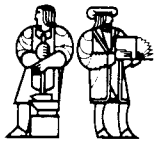
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-424

A FAULT-TOLERANT NETWORK KERNEL FOR LINDA

Andrew S. Xu

August 1988

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

A Fault-Tolerant Network Kernel for Linda

by

Andrew S. Xu

August 1988

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

Massachusetts Institute of Technology
Laboratory of Computer Science
Cambridge, Massachusetts 02139

A Fault-Tolerant Network Kernel for Linda

by

Andrew S. Xu

Submitted to the Department of
Electrical Engineering and Computer Science on July 20, 1988
in partial fulfillment of the requirements for the Degree of
Master of Science in Computer Science

Abstract

The parallel programming system Linda consists of a number of processes and a shared memory called the *tuple space*. In a distributed implementation of Linda, processes and the tuple space reside on different computing nodes connected by a communications network subject to a variety of node and network failures. This thesis develops a scheme to make the tuple space highly-available in the presence of failures.

High-availability is achieved by replication: the tuple space is replicated on several nodes so that failures usually do not disrupt program execution. Our replication method has two parts: the *operations protocol* and the *view change algorithm*. The operations protocol is a read-one-write-all scheme, that is, values are read from one of the replicas and write operations are executed at all replicas. The protocol exploits the semantics of the tuple space operations to eliminate unnecessary delay in program execution. When failures occur, the replicas are reorganized and their states are updated. This process is called a *view change* and is accomplished by the view change algorithm. A view change guarantees that a newly formed view consists of a majority of the replicas, and that all updates survive into the new view. Together, the operations protocol and the view change algorithm ensure that operations are executed in the correct order, updates to the tuple space survive failures, and processes only see the correct tuple space state in spite of failures. In addition, operations are performed by a concurrent background process whenever possible.

Thesis Supervisor: Barbara Liskov
Title: NEC Professor of Software Science and Engineering

Keywords: Fault-tolerant, Highly-available, Replication, Distributed systems, Parallel computing, View change.

This report is a minor revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on July 20, 1988 in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science.

Contents

List of Figures

Acknowledgements

Every step forward in one's life journey carries fruits from years of plowing and weeding. It is difficult to discern where gratitude begins and ends. The people who helped to guide my current career direction and who taught me all through my life are too numerous to mention by name. Limited space here only enables me to enumerate the few directly involved in this thesis writing.

I am deeply indebted to my thesis advisor, Barbara Liskov. Her excellent guidance and comments in shaping up the problem and the solution in this thesis, her indispensable assistance in the presentation, and her emotional encouragements have not only led me through the research, but also conditioned my way of thinking.

I am grateful to my graduate counselor William Weihl for his deep understanding and much needed encouragement. Brian Oki, Gary Leavens and Mark Day have taught me everything from life to research. Dorothy Curtis and Paul Johnson deserve special thanks for their help in making the simulation of my solution possible. Boaz Ben-Zvi, Toby Bloom, Sanjay Ghemawat, Elliot Kolodner, Minoru Kubota, Deborah Hwang, Rivka Ladin, Sharon Perl, Radia Perlman, Mark Vandevoorde, Cathy Yelick and all the people mentioned above have made my experience at MIT inspirational and memorable.

David Gelernter, Nicholas Carriero, and Jerrold Leichter earn special thanks for their help in clarifying my questions and their boost to get me interested in the thesis topic.

Thanks to my dearest friends Joseph Dauben, Kenneth Manning, Pamela McCorduck, Joseph Traub, Hortense Calisher, Curtis Harneck, Wai-Mee Ching and Mark Gilpin for the invaluable friendship and affection.

Finally, my deepest gratitude to all members of the extended family: mother, Helen, Robert, Shirley, Arthur, John, Alex, and Edward. Together, we have shared the worst political turbulence imaginable as well as the happiest moments in my life. Split over three continents, we have never been closer before. I couldn't wish for a family that I would love more.

Chapter 1

Introduction

In the parallel programming system Linda [4] [13], processes (called workers in this thesis) are uncoupled in time and space: they store and pick up *logical tuples*, units of data in Linda, in a shared-memory-like data structure referred to as the *tuple space*. A typical Linda system consists of several workers and a tuple space. The tuple space is directly accessible to all the workers simultaneously. The workers read their data from, and deposit the results into, the tuple space. Computations can start as soon as all the data needed are available. Linda has been implemented on Encore and Sequent shared-memory multiprocessors, the S/Net bus-based message-passing network, the Intel iPSC hypercube link-based network, and the Ethernet-based multi-computer local area network [4][7][3].

This thesis develops a mechanism to make the implementation of Linda on a distributed system possible. A *distributed system* is a collection of geographically distributed computing nodes connected to a communications network. A communications network might be a local area net, or it might consist of a number of local area nets connected by a long haul net.

Some of the potential benefits of a distributed parallel processing system are the following:

- The existing uni-processor, probably heterogeneous, computers can be used to process large jobs in parallel instead of acquiring expensive new multi-processor machines.
- The placement of computers is not geographically restricted. Numerous computers from different geographic locations can work together on a single job. For instance, computers scattered in various buildings and floors can cooperate on tasks requiring larger computing power than any single one of them can handle.

- 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.

Distributed parallel processing systems, while providing the above benefits, also give rise to some potential problems. In addition to higher communication overhead than that of multi-processor systems where inter-processor communication is commonly done via fast speed data buses, networks are susceptible to failures: messages may be lost or duplicated, the network may fail (and thus disrupt normal communication or cause systems to be partitioned into subgroups that cannot communicate), or computing nodes may crash. It is important to have programs continue to run correctly in the presence of network and node failures.

This thesis addresses the problems that arise from the system failures in a distributed implementation of the Linda tuple space and presents an efficient protocol that makes the tuple space fault-tolerant, and thus highly-available. High availability is achieved by redundancy— a tuple space is replicated onto several, usually geographically distinct, nodes so that some of the replicas are able to provide information when the 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 cause some replicas not to receive needed information; duplicate messages may cause some replicas to receive extra information; a replica may have kept out-dated information after the recovery from its failures. The protocol presented in this thesis solves these problems.

The protocol consists of two parts: *the operations protocol* and *the view change algorithm*. The operations protocol guarantees the correct execution of the operations on a replicated tuple space. The view change management algorithm guarantees that the tuple space replicas contain an up-to-date and consistent state, and that effects of all completed operations survive subsequent failures.

Our protocol provides some attractive properties. First, the replication is completely

hidden from the user program, that is, the replicated tuple space appears to the workers as 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 still talk to one another. Third, very 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.

Two of the current Linda implementations were designed for networks. But neither of them provides highly-available tuple spaces in a general communications networks. Compared with these implementations, our protocol tolerates failures that are common in general networks and provides good performance.

The thesis makes three contributions:

1. It provides a fault-tolerant, efficient, distributed implementation for Linda.
2. It indicates how fault tolerance might be achieved for other parallel systems. Many parallel computations are long lived; fault-tolerance is especially interesting for them. In addition, the other advantages of distribution apply to any parallel system.
3. It extends the work on replication techniques by showing what can be done when the semantics of the operations (that is, the tuple space operations) are taken into account.

The remainder of the thesis is organized as follows. Chapter 2 introduces Linda. Chapter 3 gives an overview of our scheme, and outlines the two parts of the scheme: the operations protocol and the view change management. The detailed descriptions of these two parts are given in chapters 4 and 5, respectively. Chapter 6 discusses related research and extensions of our work.

Chapter 2

Linda

A Linda system consists of several processes, which we will refer to as *workers*, and a memory that is logically shared by the workers. The workers cooperate on jobs, communicating through the logically shared memory. A worker with data stores the data into the memory and one that needs to receive data retrieves them from the memory. There is no centralized synchronization among the workers other than the operations on the memory. Operations are executed as soon as the data needed are available.

This chapter describes the Linda data structure and its operations, uses a simple example to explain how a Linda program runs, and introduces the notion of a Linda kernel. More detailed descriptions of Linda can be found in [13] and [9].

2.1 Logical Tuples and Operations

The basic data unit in Linda is a *logical tuple*, or *tuple* for short. A tuple contains a *logical name* followed by one or more ordered data elements, which can be either data values such as "1", "true", and "John", or *formals*, which are typed variables that can be assigned some data value. For instance, ("X", 1, **true**), ("done") and ("A", "John", **formal** score) are valid tuples. "X" is the logical name, and "1", "true" are the data values of the first tuple. "done" is the logical name of the second tuple. In the third tuple, "A" is the logical name, "John" is a data value while "**formal** score" indicates that "score", a previously declared variable, is a formal.

The term *template* is used to refer to tuples that are the arguments of two of the Linda operations (see below). A template and a tuple may *match* using the following rules:

1. both must have the same logical names and the same number of fields,
2. corresponding fields must be type consonant,
3. corresponding data items must be equal, and
4. there must be no corresponding formals.

For example:

- ("X", 1, 2, 3, 4, 5) matches ("X", 1, 2, 3, 4, 5),
- ("X", **formal** i, 3, 4, 5) matches ("X", 2, 3, **formal** j, 5),
- ("X", 1, **true**) matches ("X", **formal** i, **formal** b),
- ("X", **formal** i, **true**) matches ("X", 1, **true**), and
- ("X", **formal** i, **true**) matches ("X", 1, **formal** b)

where i and j are previously declared as integer variables and b is a variable of boolean type. On the other hand,

- ("X", 1) does not match ("X", 1, **true**) because of rule (1),
- ("X", "abc") does not match ("X", 1) because of rule (2),
- ("X", 1) does not match ("X", 2) because of rule (3), and
- ("X", **formal** i) does not match ("X", **formal** j) because of rule (4).

Tuples are stored in a logically shared memory called a *tuple space*. Workers interact with tuples in a tuple space via three basic operations: **out**, **in**, and **rd**. An **out** operation takes a tuple as its argument, and an **in** or a **rd** operation takes a template as its argument. Let t be a tuple, and s be a template. **Out**(t) causes tuple t to be added to the tuple space; the executing worker continues immediately. **In**(s) causes some tuple t that matches s to be withdrawn from the tuple space; the values of the actuals in t are assigned to the formals in s, and the executing worker continues. If no matching t is available when **in**(s) executes,

the executing worker suspends until one is, and then proceeds as before. If many matching t's are available, one is chosen arbitrarily. **Rd**(s) is the same as **in**(s), with actuals assigned to formals as before, except that the matching tuple remains in the tuple space.

In addition to the above three operations, [9] also lists three other operations: **eval**(p), **inp**(s), and **rdp**(s). **Eval**(p) starts a process to execute the procedure p. It has little to do with the tuple space, and hence will be ignored in the thesis. **Inp**(s) and **rdp**(s) are similar to **in**(s) and **rd**(s), respectively, except **inp**(s) and **rdp**(s) are non-blocking: the executing workers do not block if there is no matching tuple in the tuple space. If there is a matching tuple to s, then **inp**(s) and **rdp**(s) will behave exactly the same as **in**(s) and **rd**(s), respectively. Otherwise, "no_match_found" is signalled. **Inp**(s) and **rdp**(s) will not be included in our protocol. We will discuss these two operations in chapter 6.

2.2 Programming in Linda

The Linda operators can be incorporated into a high-level language, transforming the language into a parallel programming language. A simple program that computes the inner-product of two matrices A and B is shown in Figure 2.1. It illustrates the use of these operators. The initialization creates several workers, stores A's rows and B's columns in the tuple space, and adds the tuple ("Next", 1), where 1 is the next element to be computed, to the tuple space. A worker first gets the next task by doing **in**("Next", **formal** NextElem). Then it reads A's row and B's column from the tuple space. The result is put back to the tuple space by **out**("result", DotProduct(row,col)). These results can then be used by some other computation.

2.3 Linda Kernel

A *Linda kernel* serves as a translator between Linda operations and the accesses to physical memories. It supplies a form of logically-shared memory without assuming any physically-shared memory in the underlying hardware.

A Linda kernel implemented on a network is called a *network kernel.* The only existing kernel implementations that approximate a network kernel are the S/Net kernel and the

```
Initialization
        eval(worker())                          % create one worker
        eval(worker())                          % create another worker
        .....                                   % create some more workers
        out("A", 1, A's-1st-row)                % put A's 1st row into the tuple space
        out("A", 2, A's-2nd-row)                % put A's 2nd row into the tuple space
        .....                                   % more of A's rows into the tuple space
        out("A", n, A's-nth-row)                % put A's nth row into the tuple space
        out("B", 1, B's-1st-col)                % put B's 1st column into the tuple space
        out("B", 2, B's-2nd-col)                % put B's 2nd column into the tuple space
        .....                                   % more of B's columns into the tuple space
        out("B", n, B's-nth-col)                % put B's nth column into the tuple space
        out("Next", 1)                          % next computation

Worker
        in("Next", formal NextElem)             % get next computation
        if NextElem = -1 then out("Next", -1) done exit end
        if NextElem < n * n then out("Next", NextElem + 1) else out("Next", -1) end
        i = quotient_of((NextElem - 1)/dim + 1)  % calculate the row of the result
        j = remainder_of((NextElem - 1)/dim + 1)% calculate the column of the result
        rd("A", i, formal row)                  % get A's row from the tuple space
        rd("B", j, formal col)                  % get B's column form the tuple space
        out("result", i, j, DotProduct(row, col))  % put the result into the tuple space
```

Figure 2.1: A program segment that computes a matrix inner-product using Linda operators

VAX-LAN kernel described in [4][7]. But neither kernel provides a highly-available tuple space in the face of failures. (We will discuss these two kernels in chapter 6.) The inability of the existing mechanisms to cope with network failures motivates the design of a new scheme for a network kernel that makes the tuple space continue to be available and uncorrupted in the face of failures such as node crashes and network partitions. Our scheme provides highly-available tuple space without sacrificing performance.

Chapter 3

Overview

Replication is the standard technique to increase 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 one or more copies 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 chapter 6, we will see that this constraint can be relaxed so that each tuple can be stored on a subset of the replicas.

Replication solves the availability problem, but gives rise to the others that do not exist in a single-copy tuple space scheme. These problems include inconsistencies caused by delayed or lost messages, or out-dated replicas on nodes that recover from failures. Our scheme is designed to solve these problems.

The scheme consists of two parts: the *operations protocol* and the *view change algorithm*. The operations protocol translates each logical operation into physical operations. For example, an $\mathbf{in}(s)$ operation issued on a worker is translated into several physical $\mathbf{in}(s)$ operations performed on all the replicas. The view change algorithm is adopted from the virtual partitions protocol described in [1] and [2]. It guarantees the integrity of the accessible part of a tuple space during topological changes of the system. The term *view* will become understood as the chapter progresses.

