1993

Quantitative and qualitative evaluation of Linda in a distributed environment

Harish Vedavyasa
The University of Montana

Quantitative and Qualitative Evaluation of Linda in a Distributed Environment

by

Harish Vedavyasa

Bachelor of Technology

Karnataka Regional Engineering College, India 1991

Presented in partial fulfillment of the requirements

for the degree of

Master of Science

University of Montana

1993

[illegible]

Chairman, Board of Examiners

[illegible]

Dean, Graduate School

May 13, 1993

Date

6-21-93

Vedavyasa, Harish, M. S., April 1993 Computer Science

Quantitative and Qualitative Evaluation of Linda in a Distributed Environment.

Director: Dr. Ray Ford [illegible]

Distributed computing and database management are of paramount importance in a network environment. Distributed programming tools facilitate design and implementation of such systems. Design and implementation are two distinct phases while realizing a distributed system. In the design phase, one looks at the protocols that are required to build a system. In the implementation phase, one tries to map these protocols onto the distribut ed programming tool available. The design of a robust Object Registration System is considered here. Before this design is implemented in C-Linda, a parallel language, quantitative and qualitative evaluation of C-Linda is done in order to find out the features and limitations of this programming tool. The results of this evaluation is then used to implement a prototype Object Registration System.

Acknowledgement

I am deeply indebted to Dr. Ray Ford. Dr. Youlu Zheng . Dr. Roly Redmond , and Dr. Ramakrishna Nemani . without whose guidance and help this thesis would not have been possible. In particular. I am very grateful to Dr. Ray Ford for his valuable advice and inspirative ideas that resulted in outstanding results. My special thanks to Dr. Youlu Zheng for providing me with valuable literature on UNIX Signals which was of valuable help during system implementation.

Finally, I would like to thank the Department of Computer Science. University of Montana. for the computing resources and relevant literature that made this thesis possible.

Date : April 14 1993 Harish Vedavyasa

CONTENTS

Chapter 1

Introduction

Distributed computing and database management are of paramount importance in a network environment. The former allows faster computation because it supports parallelism and the latter allows efficient data manipulation and storage.

Distributed programming tools facilitate design and implementation of such systems. Design and implementation are two distinct phases while realizing a distributed system. In the design phase, one looks at the protocols that are required to build a system. In the implementation phase, one tries to map these protocols onto the distributed programming tool available.

Before building any distributed system, it is better, and sometimes necessary, to evaluate the programming tool available. Evaluation has two facets.

- A quantitative evaluation involves designing benchmarks to measure the performance of the programming tool. This later helps in designing the system with optimum configuration.
- A qualitative evaluation involves discovering capabilities of the given programming tool. Some designs lead to straightforward implementation, some can be implemented with certain difficulty, and still others cannot be implemented with the given programming tool.

1.1 Problem Statement

In a network environment, several users may have to share a large database of objects. In order to reduce object redundancy and eliminate object inconsistency, there is a demand for a distributed object support system that meets all user requirements. This situation demands an Object Registration System that allows users to register object definitions and instances, and to access objects registered by other users on the network. The design and implementation of an Object Registration System can be stated more formally as follows:

Design and implement a network-based Object Registration System (ORS) that allows users to access and modify a distributed object database. The database is a collection of object definitions, instances, and methods. The following are the requirements.

- ORS must allow normal startup and shutdown.
- ORS should be robust. It should be able to recover from network and/or processor failures.
- ORS must be as efficient as possible. (But efficiency can be sacrificed for greater reliability).

Given these requirements, the next problem is to design reliable and robust high level distribution system protocols to support Object Registration System functions. This design does not assume anything about the underlying implementation details. Subsequently, the problem is to implement the high level design and distribution protocols with a specific programming tool. The tool that is being experimented with is C-Linda, a combination of the coordinating language Linda with the programming language C.

Chapter 2

Design

The object registration system uses an object-oriented paradigm to create a basic data management capability. The set of data formats form the object classes. the data themselves form data object instances. and simulation components form object methods. The focus of the design is on developing the distribution protocols to meet the requirements stated earlier.

As illustrated in Figure 1, the distributed data management of ORS uses the paradigm of the Object Request Broker (ORB), patterned after that suggested by the Object Management Group [3]. The ORB provides system users transparent access to objects that may be physically located anywhere within the system. The ORS has a Object Manager (OM). which distributes ORB instances to every machine participating in the distributed system. and also monitors the ORBs' activity. Each ORB interacts with local users. other ORBs. and with the global OM. The ORB's communicate among themselves and the OM by passing messages. Thus ORB's and OM share an abstract global data space. Users add or delete data in this space through interaction with ORB. They can share information with other users through this common space.

The OM distributes ORB's such that each machine gets exactly one ORB instance. The machine on which OM executes is designated as the master and the rest are called workers. The master also has a copy of ORB executing on it. Figure 2 illustrates this point.

The ORS functionality primarily consists of three distinct phases:

- Startup : In this phase, the OM distributes the ORB's.
- Operation : In this phase, each ORB interacts with its local user(s) and responds to object requests. An object requested at processor P_i is resolved locally (i.e., by ORB_i on P_i) if the object is found in the data space associated with ORB_i. The object is resolved globally if communication is required between ORB_i and some remote ORB_j that has the required object in its data space. In both cases, communication required for object resolution (local or global) is transparent to the user (Figure 3).
- Termination : This is the concluding phase where ORB's and OM are terminated.

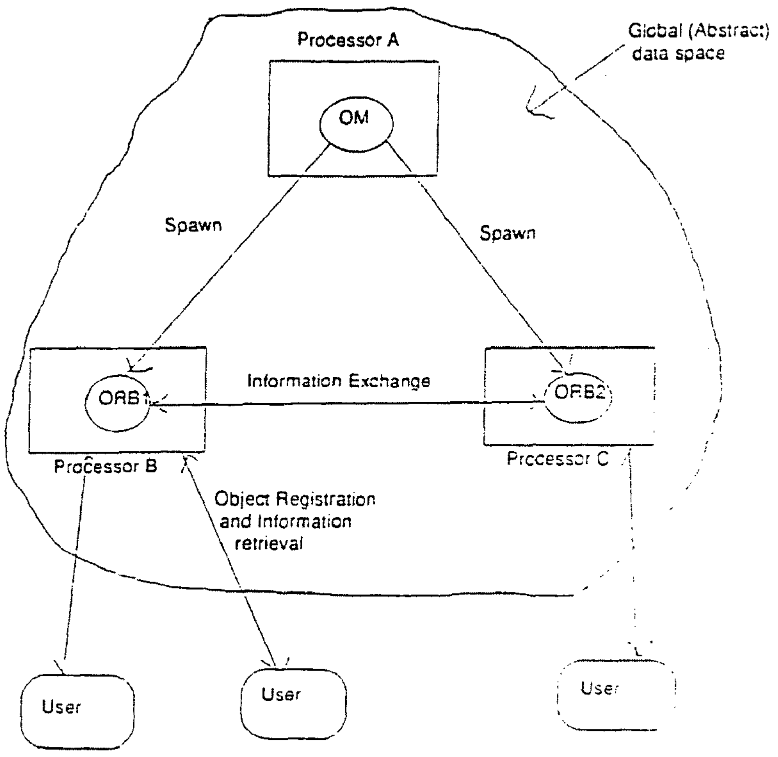


Figure 1: Relationship among OM, ORB, and Users

Figure 4 illustrates the above mentioned phases. (Appendix A describes the timing diagrams used in more detail)

2.1 Normal Startup, Operation, and Termination

At system startup, the OM spawns an instance ORB_i on each processor P_i listed in its processor table. The OM expects an acknowledgement from ORB_i immediately after it starts. Thus, each ORB_i must send an acknowledgement ack_i to OM as soon as it begins execution on processor P_i. If OM receives acknowledgements from all ORB instances spawned, normal startup is complete. Figure 5 illustrates normal startup.

Following normal startup, the system enters the operation phase. During this phase, in addition to ORB communication used to resolve object requests, the OM and ORB's periodically communicate to ensure proper functioning of the system as a whole. At a regular interval, referred to as system tick, each ORB_i sends an $alive_i$ message to OM, and OM sends an $alive_{om}$ message to one of the ORBs. Thus for N active ORB's, each ORB_i should receive an $alive_{om}$ message every N system ticks. Figure 6 shows the details of the

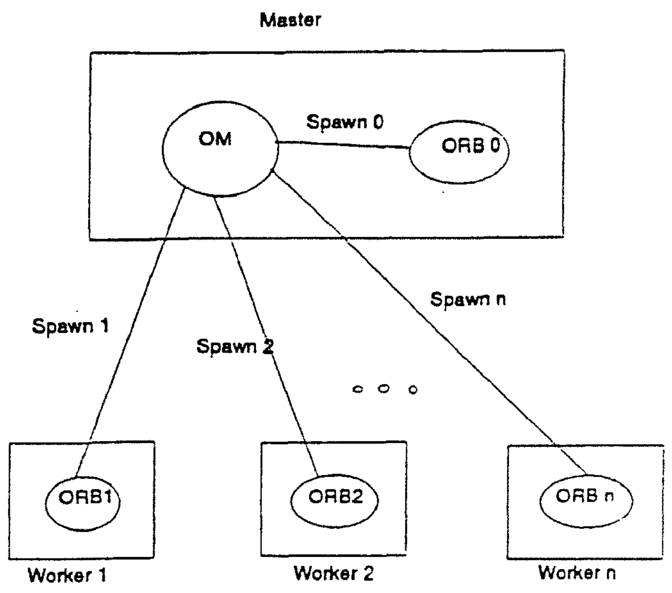


Figure 2: Master and Worker machines

normal operation.

Normal system termination is always initiated by OM. It sends a *terminate*$_i$ message to every ORB_i. The ORBs in turn respond with an acknowledgement ack_i and terminate themselves. When the OM receives N acknowledgements, it terminates ORS (Figure 7).

As illustrated in Figure 8, any individual ORB instance can initiate its own termination before overall system termination. The ORB_i sends a *terminate*$_i$ message to OM, and OM responds with an acknowledgement. ORB_i now terminates. After receiving the termination message, the OM considers ORB_i on processor P_i dead for all purposes. The OM will not try to spawn another ORB instance on processor P_i.

2.2 Abnormal Startup

The scenario described in the previous section seems very much desirable but completely ignores system or component faults. In a world full of uncertainties, network and processor failures cannot be ignored. Hence, the particular fault cases considered are

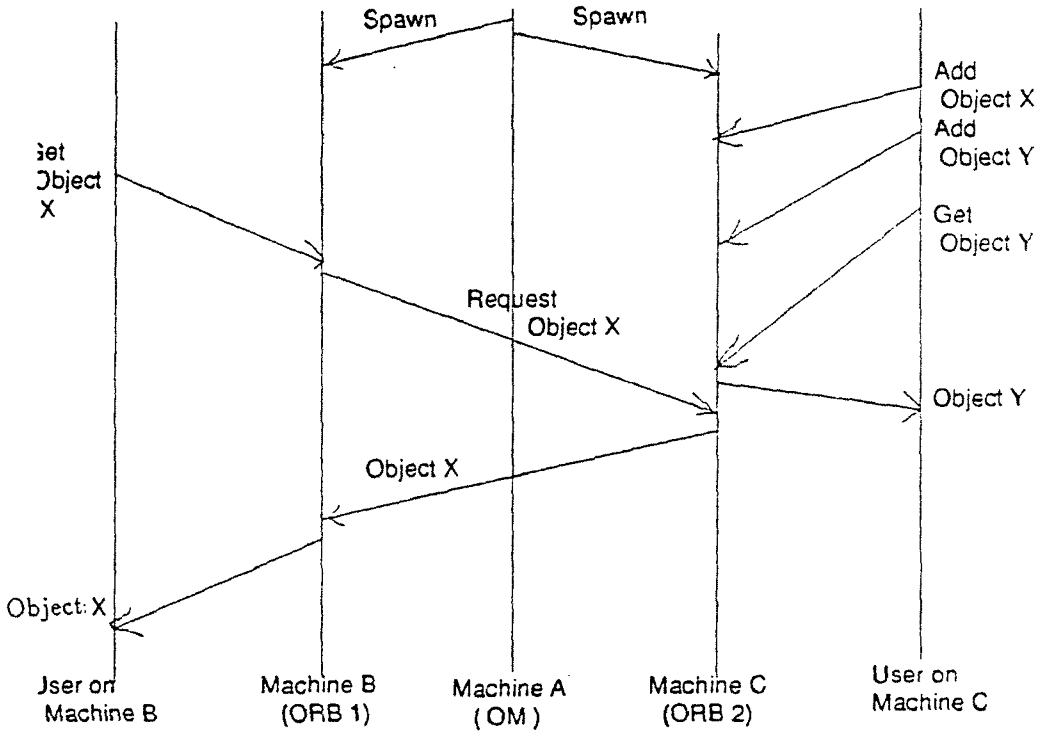


Figure 3: Local and Global resolution of Objects

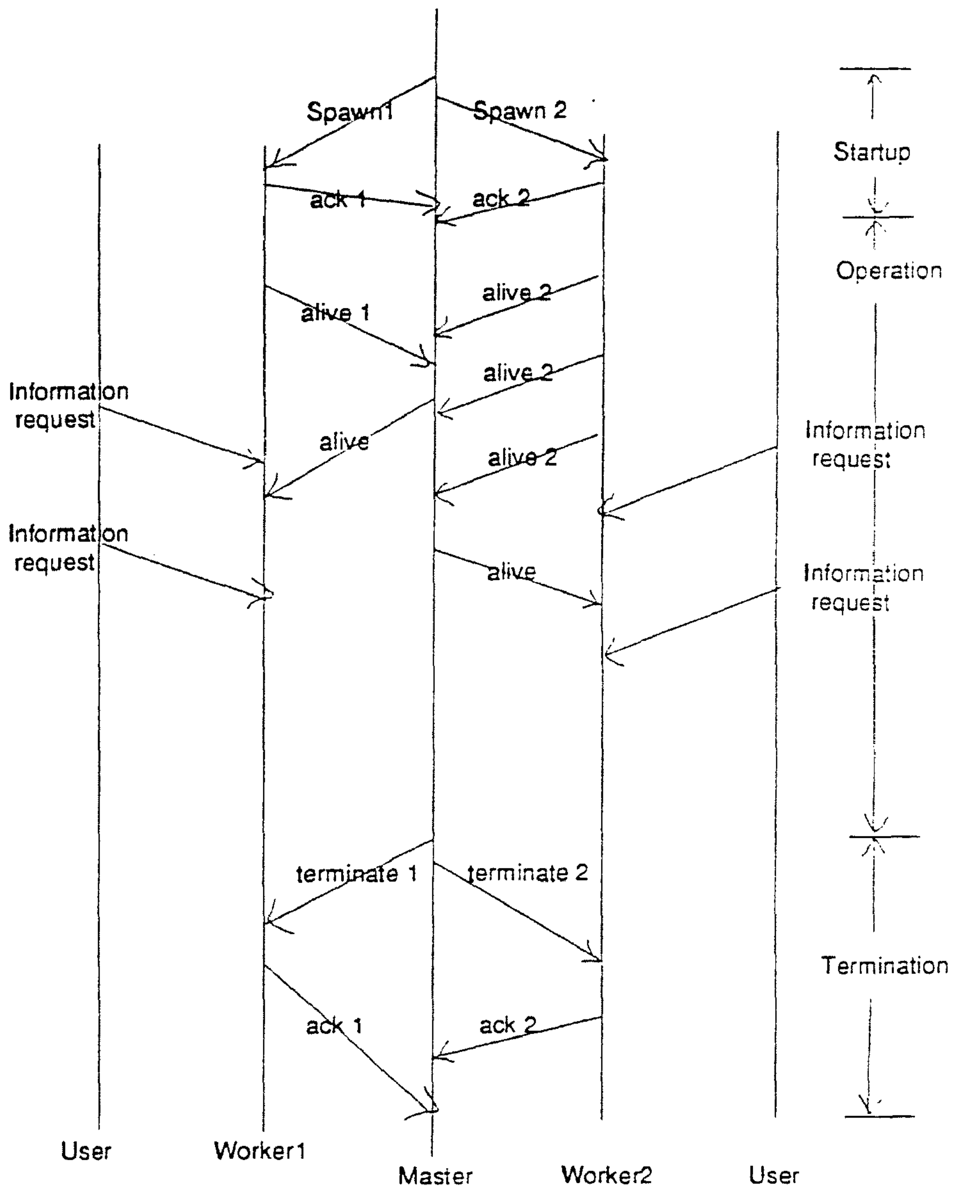
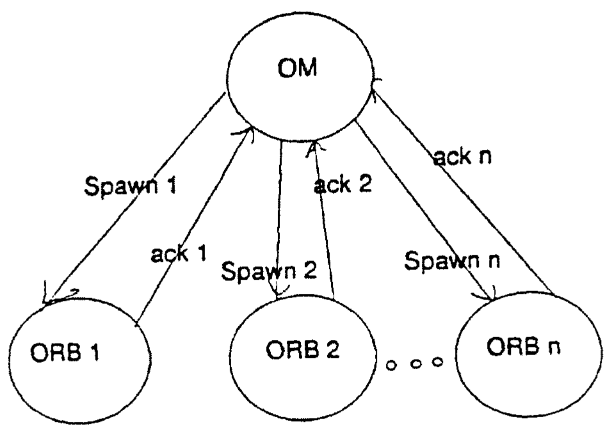


Figure 4: ORS Functionality : Startup, Operation, and Termination



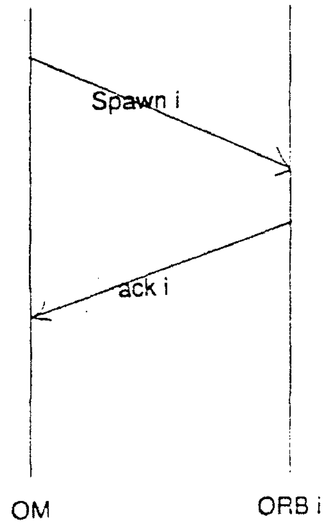


Figure 5: Normal Startup

- ORB failure during startup,
- ORB failure during operation, and
- OM failure during operation.

