second constraint may cause a delay in the execution of the worker. The worker need not wait for an **out** operation, but may be delayed by a subsequent **rd** or **in**. This would happen if an earlier **out** were still delayed because an even earlier **in** had not yet completed. Such a situation is likely to be rare.

4.3. Processing at Workers

This section provides a sketch of processing at workers; a more complete description can be found in [21]. Each worker consists of two processes, FG and BG, which communicate through a shared data structure called the *operations log*. The log contains five kinds of requests: **rd**, **out**, **in1**, **unlock**, and **in2**. The latter three requests are used to perform **in** operations: **in1** does phase one, **unlock** releases locks when this is necessary, and **in2** requests are used to do phase two. At any time, the log contains the most recent request, possibly preceded by some requests that are executed in the background (**out** and **in2**). Requests are processed when they are *ready*. An **out** request is *ready* provided all earlier **in2**s have completed; other requests are *ready* if all earlier **out** requests are *ready*.

FG is the foreground process that executes the worker program. When it encounters a tuple space operation, it adds the appropriate request (**rd**, **out**, or **in1**) to the log. Adding the request delays FG if necessary: adding a **rd** request delays it until a matching tuple is available; adding an **in1** request delays it until the tuple to be removed has been selected; adding an **out** request does not delay it, but the request will not be processed until it is *ready*.

BG is the background process. It communicates with the replicas to carry out requests in the log, and removes requests that have been carried out. Periodically it retrieves all *ready* requests from the log, puts them into a message (preserving their order), and sends the message to all replicas in the current view. Then it waits for responses from the replicas. A replica responds only after carrying out all requests in the message; its response explains what it did with the latest request in the message. BG carries out the algorithms described above and synchronizes with FG to allow FG to continue when needed information is available. For example, if BG receives a **rd** response containing a match, it removes the **rd** request from the log and enables FG to continue. If it receives **out** responses from all replicas, it removes the **out** request and all earlier ones from the log. (Removing all requests is appropriate because it has heard from all replicas, which means that all requests have been carried out at all replicas.) If it receives **in1** responses from all replicas, and if all replicas set the lock for this worker and there is a tuple in the intersection of the returned sets, it removes the **in1** request and all earlier requests from the log, adds an **in2** request to the log, and enables FG to continue with the selected tuple. If it must do an unlock, it removes all requests from the log, and adds an **unlock** request; after the unlock has been performed at all replicas, it removes it and puts the **in1** request back in the log.

Thus the log can contain the following contents: An **unlock** is always the only request in the log. Otherwise, there can be zero or one **in2** requests, followed by zero or more **out** requests, followed by a single **rd** or **in1**. If the log contains an **in2** followed by an **out**, the **out** and all requests that follow it are not *ready*; otherwise all requests are *ready*.

One possible response to a message is a "new view" response, informing the worker about a view change. BG puts the viewid of the worker's view in each message it creates. If this viewid does not match that of a receiving replica, the replica sends back the "new view" response, which contains its current view and viewid. When BG receives such a message, it replaces the worker's current view and viewid with those in the message *if* the viewid in the message is greater than its current viewid. (Recall that viewids are ordered and that later views have larger viewids. Therefore, a message with a smaller viewid represents an old state of affairs that is no longer of interest; BG discards such a message.)

The messages that contain requests or responses can be lost, delayed, or duplicated by the network. If BG does not receive all the replies within an expected time interval, it sends new messages containing the requests until it gets replies back from all replicas in its view. This method solves the problems of lost messages, but not of duplicate or delayed messages; in fact, it generates duplicate requests. Duplicate requests are also generated when the worker switches to a new view.

Replicas must not execute certain duplicate requests. For example, if a replica executes an **out** twice, the tuple would be inserted into the tuple space twice. To allow the replica to identify duplicates, we use two mechanisms. First, messages have a unique message identifier (*mid*) so that duplicates created by the network can be recognized and discarded. BG generates a new mid each time it creates a new message; the new mid is larger than any mid it generated before. Responses from replicas contain the mid of the message being replied to; BG discards any response messages that do not have the current mid. This enables it to be sure that the response is really to the current message.