This chapter gives an overview of our scheme. It starts by discussing the system model, the failure assumptions, and the definitions of partitions and views. Then it lists the goals we would like to achieve. Next we give an overview of our implementation of Linda operations,

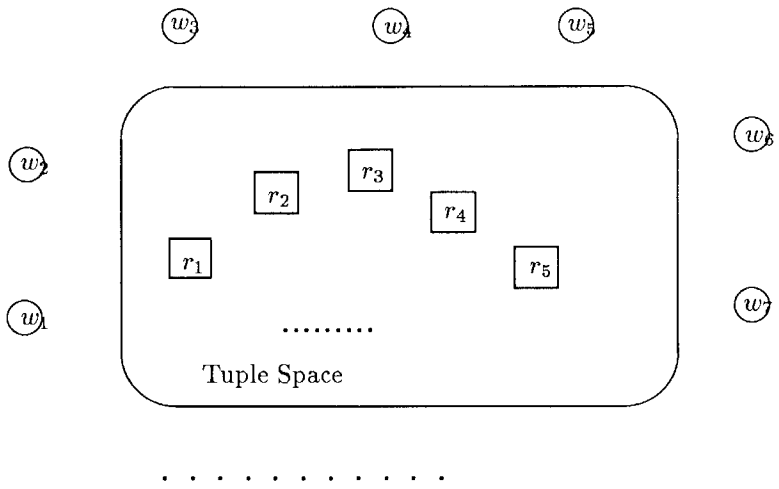


Figure 3.1: Workers and Replicated Tuple Space

and explain why the scheme works. An analysis of a set of constraints on the operations on a replicated tuple space follows. An overview of the view change algorithm is then given. Both the operations protocol and the view change algorithm will be discussed in detail in the next two chapters. Finally, we discuss the correctness conditions for our scheme.

3.1 Preliminaries

3.1.1 System Model

Our system consists of a set of tuple space replicas and a set of workers as illustrated in Figure 3.1. Squares r_1, r_2, r_3, ... are tuple space replicas; circles w_1, w_2, w_3, ... represent 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. All physical nodes are connected by a communications network subject to a variety of failures as discussed below.

Each replica is identified by its unique *replica id*. The replica ids are totally ordered. That is, if r_1_id and r_2_id are replica ids of two replicas r_1 and r_2, then there is a relation $\prec$ such that either $r_1\_id \prec r_2\_id$ or $r_2\_id \prec r_1\_id$ but not both. $\prec$ is transitive: if $r_1\_id \prec r_2\_id$

and $r_2\_id \prec r_3\_id$, then $r_1\_id \prec r_3\_id$.

3.1.2 Failure Assumptions

Failures can occur in many ways: node crashes, lost and duplicated messages, and even Byzantine failures [18], where system components may act in arbitrary, even malicious, ways. We will consider failures that have a reasonable chance of occurring in practical systems and that can be handled by algorithms of moderate complexity and cost. The failures satisfying these criteria include node and network crashes, lost or duplicate messages, message delays, and network partitions [1][10]. Byzantine failures are excluded. We assume that the nodes are failstop [24], that is, they fail by halting. Node and network crashes, lost messages, and delayed messages, cause messages *not* to be received by the receiver within a reasonable time interval. Duplicate messages cause certain messages to be received more than once. Network partitions divide a system into several subgroups where communication is possible within each subgroup, but impossible between any pair of the subgroups.

In general, it is impossible for a node to tell whether a failure to receive a message is due to a node crash or a network partition. This is because the effect of the failures, as a node perceives it, is the same — no message is received. Whether any message was ever sent, or was sent but not delivered, cannot be determined by an individual node. Thus, our scheme will not rely on distinguishing crashes from partitions.

3.1.3 Partition vs. View

A *partition* of a tuple space is a subset of replicas that can communicate with each other. We assume that the *can-communicate* relation between any two nodes is transitive and commutative. That is, if replica a can communicate with replica b, and replica b can communicate with replica c, then b can communicate with a, c can communicate with b, a can communicate with c, and c can communicate with a. Thus, every replica in a partition can communicate with every other replica in the same partition.

Partitions evolve dynamically. Initially, there is one partition containing all the tuple space replicas. The initial partition may be divided into several smaller partitions. The

smaller partitions may then merge to form larger partitions, or further subdivide into even smaller partitions. Figure 3.2 shows an example of the partition evolution process. The initial partition (r_1, r_2, r_3, r_4, r_5) becomes two partitions (r_1, r_2) and (r_3, r_4, r_5) after some failure at time t_1. In this partition situation, r_1 and r_2 can communicate with each other and r_3, r_4 and r_5 can communicate with each other, but none of the replicas in the first partition can communicate with any of the replicas in the second partition. At some time t_2, two new partitions, (r_1, r_2, r_3) and (r_4, r_5), are formed. Again, communication is possible among the replicas in the first partition and among those in the second partition, but there is no possible communication between a replica in the first partition and a replica in the second partition.

The *view of a worker w* is defined to be a set of replicas that w thinks that it can access. A *view of a replica r* is defined to be a set of replicas that r thinks that it can access [1]. A view always contains a majority of replicas in the system (to be explained in Chapter 5).

Worker and replica views can change over time. Replicas can initiate a *view change* algorithm when they think that there is a change in the network topology. The view change algorithm will be explained in more detail later. For now, it suffices to know that as the result of a view change, a new view may be established and the replicas in the new view will agree on a common view.

Associated with each view is an unique *viewid*. A viewid contains a sequence number n and the replica id, *r_id*, of the replica that initiated the view. That is:

$$viewid = \mathbf{record}[n : \mathbf{int}, r\_id : replica\_id]$$

Viewids are totally ordered by the relation $<$:

$$id_1 < id_2 \equiv (id_1.n < id_2.n) \vee ((id_1.n = id_2.n) \& (id_1.r\_id \prec id_2.r\_id))$$

where id_1 and id_2 are viewids, $id_1.n$ and $id_2.n$ are sequence numbers of id_1 and id_2, respectively, and $id_1.r\_id$ and $id_2.r\_id$ are replica ids of the replicas that initiated id_1 and id_2, respectively.

[1] Views are referred to as *virtual partitions* in [1].

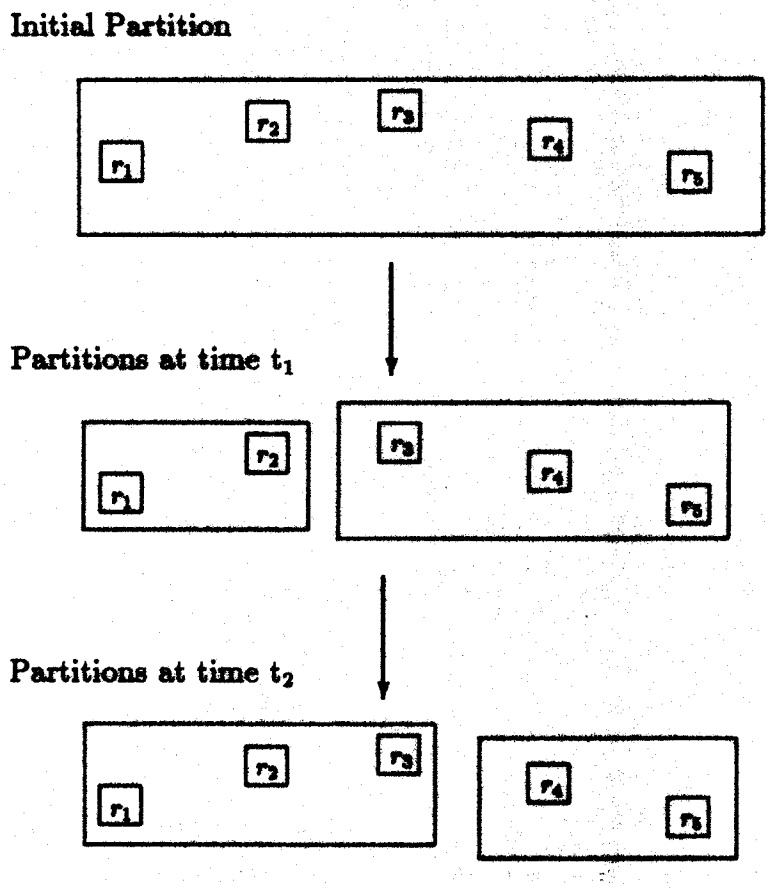


Figure 3.2: An Example of Partitioning

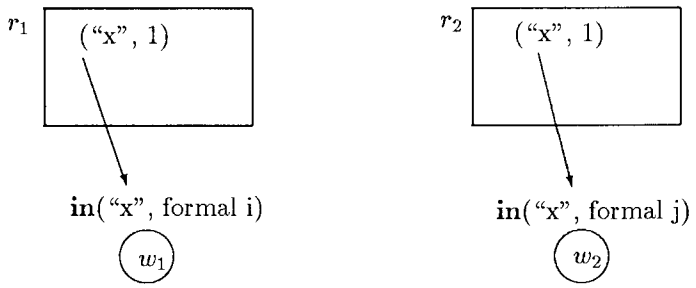


Figure 3.3: Inconsistency Scenario One: Concurrent **in** operation extract the same tuple from the replicas.

It is important to realize that views and partitions are different concepts. Partitions represent the physical configurations of a system while views are what workers and 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 many reasons. One possibility is that changes in network topology happen abruptly and replicas and workers cannot detect the changes instantly. Another possibility is that lost messages may change workers' and replicas' views of the "world" even when no physical change takes place.

3.2 Design Goals

The design of our network Linda kernel is driven by the following set of high-level goals:

- 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 (for example, 3 out of 5 or 251 out of 500) can communicate with each other, the tuple space is available.

- Consistency — The replicated tuple space ought to present a consistent state to the workers. The user programs should not be aware of whether the tuple space is replicated or not, except for the higher availability of a replicated tuple space.

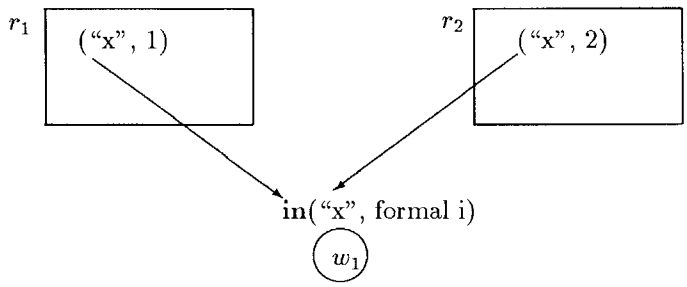


Figure 3.4: Inconsistency Scenario Two: The same **in** operation extracts different tuples from the replicas.

Therefore, multiple copies of the tuple space should not cause any anomalies for **out**, **in**, and **rd** operations. The result of these operations must be the same as if there were only one copy of tuple space available. For instance, concurrent **in** operations must not extract the same tuple from different replicas (Figure 3.3 illustrates an anomaly where the same tuple, ("x", 1), on r_1 and r_2 is extracted by concurrent **in** operations on w_1 and w_2), and the same **in** should not delete different tuples from different replicas (the problem can be seen in Figure 3.4 where two different tuples on r_1 and r_2 are extracted by the same **in** operation on w_1).

- Efficiency — Operations should perform efficiently to support requirements of the parallel programming paradigm. Except for satisfying a set of semantic constraints (as will be discussed below), no delays should be imposed on the user programs.

Having enumerated the goals, we are ready to give an overview of how operations are performed in a network Linda kernel.

3.3 General Scheme for the Operations

The principal idea behind our network kernel is to use an operations protocol in conjunction with the view change algorithm. This section gives the reader an overview of how Linda operations are implemented on a replicated tuple space. We assume that workers do not

fail; we discuss a method to cope with workers' failures in Section 6.3.2.

In this thesis, we assume that a tuple space is implemented as a set of tuple sets. Each tuple set contains all the tuples with the same logical name. There is a lock associated with each tuple set.[2] 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.

To simplify the presentation, we do not concern ourselves with view changes in this section. The assumption is that workers' views are accurate and no event occurs that would invalidate them. This assumption allows us to understand the operations without getting involved in the details of the view change.

3.3.1 Operations

Let w be a worker executing the operation. The three operations on a replicated tuple space are implemented as follows:

- **Out**(t) — The request to execute the operation is broadcast to all the replicas in w's view, and w waits for acknowledgments from the replicas.

  At each replica, t is stored into the local copy of the tuple space, and an acknowledgment is sent to w.

  If w does not receive acknowledgments from all the replicas in its view, it repeats the request until all the acknowledgments have been received. It is replicas' responsibility to discard redundant requests for the same **out**.

- **In**(s) — This is done in two phases:

  - Phase One (**in1**) — W sends template s to all the replicas in its view.

    Each replica searches its local copy of the tuple space for matching tuples. The tuple set for tuples with s's logical name is locked, and a set containing all matching tuples is returned to w. If there is no matching tuple, an empty set

[2] 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 [11].

is returned. If the tuple set is already locked by another worker, w's request is refused.

If all the replicas in the view respond, none of the replies is a refusal, and there is a non-empty intersection of all the tuple sets w received, then 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.

If all the replicas in the view have not responded within a reasonable time or if all replicas responded and the intersection is empty, phase one is repeated after a timed delay.

If a majority of the replicas in w's view refused w's request, then w instructs the replicas to release the locks, and phase one will be repeated after some random time interval.

If a minority of the replicas refused, then w repeats the first phase until it gets locks on all the replicas in its view.

- Phase Two (**in2**) — W informs all the 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 locks set during the first phase, and send an acknowledgment to w. An **in2** is finished only when all the replicas have replied. Otherwise, it is repeated until they have. Again, repeated 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 the 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 the 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 neither 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 case of several competing workers who repeatedly acquire only a minority of locks can be avoided by introducing a random delay, so that workers make their next attempts to set the lock at different times.

- **Rd**(s) — Template s is broadcast to all the 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, it informs w that no matching tuple is found.

  Whenever w receives a tuple from any of the replicas, it assigns the actuals of the returned value to the formals of s, and the execution continues. Responses from the rest of the replicas are ignored.

  If no tuple is received within a reasonable delay, the **rd** is repeated until one is.

Notice that a modification operation (**out** or **in**) is *complete* only after it has occurred at all replicas, and that a worker continues to perform the operation at all replicas in its current view until it knows the operation is complete.

3.3.2 Properties of the Operations

From the basic operations scheme stated above, we can see that an **out**(t) operation does not concern itself with the current tuple space state. It simply deposits t into the tuple space. It is analogous to a *blind write*, a write operation that does not read the value of the written object first. Therefore, there is no need for a worker issuing an **out** operation to wait until the operation is finished. The execution of an **out** operation can be carried out in the background while program execution continues.

There is no need for the executing worker to be blocked while an **in2** is in process because the **in2** will not provide any information that is needed by the worker. Thus **in2**'s can be completed in the background. Completing an **in2** guarantees that the selected tuple is removed from all the replicas in the view and the locks set by the corresponding **in1**'s are released.

It is not hard to see that the worker executing a **rd** operation must be blocked until the first matching tuple is returned from a replica. Similarly, a worker executing an **in** operation must be blocked until the tuple to be removed is selected.

The background processing of **out** and **in2** allows multiple operations to be packaged in one message. It also introduces concurrency between running a worker and its use of the tuple space. However, the executions of the background operations need to satisfy a set of constraints that ensure the Linda semantics are preserved in the face of concurrency. 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 background **in2** has not completed by the time the **rd** is executed.