During the startup process, if OM fails to receive an acknowledgement from any of the ORB's within a specified *timeout* interval, the OM assumes that either the machine or the network connecting that machine is down. It is generally impossible to distinguish between these two. It can also be the case that the ORB process might have been accidently terminated. In order to distinguish between these possibilities, OM attempts to place another instance of ORB on that machine.

Let the k^{th} attempt by OM to spawn an ORB instance, $ORB_{i,k}$, on processor P_i be represented by message $sorb_{i,k}$. in response to receipt of $sorb_{i,k}$, P_i will spawn $ORB_{i,k}$ and generate the acknowledgement $ack_{i,k}$. Assuming that for processor P_i, OM has not received the acknowledgement $ack_{i,k}$ in response to $sorb_{i,k}$, the OM will subsequently attempt to spawn another instance, $ORB_{i,k+1}$ on processor P_i by sending $sorb_{i,k+1}$ (refer to Figure 9).

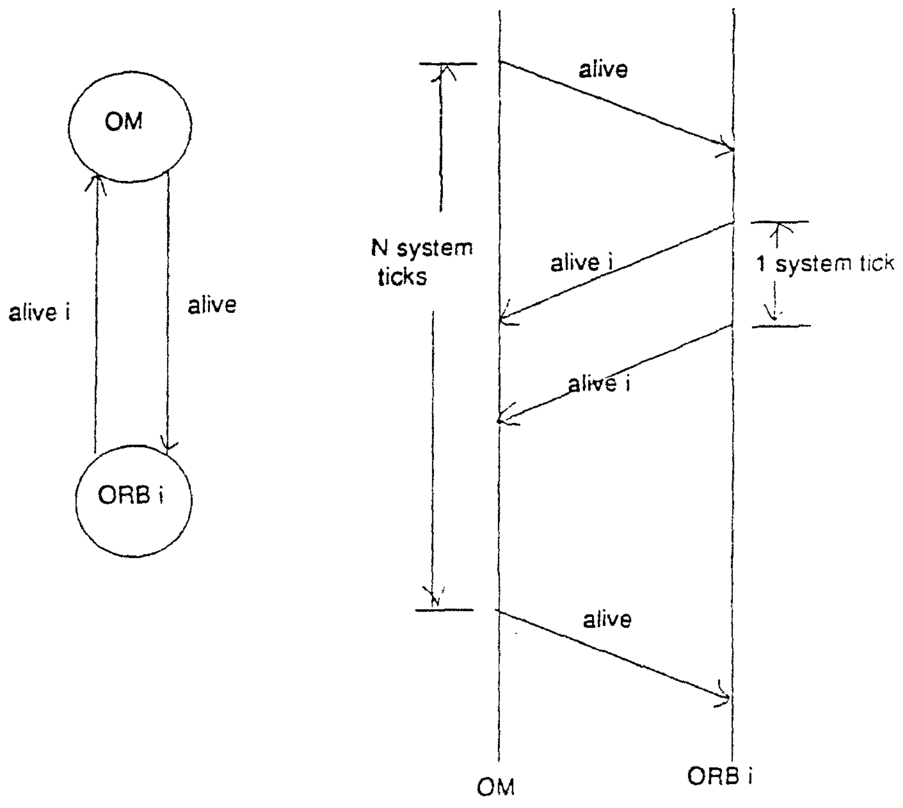


Figure 6: Normal Operation

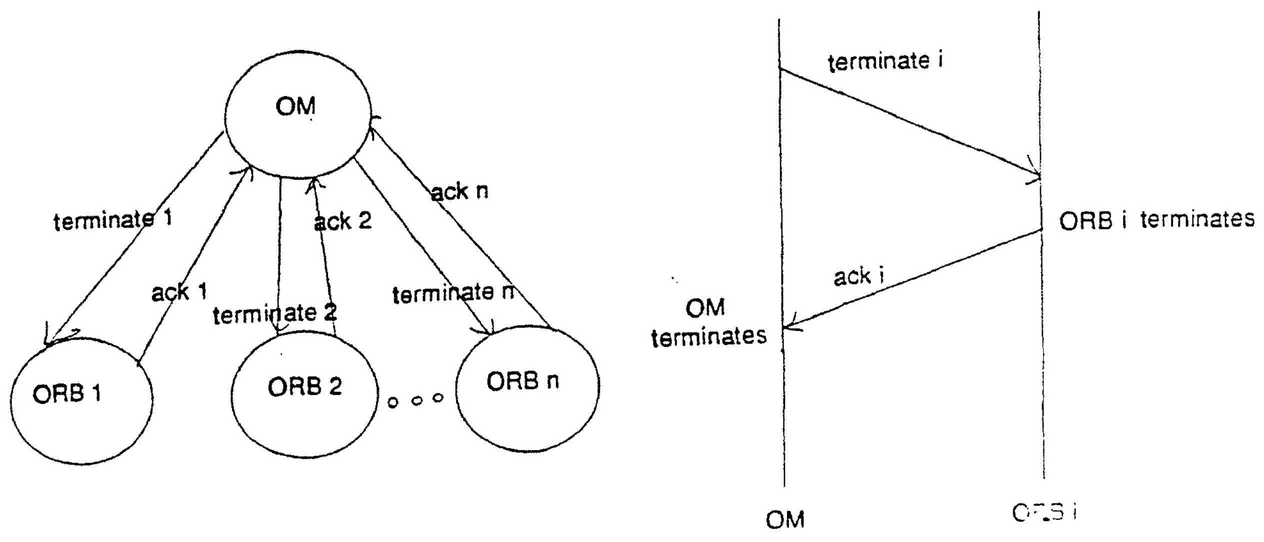


Figure 7: Normal System Termination

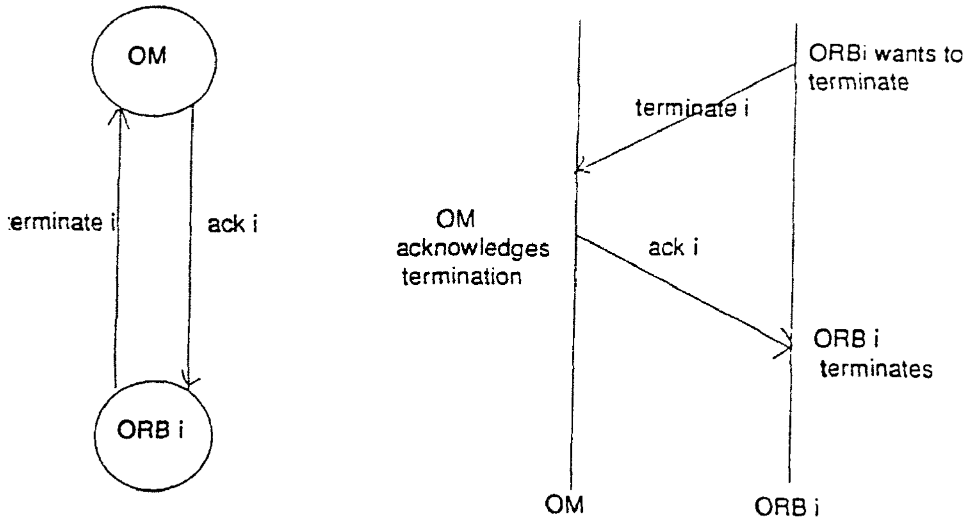


Figure 8: Normal ORB termination

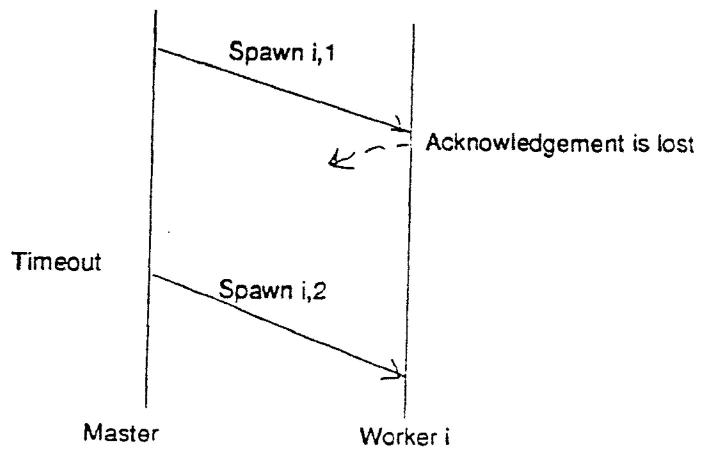


Figure 9: Abnormal Startup : Spawning another ORB instance

Several different scenarios arise at this point.

- As illustrated in Figure 10, if OM receives $ack_{i,k}$, the OM immediately sends a termination message, $terminate_{i,k+1}$, to P_i to terminate $ORB_{i,k+1}$.
- As illustrated in Figure 11, if OM receives $ack_{i,k+1}$, normal operation continues. If OM subsequently receives $ack_{i,k}$, the OM immediately sends $terminate_{i,k}$ to terminate $ORB_{i,k}$.
- If OM receives no acknowledgement from either of $ORB_{i,k}$ or $ORB_{i,k+1}$, it assumes that the network link connecting processor P_i is down or the processor P_i is dead. The OM does not attempt to spawn more ORB instances on processor P_i at this stage. However, it periodically tests access to P_i by spawning a simple $hello_i$ process. As illustrated in Figure 12, if successfully started on P_i, $hello_i$ sends a message back to OM and terminates. Upon receiving this message, the OM again enters the two stage cycle described above, attempting to start an ORB instance on P_i. If at any time the OM receives an acknowledgement from $ORB_{i,k}$ or $ORB_{i,k+1}$, which were assumed to be dead, functioning resumes normally and no further attempts to start a new ORB_i

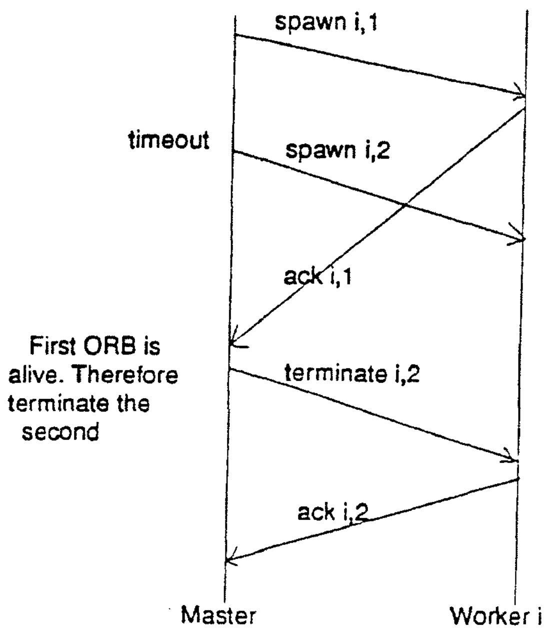


Figure 10: Abnormal Startup : Terminating recently spawned ORB instance

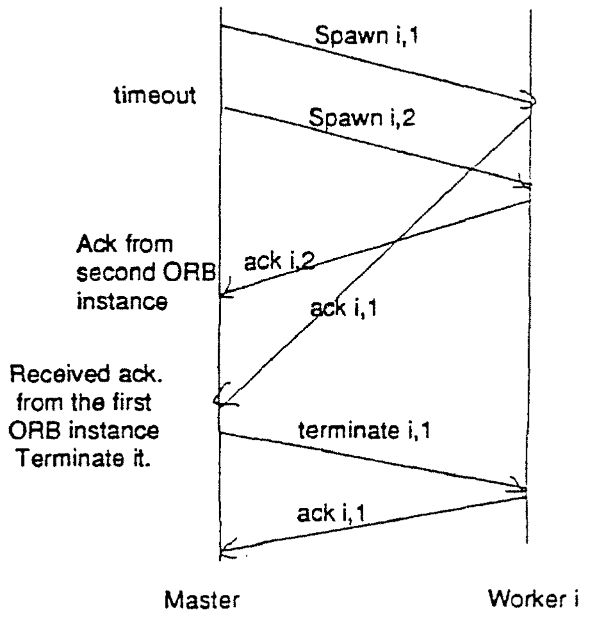

Spawn i,1
timeout
Spawn i,2
Ack from second ORB instance
ack i,2
ack i,1
Received ack. from the first ORB instance Terminate it.
terminate i,1
ack i,1
Master
Worker i

Figure 11: Abnormal Startup : Terminating previously spawned ORB Instance

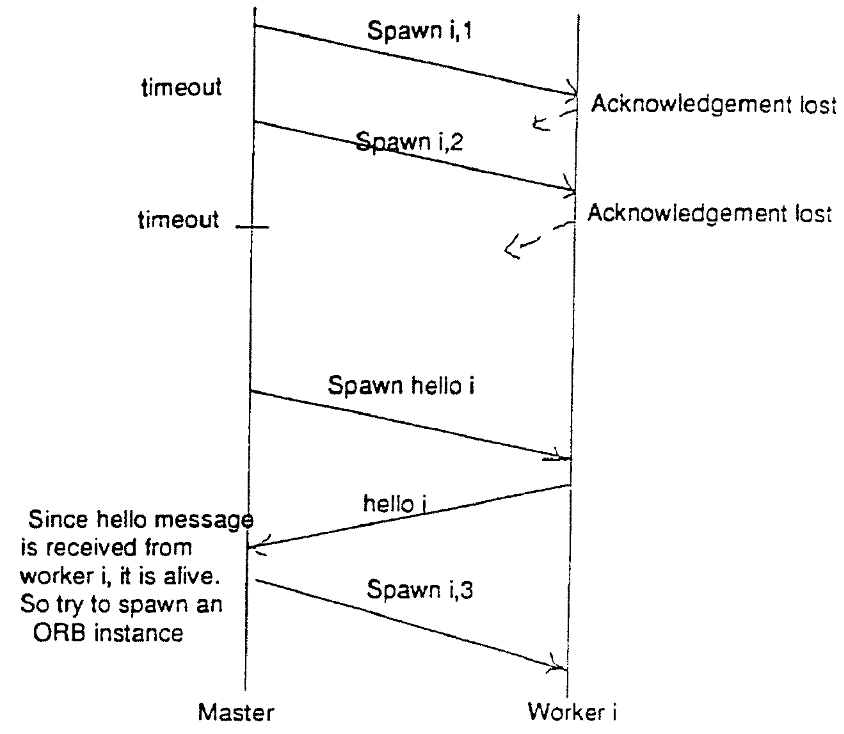


Figure 12: Abnormal Startup : Testing a Machine with "hello" message

are made (see Figure 13). In any case, the OM will terminate all but the ORB that responds with the first acknowledgment.

2.3 Abnormal ORB Termination

During normal operation, the OM uses the one-out-of-n-rule to evaluate the status of ORB_i in question. That is, if OM receives one $alive_i$ message out of the last N cycles, then it assumes that ORB_i is alive (see Figure 14). If OM gets no response from ORB_i for N system ticks, the OM attempts to spawn a new ORB instance by sending $sorb_{i,j}$ to processor P_i, where j-1 is the number of ORB instances previously spawned on processor P_i. If in the meantime, OM receives a response from a previously spawned $ORB_{i,x}$ (where x < j), the OM sends $terminate_{i,j}$ to P_i to cancel the attempt to start a new ORB.

