Distributed Linda:
Design, Development, and Characterization
of the Data Subsystem

by
Patrick Glen Robinson

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Approved:

James D. Arthur, Chairman

Dennis G. Kafura
Calvin J. Ribbens

May 1994
Blacksburg, Virginia
Distributed Linda:
Design, Development, and Characterization
of the Data Subsystem

by
Patrick Glen Robinson
James D. Arthur, Chairman
Computer Science

(ABSTRACT)

The Linda model for concurrent processing employs a shared data space
approach for interprocess communication. The development of the distrib-
uted Linda system presented in this thesis implements distributed process
creation within the shared data space framework and provides for parallel
execution of Linda processes on a network of workstations. In addition, the
design of the system's shared data space allows it to be distributed over
multiple hosts, providing for parallel access to various regions of the shared
data space, thereby reducing contention among Linda processes for this
resource. A preliminary analysis of the system's execution profile has iden-
tified particular characteristics of the system which tend to limit computa-
tional performance under conditions of heavy I/O with the shared data
space. An investigation of the system's I/O behavior has led to the identifi-
cation of a technique which can improve the I/O performance of the system
by as much as an order of magnitude. However, this technique results in
inefficient use of network bandwidth by the Linda system. Consequently,
potential alternative techniques for improving the system's I/O perfor-
manace are presented.
Acknowledgments

I am deeply grateful to Dr. James D. Arthur for his enduring patience, insight, and guidance over the past four years. He and I have spent many hours discussing the issues and problems which are addressed in this thesis. Despite my tendency to probe every tangential issue, he has managed to keep my concentration and efforts focused on the goals and objectives of my research. Without his invaluable assistance, I would still be writing code.

I would also like to thank my employer, Tom McAnge, for his patience and understanding while I was working on this degree.

My deepest thanks I give to my wife Wendy. Her love, support, and encouragement have been a source of joy and strength over the years it has taken to complete this work.

Finally, I would like to thank God for enabling me to complete this degree. Throughout the course of this project, I have been reminded of Jesus’ words: “Apart from Me, you can do nothing.” (John 15:5, NASB).
# Table of Contents

1.0 Introduction .......................................................... 1

   1.1 Motivation for Distributed Computation .......................... 1

       Uniprocessor concurrency ....................................... 1

       Two types of parallelism ....................................... 2

       Shared memory vs. distributed memory MIMD architecture .... 2

       Parallelism in a distributed context ........................... 4

       Impetus for using Unix workstations to build a distributed Linda system ... 4

   1.2 Linda's Shared Data Space Communication Model ................. 5

   1.3 Problem Statement and Proposed Solution ........................ 7

   1.4 Blueprint of the Thesis ......................................... 8

2.0 Background ............................................................ 10

   2.1 The Linda Paradigm - Tuple Space and Coordination Language ... 10

       Tuple Space .................................................... 10

       Passive and Active Tuples ..................................... 10

       The Linda Coordination Language .............................. 11

       Linda Primitives ............................................... 12

       Templates and tuple matching .................................. 13

       Use of the EVAL operation ..................................... 14

   2.2 The Yale University Single-Machine Linda Implementation ........ 15

   2.3 Separation of Tuple Space from the Linda Program ................ 17

       The Remaining EVAL Issue ..................................... 18

3.0 Overview of Distributed Linda System ............................. 19

   3.1 Goals of System Design ......................................... 19

   3.2 Topology of the Distributed Linda System ........................ 20

       3.2.1 Two views of the Linda-LAN ............................... 20

   3.2.2 Components of the Control Subsystem ....................... 23

       L-Man - the Linda-LAN System Manager ......................... 23

       L-Com - the Workstation Communications Manager ............. 23

       3.2.3 Components of the Data Subsystem ........................ 24

           TS-Man - the Tuple Space Manager .......................... 24

           L-Kernel - the Workstation Tuple Data Manager .......... 24

   3.3 Focus on design of data components ................................ 25

4.0 Remote Process Instantiation and Execution ...................... 26

   4.1 Design Goals and Issues ....................................... 26

       4.1.1 Goals for the Design of the EVAL Mechanism ............. 26

       4.1.2 Issues to be Resolved by the Design ........................ 28
Remote process instantiation ........................................ 28
Identification of the proper code for execution by a remote Linda process .... 29
"Inheritance" of the proper execution environment by the new process ...... 30

4.2 Remote Linda Process Instantiation .................................. 30

4.3 Identification of the Proper Code for Execution by a Remote Linda Process ........ 33
  Partitioning of Preprocessed Source Code into Real_main() and Case_main() .... 33
  Obtaining an EVAL ID from the Linda System ................................ 35
  Shielding EVAL ID tuples from user operations .............................. 36

4.4 "Inheritance" of the Proper Execution Environment by the New Process .......... 36
  Function arguments to EVAL operations .................................. 39
  Shielding EVAL argument list tuples from user operations ................. 40

4.5 A More Complete Example ............................................ 41

5.0 Data Subsystem Components and their Interaction .......................... 46

5.1 TS-Man and Split Tuple Space ....................................... 46
  Functions of the TS-Man component ..................................... 46
  Overall Design of TS-Man ............................................. 47
  Tuple Space Partitions ............................................... 49
  Split Tuple Space ................................................... 49

5.2 L-Kernel .............................................................. 50
  Functions of the L-Kernel component ................................... 50
  Overall Design of L-Kernel .......................................... 52
  "Handle Active Tuple" command from L-Com ................................ 53
  Header and Data Packets ............................................ 53
  L-Kernel's Handling of Requests ...................................... 54

5.3 Interaction Between Data Subsystem Components ............................ 55
  Directing Responses from TS-Man Back to the Proper Linda Process ...... 55
  Adapting L-Kernel and TS-Man to the Split Tuple Space version of the system .... 56

6.0 Execution Dynamics: A Comprehensive Example .............................. 59

7.0 Investigation of Execution Profile ................................... 73
  Linda-LAN Configuration ........................................... 73
  Expectations for the Distributed System ................................ 74

7.1 Initial Execution Time Results ....................................... 76
  Results from the rayshade Application ................................ 76
  Results from the cmatrix Application .................................. 80

7.2 Enhancing the System's Performance ................................... 83
  TCP Buffering ......................................................... 86
  Results from rayshade on the Modified System .......................... 87
  Results from cmatrix on the Modified System ............................ 89
Conclusions ................................................................. 91
7.3 A Discussion of Buffering and Possible Design Alternatives .................. 91
  7.3.1 TCP’s Output Buffering Algorithm ................................ 91
      Impact of Buffering on the Linda System .......................... 92
  7.3.2 A Discussion of Three Design Alternatives ............................... 94
      Padding Small Packets to Maximum Size ............................. 94
      Sending Header and Data as a Single Packet ......................... 96
      Handling Data Packets Independently .................................. 98
      Final Consideration ..................................................... 99
8.0 Summary and Future Work .................................................. 100
  8.1 Summary of Contributions .................................................. 100
  8.1.1 The Distributed EVAL Mechanism .................................... 101
  8.1.2 Split Tuple Space .................................................... 102
  8.1.3 Investigation and Analysis of Results ................................ 102
  8.2 Future Work ............................................................. 103
      Future work on the distributed system’s design ..................... 103
      Future Investigation of the Distributed System’s Behavior .......... 104

References and Bibliography .................................................. 106

Appendix A. Start-up and Shutdown of the Linda-LAN Environment .......... 114
  A.1 System Start-up ....................................................... 114
      Control subsystem start-up ............................................. 114
      Distributing the user’s executable Linda program to the Linda-LAN  114
      Data subsystem start-up ................................................. 117
  A.2 Shutdown of the Linda-LAN environment ................................ 120
      Data subsystem shutdown .............................................. 120
      Control subsystem shutdown .......................................... 121

Vita ................................................................. 122
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Shared Memory Architecture</td>
<td>3</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Distributed Memory Architecture</td>
<td>3</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Distributed Processing System</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>The Yale Linda System</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>EVAL causes fork() call, resulting in new process</td>
<td>16</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Separate Tuple Space Manager</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Conceptual topology of the Linda-LAN</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>One possible organization of the distributed Linda-LAN</td>
<td>22</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Instantiation of a New Linda Process by the Distributed System</td>
<td>32</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Conceptual topology of Split Tuple Space Linda-LAN</td>
<td>51</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Function call hierarchy in preprocessed Linda code</td>
<td>64</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Eval process: Steps 1 through 15, and 36 through 37</td>
<td>65</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Eval process: Steps 16 through 35</td>
<td>66</td>
</tr>
<tr>
<td>Figure 7.1</td>
<td>Linda-LAN configuration used in the investigation</td>
<td>75</td>
</tr>
<tr>
<td>Figure 7.2</td>
<td>Execution times from rayshade application</td>
<td>78</td>
</tr>
<tr>
<td>Figure 7.3</td>
<td>Another view of the times from rayshade</td>
<td>79</td>
</tr>
<tr>
<td>Figure 7.4</td>
<td>Execution times from cmatrix application</td>
<td>82</td>
</tr>
<tr>
<td>Figure 7.5</td>
<td>Another view of the times from cmatrix</td>
<td>84</td>
</tr>
<tr>
<td>Figure 7.6</td>
<td>Round-trip path taken by requests for tuples</td>
<td>85</td>
</tr>
<tr>
<td>Figure 7.7</td>
<td>Execution times of rayshade using TCP_NODELAY</td>
<td>88</td>
</tr>
<tr>
<td>Figure 7.8</td>
<td>Execution times of cmatrix using TCP_NODELAY</td>
<td>90</td>
</tr>
<tr>
<td>Figure A.1</td>
<td>Startup of the Control Subsystem</td>
<td>115</td>
</tr>
<tr>
<td>Figure A.2</td>
<td>Distribution of the Linda program to the Linda-LAN</td>
<td>116</td>
</tr>
<tr>
<td>Figure A.3</td>
<td>Startup of the Data Subsystem</td>
<td>118</td>
</tr>
</tbody>
</table>
1.0 Introduction

This thesis presents the design and preliminary investigation of a system for distributed computation implementing the Linda model of concurrent processing. This chapter describes the motivation for studying and building environments for distributed computing, and for adopting Linda’s shared data space model as the interprocess communication mechanism for our distributed environment.

1.1 Motivation for Distributed Computation

One of the goals of parallel computing systems is to exploit concurrency by providing the facilities to perform multiple computational tasks simultaneously. Such systems have the potential for increased throughput and hence, better computational performance, than their single processor counterparts.

Uniprocessor concurrency

Concurrency in single processor systems is achieved through multiprogramming, a technique giving the illusion of parallelism by providing the means for multiple processes to compete for and share the computational power and time of a single CPU. Processes share the cycles of the CPU by executing in an interleaved fashion. One process runs for a limited period of time, and then the CPU is turned over to another process. Since a uniprocessor system can be executing only a single process at any given point in time, other concurrent processes on the system (i.e. those existing at the same time) are inactive, waiting either for completion of an I/O request, or to be rescheduled to run on the CPU.
Two types of parallelism

Parallel processing systems have multiple CPUs, all of which can execute instructions in parallel. Depending on the system's architecture, all processors may execute the same instruction in parallel, each operating on a different data element, or each may execute different sets of instructions, independently. Flynn termed these types of parallelism SIMD (single-instruction stream, multiple-data streams) and MIMD (multiple-instruction streams, multiple-data streams), respectively [FLYM66].