3.4 Constraints on Operations

To determine how much concurrency we can achieve without violating correctness, we need to define constraints on each operation. A plausible requirement is that *the state of the tuple space observed by each worker does not conflict with what it has done or observed in the past*[3] We let this requirement be our correctness criterion. We will first take a look at the *sequential constraints*, the constraints on the operations of a single worker, and then the *inter-worker constraints*, those imposed on the operations of different workers.

3.4.1 Sequential Constraints

This subsection investigates the constraints in an environment that has one worker and a possibly replicated tuple space. **Out** and **in2** are executed in the background concurrently.

Concurrent **out**'s will not cause problems. This is because both **rd** and **in** are nondeter-

[3]This requirement is known as one-copy serializability [6].

ministic and blocking. **Rd**(s) can use any matching tuple in the tuple space at the moment. If an **out**(t) was issued by the worker previously and t matches s, **rd**(s) may use t (if t is already in the tuple space), or it will simply wait (if t has not yet been stored into the tuple space and there is no other matching tuples in the tuple space) until t arrives. Similarly, **in1**(s) can lock any matching tuple in the tuple space at the moment. It will wait for a matching tuple to arrive (at all replicas) if there is not one already. Since **rd** is blocking, no later **out**'s may start until the current **rd** operation has returned. Similarly, **in**'s will block later **out** operations until **in1** has returned.

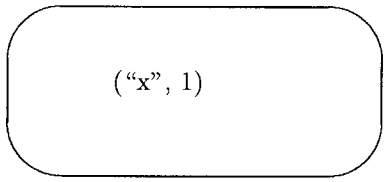
Unfinished **in2**'s may cause problems in that the tuple that was supposed to be removed by an **in** operation may still be in the tuple space when a later **rd** is executed. (A later **in** is not a problem because the locks will prevent it from seeing the effects of the earlier **in2** if both concern the same tuple set.) This is undesirable. To prevent this problem, we require that the operations be executed at each replica in the same order as they were issued by the worker. This requirement ensures that no **rd** can be executed at a replica before a previous **in2** is completed at that replica.

3.4.2 Inter-Worker Constraints

The inter-worker constraints are more subtle than those on a single worker because different workers run in parallel.

For example, Figure 3.5 illustrates the kind of problem that can arise. It shows a scenario where there are two workers and a replicated tuple space. 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. X, u, and v are previously declared integer variables in the workers' programs. In this example, the integer value associated with tuples with logical name "x" increases with time. In the figure, 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. To illustrate this,



Tuple Space Initial State

w_1	w_2
.....	.....
in("x", **formal** x)	**rd**("x", **formal** u)
out("x", $x+1$)	**rd**("x", **formal** v) % *expect* $v \geq u$
.....	.....

Figure 3.5: Inter-Worker Constraints

we use the same scenario above. Suppose the tuple space is replicated on r_1 and r_2, and both contain ("x", 1) at some point in time. Operations at w_1 and w_2 occur 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 remedy the problem above, we require that requests for an **out** operation not be sent to any replica until the previous **in** operations issued by the same worker are completed at *all* replicas in the current view. Thus, the tuple ("x", 2) cannot exist at r_2 until ("x", 1) has been removed from both r_1 and r_2 in the above example. So when **rd**("x", **formal** u) returns with ("x", 2) (from any replica), ("x", 1) has already been removed from every replica.

3.4.3 Summary

The sequential and inter-worker constraints are summarized as follows:

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

The secon d constraint is translated into "an **out** operation must not start until all previous **in2**'s issued on the same worker are completed at all replicas in the worker's view." The second constraint may cause a delay in the execution of the worker. The worker needs to wait for an **out** operation, but may be delayed by a subsequent **rd** or **in**. We expect that often there will be no delay, however, because previous **in**s will be completed by the time the **rd** or **in** is issued.

3.5 View Change Management

The failures mentioned in subsection 3.1.2 affect the replicas making up the tuple space. To mask these failures automatically and efficiently, and to preserve the single-image appearance of the tuple space, views were introduced.

Intuitively, a view reflects the changing communication capability among members of a partition. When the communication capability inherent in a view is believed to have changed, the replicas switch to a new view by executing the view change algorithm; our algorithm is a variation of the original virtual partitions protocol proposed by El Abbadi, Skeen, and Cristian [1]. As part of a view change, the view change algorithm generates a new viewid and a new view. The viewid of the new view is guaranteed to be greater than the viewid of any earlier view.

In Figure 3.6, we illustrate what the view change algorithm achieves. The original configuration of the tuple space is $\{r_1, r_2, r_3, r_4, r_5\}$, and the initial view of these replicas is $\{r_1, r_2, r_3, r_4, r_5\}$ with viewid $< 2, r_1 >$. Now suppose a communication failure makes it impossible for replica r_1 to talk to the others. When this failure is noticed, the system initiates a change in view. As a result of the view change, a new view $\{r_2, r_3, r_4, r_5\}$, is formed with viewid $< 2, r_5 >$.

A new view can be formed only when it contains a majority of the replicas in the original configuration. If this is impossible, the replicas remain in their old views. Thus, if a modification operation (**in1**, **in2**, or **out**) is completed at all the replicas in a view, this implies that at least a majority of the replicas know the effect of the operation. (Recall that a modification operation is complete only when it has occurred at all replicas in the

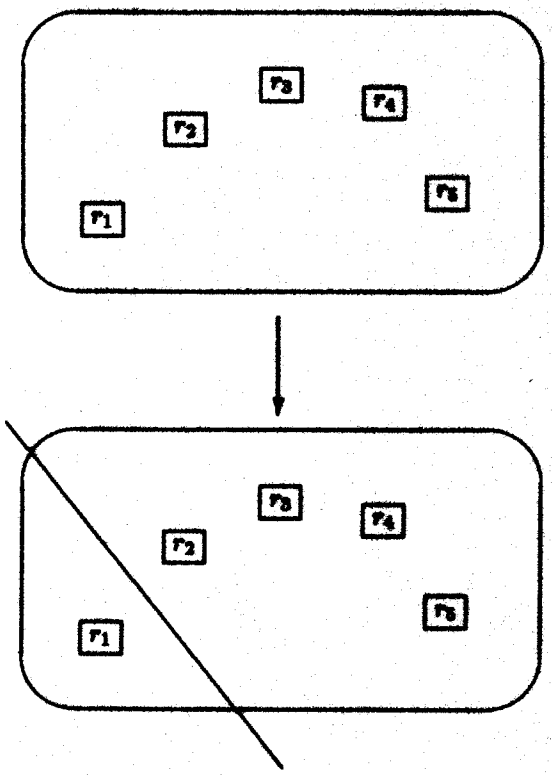


Figure 3.6: View Changes. The replicas constituting the tuple space react to a possible communication failure, such as a network partition, by changing views to exclude the inaccessible replica r_1.

worker's current view.)

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. As discussed below, this guarantees that effects of completed operations will persist into all later views.

3.6 Correctness

The correctness of our algorithm depends on the interaction of operation processing and the view change algorithm. In this section, we discuss the conditions that must be met for correct operation.

1. *The operations appear to happen in the correct order.*

   This condition is guaranteed by the two constraints summarized in subsection 3.4.3: the operations are executed at *each* replica in the order they are issued, and all **in** operations for a particular worker must be completed at *all* replicas in the current view before an **out** operation for that worker starts.

2. *Completed modification operations occur at all replicas in some view.*

   This is guaranteed by the operations protocol. Both **in** and **out** operations are completed only when their effects occur at all the replicas in the executing worker's view.

3. *The effects of completed operations survive into all subsequent views.*

   This is guaranteed by the view change algorithm. If the previous view contained a majority of replicas, and the new view also consists of a majority, then both views must have at least one replica in common that was in the previous view and is now in the new view. The state of the new view is taken from such a replica. Therefore, the new view starts out knowing what happened in the previous view. Since the effects of completed operations are known at all replicas in the old view, the effects of completed operations survive into all subsequent views.

Chapter 4

Operations Protocol

The execution of the operations protocol requires the cooperation of both the workers and the replicas. When a tuple space operation **out**, **rd**, or **in** is encountered by a worker, 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 the execution, the replica sends back either a result (if there is one) or a completion acknowledgment.

The messages that contain the requests or answers can be lost, delayed, or duplicated by the network. When a worker does not receive all the replies within an expected time interval, it repeatedly sends the requests until it gets the replies back from all the replicas in its view. This method solves the problems of lost and delayed messages, but not of duplicate messages (in fact, it generates duplicate messages). A remedy to this problem is included in the operations protocol.

The next section discusses the means of communication among workers and replicas. Section 4.2 explains a worker's participation in the operations protocol. The related activities on a replica are described in section 4.3. The operations protocol is summarized in section 4.4.

4.1 Communication Among Workers and Replicas

Communication is accomplished by sending and receiving messages using the **send** and **receive** statements. This section describes these statements, and the contents of messages exchanged between workers and replicas.

4.1.1 Send and Receive

The form of a **send** statement is

send(message_type, parm_list) **to** destination

where *message_type* is a string indicating the type of message sent, *parm_list* is a list of parameters containing the information to be sent, and *destination* is the id of the receiver, either a replica or a worker, of the message. As an example,

send("abc", my_id) **to** r_id

will send an message of type "abc" to the replica *r_id*. The parameter is *my_id*.

Messages are received using the **receive** statement. An example is the following:

receive
foo(x: **int**): S_1
bar(a: **char**, b: **string**): S_2
end.

If a message with a name matching one of those listed in an arm is waiting for the process executing the **receive**, it is selected and control continues at the statement in the matched arm. If there are several matching messages, one is selected nondeterministically. If there are no matching messages, the process waits until one arrives.

A second form of the **receive** statement allows the process to wait until a timeout expires. For example,

receive until t
foo(x: **int**): S_1
bar(a: **char**, b: **string**): S_2
end except when timeout: ... **end.**

If $t = 0$, this statment is identical to that above. Otherwise, the process waits for a matching message only so long as the time of the clock at its node is less than or equal to t; when its local time is greater than t, the statement terminates immediately with the timeout exception.

4.1.2 Contents of the Messages

This subsection describes the contents of the messages transmitted between workers and replicas.

A message from a worker to a replica is typically a request to execute a list of tuple space operations. In addition to the information needed to execute these operations, the parameters in such a message contain the worker's current viewid and the unique message's unique id, the *mid*.

The viewid in the message is compared at the receiving replica with the replica's viewid. If the worker and the replica have the same viewid, the requests are executed at the replica. Otherwise, if the replica has a more recent view, the worker is informed about the new view, and no operations are executed. If the replica has an old view, the worker's message is ignored.

The mid is used to weed out the duplicates and outdated replies. It is generated by the worker each time a message is sent. When a replica receives a message with an mid already seen before, the message is a duplicate, and is ignored. When a worker's request is completed, the replica sends back the result along with the mid received in the request. The mid received at the worker's side can be used to decide whether the reply is for the request just sent. Outdated replies (the replies with old mids) are weeded out.

4.2 Processing On a Worker

The last chapter explained that **out** and **in2** (the second phase of **in**) operations can be non-blocking — the program process does not have to wait until the results of the operations come back. In other words, the processing of **out** and **in2** operations can be done by some background process. This section introduces the notion of the foreground and background processes. Each worker contains a foreground process and a background process. The two processes communicate via a shared data structure called the *operations log*. The subsequent subsections explain the function of these components.

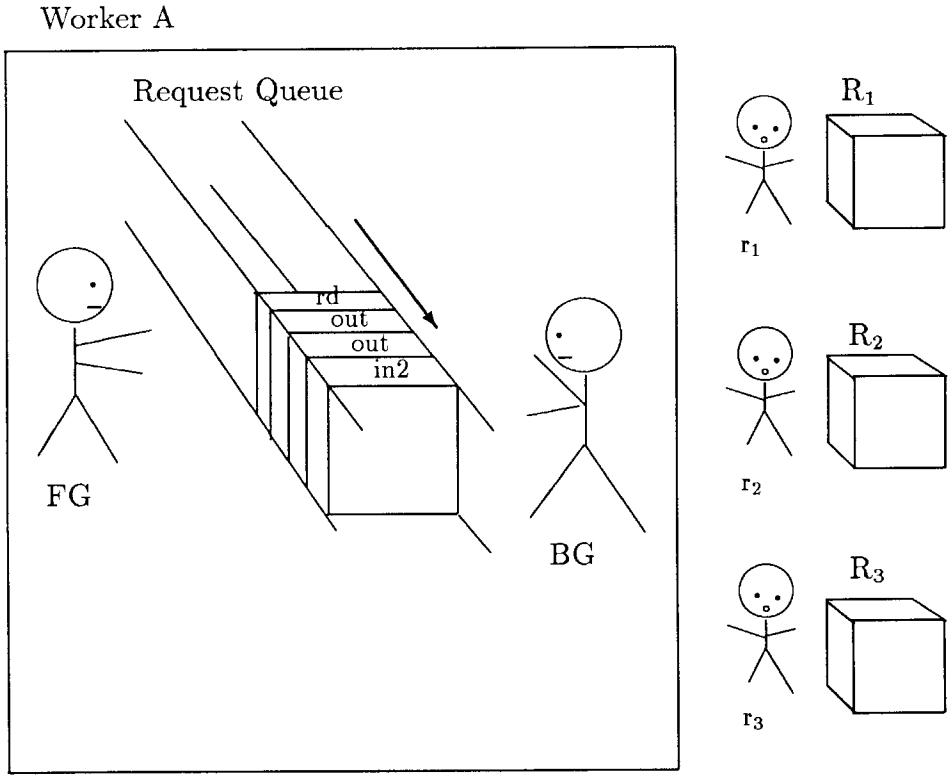


Figure 4.1: Replicas and Internals of a Worker

4.2.1 The Components of a Worker

Figure 4.1 illustrates the internals of a worker A and its relationship with the replicas (for example, R_1, R_2, and R_3). There are three major components of a worker: a foreground process (FG), a background process (BG), and an operations log that includes a request queue.

FG and BG communicate through the operations log. FG executes the program, including its accesses of the tuple space. It stores requests in the operations log. BG retrieves requests from the log, communicates with the replicas to carry them out, and stores results in the log. The requests for the operations that do not have results (**out** and **in2**) are removed from the operations log by BG after they are finished. When a result is expected (as in **rd**, **in1**, or **unlock**), BG updates the request entry on the operations log with the

result after the replies from the replicas are received. The result is picked up and the entry is removed by FG before it continues its execution.

4.2.2 Operations Log

The operations log of each worker synchronizes both FG and BG, and records the requests and answers. FG and BG can add, remove and update the requests on the operations log by calling one of the operations provided by *ops_log*, the operations-log data type shown in Figure 4.2. The internal representation of an *ops_log* is completely hidden from FG and BG.

The log contains five kinds of requests: **rd**, **out**, **in1**, **unlock**, and **in2**. The latter three requests are used to carry out an **in** operation: **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 are completed; other requests are ready if all earlier **out** requests are ready.