2.4 Abnormal OM Termination

The ORB's, in a similar way, monitor the status of OM. When an ORB instance ORB_i

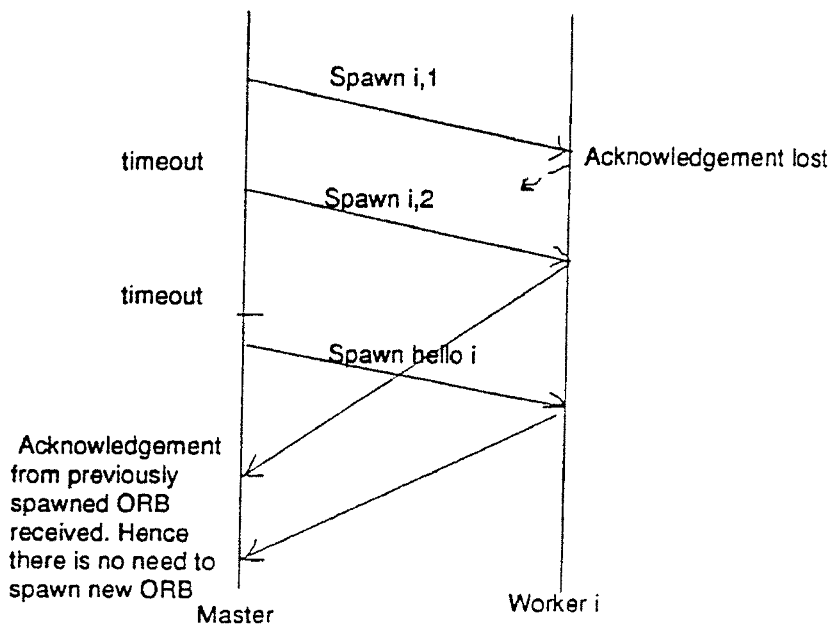

Spawn i,1
timeout
Acknowledgement lost
Spawn i,2
timeout
Spawn hello i
Acknowledgement from previously spawned ORB received. Hence there is no need to spawn new ORB
Master
Worker i

Figure 13: Abnormal Startup : No new ORB instance spawned

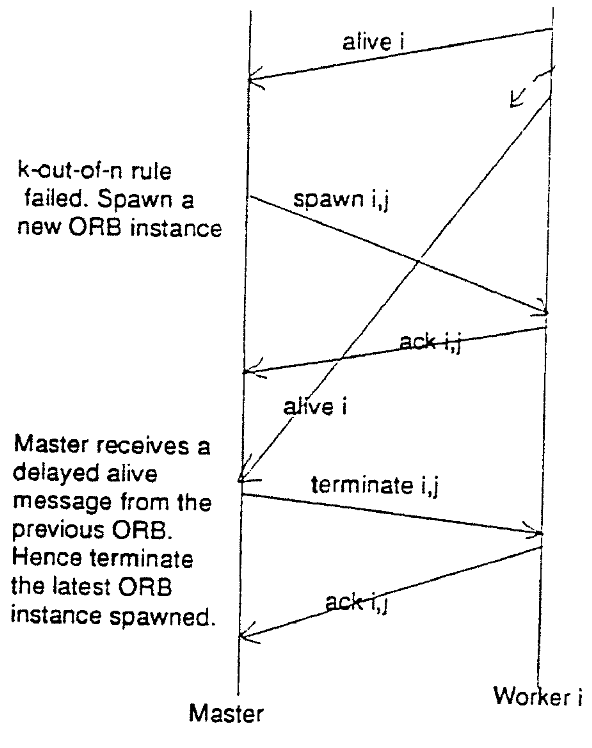

alive i
k-out-of-n rule failed. Spawn a new ORB instance
spawn i,j
ack i,j
alive i
Master receives a delayed alive message from the previous ORB. Hence terminate the latest ORB instance spawned.
terminate i,j
ack i,j
Master
Worker i

Figure 14: Abnormal Operation

does not receive any response, i.e., $alive_{om}$ message, from OM for k system ticks, it broadcasts a message $omdead_i(t)$ to all ORB's, where t is the local time at which message is sent by ORB_i (Figure 15). An ORB_j on processor P_j receiving such a message responds based on the following factors.

- If ORB_j still thinks that OM is alive, i.e., it received an $alive_{om}$ message from OM within its timeout period, it immediately sends a negative acknowledgement no_j to ORB_i (Figure 16a).
- If ORB_j thinks that OM is dead, i.e., it has also failed to receive an $alive_{om}$ message from OM within its timeout period, it constructs its reply based on the following conditions (Figure 16b and 16c).

  Let t_j be the timestamp and j be the ID of ORB_j receiving $omdead_i(t_i)$ from ORB_i.

  - If $t_i > t_j$, then send a positive acknowledgement yes_j.
  - If $t_i = t_j$, and i > j, then send a positive acknowledgement yes_j.
  - If the above two conditions fail, then send a negative acknowledgement no_j to ORB_i.

When an ORB_i that earlier initiated an $omdead_i$ message receives a negative acknowledgement no_j, it does not attempt to restart the OM, but it enters an intermediate state *WaitingForOMAlive*. If ORB_i does not receive $alive_{om}$ message within the next timeout period, it will send another $omdead_i$ message with a new timestamp. As long as the ORB_i is in the *WaitingForOMAlive* state, it responds with a positive acknowledgement yes_i to all *omdead* requests from other ORBs. ORB_i exits from the *WaitingForOMAlive* state only when it gets an $alive_{om}$ message, or when it receives no no_j messages and is selected to start a new OM.

In general there are many ways to select the new host for an OM when OM failure is detected. Here we use local timestamp and ORB ID's to choose a unique ORB to restart OM when more than one ORB has initiated *omdead* message. If ORB_i receives no negative acknowledgements within a certain timeout period, it assumes that all other ORB's have agreed that OM is down, and that it has been elected to revive OM. ORB_i now tries to spawn OM on the master machine. If it succeeds, normal operation resumes. If not, ORB_i spawns an OM on processor P_i and this OM takes over from the deceased OM (Figure 17).

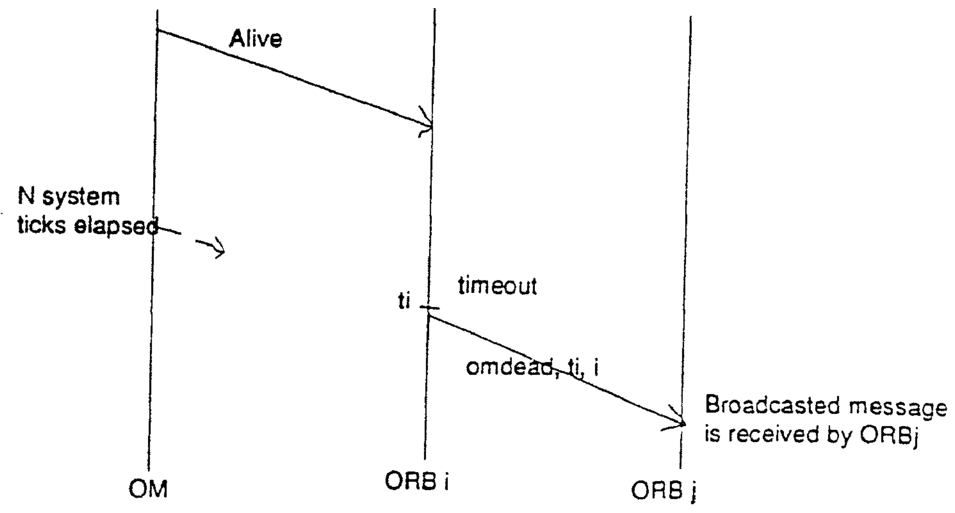
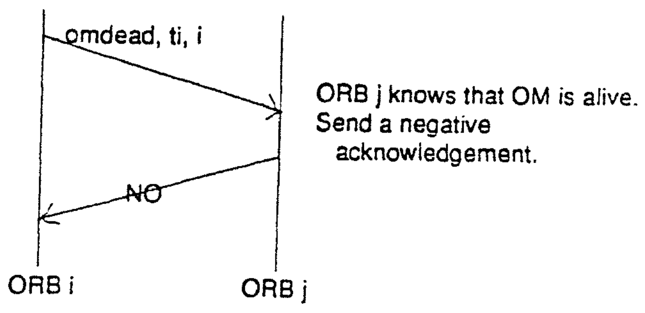


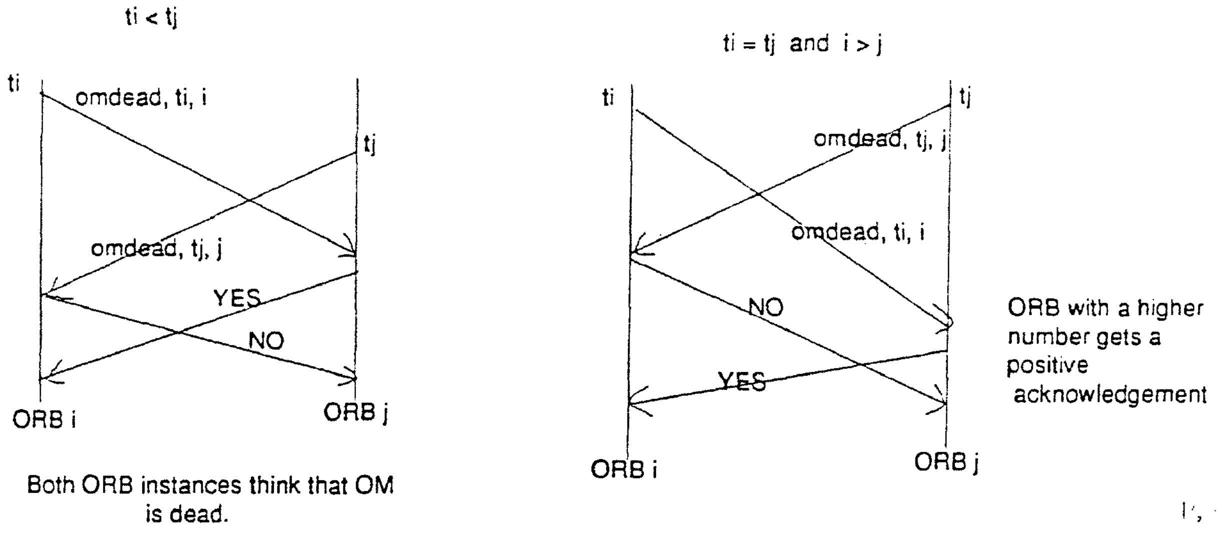
Figure 15: Requesting OM status from other ORB's

A more interesting type of OM *failure* occurs when the network link between the processor P_i and P_j, which hosts OM, is severed, thus partitioning the network. The processors in the network partition without the OM will take at most N system ticks before coming to an agreement that the OM is dead, and starting a new OM in that partition. Thus we end up with having two partitioned subnetworks, each with its own OM.

Whenever the distributed system recovers from a failure and returns to normal (either the network connection is reestablished or the original OM is restarted), the ORS will have more than one OM instance active. In order to maintain a single OM, each OM instance OM_i on processor P_i periodically broadcasts a $terminate_{om(i,t)}$ message (where t is the timestamp indicating the startup time of the OM). Any OM instance OM_j on processor P_j receiving one such message will terminate if j > i, or if j = i and $t_j > t_i$. The result is that the OM on the processor which has the smallest number survives. If there are two OM on the same processor, the OM that was started earlier survives. The original OM on the master machine always sends $terminate_{om,0}$ message. This message guarantees that the OM on the master machine will take over the job of managing ORB's when it comes back to life (Figure 18).


omdead, ti, i
ORB j knows that OM is alive.
Send a negative acknowledgement.
NO
ORB i
ORB j

(a)



(b)

Figure 16: Sending a reply to the request on OM status

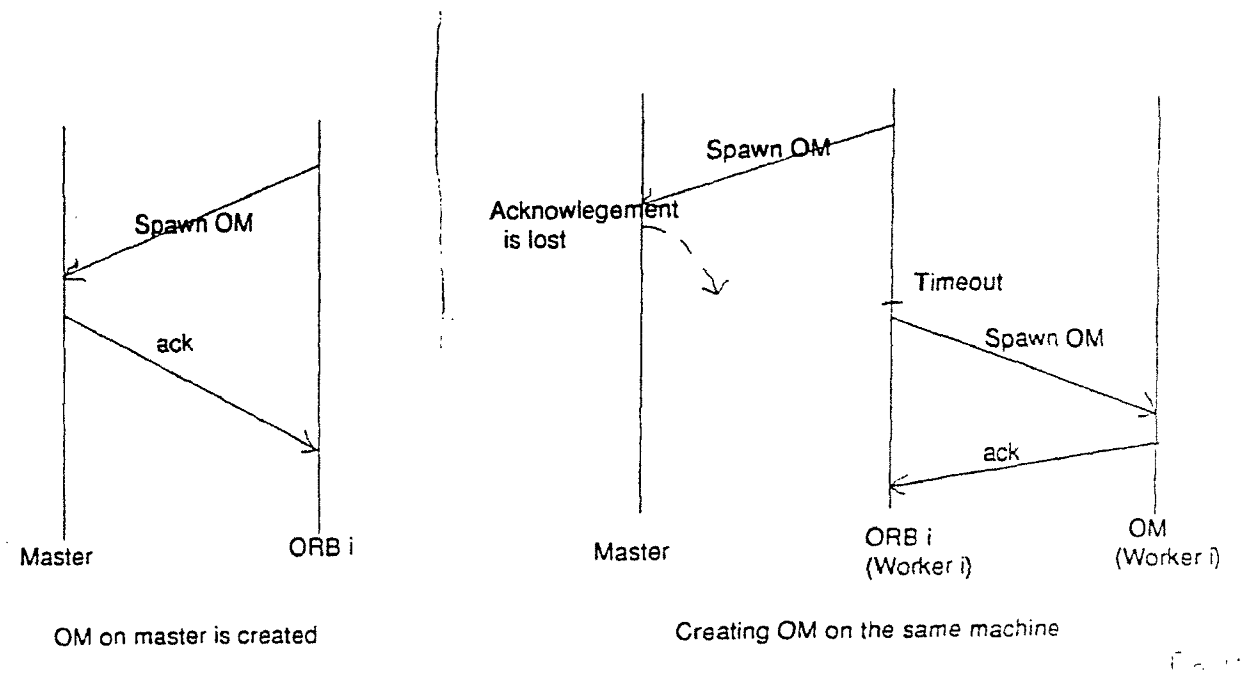


Figure 17: Creating a new OM

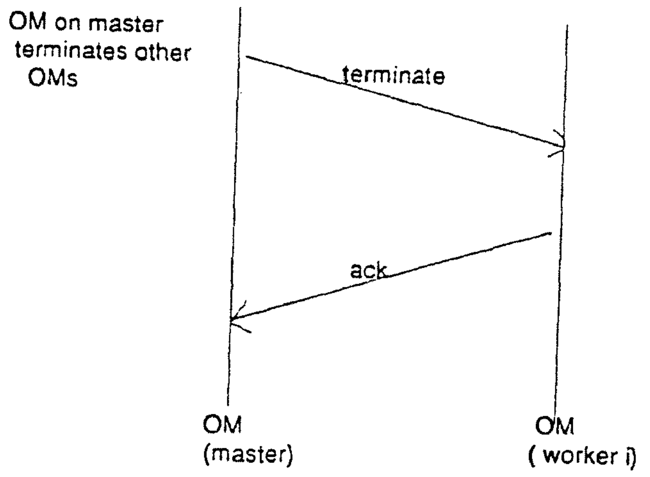


Figure 18: Original OM taking control back

2.5 Discussion:

The ORS design as described in this chapter functions normally under normal conditions. In order to establish the robustness of the ORS design, the behavior of the ORS under various fault conditions must be analyzed.

The following are the faults that are considered here :

- Failure of one or more worker machines. In the worst case, all the worker processors fail.
- Failure of the master.
- Network failure resulting in the partitioning of ORS, i.e., with M ORBs having an OM and (N - M) ORBs without OM.
- Network delay.

Let P_1,P_2, ... P_N be N worker processors and P_0 be the master processor. Let $OM_{(i,t)}$ represent the Object Manager instance on processor P_i started at local time t, and ORB_i denote the Object Request Broker instance on processor P_i.

Let U denote the set of all processors in the distributed environment, P denote the set of active processors, and Q denote the set of dead processors. At any time, $P \cup Q = U$ and $P \cap Q = \phi$

The ORS in its steady state has an Object Manager $OM_{(0,t)}$, N Object Request Brokers: ORB_1,ORB_2, ... ORB_N, P = { P_0, P_1, ... P_N }, and Q = {}.

Case 1: Failure of worker processors

When one or more worker processors fail, Q $\neq \phi$ and $P_0 \notin Q$. Assume that processor P_i fails, Q = { P_i } and P = U - Q.

A. Effect on System Operation: The OM on P_0 detects ORB_i failure when the OM stops receiving $alive_i$ messages. The OM marks ORB_i as dead, and the ORS system continues with just one less ORB in the ORS system. When P_i is restarted, the $hello_i$ process that is periodically spawned by the OM reaches P_i and successfully sends a message back to OM. The OM then restarts ORB_i on processor P_i bringing the system to normal. Multiple processor failures are identified as a sequence of single processor failures, each treated as described above.

B. Effect on user requests: We assume that failure of processor P_i means that there cannot be an active user on P_i, so there can be no local object requests. User requests emanating from users on processors in set P will see a mixed response. Locally resolvable requests are not at all affected by other processor failures. Requests that get globally resolved on a processor in set P will continue to function normally. However, objects registered on a processor in set Q cannot be retrieved. The result of an attempt to access such an object will be "object unavailable due to processor/network failure". When the processor hosting the object is revived, the object again becomes accessible.

Case 2: Failure of the master

When the master processor fails, $P_0 \in Q$.

A. Effect on system Operation: All ORB's stop receiving $alive_{om}$ messages, but due to the cyclic nature of $alive_{om}$ message generation, one of the ORB's, say ORB_i, senses OM failure first. It sends an $omdead_i$ message to rest of the ORBs; however, they all respond with a negative acknowledgement, *no*. Upon receiving one or more *no* messages, ORB_i enters the intermediate state where it waits for an $alive_{om}$ message. ORB_{i+1} senses failure next, and it sends $omdead_{i+1}$ to all ORBs. ORB_{i+1} will receive a yes_i from ORB_i and *no* from the rest, thus forcing ORB_{i+1} into the intermediate state. This process continues until the last ORB detects OM failure. In response to its *omdead* message the last ORB receives a positive acknowledgement from all other ORBs, which are all now in the intermediate state. The algorithm sketched earlier(Section 2.4 on Page 18) is used to select a processor to host a replacement OM. For N ORBs, this process takes N system ticks.

If one or more worker processors fail along with the master, the surviving ORBs still detect OM failure in the same way. An ORB instance that initiates an *omdead* message will receive $M < N$ messages in response. However, as long as one or more worker processors survive, the active ORBs can still proceed with electing a single active ORB to restart OM because the absence of messages from dead ORBs will not affect the election process. That is, if an active ORB_i does not receive any message from ORB_j within its timeout period, it proceeds with its operation, assuming that this represents agreement that OM is dead.

When the master recovers and the original OM starts functioning, there will be two OM in the ORS system, i.e., $OM_{0,tj}$ and $OM_{i,ti}$. When OM on the master sends a $terminate_{om(0,tj)}$ message, $OM_{i,ti}$ voluntarily terminates. The OM will then revive the dead ORBs, if any (Refer to Case 1).

B. Effect on user requests: The failure of master does not affect the user requests. Local and global resolution of objects still function normally for objects on accessible processors.