SIMD machines are most useful for the narrow problem set domain in which algorithms can be written to perform operations on all elements of arrays or vectors simultaneously. MIMD machines, however, are useful for a much broader and more extensive domain of problems, since the processors of such systems are not constrained to execute the same instruction at the same time.

Shared memory vs. distributed memory MIMD architecture

MIMD machines may be designed so that memory and I/O subsystems are accessible to all processors. This organization is called a shared memory architecture (see Figure 1.1). Alternatively, these subsystems may be local and private to each processor, in which case the machine is said to have a distributed memory architecture (see Figure 1.2) [STOH87]. Additionally, there must exist some mechanism for independent processes executing on different processors of a MIMD system to communicate with each other. Such interprocess communication (IPC) provides processes with the means to share data, and to synchronize their activities.
Figure 1.1. Shared Memory Architecture

Figure 1.2. Distributed Memory Architecture
Parallelism in a distributed context

A distributed processing system provides for parallel computation by combining the power of a network of machines into a unified system. Each computer on the network has its own local memory, and communication between processes executing on different nodes of the system occurs over the network. This system of networked machines can be viewed as a "distributed memory architecture" parallel computation system (see Figure 1.3).

![Diagram of a distributed processing system with four machines connected through a communications network.]

Figure 1.3. Distributed Processing System

Impetus for using Unix workstations to build a distributed Linda system

The distributed system presented in this thesis is implemented on a group of Unix\(^1\) workstations connected to each other through an Ethernet\(^2\) network. Our decision to build a distributed processing system using workstation class machines running Unix is motivated by several factors:

---

1. Unix is a trademark of Novell Corporation.
2. Ethernet is a trademark of Xerox Corporation.
• Workstations are relatively inexpensive, commonplace, and accessible, in comparison with more expensive parallel computers.

• Workstations are underutilized, and are often completely idle. Theimer and Lantz in [THEM89] report that of the approximately 70 workstations studied, as many as one third remained completely idle, even during the busiest times of the day. Harnessing the unused CPU cycles of such a network of machines can provide a cost effective source of computational power.

• The Unix operating system [BACM86, LEFS89, BERK86] is the traditional system of choice for workstations, and provides a wealth of convenient services to the system designer (e.g. multitasking, process creation, and IPC).

• The original Linda system, which the VPI Linda Research Group has used as a foundation for various Linda research efforts, was implemented on and for Unix, and is dependent on several system facilities specific to Unix.

1.2 Linda’s Shared Data Space Communication Model

Computing systems allowing programs to have multiple threads of control must provide interprocess communication services, enabling process creation, process synchronization, and data sharing. These services are especially vital for parallel and distributed systems designed to exploit concurrency by simultaneously executing related processes.

While several models for such communication have been proposed and studied (e.g. message passing, as in CSP [HOAC78, MAYM83], monitors [HOAC74, BLAA86, JULE88, BRIP75, WIRN77, LAMB80, HEWC73, AGHG86], concurrent logic programming [RING88, SHAE87], and functional programming [CHEM86, CHEM88, TURD82]),
Gelertner's Linda [GELD85a, CARN86b] employs a unique shared data space approach. That is, it provides a repository for information storage and retrieval that is globally accessible to all processes participating in the parallel execution environment. Gelertner terms his model “generative communication” [GELD85a] because one process communicates with another process by “generating” a persistent data element in the globally shared data space. The other process receives the communication by accessing the data element asynchronously.

The fact that shared data space is global to all processes in a parallel or distributed system greatly simplifies some of the potential complexities of developing parallel programs. For example, a process can synchronize its activities with those of another process by reading a message placed in the shared data space by that other process. A process shares data with one or more other processes by placing that data in the shared data space. Data producers require no information concerning the existence or identity of data consumers, since they simply place data in the shared data space. Likewise, consumers obtain data from the shared data space, rather than directly from producers. As a result, complex process addressing and message routing schemes are not needed because all messages between processes are data elements written to and read from the shared data repository.

It is important to understand that the term “shared data space” refers to a logical, rather than a physical construct, although it can be implemented using physically shared memory on a uniprocessor or multiprocessor machine. A distributed system, however, would require an alternative implementation, as we discuss in Chapter 2.
1.3 Problem Statement and Proposed Solution

This thesis discusses the development of a distributed system for concurrent process execution that reflects the shared data space model of communication found in Gelertner’s Linda computational system. The Linda paradigm, as discussed above, features globally shared data space and provides an effective, yet conceptually simple framework for parallel programming.

The VPI Linda Research Group obtained source code for the Linda compiler, run-time library, and related utilities from Yale University. This Linda implementation was designed to exploit parallelism on shared memory architecture multiprocessor computers. Initial research efforts by Schumann focused on the design of a distributed version of the shared data space, and requisite support systems [SCHC93].

The objective of the effort presented in this thesis is the evolution of Schumann’s modified Linda system into a distributed system which provides for simultaneous execution of Linda processes on a network of Unix uniprocessor workstations. Toward that end, we have identified and carried out the following steps:

- Modification of the Linda compiler to generate code which enables a Linda program to create new Linda processes on remote machines. We placed considerable emphasis on designing this mechanism to be consistent with the Linda paradigm. More specifically, we wanted this mechanism to use shared data space for interprocess communication.

- Creation of a “Linda-LAN” run time environment to facilitate the instantiation of remote Linda processes, and to provide for Linda network and process management. This has been a cooperative effort, and only those aspects and functions of the Linda-LAN which are pertinent to the distribution of Linda processes are discussed in detail.
in this thesis.

- Distribution of the globally shared data space over multiple hosts by implementing separate data space managers for the various logical data partitions of the shared data space. This is done to reduce potential contention among Linda processes for access to the shared data space, and to reduce the performance bottleneck that such contention would introduce.

- Preliminary analysis of the system's performance. The intent of the investigation is to identify characteristics of the distributed system that tend to enhance or limit computational performance. In particular we investigate the impact on program execution time of varying the number of processors and processes in the system.

1.4 Blueprint of the Thesis

In preparation for discussing the design of the distributed Linda system, Chapter 2 presents a brief introduction to Linda concepts and the Linda coordination language. We then describe the original single-machine version of Linda, and follow that with a discussion of Schumann's "separate shared data space" version on which our distributed system is based. We focus attention on issues, limitations, and deficiencies of these systems pertinent to the transformation of the current Linda system into one supporting truly parallel process execution in a distributed context.

Chapter 3 presents the goals for the overall design of the Linda-LAN's topology, followed by an overview of that topology, and a brief description of the role of each component in the system. This overview will serve as a framework for a detailed discussion of the distributed system's design in Chapters 4 and 5.
Chapter 4 presents the issues which had to be addressed in designing the distributed system's mechanism for remote Linda process instantiation, and the specific design goals reflected by their resolution. We then present and discuss the design and pertinent implementation details, focusing on how design goals are achieved.

Chapter 5 presents particular components of the Linda-LAN’s data subsystem, and describes a feature of the distributed system which enables concurrent access to various regions of the shared data space.

Chapter 6 presents a section on execution dynamics, which serves to complete the overall picture of the system’s provision for distributed computation.

Chapter 7 presents the results, discussion, and conclusions of the preliminary investigation of the distributed system’s execution profile. In addition, we describe several potential design alternatives which could enhance overall performance of the system.

Chapter 8 summarizes the design and analytical work presented in this thesis. We reiterate important design considerations, and discuss how the design goals were achieved. Following that, we present insights gained and conclusions reached from the investigation of the distributed system’s execution profile. Finally, we suggest several topics for future investigation in the context of our distributed Linda system.
2.0 Background

This chapter presents background material on basic Linda concepts. It then provides a discussion of the two successive implementations of Linda on which we have based our development of the distributed Linda system. This material serves as a basis for the design issues to be discussed and resolved in Chapters 3 and 4.

2.1 The Linda Paradigm - Tuple Space and Coordination Language

Tuple Space

We have characterized Linda as a paradigm that provides a framework which is suitable for writing and executing parallel programs, and one that employs the shared data space model of interprocess communication. In Linda parlance, this shared data space is called Tuple Space (TS) [GELD85a, CARN87]. As might be expected, its member objects are called tuples. A tuple is defined as an ordered set of data elements. Tuples are not ordered per se in Tuple Space, but are said to be “adrift” in Tuple Space [CARN89a]. An individual tuple’s identity is composed of the number, order, data types, and values of its fields. Tuple Space can be viewed as a globally accessible associative memory [GELD85a]. That is, member tuples are accessed by content, and by description of content, rather than by address.

Passive and Active Tuples

A given tuple in Tuple Space can be categorized as either a passive data tuple or an active process tuple. Passive tuples can be viewed as conventional static data elements. They are placed directly into Tuple Space, and can be read and removed from Tuple Space by Linda
processes. An active tuple, however, contains specifications for the execution of a Linda process which is eventually instantiated by the Linda system. This new Linda process removes the active tuple from Tuple Space, transforms it into a passive data tuple, and then places this new passive tuple in Tuple Space. An active tuple, or more precisely, the Linda process it specifies can also produce and consume passive data tuples, and generate more active process tuples.

Examples of passive tuples:

- `<“arrayA”, 3, 5>` - the 3rd element of the integer array called “arrayA”, the value of which is 5.
- `<“plane”, 5.6, 0.3>` - a point on a plane.

Examples of active tuples:

- `<“Fibonacci”, 10, fib_compute(10)>` - specifications for a Linda process which will evaluate the tuple and, assuming the function “fib_compute()” does what we expect, place the passive tuple `<“Fibonacci”, 10, 55>` in Tuple Space.
- `<worker()>` - specifications for a Linda process which will call the function worker(), presumably to perform some computation in parallel with the calling process. The passive tuple formed by evaluating this active tuple may not be of interest; nevertheless, it is placed in TS.

The Linda Coordination Language

The Linda “language” is actually a coordination language [GEL92], rather than a complete parallel programming language. A coordination language focuses on providing the primitives to create processes as well as to coordinate communication among those processes. These primitives are embedded by the programmer in a base language. Although several languages have been integrated with Linda [CARN89a, GELD90, JELR90], the
base language used by the Linda implementations discussed in this thesis is C, and the resulting hybrid is called C-Linda. Carriero and Gelbert have stated that the Linda coordination language is “independent of and orthogonal to” the base language [GELD92]. That is, the two languages are conceptually unrelated. The base language is concerned with computation rather than interprocess communication. Linda, on the other hand, is concerned with process creation and coordination rather than computation. Process creation, coordination, and communication are achieved by individual Linda processes manipulating the contents of Tuple Space by means of the Linda language primitives.

**Linda Primitives**

Linda primitives (also called Linda operations) fall into two basic categories. *Generative* operations place tuples into Tuple Space, and *retrieval* operations obtain tuples from Tuple Space.

**Generative operations:**

- **OUT** - places a new passive tuple into Tuple Space.
- **EVAL** - places a new active tuple into Tuple Space.

**Retrieval operations:**

- **IN** - requests a passive tuple from Tuple Space. If a suitable tuple is found in TS, it is returned to the calling Linda process. Otherwise, IN blocks until a suitable tuple arrives in TS.
- **INP** - similar to IN, except if no suitable tuple is found in TS, INP does not block, but rather returns an indication of failure.
- **RD** - requests a copy of a passive tuple in TS. If a suitable tuple is found in TS, a copy of it is returned to the calling process, and the original tuple is not removed from TS. Otherwise, RD blocks until a suitable tuple arrives in TS.
RDP - similar to RD, except if no suitable tuple is found in TS, RDP does not block, but rather returns an indication of failure.