An operations log can be created by means of the *new* operation. FG calls *out*, *rd*, and *in* to add **out**, **rd**, or **in1** requests, respectively. The remaining operations are called by BG. The result of a **rd** request or an **in1** request can be delivered using *rd_ans* or *in1_ans*. The **out** request does not have a result. The completed requests can be removed from the operations log via the *finished* operation. A list of outstanding requests in the operations log can be obtained by calling *get_ops*.

Get_ops returns a list of ready operation requests; the list contains the requests in order. Figure 4.3 shows the format of these requests. *Rd_op* contains the template. *Out_op* contains t (the tuple to be stored in the tuple space) and *t_stamp* (the timestamp of the operation, to be explained later). *In1_op* contains the template, and **in2** contains the template s (whose matching tuples in the tuple space need be unlocked), t (the tuple to be deleted from the tuple space), and *t_stamp* (the timestamp of the operation). Finally, *unlock_op* contains the template whose matching tuples are to be unlocked.

ops_log = **abstract data type providing operations** new, rd, rd_ans, out, in, in1_ans, finished, get_ops

% *Ops_log* is a queue where requests for **rd**, **out**, and **in** are added to the
% top, and the finished requests are removed from the bottom or the top.

new = **proc**() **returns**(ops_log)
Return a new, empty operations log.

get_ops = **proc**(ol: ops_log) **returns**(ops) % *Ops is defined in Figure 4.3.*
If the operations log *ol* is not empty, return the operations in the log.
Otherwise, wait until *ol* is not empty and then return the operations.

out = **proc**(t: tuple, ol: ops_log)
Form an **out** request and add it to the operations log *ol*.

rd = **proc**(s: tuple, ol: ops_log) **returns**(tuple)
Form a **rd** request and add it to *ol*. Return with the result (a matching tuple to *s*) of the **rd**; at this point the **rd** request has been removed from *ol*.

rd_ans = **proc**(t: tuple, ol: ops_log)
Deliver a **rd** answer *t* to the **rd** request entry on *ol*.

in = **proc**(s: tuple, ol: ops_log) **returns**(tuple)
Form an **in1** request and add it to *ol*. Return a copy of the selected tuple matching *s*; at this point all other matching tuples are locked. An **in2** request is formed and added to *ol* before returning.

in1_ans = **proc**(lock_set, cur_view: replica_set, t_set: tuple_set, ol: ops_log)
Deliver the **in1** answer to the **in1** request entry on *ol*.
Lock_set is a set of replicas having locks. *Cur_view* is the worker's current view.
T_set is a set of tuples locked at all the replicas.

unlock_ans = **proc**(ol: ops_log)
Inform the **unlock** entry on *ol* about its completion.

finished = **proc**(k: **int**, ol: ops_log)
Remove the first k requests from ol, and $k \geq 0$.

Figure 4.2: Specification for Operations Log

```
ops = array[op]
op = oneof[rd: tuple, out: out_op, in1: tuple, in2: in2_op, unlock : tuple]
out_op = record[t: tuple, t_stamp: int]
in2_op = record[s: tuple, t: tuple, t_stamp: int]
```

Figure 4.3: Ops Type

```
        cur_view : view          % Initial value = set of all replicas
        cur_viewid: viewid       % Initial value = < 0, my_id >
        mid: int                 % Message id, initial value = 0
        my_id: worker_id         % Worker's id
        ol: ops_log              % Initial value = ops_log$new()
where
        view = replica_set
```

Figure 4.4: State of a Worker

There is at most one **rd**, **in1** or **unlock** request in the operations log at any given moment. This is because these operations block FG from further processing until the results or completion acknowledgments are received. The completed **rd**, **in1**, or **unlock** request is deleted from the operations log before FG continues its execution.

4.2.3 Worker State

Both FG and BG of a worker can change the worker's state. The state of a worker is summarized in Figure 4.4. *Cur_view* contains the set of replicas in the worker's current view. It always contains a majority of replicas in the system. No attempt is made to communicate with the replicas outside of *cur_view*. The variables in Figure 4.4 are initialized to their initial values before a program starts.

4.2.4 FG Processing

FG of a worker carries out the program processing. Whenever FG encounters an **out**, **rd**, or **in**, it invokes the corresponding procedure shown in Figure 4.5. These procedures interact with the operations log by adding the requests and picking up the results.

```
out = proc(t: tuple)
    ops_log$out(t, ol)
    end out

rd = proc(s: tuple)
    % s is mutated so that its formals are assigned the actuals of a matching tuple.
    t: tuple := ops_log$rd(s, ol)
    tuple$assign(s, t) % Assign the actuals of t to the formals of s.
    end rd

in = proc(s: tuple)
    % s is mutated so that its formals are assigned some values.
    t: tuple := ops_log$in(s, ol)
    tuple$assign(s, t) % Assign the actuals of t to the formals of s.
    end in
```

Figure 4.5: *Out*, *Rd*, and *In* Procedures

4.2.5 BG Processing

BG actively checks if there are outstanding operation requests on the operations log. If so, it sends a copy of the operations to all the replicas in the worker's view, and waits until it is informed that the operations have been executed at all the replicas. When a list of operations is sent to a replica, it is guaranteed that the order of the operations remains the same during the transmission. At the replica, the operations are executed in the same order.

The worker's *cur_viewid* is piggybacked on the operations list. If the worker's view is the same as the replica's, the operations are executed, and their completion and results (if any) are acknowledged by the replica. If the worker's view is more recent than that of the replica's, the operations are ignored. If the worker's view is old, the operations are ignored, and BG is informed about the new view. Whenever BG receives a new view, it updates *cur_view* and *cur_viewid* of its worker.

If, within a reasonable delay, BG does not receive acknowledgments from all the replicas in its *cur_view* for the operations sent, the same message is repeated (with a new mid) until all the replies are received.

```
while true do
    mid := mid + 1
    ops_list: ops := ops_log$get_ops(ol)
    k: int := ops$size(ops) % The number of operations in ops_list.
    % The following four variables keep track of reply information to various requests.
    r_set: replica_set := {} % The set of replicas that have replied.
    in1ans: tuple_set := tuple_set$all() % Set containing all tuples in the tuple space.
    lock_set: replica_set := {} % The set of replicas that have locks for in1.
    returned?: bool := false % Indicating if a result has been delivered to a rd request.

    for r: replica in cur_view do
        send("ops", ops_list, mid, my_id, cur_viewid) to r
        end % for

    t1: int := current_time() + δ1
    while true do
        receive until t1
            tag rd_ans(m: int, rr: replica, found?: bool, t: tuple):
                if m ≠ mid then
                    continue % continue to the next iteration of inner while loop.
                    end % if
                if found? & ~returned? then
                    ops_log$rd_ans(t, ol)
                    if k = 1 then break % exit inner while loop
                        else returned? := true
                        end % if
                    end % if
                r_set = r_set ∪ {rr}
                if (|r_set| = |cur_view|) then
                    ops_log$finished(k − 1, ol)
                    break
                    end % if
            tag in1_ans(m: int, rr: replica, locked?: bool, t_set: tuple_set):
                if m ≠ mid then continue end % if
                if locked? then
                    lock_set := lock_set ∪ {rr}
                    in1ans := in1ans ∩ t_set
                    end % if
                r_set := r_set ∪ {rr}
                if |r_set| = |cur_view| then
                    ops_log$finished(k − 1, ol)
                    ops_log$in1_ans(lock_set, cur_view, in1ans, ol)
                    break
                    end % if
```

Figure 4.6: BG Routine Part I

```
            tag unlock_ans(m: int, rr: replica):
                if m ≠ mid then continue end % if
                r_set := r_set ∪ {rr}
                if |r_set| = |cur_view| then
                    ops_log$unlock_ans(ol)
                    break
                    end % if
            tag in2(m: int, rr: replica):
                if m ≠ mid then continue end % if
                r_set := r_set ∪ {rr}
                if |r_set| = |cur_view| then
                    ops_log$finished(k, ol)
                    break
                    end % if
            tag out(m: int, rr: replica):
                if m ≠ mid then continue end % if
                r_set := r_set ∪ {rr}
                if |r_set| = |cur_view| then
                    ops_log$finished(k, ol)
                    break
                    end % if
            tag newview(#: viewid, t_view: view):
                if # > cur_viewid then
                    cur_view := t_view
                    cur_viewid := #
                    break % continue to the outer loop
                    end % if
            end % receive
                except when timeout: break end % except
        end % while
end % while
```

Figure 4.7: BG Routine Part II

The BG routine is shown in Figures 4.6 and 4.7. (In the code, a **break** statement causes an exit from the smallest containing loop; a **continue** statement causes control to continue with the next iteration of the smallest containing loop.)

A completion acknowledgment or a result received by a worker from a replica corresponds to the last operation in the operations list sent. It is also an indication that all previous operations have been completed at that replica. Recall that if a **rd**, an **in1**, or an **unlock** is present in the operations log, it must be the last entry in the list. There might be any number of **out** operations in the list. Only one **in2** entry is possible at any given time since the completion of an **in1** operation implies that all previous operations (including **in2**'s) are completed.

For a **rd** answer, the first matching tuple returned (from any replica) is used to update the **rd** request entry in the operations log. If the **rd** is the only request on the operations log, the replies from all other replicas are ignored. Otherwise, BG has to wait until the replies from all the replicas in its *cur_view* are received, though only the first matching tuple is used in the result of the **rd** operation. This is because all previous operations (**out**'s or an **in2** or both) must be completed on all replicas in *cur_view* before the requests are removed from the operations log.

For an **in1** answer, BG must receive replies from all the replicas in the view in order to make the decision about which tuple to remove from the tuple space. Once all the replies are received, the previous requests are removed from the operations log.

When an **in1** cannot get the locks on a majority of the replicas in the view, the worker tries to release the locks by replacing the **in1** entry on the operations log by an **unlock** entry. **Unlock** must be the only entry on the operations log since all the previous requests are removed by **in1**. Therefore, when the replies from all the replicas in the view are received for an **unlock** entry, there is no need to remove any more requests from the operations log other than the **unlock** request itself.

For an **out** or **in2** request, when BG receives replies from all the replicas in *cur_view*, the completed requests can be removed from the operations log.

When the replica receives a new view message, it updates the local view and viewid if

```
reqs = array[req]
req = oneof[rd: rd_req, out: out_req, in1: in1_req, in2: in2_req, unlock : unlock_req]
rd_req = record[s: tuple, t: tuple]
out_req = record[t: tuple, t_stamp: int]
in1_req = record[s: tuple, t_set: tuple_set, all?: bool, maj?: bool]
in2_req = record[s: tuple, t: tuple, t_stamp: int]
unlock_req = record[s: tuple]
```

Figure 4.8: Request Queue Type

the viewid in the message is more recent. Otherwise, the new view message is ignored.

If not all the replicas have responded to the requests within a reasonable time, the requests in the operations log are sent to the replicas again, and the whole process is repeated.

Note that the log can contain the following requests: 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.

4.2.6 Implementing the Operations Log

This section describes the implementation of the operations log specified in Figure 4.2. In addition to synchronizing FG and BG and recording requests and answers, the operations log also assigns timestamps to requests that need them. The importance of the timestamps will be discussed in the next section.

An operations log consists of a request queue, a timestamp generator, two boolean flags, and the tickets. The request queue, *reqs*, is an array of requests. The format of the requests is shown in Figure 4.8. A request is enqueued by calling *addh*, which appends the request at the back of the array; a request is dequeued by calling *reml* or *remh*; these operations remove an entry from the front or the end of the array, respectively. The array operations *addh* and *reml* are *indivisible*, that is, no other operations can be executed on the array when *addh* and *reml* are in progress. This keeps the queue from being updated by both

ticket = **abstract data type providing operations** init, await_ge, await, dec, inc

% *Ticket* is a mutable container of an integer.

init = **proc**() **returns**(ticket)
Return a new ticket containing zero.

await = **proc**(t: ticket, n: int)
Return when the ticket t contains n.

await_ge = **proc**(t: ticket, n: int)
Return when the ticket t contains a value greater than or equal to n.

dec = **proc**(t: ticket, n: int)
Reduce t by n. *Dec* is indivisible.

inc = **proc**(t: ticket, n: int)
Increase t by n. *Inc* is indivisible.

end ticket

Figure 4.9: Specification for Ticket

FG and BG simultaneously.

The timestamp generator *timestamp* is implemented as an integer counter that assigns a new timestamp to a request to be enqueued when needed. A new timestamp is generated by incrementing the integer.

The flags are used to determine when requests are ready. Flag *in2?* is true whenever there is an **in2** request in the log; *inout?* is true if an **out** request follows this **in2** request.

The tickets *#reqs* and *#ans* are used to keep track of the number of outstanding requests in the queue and the number of outstanding answers.

Tickets are specified in Figure 4.9. They provide operations to increment and decrement their values, and also to allow a process to wait for a ticket to have a specified value. Tickets allow FG and BG to synchronize with one another, for example, BG can wait until *#reqs* contains a value greater than or equal to 0.

```
ops_log = cluster is new, rd, rd_ans, out, in, in1_ans, unlock_ans, finished
                     get_ops

    rep = record[request_queue: reqs, timestamp: int, in2?, inout?: bool,
                 #ans, #reqs, : ticket]

    new = proc() returns(cvt)
        return(rep${request_queue: reqs$new(), timestamp: 0, in2?: false,
                          inout?: false, #ans, #reqs: ticket$init()}
        end new

    get_ops = proc(ol: cvt) returns(ops)
        % If ol.request_queue is not empty, return all ready request entries. Otherwise,
        % wait until ol.request_queue is not empty.
        ticket$await_ge(ol.#reqs, 1)
        temp_ops: ops := ops$new()
        for request: req in reqs$elements(ol.request_queue) do
            % req2op returns the corresponding op of req
            ops$addh(temp_ops, req2op(request))
            if inout? then return end % just return first element in this case
            end % for
        return(temp_ops)
        end get_ops

    out = proc(t: tuple, ol: cvt)
        % Log the out request on ol.
        ol.timestamp := ol.timestamp + 1
        oe: out_req := out_req${t: t, t_stamp: ol.timestamp}
        reqs$addh(ol.request_queue, oe)
        if in2? then inout? := true end
        ticket$inc(ol.#reqs, 1)
        end out

    rd = proc(s: tuple, ol: cvt) returns(tuple)
        % Return a copy of a tuple matching s.
        re: rd_req := rd_req${s: s, t: tuple$nil()}
        reqs$addh(ol.request_queue, re)
        ticket$inc(ol.#reqs, 1)
        ticket$await_ge(ol.#ans, 1)
        ticket$dec(ol.#ans, 1)
        reqs$remh(ol.request_queue)
        return(re.tuple)
        end rd
```