To recognize duplicate requests that arrive in different messages, the worker stamps each modification request with a unique *timestamp*; the timestamp enables a replica to discard a request that has already been processed. The timestamp for an **out** operation request is generated for it when FG adds it to the log. The other requests that have timestamps are **unlock** and **in2**; their timestamps are generated when BG adds them to the log. Timestamps increase with time: later requests receive larger timestamps.

4.4. Processing at Replicas

Replicas are responsible for executing operation requests. In addition, they must discard duplicate requests and notify workers about new views.

To recognize duplicates, each replica maintains a *table* in which it stores information about mids and timestamps. For each worker, the table records the latest mid and timestamp for that worker. If there is no entry for a worker, the worker is associated with a zero timestamp and a zero mid.

When a replica receives a message, it proceeds as follows:

1. If the mid of the message is less than or equal to that stored for its worker in the table, the message is discarded.
2. If the viewid in the message is less than the replica's current viewid, a "new view" message containing the replica's current view and viewid is sent to the worker, and the worker's message is discarded.
3. Otherwise, the replica updates the table to contain the new mid, and then performs (in order) the requests in the message. For each **out**, **unlock**, or **in2** request, it checks the table and ignores the request if its timestamp is less than or equal to that stored for the worker in the table; otherwise it updates the table and performs the request. When all requests are finished, it sends back a response indicating the outcome of the last request in the message. If the last request is an **out**, **unlock**, or **in2**, the

response is just an acknowledgment (and this response is sent even if the request is ignored); for a **rd** request, the response is the matching tuple, or an indication that there is no match; for an **In1**, the response is either the set of matching tuples, or a refusal if the set is already locked for some other worker.

4.5. Optimizations

There are a number of optimizations that can be added to improve performance. For example, BG need not send **rd** requests to all replicas. Instead, these requests can be sent only to the "closest" replica in the current view. Also, when BG sends new messages because some replicas did not respond to a previous message, it can send them only to replicas that did not respond. Both optimizations reduce the number of messages, and consequently the processing required at both replicas and workers. The first one may introduce some delay if the closest replica is slow to respond.

5. Changing Views

Out, **In1**, **In2** and **unlock** requests are *complete* only if the executing worker knows they have happened at every replica in its view. A worker continues to send requests for these operations until all replicas in its current view inform it that the request has been processed. If a view change occurs before a request is complete, the worker redoes the request in the new view.

However, once a request is complete, the worker forgets about it. The job of the view change algorithm is to ensure that the effects of completed requests survive into subsequent views. It does this by ensuring that every view contains at least a majority of replicas and by starting up the new view in the state of a replica that was also a member of the previous view.

This section gives a brief overview of the view change algorithm; additional details can be found in [21]. Intuitively, a view reflects the changing communication capability among members of a partition. When the communication capability inherent in a view appears to have changed, the replicas switch to a new view by executing the view change algorithm. As part of a view change, the view change algorithm generates a new, unique viewid and a new view. Any replica can be the *manager* of the view change algorithm; the manager selects the viewid of the new view. The members of the new view will be the replicas that the manager can communicate with. A new view can be formed only when it contains a majority of the replicas in the original configuration; the viewid of the new view will be larger than that of any previous view. If a manager cannot talk to enough other replicas, it remains in the old view.

It is important to realize that views and partitions are different concepts. Partitions represent the physical configurations of a system while views are what replicas think the system configurations are. For instance, if r_1, r_2, r_3, r_4 and r_5 are replicas of some tuple space, and at some instance there are two partitions (r_1, r_4, r_5) and (r_2, r_3), then the views of r_1, r_4, and r_5 may be $\{r_1, r_4, r_5\}$, and those of r_2 and r_3 may be $\{r_1, r_2, r_3\}$. The inconsistencies between views and partitions result, for example, because changes in network topology happen abruptly and replicas cannot detect the changes instantly.

As part of a view change, the algorithm selects an initial state for the new view; all replicas in the new view will be initialized with this state. The chosen state is the state of the replica in the new view whose previous viewid is greater than or equal to the previous viewids of all other replicas in the new view. Since views contain majorities, the new view will contain at least one replica that was in the previous view. The state will thus be taken from a replica in the previous view. Since requests are complete only if they ran at all replicas in a view, this implies that the chosen state will contain the effects of all requests that completed in the previous view. Thus we guarantee that effects of completed requests will persist into all later views.