Case 3: Network Failure

A. Effect on System Operation: Network failure causes a partitioning of ORS. Assuming that the system is in steady state before network failure, Q = {}, and P = U, and that network partitioning divides the set of active processors P into Π_1 and Π_2 such that $\Pi_1 \cup \Pi_2 \equiv$ P. For the network partition Π_1, $Q_1 = \Pi_2$ and for network partition Π_2, $Q_2 = \Pi_1$. Let $P_0 \in \Pi_1$. This subsystem will behave as in Case 1, with M_1 worker processor failures. For partition Π_2 the situation appears as OM failure plus M_2 worker processor failures (Case 3). Thus the two subsystems function independently with separate OMs.

B. Effect on user requests: Objects registered on ORBs in one network partition cannot be retrieved by ORBs on the other partition. Thus some user requests may not be satisfied. When the network connection is reestablished, objects on the other side of the partition become visible to the entire system.

Case 4: Network delay

A. Effect on System Operation: Network delays may create situations where an ORB or an OM makes a wrong decision, i.e., where a process that is actually alive is assumed to be dead. However, in cases of both ORB and OM processes, the protocol described above guarantees that an attempt to start a new ORB_i/OM will eventually result in a single process domination. That is, if the *alive* message from ORB_i is excessively delayed, the OM presumes that the ORB is dead and starts a new ORB. When the OM receives the delayed message, it acts to terminate the recently spawned ORB. The system thus comes back to normal. In situations where the message from the OM to ORB_i gets delayed and ORB_i

sends *omdead* message to other ORBs, ORB_i should get back negative acknowledgements that prevent it from spawning a new OM.

B. Effect on user requests: Sometimes network delay may cause delay in the system response to user requests. If the network delay exceeds certain timeout periods associated with object resolution, an "inaccessible object" result might be sent to the user when there is no processor or network failure.

Chapter 3

Qualitative Evaluation

Following the design of high level protocols to meet distribution and reliability requirements, the next goal is to realize these protocols in C-Linda, the parallel programming language selected to implement the ORS design. Following a brief introduction to C-Linda, I will address several questions that need to be answered before proceeding with the actual implementation. These questions will be answered by a systematic evaluation of C-Linda (Qualitative Evaluation in Chapter 3 and Quantitative Evaluation in the following chapter).

3.1 Introduction to Linda

Linda is a programming model based on a shared global tuple space and several tuple space operations [2]. These tuple space operations can be embedded in any standard language such as C or FORTRAN, creating a new parallel language. Linda's tuple space abstraction permits both communication and synchronization, as well as mechanisms for creating and coordinating multiple execution threads. The tuple space forms an associative shared memory that consists of sets of data called tuples.

There are six basic tuple-space operations in C-Linda.

- **out(t)** - causes a new tuple **t** to be evaluated and added to the tuple space.
- **in(t)** - causes some tuple **s** to be withdrawn from the tuple space. The tuple **s** is chosen from among those that match the template **t**. The values of the actuals in s are assigned to the formals in **t**. If no matching **t** is available when in(t) executes, the invoking process is blocked until one such matching tuple is available. If many matching s's are available, one is chosen arbitrarily.
- **rd(t)** - it is identical to **in(t)** except that the matched tuple remains in the tuple space for use by other processes.
- **inp(t)** - it is a non-blocking form of **in(t)**. It returns a 0 if no matching tuple exists or

else withdraws the matching tuple and returns a 1.

- **rdp(t)** - it is a non-blocking form of **rd(t)**
- **eval(t)** - it is very similar to **out(t)** except that the tuple **t** is evaluated after it is placed in the tuple space, rather than before. This implicitly creates new process to evaluate each field of **t**.

There are two kinds of tuples : process tuples, which are under active evaluation. and data tuples, which are passive [4]. Processes accomplish work by generating. using. and consuming data tuples. A process tuple is a process that executes. then turns into a data tuple at termination time conceptually indistinguishable from all data tuples.

A tuple is a sequence of typed values. e.g., a tuple with a string. a real number. an integer. a variable, and a function as its parameters is shown below:

("a string", 15.25 , 22 , x, function(p))

An **out** operation adds a passive data tuple into the tuple space. The following operation adds a 3-parameter passive data tuple into the tuple space:

out("a string", 15.25 , 22)

An **eval** operation adds a process tuple into the tuple space. The following operation adds a 2-parameter active process tuple into the tuple space:

eval(square-root(10), mean(12,23,34))

This process tuple has two processes associated with it; the square-root process and the process to compute the mean of three numbers.

An anti-tuple is a sequence of typed fields; some of which may be actuals, whereas others may be formals. A formal is prefixed with a question mark. e.g.,

("a string",?r,?i,30,?z)

Here the first and the fourth fields are actuals. and the rest are formals. **in**, **inp**, **rd**. and **rdp** attempt to match a passive data tuple in the tuple space with the anti-tuple supplied as the operation's argument.

- **in("coord", 10 , 20)** matches the tuple (**"coord", 10, 20**) in the tuple space.
- **in("coord", ?x , 30)** matches all tuples with the first argument as string "coord". and the third argument equal to 30. For instance, the tuple (**"coord", 75 , 30**) will match the given anti-tuple (**"coord", ?x , 30**). When the tuple is retrieved from the tuple space, the value of formal parameter x is set to 75.

The **eval** operation adds process tuples, which means that it provides a mechanism for dynamically creating processes. In some ways it is similar to the standard UNIX **fork** system call. Both **eval** and **fork** create new processes. However the process created by **fork** is inherently related to the parent process. whereas in Linda. processes created by **eval** have no special relationship with the process that created them: they pass their result into the tuple space, not to their parent process.

An example of a Linda system that creates 100 (parallel) processes to perform square-root on the first 100 integers is:

```
        for ( i = 0 ; i < 100 ; i++) eval(sqrt(i)):
```

100 process tuples are thrown into the tuple space. Each process computes the *sqrt* function concurrently for one value, then converts it into a passive data tuple. Thus 100 result tuples are eventually formed, which can be read from the tuple space by the parent or any other active Linda process.

3.2 ORS Implementation in Linda

The decision to implement the distribution protocol for ORB using Linda has two important ramifications.

- The abstract design must be translated into *abstract* Linda.
- The *abstract* Linda must be translated into a *real* Linda, taking into account particular implementation restrictions and constraints.

Thus the following questions need to be answered before one can proceed with implementation.

Abstract Linda:

- Is it possible to map a process onto a specific processor using Linda's eval?
- How can we map exactly one process instance to each processor (e.g., exactly one ORB_i on processor P_i)?
- How can we pass messages from one process to another?

Real Linda:

- How can a user communicate with an ORB_i?
- How does the master (OM) detect process (ORB_i) failures?
- How does message size affect system performance?
- In a non-homogeneous collection of processors, which machine should run the OM?
- What other restrictions on abstract Linda are imposed by its implementation. and how do they affect the implementation of ORS design?

In order to answer these questions, qualitative and quantitative analyses of real and abstract Linda are necessary.

3.3 Methodology :

- **Environment.** All experiments are conducted on a cluster of IBM RS6000 workstations running AIX 3.2. The cluster is supported by a central server which is slightly more powerful than other workstations. With the remaining machines of equal processing power, the environment can be considered as homogeneous. Linda tuple space operations create and manage a tuple space that is logically distributed across all workstations. In fact, the tuple space is implemented as a collection of local **tuple lists**, one per workstation, each containing those tuples generated locally. Access to non-local tuple lists is supported via interprocess communication.

- **Programs.** All experimental programs are written in C-Linda and compiled with **clc(v2.4.6)** compiler from Scientific Computing Associates. Programs are distributed to all the machines using **tsnet** utility program and then executed. **tsnet** is an utility for invoking network Linda executables, setting up configuration files, distributing executables to remote machines, executing, removing executables, and killing remote processes. The configuration file contains the names of all machines participating in the distributed system and the datagram port number used for interprocess communication.
- **Timing.** Three functions are provided in the C-Linda toolkit for timing modules. The timing functions implement a stopwatch facility that is useful for collecting statistics on parallel execution. These timing functions measure real time.

  The three timing functions are :

  - **starttimer()** : initializes the stopwatch.
  - **timersplit(label)** : takes a time split (i.e., a stopwatch reading) when called, and labels the time split with the specified label.
  - **printtimes()** : prints all time splits executed so far in the tabular format with labels.
- **Results.** Each test program is executed several times, with execution times averaged to produce the final result. All the programs are run under conditions when the load on the participating workstations, the server, and the local network are fairly static and minimal (i.e., when no other user is executing programs, leaving only system daemons active).

3.4 Qualitative evaluation :

Qualitative evaluation is an attempt to answer some of the specific questions raised during implementation of the ORS design in C-Linda. This evaluation is driven by the problems and concerns that arise during implementation.

Question 1: Is it possible to spawn a process on a specific processor using Linda operations?

```
MAIN
Begin
    eval (Worker-process)
    in (destination-machine-name)
End /* MAIN */

Worker-process
Begin
    gethostname(Host-machine-name)
    out (Host-machine-name)
End /* Worker Process */
```

Figure 19: Program 1

Importance : The ORS design requires that instances of both OM and ORB be located on particular processors.

1-A (Abstract Linda): Abstract Linda is a programming model based on a shared global tuple space with several tuple space operations. Because abstract Linda makes no reference to processors and their names, it does not directly support processor directed primitive process management.

1-B (C-Linda):

In Linda, **eval** is the only operation that creates active processes on remote machines. The **eval** primitive in C-Linda provides no direct form of process management, but a C-Linda based process management protocol can be developed.

Referring to Figure 19 [1], Program 1, the main-program spawns a worker on one of the processors from the list of processors in the "tsnet.nodes" file that defines the processors participating in the distributed system [2]. In C-Linda, the following default strategy is used to map processes to processors. Each Linda program has a "tsnet.nodes" file that lists the processors participating in the distributed program. The processor listed in the middle of

[1] Source code for all programs is listed in Appendix B. The outline of programs is shown here.

[2] The name of the processors is sequentially listed on each line in the "tsnet.nodes" file. In the discussion that follows, the first item refers to the processor listed on the first line, the middle item refers to the processor listed in the middle line.

```
MAIN
Begin
    loop N times
        eval (Worker-process)
    loop N times
        in (destination-machine-name)
End /* MAIN */


Worker-process
Begin
    gethostname(Host-machine-name)
    out (Host-machine-name)
End /* Worker Process */
```

Figure 20: Program 2

the "tsnet.nodes" file is used as the **eval**'s target unless it is the local host. If the middle item is the name of the local host, then the processor listed just above it is selected. Here we have some indication that the **evaled** process does not get mapped to the local host, which is confirmed later by another test. If more than one **eval** call is made, although the destination of the first call be predicted, the destination processors for subsequent calls to eval are randomly chosen (See Figure 20, Program 2).

The protocol to do *directed eval* is illustrated in Figures 21 and 22. The main routine simply **evals** a worker process tuple with one argument. Upon reaching a destination machine, the process discovers the name of its destination machine by calling a built-in C-function **gethostname**. The process then compares its destination with its intended destination. If they do not match, the **evaled** process terminates and sends a message to the main routine indicating failure. The master routine **evals** another such process, and continues until it succeeds in placing a process at the correct destination. This method of random spawning works because C-Linda maps processes to processors in a round robin fashion. The exact order in which the processors are chosen is random. This program may not always succeed in placing the process on a given machine. If one tries to do a directed **eval** to the local host, it never succeeds because the current C-Linda implementation never makes an attempt to **eval** a process on the local host.

The program also fails if one tries to make a directed **eval** to a processor which already

```
MAIN
Begin
    out (Destination-address)
    loop forever
    begin
        eval (Worker-process)
        in (success-flag)
        if (success-flag = TRUE)
            Exit the loop
    end
End /* MAIN */
```

Figure 21: Program 3 (Part-1)

has one **evaled** process running. This gives an indication that two processes are not **evaled** on the same processor. The following test confirms this claim. Program 4 [3], listed in the Appendix B, contains a main routine which **evals** a few processes, then terminates after all the processors reach their destination. The program executes without any problem if the number of processes spawned is less than the number of actual processors listed in the "tsnet.nodes" file. However, if the number of **evals** is more than the number of processors listed in the processor list, the program never terminates. There is nothing wrong with the program logic, but the program fails to terminate because all the processes do not run simultaneously. When the number of processes **evaled** is more than the number of processors listed in the "tsnet.nodes" file, the first N processes are mapped one-to-one on the first N processors (barring the host). The remaining processes, that are stacked in a waiting list, get mapped to a processor as soon as the process running on that processor terminates. In program 5, shown in Appendix B, the process that gets mapped to a processor terminates only after all the processes get mapped to a processor. Because this condition can never be met, the program enters a deadlock.

C-Linda Result Summary (Question 1):

- C-Linda does not directly support process (**eval**) to processor mapping.
- It is possible to predict the destination of the only the first **eval** operation.

[3] Program 4 is a slight modification of Program 3. See Appendix B source code listing for more details.

- In most cases it is possible to implement a process to processor mapping protocol based on making repeated **evals** which succeed only when mapped to the correct destination.
- In some cases the process to processor mapping protocol does not succeed in placing a process on the required processor.

Question 2: How can we spawn exactly one process (ORB) on one processor ?

Importance : The ORS design requires that exactly one ORB instance is running on one processor when the ORS is in the operating phase.

2-A (Abstract Linda): Abstract Linda, as explained before, does not support processor based process management. Hence any notion that is processor specific cannot be realized in Abstract Linda.

2-B (C-Linda):

The protocol to map exactly one process on one processor is illustrated in Figure 23, Program 6. This program **evals** N - 1 processes. The destination machines are chosen randomly but when all N - 1 processes find a destination, each processor except the local host has exactly one process running on it. The ORS design requires that if one of the ORBs stops communicating with the OM, a new ORB instance has to be spawned on that processor. However, in C-Linda it is not possible to **eval** another process on a processor which already has one process running (See 1-B).

C-Linda Result Summary (Question 2):

- By spawning as many processes as there are processors, one can have exactly one process on a processor.
- Only normal startup is supported by the C-Linda implementation.
- C-Linda cannot always maintain exactly one process on a processor. When processor or network failure disturbs normal operation, the one-to-one mapping can also be disturbed.

```
Worker-process
Begin
    gethostname(Host-machine-name)
    rd (Destination-machine-name)
    if (Destination-machine-name = Host-machine-name)
    begin
        out (TRUE)
    end
    else /* Host-machine-name is not Destination-machine-name */
    begin
        out (TRUE)
        terminate
    end
End /* Worker Process */
```

Figure 22: Program 3 (Part 2)

Question 3: How can a user communicate with the **evaled** process (ORB)?

Importance : The primary objective of developing an Object Registration System is to enable users to add, delete, and execute object or object instances in a distributed environment. It is very important that the user on processor P_i be able to communicate with the ORB on that processor.

3-A (Abstract Linda): A user is associated with a processor. Thus communication between user and an **evaled** process is, in effect, communication between the **evaled** process and a user on a particular processor. Abstract Linda does not support processor based themes; hence a specific communication channel cannot be established between an **evaled** process and a user in Abstract Linda.

3-B: C-Linda:

A simple protocol that attempts to establish communication between a user and an **evaled** process is described in Program 7 (Figure 24). The MAIN-MODULE(OM), running on processor P_j **evals** an ORB process on processor P_i. The ORB uses a **gets** routine (**gets** is a standard C function to read a string from the user) to read a message from the user, then

```
MAIN
Begin
    loop (N - 1) times
        eval (Worker-process)
End /* MAIN */
```

Figure 23: Program 6

echos the message back on the user's screen using a **printf** statement. The message should be read from the user on the processor on which the ORB-process is running. However, in the C-Linda implementation, the spawned ORB process spawned is unable to read any information from the user on either the remote machine or the local machine. Further, any output generated is printed on the machine that hosts OM. This test shows that a process **evaled** on processor P_i cannot directly communicate with the user on that processor via simple I/O.

The communication protocol is modified in Program 8 (Figure 25) so that the evaled ORB process reads from one file and writes messages into another file on its host processor P_i. Test execution shows that spawned processes are able to read and write into files on their host processor, so that files can be used as logical "pipes" between ORB and user processes.

C-Linda Result Summary (Question 3):

- User and **evaled** processes communicate indirectly through files. That is, the user process writes commands into an input file and reads the result from the output file, while the ORB reads the commands from input file and writes the results into the output file.

Conclusion:

- Direct communication between user processes and **evaled** processes (ORB) on a processor is not possible.

Question 4: How does the master (OM) detect ORB failure and vice-versa ?

Importance : To build a robust distributed system, it is necessary for both the OM and the ORBs to detect the failures of other components.

4-A (Abstract Linda): In abstract Linda, the OM and ORBs are process tuples which periodically exchange information to ensure proper functioning. When the OM process tuple fails to get messages from the ORB process tuple within the timeout period, the OM detects ORB failure and the OM creates a new process tuple to function as an ORB. Likewise the ORBs can also detect OM failure when they stop receiving $alive_{om}$ message from the OM.

4-B: C-Linda

All C-Linda programs are executed using **tsnet** utility program. This utility program distributes the executables to all the machines listed in the "tsnet.nodes" file[2]. Program 9[4] illustrates a trivial ORS system where the OM on the master **evals** ORBs on all the worker processors. When the normal startup is complete, the ORBs periodically send alive message to OM and the OM sends $alive_{om}$ message to all ORBs. Under this steady state, if an ORB_i fails, the OM detects failure only when it stops receiving the periodic $alive_i$ message within its timeout period. Thus failure detection is not instant and it takes at least one system tick to sense failure. In the experiment, ORB failure was simulated by terminating the ORB process on processor P_i through an external termination signal. As soon as one of the ORB process is killed, the **tsnet** utility immediately sensed ORB failure and terminated all other ORBs and the OM. Thus the OM never was able to detect ORB failure, as the **tsnet** utility provided an immediate, conflicting response (i.e., to terminate all system processes).

