# Recovery With Limited Replay: Fault-Tolerant Processes In Linda

*Srikanth Kambhatla and Jonathan Walpole*

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 90-019

September, 1990

# RECOVERY WITH LIMITED REPLAY: FAULT-TOLERANT PROCESSES IN LINDA

**Srikanth Kambhatla and Jonathan Walpole**

Department of Computer Science and Engineering,
Oregon Graduate Institute of Science and Technology,

ABSTRACT

Research in the area of fault-tolerant distributed systems has focused to a large extent on data surviving various forms of failure. The replica control algorithms for maintaining mutually consistent replicas abound in number. However, comparatively little work has been devoted to making processes recoverable. In domains other than databases and transaction processing, fault-tolerance generally implies both fault-tolerant data and fault-tolerant processes. In environments where cooperation among processes is important we argue that high availability of processes in addition to their recoverability is crucial.

Our specific interest is in the Linda tuple space paradigm. In this paper we discuss efficient techniques for making Linda processes recoverable and outline some characteristics of Linda that make it particularly suitable for implementing fault-tolerance. We also propose a simple extension to our recoverable process mechanism that makes processes highly available.

[**keywords**: fault-tolerant processes, high availability, recovery, Linda tuple space, replay, message logging].

## 1. Introduction

Local area network based distributed systems are low in cost, widely available and generally have a considerable amount of unused processing capacity. These characteristics make them potentially attractive as economical vehicles for supporting long-lived computation intensive applications such as scientific computations and simulations.

Such computations are typically structured as a set of processes that run in parallel and cooperate to achieve a common goal. In this paper, we address the issues associated with supporting long-lived parallel applications using the Linda programming paradigm [1,2,3]. Linda presents a simple and an elegant model of parallel computation in which Linda programmers view a distributed system as a virtual uniprocessor containing a bag of tuples. When a process wishes to communicate with another process it leaves a tuple in the tuple space. Likewise, a process wishing to receive a message looks for it in the tuple space. This indirection makes Linda processes location independent.

The long duration of computations in the application domain we have outlined makes *reliability* an important concern. Failures can affect such applications through a loss of data, a loss of computation, or both. Different approaches can be taken to

cushion against these failures:

- Data can be made *recoverable*. This ensures that the data is left in a consistent state following failures.
- Data can be made *highly available*. This ensures that the data remains accessible in the presence of failures.
- Processes can be made *recoverable*. This ensures that the computation need not be restarted from the beginning when a failed node recovers.
- Processes can be made *highly available*. This ensures that the entire computation can make progress in the presence of failures.

In an application domain where data is persistent, it is often useful to make data highly available even when processes are not. The reason for this is that such applications usually involve transactions that run in isolation. Such transactions are independent of all other transactions, and their results are solely dependent on the state of the database when the transaction starts. However, in a cooperative parallel processing environment, like that of scientific computation, maintaining highly available data (i.e. the tuples in Linda context) is not enough. In such environments, the completion of the computation requires the successful completion of all the processes involved in the computation. Furthermore, it is usually the case that no one process can remain in a failed state while the others continue. Therefore, in such applications high availability can only be attained by supporting both highly available data and highly available processes.

Various solutions to the problem of highly available processes have been suggested in the literature. These include backup processes [4], single recorder process [5], troupes [6], and Available Processes [7]. These schemes provide highly available processes at varying costs. The backup process scheme, troupes and available processes all involve one or more redundant process, per active process in the system. Such approaches gain availability at the cost of considerable processing capacity that could otherwise be used by the application. In the recorder process scheme, the recorder process itself is a critical point.

In this paper, we present an efficient technique for making Linda processes highly available. Our approach does not require additional language constructs, and does not impose additional synchronization constraints on processes. Furthermore, we do not require any special hardware to record messages, nor is the recording of information restricted to a single centralized node.

The main technical contributions of this paper are the following:

- It suggests that the requirements of a cooperative distributed computing environment necessitate both highly available processes and highly available data.
- It draws attention to Linda as a particularly suitable model for fault-tolerant applications and gives an efficient design for highly available processes in Linda. This is achieved by integrating the techniques of recoverable processes and highly available data.

The paper is organized as follows. In section 2, we present our failure model. In section 3, we discuss the concurrency and recovery model of our system, and the

properties of Linda that make it suitable for fault-tolerant applications. In section 4, we present our design for highly available processes in Linda. Section 5 analyzes the performance of the proposed mechanism, and section 6 discusses related work. Finally, section 7 concludes the paper.

## 2. The Failure Model

Failures can be classified into several broad categories, based on the behavior of processors [8].