For example, consider a tuple space with five replicas r_1, r_2, r_3, r_4, r_5. Assume that we have an initial view $\{r_1, r_2, r_3, r_4, r_5\}$ with viewid $v_1 = <1, r_1>$. Now suppose a communication failure makes it impossible for replica r_1 to talk to the others, and assume this failure is noticed by r_5, which initiates a view change. As a result of the view change, a new view $\{r_2, r_3, r_4, r_5\}$, is formed with viewid $v_2 = <2, r_5>$. The state of the new view can be taken from any of r_2, ..., r_5, since all of them were in the old view and any complete request has happened at them all. When r_1 notices the failure, it also will initiate a view change. However, since it is unable to talk to enough other replicas, it will remain in the old view.

View changes take place as follows: Replicas send periodic "I'm Alive" messages one another. If a replica notices a change in the communication pattern, or if it has just recovered from a crash, it initiates a view change. It is the *manager* of this protocol; the other replicas are the *underlings*. There are two viewids used in the protocol: the *current viewid*, which is the viewid of the view to which the replica currently belongs, and the *proposed viewid*. When a replica is not involved in a view change, its current viewid equals its proposed viewid; during a view change the proposed viewid is greater than the current viewid.

The manager carries out the following steps:

1. It generates a new, proposed viewid and sends it in an invitation message to the other replicas. A viewid contains a sequence number and a replica id. The manager generates the new proposed viewid by incrementing the sequence number in its proposed viewid and concatenating it with its replica id.
2. It waits for acceptances from enough replicas so that a majority have accepted the new view. If not enough respond, or if some refuse, it tries again with a higher viewid. If it receives an invitation containing a higher viewid, it becomes an underling as described below; if the viewid in the invitation is too small, it sends a refusal.
3. Acceptance messages contain the current viewid of the replica that sent them and also the replica's state (its tuple space and table). The manager selects the replica whose acceptance message contained the highest viewid, initializes its state to that in the acceptance message, and sends commit messages containing the new viewid, view, and new state to the other replicas.

A replica becomes an underling if it receives an invitation with a larger viewid than its current viewid. (If the message contains a smaller viewid, it is old and is discarded.) Underlings do the following:

1. Send an acceptance message containing the replica's state and current viewid and record the new proposed viewid.
2. Wait for a commit message containing the largest proposed viewid seen by this replica. If one arrives, initialize the replica state with the information in the message. If none arrives after a sufficient time has passed, become a manager. If an invitiation with a higher viewid arrives, return to step 1; if the viewid in the invitation is too small, send a refusal.

Our acceptance messages are large since they include the entire tuple space and table. This cost can be reduced by having acceptance messages contain just the replying replica's current viewid. In the second phase, the manager informs the replica with the highest current viewid to distribute its local copy of the tuple space and table to all replicas in the new view. If the manager itself has the most recent viewid, this is a one-and-a-half-phase scheme, just like the algorithm above; otherwise, it is a two-phase scheme.

Our view change protocol is robust in the face of difficulties such as lost messages and further events such as crashes or recoveries that occur while a view change is taking place. It also works properly when several replicas become managers simultaneously. We will not argue here why these properties hold; they are discussed in [6, 7, 18, 21].

5.1. Correctness

We claim that (1) the effect of a tuple space operation either survives into the new view (if it is completed at all replicas in the old view) or the operation will be retried in the new view (if it is not completed) and (2) requests that cause modifications are executed at most once, even across the view changes.

The intuition behind the first claim is that every view has at least a majority of replicas. Thus it contains at least one replica that knows about the effects of all operations that completed in earlier views. That replica is used to update the state of the replicas in the new view. Workers retry operations until all replicas in the current view acknowledge them. Retry is needed because if a view change takes place before an operation completes at all replicas in the old view, the new view may not contain its effects.

Repeated attempts to complete operations do not cause modification operations to be executed more than once. Duplicate requests for the same operations are filtered out using the timestamp-mid table. Furthermore the table is accurate since it is taken from the same replica whose tuple space was used to initialized the state of the new view.