In order to prevent **tsnet** from detecting failures, the ORB must be able to trap all the **termination signals** that can cause its termination and perform a normal termination prior to destroying its tuple space. However there are signals which can never be trapped by ORB: when one of these signals are generated, the entire system falls apart.

C-Linda Result Summary (Question 4):

- An accidently terminated ORB terminates the entire ORS system.
- Termination signals sent by the operating system cause ORB termination. Most of these termination signals can be trapped by the ORB and the ORB can initiate normal termination upon receiving these signals.

[4] See Appendix B

```
MAIN
Begin
    eval (ORB-process)
End /* MAIN */

ORB-process
Begin
    gets (message)
    printf (message)
End /* ORB-process */
```

Figure 24: Program 7

- Some termination signals can never be trapped by the ORB. These signals eventually lead to system termination.

  Conclusion:

- C-Linda does not allow ORB and the OM to detect failure.
- A completely robust ORS system cannot be built using the current Linda implementation.

Question 5: Can Linda program work with UNIX **fork** calls ?

Importance: One of the functions of ORB in the ORS design is to support multiple users on a single machine. In order to do this, either the ORB has to serve each user in a timesharing fashion or it has to spawn a new process to serve every user.

5-A: Abstract Linda: Abstract Linda is system independent. It does not assume anything about the underlying architecture or the operating system. Hence, there should not be any problem with using UNIX **fork** calls in abstract Linda programs.

5-B: C-Linda:

There are two approaches to support multiple ORS users. One approach is to have a single ORB process that attends to all users' need by timesharing. The second approach is to have the ORB instance on a processor create a separate process for each user requesting

```
MAIN
Begin
    eval (ORB-process)
End /* MAIN */

ORB-process
Begin
    Open FILE-A
    message = Read (FILE-A)
    Open FILE-B
    Write (FILE-B , message)
End /* ORB-process */
```

Figure 25: Program 8

service. This latter approach is a standard method used normally in client-server based applications. In order to realize the second approach, a process has to be spawned to serve each user. However, using **eval** to create a server process for each user will not work because in C-Linda the destination processor of an **eval** operation is never the local host. Moreover, C-Linda does not allow more than one process to be spawned on one processor (See 1-B).

The only other option left is to use the UNIX **fork** command to create a new process. The process created inherits all run-time information from the parent process. Program 10[5] demonstrates the effect of using UNIX **fork** calls in C-Linda routines. In this program, the main-module spawns the processes using **eval** and each spawned process spawns two sub-processes using **fork**. The new subprocesses created communicate with the main-routine by sending messages. Upon testing the program, it was noted that the main-routine received multiple messages from a sub-process when only one message was sent. The Linda system behaved unpredictably, and the system finally crashed. This behavior results because the **fork**ed process inherits a copy of all data structures from the parent process, including the message table and hash table used to manage the C-Linda tuple space. Thus, there will be multiple hash tables and message tables. Further, the alarms for the child process are cleared when **fork** is called [5]. C-Linda uses the alarm signal (SIGALRM), hash tables, and message tables for its normal tuple space management. Calling **fork** interferes with this normal operation and produces unpredictable results.

[5]See Appendix B

C-Linda Result Summary (Question 5):

- C-Linda does not work properly when the UNIX **fork** function is called. System execution becomes chaotic and terminates abnormally.
- It is impossible to create a new process to handle each user. Instead, the ORB should serve each user in a timeshared fashion.

Other Limitations of Linda Implementation

- Linda uses SIGTERM (Software termination signal), SIGALRM (Alarm signal), and SIGIO (I/O signal) signals for its proper operation. These signals should not be redefined in the C-Linda program. Further, functions like sleep, usleep, longjump, setjmp, alarm, ualarm should not be used in the program. These function calls redefine signals used by C-Linda, which may result in improper operation of the C-Linda system.
- The program should not contain functions that allocate or deallocate memory segments. Function calls like calloc, malloc, and sbrk should not be used in C-Linda programs.
- The processes created by using **eval** can only accept simple data types as arguments. Aggregates (an array or a dynamically allocated chunk of memory) cannot be passed as parameters to the **eval**ed processes.

  C-Linda allows the following **eval** operation where the spawned process has two parameters: an integer, and a real number.

  eval(process(10 , 13.34))

  C-Linda does not allow the following **eval** operation where the parameter passed to the **eval**ed process is an array of numbers.

  eval(process(array))

  This limitation of C-Linda forces a restriction during ORS implementation. Complex information that needs to passed to a spawned process has to be passed as a message in the global tuple space. Further the tuple matching is fairly rigid and objects of dynamic size cannot be retrieved from the tuple space using anti-tuples.

Chapter 4

Quantitative Evaluation

This chapter describes benchmarks that reflect performance measures for different primitives of parallel computation with ORS implementation in mind. The benchmark is based on Linda tuple space. The results of a benchmark are influenced not only by the performance of the underlying architecture. but also by the implementation of Linda on that architecture [1].

Srikanth Kambhatla, Jon Inouye. and Jonathan Walpole[1] describe the benchmarking of Linda on parallel machines via a software architecture. Benchmarking using software architecture makes the resulting benchmarks portable since the use of software architecture masks the diversities of underlying architectures. "BeLinda" is the software benchmark that is based on Linda tuple space. "BeLinda" defines an appropriate level of abstraction for comparing different parallel platforms. Using "BeLinda". three different Linda based parallel architectures were evaluated: Sequent Symmetry [6], The Intel iPSC/2 [7], and the Cogent XTM [8]. Benchmark results on these architectures were then compared. Results showed that the cost of doing any Linda primitive operation on shared memory architecture is substantially less than on any other machine. Distributed memory architecture and hybrid network architecture incur communication overhead during tuple space operations. Overall performance of Sequent Symmetry is better than the rest. iPSC/2, which is based on distributed memory model, is comparatively better than the Cogent XTM, which is based on hybrid networking. These results led to the conclusion that communication plays a big role in the implementation of any parallel architecture.

My approach to benchmarking specifies the performance with respect to ORS on a specific parallel architecture: A cluster of workstations in a network environment. Whereas "BeLinda" gives a comparision of Linda implementation on different parallel architectures.

[6] Sequent Symmetry is based on shared memory architecture.

[7] Intel iPSC/2 architecture consists of a distributed memory model.

[8] Cogent XTM workstations formed a hybrid network of a shared bus and a crossbar.

4.1 Specification :

The benchmark suite is a set of seven individual programs that evaluate the characteristics of C-Linda. The benchmark suite is designed with the ORS design in mind. and with the specific goal of providing answers to the following questions.

- What is the average time taken to perform each primitive operation ?
- It is known that messages and data should be placed in the tuple space as tuples. What is the effect of the number of parameters in the tuple for each primitive operation?
- Tuples are extracted from the tuple space by tuple matching. What is the effect of the number of actual (known) and formal (unknown) parameters in an anti-tuple?
- Messages and data have to be moved in and out of the tuple space during ORS operation. What is the effect of the size of the message on system performance? Is it better to send data in numerous small packets or as a single huge packet?
- As user adds more objects into the tuple space. the size of the tuple space grows. Is there any limit on tuple space size ? If so, what is the maximum limit ?
- ORBs and OM periodically exchange alive messages. What is the average latency time between sending and receiving messages ?
- In the ORS, there is a single master and several worker machines. What factors determine the selection of a master machine ?

Benchmark 1 (primitives.cl):

Benchmark Program 1[9] evaluates the cost of doing the basic Linda primitive operations by performing N primitive operations of each type. The time is then divided by N to obtain the average time for each individual operation.

[9] Source code listing of all the Benchmark Programs is listed in Appendix C. Inline code shows the algorithm used in pseudo code.

Operation	Time (sec)
IN	0.0050272
OUT	0.0030588
RD	0.0049597
INP	0.0050056
RDP	0.0050089
EVAL	0.0032121

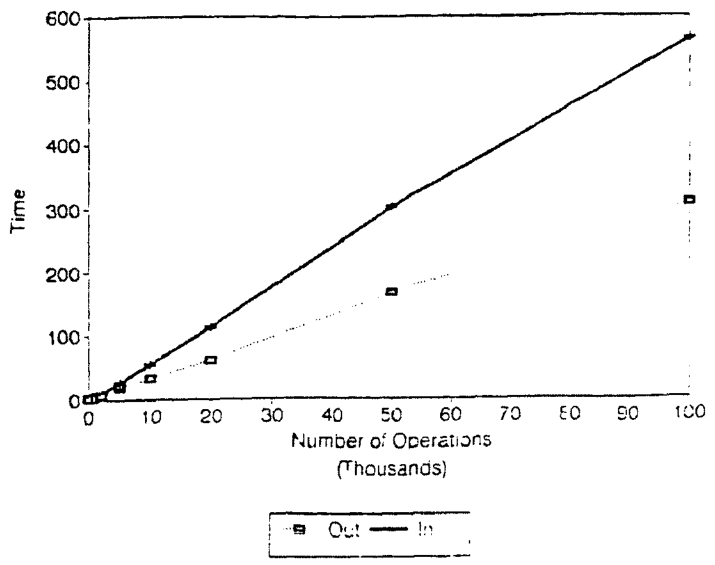
Table 1

Benchmark Program 1:

```
MAIN
Begin
    StartTime = GetTime()
    loop N times
        do-a-primitive-operation /* e.g., OUT */
    StopTime = GetTime
    AverageTime = ( StopTime - StartTime ) / N
End /* MAIN */
```

Primitive operations that add tuples into the tuple space (**out** and **eval**) take almost the same time. Primitive operations that remove the tuple from the tuple space fall under another category and they take almost the same time, which is higher than the time taken by **out** and **eval** (See Table 1). Tuple reading operations take more time since they have to do tuple matching before removing the tuple from the tuple space. Hence the difference in time roughly corresponds to the tuple-matching overhead.

Graph 1 shows the effect of sending multiple messages. From the graph it is clear that there is no additional overhead when multiple messages are sent in succession.



Graph 1

Benchmark 2 (actuals.cl):

Benchmark Program 2 evaluates the cost of executing a primitive operation with a varying number of actuals in the tuple. This is achieved by varying the number of actuals and timing the **out**, **in**, and, **rd** operations.

Benchmark Program 2:

```
MAIN
Begin
    StartTime = GetTime()
    loop N times
        OUT ( param1, param2, ... , paramX )
    StopTime = GetTime
    AverageTime = ( StopTime - StartTime ) / N
End /* MAIN */
```

The results in Table 2 and Graph 2 show that the number of actual parameters in a tuple does not add significant overhead while adding or retrieving tuples. The only overhead is the time taken to perform a primitive operation. This would mean that instead of sending several single parameter messages, it is more efficient to send a few multi-parameter messages. Moreover, in cases where several parameters have to be passed from one node to another,

Number of Actual parameters

(1000 messages)

	1	2	4	8	16	32	64	512
OUT	3.387672	3.432154	3.308180	3.295579	3.256871	3.550132	3.473919	5.299545
IN	5.855959	5.988766	6.003955	6.219705	6.400188	6.187505	6.330471	6.928326
RD	5.886401	6.089635	5.920350	6.101180	6.255173	6.239125	6.256238	6.833687

Table 2

the cost of adding or deleting a few parameters is insignificant when compared to the time taken to perform the primitive operation.

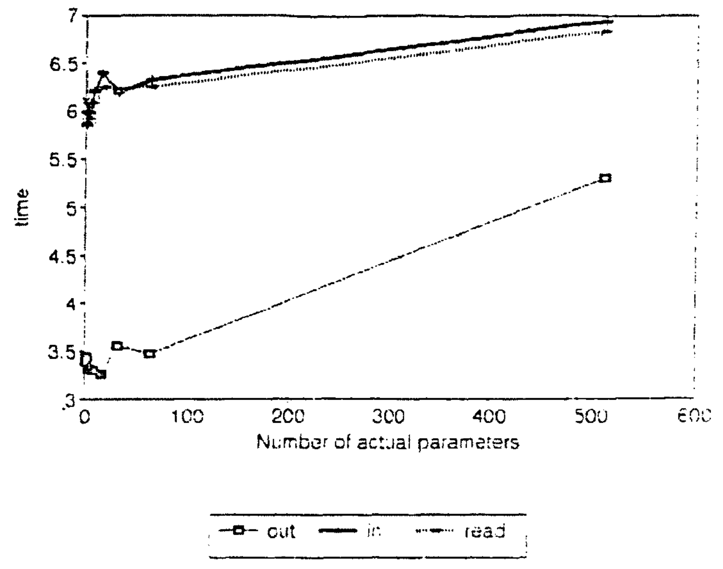
Benchmark 3 (formals.cl):

Benchmark Program 3 evaluates the cost of executing a primitive operation with a varying number of formals in the anti-tuple. This is achieved by varying the number of formals in the tuple ans timing **rd** and **in** operations.

Benchmark Program 3:

```
MAIN
Begin
    StartTime = GetTime()
    loop N times
        IN ( formal1. actual1. ... , actualX )
    StopTime = GetTime
    AverageTime = ( StopTime - StartTime ) / N
End /* MAIN */
```

The number of known and unknown parameters does not have much effect on the time to perform a primitive operation. No conclusive stand, regarding the exact nature of tuple



Graph 2

matching, can be taken from the results obtained (See Table 3). The results seem to indicate that tuple matching takes more time when the number of formal parameters in the anti-tuple is zero or maximum and takes the least time for an intermediate number of formal parameters.

Benchmark 4 (message.cl):

Benchmark Program 4 measures the time to send messages of different sizes.

Benchmark Program 4:

-Main-module spawns two processors P_1 and P_2.
-P_1 sends a sequence of packets of size N to P_2.
-Time taken by P_2 to receive packets is measured.
-Experiment is repeated for different packet sizes.

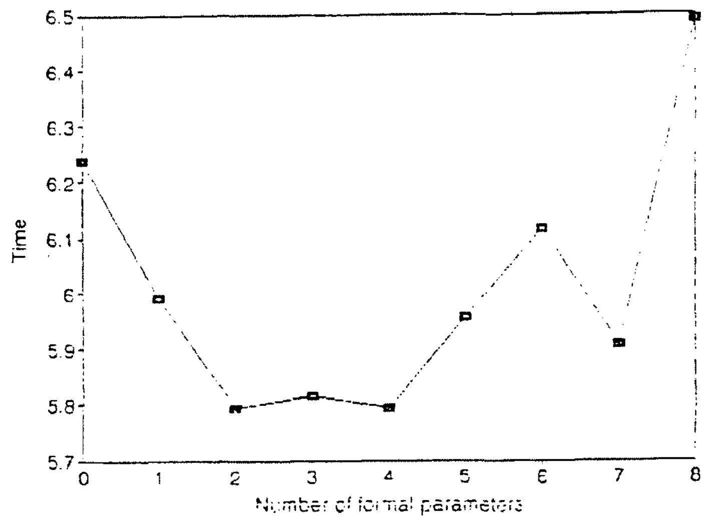
From the results, it is clear that up to a certain extent, it is better to send a few large packets than to send several small packets of data. For example, the results in Table 4 show that the time to send 10000 bytes of message would be 33.07 seconds for 10,000 packets of size 1, 0.7 seconds for 100 packets of size 100, and 0.0424 seconds for 1 packet of size 10,000. However, to send a very huge packet of size 2,000,000, it is more efficient to send two packets of size 1,000,000 rather than a single packet, i.e., 219.629 seconds vs. 20.672

Number of Formal parameters

(1000 Operations)

IN (No of params)	0	1	2	3	4	5	6	7	8
1	6.77659	6.42703							
2	6.36269	6.20184	6.02628						
4	5.95185	6.14900	5.78327	6.25902	5.81890				
8	6.23779	5.99157	5.79232	5.81393	5.79296	5.95756	6.11530	5.90666	6.48850

Table 3



Graph 3

	Number of packets						
Size	1	10	50	100	1000	10000	50000
1	0.0128	0.0808	0.163	0.68	6.738	32.07	323.008
100	0.012	0.075	0.318	0.70	3.188	30.098	163.387
1000	0.0134	0.101	0.189	0.359	3.628	82.97	>452
10000	0.0424	0.428	1.412	3.317	31.129	>245	
50000	0.1218	1.099	5.546	12.297	>213		
100000	0.245	2.097	10.436	20.84	>152		
500000	1.029	10.627	135.6	>215			
1000000	2.097	20.672	>159				
2000000	219.629	>219					

Table 4

seconds. This implies that there is a limit on the message size that can be placed in the tuple space. Results show that packet size greater than 2.000.000 cannot be efficiently sent from one process to another.

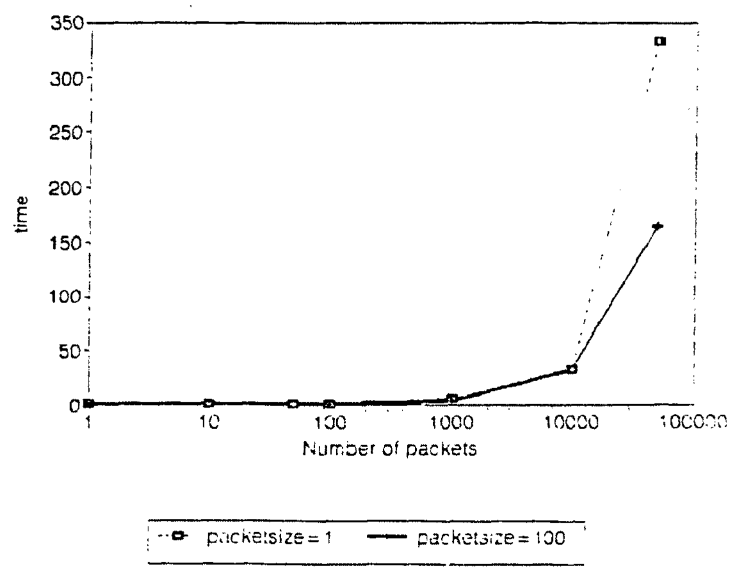
Benchmark 5 (flood.cl):

Benchmark Program 5 measures the impact of flooding the tuple space with messages.