Figure 4.10: Operations Log Cluster Part I

```
rd_ans = proc(t: tuple, ol: cvt)
    % Deliver a rd ans t to ol; rd must be the top entry in ol.
    tagcase reqs$top(ol.request_queue)
        tag rd(re: rd_req):
            re.t := t
            ticket$dec(ol.#reqs, 1)
            ticket$inc(ol.#ans, 1)
        others: % Not possible.
        end tagcase
    end rd_ans

in = proc(s: tuple, ol: cvt) returns(tuple)
    % Return a copy of a selected tuple matching s while all matching
    % tuples are locked, and in2 request is logged on ol.
    ie: in1_req := in1_req${s: s, t_set: tuple_set$nil(), all?: false, maj?: false}
    reqs$addh(ol.request_queue, ie)
    ticket$inc(ol.#reqs, 1)
    while true do
        ticket$await_ge(ol.#ans, 1)
        ticket$dec(ol.#ans, 1)
        if ie.all? then % All replicas have locks.
            if ~ tuple_set$empty?(ie.t_set) then
                res: tuple := tuple_set$select(ie.t_set) % Any one will do.
                ol.timestamp := ol.timestamp + 1
                i2e: in2_req := in2_req${s: s, t: res, t_stamp: ol.timestamp}
                reqs$remh(ol.request_queue)
                reqs$addh(ol.request_queue, i2e)
                ol.in2? := true
                ticket$inc(ol.#reqs, 1)
                return(res)
                else % i.e., if ie.all?=true & ie.t_set={}, repeat in1.
                    ie.all? := false
                    ticket$inc(ol.#reqs, 1)
                end % if
```

Figure 4.11: Operations Log Cluster Part II

```
            elseif ~ ie.maj? then % No majority locks — Unlock.
                ue: unlock_req := unlock_req${s: s}
                reqs$remh(ol.request_queue)
                reqs$addh(ol.request_queue, ue)
                ticket$inc(ol.#reqs, 1)
                ticket$await_ge(ol.#ans, 1)
                ticket$dec(ol.#ans, 1)
                reqs$remh(ol.request_queue)
                reqs$addh(ol.request_queue, ie)
                ticket$inc(ol.#reqs, 1)
            else % i.e., ie.all? = false, ie.maj = true, repeat in1.
                ie.maj? := false
                ticket$inc(ol.#reqs, 1)
            end % if
        end % while
    end in

in1_ans = proc(lock_set: replica_set, cur_view: view, t_set: tuple_set, ol: cvt)
    % Inform ol that all the replies to the top entry (in1) are received.
    % lock_set is the set of replicas having locks.
    % t_sets is a set of common tuples locked by all replicas.
    tagcase reqs$remh(ol.request_queue)
        tag in1(ie: in1_req):
            if |lock_set| = |cur_view| then
                ie.all? := true
                ie.t_set := t_set
            else ie.maj? := ismaj?(lock_set, cur_view)
                % ismaj?(s1, s2) returns true if s1 is a majority of s2, and
                % returns false otherwise.
                end % if
            ticket$dec(ol.#reqs, 1)
            ticket$inc(ol.#ans, 1)
        others: % Not possible.
        end % tagcase
    end in1_ans

unlock_ans = proc(ol: cvt)
    % The unlock entry is done.
    ticket$dec(ol.#reqs, 1)
    ticket$inc(ol.#ans, 1)
    end unlock_ans
```

Figure 4.12: Operations Log Cluster Part III

```
    finished = proc(k: int, ol: cvt)
        % Notify ol that the first k entires have been processed. Purge all
        % these entries from ol.request_queue, and decrement #reqs and #in2s.
        for i: int in int$from_to(reqs$low(ol.request_queue),
                    reqs$low(ol.request_queue) + k - 1) do
            ticket$dec(ol.#reqs, 1)
            end % for
        in2? := false % reset flags since any in2 requests have now been removed
        inout? := false
        end finished
    end reqs_log
```

Figure 4.13: Operations Log Cluster Part IV

The implementation of the operations logs is shown in Figures 4.10–4.13. The basic strategy is the following:

1. Requests are added by enqueuing them on the request queue, incrementing *#reqs*, and setting *in2?* and *inout?* accordingly.

2. If an answer to a **rd**, an **in1**, or an **unlock** request is ready, the request on the request queue is updated, the *#reqs* ticket is decremented, and the *#ans* ticket is incremented. When the answer is picked up, the *#ans* ticket is decremented, and the entry on the request queue is deleted.

3. *Finished* removes the specified number of (**out** and **in2**) entries from the bottom of the request queue, decrements *#reqs* accordingly, and resets the *in2?* and *inout?* flags. Resetting the flags is appropriate since if there was an **in2** entry in the log, it has now been removed.

4. *Get_ops* blocks the calling process until the *#reqs* is greater than zero and then returns a list of operations corresponding to the ready requests in the request queue.

The tuple space operations **out**, **rd**, and **in** are processed as follows (refer to Figures 4.5, 4.6, 4.7, and 4.10 - 4.13):

- **Out**(t) — FG forms an **out** request and enqueues the request on the request queue, *inout?* is set to true if there is an **in2** entry in the log (*in2?* = **true**), and *#reqs* is incremented (the **out** operation can return at this point). This enables BG to receive the operation request using *get_ops*. When the **out** is finished, BG calls *finished* to remove the request from the request queue and to decrement *#reqs*.

- **Rd**(s) — The **rd** operation places the request on the request queue, increments the *#reqs* ticket, and waits until *#ans* becomes nonzero. When that happens, **rd** resets *#ans*, picks up the result in the **rd** entry on the queue, deletes the entry, and assigns the actuals in the result to the formals in s.

  The answer to the **rd** entry is delivered by BG by calling *rd_ans* when one of the replicas responds with a matching tuple. *Rd_ans* decrements *#reqs*, updates the **rd** request with the matching tuple, and increments *#ans*.

- **In**(s) — The **in** operation places an **in1** request on the request queue and increments *#reqs*. This causes BG to do the request and to return the answer by calling *in1_ans*, which stores the information obtained by BG in the entry, decrements *#reqs*, and increments *#ans*. Meanwhile **in** waits until *#ans* is nonzero. Then it resets *#ans* and checks the information in the updated **in1** entry. If all the replicas in the view have set the locks and the intersection of the returned tuple sets is not empty, a random tuple is selected from the intersection, the **in1** entry is replaced by an **in2** on the request queue, *in2?* is set, *#reqs* is incremented, the actuals of the selected tuple are assigned to the formals of s, and **in** returns. If a majority, but not all, of the replicas in the view have set locks, or if all have locks but the intersection is empty, the **in1** entry is left on the queue and *#reqs* is incremented to cause the request to be repeated by BG. Otherwise, the **in1** entry on the request queue is replaced by an **unlock** entry, and *#reqs* is incremented; this causes BG to release the locks. After the locks are released, the **unlock** entry is replaced by the **in1** request so that the **in1** can be tried again.

4.3 Processing On a Replica

The processing of a worker described above is coupled with the processing of a replica. Replicas are not only responsible for executing the operations on tuple space copies, but also for discarding out the duplicate messages. This section describes these activities.

4.3.1 Timestamp-Mid Table

Replicas may receive more than one message for the same operation, either because BG sends a request more than once or because of duplication in the network. The unequal mids are used to recognize and discard duplicates generated by the network, but are not sufficient to discard operation requests sent multiple times by a worker because a new mid is used every time a message is sent. Some operations can be repeated without causing any inconsistencies; others cannot. For instance, repeated **out**(t)'s will store multiple copies of t when only one is appropriate; repeated **in2**'s may cause too many tuples to be deleted. On the other hand, **rd** and **unlock** can be repeated without creating inconsistencies. We call **out** and **in2** *unrepeatable* operations. To avoid unrepeatable operations being executed more than once at a replica, a timestamp is associated with each unrepeatable operation. Each replica keeps a table of the last timestamp seen for each worker. These timestamps indicate the workers' *high water marks* — all the unrepeatable operations issued by a worker with timestamps at or below the worker's high water mark have already been executed, and should not be executed again.

Information about mids is also stored in the table. If a replica has seen the n-th message from a worker, then any message before the n-th is obsolete and can be ignored. Storing mids is not necessary for the correctness of the protocol. It is merely an optimization.

The timestamp and mid information about all the workers is kept by a replica using a table called the *timestamp-mid table*. Figure 4.14 gives the specification of the table. A table resides at each replica. It records the timestamp of the last unrepeatable operation the replica has executed, and the latest mid the replica has seen for each worker. There is at most one entry for each worker.

table = **abstract data type providing operations** new, get_ts, get_mid,
update_ts, update_mid

% A *table* contains the last timestamp seen and last mid received
% by a replica for each worker. There is at most one entry for each worker.
% Tables are mutable.

new = **proc**() **returns**(table)
Return a new table containing no entries.

get_ts = **proc**(tb: table, w: worker_id) **returns**(**int**)
Return the timestamp of the last unrepeatable operation issued by w.
If w is not already in *tb*, add an entry for w in *tb* with the
initial timestamp and mid, and return the initial timestamp.

get_mid = **proc**(tb: table, w: worker_id) **returns**(**int**)
Return the most recent mid of w. If there is no entry for w in *tb*, create
one with the initial timestamp and mid, and return the initial mid.

update_ts = **proc**(tb: table, w: worker_id, ts: int)
Update w's timestamp field with *ts*.

update_mid = **proc**(tb: table, w: worker_id, mid: int)
Update the mid field of w with *mid*.

end table

Figure 4.14: Specification for Timestamp-Mid Table

4.3.2 Tuple Space

Each replica keeps a copy of the tuple space. In our protocol, we have assumed that a tuple space is implemented as a set of tuple sets. The tuples with the same logical name are grouped into the same set. Each set has a lock. When a set is locked by one worker, no other worker can place a lock or delete any of the tuples from the set until the lock is released. Reading of a locked tuple is allowed, however.

The specification of the tuple space and its operations is given in Figure 4.15.

Notice that there can be only one lock on a tuple set at any given moment. When a tuple set is locked, the tuples in the set can be deleted only by the worker that set the lock. A locked tuple set can still accept tuples stored by other workers. The new incoming tuples are automatically locked once they enter a locked set.

4.3.3 Replica State

The part of a replica's state that affects the operations protocol is summarized in Figure 4.16. Initially, the local view and its id are undefined on each replica. An execution of the view change protocol is necessary to form a meaningful view. This will become clear in the next chapter. The local tuple space copy and the timestamp-mid table are initialized to using *tuple*$*new*() and *table*$*new*(), respectively.

4.3.4 Executing Operations

When a replica is "active", it calls the procedure *execute_ops*, shown in Figures 4.17 and 4.18, whenever it receives an operations list from a worker. The arguments needed are the following: *ops_list* (the operations list), *mid* (the mid corresponding to the message sent by the worker), w (worker's id), and # (worker's view id).

4.4 Summary

This chapter has described the operations protocol in detail. Its execution requires the coupling of both the worker's processing and part of the replica's processing.

---

tuple_space = **abstract data type providing operations** new, delete_unlock, lock, unlock, search, store

new = **proc**() **returns**(tuple_space)
Return a new empty tuple space.

store = **proc**(tspace: tuple_space, t: tuple)
Store t in *tspace*.

lock = **proc**(tspace: tuple_space, s: tuple, w: worker) **returns**(tuple_set) **signals**(refused)
If the set containing tuples with s's logical name is not yet locked by a worker other than w, lock the set and return the set of matching tuple(s). (If there are no matching tuples, return an empty set.) If the tuple set has already been locked by a worker other than w, signal *refused*.

unlock = **proc**(tspace: tuple_space, s: tuple, w: worker)
Unlock the set that has the same logical name as s and is locked by w, if such a set exits. Otherwise, do nothing.

delete_unlock = **proc**(tspace: tuple_space, t: tuple, s: tuple)
Delete t from *tspace* and unlock the tuple set that matches s.

search = **proc**(tspace: tuple_space, s: tuple) **returns**(tuple) **signals**(not_found)
Search *tspace* for a tuple that matches s. If one is found, return it. Otherwise, signal *not_found*.

end tuple_space

Figure 4.15: Specification for Tuple Space

```
status: status          % replica is active or doing a view change
cur_view: view          % Initial value = undefined.
cur_viewid: viewid      % Initially = undefined.
my_id: replica id       % Replica's id.
t_space: tuple_space    % Initial value = tuple_space$new()
tbl: table              % Initial value = table$new().

where

    status = oneof[active, view_manager, underling: null]
    viewid = <n: int, r: replica_id>
    view = replica_set
```

Figure 4.16: Replica State (Partial)

In addition to ensuring that the tuple space operations are executed on all the replicas in *cur_view* eventually, the protocol guarantees that no undesirable effects, such as storing or deleting too many tuples, result. To achieve this, the workers send their requests repeatedly until they are satisfied with the returned results, and the replicas discard the operations they have already executed. Timestamps and mids are used to detect duplicate operations and messages.

Unnecessary delay of program processing is avoided by the introduction of background process (at each worker), which continuously processes the requests generated by the program process. The program process is blocked only when it needs to know the result or to ensure the constraint that **in2**'s must be finished before **out**'s begin is obeyed.

The program (foreground) process and the background process at each worker communicate with each other via a data structure called the operations log. The log synchronizes the processes, logs the outstanding requests and results, and generates timestamps to prevent duplicate processing or unrepeatable operations. Another attractive feature of the operations log and the background process is that they provide a level of abstraction that hides the tuple space replication from the program process.

The correctness and efficiency of the operations protocol depend largely on the assumption that view changes are correctly taken care of by the view change algorithm. The next

```
execute_ops = proc(ops_list: ops, mid: int, w: worker_id, #: viewid)

    if mid <= table$get_mid(tbl, w) then return
        else table$update_mid(tbl, w, mid)
        end % if
    if # ≠ cur_viewid then
        send("newview", cur_viewid, cur_view) to w
        return
        end % if

    for operation: op in ops$elements(ops_list) do
        tagcase operation
            tag rd(e: tuple):
                found?: bool := true
                t : tuple := tuple$nil()
                t := tuple_space$search(t_space, e)
                    except when not_found: found? := false end % except
                send("rd_ans", mid, my_id, found?, t) to w
                return

            tag out(e: out_op):
                if e.t_stamp > table$get_ts(tbl, w) then
                    table$update_ts(tbl, w, e.t_stamp)
                    tuple_space$store(t_space, e.t)
                    end % if

            tag in1(e: tuple):
                locked?: bool := true
                t_set: tuple_set := tuple_set$nil()
                t_set := tuple_space$lock(t_space, e, w)
                    except when refused: locked? := false end % except
                send("in1_ans", mid, my_id, locked?, t_set) to w
                return

            tag in2(e: in2_op):
                if e.t_stamp > table$get_ts(tbl, w) then
                    table$update_ts(tbl, w, e.t_stamp)
                    tuple_space$delete_unlock(t_space, e.t, e.s)
                    end % if
```

Figure 4.17: Execute Operations Procedure I

```
            tag unlock(e: tuple):
                tuple_space$unlock(t_space, e, w)
                send("unlock_ans", mid, my_id) to w
                return

            end % tagcase
        end % for

        % If the last entry of ops_list is either an out or an in2 operation,
        % send a msg to w. The other three cases, rd, in1 and unlock, have
        % already had replies.

        tagcase ops$top(ops_list)
            tag out: send("out_ans", mid, my_id) to w
            tag in2: send("in2_ans", mid, my_id) to w
            others: % ignore
            end % tagcase
    end % execute_ops
```

Figure 4.18: Execute Operations Procedure II

chapter describes this algorithm.