Again, active tuples are not removed from Tuple Space by Linda operations. Linda processes transform them into passive tuples, and then place them back into Tuple Space. Therefore, none of Linda’s retrieval operations deal with active tuples.

*Templates and tuple matching*

A retrieval operation specifies a template that a tuple in Tuple Space must match, rather than specifying an actual tuple. A template, like a tuple, is an ordered set of fields, but a template specifies the structure and content of tuple(s) which may be returned to the caller of the retrieval operation. Each field of a template is either an actual field or a formal field. An actual field specifies the value (and implicitly, the data type) of a matching tuple’s corresponding field. A formal field specifies only the data type of a matching tuple’s corresponding field, and is assigned the tuple field’s value on successful return from the operation. In general, a template matches a tuple in TS when corresponding tuple and template fields match in number and in type. In addition, any actual fields in the template must equal corresponding actual fields of the matching tuple.

As stated, a retrieval operation returns a matching tuple to the calling Linda process. This occurs by template formal fields being “filled in” with the values of corresponding fields from a matching tuple in Tuple Space. For example, a Linda process can retrieve the tuple

<"arrayA", 3, 5>

by calling the IN operation as follows:

\[ \text{in("arrayA", 3, ?i).} \]
The first and second fields of the tuple in this example match the corresponding actual fields of the template. The \( ?i \) notation in the template’s third field specifies a formal field, and we assume that the data type of \( i \) is integer. The third field of the tuple then matches this formal field, since it is an integer field. When the IN operation returns, the value of \( i \) will be 5.

Use of the EVAL operation

Typically, at least one of the fields of an active tuple (that is, one of the arguments to an EVAL operation) is a call to a function. That function will not be called until the active tuple is transformed by a newly instantiated Linda process. Hence, a Linda program uses the EVAL operation to create an independent Linda process to perform some computation, conceptually in parallel with its calling process. We discuss the distributed system’s handling of “EVAL-ed functions” in Chapter 4, Section 4.2. In the Linda implementations discussed in this thesis, simple computations appearing in arguments to EVAL operations are evaluated prior to the active tuple’s being placed in Tuple Space. For example, consider the operation

\[
\text{eval("element", i+6, compute(j*3))}.
\]

If \( i \) has the value 2, and \( j \) has the value 3, then the active tuple placed in TS is

\[
<"element", 8, compute(9)>. 
\]

The next two sections present brief overviews of two successive implementations of Linda on which the distributed Linda system is built. Discussion is focused on the details which are germane to the design problems of the distributed Linda system.
2.2 The Yale University Single-Machine Linda Implementation

The Yale University implementation of Linda is designed to run on a single uniprocessor or multiprocessor computer under the Unix operating system. The Linda compiler generates a single executable program, using a preprocessor approach to transform calls to Linda operations into C source code. That is, each call to a Linda operation in a C-Linda program is analyzed in the context of the other Linda operations, and is transformed into a call to a new C function. The code for these new functions is also generated by the Linda compiler.

The Tuple Space data structures reside in memory physically shared between the main Linda program and Linda processes created by calls to the EVAL operation (see Figure 2.1). Access to the shared memory comprising Tuple Space is synchronized and serialized by the use of semaphores. Tuple Space access functions (also known as "kernel access functions") reside in a C run time library and are part of the executable program.

![Diagram of the Yale Linda System](image)

**Figure 2.1.** The Yale Linda System
The Linda EVAL operation, used to create new Linda processes to handle active tuples, is implemented using the Unix fork() system call. This approach enables a Linda process to spawn an exact copy of itself to execute the code generated by the Linda compiler implementing the transformation of an active tuple into a passive tuple in Tuple Space. The first action taken on entering the C function implementing a particular EVAL operation is a call to the fork() system call. The parent process immediately returns from the function implementing the EVAL, whereas the child process executes the function, and then exits. The function executed by the child process transforms the active tuple into a passive tuple (by evaluating any function call fields of the active tuple), and then deposits the passive tuple into Tuple Space.

![Diagram](image)

**Figure 2.2.** EVAL causes fork() call, resulting in new process
The degree of parallelism potentially achievable under this system would, of course, depend on the number of processors available to Linda processes on the host computer. As described in Chapter 1, if this Linda system were run on a common uniprocessor system, then the concurrent Linda processes would execute in an interleaved fashion, and share the time of the single CPU.

2.3 Separation of Tuple Space from the Linda Program

The Yale Tuple Space implementation is feasible on a single uniprocessor or multiprocessor machine running Unix because shared memory and semaphores are kernel services of the operating system. However, since there can be no physically shared memory between processes in a distributed system, the first step taken by the VPI Linda Group toward implementing a distributed Linda system addressed the use of shared memory between Linda processes in the implementation of Tuple Space, and the use of semaphores to control access to the shared memory. In particular, Tuple Space and its associated kernel access code were separated from the actual Linda program, and a separate Tuple Space Manager program was developed to handle all requests for access to Tuple Space [SCHC93].

Tuple Space itself is implemented through data structures local to the Tuple Space Manager. Communication between Linda processes and the Tuple Space Manager occurs through the use of stream sockets. As a result, Tuple Space can reside on a machine separate from the one running Linda processes (see Figure 2.3). Since the Tuple Space kernel access code was moved to the new Tuple Space Manager, the functions in the Linda runtime library were replaced with code to access the stream socket communication interface between the Linda process and the Tuple Space Manager process.
The Remaining EVAL Issue

With Tuple Space cleanly separated from the Linda computational process, the primary limitation remaining to be resolved in the design of the distributed Linda system is the dependence of the Linda EVAL operation on the Unix fork() system call as the mechanism for creating new Linda processes. Use of the fork() call implies that the Linda system can only create new processes on the same machine on which the main Linda program is currently running. However, the distributed Linda system must accommodate the creation of Linda processes on remote machines. Because no semantically equivalent remote fork() call is available, a new approach must be devised to instantiate a new Linda process on a given machine. The design of a new EVAL mechanism which will provide for truly distributed process execution is the topic of Chapter 4.
3.0 Overview of Distributed Linda System

Before presenting the details of the design of the distributed system, and its mechanism for handling distributed EVAL operations, we discuss some considerations which motivate our design of the overall topology of the distributed Linda environment (or "Linda-LAN"). We then present that topology, briefly describing the environment's subsystems and the role of each component. This overview of the topology will serve as a framework for the design discussion in Chapter 4.

3.1 Goals of System Design

Several factors motivate our design for the distributed Linda environment:

- First and foremost, we need a mechanism for implementing the EVAL primitive in a distributed context. As discussed in Chapter 2, the Linda system on which this work is based depends on the Unix fork() system call to create new Linda processes. We need a mechanism to simulate a "remote fork()" for the distributed system.

- For simplicity and convenience, we want control of the distributed system to be centralized. That is, we want a single process to be in charge of managing and monitoring the Linda-LAN. Centralized control simplifies collection of system status data, and seems to be the most straightforward design from an implementation point of view.

- Implementation of the Linda-LAN has been a cooperative effort of the VPI Linda Research Group. Early in our design planning discussions, we recognized that the Linda-LAN would be comprised of both LAN control functions and tuple data flow functions. We decided to organize these functions into two distinct subsystems, control and data. Once the overall functions of the Linda-LAN were established, members of
the VPI Linda group would be able to design and implement these subsystems in parallel. Ideally, the only “overlap” would concern the various communications interfaces between the subsystems.

- Again for simplicity and ease of implementation, we want to use stream sockets for all communication between components of the distributed Linda system.

3.2 Topology of the Distributed Linda System

The Linda-LAN distributed processing environment is comprised of various component processes, each of which is a member of either the control or data subsystem. Control components are responsible for overall system management and decision making, whereas data components primarily handle the flow of tuples between Linda processes and Tuple Space.

3.2.1 Two views of the Linda-LAN

Figure 3.1 portrays a conceptual view of the distributed Linda system’s topology, showing the two subsystems, their components, and the relationships between the various components of both subsystems, in terms of their communications interfaces. The function of each component is described below.

Figure 3.2 shows one likely organization of the distributed Linda system, operating on a network of five computers. The independent control and data information paths are depicted as two distinct “buses” connecting various components of the Linda-LAN. This illustration also indicates that any Linda-LAN component can co-reside on a machine with any other component. This characteristic of the Linda-LAN provides flexibility for use on both large and small networks.
Figure 3.1. Conceptual topology of the Linda-LAN
Figure 3.2. One possible organization of the distributed Linda-LAN
3.2.2 Components of the Control Subsystem

The control subsystem is made up of the Linda-LAN System Manager (L-Man) and the per-workstation Communications Manager (L-Com).

**L-Man - the Linda-LAN System Manager**

A single L-Man System Manager process maintains control over the entire distributed environment and has knowledge of the state of the distributed system and the status of all machines participating in the system. L-Man functions as a command dispatcher. Commands needing to be sent to and processed by all nodes on the Linda-LAN (such as distribute executable program, prepare for program start-up, and program shutdown) are sent by L-Man to the L-Coms on each node. L-Man is also responsible for selecting machines on which to instantiate new Linda processes. It makes these decisions based on load average information obtained from the L-Com processes on each machine in the environment.

**L-Com - the Workstation Communications Manager**

L-Man spawns an L-Com process on each machine participating in the distributed environment. L-Com serves as the point of communication for L-Man on each workstation. That is, any messages (from Linda processes or from users) which need to be processed by L-Man are transmitted to L-Com, and L-Com forwards them to L-Man. Similarly, L-Man distributes commands to the system by sending them to the L-Coms. When L-Man selects a machine on which to instantiate a new Linda process, it initiates the action by sending an appropriate message to that machine’s L-Com. Each L-Com also periodically (and asynchronously) reports its workstation’s load average to L-Man, so that L-Man has up-to-date information for deciding where to instantiate new Linda processes.
3.2.3 Components of the Data Subsystem

The data subsystem is made up of the Tuple Space Manager (TS-Man) and the per-workstation Tuple Data Manager (L-Kernel), which is the Linda process's link to TS-Man and Tuple Space.

TS-Man - the Tuple Space Manager

TS-Man handles all access to Tuple Space, including incoming tuples, incoming requests for tuples, and outgoing tuples (i.e. responses to requests). TS-Man has a data stream socket link to each workstation's L-Kernel. All tuple and template communication with Linda processes on a workstation occur over this link. TS-Man also maintains a control stream socket link to L-Man. This link is used to notify L-Man whenever an active tuple arrives in Tuple Space (this mechanism is discussed further in Chapter 4). As in Schumann's Separate Tuple Space Manager version of Linda described in Section 2.3, TS-Man processes each request completely before receiving the next request, thereby serializing and synchronizing access to Tuple Space.

L-Kernel - the Workstation Tuple Data Manager

L-Kernel (whose name is historical, rather than derived from its function) is responsible for handling the flow of all tuple information between Linda processes on a workstation and the Linda-LAN's TS-Man. L-Kernel's primary purpose is to reduce the number of stream socket connections required between TS-Man and Linda processes by multiplexing tuple and template data from all the Linda processes on the workstation into a single socket connection to TS-Man. L-Kernel also must demultiplex all tuple data received from TS-Man, and forward this data to the proper Linda processes.
Another advantage of our design is that Linda processes establish connections to L-Kernel (rather than to TS-Man, as in Schumann's Linda system), removing the burden of connection management tasks from the Tuple Space Manager. TS-Man's purpose is to service requests for access to Tuple Space. It's performance would suffer unnecessarily from having to manage this additional activity. Our design distributes this connection management function over all the machines in the environment. All socket connections between TS-Man and L-Kernels are established before the Linda program runs.