Benchmark Program 5:

```
MAIN
Begin
    Do Until ProgramCrashes
        OUT ( A-Message )
End /* MAIN */
```

This benchmark finds the size of the tuple space. From the results in Table 5 and Graph 5, it can be seen that as the size of the message increases, the number of messages that can be accomodated in the tuple space decreases. The graph is similar to a hyperparabolic



Graph 4

function, where the product of x and y is a constant. This indicates that the size of the tuple space is a constant defined in the implementation as (about) 3,000,000 bytes.

Benchmark 6 (latency.cl):

Benchmark Program 6 measures the average time to exchange messages between two processes running on different machines.

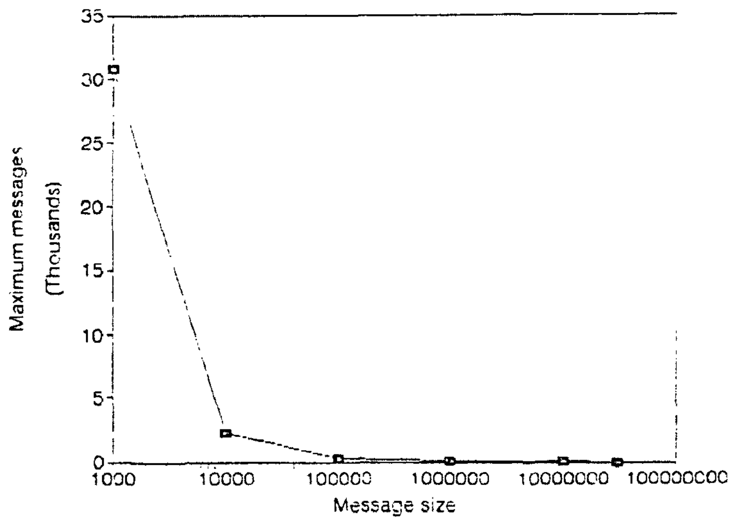
Benchmark Program 6:

-Spawn processes P_1 and P_2.
-P_1 and P_2 exchange N messages.
-The time taken is measured.

Latency, in this context, is the time taken to send a message to a process on a different host and receive a response back from that process. From the results in Table 6, it can be noted that the average latency time between any two processes running on different workstations in a homogeneous environment is the same.

Message size	Maximum Messages
1000	30791
10000	2234
100000	272
1000000	27
10000000	2
30000000	0

Table 5



Graph 5

(For 500 message transfers to and fro)

Machine 1	Machine 2	Time (sec)
Huckle	Tincup	6.3
Schively	Bannack	6.2
Capron	Bighole	6.2

Table 6

Benchmark 7 (distribution.cl):

Benchmark Program 7 measures the time taken to distribute N identical processes from a machine.

Benchmark Program 7:

-Spawn N processes.
-Find out the destination of the spawned processes.
-Determine the time taken.

Discussion:

Distribution of N processes to machines from a given machine will give a distribution of processes as discussed in Chapter 3. This distribution is a measure of the power of the machine, in the sense that a given processor will receive more processes only if it is able to execute them to termination, one by one. From the results in Table 7, it can be seen that the processor "huckle" executes to completion more processes than the rest, indicating that it is more powerful than the rest. The time taken indicates the time taken to distribute and realize the work. From the result, it can be seen that work distribution from "tincup" took less time than the rest.

(Host Machine has no processes spawned on itself)

Number of processes spawned : 140

huckle	tincup	bannack	schively	stillwater	lemhi	capron	bighole	TIME(sec)
0	20	19	20	19	21	20	21	11.599
31	0	17	23	19	19	17	17	12.568
30	19	0	19	19	23	17	15	11.777
28	18	19	0	20	19	19	18	10.613
23	21	21	19	0	20	16	20	11.633
23	21	21	20	21	0	12	21	11.334
22	20	20	18	21	21	0	18	10.755
26	19	20	20	22	22	11	0	12.495

Table 7

4.2 Impact of Quantitative evaluation on ORS Implementation:

- In **ORS** implementation, when messages have to be read, it is preferable to use non-blocking versions (**inp** and **rdp**) than blocking versions (**in** and **rd**). This allows the C-Linda program to handle timeouts in failed read operations without taking additional time.
- While transferring information from one ORB to another. it is generally more efficient to send a few large packets than numerous small packets. However, a very large packet of data can fill the entire tuple space. which can crash the system.
- The size of the tuple space is limited. The sum of information stored in the tuple space, objects and messages, cannot exceed this limit. If the ORS has stored lot of information, then only a part of it can be in the active tuple space and the rest has to be on secondary storage.
- In a homogeneous network, the average time to pass messages between any two processors is the same irrespective of the two processors involved. Further, the distribution

experiment shows that in a homogeneous network with processors of unequal power. any processor can have the OM running on it. For example, in the experimental environment, it is better to have the master on the processor "tincup" than on the more powerful server processor "huckle".

Chapter 5

Summary and Conclusions

5.1 Summary

Chapter 2 describes the ORS design that satisfies all requirements stated in Chapter 1. The design is robust as it can absorb processor and network failures. However, as stated earlier, all the protocols described in the design cannot be directly mapped to the actual implementation in C-Linda. The qualitative and quantitative evaluation showed that it is indeed true. Chapter 3 exposes some of the implementation limitations of C-Linda through a series of experiments. Qualitative evaluation shows that fault tolerance, which is a key note in the design, cannot be achieved with the current C-Linda implementation. Chapter 4, through a set of benchmark programs, shows that message size and tuple space size are constrained by C-Linda which can cause serious problems in ORS implementation. Results garnered from the preceding two chapters give enough hints to direct the actual ORS implementation in C-Linda. For instance, the implementation need not concentrate on having sub-sections of programs to handle processor failures since C-Linda does not tolerate processor failures.

5.2 Prototype implementation of ORS

The prototype system has the following features.

- The system is capable of normal startup, operation, and termination.
- The system is not robust and is prone to processor failures.
- Communication between the user and ORB is through files. The user interacts with an independent program which interacts with the ORB on that machine through files.
- The user can register simple objects and retrieve them.
- The system supports a "super-ORS-user" who can control system behavior and termination.
- The ORB keeps a log of all user commands and the status of OM.

- The OM saves all the objects in the tuple space at regular intervals. It also keeps a log of other system information like ORB alive messages.
- The system also saves all object tuples immediately before normal system termination. Thus, information can be accessed across different ORS sessions.

Given the ORS, one should have a distributed environment to run this software. Network programs generated by C-Linda execute with the following restrictions. All the processors in the distributed system must be listed in the configuration file "tsnet.nodes". Processors cannot be added dynamically to this file during program execution. The program has to be redistributed and executed using **tsnet** in order to have a new configuration for the distributed system. Merely listing processor names in "tsnet.nodes" is not the only requirement for program execution in the distributed environment. There must be a login account with the same name on every machine listed in "tsnet.nodes" file. Further, the login accounts have to set up such that each machine trusts the other (this information has to be given in the file ".rhosts" file in UNIX environment). Once this has been set up, the user on one machine can log on to another machine with the same user name without typing in the password. This environment is strictly required by C-Linda programs because C-Linda internally uses **rsh** command to execute remote processes.

5.3 Conclusion

Design

The ORS design is robust as it can withstand processor failure and network partitioning. The Object Registration System returns back to the normal configuration when processor and/or network fault is rectified.

Implementation

- Abstract Linda is well suited for implementing ORS design because its high level approach to distributed programming hides most of the details of communication and process management. The only limitation it has is directed process to processor mapping, which is easily circumvented by a simple protocol.

- The current C-Linda implementation of abstract Linda is not robust, and cannot provide a robust implementation of the ORS.
- In order to build a robust ORS in real Linda, the following changes need to be made in C-Linda implementation.
  - The **tsnet** utility program which manages the tuple space should be more flexible. Instead of terminating the entire ORS system upon detecting termination of a remote process (ORB), **tsnet** should allow programmer intervention at that point. This concept is very similar to UNIX signal handling, where one can call a function when a signal is raised.
  - C-Linda should allow the use of signals in C-Linda programs. Instead of reserving the signals for system use and thus preventing user handling of those signals, an alternate C-Linda implementation should be provided so that the C-Linda system distinguishes between program invoked signals and its internal signals. Likewise, C-Linda should provide its own version of **malloc** function.

Future Directions:

- Isolation of an ORB instance either due to processor failure or network partitioning leaves the objects registered on the isolated processor inaccessible to user requests from ORBs on other processors. This problem is specific to data management in a distributed system. The ORS is a form of distributed database system where each processor holds a part of the data and no two processors store common information. When a processor fails, all information that is associated with it is lost unless the processor periodically saves data on its secondary storage. The stored data, however, will not be available to the distributed system as long as that processor is down. In order to avoid temporary data loss due to processor failure, objects that are registered through an ORB must be stored on more than one processor so that the objects are still available to the ORS when one processor fails. The data still may not be retrievable though if both the processors fail, though the probability that both fail is much lower than a single processor failure, data redundancy requires special protocols to simulate concurrent updates to replicated objects. These are the issues that are not dealt in the ORS design. Determining the type of data distribution is another major step in having a robust system. Further investigation needs to be done in this direction.

- The structure of objects has to be carefully designed depending upon user needs. This is an essential step in having a working ORS system.
- The user interface, which is a simple command line. can be improved with a window interface.
- Communication between the ORB and the user is established through files in the current version of ORS. This mode of communication is very slow when compared to other forms of communication channels available. Use of shared memory for communication is an alternative method which can be given some thought.

References

1. Srikanth Kambhatla, Jon Inouye, and Jonathan Walpole, *Benchmarking Parallel Machines via a Software Architecture*, Oregon Graduate Institute of Science and Technology. Technical Report No. Cs/E 90-002, January 1990.

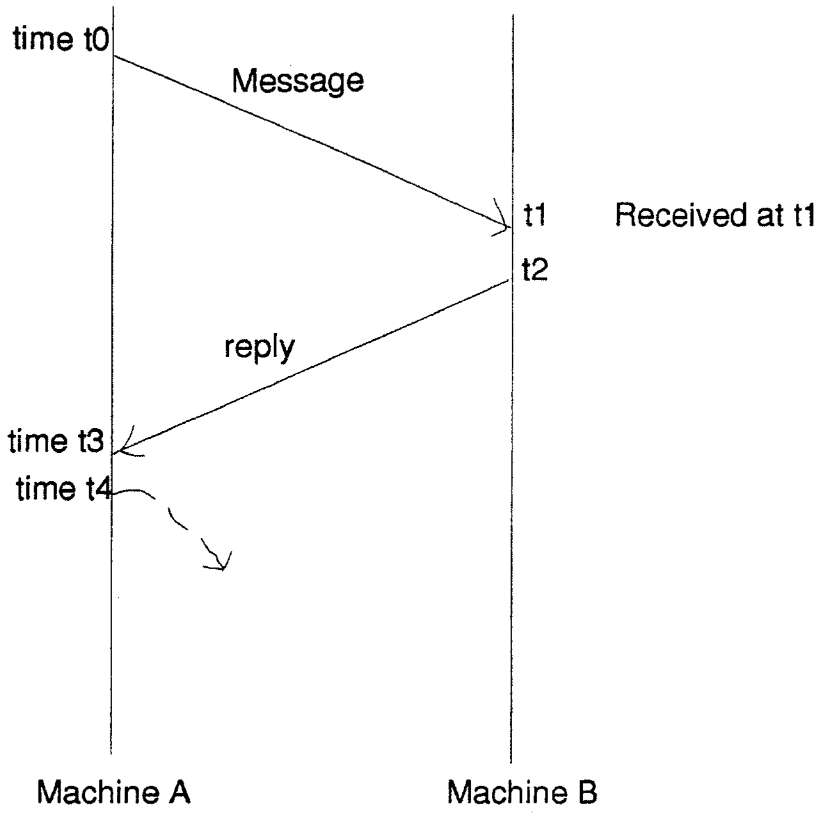
2. Scientific Associates Inc., *C-Linda Reference Manual.* August 1990.

3. Andrew Binstock, *Emerging Standards*, Unix Review.

4. Nicholas Carriero, and David Gelernter, *How to Write Parallel Programs: A First Course* , The MIT Press, Cambridge, MA, 1990.

5. Richard Stevens, *Advanced Programming in the UNIX environment*. Addison-Wesley Publishing Company, Inc., September 1992.

APPENDIX A



(i) At time t0, machine A sends a message to machine B

(ii) Machine B receives the message at time t1

(iii) Machine B sends a reply at time t2

(iv) Machine A receives reply at time t3

(v) Machine A sends another message at time t4 and is lost.