Chapter 5

View Change Algorithm

The operations protocol explained above is a read-one-write-all scheme, that is, **rd** can return a result from any replica in the executing worker's view, but **out** and **in** operations are completed only if the executing worker knows that their effects are visible at every replica in its view. Thus, every replica in the worker's view knows all the completed operations that change the tuple space state.

Network and node failures cause some of the replicas to be inaccessible from the workers. If we let the workers access whichever replica they can access at the moment, an inconsistency may result. For example, suppose a network failure separates replica r from the rest of the system. While r is inaccessible, updates are made to other replicas. When the network is repaired and r becomes accessible, r's state is out of date, and must be brought up to date before being used again. The view change algorithm is used to mask the problems like this as well as to ensure that updates to the tuple space are not lost during failures.

The algorithm works roughly as follows: each replica processes a view consisting of the set of replicas it believes that it can communicate with. When a replica discovers that it no longer can communicate with some replica, or communication is re-established with a replica it could not hear from before, it starts a view change, and acts as the *view change manager* of the view change. During the view change, the manager constructs a globally unique new viewid, and sends a message to all other replicas, inviting them to join the new view. The invited replicas can choose to accept the invitation. Those that have accepted the invitation are called *underlings*. If a majority of replicas accept the invitation, a new view is formed and an up-to-date tuple space state is chosen to be used to initialize the

tuple space state of all members of the new view. During a view change, the manager and the underlings are blocked from workers' operation requests.

In the next section, we introduce the state information needed in order for a replica to provide service to workers' requests and run the view change algorithm. The mechanism to test accessibility of replicas is described in section 5.2. Section 5.3 gives an overview of the view change algorithm. Each replica is in one of three states: *active*, *view_manager* and *underling*. Active replicas execute workers' requests, monitor the topological changes in the network, and monitor view change invitations. View change managers coordinate view changes while monitoring invitations. Replicas in the underling state monitor invitations as well as participate in view changes. The replica activities in each of these states are detailed in sections 5.4, 5.5, and 5.6, respectively. Section 5.7 gives an example to illustrate the view change algorithm. An informal correctness argument is stated in section 5.8. We will make certain assumptions about crash failures during the discussion of the algorithm, namely that the replica state is stable and survives crashes. The full discussion of crashes is delayed until section 5.9, in which we will discuss a number of possible solutions to crash problems. A possible optimization is also discussed in section 5.9.

5.1 Replica State

The view change algorithm requires some information to be recorded in the replica state. This information is summarized in Figure 5.1 (an extension of Figure 4.16).

The current state of the replica is indicated by *status*, which is updated by the view change algorithm. Each replica knows the current view, *cur_view*, of which it is a member. *Cur_view* is identified by an unique viewid, *cur_viewid*. A replica also keeps a copy of the highest viewid it has seen, *max_viewid*. It is always true that *cur_viewid* is less than or equal to *max_viewid*. The set of all replicas in the system is represented by *orig_config*, which stands for original the configuration. The state of the tuple space copy is in *t_space*, and the timestamp-mid table described in the last chapter is kept using *tbl*.

When a replica is first created, *status* is *view_manager*; *t_space* has the value *tuple_space*$*new*(); *tbl* is *table*$*new*(); *my_id* is assigned the replica's id; *cur_view* and

```
status: status              % replica is active or doing a view change
t_space: tuple_space        % tuple space copy
tbl: table                  % timestamp-mid table
my_id: replica_id           % replica id
cur_viewid: viewid          % current viewid
cur_view: view              % current view
max_viewid: viewid          % highest viewid seen so far
orig_config: replica_set    % set of all replicas

where

status = oneof[active, view_manager, underling: null]
viewid = <n: int, r: replica_id>
view = replica_set
```

Figure 5.1: Replica State (Complete)

cur_viewid are undefined; *max_viewid* has the initial value {*0, my_id*}; and *orig_config* contains the ids of all the replicas in the system. One view change is necessary to let the replicas have a common *view* and *viewid* to work with.

We assume that the entire replica state is stored on stable storage [19]; we discuss this assumption in section 5.9.

5.2 Probes

The topological changes in the network are detected by sending and receiving probes. This is accomplished using two processes at each replica, one that sends probes and the other that receives them.

The probing procedure is shown in Figure 5.2, and works as follows. Probes are sent out to all other replicas in the system periodically, one every *probe_interval*. Every time a probe is sent, the probing process waits to collect the replies. It adds the replying replica's id in a temporary set *reply_set* if the reply is to the current probe. After a time interval δ_2, long enough for a round trip probe in the normal situation, the process checks to see if *reply_set* contains the same replicas as its current view. Any discrepancy, while the replica is in the active state, indicates that there *may be* a change in the network's configuration,

```
send_probes = proc()
    probe_interval: int := % fill in the appropriate probe period
    probe_seq: int := 0
    while true do
        if is_active(status) then
            for rr: replica_id in replica_set$elements(orig_config - {my_id}) do
                send("probe", my_id, probe_seq) to rr
                end % for
            reply_set: replica_set := {my_id}
            t2: int := current_time + δ2
            while true do
                receive until t2
                    probe_resp(r: replica_id, m: int):
                        if m = probe_seq then reply_set := reply_set ∪ {r} end % if
                    end % receive
                        except when timeout: break end % except
                end % while
            if is_active(status) cand (reply_set ~= cur_view) then
                send change(cur_viewid) to my_id
                end % if
            end % if
        probe_seq := probe_seq + 1
        sleep(probe_interval)
        end % while
    end send_probes
```

Figure 5.2: Send Probe

```
monitor_probes = proc()
    while true do
        receive
            probe(r: replica_id, m: int):
                send("probe_resp", my_id, m) to r
            end % receive
        end % while
    end monitor_probes
```

Figure 5.3: Monitor Probe

so the probing process sends a *change* message to another process (to be discussed later) of the same replica, which triggers a view change.

Notice that said there *may be* a reconfiguration instead of there *is* a reconfiguration. This is because lost or delayed messages may cause *reply_set* to be inconsistent with the replica's current view. But occasional message loss or delay does not always mean there is a topological change. Also notice that probes are sent only when a replica is in the "active" state.

The probes are monitored by the monitoring processes running *monitor_probes* (shown in Figure 5.3) at each replica. To ensure that replies correspond to the current probe, a sequence number is piggybacked on the probing message, and returned on the reply. This allows the probing process to consider only current replies.

5.3 Overview of the View Change Algorithm

As we said earlier, the probes provide a means of detecting possible network reconfigurations. Once a replica believes that it can no longer communicate with the same set of replicas it could previously, a *change* message is sent by the probing process. This message is received by the third process (on the same replica) which in turn initiates a view change. The replica switches from being active to being the manager of the view change.

The view change algorithm operates in one and a half phases. In the first phase, the manager constructs a new globally unique viewid, invites all replicas in the system to join

the new view, and waits for responses. A replica accepts the invitation only if it has not already received another invitation to join a higher-numbered view; each acceptance message contains the latest viewid and a copy of the replica's tuple space. We assume that a crashed replica recovers with its old state restored. This assumption guarantees that a replica either does not reply to an invitation, or replies with the tuple space state that corresponds to its current viewid. In section 5.9, we will discuss mechanisms to support this assumption.

The manager keeps a temporary copy of the tuple space and the timestamp-mid table, which has the replica's own tuple space copy at the beginning of the view change. Each incoming acceptance is checked, and the more up-to-date tuple space and table copy (indicated by the accompanying viewid) is used to update that temporary copy. So the temporary copy of the tuple space is always the most up-to-date copy the manager has seen.

If less than a sub-majority[1] of replicas accept the invitation, no new view can be formed. The replicas will repeatedly attempt to form another view until a view change succeeds. Otherwise, the view change enters the last half phase during which the manager sends a commit message to all the replicas that have agreed to join the view. The temporary copy of the tuple space and the timestamp-mid table is piggybacked on the commit message, and is used to update the state of all the replicas in the new view. The view manager becomes active once its local state is updated and the commit message is sent. The participating replicas become active when they receive the commit message and their local states are updated.

The algorithm is implemented as the third process of a replica (the first two being sending and monitoring probes). We call this process the *main process*. The main process is also responsible for executing a worker's tuple space operation requests. Figure 5.4 shows the state diagram of the view change algorithm.

In the "active" state, the replica sends and monitors probe messages, monitors view change invitations, and executes the operation requests from the workers. If probing triggers a view change, the replica moves to the "view_manager" state. If it receives an invitation

[1] A sub-majority is one less than a majority.

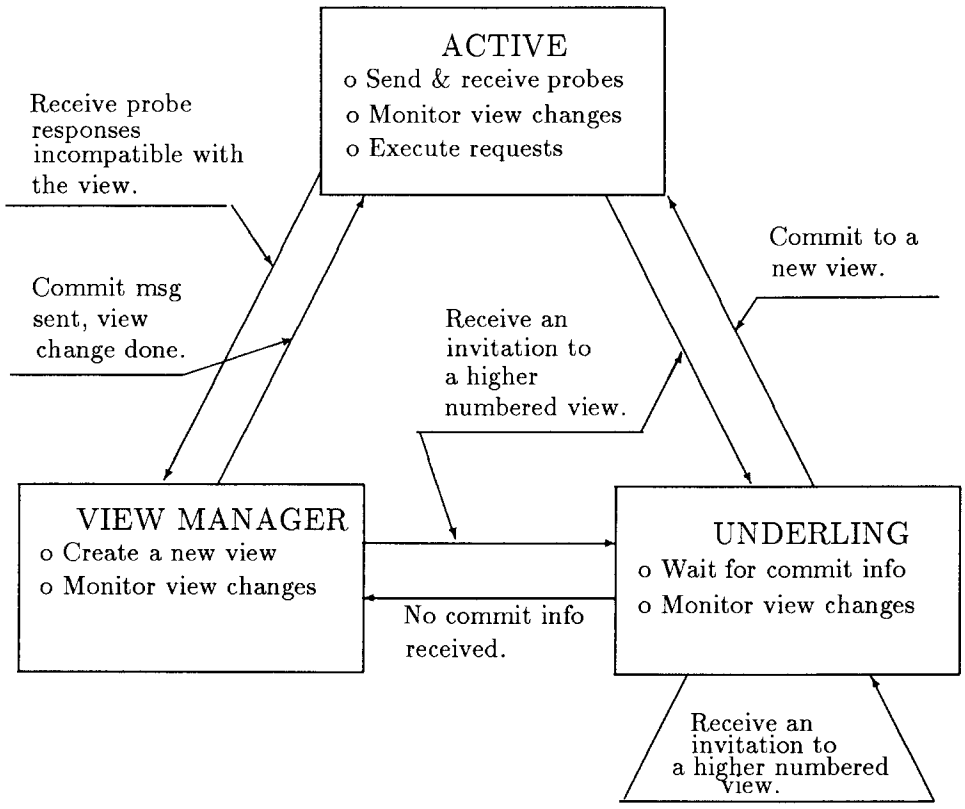


Figure 5.4: State Diagram for the View Change Algorithm

```
while true do
    tagcase status
        tag active: active()
        tag view_manager: view_manager()
        tag underling: underling()
        end % tagcase
    end % while
```

Figure 5.5: The View Change Algorithm

to join a view with higher viewid than the maximum it has seen, it changes to "underling" state and participates in a view change.

In the "view_manager" state, the replica coordinates a view change as well as monitors view change invitations. When a view change is done, it resumes execution in the "active" state. If, during the view change, the replica receives an invitation to join a view with a higher viewid than any it has seen, it becomes an "underling."

When a replica is an "underling," it is a participant in a view change. When it receives a commit message, it commits itself to the new view and enters the "active" state. If it does not receive a commit message within a reasonable time, it becomes to be a view manager and starts a view change. If it receives an invitation to join a higher numbered view, it accepts the invitation and remains in the "underling" state.

Figure 5.5 shows the program of the above state diagram. It is structured as an infinite loop. The replica determines its current state and calls the procedure that is executed while it is in that state. The next three sections discuss these procedures.

5.4 Active Replicas

Figure 5.6 shows the procedure for the active state. The main process in the "active" state receives three types of messages: *change*, *invite*, and *ops*. *Change* messages are sent by the probing process on the same replica when it suspects changes in communication capability. *Invite* messages are sent by other replicas when they start view changes. *Ops* messages are operation requests sent by the workers. The *execute_ops* procedure called

```
active = proc()
    receive
        change(vid: viewid):
            if vid < cur_viewid then return end % if view is already changed
            status := view_manager
        invite(vid: viewid, r: replica_id):
            if vid <= cur_viewid then return end % if an out-dated invitation
            max_viewid := vid
            send("accept", my_id, vid, cur_viewid, t_space) to r
            status := underling
        ops(ops: ops_type, mid: int, w: worker_id, vid: viewid):
            execute_ops(ops, mid, w, vid)
        end % receive
    end active
```

Figure 5.6: Active

upon receiving an *ops* message was illustrated in Figures 4.17 and 4.18 of the last chapter.

It is worth pointing out a possible race situation here. Suppose that the probing process of a replica r_1 sends a *change* message to the main process, at the same time r_1 receives an invitation to join a new higher numbered view from replica r_2. The main process of r_1 can nondeterministically select either message to receive first. If the *change* message is selected first, r_1 enters the "view_manager" state and competes with r_2 to change the view (we will mention the current view changes in a later section). If r_1's view change succeeds, its *cur_viewid* will be updated to reflect the new view. When the *invite* message is processed, the *vid* in the message is likely to be less than or equal to *cur_viewid*, and the message is thus ignored. On the other hand, if the *invite* message is received first, r_1 participates in r_2's view change. When the view change is completed and r_1 becomes "active" again, its *cur_viewid* is updated. This causes the *change* message to be ignored when it is received, since *vid* in the *change* message is the old *cur_viewid* on r_1, which must be lower than the new *cur_viewid*.

5.5 View Managers

Figure 5.7 shows the procedure run by the view managers. The local variable *t_ts* is the

```
view_manager = proc()
    t_ts: tuple_space := t_space
    t_vid: viewid := cur_viewid
    t_tbl: table := tbl
    n_view: replica_set := {my_id}
    max_viewid := <max_viewid.n + 1, my_id>

    for rr: replica_id in replica_set$elements(orig_config - {my_id}) do
        send("invite", max_viewid, my_id) to rr
        end % for

    t2: int := current_time + δ2
    while true do
        receive until t2
            accept(r: replica_id, vid, rtn_viewid: viewid, ts: tuple_space, ta: table):
                if vid = max_viewid then
                    n_view := n_view ∪ {r}
                    if t_vid < rtn_viewid then
                        t_vid := rtn_viewid
                        t_ts := ts
                        t_tble := ta
                        end % if
                    if |n_view| = |orig_config| then break end % if
                    end % if
            invite(vid: viewid, r: replica_id):
                if vid <= max_viewid then continue end % if
                max_viewid := vid
                send("accept", my_id, vid, cur_viewid, t_space) to r
                status := underling
                return
            end % receive
                except when timeout: break end % except
        end % while

    if ~ismaj?(n_view, orig_config) then return end % if
    cur_view := n_view
    cur_viewid := max_viewid
    t_space := t_ts
    tbl := t_tbl
    for rr: replica_id in replica_set$elements(n_view - {my_id}) do
        send("commit", cur_viewid, cur_view, t_space, tbl) to rr
        end % for
    status := active
    end view_manager
```

Figure 5.7: View Manager

temporary copy of the tuple space. It records the most recent copy of the tuple space the manager has seen. *T_ts* is initialized to the view manager's local tuple space. *T_vid* keeps a copy of the viewid corresponding to *t_ts*. *T_tbl* keeps the most up-to-date timestamp-mid table. *N_view* is a temporary replica set containing the ids of the replicas that have accepted the view change invitation. The globally unique viewid is created by pairing the sequence number that is the successor of the largest sequence number in a viewid seen so far and its replica id.