- *Crash Failures*: the failed nodes simply stop on failing and all processes on the node die.
- *Omission Failures*: nodes occasionally fail to send or receive messages that they should.
- *Byzantine Failures*: processors malfunction by sending spurious messages, and possibly, even contradictory messages.

In this paper, we assume the crash failure model. This is a reasonable assumption because the abstraction of a crash failure model can be built on top of a system with more complex modes of failure, and algorithms developed on this model can be extended to systems with other failure models [9,10]. We make further assumptions that the individual processes that make up the parallel computation are deterministic, and that appropriate communication protocols are used to ensure reliable and ordered message delivery. When a node fails we assume that the tuple space on the node, the unprocessed messages which are residing in the receiver buffer, and the messages which are in transit to the node are all lost.

## 3. The computational model

Before presenting our solution to fault-tolerant and highly available processes in Linda, it is necessary to outline a number of characteristics of our computational model. These characteristics are determined both by the nature of the application domain and by the use of the Linda parallel programming paradigm.

### 3.1. Characteristics of the application domain

The concurrency model is one of cooperating parallel processes rather than isolated independent processes as in database applications, for example. In the latter case, the *non*interference among processes makes *serializability* [11] an appropriate notion of consistency. The cooperative nature of the processes in our computational domain suggests that serializability is no longer an appropriate consistency condition. In message passing systems, approaches to consistency have tended to vary from models where processors run in total synchrony, to systems which provide little more than the basic message passing mechanism thereby rendering only probabilistic behavioral statements possible [12]. Among the different models, the *virtual synchrony* model of Birman and Joseph is the most appropriate for our applications.

The recovery model is also distinct from that of a typical distributed database application which is generally based on the concept of *atomic actions*. The motivation in applications where data is persistent is that the intermediate state information of one transaction should not be visible to other transactions in the event of a failure.

This is achieved by ensuring that transactions execute once (to completion) or not at all (i.e. they abort). In our application it is inappropriate for parts of a parallel computation to abort. Instead, we want to ensure the eventual completion of each part of the computation. Therefore, we must ensure more than atomicity. Furthermore, the long life of processes makes it inappropriate to roll back the entire computation in order to restart following a failure. Consequently, we are interested in ensuring the continued execution of the computation even in the presence of failures. This can be achieved using techniques such as checkpointing.

### 3.2. Characteristics of Linda

Linda has several useful features, which make it a good platform for developing fault-tolerant applications in. Firstly, processes in Linda are *decoupled in time*. Both the **in()** and the **rd()** operations on the tuple space block the calling process until they find a tuple that satisfies them. The combination of the blocking operations and the level of indirection and buffering introduced by performing operations on the tuple space rather than directly on the cooperating processes means that Linda programmers do not need to assume anything about the relative execution speeds of cooperating processes.

The above characteristic is particularly useful for fault-tolerance because it suggests that individual processes in a parallel computation may be able to recover from failures independently. Any other process that depends on the results of a recovering process will eventually block on an **in()** or a **rd()** until the failed process recovers.

Secondly, the processes in Linda are *decoupled in space*. Since all communication takes place via the tuple space, processes need make no assumption about the location of the processes with which they are communicating. This is a useful characteristic for reliable distributed systems because it simplifies tasks such as restarting failed processes on other nodes and replicating processes.

Another important feature is the nondeterminism that is built into the receiver operations of Linda. When a process does a receive operation, it will arbitrarily get one of possibly several matching tuples. Since a given process may only be generating some of possibly many matching tuples, the runtime system may be in a position to sustain the computation without blocking in the presence of some transient failures.

Nondeterminism together with the time and space decoupling features imply some interesting properties for fault-tolerance. These are illustrated in the following section. Discussion of our scheme requires the following definitions [13].

The *state* of a process is the state of all the variables of the process along with its program counter, and its operating system state. The *global state* of a distributed computation comprises of the state of all its processes, the states of the receive buffers, and the states of the channels carrying the messages. The *initial state* of a distributed system is the state in which each process is in its initial state and each of its receive buffers is empty, and there are no messages in transit.

An *execution* of a distributed system is a sequence of the form

$$S_0 \xrightarrow{\alpha 1} S_1 \xrightarrow{\alpha 2} S_2 \xrightarrow{\alpha 3} \ldots,$$

where, each $S_i$ is a global system state, and each $\alpha i$ is an action. Each action is the execution of a primitive statement by a process which atomically transforms the global

state of the system. An execution is called ***failure-free*** if no process failure occurs during the system execution. A global system state S is a ***valid state*** if S occurs in some failure-free execution.