3.3 Focus on design of data components

As stated at the beginning of this chapter, this overview of the distributed Linda environment's topology serves as a framework for the design and implementation discussion in Chapter 4. The primary design work presented in this thesis is the mechanism for distributing EVAL operations. This mechanism enables a Linda process to instantiate a new Linda process on a remote machine. We focus entirely on the design of the data subsystem of the Linda-LAN, and refer to the control subsystem only to describe how its functionality is used by components of the data subsystem in handling EVAL operations.
4.0 Remote Process Instantiation and Execution

Chapter 3 presents the overall topology of the Linda-LAN environment, and discusses factors which serve to motivate our design of that topology. This chapter presents the design of the distributed Linda system’s mechanism for concurrent transformation of active tuples into passive tuples. This mechanism is invoked by calling Linda’s EVAL operation.

4.1 Design Goals and Issues

Before discussing the details of the EVAL handling mechanism, we list specific goals for the design, and then examine various issues which have been addressed and resolved by the design.

4.1.1 Goals for the Design of the EVAL Mechanism

One of our major goals in designing the distributed Linda system was to reuse the Separate Tuple Space Manager version of Linda, on which we have based the development of the distributed system. For convenience, we refer to the Separate Tuple Space Manager version as the “D-Kernel” implementation, since it provides a Distributable Kernel. Other goals we wanted to realize in our design include the following:

• *Leave D-Kernel’s Tuple Space access routines intact.* In fact, our goal has been to use D-Kernel’s Tuple Space Manager program as an almost-complete model for the TS-Man component of the Linda-LAN run time system. Only minor modifications need to be made to utilize the Tuple Space Manager in a distributed context, and to enable TS-Man to accept and store active tuples.
• **Reuse D-Kernel's stream socket communication interface between Linda processes and the Tuple Space Manager.** This interface provides the mechanism by which Linda processes and the Tuple Space Manager are able to send and receive tuples over a serial stream socket.

• **Generate a single executable file.** Both the Yale and D-Kernel implementations of the Linda compiler generate a single executable file from the user's Linda source code. Because of the conceptual simplicity of this model, we want the distributed system to operate in a similar fashion. That is, the Linda compiler is to generate a single executable file which is distributed to all workstations participating in the Linda-LAN. (Distribution of executable files is described in Appendix A.)

• **Design an EVAL mechanism which is consistent with the Linda paradigm.** The D-Kernel and Yale systems simply invoke the `fork()` system call to spawn new processes to execute active tuple transformation code, but we want Linda processes in the distributed system to place actual active tuples in Tuple Space. As stated in Section 2.1, active tuples in Tuple Space are transformed into passive tuples "by the Linda system," rather than by the Linda processes which create them.

• **Exploit Tuple Space and Linda's IN and OUT operations in the design of the distributed EVAL operation.** Since the D-Kernel implementation already provides for the transfer of tuple data between Linda processes on one machine and Tuple Space on another, the design of the distributed system's EVAL mechanism should exploit this capability. In particular, Tuple Space and Linda's IN and OUT primitives should be utilized in the system's implementation of EVAL. That is, we want *use Linda* in the implementation of the distributed version of Linda.
4.1.2 Issues to be Resolved by the Design

The remaining obstacle of the D-Kernel system to be overcome is its dependence on the semantics of the Unix `fork()` system call in implementing the Linda EVAL operation. As outlined in Chapter 2, `fork()` allows a Linda process to spawn an exact copy of itself to execute the code implementing a given EVAL call (that is, to transform an active tuple into a passive tuple). However, `fork()` can create a new process only on the same machine as the process that called `fork()`. But, the distributed system requires a Linda process to effect the creation of new Linda processes on remote machines. Since no "remote fork()" is available, a new mechanism for handling the EVAL operation must be constructed. As discussed in the previous section, we want the distributed system to handle EVALs in a manner which is consistent with the Linda paradigm.

Several issues must be addressed in the design of such an EVAL mechanism:

1. Remote process instantiation,
2. Identification of the proper code for execution by a new Linda process, and
3. "Inheritance" of the proper execution environment by the new Linda process

Again, our approach in resolving these issues is guided by the desire to minimize the extent of modifications to the D-Kernel implementation on which the distributed system is based.

Remote process instantiation

The first step in creating a new Linda process is the actual instantiation of a new process on a remote machine. That is, once a Linda process has placed an active tuple in Tuple Space, there must be a way for the Linda system to (1) recognize that an active tuple
awaits transformation in Tuple Space, and (2) instantiate a new process on a participating Linda-LAN machine to transform this active tuple into a passive tuple. Section 4.2 describes how these steps are carried out by components within the Linda-LAN environment.

**Identification of the proper code for execution by a remote Linda process**

Once the Linda system has instantiated a new process to handle an active tuple, that process must determine which active tuple it is to evaluate, or more specifically, what code it must execute. Section 2.2 describes the process creation mechanism in the Yale and D-Kernel systems. When a process in one of these systems encounters a call to EVAL, it invokes `fork()` to create a new child process. This child process executes the code implementing the EVAL call. The parent process continues executing its own code without waiting for the child to complete. In these implementations, the child process commences execution at the point in the code where the `fork()` call is initiated. This feature is provided by `fork()`. The new process it creates is an almost exact duplicate of the calling process. In particular, values of all the parent’s system and user data are preserved in the child, including the current instruction pointer.

The distributed Linda system cannot enjoy this benefit of the `fork()` call since it is not the Linda program which creates new processes with `fork()`. In the distributed version, it is the responsibility of the Linda system, rather than the user’s Linda program, to create a new Linda process. Since the distributed system’s compiler generates only a single executable program file from the user’s Linda source code, an instantiation of that program “inherits” the code of the process which called the EVAL operation. However, this instantiation must be informed of the point in the code at which it is to begin execution. That is,
the new process must be told the address of the block of code which implements an evaluation of the specific active tuple.

"Inheritance" of the proper execution environment by the new process

Within the Linda paradigm, when an active tuple is placed in Tuple Space, it contains all pertinent specifications to ensure a correct transformation to a passive tuple. In the Yale and D-Kernel versions of Linda, physically acquiring those pertinent specifications relies again on the semantics of the fork() system call. In particular, when a new process is forked to evaluate an active tuple, it inherits a copy of the execution environment (or address space) of the forking process. As discussed above, this means that the child process will have its own copy of all the parent's system and user data. The implication is that the new Linda process automatically receives the necessary transformation specifications. No further interprocess communication is required to transform the associated active tuple into a passive one.

Similarly, when a process is instantiated by the distributed Linda system on a remote machine, it too needs all pertinent specifications to evaluate an active tuple. Because the direct sharing of address space is not possible in a purely distributed environment, "inheritance" of such specifications must be achieved in an indirect manner. Essentially, a list of arguments that conveys the execution parameters associated with the active tuple must be passed to the new remote process.

4.2 Remote Linda Process Instantiation

One of the major functions of the Linda-LAN environment is instantiating new Linda processes to (1) transform active tuples into passive tuples, and (2) place those passive tuples
back in Tuple Space. This instantiation process is accomplished by the following sequence of events (see Figure 4.1):

1. A Linda process uses the EVAL operation to send an active tuple to Tuple Space.

2. The Tuple Space Manager (TS-Man) receives the active tuple, places it into Tuple Space, and notifies the Linda-LAN System Manager (L-Man) that an active tuple awaits evaluation.

3. L-Man selects a Linda-LAN node to evaluate the active tuple (based on its knowledge of the current load average of each machine participating in the Linda-LAN), and sends a *handle active tuple* command to that node’s L-Com.

4. L-Com forwards this command to its L-Kernel.

5. L-Kernel receives the command, and calls `fork()` to create a new process.

6. The parent L-Kernel process continues its normal processing, but the new child process immediately calls the Unix `exec1()` system call to transform itself into a new process, running the executable Linda program generated by the Linda compiler.

Whereas the instantiation of a new Linda process is *direct* in the Yale and D-Kernel implementations (where a Linda process’s calling EVAL results in its direct invocation of the `fork()` system call), it is *indirect* in the distributed Linda system, as implied by Figure 4.1. The `fork()` call is used by the distributed system to create new processes, but this system does not depend on any other semantics of `fork()`. In particular, the new Linda process employs the `exec1()` system call to enable execution of the compiler-generated program from its beginning. The following section describes the distributed system’s
Figure 4.1. Instantiation of a New Linda Process by the Distributed System
mechanism for providing a new Linda process with the information it needs to address the proper code in the executable program.

4.3 Identification of the Proper Code for Execution by a Remote Linda Process

As stated, the Linda compiler generates a single executable program file from the user’s source code. The implication is that the Linda program run by the user is the same program which is run by the Linda system when new Linda processes are instantiated (either locally or remotely). This section describes how a new Linda process identifies and executes the block of code which implements the evaluation of a specific active tuple.

Partitioning of Preprocessed Source Code into Real_main() and Case_main()

When a programmer writes a Linda program, he names his “main()” function real_main(). The actual main() function (which is where a C program’s execution begins) resides in the Linda run time library. The D-Kernel system’s main() function connects the Linda process to the Tuple Space Manager and then calls the user’s real_main() function. Since the main Linda program is invoked only once in this system, this arrangement functions properly.

However, the distributed system’s main Linda program is run by each new Linda process. Since there is only a single executable Linda program, each remote instantiation must recognize that it is to evaluate an active tuple rather than execute the user’s real_main() code. In other words, remote Linda processes should not commence execution of the Linda program from the beginning, but should jump directly to the code which implements the evaluation of the appropriate active tuple.
To implement this requirement, the Linda compiler (that is, the preprocessor which transforms embedded Linda operations into C code) has been modified to generate a special function that is to be executed by remote Linda processes instead of the user’s real_main() function. This function, which we refer to as “case_main()”, determines (through a mechanism discussed in the next section) which EVAL operation is to be “called”, and then calls the appropriate functions implementing that particular EVAL operation.

A Linda process decides whether to execute real_main() or case_main() by determining whether it was started by the user, or by the Linda system. It makes this determination by examining the process id of its parent process. A Linda process started by the Linda system will have L-Kernel as its parent process, as discussed in Section 4.2.

case_main() is a special Linda process start-up function, so named because the calls to the functions implementing the various EVALs are grouped together inside a C language “switch()” decision code block. Each “case” in the switch block represents a different EVAL call in the original Linda source program.

To accommodate this approach, the Linda compiler has been modified to assign a unique ID number to each EVAL call encountered in the Linda source code. The Linda compiler uses this EVAL call ID as the basis for the switch() block. That is, the case_main() function first determines which EVAL is to be executed by obtaining an EVAL call ID from the Linda system (as more fully described in the next section), and then calls the appropriate functions which implement the corresponding EVAL based on the value of that EVAL call ID. This is illustrated by the following pseudocode fragment:

```c
    case_main() { 
        get new EVAL_id 
        switch ( EVAL_id ) { 
```
case 1: make calls for EVAL 1
case 2: make calls for EVAL 2
...
}
}

Obtaining an EVAL ID from the Linda System

When a remote process has been instantiated to handle an EVAL operation, the first information it requires from the parent process (i.e. the process that called EVAL) is the ID of the particular EVAL which it is intended to handle. Therefore, when a process executing a Linda program encounters an EVAL call, the first thing it must do is impart the corresponding EVAL call ID to the process which will actually execute the EVAL code.