Understanding Timing Diagrams

APPENDIX B

```
/* Benchmark Program 1 */

/* File : primitives.cl
 * Description :
        This program evaluates the cost of doing Linda out(), in()
        rd(), inp(), rdp(), and eval() operations by performing
        size primitive operations of each type. The time is then
        In case of eval(), a null eval which does no processing is
        performed.
 */

real_main(int argc, char ** argv) {
        int i;
        int size;

        if (argc != 2)
                exit();
        size = atoi(argv[1]);

        start_timer(); /* Start the timer */
        for (i = 0; i < size; i++) /* Do size out() operations */
           out("hello");
        timer_split("out"); /* Get the time */
        print_times();      /* print the time */
        start_timer();/* Start the timer */
        for (i = 0; i < size; i++)/* Do size rd() operations */
           rd("hello");
        timer_split("rd");/* Get the time */
        print_times();      /* print the time */
        start_timer();/* Start the timer */
        for (i = 0; i < size; i++)/* Do size in() operations */
           in("hello");
        timer_split("in");/* Get the time */
        print_times();      /* print the time */
        start_timer();/* Start the timer */
        for (i = 0; i < size; i++)/* Do size out() operations */
           out("hello");
        timer_split("out");/* Get the time */
        print_times();      /* print the time */
        start_timer();/* Start the timer */
        for (i = 0; i < size; i++)/* Do size rdp() operations */
           rdp("hello");
        timer_split("rdp");/* Get the time *
        print_times();      /* print the time */
        start_timer();/* Start the timer */
        for (i = 0; i < size; i++)/* Do size inp() operations */
           inp("hello");
        timer_split("inp");/* Get the time */
        print_times();      /* print the time */
        start_timer();/* Start the timer */
        for (i = 0; i < size; i++)/* Do size eval() operations */
           eval();
        timer_split("eval");/* Get the time */
        print_times();      /* print the time */
}
```

```
/* Benchmark Program 2 */

/* actuals.cl */

real_main(int argc, char** argv) {
        int i,size,type1;
        int worker(int);

        if (argc != 2) {
          printf ("Usage : %s FormalNum (1..7)\n",argv[0]);
        }
        type1 = atoi(argv[1]);
        eval(worker(type1));
}

int worker(int type) {

        int i , j;
              for (j = 0; j < 10; j++) {

                start_timer();
                if (type = 1)
                for (i = 0; i < 1000 ; i++)
                        out(1);
                else if (type = 2)
                for (i = 0; i < 1000 ; i++)
                        out(1,2);
                else if (type = 3)
                for (i = 0; i < 1000 ; i++)
                        out(1,2,3,4);
                else if (type = 4)
                for (i = 0; i < 1000 ; i++)
                   out(1,2,3,4,5,6,7,8);
                else if (type = 5)
                for (i = 0; i < 1000 ; i++)
                   out(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
                else if (type = 6)
                for (i = 0; i < 1000 ; i++)
                  out(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
             17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32);
                else if (type = 7)
                for (i = 0; i < 1000 ; i++)
                   out(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
                   16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,
                   31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,
                   46,47,48,49,50,51,52,53,54,55,56,57,58,59,
                   60,61,62,63,64);
                else
                for (i = 0; i < 1000 ; i++)
                        out(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
                16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,
                33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,
                50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,1,2,3,
                4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
                23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,
                40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,
                57,58,59,60,61,62,63,64,1,2,3,4,5,6,7,8,9,10,11,12,
                13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,
                30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,
                47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,
                64,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,
                20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,
                37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,
                54,55,56,57,58,59,60,61,62,63,64,1,2,3,4,5,6,7,8,9,
                10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
                27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,
    44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,
    61,62,63,64,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
    17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,
    34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,
    51,52,53,54,55,56,57,58,59,60,61,62,63,64,1,2,3,4,5,
    6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,
    25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,
    42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,
    59,60,61,62,63,64,1,2,3,4,5,6,7,8,9,10,11,12,13,14,
    15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
    32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,
    49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64);
    timer_split("Out done");
    if (type = 1)
    for (i = 0; i < 1000 ; i++)
            rd(1);
    else if (type = 2)
    for (i = 0; i < 1000 ; i++)
            rd(1,2);
    else if (type = 3)
    for (i = 0; i < 1000 ; i++)
            rd(1,2,3,4);
    else if (type = 4)
    for (i = 0; i < 1000 ; i++)
            rd(1,2,3,4,5,6,7,8);
    else if (type = 5)
    for (i = 0; i < 1000 ; i++)
            rd(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
    else if (type = 6)
    for (i = 0; i < 1000 ; i++)
      rd(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32);
    else if (type = 7)
    for (i = 0; i < 1000 ; i++)
      rd(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
      17,18,19,20,21,22,23,24,25,26,27,28,29,30,
      31,32,33,34,35,36,37,38,39,40,41,42,43,44,
      45,46,47,48,49,50,51,52,53,54,55,56,57,58,
      59,60,61,62,63,64);
    else
    for (i = 0; i < 1000 ; i++)
      rd(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
        17,18,19,20,21,22,23,24,25,26,27,28,29,30,
        31,32,33,34,35,36,37,38,39,40,41,42,43,44,
        45,46,47,48,49,50,51,52,53,54,55,56,57,58,
        59,60,61,62,63,64,1,2,3,4,5,6,7,8,9,10,11,
        12,13,14,15,16,17,18,19,20,21,22,23,24,25,
        26,27,28,29,30,31,32,33,34,35,36,37,38,39,
        40,41,42,43,44,45,46,47,48,49,50,51,52,53,
        54,55,56,57,58,59,60,61,62,63,64,1,2,3,4,5,
        6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,
        22,23,24,25,26,27,28,29,30,31,32,33,34,35,
        36,37,38,39,40,41,42,43,44,45,46,47,48,49,
        50,51,52,53,54,55,56,57,58,59,60,61,62,63,
        64,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
        17,18,19,20,21,22,23,24,25,26,27,28,29,30,
        31,32,33,34,35,36,37,38,39,40,41,42,43,44,
        45,46,47,48,49,50,51,52,53,54,55,56,57,58,
        59,60,61,62,63,64,1,2,3,4,5,6,7,8,9,10,11,
        12,13,14,15,16,17,18,19,20,21,22,23,24,25,
        26,27,28,29,30,31,32,33,34,35,36,37,38,39,
        40,41,42,43,44,45,46,47,48,49,50,51,52,53,
        54,55,56,57,58,59,60,61,62,63,64,1,2,3,4,5,
        6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,
        22,23,24,25,26,27,28,29,30,31,32,33,34,35,
        36,37,38,39,40,41,42,43,44,45,46,47,48,49,
          50,51,52,53,54,55,56,57,58,59,60,61,62,63,
          64,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
          17,18,19,20,21,22,23,24,25,26,27,28,29,30,
          31,32,33,34,35,36,37,38,39,40,41,42,43,44,
          45,46,47,48,49,50,51,52,53,54,55,56,57,58,
          59,60,61,62,63,64,1,2,3,4,5,6,7,8,9,10,11,
          12,13,14,15,16,17,18,19,20,21,22,23,24,25,
          26,27,28,29,30,31,32,33,34,35,36,37,38,39,
          40,41,42,43,44,45,46,47,48,49,50,51,52,53,
          54,55,56,57,58,59,60,61,62,63,64);
       timer_split("Read done");
       if (type = 1)
       for (i = 0; i < 1000 ; i++)
               in(1);
       else if (type = 2)
       for (i = 0; i < 1000 ; i++)
               in(1,2);
       else if (type = 3)
       for (i = 0; i < 1000 ; i++)
               in(1,2,3,4);
       else if (type = 4)
       for (i = 0; i < 1000 ; i++)
               in(1,2,3,4,5,6,7,8);
       else if (type = 5)
       for (i = 0; i < 1000 ; i++)
               in(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
       else if (type = 6)
       for (i = 0; i < 1000 ; i++)
         in(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
        17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32);
       else if (type = 7)
       for (i = 0; i < 1000 ; i++)
               in(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
         17,18,19,20,21,22,23,24,25,26,27,28,29,30,
         31,32,33,34,35,36,37,38,39,40,41,42,43,44,
         45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,
         60,61,62,63,64);
    else
    for (i = 0; i < 1000 ; i++)
       in(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
       17,18,19,20,21,22,23,24,25,26,27,28,29,30,
       31,32,33,34,35,36,37,38,39,40,41,42,43,44,
       45,46,47,48,49,50,51,52,53,54,55,56,57,58,
       59,60,61,62,63,64,1,2,3,4,5,6,7,8,9,10,11,
       12,13,14,15,16,17,18,19,20,21,22,23,24,25,
       26,27,28,29,30,31,32,33,34,35,36,37,38,39,
       40,41,42,43,44,45,46,47,48,49,50,51,52,53,
       54,55,56,57,58,59,60,61,62,63,64,1,2,3,4,5,
       6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,
       22,23,24,25,26,27,28,29,30,31,32,33,34,35,
       36,37,38,39,40,41,42,43,44,45,46,47,48,49,
       50,51,52,53,54,55,56,57,58,59,60,61,62,63,
       64,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
       17,18,19,20,21,22,23,24,25,26,27,28,29,30,
       31,32,33,34,35,36,37,38,39,40,41,42,43,44,
       45,46,47,48,49,50,51,52,53,54,55,56,57,58,
       59,60,61,62,63,64,1,2,3,4,5,6,7,8,9,10,11,
       12,13,14,15,16,17,18,19,20,21,22,23,24,25,
       26,27,28,29,30,31,32,33,34,35,36,37,38,39,
       40,41,42,43,44,45,46,47,48,49,50,51,52,53,
       54,55,56,57,58,59,60,61,62,63,64,1,2,3,4,5,
       6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,
       22,23,24,25,26,27,28,29,30,31,32,33,34,35,
       36,37,38,39,40,41,42,43,44,45,46,47,48,49,
       50,51,52,53,54,55,56,57,58,59,60,61,62,63,
       64,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
                17,18,19,20,21,22,23,24,25,26,27,28,29,30,
                31,32,33,34,35,36,37,38,39,40,41,42,43,44,
                45,46,47,48,49,50,51,52,53,54,55,56,57,58,
                59,60,61,62,63,64,1,2,3,4,5,6,7,8,9,10,11,
                12,13,14,15,16,17,18,19,20,21,22,23,24,25,
                26,27,28,29,30,31,32,33,34,35,36,37,38,39,
                40,41,42,43,44,45,46,47,48,49,50,51,52,53,
                54,55,56,57,58,59,60,61,62,63,64);
                timer_split("In done");
                print_times();
        }
           return type;
    }
```

```
/* Benchmark Program 3 */

/* formals.cl */

real_main(int argc, char** argv) {
        int i , j;
        int type,count;
        int f;

        if (argc != 3) {
         printf ("Usage : %s FormalNum (1..4) actuals \n",
                                  argv[0]);
         exit(1);
        }
        type = atoi(argv[1]);
        count = atoi(argv[2]);
                for (j = 0; j < 5 ; j++) {
                start_timer();
                if (type = 1)
                for (i = 0; i < 1000 ; i++)
                        out(1);
                else if (type = 2)
                for (i = 0; i < 1000 ; i++)
                        out(1,2);
                else if (type = 3)
                for (i = 0; i < 1000 ; i++)
                        out(1,2,3,4);
                else if (type = 4)
                for (i = 0; i < 1000 ; i++)
                        out(1,2,3,4,5,6,7,8);
                timer_split("Out done");
                if (type = 1)
                for (i = 0; i < 1000 ; i++) {
                     if (count == 0)
                        rd(1);
                     else
                        rd(?f);
                }
                else if (type = 2)
                for (i = 0; i < 1000 ; i++) {
                  if (count == 0)
                        rd(1,2);
                  else if (count == 1)
                        rd(?f,2);
                  else
                        rd(?f,?f);
                }
                else if (type = 3)
                for (i = 0; i < 1000 ; i++) {
                  switch(count) {
                        case 0 : rd(1,2,3,4); break;
                        case 1 :rd(?f,2,3,4); break;
                        case 2 :rd(?f,?f,3,4);break;
                        case 3 :rd(?f,?f,?f,4);break;
                        default :rd(?f,?f,?f,?f);
                   }
                }
                else if (type = 4)
                for (i = 0; i < 1000 ; i++) {
                switch(count) {
                case 0: rd(1,2,3,4,5,6,7,8); break;
                case 1: rd(?f,2,3,4,5,6,7,8); break;
                case 2: rd(?f,?f,3,4,5,6,7,8); break;
                case 3: rd(?f,?f,?f,4,5,6,7,8); break;
                case 4: rd(?f,?f,?f,?f,5,6,7,8); break;
                case 5: rd(?f,?f,?f,?f,?f,6,7,8); break;
```

```
            case 6: rd(?f,?f,?f,?f,?f,?f,7,8); break;
            case 7: rd(?f,?f,?f,?f,?f,?f,?f,8); break;
            default:rd(?f,?f,?f,?f,?f,?f,?f,f);
            }
            }
            timer_split("Read done");
            if (type = 1)
            for (i = 0; i < 1000 ; i++) {
                 if (count == 0)
                    in(1);
                 else
                    in(?f);
            }
            else if (type = 2)
            for (i = 0; i < 1000 ; i++) {
              if (count == 0)
                    in(1,2);
              else if (count == 1)
                    in(?f,2);
              else
                    in(?f,?f);
            }
            else if (type = 3)
            for (i = 0; i < 1000 ; i++) {
              switch(count) {
                    case 0 : in(1,2,3,4); break;
                    case 1 :in(?f,2,3,4); break;
                    case 2 :in(?f,?f,3,4);break;
                    case 3 :in(?f,?f,?f,4);break;
                    default :in(?f,?f,?f,?f);
               }
            }
            else if (type = 4)
            for (i = 0; i < 1000 ; i++) {
            switch(count) {
            case 0 :        in(1,2,3,4,5,6,7,8); break;
            case 1 :        in(?f,2,3,4,5,6,7,8); break;
            case 2 :        in(?f,?f,3,4,5,6,7,8); break;
            case 3 :        in(?f,?f,?f,4,5,6,7,8); break;
            case 4 :        in(?f,?f,?f,?f,5,6,7,8); break;
            case 5 :        in(?f,?f,?f,?f,?f,6,7,8); break;
            case 6 :        in(?f,?f,?f,?f,?f,?f,7,8); break;
            case 7 :        in(?f,?f,?f,?f,?f,?f,?f,8); break;
            default:in(?f,?f,?f,?f,?f,?f,?f,f);
            }
            }
            timer_split("In done");
            print_times();
        }
}
```

```
/* Benchmark Program 4 */

/* message.cl */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

real_main(int argc, char** argv) {
        int i,try,status,size,freq;
        char name[30], host[30];
        int worker(int,int,int);

        if (argc != 3) {
                printf("Usage : %s SIZE FREQ\n",argv[0]);
                exit(0);
        }
        size = atoi(argv[1]);
        freq = atoi(argv[2]);
        eval(worker(0,size,freq));
        eval(worker(1,size,freq));
        out("Start");
}

int worker(int id,int SIZE,int FREQ) {
        int i,len,type;
        FILE *fptr;
        char *buffer;

        buffer = (char *)malloc(SIZE);
                if (id == 0) { /* the Source worker */
                system("getload");
                for(i = 0; i < SIZE - 1 ; i++)
                        buffer[i] = 'A';
                buffer[i] = '\0';
                out("source-ready");
                in("dest-ready");
                rd("Start");
                start_timer();
                for ( i = 0; i < FREQ ; i++)
                        out("file",buffer:SIZE);
                timer_split("Writing DONE");
                print_times();
                }
                else { /* the Destination worker */
                system("getload");
                in("source-ready");
                out("dest-ready");
                rd("Start");
                start_timer();
                for ( i = 0; i < FREQ ; i++)
                        in("file",?buffer:SIZE);
                timer_split("Reading DONE");
                print_times();
                }
        return id;
}
```

```
/* Benchmark Program 5 */

/* flood.cl */

real_main() {

long i;
char buffer[20000000];
long j;
long size;


        if (argc != 2) {
                printf("Usage: %s <size>\n",argv[0]);
                exit(1);
        }
        j = atol(argv[1]);
        i =0;
        while(1) {
                i++;
                out(buffer:j);
                printf("%d Successful\n",i);
        }
}
```

```
/* Benchmark Program 6 */

/* File : latency.cl
 * Description :
        This program evaluates the communication overhead in
    sending and receving messages between two machines
 */
real_main() {
        int ping(int),pong(int);
        char host[30],host1[30];
        int i,len;
        for (i = 0; i < 10; i++) {
        eval(ping(0));
        eval(pong(1));
        in("host",?host:len);
        in("host",?host1:len);
        printf("The processes are on %s and %s\n",
                                        host,host1);

        in(?int);
        in(?int);
        }
}

int ping(int id) {
        int i;
        char host[30];

        gethostname(host);
        out("host",host:30);
        start_timer();
        for (i = 0; i < 500; i++)
                {
                out("catch");
                in("throw");
                }
        timer_split("Done");
        print_times();
        return id;
}


int pong(int id) {
        int i;
        char host[30];

        gethostname(host);
        out("host",host:30);
        for (i = 0; i < 500; i++)
                {
                in("catch");
                out("throw");
                }
        return id;
}
```

```
/* Benchmark Program 7 */

/* File : distribution.cl
 * Description :
   This program distributes 120 similar processes to all the
   processors/machines in the distributed environment. Since
   there is no directed mapping onto a machine, the mapping
   is left to the program itself and during run-time, the
   actual mapping to the processor is obtained. Finally, the
   distribution stattistics is obtained.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>


real_main() {
char hostname[30];
int i,j,k,len;
int worker(int);
int tincup,bannack,schively,stillwater,
        lemhi,capron,bighole,huckle;
        gethostname(hostname);
        printf("Result on %s \n",hostname);
        printf(" huckle tincup  bannack
          schively stillwater lemhi capron bighole\n");
        for(k = 0; k < 5; k++) {
        start_timer(); /* start the timer */
        tincup = bannack = schively = stillwater = 0;
        lemhi = capron =bighole= 0;
        huckle = 0;
        for(i = 0; i < 140; i++)
           /* Spawn the worker() */
                eval(worker(i));
        for(i = 0; i < 140; i++) {
                in("worker",?j,?hostname:len);
        /* Get the name of the machine on which the
                                  worker is spawned */
        if (hostname[0] == 't')  /* Machine is tincup */
                tincup++;
        else if (hostname[3] == 'n') /* bannack */
                bannack++;
        else if (hostname[2] == 'g') /* bighole */
                bighole++;
        else if (hostname[0] == 'l') /* lemhi */
                lemhi++;
        else if (hostname[0] == 'c') /*  capron */
                capron++;
        else if (hostname[1] == 'c') /* schively */
                schively++;
        else if (hostname[1] == 't') /* stillwater */
                stillwater++;
        else if (hostname[0] == 'h') /* huckle */
                huckle++;
        }
        for(i = 0; i < 140; i++)
                in(?int);
        timer_split("Done."); /* stop the timer */
 /* print the time taken to do the distribution */
        print_times();
    printf(" %3d   %3d    %3d     %3d     %3d
             %3d     %3d   %3d\n",
     huckle,tincup,bannack,schively,stillwater,
                        lemhi,capron,bighole);
}


/* This is the worker process */
worker(int id) {
char hostname[30];

 /* get the name of the machine on which it is mapped */
gethostname(hostname);
 /* send that to the master */
out("worker",id,hostname:30);
return id;
}
```