## 4. Our design for highly available processes

Due to the interdependencies among the processes in the computation, failure of one node generally affects the entire computation. In order to avoid having to restart the computation from the beginning, we base our recovery technique on periodic checkpointing of process states. The checkpointing operation is performed unilaterally by each process. Thus on a failure, a recent checkpointed state is available from which the failed process can restart.

An additional constraint during recovery is that if two processes exchange a message at some global state, they should ***logically*** exchange the message again when recovery is being attempted from an earlier state. We achieve this consistency in message interaction by means of message ***logging.*** The interactions of each process are recorded in the message logs before the interaction actually takes place.

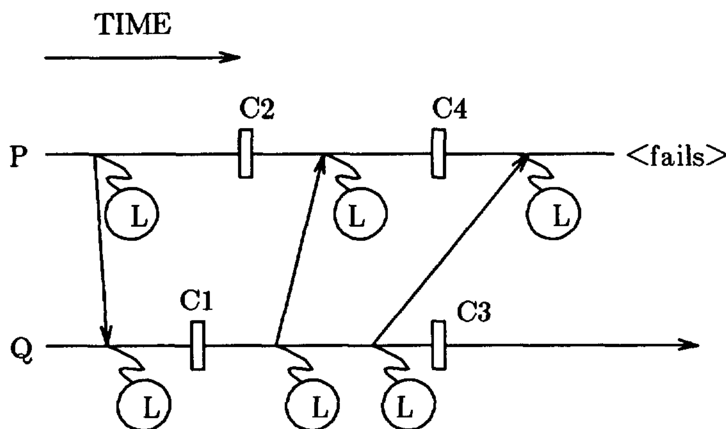


**Figure 1.** Checkpointing with message logging.

In figure 1 for example, we have two communicating processes P and Q. The C's represent checkpointing operations, while L's represent the message logging operations. The processes record the messages in the message log before they are sent out. Similarly, when a process receives a message, a copy of that message is inserted in the logs. Thus the contents of the messages exchanged before failure are available for reuse during recovery.

Since this recording is being done on a per-process basis, the recovering processes need not depend on information recorded by any other process to complete its recovery. Thus the recovery operations of each process are independent of the other processes. Similarly, message logging gives the processes considerable freedom in unilaterally deciding to take checkpoints. This is because different checkpoints need not satisfy any consistency criteria with respect to any messages exchanged.

The message log is kept by means of a *log space.* A log space is functionally similar to the tuple space because both are repositories of tuples. However, the log space stores the information necessary for recovery purposes. This information includes the checkpoints of the processes and the logs of the tuple exchange and certain other information which we describe a little later. The log space and the tuple space need to be kept separate in order to prevent the log tuples from matching any template in the normal operation and conversely, to prevent the tuples in the tuple space from matching any templates during recovery.

### 4.1. Overview of the recovery process

An important difference between the tuple space and the log space is based on the ordering of tuples. While a tuple space is an unordered collection of tuples, the tuples within the log space have a total order based on the process*Id* and the time of insertion. The process*Id* needs to be tagged in order to identify the process to which the log belongs. The tuples of each process are ordered based on the sequence in which they were inserted into the log space. The reason for the latter ordering is given below.

The nondeterminism feature of Linda implies that the **in()** and **out()** operations can interact with one of possibly many processes. When a process does an **out()**, it reveals its internal state to the other processes, which might take some action based on it. An **in()** affects the processes in another way. It removes some state from the tuple space, that might otherwise have caused some process to take different actions had they removed the tuple instead, thereby resulting in a different global state.

Therefore, if the result of the computation is to be the same regardless of failures, it has to be ensured that the order in which tuples were matched before the failure is maintained during recovery. This is stricter condition than is required in send receive type of communication. Intuitively, the type of communication via the tuple space is different because the **in()** operation can be used to communicate information to other processes [13].

During the normal operations on the tuple space, a copy of the tuple is sent by the process to the log space. Figure 2 illustrates the process.

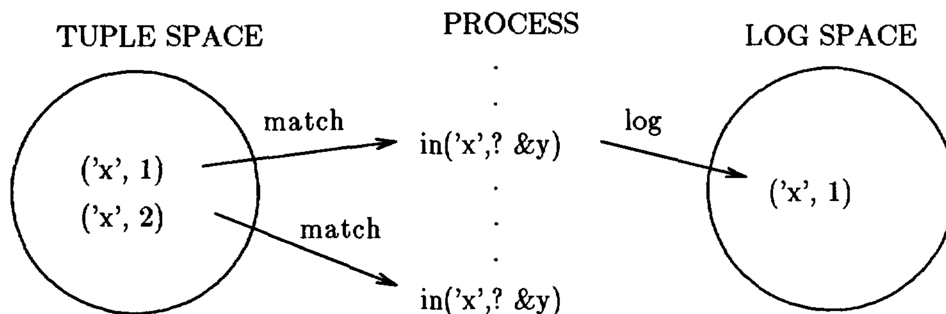


**Figure 2.** Use of a separate log space.

During recovery, the process *replays* the execution from its checkpointed state. The recovery scheme essentially consists of the following steps:

1. Load the checkpointed state.
2. For each send operation the recovering process verifies from the log space whether the message had been sent before. If it had, the current send logically becomes a null operation. If it had not, the message is sent and logged.
3. Similarly, if ensuing operation is a receive, the process verifies from the log space whether the message had been received before. If it had, the current receive is satisfied by the tuple present in the log space. The tuple from the log space is not removed, but it is ensured that the tuple match follows the same order in which matching was done prior to a failure. If a matching tuple does not exist in the log space, the operation is performed on the tuple space.

### 4.2. Consistency of the log space and the tuple space

We need to ensure that the states of the log space, the tuple space and the processes are consistent with each other at all times. For example, suppose that a process P has done an **in()** on a tuple t. The operations involved are:

- t should be atomically removed from the tuple space.
- t should be inserted into the log space.
- the process should then continue its operation with t.

The algorithm should ensure that consistency is maintained, in the presence of failures. When a successful match occurs, it has to be ensured that the same tuple does not match any other template. We achieve correctness by means of *marking* tuples. On a successful match, the tuple is marked as *matched* to indicate a logical removal from the tuple space. A copy of the *matched* tuple is now put into the log space. After the copying operation onto the log space is successful, the tuple in the log space is marked as *copied*; and the tuple is removed from the tuple space. The tuple in the log space now becomes a *log* tuple.

Any *matched* tuples left in the tuple space by a failed process are indicative of the fact that the **in()** operation was not completed successfully. If the copying operation was not successful as well, the tuple should be available for reuse. These are precisely the tuples which have been *matched* by the process in the tuple space, but which do not exist in the log space. When the recovery manager detects the failure of a process, it unmarks all these tuples and makes them available for use. On the other hand, if the recovery manager finds any *matched* tuples in the tuple space, a copy of which exists as a *copied* tuple in the log space, those tuples are removed from the tuple space. It is ensured that the process*Ids* are tagged onto the tuples to prevent inconsistencies. The transitions of the state of the tuple can be represented as shown in figure 3. A summary of the recovery actions based on the tuple state is given in table 1. Similar actions are performed for **out()** and **rd()**.

It is important that the checkpointing operation be done without loosing previous information. This is ensured by deletion of the previous checkpoint only after the current checkpoint has been taken. When the new checkpoint is written the log tuples of the process can safely be removed. Since, the log tuples need only be kept in the log

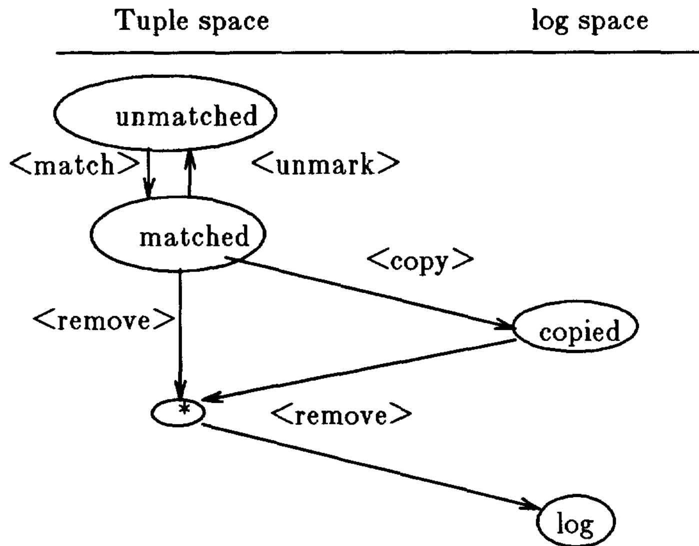


**Figure 3.** Life and times of a tuple.

| in tuple space | in log space | recovery manager | recovering process |
|---|---|---|---|
| unmatched | - | - | - |
| matched | - | unmark | - |
| matched | copied | remove | use from log space |
| - | copied | - | use from log space |
| - | log | - | use from log space |

**Table 1.** Action performed by on failure by the recovery manager.

space between checkpoints for each process, and since the number of processes is finite, the log space always contains a finite number of tuples, if the checkpointing operation is done periodically. The algorithms for checkpointing and Linda operations are outlined in figures 4 and 5.