The semantics of the EVAL operation suggest that the process calling the EVAL place this piece of information in Tuple Space, rather than somehow pass it as a parameter to the remotely instantiated process or send it by means of some new protocol. If the process calling the EVAL simply OUTs the EVAL call ID to Tuple Space at the time of the EVAL call, then the remote process can IN the EVAL call ID at the time of EVAL execution. In this way, the remote process "inherits" from its parent process information which specifies the part of the Linda code which it must execute. (In the Yale and D-Kernel Linda systems, this issue does not arise, because the information is known implicitly by the new process, through those systems' use of the fork() call.) Since the new Linda process obtains the EVAL ID from the globally shared Tuple Space, this process has no need to communicate directly with the process that called EVAL.
*Shielding EVAL ID tuples from user operations*

The OUT and IN calls which send EVAL IDs to Tuple Space and retrieve them from Tuple Space are different from OUTs and INs found in user programs. Internally, the distributed Linda compiler uses its own functions to handle EVAL call IDs. Aptly, these functions are called `out_eval_id()` and `in_eval_id()`. These functions differ from user-called OUTs and INs only in that they manipulate a reserved area of Tuple Space which is not accessible to user operations. Each EVAL ID tuple contains a single integer field which contains the ID of the EVAL call. Because of the consistent structure of all EVAL ID tuples, `out_eval_id()` and `in_eval_id()` can be used as reusable implementations of OUT and IN operations for manipulating the special EVAL ID area of Tuple Space.

The approach described in this section effectively partitions the preprocessed source code into `real_main()` and `case_main()`. By organizing the code this way and transferring EVAL IDs through Tuple Space, we have addressed the problem of how to inform a new independent process of the names of the functions it is to call to evaluate a specific active tuple. These functions are identified by a corresponding ID number, which the process obtains from Tuple Space. By transferring EVAL IDs through Tuple Space using Linda's OUT and IN operations, the distributed system takes advantage of Linda's existing interprocess communication (IPC) facilities.

### 4.4 “Inheritance” of the Proper Execution Environment by the New Process

As discussed, when a new Linda process is instantiated, it must be able to inherit the execution environment of its parent process. Again, Linda's IPC facilities are employed to transfer this information from "parent" to "child" process. As stated, this approach avoids
the need to design and implement a new set of protocols to transfer the necessary transformation specifications to the new Linda process.

The remaining specifications needed by the remote process are the execution parameters for the EVAL at runtime. These are the values of the fields of the active tuple, i.e., the arguments to the EVAL call. As with the EVAL call ID, this argument list can be OUT-ed to Tuple Space, and the remote Linda process can later IN the arguments. However, since the number and types of the fields of active tuples can vary from EVAL call to EVAL call, the argument list must be uniquely associated with the corresponding EVAL call ID before being sent to Tuple Space. Similarly, the remote Linda process must use the EVAL ID in its request for the corresponding argument list from Tuple Space.

This association is accomplished by prepending the EVAL call ID to the argument list tuple as a new field before the tuple is sent to Tuple Space. The remote process then prepends the EVAL ID (previously retrieved from Tuple Space with an IN call) to its request for the active tuple's argument list. For example, if the EVAL ID retrieved by the remote process is 1, then the process can request the argument list for the active tuple whose EVAL ID is 1.

The code for the EVAL call is conceptually redefined as follows:

```
EVAL(tuple)
```

is translated into

```
OUT(EVAL_id)
OUT(EVAL_id, tuple_fields)
```
The `case_main()` code is then generated as follows:

```c
IN( EVAL_id )
switch( EVAL_id ) {
  case 1:
    IN( 1, tuple_1_fields )
    OUT( tuple_1 )
    break
  case 2:
    IN( 2, tuple_2_fields )
    OUT( tuple_2 )
    break
...
}
```

It is essential that the EVAL call IDs be sent to and retrieved from Tuple Space separately from the argument lists. If the `EVAL(tuple)` call were implemented simply as `OUT(EVAL_id, tuple_fields)`, then the remote process would have no way of knowing which particular `IN(EVAL_id, tuple_fields)` function to invoke that has a corresponding tuple in Tuple Space. Not only must the EVAL ID match, so must the number and types of the tuple fields. Our solution as described above allows a remote process to first retrieve an active tuple ID from Tuple Space. If more than one active tuple is waiting in Tuple Space for evaluation, it doesn’t matter which one is retrieved. The remote process will be able to correctly retrieve the corresponding execution parameters by calling the `IN(EVAL_id, tuple_fields)` function with the matching EVAL ID.
Function arguments to EVAL operations

As discussed in Section 2.1, the EVAL operation is used in a Linda program to create concurrent processes for the purpose of executing functions. Typically, at least one of the fields of an EVAL’s argument list is a call to a function. This is referred to as an “EVAL-ed function”. Ideally, the address of such a function would be transferred to the new remote Linda process by sending a pointer to the function as a member of the EVAL operation’s argument list. However, because pointers cannot be passed meaningfully through Tuple Space to remote processes (since a remote process does not share code address space with the process which requested its instantiation), only the values of any arguments to EVAL-ed functions (along with the values of any non-function fields) are included in such an active tuple’s argument list.

For example, consider the following:

\[
\text{EVAL}( f( i ) )
\]

If we assume that the ID of this EVAL call is 3, then this EVAL call is translated into the following operations:

\[
\text{OUT}( 3 ) \quad \text{OUT}( 3, i )
\]
The corresponding `case_main()` code would be:

```c
IN( ?EVAL_id )
switch( EVAL_id ) {
  ...
  case 3:
    IN( 3, ?i )
    OUT( f( i ) )
    break
  ...
}
```

Note that the function to be EVAL-ed, `f(i)`, appears in the OUT operation in the `case_main()` code, rather than in the call to EVAL. The semantics of an OUT require that this function be evaluated before the corresponding passive tuple is placed in Tuple Space. But, this is exactly what we want because the remotely instantiated process is executing the `case_main()` code.

*Shielding EVAL argument list tuples from user operations*

As with EVAL ID tuples, EVAL argument list tuples must not be accessible to Linda operations in the user’s program. Unlike EVAL ID tuples, the structure of each EVAL argument list tuple can be unique. That is, the fields of these tuples can vary in number and type. As described, the Linda compiler manages the problem of varying numbers and types of tuple fields by generating a new unique C function for each Linda operation. We want to use this capability of the Linda compiler in managing EVAL argument list tuples.

The Linda compiler analyzes all the Linda operations in a user’s Linda program, and places them together in various “sets”. These sets are defined by the number and types of
the fields found in each tuple and template. All tuples and templates belonging to a given set must have the same number of fields, and fields in a given field position must have the same data type throughout the tuple set.

To shield EVAL argument list tuples from user operations, the distributed Linda compiler has been modified to force EVAL argument list tuples into tuple sets which do not also contain user tuples. This prevents user tuples and Linda operations from interfering with the EVAL mechanism.

4.5 A More Complete Example

We can now demonstrate more precisely how an EVAL operation will be translated by the Linda compiler. Consider again the following example:

\[
\text{eval}( f( i ) )
\]

The Linda compiler transforms this operation into a new C function and a corresponding call to the new function. The Linda compiler ensures that the names of these new functions do not conflict with the names of any user or library functions. For simplicity, we will assume that the name of the C function generated by the Linda compiler for this EVAL operation is “new_eval”. The call to the EVAL operation is then transformed into:

\[
\text{new_eval}( i )
\]

If this EVAL operation’s ID is 6, then the new_eval() function is generated by the compiler as follows:
new_eval ( int arg1 ) {
    out_eval_id( 6 );
    new_eval_a( arg1 );
}

As described, the `out_eval_id()` function sends the EVAL operation's ID to Tuple Space:

```c
out_eval_id( int arg1 ) {
    tuple_type t;
    t.nbr_fields = 1;
    t.field[1] = arg1;
    out_eval( &t );
}
```

`out_eval()` is one of the functions in the Linda run time library which uses D-Kernel's stream socket communication interface. It specifically addresses the request to the reserved EVAL area of Tuple Space and writes the request to the socket.

The `new_eval_a()` function is also generated by the compiler. It sends the EVAL's argument list to Tuple Space as follows:

```c
new_eval_a( int arg1 ) {
    tuple_type t;
    t.nbr_fields = 2;
    t.field[1] = 6;
    t.field[2] = arg1;
    out_partition( &t );
}
```
Notice that the EVAL ID 6 has been inserted into this tuple at field position 1. Similar to out_eval(), the out_partition() library function addresses the request to a specific part of Tuple Space and writes the request to the socket. "Partition" is one of three logical partitions of Tuple Space. These partitions are described in Chapter 5.

The following pseudocode represents the case_main() function:

```c
void case_main() {
    int op_id;
    in_eval_id( &op_id );
    switch( op_id ) {
        ...
        case 6: {
            int arg1;
            new_eval_b( &arg1 );
            new_eval_c( arg1 );
            break;
        }
        ...
        default: {
            break;
        }
    }
}
```
The `in_eval_id()` function is the counterpart to `out_eval_id()`. It obtains an EVAL operation ID from the EVAL area of Tuple Space:

```c
in_eval_id( int *arg1 ) {
    tuple_type t;
    t nbr_fields = 1;
    t field[1] = arg1;
    in_eval( &t );
}
```

The first (and only) field of the template sent to Tuple Space by this function is a formal field. When `in_eval()` returns, the value of the variable pointed to by `arg1` is the EVAL ID of an active tuple waiting for evaluation in Tuple Space.

Similarly, the `new_eval_b()` function is the counterpart to `new_eval_a()`. It retrieves the EVAL operation's argument list from TS.

```c
new_eval_b( int *arg1 ) {
    tuple_type t;
    t nbr_fields = 2;
    t field[1] = 6;
    t field[2] = arg1;
    in_partition( &t );
}
```

Again, the first field of the template is the EVAL ID. The second field is a formal field which specifies the variable in which to place the corresponding field from the matching tuple in Tuple Space.
Finally, the `new_eval_c()` function actually evaluates the newly retrieved active tuple, and places the resulting passive tuple back into TS:

```c
new_eval_c( int arg1 ) {  
tuple_type t;  
t.nbr_fields = 1;  
t.field[1] = f( arg1 );  
out_partition( &t );
}
```

This example demonstrates that each EVAL operation in a Linda program is transformed by the distributed Linda system's compiler into four new functions:

1. The original EVAL call is translated into a call to a new function, which we refer to as `new_eval()`. The `new_eval()` function calls `out_eval_id()` and `new_eval_a()`.

2. The `new_eval_a()` function sends the EVAL's argument list to TS.

3. The `new_eval_b()` function retrieves the EVAL's argument list from TS.

4. The `new_eval_c()` function evaluates the active tuple, and places the resulting passive tuple in TS.

In addition, the compiler generates the code for a "case" in the `switch()` block of `case_main()`. This case code actually calls `new_eval_b()` and `new_eval_c()`.

Each of these compiler-generated functions uses existing Tuple Space access mechanisms to transfer the necessary transformation specifications to new remote Linda processes. Essentially, we have used Linda itself to implement a distributed version of Linda.
5.0 Data Subsystem Components and their Interaction

This chapter presents the components of the Linda-LAN's data subsystem. Section 5.1 presents TS-Man and the Split Tuple Space version of the distributed system. Section 5.2 describes the purpose and design of L-Kernel. Finally, Section 5.3 discusses issues related to the interaction between the Linda process, L-Kernel, and TS-Man.