To manage a view change, the manager first sends the invitation to all the replicas in *orig_config* excluding itself and then waits for responses. There are two possible types of messages to be received — *accept* messages (sent by the accepting replicas) and *invite* messages (sent by the replicas that start new view changes).

When an *accept* message is received, the manager checks if the acceptance is to the invitation just sent. If not, the message is ignored. Otherwise, *n_view* is updated to include the id of the accepting replica, and *t_ts*, *t_vid*, and *t_tbl* are updated if necessary.

When an *invite* message is received, the invitation is accepted only if the view the replica is invited to join has a higher viewid than any it has seen so far. By accepting the invitation, the manager abandons the current view change in progress and changes its state to the "underling" to participate in the new view change.

The receiving loop can be exited in two ways: either all the replicas in *orig_config* have accepted the invitation or δ_2 times out. δ_2 is set up so that it is sufficient for a normal round-trip of inviting and accepting messages to be transmitted.

In order to form a new view, there must be a majority of replicas accepting the invitation. (Recall that function *ismaj?(s1, s2)* checks if replica set $s1$ contains a majority members of $s2$.) If this is not true, the current view change is abandoned, and the manager will attempt to form another new view. Otherwise, the manager's current state (*cur_view*, *cur_viewid*, *t_space*, and *tbl*) is updated, and a commit message is sent to all the accepting replicas along with the new view, viewid, tuple space, and timestamp-mid table copy. Upon completion, the manager enters the "active" state.

```
underling = proc()
    t3: int := current_time + δ3
    while true do
        receive until t3
            commit(vid: view_id, n_view: view, tsp: tuple_space, ta: table):
                if vid = max_viewid then
                    cur_view := n_view
                    cur_viewid := max_viewid
                    t_space := tsp
                    tbl := ta
                    break
                    end % if
            invite(vid: viewid, r: replica_id):
                if vid <= max_viewid then continue end % if
                max_viewid := vid
                send("accept", my_id, vid, cur_viewid, t_space) to r
                return
            end % receive
                except when timeout:
                    status := view_manager
                    return
                    end % except
        end % while
    status := active
    end underling
```

Figure 5.8: Underling

5.6 Underlings

Figure 5.8 shows the code executed by the main process in the "underling" state. A replica becomes an "underling" if it accepts an invitation to join a new view. While it is in the "underling" state, it expects to receive a commit message from the manager. It is also possible to receive an invitation to join a new view.

The time interval δ_3 in the receive statement is set in such a way that is sufficiently long to allow the acceptance massage to go from the underling to the manager and the commit message to go from the manager to the underling in the normal situation. If the timeout expires, the underling starts a new view change by switching to the "view_manager" state.

When a commit message is received, the underling checks if the commit request is for the view it has agreed to join. If not, the commit message is ignored. Otherwise, the underling uses the information piggybacked on the commit message to update its local state and switch to the "active" state.

If an invitation for a higher viewid is received, the underling accepts the invitation, ceases its involvement in the current view change, and stays in the "underling" state to wait for the new commit message. Otherwise, the invitation is ignored.

5.7 Examples

This section gives an example to illustrate that the view change algorithm is robust in both the simple case where there is only one view manager coordinating a view change and the case when multiple view managers compete to form new views.

5.7.1 Simple Case

Let us suppose that we have five replicas in the original configuration, r_1, r_2, r_3, r_4, and r_5. At some point, the view contains all five replicas that are in the "active" state. Then a failure occurs, which makes r_1 inaccessible from the other replicas. We assume for simplicity that following the initial failure, no additional failures occur during the view change; once r_1 becomes inaccessible, it remains inaccessible for the duration of the algorithm.

At the point of failure, all five replicas have the same viewid v_1, $<1, r_1>$, identifying view $\{r_1, r_2, r_3, r_4, r_5\}$. When r_1 becomes inaccessible, the other replicas stop hearing from it. We suppose that r_3 detects this change and starts the view change. (More than one replica may detect this change and trigger the algorithm; this is the topic of the next subsection). R_3 becomes the view manager and enters the first phase of the algorithm. It computes a new viewid $<2, r_3>$, which is higher than anything r_3 has seen. Next, it sends the invitation message containing the new viewid to other replicas in the original configuration and waits for responses.

Each of r_2, r_4, and r_5 receives the invitation message and sends back an acceptance message containing, among other things, its current viewid, a copy of its local tuple space

and the timestamp-mid table. No reply is forthcoming from r_1 since it is inaccessible. R_3 collects the responses and keeps the most up-to-date state it has seen. (In this case, the tuple space of r_2, r_3, r_4 and r_5 are all equally up-to-date, so r_3's tuple space will be used.)

In the later half phase, r_3 forms a new view containing r_2, r_3, r_4 and r_5. This is possible because the new view has a majority of replicas. After updating its own local state, r_3 sends a commit message containing the new viewid ($<2, r_3>$), the new view ($\{r_2, r_3, r_4, r_5\}$), and an up-to-date copy of the tuple space and the table to r_2, r_4 and r_5, and becomes "active" to accept operation requests, send and receive probes, and monitors new view changes. When r_2, r_4, and r_5 receive the commit message, they update their local state and switch to the "active" status.

In the meantime, while all this is going on r_1 is also running the algorithm and is trying to form a view. As the view manager, it computes the new viewid and sends invitation messages to the other replicas. No responses are forthcoming due to the communication failure. It waits in vain for acceptances and eventually times out, remaining in the "view_manager" state.

In this scenario, the algorithm forms a new view excluding inaccessible replicas. The algorithm works similarly in the case of including replicas that become accessible when a failure is repaired.

5.7.2 Concurrent View Managers

If, in the above scenario, more than one replica detects a change in the communication capability, several replicas may become view managers simultaneously. Our view managerment algorithm handles this case of multiple concurrent view managers in the following way.

The viewids generated by different replicas are distinct, since we include the replica id as part of the viewid. In the previous example, let us imagine that r_1 through r_5 are labeled in increasing order. Suppose replicas r_2 and r_3 start up as view managers. R_2 computes $<2, r_2>$ and r_3 computes $<2, r_3>$. Both send invitation messages to everybody else in the configuration. The following events happen:

1. R_2 receives an invitation from r_3. Since $<2, r_3> > <2, r_2>$, r_2 accepts the invitation

and stops acting as a view manager.

2. R_3 receives an invitation from r_2. Since <2, r_2 > < <2, r_3 >, r_3 knows of a higher viewid, so it ignores the invitation from r_2.

3. R_4 and r_5 receive invitation messages from both r_2 and r_3. If they receive the invitation from r_2 first, they will accept the invitation and wait for the commit message from r_2. When they receive the invitation from r_3, they will stop participating in the previous view change and start participating in the view change initiated by r_3. On the other hand, if r_4 and r_5 receive the invitation from r_3 first, the later invitation, the one from r_2, will be ignored because it has a lower viewid.

Thus, no matter in what order the messages arrive the outcome is the same: r_3's new viewid is the one that prevails because its viewid is higher. This conclusion can be generalized to any number of concurrent managers.

5.8 Correctness

We claim that (1) the effects of the tuple space operations either survive into the new view (if the operations are completed at all the replicas in the old view) or will be retried in the new view (if the operations are not completed at all the replicas in the old view), and (2) the unrepeatable operations are executed at most once 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. If the operations are not completed at all the replicas in a view, the executing worker will be repeatedly trying until all the replicas in the current view have acknowledged the completion (this was explained in the last chapter). This is because if a view change takes place before an operation is completed at all the replicas in the old view, the new view may or may not contain any replica that is aware of the operation.

Repeated attempts to complete operations do not imply that the operations are executed more than once. Duplicate requests for the same operations are filtered out using the

timestamp-mid table at each replica, as described in the last chapter. Furthermore the timestamp-mid table is accurate since it is taken from the replica whose tuple space is used to initialized the state of the new view. This satisfies our second claim.

We are also interested in whether the algorithm makes progress, that is, whether it succeeds in forming new views as along as a sufficient number of replicas can communicate. Of course, it can only make progress provided that failures happen rarely, but this is a reasonable assumption. To increase the probability of a view change, the algorithm needs to be tolerant of slow responses and lost messages. For example, suppose a manager waits only until it hears from enough replicas to form a view even though there are other replicas that could respond. This would result in those other replicas being excluded from the new view, which in turn means another view change will occur shortly. If that next view change also excludes some potential members, that will lead to another view change, and so on.

To avoid such a situation, a manager should use a fairly long timeout while it waits to hear from all replicas that the "I'm Alive" messages indicate should reply. Similarly, an underling should use a fairly long timeout before it becomes a manager. In addition, it is worthwhile to mask lost messages by sending duplicates, so that a lost message will not trigger another view change.

5.9 Discussion

In concluding this chapter, we discuss a number of approaches to handling crashes and a possible optimization.

5.9.1 Crashes

In the above discussion, we assumed that after a node crash, a replica recovers with all its pre-crash state restored. That is, no information is lost during crashes. This subsection discusses two extant implementations, and gives a reference to a method that can be used when this assumption does not hold.

An easy solution is to provide stable storage [19] at each replica. Each replica has some form of nonvolatile storage (for example, disks). The updates to the replica state are

recorded on the log in the order they occur. The log is kept in the stable storage. During a recovery from a crash, the contents of the log are replayed in the order they were stored to restore the pre-crash replica state.

Although simple, this approach is usually undesirable. In order to make the storage truly stable, duplicate copies are needed. This makes the writing unacceptably slow. For example, if stable storage is implemented using two disks, both disks need to be written. Each update needs to be done sequentially: first it is written to one disk and then that disk must be read to ensure that the write happened successfully; then the same process must be repeated on the other disk.

An alternative to the above approach is to supply each replica with a disk and an uninterruptible power supply (UPS). Because of the UPS's, replicas can acknowledge operations as soon as the information resides in main memory. If the replica's node crashes, the UPS will permit it to write volatile memory to disk before it shuts down.

Oki has discussed a replication scheme that uses only a little nonvolatile or stable storage. Interested readers can refer to [22][23] for a discussion of this scheme.

5.9.2 Optimization

Our algorithm uses one-and-a-half-phases. During the first phase, the manager sends out invitations to all other replicas, and the underlings respond to the invitations. The underlings' current viewids and their tuple space copies are piggybacked on the responses. During the last half phase, the manager tries to form a view and, if one can be formed, sends a commit message along with the selected most up-to-date tuple space copy to all the underlings. No responses to the commit messages are necessary.

This scheme is simple, but costly in terms of the amount of information being transmitted and the amount of storage required if the tuple space is large. This is because the entire tuple space and table are sent on every underling's acceptance message to an invitation, and the manager has to keep a temporary copy of the tuple space and table in addition to its local copies. An alternative to the one-and-a-half-phase scheme is a two-phase scheme. During the first phase, the manager sends the invitation to all other replicas, and the un-

derlings respond by sending back their local viewids only. In the second phase, the manager informs the replica with the highest viewid to distribute its local copy of the tuple space and table to all the replicas in the new view. If the manager itself has the most recent viewid, this becomes a one-and-a-half-phase scheme.

Chapter 6

Discussion

In the previous chapters, we 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 protocol was simulated using Argus [20] on several VAXstations connected by a local area network. Deliberate failures were generated to simulate the possible failures in a general communications network. The simulation survived the various failures we were able to construct.

Our work contributes in the following two areas:

- The protocol makes it possible to implement a highly-available tuple space on a communications network where nodes may crash and recover, and the network may crash, partition, and be repaired. This establishes the foundation for building Linda systems on a communications network. Our research indicates how fault-tolerance might be achieved for other parallel systems. Many parallel computations are long lived; fault-tolerance is particularly important for them. In addition, the other advantages of distribution (using inexpensive machines over a network and scalability) apply to any parallel system.

- The protocol described in this thesis is an addition to the general replication schemes that provide fault-tolerant and highly-available services in distributed systems. It shows what can be done when the semantics of the operations are taken in account. We were able to devise an implementation that outperformed the general voting technique.

However, as discussed below, 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.

In the remainder of this chapter, we discuss the relationship between our technique and other related research, additional Linda operations, and some extensions to our method and areas for further work.

6.1 Related Work

The only other Linda kernels that approximate a distributed kernel are the S/Net kernel and the VAX-LAN kernel. We will discuss why they are inappropriate for use in a general communications network. We will also discuss two other replication approaches that can be alternatives to our scheme: the voting scheme and the viewstamped replication scheme.

6.1.1 S/Net Kernel

The S/Net kernel is described in detail in [8]. The S/Net consists of several MC-68000's with local memory, connected by a bus. The operations are executed as follows: executing **out**(t) causes tuple t to be broadcast to every node in the network; thus every node stores a complete copy of the tuple space. Executing **in**(s) triggers a local search for a matching t. If one is found, the local kernel attempts to delete t network-wide; if the attempt succeeds, t is returned to the worker that executed **in**(s). If the attempt fails, the deleted tuples are put back and the operation is tried again. An attempt can fail for two reasons: (1) some other worker has simultaneously attempted to delete t and has succeeded on some nodes; and (2) some other worker is executing a concurrent **out** operation and t has not yet reached all the nodes. If the local search triggered by **in**(s) turns up no matching tuple, all newly-arriving tuples are checked until a match occurs, at which point the matching tuple is deleted and returned as before. **Rd** works in the same way as **in**, except that no tuple deletion is attempted; as soon as a matching tuple is found, it is returned immediately to the reading worker.

The S/Net kernel assumes reliable broadcast of messages, so it does not tolerate failures

of network or nodes. For instance, if an **out**'s 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. In addition, the S/Net requires that a copy of tuple space be stored on every node while our scheme does not.

Our protocol performs[1] as well as the S/Net kernel's protocol on **rd**, **out**, and **in** operations (assuming that the S/Net executes **out** operations in the background). 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 to avoid blocking the program process.

6.1.2 VAX-LAN Kernel

In a VAX-LAN [4], computing nodes are connected by an Ethernet-based local area network. The VAX-LAN kernel uses the following scheme: **out**(t) stores t on one of the nodes; **in**(s) activates a global search for a match to s on all nodes; **rd**(s) also requires a global search. In this scheme, **out** is simple. **In**(s) causes the template s to be broadcast to all nodes. Each node searches for matching tuples in its local memory. If a matching tuple is found, it is deleted from the local memory and shipped to the template-originating node using a point-to-point protocol; otherwise the template is stored locally for x ticks. All the tuples arriving within these x ticks are checked, and matching ones are sent off. The template is thrown away after x ticks. If the template's originating node has not received any tuple for x ticks, then it broadcasts the template again. If the originating node receives more than one matching tuple, one of them is chosen, and the rest are stored on some nodes. **Rd**(s) is similar.