### 4.3. Replication for availability

Message logging and checkpointing ensures the fault-tolerance of processes. In order to achieve high availability in the tuple spaces, we need to replicate the tuple space. Some of the replication issues have been addressed in an earlier paper [14]. In order to make the processes highly available as well as fault-tolerant we replicate the recovery information associated with each process. In our scheme, we treat the tuple space and the log space analogously and apply the tuple space replication algorithms to provide the same degree of availability to for the log space as for the tuple space itself.

```
procedure checkpoint(P)
begin
      increment checkpoint number for P in log space
      put the state of P onto log space
      delete the all messages logged for P
      delete the previous checkpointed state
end
```

**Figure 4.** Checkpointing operation

```
/* Linda operations at run time */
procedure LindaOperations(operation)
begin
      case operation of
      in():
      rd():
            match a tuple locally
            mark the tuple as matched
            copy the tuple onto the log space and mark LS tuple as copied
            if (operation = in()) remove tuple from the tuple space
            mark LS tuple as log
            tag the tuple in the log space
      out():
            put the tuple in the log space and mark it as created
            copy it onto the tuple space and mark it as unmatched
            mark the LS tuple as log
end
```

**Figure 5.** Linda operations at run time.

### 4.4. Failure Detection and the recovery manager

The recovery of the processes is managed by a distinguished process called the *recovery manager*. When the recovery manager detects a failed process, it waits until the node recovers and spawns a new process with the checkpointed state. If the failed node does not recover in some time T, the recovery manager spawns a new process on a different node and loads the replicated checkpointed state on that process.

Failure detection is tricky because the processes seldom interact with each other directly. Our solution is based on the recognition of the fact that the common meeting point for all the processes is the tuple space itself. So, each process periodically puts an *I am alive tuple* into the tuple space. The recovery manager regularly reads the tuple space for these tuples. Failure to get a tuple from a process would indicate the necessity of a corrective action. The *I am alive tuples* of different processes need to be distinguishable from each other. They need not be distinguishable among themselves because at any given time only one would be residing in the tuple space.

The failure of the recovery manager itself can be detected by making the processes check if their previous *I am alive* tuples are still present in the tuple space, before they

insert the new ones. If the tuple space has some unconsumed tuples, another recovery manager needs to be invoked. It is conceivable that the recovery manager fails in the middle of a repair operation. To take care of this eventuality, the recovery manager needs to log the fact that it is starting the recovery process at some node. This log is not logically a part of the tuple space, thus it is put into the log space. The interaction of the recovery manager with the tuple space and the log space can be shown as in figure 6. The recovery Manager algorithm is outlined in figure 7. We are exploring the possibilities of making the recovery manager more efficient by decentralizing it and are also looking into ways of piggybacking the *I am alive* tuples for minimum overhead on the processes.

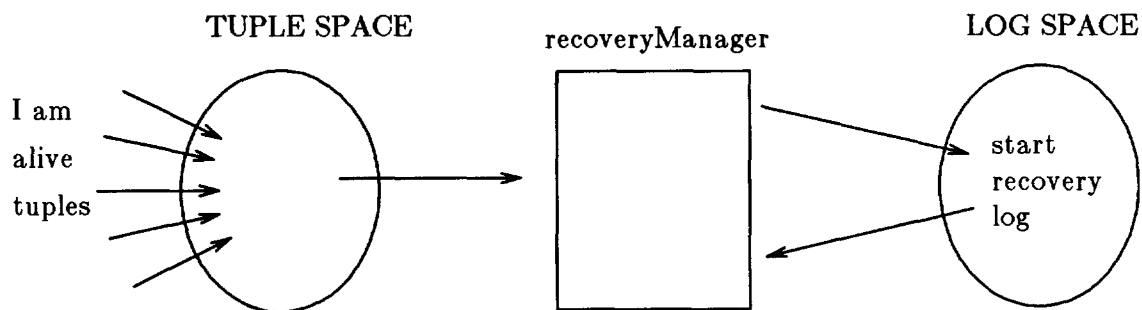


**Figure 6.** Interactions of the recovery manager

```
/* invoked on a process failure detection */
procedure recovery_manager()
begin
    periodically do
        look for I am alive tuples for all processes
        remove them from the tuple space
    if (failure detected) then
        write a log stating failed process and start attempt
        wait till node comes back up
        if (timeout) spawn a new process on a different node
            else spawn a process on the failed node
        retrieve the checkpointed state for the failed process
        remove inconsistencies in tuple space and log space for the process
        start executing the process
        remove start recovery attempt log
end
```