5.1 TS-Man and Split Tuple Space

As stated in 4.1.1, the TS-Man component of the Linda-LAN's data subsystem was modeled directly from the D-Kernel implementation's Tuple Space Manager. This section lists the major functions of TS-Man and provides an overview of its code organization. Next, we briefly discuss the internal partitions of Tuple Space. This discussion serves as background to a presentation of the Split Tuple Space version of the distributed system, which provides for concurrent access to the various Tuple Space partitions.

Functions of the TS-Man component

- Accept connection requests from L-Kernels. These connections are established as part of the process of starting up the data subsystem, before the user's Linda program begins execution. (Data subsystem startup is detailed in Appendix A.)

- Accept and service requests for access to Tuple Space. These requests originate with Linda processes, and are forwarded to TS-Man by the L-Kernels.

- Notify L-Man of the arrival of all active tuples.
Overall Design of TS-Man

TS-Man is implemented as a single interrupt-driven loop, as shown by the following pseudocode:

```c
while ( not done ) {
    select( set of connected L-Kernel sockets )
    for ( each L-Kernel socket ) {
        if ( data is available on socket ) {
            service_request( socket )
        }
    }
}
```

TS-Man waits for data to arrive from any of the connected L-Kernels. When data arrives on any socket, `select()` returns and TS-Man processes requests on all sockets which currently have data available to be read. TS-Man processes several types of requests from Linda processes, including IN, RD, OUT, and EVAL:

- When TS-Man receives an IN or RD request, it searches Tuple Space for a matching tuple. If no matching tuple is found, the request is placed on a “wait list”, where it remains until a matching tuple arrives in Tuple Space.

- When TS-Man receives an OUT request, it first checks the wait lists for matching RD and IN requests. If the newly arrived tuple is not consumed by a waiting IN request, then TS-Man places it in Tuple Space.

- When TS-Man receives an EVAL request, it places the new tuple in the reserved EVAL ID area of Tuple Space, and sends notification to L-Man that an active tuple awaits evaluation in TS.
TS-Man’s request servicing function is represented by the following pseudocode:

```plaintext
service_request() {
    read request from socket
    if ( request == (IN or RD) ) {
        if ( a matching tuple is found in TS )
            return matching tuple
        else
            place request on appropriate wait list (IN or RD)
    }
    else if ( request == OUT ) {
        for ( each request on the RD wait list ) {
            if ( new tuple matches RD request )
                return copy of new tuple
        }
        if ( a matching request is found on the IN wait list )
            return new tuple
        else
            place new tuple in TS
    }
    else { /*--- ( request == EVAL ) ---*/
        place new tuple in EVAL area of TS
        notify L-Man
    }
}
```
**Tuple Space Partitions**

As discussed in section 4.4, the Yale and D-Kernel systems' Linda compiler places Linda operations together into sets. In addition, the compiler assigns one of three tuple management strategies to each tuple set, based on specific characteristics of the tuples and templates in each set. TS-Man employs a different tuple storage data structure and tuple searching and matching method for each of the tuple management strategies. These different data structures and access methods comprise three distinct partitions in Tuple Space. The three strategies, named for the type of algorithm they use, are Counting Semaphore, Queue, and Hash.

The primary reason for the various tuple management strategies and corresponding Tuple Space partitions is to optimize matching of tuples and templates. Schumann gives a thorough discussion of the internal structure of Tuple Space in [SCHC93].

Section 4.3 introduces a reserved area of Tuple Space, which is used only for EVAL ID tuples. This reserved area is actually a fourth Tuple Space partition: the Eval Partition.

**Split Tuple Space**

Since Tuple Space is already divided into disjoint partitions in the Yale and D-Kernel implementations, we decided to take advantage of this partitioning to exploit concurrency. That is, we have created a Split Tuple Space version of the distributed system. In this system, a separate Tuple Space Manager process is run for each distinct partition of Tuple Space. Running each of these TS-Man processes on a separate machine of the Linda-LAN provides for concurrent access to the various partitions of Tuple Space. For example, one Linda process can access the Queue Partition while another Linda process simultaneously accesses the Hash Partition.
Conveniently, the organization of the previous system's Tuple Space and TS-Man program has made the actual construction of the components of Split Tuple Space relatively easy. The major problems which had to be addressed involved the interaction of the various TS-Man processes with the L-Kernels, with each other, and with the control subsystem (which we did not want to modify for this purpose). These considerations are discussed in section 5.3.

Figure 5.1 depicts a conceptual view of the Split Tuple Space version of the distributed system.

5.2 L-Kernel

This section provides an overview of the function and design of the L-Kernel component of the data subsystem, and describes how L-Kernel handles commands and requests.

*Functions of the L-Kernel component*

We stated in 3.2.3 that the primary *purpose* of the L-Kernel component of the data subsystem is to reduce the number of stream socket connections required between TS-Man and Linda processes. This is accomplished by multiplexing requests from Linda processes, and demultiplexing responses from TS-Man. That is, all Linda processes executing on a Linda-LAN machine send their requests for Tuple Space access to L-Kernel. L-Kernel reads these requests from multiple sockets and forwards them to TS-Man on a single socket (or on one socket per TS-Man process in the Split Tuple Space system).
Figure 5.1. Conceptual topology of Split Tuple Space Linda-LAN
Overall Design of L-Kernel

L-Kernel communicates through socket connections with various components in both the control and data subsystems. These components include:

- **L-Com** - L-Kernel receives several commands from L-Com, including
  handle active tuple and system shutdown.

- **TS-Man** - L-Kernel forwards all tuples and requests for tuples to TS-Man through this socket, and receives all responses from TS-Man from this socket.

- **Linda processes** - L-Kernel has a socket connection to each Linda process on the machine on which L-Kernel is running.

L-Kernel is implemented as an interrupt-driven loop, as shown by this pseudocode:

```pseudocode
while ( not done ) {
    select ( set of connected sockets ) {
        if ( data is available from L-Com )
            service L-Com socket
        if ( data is available from TS-Man )
            service TS-Man socket
        for ( each Linda process socket )
            if ( data is available from Linda process socket )
                service Linda process socket
    }
}
```

L-Kernel waits for data to arrive from any of the connected processes. When data arrives on any socket, `select()` returns and L-Kernel processes requests on all sockets which currently have data available to be read.
L-Kernel services its socket connections in the following order:

1. L-Com socket
2. TS-Man socket
3. Linda process sockets

The sockets are serviced in this order because of the relative importance we attach to the data arriving from their respective processes (L-Com, TS-Man, and Linda processes).

"Handle Active Tuple" command from L-Com

As described in section 4.2, when L-Kernel receives a handle active tuple command from its L-Com, it calls the Unix fork() system call to create a copy of itself. The original (parent) L-Kernel returns to service the next socket. The newly created (child) copy of L-Kernel closes all its socket connections (since it inherits these from the parent process), and then immediately invokes the Unix exec1() system call to replace the L-Kernel code with Linda process code.

Header and Data Packets

One of L-Kernel's main tasks is to handle requests from both Linda processes and TS-Man. Before discussing L-Kernel's handling of these requests, it should be made clear that each request from a Linda process (whether tuple or template) and each response from TS-Man is composed of a "header" packet, which is of fixed length, and a "data" packet, which is of variable length. The length of the data packet is stored as a field in the header packet. Therefore, reading a request from a socket involves the following steps:
(1) Read the header packet, which is always the same size

(2) Extract the header’s field which specifies the length (in bytes) of the data packet,

(3) Read the appropriate number of data packet bytes from the socket

(4) If necessary, reconstruct the tuple or template.

This scheme prevents data belonging to different requests from becoming scrambled, and enables all the processes in the data subsystem to determine the number of data bytes to attempt to read from a socket for each request.

*L-Kernel’s Handling of Requests*

When L-Kernel determines that a request is waiting to be read from a Linda process or from TS-Man, it services that request by performing the following sequence of actions:

(1) Read the header packet from the appropriate socket,

(2) Forward the header packet to the destination process by writing it to the corresponding socket,

(3) Read the appropriate number of data packet bytes from the socket into an unstructured block of memory,

(4) Forward the data packet to the destination process.

This scheme is advantageous in two respects:

- It exploits concurrency by sending a request’s header packet to the destination process before reading the potentially large data packet. Consequently, the destination process,
which may be otherwise idle, can begin processing the data before it all has been sent by L-Kernel.

- L-Kernel does not need to understand the internal structure of tuples and templates. It simply reads the proper number of bytes from one socket, and writes them to another. As a result, L-Kernel can forward data more quickly.

### 5.3 Interaction Between Data Subsystem Components

This section discusses several issues related to the interaction between the various components of the data subsystem. These issues include:

- Directing responses from TS-Man back to the proper Linda process.

- Adapting L-Kernel and TS-Man to the Split Tuple Space version of the system

*Directing Responses from TS-Man Back to the Proper Linda Process*

The D-Kernel system's Tuple Space Manager (called "dkernel") is able to respond directly to a retrieval request since in that system, Linda processes have direct socket connections to dkernel. One of the fields of the data structure into which dkernel reads each request is the socket from which the request is read. When a retrieval request is matched with a tuple, that tuple is returned to the Linda process which sent the request by writing the response (which includes the matching tuple) to the same socket from which the request was originally received.

However, the distributed system's TS-Man reads requests from all Linda processes executing on a given machine from the same socket. This socket is connected to the L-Kernel process on that machine. A Linda process sends a TS access request to its L-Kernel, rather
than directly to TS-Man. L-Kernel forwards each request to TS-Man. When TS-Man is ready to respond to a request, it simply writes the response back to the same socket from which the request was originally read, just as in the D-Kernel system. This results in the response being delivered to the proper L-Kernel, but the problem now is for L-Kernel to determine which of its connected Linda processes is the proper recipient.

This problem is solved by adding a "Linda process socket" field to TS-Man’s request and response storage structure, and by using this same structure in the L-Kernel. When L-Kernel receives a request from a Linda process, it stores in the "Linda process socket" field the address of the socket from which the request is read, and then forwards the request to the appropriate TS-Man. When TS-Man receives the request, it stores the address of the L-Kernel socket from which the request is read in the "L-Kernel socket" field. This is the socket to which it sends its response. On receiving the response, L-Kernel forwards it to the socket specified in the "Linda process socket" field.

Adapting L-Kernel and TS-Man to the Split Tuple Space version of the system

As stated in section 5.1, the difficulties in creating the Split Tuple Space version of the distributed system involve the interaction of the various TS-Man processes with other components in the Linda-LAN. The issues which had to be addressed include the following:

1. L-Man (a member of the control subsystem) creates only a single TS-Man process. A mechanism must be designed for creating the other TS-Man processes.

2. Each L-Kernel must connect to each TS-Man process during data subsystem start-up. However, L-Com, a member of the control subsystem, informs L-Kernel only of the machine name and socket address of a single TS-Man component. We must design a mechanism for connecting L-Kernels to the other TS-Man processes.
(3) The manager of the Eval Partition of TS must notify L-Man when active tuples arrive, and therefore must have a socket connection to L-Man.

(4) L-Kernels must now examine each request from Linda processes to determine the destination TS-Man process.

The design for the Split Tuple Space version of the distributed system addresses each of these issues as follows:

(1) Since L-Man is involved with TS-Man in two respects (data subsystem start-up and active tuple notification), we decided that the TS-Man process started by L-Man would be the one to manage the Eval Partition, and would be responsible for starting the other TS-Man processes as part of its own start-up routine.