5.2. Additional Linda Operations

As mentioned in Section 2, additional operations have been proposed for Linda [3]. A **rdp** does not wait for a tuple when none matches; instead it signals an exception. Similarly, an **inp** does not wait when there is no match, but instead signals an exception.

These operations could be added to our implementation without difficulty in we ignore view changes. For example, we could implement **rdp** by reading from all replicas; the signal would occur only if none has a match. The following example illustrates the difficulty:

W_1	W_2
out("x", 3)	**rd**("x", **formal** u) % u = 3
	rdp("x", **formal** v) % signals an exception

Suppose initially there is no tuple ("x", *), where * is an integer. When worker W_2 reads 3 into variable *u*, this implies that the **out** has happened. Now suppose there is a view change, and the effects of the **out** are not part of the initial state of the new view. Then the **rdp** of W_2 occurs and observes that the **out** has not yet occurred. Note that this problem will not occur if W_2's second operation is a **rd**, since the **rd** will simply wait until the effect of the **out** can be observed.

Our implementation could support these operations by having **rd** (and **rdp**) read all replicas in the current view, and return a tuple only if it is in the intersection of the tuples returned by the replicas; if the intersection is empty **rd** would try again and **rdp** would signal. However, the result of this change is a slower implementation than the one proposed.

6. Related Work

In this section we discuss other distributed implementations of Linda, and two general replication methods that work in the presence of partitions.

6.1. Distributed Implementations of Linda

The S/Net kernel [4] is similar to our scheme, except that copies of tuples are stored at every node in the network, rather than at a small set of replicas. Thus, the scheme uses much more memory than we do. Also, it does not tolerate failures of the network or the nodes. For instance, if an **out** request does not reach all nodes, the tuple space becomes inconsistent; an **in** can never succeed if one copy of the tuple space becomes inaccessible.

Both schemes **rd** from one copy and **out** to all copies. The S/Net kernel executes **in**s in one phase; our protocol executes **in**s in two phases, but the second phase is done in the background. Thus the two schemes perform[3] similarly, assuming that the S/Net executes **out** operations in the background; this is not described in [4].

The VAX-LAN kernel [1] stores each tuple at only one node; **out** modifies its node's local copy, while **rd** and **in** search for a match at all nodes. Since only one copy of each tuple is stored system-wide, the scheme does not provide high availability. If the node owning tuple *t* crashes, or a message containing *t* in response to an **in**(*s*) is lost, then *t* is lost. A network partition may make some tuples unavailable temporarily.

The two schemes perform similarly for **rd** and **out** operations. The VAX-LAN scheme performs better on **in** operations since the worker can continue as soon as the first response arrives. But the better performance on **in** operations comes from the fact that there is just one copy of each tuple, and this is the reason that the scheme cannot be made highly available.

6.2. Replication Techniques

Weighted voting [11] provides a general replication method by dividing a certain number of *votes*, *n*, among replicas. A read operation has to acquire a *read quorum* of *r* votes before it can complete and a write operation has to acquire a *write quorum* of w votes. The requirement that $r + w > n$ and $w > n/2$ ensures that every read quorum intersects every write quorum and that write quorums intersect, which in turn implies that there is at least one up-to-date copy in both read and write quorums.

Our protocol is a special case of the voting scheme where the read quorum is one and the write quorum is all the replicas. In any such scheme, write operations cannot be performed if a replica is down or inaccessible. In each case, this problem is overcome by using a view change protocol such as the one described in Section 5.

Herlihy [12] has extended voting to take advantage of operation semantics, and thus made the algorithm more efficient. Like Herlihy's scheme, our method utilizes the Linda operation semantics to achieve better performance: **Out** operations and the second phase of **in** operations are performed in the background, which makes **out**'s appear to be zero-phase, and **in**'s to be one-phase. This outperforms voting where all modification operations need to be two-phase.

Another way of keeping replicas consistent is to use an atomic broadcast protocol [5, 2, 20]. These protocols ensure that all operations are performed at all replicas in the same order, and in addition that operations of workers are performed in the order issued. These guarantees are sufficient to provide linearizability but are stronger than necessary since the semantics of the operations are not considered. Atomic broadcast protocols introduce more delay than our method, but fewer messages may be exchanged.