**Figure 7.** Recovery manager process

### 4.5. Some attractive features of our scheme

Independent checkpointing and message logging are central to our scheme. An alternative might be not to log messages at all, but keep the checkpoints *consistent* [15,16,17,18]. In these schemes, several processes need to rollback to their earlier consistent state so that replay will correctly handle message interactions. Some of the advantages of our scheme are:

- The checkpointing operation can be performed unilaterally by each process. In particular, the checkpoints of different processes need not be consistent in order to ensure recovery.
- Replay is limited to just the failed process. The alternative mentioned above might result in several functional processes to rollback and re-execute from their checkpointed state.
- There is no roll back involved. The processes which fail, start again from their previous checkpointed state. This is the only way in which a process goes back to a previous state in time. As a consequence of the above, we do not have the problem of undoing interactions or side effects.
- The proposed scheme implements recovery efficiently. Since interprocess communication does not take place during replay, all interactions are logically performed only once. Thus, we reduce communication overheads. Also, recovery is faster because we are no longer bothered about determining all the processes we had interacted with in order to rollback to consistent checkpointed states.

The benefits listed above are gained at the expense of the space that is required for storing the message logs. However, we contend that the space overhead is reasonable. The additional space required is directly proportional to the number of messages that need to be logged before a checkpoint happens. In the trivial case, we can checkpoint after every message received, thereby ensuring no space overhead at all, while in a more realistic case, we might want to take checkpoints based on the event of the space allocated for logs getting full.

### 4.6. Fault-tolerance conditions and correctness of our scheme

Any implementation supporting fault-tolerant processes must ensure the following fault-tolerance conditions.

FT1. *Correctness condition.* The result of a computation should be the *same* as the result of some failure-free execution. Here the meaning of *same* depends on the semantics of the domain in which the computation is taking place.

FT2. *Termination condition.* The computation will reach a final state in finite amount of time if at least one node is operational.

FT2 ensures that any failure detection and recovery algorithms should all terminate and have the effects which ensure FT1.

**Correctness Argument**

Here we give an informal argument to say that our scheme satisfies the fault-tolerance conditions FT1 and FT2.

First, we assume that a failure-free execution satisfies the condition FT2. What remains to be shown is that the recovery scheme will put the failed process in the *same* state as it was before the failure. We also need to show that this happens in a finite amount of time given the fact that at least one (any) node is functional.

For a failed process to recover, the *recovery manager*, loads the checkpointed state into a new process. A deterministic process would, given the checkpointed state, go through the same sequence of states that it went through before the failure. When it reaches the point where it had interacted with some other process, it looks into its message log. Since the tuples match templates in the order in which they entered the log space, the recovering process is fed with the same tuples, which it had seen before the failure and in the same order.

For each tuple operation, we follow the approach outlined in the recovery scheme. It is important to realize that any tuple which has been sent or received before the failure is present in the logs. The converse is also true. Any tuple which is not there in the log has not been received or sent. Once such a point is reached, we know the process is back to where it was before the failure. Therefore, the scheme puts the recovering process back in the *same* state as it was in before failure. *Same* in the context of Linda is with regard to the nondeterminism built into the tuple matching operation.

For the termination condition to be satisfied, we note that our scheme spawns the process on a separate node if the failed node does not come back up in some time T. The rest of the termination proof follows from the fact that since there can only be finitely many tuples in the log space the amount of time taken for the recovery from the loading of checkpointed state to normal running condition, is finite.

## 5. Estimation of Overheads

One way of estimating overheads in our recovery scheme is to compare the number of logical tuple operations which occur in our scheme with the number in an implementation without fault-tolerance. The number of logical tuple operations can be expressed as:

Number of operations = number of tuple operations as given in the program

+ number of tuples logged onto the log space
+ expected number of additional operations incurred because of failures
+ number of operations for flushing of the log space
+ the number of operations for checkpointing
+ the number of tuples for failure detection
+ the number of operations when the recovery manager fails
+ number of *start recovery* tuples.

Suppose that,

N = number of tuple operations to be performed by the program
p = number of processes
$\mu$ = failure rate
$\lambda$ = recovery rate

$n$ = average number of tuples present in the log space on failure
$f$ = frequency of checkpointing operation
$T$ = duration of the program

The number of tuple operations on a failure is the average number of tuples read from the log ($n$). During recovery, these tuples are not written back onto the log space. So, the overall rate at which the tuples are read from the log space is given by $\mu Tn$. The number of data tuples written onto the log space is, of course, N. The number of checkpoints taken per process is given by $fT$. So, $pfT$ checkpoint tuples are put into the system. Therefore the log space is flushed $pfT$ times. On a flush, all the tuples in the log space are removed. Since the average size of the log space is $n$, the number of tuples read off during the flushing operations is $npfT$.