(2) When each of the other TS-Man processes starts, it creates a socket for communicating with L-Kernels, and informs the Eval TS-Man of the address of this socket. The Eval TS-Man also creates an L-Kernel socket, and informs L-Man of its address. When an L-Kernel is created, it connects to the Eval TS-Man. As part of the initialization between these two processes, the Eval TS-Man informs L-Kernel of the host names and socket addresses of the other TS-Man processes.

(3) When L-Man starts TS-Man, it informs TS-Man of its own host name and socket address. Since the TS-Man started by L-Man has been designated as the Eval TS-Man, it connects to L-Man as part of its initialization. When active tuples arrive, the Eval TS-Man notifies L-Man of this fact over this communications link.

(4) When L-Kernel receives requests for access to Tuple Space from connected Linda processes, it must first read the header packet. From the header, it extracts not only the length of the data packet, but also the type of request. The request type dictates
which Tuple Space partition is to be addressed by the request, and hence, the destination TS-Man process.
6.0 Execution Dynamics: A Comprehensive Example

To complete the overall picture of the distributed system's provision for concurrent computation, we present an example EVAL call, and list the series of events and the flow of information which result from the call. Use of the Split Tuple Space version of the distributed Linda system is assumed.

Once again, consider the following example Linda code fragment (examined in section 4.5) which includes a single Eval call having a single function argument:

```c
int f( int i );
real_main() {
    int i;
    ...
    eval( f( i ) );
    ...
}
```

The `main()` function resides in the distributed system's run time library, and is represented by the following pseudocode:
/**-- main() is always the point of entry --*/
main() {
    if ( getpid() == 1 ) {  
        case_main();
        /**<-- drop the connection to L-Kernel --*/
        case_exit();
    }
    else {
        real_main();
        /* drop L-Kernel connection, causing system shutdown */
        lexit();
    }
}

The Linda compiler's preprocessor generates C code from the Linda program. That C code is represented by the following pseudocode:

 /**<-- real_main() is the user's "main" Linda program --*/
real_main() {
    int i;
    ...
    new_eval( i );
    ...
}

/**<-- the eval() was transformed into new_eval() --*/
new_eval( int arg1 ) {
    /**<-- assume this is the 6th EVAL encountered --*/
    out_eval_id( 6 );
    new_eval_a( arg1 );
}
/**--- out_eval_id() sends the EVAL ID to TS ---*/
out_eval_id( int arg1 ) {
    tuple_type t;
    t.nbr_fields = 1;
    t.field[1] = arg1;
    out_eval( &t );
}

/*--- new_eval_a() sends the EVAL’s argument list to TS ---*/
new_eval_a( int arg1 ) {
    tuple_type t;
    t.nbr_fields = 2;
    t.field[1] = 6;
    t.field[2] = arg1;
    out_queue( &t );
}
case_main() { 
    int op_id;
    in_eval_id( &op_id );
    switch ( op_id ) {
        ...
        case 6: {
            int arg1;
            new_eval_b( &arg1 );
            new_eval_c( arg1 );
            break;
        }
        ...
        default:
            break;
    }
}

in_eval_id( int *arg1 ) { 
    tuple_type t;
    t.field[1] = arg1;
    t.nbr_fields = 1;
    in_eval( &t );
}
/*--- new_eval_b() retrieves the EVAL's arg list from TS ---*/
new_eval_b( int *arg1 ) {
    tuple_type t;
    t.nbr_fields = 2;
    t.field[1] = 6;
    t.field[2] = arg1;
    in_queue( &t );
}

/----------

new_eval_c() evaluates the active tuple, and places
the resulting passive tuple into TS

----------*/
new_eval_c( int arg1 ) {
    tuple_type t;
    t.nbr_fields = 1;
    t.field[1] = f( arg1 );
    out_queue( &t );
}

Figure 6.1 depicts the "hierarchy" of the function calls represented by the above pseudocode.

When the above code is compiled and executed (after the Linda-LAN Control and data
subsystems have been started), it proceeds through the sequence of events listed on the fol-
lowing pages. Figure 6.2 depicts steps 1 through 15, and 36 through 37, and Figure 6.3
depicts steps 16 through 35.
Figure 6.1. Function call hierarchy in preprocessed Linda code
Figure 6.2. Eval process: Steps 1 through 15, and 36 through 37
Figure 6.3. Eval process: Steps 16 through 35
Steps 1 through 8 are executed by the main Linda program:

1. The main Linda program begins execution of the code at the main() function, as does any C program.

2. main() determines that the process id of its parent process is not that of L-Kernel.

3. main() calls real_main().

4. real_main() calls new_eval(), passing it the value of the integer variable i as a parameter.

5. new_eval() calls out_eval_id(), passing it the ID number of this particular Eval operation, which is 6 in this case.

6. out_eval_id() constructs a tuple containing the Eval ID 6, addresses it to the Eval TS-Man process, and sends it (with the out_eval() library function) to the local L-Kernel.

7. new_eval() then calls new_eval_a(), passing it the value of its own arg1, which is the value of real_main()’s integer variable i.

8. new_eval_a() constructs a tuple containing 2 fields. The first field contains the Eval ID 6 (allowing the Linda system to relate the Eval ID tuple with the tuple being constructed in this step). The second field contains the value of real_main()’s integer variable i. This tuple contains the execution parameters for the active tuple being placed in Tuple Space by this Eval operation. It is addressed to the Queue TS-Man process, and sent (with the out_queue() library function) to the local L-Kernel.
Steps 9 through 15 are executed by components of the Linda-LAN:

(9) L-Kernel reads the Eval ID tuple's header packet, determines that the tuple is destined for the Eval TS-Man, and forwards the header packet to the Eval TS-Man. L-Kernel then reads the Eval ID tuple's data packet, and forwards it to the Eval TS-Man.

(10) L-Kernel reads the tuple containing the execution parameters for the active tuple, determines that the tuple is destined for the Queue TS-Man, and forwards the header packet to the Queue TS-Man. L-Kernel then reads this tuple's data packet, and forwards it to the Queue TS-Man.

(11) The Eval TS-Man reads the Eval ID tuple, stores it in Tuple Space, and sends notification to L-Man that an active tuple awaits evaluation in TS.

(12) The Queue TS-Man reads the tuple containing the execution parameters for the active tuple, and stores it in Tuple Space.

(13) L-Man selects a participating Linda-LAN machine, and informs that machine's L-Com that it is to evaluate an active tuple.

(14) L-Com forwards this command to its L-Kernel.

(15) L-Kernel receives the command and creates a copy of itself with fork(). The new copy of L-Kernel replaces itself with the Linda program code.

Steps 16 through 20 are executed by the newly instantiated Linda process:

(16) The new Linda process begins execution at the main() function.

(17) main() determines that the process id of its parent process is that of L-Kernel.
(18) main() calls case_main().

(19) case_main() calls in_eval_id(), passing it the address of op_id.

(20) in_eval_id() constructs a template to request retrieval of an Eval ID tuple from TS, addresses it to the Eval TS-Man, and sends it (with the in_eval() library function) to the local L-Kernel.

Steps 21 through 23 are executed by components of the Linda-LAN:

(21) L-Kernel reads the Eval ID request's header packet, determines that the request is destined for the Eval TS-Man, and forwards the header packet to the Eval TS-Man. L-Kernel then reads the request's data packet, and forwards it to the Eval TS-Man.

(22) The Eval TS-Man reads the request for an Eval ID tuple, finds the next available Eval ID tuple in TS, removes the tuple from TS, fills in the request template with the fields from this tuple, and returns the response tuple to the L-Kernel from which it originally read the request.

(23) L-Kernel reads the Eval ID tuple's header packet from the Eval TS-Man, determines which Linda process the tuple is destined for, and forwards the header packet to that Linda process. L-Kernel then reads the tuple's data packet and forwards it to the Linda process.

Steps 24 through 26 are executed by the new Linda process:

(24) The Linda process reads the Eval ID response tuple from L-Kernel. When in_eval_id() returns, the integer variable op_id contains the ID of an active tuple awaiting evaluation in TS (which is 6, in this case). case_main() executes
the `switch()` statement, and based on the value of this `op_id`, selects "case 6" for execution.

(25) case 6 defines the integer variable `arg1` to hold this particular active tuple’s execution parameters, and then calls `new_eval_b()`, passing it the address of `arg1`.

(26) `new_eval_b()` constructs a template to request retrieval of the execution parameters for the active tuple represented by the Eval ID 6, addresses it to the Queue TS-Man, and sends it (with the `in_queue()` library function) to the local L-Kernel.

*Steps 27 through 29 are executed by components of the Linda-LAN:*

(27) L-Kernel reads the request’s header packet, determines that the request is destined for the Queue TS-Man, and forwards the header packet to the Queue TS-Man. L-Kernel then reads the request’s data packet, and forwards it to the Queue TS-Man.

(28) The Queue TS-Man reads the request, finds the matching tuple in TS, removes the tuple from TS, fills the request template with the fields from this tuple, and returns the response tuple to the L-Kernel from which it originally read the request.

(29) L-Kernel reads the response tuple’s header packet from the Queue TS-Man, determines which Linda process the response is destined for, and forwards the header packet to that Linda process. L-Kernel then reads the response’s data packet and forwards it to the Linda process.
Steps 30 through 33 are executed by the new Linda process:

(30) The Linda process reads the response tuple from L-Kernel. When
new_eval_b() returns, arg1 will contain the value of the main Linda
program's integer variable i. Case 6 of case_main()'s switch() statement
then calls new_eval_c(), passing it the value of arg1.

(31) new_eval_c() calls the user function f(), passing it the value of arg1. This
is the action specified in the original Eval call in the main Linda program's source
code.

(32) new_eval_c() uses the return value from f() to construct a passive tuple with
a single integer field, addresses it to the Queue TS-Man, and sends it (with the
out_queue() library function) to the local L-Kernel.

(33) case_main() returns to main(), and main() calls case_exit() to
disconnect the Linda process from its L-Kernel and exit.

Steps 34 and 35 are executed by components of the Linda-LAN:

(34) L-Kernel reads the passive tuple's header packet, determines that the tuple is
destined for the Queue TS-Man, and forwards the header packet to the Queue
TS-Man. L-Kernel then reads the passive tuple's data packet, and forwards it to the
Queue TS-Man.

(35) The Queue TS-Man reads the passive tuple, and stores it in Tuple Space.
Steps 36 and 37 are executed by the main Linda program:

(36) `real_main()` returns to `main()`

(37) `main()` calls `lexit()` to inform L-Kernel that the program has run to completion, and then exits.
7.0 Investigation of Execution Profile

This chapter presents the results of the preliminary investigation of the execution profile of the distributed Linda system. The goal of this investigation is to identify characteristics of the system that tend to enhance or limit computational performance. More specifically, our objective is to identify critical aspects and characteristics of components of the distributed system and to assess their impact on system performance given a number of different scenarios.

Section 7.1 presents and discusses the initial results obtained by executing two Linda applications with different characteristics on the system. Section 7.2 briefly describes a potential technique for enhancing the performance of tuple I/O in the Linda system, and presents the results obtained when this technique is implemented. Finally, Section 7.3 discusses the details and impact of the modification introduced in 7.2, and describes three possible alternative designs for the Linda system’s communications interface.

First, we describe the basic Linda-LAN configuration selected for the investigation, and present our general expectations for the distributed system’s performance.