7. Extensions

There are two ways that our method can be extended. First, we can partition the tuple space among different sets of replicas. Keeping the entire tuple space at one set of replicas is not desirable if the space is large because each node where a replica resides must provide a large amount of storage. Also, if accesses to the tuple space are frequent, the replicas' nodes may become overloaded and slow down workers more than is acceptable. These problems can be overcome by partitioning the tuple space among different sets of replicas. The obvious way to distribute the tuples is by logical name. For example, all tuples with logical name "x" will be in set S and all those with logical name "y" will be in set T. Each partition has its own set of replicas, and replicas belonging to different partitions probably reside on different nodes.

When a worker performs an operation, it sends the request to the

[3]Our analysis of performance is based on the number of messages in the protocol and the associated delays.

replica set that contains information about that tuple or template. Determining what set to use could be done statically or dynamically. An example of a static mechanism is a hash function that maps logical names into sets. An example of a dynamic mechanism is a (replicated, highly-available) *location server* that stores the mapping; workers would maintain a cache containing the mapping for recently used tuples and consult the server only when there is a cache miss or when the information in the cache is found to be out of date. Implementations of location servers are discussed in [9, 14, 17].

Constraints on operation execution are similar to those given in Section 4. The background process BP can interact with different partitions in parallel. Operations of a single worker that occur in the same partition must be done in order. In addition, prior **In2**'s must complete before an **out** can be executed, even when the **In2**'s use different partitions than the **out**. View changes occur independently at each partition using the protocol described in Section 5.

The second extension allows Linda programs to survive failures of workers. Our method as described so far does not perform correctly if a worker fails. For example, if a worker crashes after starting an **In1** but before completing the corresponding **In2**, some tuples may be locked forever. It is important to tolerate failures of workers for computations in which there are more workers than replicas, since the probability of a worker failure is relatively high in this case.

To tolerate workers' failures, we need to release locks held by crashed workers and also undo or complete their incomplete **out** and **In2** requests. However, as mentioned earlier, it is not possible in general to distinguish a node crash from a partition. Releasing the worker's locks in the case of a partition would be a problem, because the worker is still running and therefore depends on its locks.

We can solve this problem by forcing a worker that cannot communicate because of a partition to crash. The idea is for replicas to maintain two views (and viewids): the *replica-view* discussed earlier, and also a *worker-view*. Initially all workers are in the worker-view. Replicas exchange "I'm alive" messages with workers. If a worker does not respond after a sufficient number of tries, the replicas carry out a *worker view change*, during which replicas agree on a new worker-view and worker-viewid in addition to a new replica-view and viewid. As part of the view change, an initial state is selected for the new view as usual, except that all locks held by the excluded worker are released in this new state. As usual a majority of replicas must participate in the view change.

Whenever a replica receives an operation request from a worker, it checks to be sure the worker is in the current worker-view. If not, the request is rejected, and the worker is sent a "you must crash" message. When a worker receives such a message it stops processing.

This semantics supports fault-tolerant Linda programs. Workers in such a program monitor the progress of other workers; if a worker does not finish a task quickly enough, some other worker will do the task. (An example of such a program is given in [21].) This method may lead to a task being (partially) performed several times. If this is an error, tasks need to be run as atomic transactions [8] so that earlier attempts to do a task can be aborted. Adding atomic transactions to Linda requires further research.

8. Conclusions

This paper has described a technique for constructing a highly-available tuple space that works in a general communications network. The method involves little delay of workers: a **rd** waits for only one response, an **out** does not delay the worker at all, and an **In** delays the worker only during the first phase. The method can be extended to a system in which the tuple space is partitioned with each partition stored at its own set of replicas; this extension can prevent the tuple space from becoming a bottleneck. We also described a way of tolerating failures of workers.

In addition to showing how a fault-tolerant implementation of Linda can be provided, our work indicates how fault tolerance might be achieved for other parallel systems. Many parallel computations are long lived; fault tolerance is particularly important for them. The other advantages of distribution (using inexpensive machines over a network and scalability) also apply to any parallel system.

Finally, our technique extends work on replication by showing what can be done when the semantics of operations are taken into account. We were able to devise an implementation that had lower delay than both the general voting technique and atomic broadcast. However, our scheme works only because of the semantics of **In**, **out**, and **rd**; the addition of **rdp** and **Inp** changed the semantics sufficiently that our special optimizations can no longer be used.