Assuming that the *I am alive* tuples are put in the tuple space at the same rate as $f$, the number of *I am alive* put in the tuple space are $fTp$. Since a read operation precedes the insertion, (to check for the recovery manager failure), and the fact that the recovery manager reads the same number of tuples to detect failures, the number of tuple operations is $3fTp$. Noting the fact that the recovery manager is just another process, the number of failures of the recovery manager is $\mu T$. No tuples are inserted in the tuple space or the log space due to this. The number of *start recovery* tuples is the same as the number of failures ($\mu pT$).

We have an equality which we have not mentioned so far. Since the average number of tuples in the log space is $n$ and the rate at which the log space is flushed per process $fT$ times,

$$\frac{N}{fT} = n.$$

Therefore, the total number of tuples operations performed when highly available processes are implemented, $N_{hap}$, is

$$N_{hap} = N + \mu\,Tn + N + npfT + 3fTp + fTp + \mu\,T + \mu\,pT.$$

Or,

$$N_{hap} = N\left[2 + p + \frac{\mu}{f}\right] + T\,\left[4fp + \mu + \mu p\right]$$

We see, that the number of tuple operations is linear in N.

## 6. Related work

Considerable work has been reported on reliable computing in distributed systems. We briefly describe the work on achieving reliable processes.

### Recoverable Processes

Recoverable processes are usually based on checkpointing of the process state. The schemes differ in the model of interprocess communication and in the way of maintaining correctness in the global state. The scheme proposed by Bartlett in [19] assumes a synchronous model of communication. Asynchronous message passing has been considered by many researchers [5,20,4,21]. While most of them have some form of message logging, they differ in issues of the identity of the process doing the logging, the amount of redundancy required, and the nature of the recovery process.

A fundamentally different view for achieving fault-tolerance is by distributed consistent checkpointing [15,16,17,18]. This technique removes the requirement for logging. However the failure of one process may cause other processes in the system to roll back. Multiple failures require complicated recovery techniques.

**Highly Available processes**

Borg *et al* [4] propose an approach to making processes highly available that is based on the concept of providing a backup process for each active process. The backup process resides on a different processor and is passive. Whenever a message is sent to a process, the same message is also sent to its backup. Similarly, replies are also sent to the backup process. There are two disadvantages to this mechanism. First, there is a twofold increase in the number of processes and a threefold increase in the communication traffic in the system. Second, the communication mechanism needs to ensure that a process and its backup always agree on the interactions that the process is involved in.

Another approach, suggested by Powell and Presotto [5], is to have only one process in the system which records the messages that are passed between the others. When a failure occurs, the failed process must be restarted and must ask the central process for a record of all the interactions the original process had participated in. There are several problems with this approach. First, the recorder process is critical. Second the recorder process is a bottleneck. Finally, the efficiency of the approach depends to a large extent on the topology of the underlying network.

The approaches mentioned before have to get another process started when a failure occurs. Unfortunately, the delay involved in doing so might not be satisfactory for some real time applications. Work reported in the area of active replicas has been in terms of *troupes* [6] and *Available Processes* [7]. This approach results in an increase in the number of processes by a factor of N and the number of messages by a factor of $N^2$, where N is the degree of replication.

Our approach to highly available processes has been to integrate the approaches for highly available data along with recoverable processes. This approach lends us immunity against the dependence on one distinguished process, while eliminating the redundancy in the number of processes as well.

**Fault-tolerance in Linda**

To our knowledge, only one project has specifically addressed the issue of fault-tolerance in Linda [22,23]. However, this research only addressed the issue of providing a highly available tuple space via replication of the entire tuple space. The virtual partition algorithm [24] is used to maintain the mutual consistency of the replicas. Each replica has an associated view which changes as the communication capability of the replica changes. Accesses within a view are based on the *read-any-write-all* protocol. A fundamental requirement of this scheme is that each view has a majority of the replicas and that during the creation of a new view at least one member of the previous view is included. Hence, the availability of the tuple space depends on the availability of a majority of the replicas.

Our research differs from that of Xu and Liskov in a number of respects. First, we address the full spectrum of reliability issues for loosely-coupled distributed Linda systems, including recoverable and highly available processes as well as recoverable and highly available tuple spaces. Second, our research in the specific area of tuple space replication is distinct from that of Xu and Liskov in the sense that we investigate methods for replicating different granularities of tuple space data and we require a higher level of availability for out tuple space especially in the presence of network partitions and the failure of a majority of the nodes [14].

## 7. Conclusions

In this paper we addressed the issues in reliable support for long lived parallel applications in a loosely coupled distributed environment. The basis for the paper was that in these domains, cooperation among processes necessitates the requirement for high availability in data and processes.