APPENDIX C

```
/* Program 1 */

/* File : eval.cl
 * Description :
 *      This program illustrates the nature of eval operation in
 *      C-Linda.
 */

/* The Master program */
real_main() {

        int worker(int);
        char host[30];
        int len;

        /* Spawn a worker using an eval operation */
        eval(worker(1));

        /* Get the name of the host machine on
                    which the master is running */
        gethostname(host,30);
        printf("The Host machine is : %s \n",host);

        /* Get the name of the destination
                      machine from the worker */
        in(?host:len);

        /* print the name of the machine */
        printf("The worker was mapped on : %s \n",host);

        /* read the passive tuple created
                        after the worker terminates */
        in(?int);
}


/*The worker process */
int worker(int id){

        char host[30];

    /* The worker process gets spawned by the master
       by an eval operation. As soon as it reaches a
       destination machine, it gets the name of the machine */
        gethostname(host,30);

   /* The worker sends the destination machine
                          name to the master */
        out(host:30);

        /* The worker terminates and thus the worker
           process becomes a  passive data tuple */
        return id;
}

/* The following machines were listed in the tsnet.nodes file

 * tincup.cs.umt.edu
 * schively.cs.umt.edu
 * capron.cs.umt.edu
 * bighole.cs.umt.edu
 * bannack.cs.umt.edu
 * lemhi.cs.umt.edu
 * stillwater.cs.umt.edu

     Linda tries to map a process spawned by an
     eval operation on one of these machines.
 */

/************************ RESULTS *************************/

/*Run 1 */
The Host machine is : schively.cs.umt.edu
The worker was mapped on : bighole.cs.umt.edu

/*Run 2 */
The Host machine is : schively.cs.umt.edu
The worker was mapped on : bighole.cs.umt.edu

/* When this program was executed several times on schively,
   it consistently spawned the worker process on bighole
   (Notice that bighole is middle element in the tsnet.nodes file).
 */

/* When the same program was run on bighole with the same
   tsnet.nodes file, the worker process was spawned on
   capron (Note that capron is just above the middle element
   bighole. The Linda eval operation did not spawn a process
   on the host machine
 */

The Host machine is : bighole.cs.umt.edu
The worker was mapped on : capron.cs.umt.edu

/*************************   END   *************************/
```

```
/* Program 2 */

/* File multipleeval.cl
 * Description :
        This program shows the behaviour of Linda eval when
 *      multiple eval operations are carried out
 */

/* The master program */
real_main() {
        int worker(int);
        char host[30];
        int i , len;

        /* spawn six workers and get their
                destination processor names */

        eval(worker(1));
        in(?host:len);
        printf("The worker was mapped on : %s \n",host);

        eval(worker(2));
        in(?host:len);
        printf("The worker was mapped on : %s \n",host);

        eval(worker(3));
        in(?host:len);
        printf("The worker was mapped on : %s \n",host);

        eval(worker(4));
        in(?host:len);
        printf("The worker was mapped on : %s \n",host);

        eval(worker(5));
        in(?host:len);
        printf("The worker was mapped on : %s \n",host);

        eval(worker(6));
        in(?host:len);
        printf("The worker was mapped on : %s \n",host);

  /* Master gives permission for the workers to terminate */
        for (i = 0; i < 6 ; i++)
                out("perm");

  /* The master collects all the passive data tuples
        created by workers after termination */
        for (i = 0; i < 6 ; i++)
                in(?int);
}


/* The worker process */
int worker(int id){

        char host[30];

        /* Get the name of the machine on which the
                        worker is mapped */
        gethostname(host,30);

   /* send the destination machine name to the master */
        out(host:30);

   /* Wait for permission from the master to terminate */
        in("perm");
        /* terminate the worker process */
        return id;
}

/********************* RESULTS *************************/
/*Run 1 */

The worker was mapped on : bighole.cs.umt.edu
The worker was mapped on : capron.cs.umt.edu
The worker was mapped on : tincup.cs.umt.edu
The worker was mapped on : lemhi.cs.umt.edu
The worker was mapped on : stillwater.cs.umt.edu
The worker was mapped on : bannack.cs.umt.edu

/*Run 2 */

The worker was mapped on : bighole.cs.umt.edu
The worker was mapped on : tincup.cs.umt.edu
The worker was mapped on : capron.cs.umt.edu
The worker was mapped on : stillwater.cs.umt.edu
The worker was mapped on : bannack.cs.umt.edu
The worker was mapped on : lemhi.cs.umt.edu

/*Run 3 */

The worker was mapped on : bighole.cs.umt.edu
The worker was mapped on : lemhi.cs.umt.edu
The worker was mapped on : stillwater.cs.umt.edu
The worker was mapped on : bannack.cs.umt.edu
The worker was mapped on : tincup.cs.umt.edu
The worker was mapped on : capron.cs.umt.edu

/* From the results, it can be seen that the destination
   of the first eval operation can be predicted. The
   destination for the successive eval opearations is
   randomly selected (Note that the same processor is
 * not selected twice */

/************************    END    *************************/
```

```
/* Program 3 */

/* File : directed_eval.cl
 * Description :
     This program uses C-Linda primitives to do a directed
 *                processor to process mapping.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

real_main() {
        char name[30];
        int try;

        gethostname(name,30);
        printf("The Master is running on : %s \n",name);
        do {
        printf("Enter the name of the Target machine :");
        do {
        gets(name);
        } while(strlen(name) < 2);
        if (strcmp(name,"quit") == 0) exit(1);
        try = directed_eval(name,10);
        if (try < 0) printf("I failed\n");
        else printf("I succeeded\n");
        } while(1);
}

/* This function simulates directed eval in Linda.
 *
 * It makes "retry" number of attempts to get a process
   on the target processor "name". The function returns
   a negative number if it fails to do a directed eval */

int directed_eval(char *name , int retry){
        int try,status;
        int worker(int);
        char host[30];

        try = 0;

  /* output the destination machine name to the tuple space */
        out("target",name:strlen(name));

  /* Keep spawning a worker process until it reaches the
   * destination or until the number of retries exceeds the
   * specified limit */
        do {
        /* spawn a worker process using eval function */
                eval(worker(try));

        /* get the status of eval from the worker */
        /* The worker returns 1 if it got mapped on to the
         * correct destination. Or else it returns a 0,
         * indicating failure */
                in("status",try,?status,?host);

        /* If success, then exit the loop */
                if (status == 1) break;

                try++;
        } while(try < retry);
        if (try == retry)
                return -1;
        else
                return try;
}
```

```
/* This is the worker process that gets spawned
                randomly onto a processor */
int worker(int id) {
        char host[30],target[30];
        int len;

        /* The worker gets the name of the processor
           on which it gets spawned. */
        gethostname(host,30);

     /* It reads the actual destination machine name */
        in("target",?target:len);

    /* If the current host name is not the destination,
       send a failure signal to the process that evaled
       this worker process */
        if (strncmp(target,host,len) != 0) {
                out("status",id,0,host);
                out("target",target:len);
        }
   /* If the current host is the destination,
                indicate success */
        else {
                out("status",id,1,host);
        }
        return id;
}

/************************* RESULTS ************************/

The Master is running on : schively.cs.umt.edu

/* Any attempt to spawn the process on the
                        host machine fails */

Enter the name of the Target machine :schively
I failed

Enter the name of the Target machine :schively
I failed

Enter the name of the Target machine :bannack
I succeeded

Enter the name of the Target machine :tincup
I succeeded

Enter the name of the Target machine :stillwater
I succeeded

Enter the name of the Target machine :capron
I succeeded

Enter the name of the Target machine :lemhi
I succeeded

Enter the name of the Target machine :bighole
I succeeded


/***********************   END   **********************/
```

```
/*Program 4 */
```

```
/* This program demonstrates how to spawn
                processes on machines*/

#define NUM 5

real_main() {
        int worker(int);
        int i,id;

        /* There are 7 processors (including the
                host machine) listed in tsnet.nodes file */

 /* Spawn NUM processes using the eval operation */
        for (i = 1; i <= NUM; i++)
            eval(worker(i));

        /* Read messages from all workers */
        for (i = 1; i <= NUM; i++)
                in("worker");

     /* Send a terminate message to all workers. At this point,
      * the master is absolutely certain that all the worker
      * processors have reached a processor */
        for (i = 1; i <= NUM; i++)
                in("terminate");

    /* Get the passive data tuple created after the
                        worker terminates */
        for (i = 1; i <= NUM; i++) {
            in(?id);
            printf("Worker %d terminated\n",id);
        }
}

/* This is the worker process */
int worker(int id) {

        /* Send a message to the master indicating that
              it reached a destination machine */
        out("worker");

        /* wait for terminate message from the master */
        in("terminate");
        return id;
}

/************************* RESULTS **********************/

/* Sample run  when NUM is 5*/
Worker 2 terminated
Worker 3 terminated
Worker 1 terminated
Worker 5 terminated
Worker 4 terminated

/* Another sample run  when NUM is 5*/
Worker 1 terminated
Worker 3 terminated
Worker 4 terminated
Worker 2 terminated
Worker 5 terminated

/* It is interesting to note that the program never
```

```
   terminates when the number of evals exceed the number
   of active machines on which process can be evaluated
   (excluding the host machine). */

/***************************    END    *************************/
```

```
/* Program 5 */

/*This program demonstrates how multiple processes
       spawned map onto the machines*/

/* The program is run on bannack.cs.umt.edu (The Master) and
   the workers are spawned on all machines(if possible) except
   the host(Master) machine */

#include <string.h>
real_main() {
        int worker(int);
        int i,id,j,k,len;
        char host_name[30];
        int count = 0;

        /* Spawn 36 processes. Since all the machines are
           of equal CPU power and have similar load conditions,
           it is assumed that each machine gets 6 processes */
        for (i = 0; i < 36; i++)
            eval(worker(i));
        do {
        j = 0;
        /* get the mapping of all processes that are
                                 currently mapped */
        for (i = 0; i < 10 ; i++) {
                sleep(5); /* Give sufficient time for
                                 the workers to settle */
                if (inp("Host",?id,?host_name:len) == 1) {
                  count++;
                  j++;
                  printf("Worker %d evaluated on %s\n",
                                 id,host_name);
                }
        }
        /* this print statement demarcates the
                         set of active processes */
   printf("-------------------------------------------------------\n");
   /* send terminate signal to all those active processes */
        for (k = 0; k < j; k++) {
             out("terminate");
             in(?int);
        }
        } while(count < 36);
                /* Execute this loop till you get the
                  whereabouts of all processes spawned */
}

/* the worker process */
int worker(int id) {
        extern char* get_host();
        char host_name[30];

   /* Get the name of the machine on which I am spawned */
        strcpy(host_name,get_host());
        /* send that message to the master */
        out("Host",id,host_name:30);
   /* Wait till the master sends a terminate message */
        in("terminate");
        return id;
}

/* clearly the sample run shows that all the 36 processes
   are not simultaneously evaluated on 6 machines. It has to
   be noted that the work is distributed among only 5 machines
   The host machine never gets involved in the eval operation).
   Each machine handles exactly one process at a given time.
   The rest of the evaled, but not mapped, processes wait for
   a process to terminate. The buffer where the evaled processes
   wait is clearly a stack. The LIFO structure clearly has a
   disadvantage since it provides ample scope for starvation if
   the eval is carried out continuously, such that the size of
   the stack, on an average, remains constant. Under this
   condition, the process, that was evaluated in the
   beginning,(but was unable to catch the first flight !)
   remains at the bottom of the stack  forever */


/* ************  Sample run *************

/* In the first batch, the first five evaluated processes
   get mapped to a processor */

Worker 4 evaluated on tincup.cs.umt.edu

Worker 3 evaluated on capron.cs.umt.edu

Worker 2 evaluated on bighole.cs.umt.edu

Worker 1 evaluated on schively.cs.umt.edu
Worker 0 evaluated on stillwater.cs.umt.e
-----------------------------------------------
/* By the time the first batch completes its work, the
   remaining processes spawned by eval are waiting in the
   tuple space to get mapped to a processor. Since the
   processes evaled last get mapped to processes first,
   it can be concluded that the evaled processes are
   placed on a stack */

Worker 31 evaluated on tincup.cs.umt.edu

Worker 32 evaluated on capron.cs.umt.edu

Worker 33 evaluated on bighole.cs.umt.edu

Worker 34 evaluated on schively.cs.umt.edu
Worker 35 evaluated on stillwater.cs.umt.e
-----------------------------------------------
Worker 26 evaluated on tincup.cs.umt.edu

Worker 27 evaluated on bighole.cs.umt.edu

Worker 28 evaluated on capron.cs.umt.edu

Worker 29 evaluated on schively.cs.umt.edu
Worker 30 evaluated on stillwater.cs.umt.e
-----------------------------------------------
Worker 21 evaluated on tincup.cs.umt.edu

Worker 22 evaluated on bighole.cs.umt.edu

Worker 23 evaluated on capron.cs.umt.edu

Worker 24 evaluated on schively.cs.umt.edu
Worker 25 evaluated on stillwater.cs.umt.e
-----------------------------------------------
Worker 16 evaluated on tincup.cs.umt.edu

Worker 17 evaluated on bighole.cs.umt.edu

Worker 18 evaluated on capron.cs.umt.edu
Worker 19 evaluated on schively.cs.umt.edu
Worker 20 evaluated on stillwater.cs.umt.e
-----------------------------------------------
Worker 11 evaluated on tincup.cs.umt.edu

Worker 12 evaluated on bighole.cs.umt.edu

Worker 13 evaluated on capron.cs.umt.edu

Worker 14 evaluated on schively.cs.umt.edu
Worker 15 evaluated on stillwater.cs.umt.e
-----------------------------------------------
Worker 6 evaluated on tincup.cs.umt.edu

Worker 7 evaluated on bighole.cs.umt.edu

Worker 8 evaluated on capron.cs.umt.edu

Worker 9 evaluated on schively.cs.umt.edu
Worker 10 evaluated on stillwater.cs.umt.e
-----------------------------------------------
Worker 5 evaluated on stillwater.cs.umt.e
-----------------------------------------------
******************************   END ******************/
```

```
/* Program 6 */

/*This program shows  how to spawn exactly one
                            process on a processor */

/* The  processors listed in tsnet.nodes file are :
tincup.cs.umt.edu
schively.cs.umt.edu (Host machine)
capron.cs.umt.edu
bighole.cs.umt.edu
bannack.cs.umt.edu
lemhi.cs.umt.edu
stillwater.cs.umt.edu
*/

#include <string.h>

/* The Master program */
real_main() {
        int worker(int);
        int i,id,len;
        char host_name[30];

        for (i = 1; i <= 6; i++)
            eval(worker(i));
        for (i = 1; i <= 6; i++) {
         in("Host",?id,?host_name:len);
         printf("Worker %d mapped on host %s\n",id,host_name);
        }

        /* terminate all the worker processes */
        for (i = 1; i <= 6; i++)
                out("terminate");

        /* Clean up the passive data tuples */
        for (i = 1; i <= 6; i++)
                in(?int);
}

/* The worker process */
int worker(int id) {
        char host_name[30];

    /* get the name of the machine to which it is mapped */
        gethostname(host_name,30);
        out("Host",id,host_name:30);

    /* wait for the terminate message from the master */
        in("terminate");

        return id;
}

/************************* RESULTS  ************************/

/*Sample run 1*/

Worker 4 mapped on host bighole.cs.umt.edu
Worker 5 mapped on host capron.cs.umt.edu
Worker 6 mapped on host lemhi.cs.umt.edu
Worker 1 mapped on host tincup.cs.umt.edu
Worker 2 mapped on host stillwater.cs.umt.edu
Worker 3 mapped on host bannack.cs.umt.edu

/*Sample run 2*/
```

```
Worker 2 mapped on host lemhi.cs.umt.edu
Worker 1 mapped on host bighole.cs.umt.edu
Worker 6 mapped on host capron.cs.umt.edu
Worker 3 mapped on host tincup.cs.umt.edu
Worker 5 mapped on host stillwater.cs.umt.edu
Worker 4 mapped on host bannack.cs.umt.edu

/************************    END    ********************/
```

```
/* Program 7 */

/* File : usercommn.cl
 * Description :
 *      This program tests the input-output communication
 *      abilities of a evaled process (ORB) with the
 *      external world */

#include <stdio.h>

/*The Master program (OM) */
real_main() {
        int worker(int);
  /* Spawn/eval a worker (ORB) */
        eval(worker(0));
        in(?int);   /* read the passive tuple generated
                        when the worker terminates */
}


/* The Worker Process (ORB) */
int worker(int id) {

        char message[40];

  /* This command will make this line appear on the screen
   immediately. If this line is not used, the messages
   sent out by the worker process are not flushed out
   immediately, but are held in a buffer to be fulshed
   out when the program terminates or when the buffer
   becomes full */

        setlinebuf(stdout);

   /* Upon reaching a machine or processor Pi, the
      worker(ORB) executes the following commands */

  /* The worker prints a prompt on the screen. Ideally, this
   message should appear on the screen associated with
          processor Pi. */

        printf("Enter your name please ! :");
   /* The worker then reads a message from the user.
      The preferred situation is that the input from
      user on processor Pi is read in */
        gets(message);

   /* The message is then printed back on the screen */
        printf("Your name is %s\n",message);

   /* The worker process (ORB) now terminates, thus forming a
   passive integer data tuple which is eventually collected
   by the waiting master process */
        return id;
}


/************************* RESULTS *************************/
```

The program shown above never terminates because, the worker is unable to read a message either from the user on the local host (The machine on which the master is running) or ther remote host (The machine on which the spawned process gets mapped).

However, the output from the worker appears on the local machine.

```
/* Program 8 */

/* File : filecommn.cl
 * Description :
 *      This program illustrates that an evaled process can
 * communicate with the remote user through files */

#include <stdio.h>

/*The Master program (OM) */
real_main() {
        int worker(int);
 /* Spawn/eval a worker (ORB) */
        eval(worker(0));
        in(?int);   /* read the passive tuple generatedwhen
                        the worker terminates */
}


/* The Worker Process (ORB) */
int worker(int id) {

        char message[40];
        char host[30];
        char infile[50],outfile[50];
        FILE  *inptr,*outptr;


        gethostname(host,30);
        sprintf(infile,"%s.in",host);

        /* Open the in file */
        inptr = fopen(infile,"r");
        sprintf(outfile,"%s.out",host);

        /* Open the out file */
        outptr = fopen(outfile,"w");

        /* Read a message from the infile */
        fscanf(inptr,"%s",message);

        /* Write the message to the out file */
        fprintf(outptr,
                "The message read from %s.in is %s\n",
                        host,message);

        return id;
}

/************************** RESULTS *************************/

All the  *.in files had a message :

 test

The worker got mapped to machine bighole and a new file
bighole.cs.umt.edu.out was created which had the following line :

The message read from bighole.cs.umt.edu.in is test

/**************************   END   ****************************/
```

```
/* Program 9 */
```

```
/* File : fork.cl
 * Description : This program shows UNIX fork and C-Linda
   eval interaction.
 */

real_main() {
        int worker(int);
        char who[40];
        int i, len;

        /* The main program evals a worker process */
        eval(worker(0));
        /* The main program reads messages sent by the worker
         * and the children of the worker processes created by
         * UNIX fork command */
        for ( i = 0; i < 30; i++){
            if (inp("message",?who:len) == 1)
                printf("Message from %s\n",who);
        }
}

/* The worker process is created by an eval call */
int worker(int id) {
        char message[40];

        /* The worker process forks once to create
                          a child process */
        if (fork() == 0) { /* This is the child process */
           strcpy(message,"child1");
  /* The child process sends a message to the master process */
           out("message",message:40);
           return id;
        } else if (fork() == 0){
          /* The worker process creates another child*/
           strcpy(message,"child2");
 /* The second child process sends a message to
                         the master process */
           out("message",message:40);
           return id;
        } else {
 /* The worker process also sends a message to the master */
           strcpy(message,"parent");
           out("message",message:40);
           return id;
        }
}

/************************ RESULTS ***************************/
        The program never executes properly. The master is
able to read the three messages sent by the worker and
its children, but the program crashes sending a message saying
that Linda is in panic mode. Obviously C-Linda process cannot
run normally when UNIX fork calls are inserted in C-Linda code.
/************************************************************/
```