References

1. Ahuja, S., Carriero, N. and Gelernter, D. "Linda and Friends". *IEEE Computer 19*, 8 (August 1986), 26-34.

2. Berman, K. P., and Joseph, T. A. "Reliable Communication in the Presence of Failures". *ACM Trans. on Computer Systems 5*, 1 (February 1987), 47-76.

3. Carriero, N. Implementation of Tuple Space Machines. YALEU/DCS/TR 567, Dept. of Computer Science, Yale University, December, 1987.

4. Carriero, N., and Gelernter, D. "The S/Net's Linda Kernel". *ACM Trans. on Computer Systems 4*, 2 (May 1986), 111-129.

5. Chang, J., and Maxemchuk, N. "Reliable Broadcast Protocols". *ACM Trans. on Computer Systems 2*, 3 (August 1984), 251-273.

6. El Abbadi, A., Skeen, D., and Cristian, F. An Efficient, Fault-Tolerant Protocal for Replicated Data Management. Proc. of the 4th ACM SIGACT/SIGMOD Conference on Principles of Database Systems, ACM, 1985.

7. El Abbadi, A., and Toueg, S. Maintaining Availability in Partitioned Replicated Databases. Proc. of the 5th ACM SIGACT/SIGMOD Conference on Principles of Data Base Systems, ACM, March, 1986.

8. Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. "The Notion of Consistency and Predicate Locks in a Database System". *Comm. of the ACM 19*, 11 (November 1976), 624-633.

9. Fowler, R. J. Decentralized Object Finding Using Forwarding Addresses. Technical Report 85-12-1, Dept. of Computer Science, University of Washington, Seattle, WA, December, 1985.

10. Gelernter, D., Carriero, N., Chandran, S., and Chang, S. Parallel Programming in Linda. Proc. of International Conference on Parallel Processing, IEEE, 1985.

11. Gifford, D. K. Weighted Voting for Replicated Data. Proc. of the Seventh Symposium on Operating Systems Principles, ACM, December, 1979.

12. Herlihy, M. P. Replication Methods for Abstract Data Types. Technical Report MIT/LCS/TR-319, MIT Laboratory For Computer Science, Cambridge, MA, May, 1984.

13. Herlihy, M. P., and Wing, J. M. Axioms for concurrent objects. 14th ACM Symposium on Principles of Programming Languages (POPL), January, 1987, pp. 13-26. Also Carnegie Mellon University Report CMU-CS-86-154.

14. Hwang, D. J. Constructing a Highly-Available Location Service for a Distributed Environment. Technical Report MIT/LCS/TR-410, MIT Laboratory for Computer Science, Cambridge, MA, November, 1987. Master's thesis.

15. Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System". *Comm. of the ACM 21*, 7 (July 1978), 558-565.

16. Lamport, L., Shostak, R., and Pease, M. "The Byzantine Generals Problem". *ACM Trans. on Programming Languages and Systems 4*, 3 (July 1982), 382-401.

17. Mullender, S., and Vitanyi, P. "Distributed Match-Making for Processes in Computer Networks -- Preliminary Version". Proc. of the Fourth ACM Symposium on the Principles of Distributed Computing, ACM, August, 1985. Held at Minaki, Ontario, Canada.

18. Oki, B. M., and Liskov, B. Viewstamped Replication: A general Primary Copy Method to Support Highly-Available Distributed Systems. Proc. of the 7th ACM Symposium on Principles of Distributed Computing, ACM, Aug., 1988.

19. Fred B. Schneider. Fail-Stop Processors. Digest of Papers from Spring CompCon '83 26th IEEE Computer Society International Conference, IEEE, March, 1983, pp. 66-70.

20. Schneider, F. B. The State Machine Approach: A Tutorial. Technical Report TR 86-600, Cornell University, Dept. of Computer Science, Ithaca, N. Y., December, 1986.

21. Xu, A. A Fault-Tolerant Network Kernel for Linda. Technical Report MIT/LCS/TR-424, MIT Laboratory for Computer Science, Cambridge, MA, August, 1988. Master's Thesis.