Since only one copy of each tuple is stored system-wide, the VAX-LAN scheme does not provide high-availability: if the node owning the tuple t crashes, or a message containing t in response to an **in**(s) is lost, then t becomes unavailable or, worse, is lost forever. A network partition may also make some tuples unavailable.

[1] Our analysis of performance is based on the amount of messages and delays at protocol level. We are not able to make comparisons on any real implementation at this writing.

Our scheme performs as well as the VAX-LAN kernel protocol on **rd** and **out** operations. The VAX-LAN performs better on **in** operations since the program process can continue as soon as the first response arrives. But the better performance on **in** operations comes from the fact that only one copy of each tuple is stored — the reason that VAX-LAN can not be made highly-available.

6.1.3 Voting

Gifford's Weighted Voting [14] 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 and a write operation has to acquire a *write quorum* of w votes. The requirement that $r + w > n$ and $2w > n$ 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. The up-to-date copy is identified by the copy's *version number*. In addition to the version number, each copy also contains its state and the number of votes assigned to it. Herlihy [16] extended the above voting scheme to take the advantage of operation semantics, and thus made the algorithm more efficient.

Our protocol is a special case of the voting scheme where the read quorum is one and the write quorum is all the replicas. 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 write operations need to be two-phase.

In voting schemes where writes are done to all copies, write operations cannot be performed if a replica is down or inaccessible. This problem was overcome by the invention of the virtual partition protocol [1][2]. Our view change algorithm is an optimization of this protocol.

6.1.4 Viewstamped Replication

The viewstamped replication scheme is described in [23][22]. It is an integration of a modified primary copy scheme [5] and the virtual partition algorithm [1]. This method works as follows. The tuple space is replicated. Among the replicas, there is a *primary* that executes workers' operation requests. The updates are propagated to the rest of the replicas, called *backups*, in background mode. Whenever a failure is detected, the replicas activate a view change algorithm similar to ours. A new view is formed if a majority of replicas agree to join the new view. A new primary is elected when the new view is formed.

To identify the latest state of the new view, *viewstamps* are used. A viewstamp is the concatenation of the viewid of the view in which the operation is executed and the timestamp of the operation. The viewstamps help the view manager to identify the replica that has the most up-to-date state.

The viewstamped replication scheme is efficient (the workers only have to talk to one replica in the normal case), fault-tolerant (it tolerates common failures from general partitionable networks), and highly-available (the data are replicated). But the current scheme is defined to work only when workers' computations run as atomic transactions [19]. How to adopt the viewstamp replication scheme to Linda is a matter for future research.

6.2 Additional Linda Operations

As mentioned in Chapter 2, additional operations have been proposed for Linda. 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 are not compatible with our implementation. Our current scheme allows a **rd** to observe the results of a partially completed **out** (that is, an **out** that has been completed at only some of the replicas in the current view). The following example illustrates the difficulty:

worker w	worker z
out("x", 3)	
	rd("x", **formal** u) % $u = 3$
	rdp("x", **formal** v) % *signals an exception*

When worker z 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 z occurs and observes that the **out** has not yet occurred. Note that this problem will not occur if z'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 only return a tuple 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.

The S/Net kernel also does not support these operations. The problem here comes up in the interaction of **in** with **rdp**:

worker w	worker z	
in("x", **formal** v)	**rdp**("x", **formal** v)	% *signals*
	rdp("x", **formal** u)	% *returns 3*

Here w's **in** is running in parallel with z's **rdp**'s. Suppose that ("x", 3) is in the tuple space before w's **in**. The first **rdp** observes the situation when w is attempting to remove the tuple, and this tuple has been removed at z's node. However, suppose the **in** fails and the tuple is put back. In this case the second **rdp** observes the result of putting the tuple back.

The VAX-LAN kernel could support **rdp** and **inp**, but, as mentioned earlier, this approach cannot be made highly-available.

6.3 Extensions of Our Scheme

In describing our system model in chapter 3, we assumed that replication is uniform— every tuple is replicated onto all replicas. In this section, we will show that our protocol works even when the tuple space is not replicated uniformly. We will also describe a proposal for tolerating workers' failures.

6.3.1 Nonuniform Replication

There are two problems with keeping the entire tuple space at one set of replicas. First, if the tuple space is large, then each node where a replica resides must provide a large amount of storage. Second, 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 set of replicas operates completely independently from the other sets. Each set contains its own replicas. For example there might be two sets:

$S = \{r_1,...,r_5\}$
$T = \{r_6,...,r_{10}\}$

Some of these replicas might reside at the same node, for example, r_1 and r_7 might both be at node N, but more likely the nodes containing the replicas would be disjoint. The reason for this is that replica sets are useful, as mentioned above, for alleviating storage problems at nodes and for reducing contention. These benefits would not be obtained if replicas in different sets were located at the same node.

When a worker performs an operation, it sends the request to the replica set that contains information about that tuple or template. Obviously, there must be a mechanism to determine what set to use. This could be done either 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 [12][15][17][21].

The portion of a worker's code that interacts with replicas would need to take the multiple sets into account. Operations concerning the same set would be done in order just as described in Chapter 3. Operations that make use of different sets can be done in the background in parallel, except that we still need the same synchronization we have now, namely that prior **in2**'s must complete before an **out** can start. For example, suppose logical tuple "x" is stored at set S and logical tuple "y" is stored at set T. Consider first

in("x", ...)
in("y", ...)
rd("x", ...)
rd("y", ...)

The **rd**'s of "x" and "y" will not observe the old tuples removed by the respective **in**'s because operations are done in order at each set. Thus at S we do **in**("x") before the **rd**("x"), and at T we do **in**("y") before the **rd**("y"). Now consider

in("x", ...)
in("y", ...)
out("x", ...)

The start of the **out** will be delayed until both **in**("x") and **in**("y") are completed. This will ensure that some other worker that observes the effect of the **out** will not subsequently be able to observe the tuples removed by either **in**("x") or **in**("y").

View changes occur independently at each set, using the protocol described in Chapter 5.

6.3.2 Workers' Failures

This thesis proposed a scheme to build a fault-tolerant kernel that makes the Linda tuple space highly-available. But even with such a kernel, Linda programs are not completely

fault-tolerant since the failure of a worker can cause problems. In particular, if a worker crashes after starting an **in1**, but before completing the corresponding **in2**, some tuples may be locked forever. This section proposes a way to tolerate workers' failures.

What we would like is to release locks held by crashed workers. However, as mentioned earlier, it is not possible in general to distinguish a node crash from a partition. Thus, the absence of messages from a worker may mean either that it is crashed, or it cannot communicate because of 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* as discussed earlier in the thesis, and also a *worker-view*. Initially all workers are in the worker-view. Replicas send probe messages to workers and workers respond to these messages. If a worker does not respond to probes after a sufficient number of tries, the replicas carry out a *worker view change*, during which all replicas in the current replica-view agree on a new worker-view and worker-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. As is the case in any view change, a majority of replicas must participate in the view change.

Whenever a replica receives an operations 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 immediately.

Given this semantics, fault-tolerant programs can be written in Linda. Figure 6.1 shows the form of such a program. The idea here is that the workers collaborate to carry out *task-numbers* of tasks; information about these tasks is contained in the *tasks* array in the tuple space. To keep track of what workers are doing, we use the *status* array in tuple space. *Status*$[i] = 0$ means that task i has not yet been worked on; *status*$[i] < 0$ means that task i has been completed; *status*$[i] > 0$ means that task i is being worked on. In this latter case the value of *status*$[i]$ tells how many times workers have attempted to perform task i.

```
cnt: array[record[round, time: int]] % a local array at each worker
                                     % initially cnt[i] = <0, 0> for all i
while true do
    done: bool := true
    for i in task-numbers do
        in("status", i, formal v)
        if v < 0 then
            out("status", i, v)
            continue % to the next iteration of the for loop
            elseif v = 0 or (cnt[i].round = v and cnt[i].time < current_time) then
                out("status", i, v+1)
                % do tasks[i] here ...
                in("status", i, formal v)
                out("status", i, −1)
                continue
            elseif cnt[i].round ~= v then cnt[i] := <v, current_time + δ>
            end % if
            done := false
        end % for loop
        if done then
            return % only get here when status[i] < 0 for all i
            end % if
    end % while
```

Figure 6.1: A Fault Tolerant Worker

A worker cycles through the *status* array looking for a task to be done. Such a task is either one that has never been attempted, or one that has been attempted by another worker in the past but not completed within a reasonable delay δ. When it first discovers a task being worked on by another worker, it records this fact in its local *cnt* array, together with an estimation of when that worker should complete. If it later discovers that the task is still being worked on by that worker, but the time estimate has been exceeded, it takes on the task itself. In this case it stores a larger round number in the *status* array to prevent other workers from also redoing the task at this point. (The estimated time of completion could be stored in the *status* array provided we assume that the clocks of the workers are loosely synchronized. If workers do not have clocks, they can keep track of how many times they have noticed that a particular round for task i is occurring, and take on the task themselves when this number reaches some maximum.)

Note that there is an assumption here: it is all right to do a computation more than once. This is sometime undesirable. If the computation is not repeatable, additional techniques are needed that allow computations to run as atomic actions [11]. Adding atomic actions to Linda requires further research.

References

[1] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An Efficient, Fault-Tolerant Protocal for Replicated Data Management. In *Proceedings of the 4th ACM SIGACT/SIGMOD Conference on Principles of Database Systems*, ACM, 1985.

[2] Amr El Abbadi and Sam Toueg. Maintaining Availability in Partitioned Replicated Databases. In *Proceedings of the 5th ACM SIGACT/SIGMOD Conference on Principles of Data Base Systems*, ACM, March 1986.

[3] S. R. Ahuja, N. J. Carriero, D. Gelernter, and V. Krishnaswamy. Progress Towards a Linda Machine. In *Proceedings of International Conference on Computer Design*, IEEE, 1986.

[4] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and Friends. *IEEE Computer*, 26–34, August 1986.

[5] Peter A. Alsberg, Geneva H. Belford, John D. Day, and Enrique Grapa. *Multi-Copy Resiliency Techniques.* CAC Document 202, Center for Advanced Computation, University of Illinois, Urbana, Illinois, May 1986.

[6] Philip A. Bernstein and Nathan Goodman. The Failure and Recovery Problem for Replicated Databases. In *Second ACM Symposium on the Principles of Distributed Computing*, pages 114–122, August 1983.

[7] Robert Bjornson, Nicholas Carriero, David Gelernter, and Jerrold Leichter. *Linda, the Portable Parallel.* Technical Report, Department of Computer Science, Yale University, February 1987.

[8] Nicholas Carriero and David Gelernter. The S/Net's Linda Kernel. *ACM Transaction on Computer Systems*, 4(2):111–129, May 1986.

[9] Nicholas J. Carriero. *Implementation of Tuple Space Machines.* YALEU/DCS/TR 567, Yale University, Department of Computer Science, December 1987.

[10] F. Cristian, H. Aghili, R. Strong, and D. Dolev. *Fault-Tolerant Atomic Broadcasts: from Simple Message Diffusion to Byzantine Agreement.* Tech. Report, IBM Research San Jose, 1984.

[11] Kapal P. Eswaran, James N. Gray, Raymond A. Lorie, and Irving L. Traiger. The Notion of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[12] Robert J. Fowler. *Decentralized Object Finding Using Forwarding Addresses.* Technical Report 85-12-1, University of Washington, Department of Computer Science, Seattle, Washington, December 1985.

[13] David Gelernter, Nicholas Carriero, Sarat Chandran, and Silva Chang. Parallel Programming in Linda. In *Proceedings of International Conference on Parallel Processing*, IEEE, 1985.

[14] David K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, ACM, December 1979.

[15] Cecilia Henderson. "Locating Migratory Objects in an Internet". 1982. Master's thesis, MIT Laboratory for Computer Science. Available as Computation Structures Group Memo 224, MIT LCS.

[16] Maurice Peter Herlihy. *Replication Methods for Abstract Data Types.* Technical Report 319, Laboratory For Computer Science, Massachusetts Institute of Technology, May 1984.

[17] Deborah J. Hwang. *Constructing a Highly-Available Location Service for a Distributed Environment.* Master's thesis, M.I.T. Laboratory for Computer Science, November 1987. Master's thesis.

[18] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[19] B. W. Lampson. *Atomic Transactions*, pages 246–265. Volume 105 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, N.Y., 1981. This is a revised version of Lampson and Sturgis's unpublished *Crash Recovery in a Distributed Data Storage System.*

[20] Barbara Liskov. *Overview of the Argus Language and System.* Programming Methodology Group Memo 40, Laboratory For Computer Science, Massachusetts Institute of Technology, February 1984.

[21] Sape Mullender and Paul Vitanyi. "Distributed Match-Making for Processes in Computer Networks—Preliminary Version". In *Proceedings of the Fourth ACM Symposium on the Principles of Distributed Computing*, August 1985. Held at Minaki, Ontario, Canada. Sponsored by ACM.

[22] Brian M. Oki. *Viewstamped Replication for Highly Available Distributed Systems.* PhD thesis, Massachusetts Institute of Technology, May 1988.

[23] Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A general Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, ACM, August 1988.

[24] Fred B. Schneider. Fail-Stop Processors. In *Digest of Papers from Spring CompCon '83 26th IEEE Computer Society International Conference*, pages 66–70, IEEE, March 1983.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-424	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) A Fault-Tolerant Network Kernel for Linda	
12. PERSONAL AUTHOR(S) Xu, Andrew S.	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 15. PAGE COUNT
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Fault-tolerant, highly-available, replication, distributed systems, parallel computing, view change

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The parallel programming system Linda consists of a number of processes and a shared memory called the tuple space. In a distributed implementation of Linda, processes and the tuple space reside on different computing nodes connected by a communications network subject to a variety of node and network failures. This thesis develops a scheme to make the tuple space highly-available in the presence of failures.

High-availability is achieved by replication: the tuple space is replicated on several modes so that failures usually do not disrupt program execution. Our replication method has two parts: the operations protocol and the view change algorithm. The operations protocol is a read-one-write-all scheme, that is, values are read from one of the replicas and write operations are executed at all replicas. The protocol exploits the semantics of the tuple space operations to eliminate unnecessary delay in program execution. When failures occur, the replicas are reorganized and their states are updated. This process is called a view change and is accomplished by the view change algorithm. A (continued..)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

GPO U.S. Government Printing Office: 1985—507-047

19. view change guarantees that a newly formed view consists of a majority of the replicas, and that all updates survive into the new view. Together, the operations protocol and the view change algorithm ensure that operations are executed in the correct order, updates to the tuple space survive failures, and processes only see the correct tuple space state in spite of failures. In addition, operations are performed by a concurrent background process whenever possible.