We looked at the features in Linda that make it a suitable environment for fault-tolerance. We then introduced a way of implementing highly available processes in Linda. Our approach for achieving high availability was to integrate the approaches of fault-tolerance with replication. We were motivated in the design of our scheme by the ease of checkpointing, the reduced effort during recovery, and by the limited amount of replay necessary.

## References

1. R. Bjornson, N. Carriero, D. Gelernter, and Jerrold Leichter, "Linda, the Portable Parallel," Yale Univ. Dept. Comp. Sci. RR-520, January 1988.
2. N. Carriero and D. Gelernter, "Linda in Context," *Communications of the ACM*, vol. 32, 4, pp. 444-458.
3. N. Carriero, D. Gelernter, and Jerrold Leichter, "Distributed Data Structures in Linda," *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1986.
4. A. Borg, J. Baumbach, and S. Glazer, "A Message System Supporting Fault Tolerance," *Proceedings of the ninth Symposium on Operating Systems Principles*, pp. 90- 99, 1983.
5. M. L. Powell and D. L. Presotto, "Publishing- A Reliable Broadcast Communication Mechanism," *Proceedings of the ninth Symposium on Operating Systems Principles*, pp. 100-109, 1983.
6. E. Cooper, *Replicated Distributed Programs*, pp. 63-78, ACM, 1985.
7. L. V. Mancini and S. K. Shrivastava, "Replication within Atomic Actions and Conversations: A Case Study in Fault-Tolerance Duality," *Proceedings of the nineteenth Symposium on Fault Tolerant Computing*, pp. 454-461, 1989.
8. T. A. Joseph and K. P. Birman, "Reliable Broadcast Protocols," *Cornell University, TR 88-918*, June 1988.
9. R. D. Schlichting and F. B. Schneider, "Fail Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computing Systems*, vol. 1,3, pp. 222-238, 1983.
10. G. Neiger and S. Toueg, "Automatically Increasing the Fault-Tolerance of Distributed Systems," *Proceedings of the seventh ACM Symposium on Principles of Distributed Computing*, 1988.

11. K. Eswaren, J. Gray, R. Lorie, and I. Traiger, "The notion of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, pp. 624-633, Nov 1976.

12. K. P. Birman and T. A. Joseph, "Exploiting Replication," *Cornell University TR-88-917*, June 1988.

13. P. Jalote, "Fault Tolerant Processes," *Distributed Computing*, vol. (1989)3, pp. 187-195, Springer Verlag.

14. J. Walpole and S. Kambhatla, *Replication issues for Long-Lived Parallel Computations in a Loosely-Coupled Distributed Environment*, 1990. The Workshop on Management of Replicated Data, 1990.

15. P. Leu and B. Bhargava, "Concurrent Robust Checkpointing and Recovery in Distributed Systems," *Proceedings of the Eighth International Conference on Data Engineering*, pp. 154-163, 1988.

16. G. Barigazzi and L. Strigini, "Application Transparent Setting of Recovery Points," *Proceedings of the thirteenth IEEE Symp on Fault Tolerant Computing*, Milano, Italy, 1983.

17. R. Koo and S. Toueg, "Checkpointing and Rollback Recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-13, 1, pp. 23- 31, 1987.

18. Y. Tamir and C. H. Sequin, "Error recovery in multicomputers using global checkpoints," *Proceedings of the thirteenth International Conference on Parallel Processing*, 1984.

19. J. F. Bartlett, "A Nonstop Kernel," *Proceedings of the eighth Symposium on Operating Systems Principles*, pp. 22- 29, 1981.

20. R. E. Strom and S. Yemini, "Optimistic Recovery: an asynchronous approach to fault tolerance in distributed systems," *Fourteenth International Fault Tolerant Computing Symposium*, pp. 374- 379, 1984.

21. D. B. Johnson and W. Zwaenepoel, "Sender based Message Logging," *Seventeenth International Fault Tolerant Computing Symposium*, pp. 14- 19, 1987.

22. A. S. Xu, "A Fault Tolerant Network Kernel for Linda," *MIT Laboratory for Computer Science, Master's Thesis*, 1988.

23. A. S. Xu and B. Liskov, "A design for a Fault Tolerant, Distributed Implementation of Linda," *Proceedings of the ninth International Symposium on Fault Tolerant Computing*, pp. 199 - 206, IEEE, 1989.

24. A. El Abbadi, D. Skeen, and F. Christian, "An Efficient Fault Tolerant Protocol for Replicated Data Management," *Proceedings of the Fourth ACM Symposium on Principles of Database Systems*, pp. 215-228, Portland, 1985.