Linda-LAN Configuration

Although both Single and Split Tuple Space versions of the distributed Linda system have been constructed, the results presented in this thesis are limited to those obtained by running Linda applications on the Split Tuple Space system. As discussed in Section 5.1, we assume that the Split Tuple Space system will more fully exploit concurrency.
In our investigation we employ a network of six NeXT\textsuperscript{1} uniprocessor workstations. It is assumed that placing each of the various TS-Man processes on a separate machine of the Linda-LAN will yield the highest degree of concurrency. However, our primary objective in this preliminary investigation is to gain insight into the performance of the system as the number of processors and the number of processes is varied. Therefore, one of the workstations is dedicated to L-Man and the various TS-Man processes, and the other five machines are reserved for executing Linda processes. This configuration is depicted in Figure 7.1.

\textit{Expectations for the Distributed System}

- In general, we expect that distributing Linda processes \textit{over multiple processors} will result in increased performance, since this will provide for truly parallel process execution. Distributing processes over several processors requires that the programmer partition the Linda program's task into a number of independent processes. The expectation for enhanced performance also assumes that the computational work to be done by each of the Linda processes is significant in comparison with the overhead work required to distribute them. Although this overhead has not been quantified, it is understood that a nontrivial cost is associated with invocation of the EVAL mechanism (described in Chapter 4).

- We also expect that distributing work \textit{to an increasing number of Linda processes} on each Linda-LAN processor can yield better performance in certain circumstances. In particular, if the work which is to be done by the Linda processes consists of a mix of computation and I/O (specifically, I/O with Tuple Space), then distributing the work to several processes on each machine can enhance performance. This potential speedup

\textsuperscript{1} NeXT is a trademark of NeXT Computer, Inc.
Figure 7.1. Linda-LAN configuration used in the investigation
exists since the CPU can be used by one Linda process while others are awaiting completion of I/O requests. However, we recognize that there is saturation point beyond which adding more processes will actually decrease performance, due to thrashing. This phenomenon occurs when there are so many competing processes that the system spends more time scheduling processes and swapping pages in and out of memory than actually executing the processes’ code [LEFS89].

7.1 Initial Execution Time Results

This section presents the results obtained by running two different Linda applications on the distributed system: the rayshade ray-tracing program and the cmatrix matrix multiplication program.

Results from the rayshade Application

Rayshade is a computationally intensive ray tracing program which reads a multi-line ASCII text file describing a scene composed of a number of primitive objects. It then renders the scene, producing a Utah Raster RLE (“run length encoding”) format file of the ray-traced image. Specifically, the main Linda process functions as a supervisor, and the worker processes perform all the computation. The supervisor creates the workers with eval(), and then retrieves the result lines of the computed RLE image and writes them to the output file. Each worker reads the input file, and then computes result image lines until all lines have been computed. Each time a worker wants to compute a new result line, it retrieves the next line tuple from Tuple Space with in(), increments its value, and places it back in Tuple Space. It then computes the line, places it in Tuple Space, and again tries to retrieve the next line tuple.
Figure 7.2 presents a graph which reflects the execution times obtained from test runs of rayshade. The input description file used for these runs specifies a relatively simple scene consisting of only a few objects. The test was run using 5, 10, and 15 worker processes. From the graph, we observe the following:

- As the number of processors increases from 1 to 5, execution time decreases. Specifically, execution time decreases at a rate roughly proportional to the rate of increase of number of processors. For example, 2 processors are twice as many processors as 1, and the corresponding execution times decrease from an average of 310 seconds in the 1 processor case to an average of 168 seconds (which is roughly half of 310) in the 2 processor case. Hence, rayshade makes very efficient use of additional processors.

- Adding more Linda processes has a slight negative effect on rayshade's execution time. That is, as more processes are used, execution time actually increases. The fastest execution time on this graph represents a run in which each of 5 processors is running only 1 Linda process. We speculate that rayshade's performance does not benefit from additional processes per processor because it performs relatively little I/O with Tuple Space. Most of its work is computation.

Figure 7.3 displays the same set of results, but emphasizes the fact that an increase in the number of processors, rather than in the number of Linda processes, has the greatest (positive) effect on execution times.

These findings support our expectations described in Section 7.1. In particular, since rayshade's primary work is computation (rather than I/O), we expect that adding more processors should result in increased performance. Moreover, we do not necessarily
Figure 7.2. Execution times from rayshade application
Figure 7.3. Another view of the times from *rayshade*
expect multiple Linda processes on each workstation to enhance performance, for the following reasons:

(1) The workstations are uniprocessors and hence, can execute only one process at a time. Multiple processes all must compete for the computational power and time of the CPU.

(2) There is relatively little I/O to be done by the Linda processes. In the case of rayshade, multiple Linda processes on each processor results in an increase in context switching activity without a corresponding increase in CPU utilization.

In general, the rayshade application exhibits performance characteristics which are consistent with our expectations for the distributed Linda system.

Results from the cmatrix Application

The next set of execution times was obtained from a matrix multiplication application called cmatrix. This program is designed to demonstrate how a "real world" problem (multiplication of matrices) can be solved using the Linda approach to problem solving. However, cmatrix is somewhat limited in that it can only multiply two N x N integer matrices. As in rayshade, the main Linda program functions as a supervisory process, and the EVAL-ed workers perform the computation. The supervisor OUTs the rows of the first input matrix (A) and the columns of the second input matrix (B) to Tuple Space, and then INs the rows of the resulting product matrix (C). Each worker repeatedly retrieves a row of A (with INP) and all the columns of B (with RD) from Tuple Space, computes the corresponding row of C, and places that result row in Tuple Space. When all result rows have been computed, the workers exit.
While perhaps not optimal from the standpoint of efficiency, the particular design of \texttt{cmatrix} results in heavy tuple traffic between worker processes and Tuple Space. A worker retrieves copies of all columns of \( B \) \textit{each time} it computes a new row of \( C \). Ideally, a worker should retrieve a copy of \( B \) only once, and reuse this copy for each result row computation. This design aspect causes \texttt{cmatrix} to be very I/O intensive, and therefore useful for measuring the performance of the distributed Linda system under conditions of heavy I/O. Each time a worker computes a result row, it retrieves 1 row tuple and \( N \) column tuples from Tuple Space, and sends 1 result row tuple back to Tuple Space. Hence, \( N \) rows of matrix \( A \) and \( N^2 \) columns of matrix \( B \) are retrieved, and \( N \) rows of matrix \( C \) are generated by the workers over the life of the program. The only computations to be done by workers are the \( N \) integer multiplication operations and \( N \) integer addition operations for each position of each result row calculated. A total of \( N^3 \) multiplications and \( N^3 \) additions are completed by the worker processes over the life of the program.

Figure 7.4 graphs the execution times obtained from various runs of \texttt{cmatrix}. Each point on the graph represents an \( N \times N \) matrix multiplication, where \( N=50 \). Runs using 10, 15, 20, and 25 workers were timed on Linda-LAN configurations consisting of 1 to 5 processors. The results from \texttt{cmatrix} are somewhat similar to those from \texttt{rayshade}. From this figure, we observe the following:

- As the number of processors increases from 1 to 5, execution time decreases. In fact, \texttt{cmatrix} uses the additional processors almost as efficiently as \texttt{rayshade}.

- Adding Linda processes also results in a decrease in execution time, to an extent. A threshold exists beyond which adding more processes is not beneficial (as seen in the 1, 2, and 3 processor cases). This trend conforms to our expectations. Since \texttt{cmatrix} is I/O intensive, we expect that adding more Linda processes to each processor will
Figure 7.4. Execution times from cmatrix application
exploit concurrency by providing enough work to keep the CPU busy, even when most
of the Linda processes are waiting for tuples from TS-Man.

Figure 7.5 presents another view of the results from cmatrix. This graph emphasizes the
fact that an increase in processors has the most pronounced effect on execution times. Dis-
tributing the work to multiple Linda processes becomes beneficial when those processes
are executing on several processors.

7.2 Enhancing the System’s Performance

A substantial portion of the overall wall clock times reported in Section 7.1 can be attrib-
uted to Linda processes’ waiting for tuples. In particular, the I/O intensive cmatrix pro-
gram spends a large amount of time waiting for response tuples to arrive from TS-Man. As
an example, consider the five 10-worker cases from the test runs of cmatrix. A given
Linda process spends an amount of time roughly equal to 90% of the run’s total execution
time waiting for responses to requests for tuples. On the other hand, we have data which
indicates that the total amount of time required for TS-Man to respond to all of a given
Linda process’s requests is on the order of 1 second. Therefore the relatively large wait
times must be accumulated by requests en route from the Linda process to TS-Man, or
from TS-Man to the Linda process.

Recall from Section 5.2 that each request (we use the generic term request to refer to a
tuple or a request for a tuple) sent by a Linda process (or by any other component of the
data subsystem) is comprised of a fixed length header packet and a variable length data
packet. Figure 7.6 illustrates the “round-trip” path taken by these two-packet requests.
(The arrows and the arrangement of the boxes in the figure indicate the sequence of events
in time. Although this diagram shows separate arrows for header and data packets, these
Figure 7.5. Another view of the times from cmatrix
Figure 7.6. Round-trip path taken by requests for tuples
packets are actually sent over the same socket connection, as described in Section 5.2.) When a Linda process issues a request for a tuple, the request must travel from the Linda process to its L-Kernel, and from L-Kernel to the proper TS-Man process. Once TS-Man has matched the request with a tuple, that tuple travels from TS-Man back to the L-Kernel, and from L-Kernel back to the originating Linda process.

Since Linda processes spend so much time waiting for tuples, it is reasonable to investigate ways to increase the speed at which requests traverse the round-trip depicted in Figure 7.6.

TCP Buffering

As stated in Chapter 4, the distributed system has been designed to use the D-Kernel system's stream socket interface between Linda processes and the Tuple Space Manager. However, a number of unexplored options are available which change the default behavior of the system's stream socket interface and its TCP/IP software. We recognize that the default behavior may not be optimal for the distributed Linda system.

In particular, TCP employs a buffering scheme designed to increase network efficiency. When a process writes data to a stream socket, TCP may delay actually sending that data until a full buffer of data can be sent. This buffering scheme reduces the total number of packets required to send a collection of data. Since the overhead associated with sending a single packet can be substantial, the scheme results in more efficient use the network’s bandwidth.

We speculate that TCP’s buffering scheme may not be appropriate for the distributed Linda system. Since the nodes of the Linda-LAN communicate over a fast Ethernet network, we suspect that the increased network efficiency is not needed. In fact, the TCP’s
buffering behavior may cause delay in the delivery of requests' header and data packets. We observe that Linda processes spend a substantial amount of time waiting for responses to tuple requests. After sending a request for a tuple to Tuple Space, a Linda process immediately attempts to read the response. Since the response is not yet available, the Linda process blocks. It does not continue with other processing, but must wait for the arrival of each tuple it requests. Therefore, we may be able to enhance overall performance of the system by expediting delivery of these requests.

A socket option is available which defeats the default buffering action of TCP. This TCP level option is called TCP_NODELAY, and is put into effect with the setsockopt() system call.

To determine the impact of TCP's buffering algorithm, the components of the data subsystem (including the Linda processes, the L-Kernels, and the various TS-Man processes) were modified to use the TCP_NODELAY option on their data sockets, and the rayshade and cmatrix applications were rerun.

*Results from rayshade on the Modified System*

Figure 7.7 shows the execution times of rayshade on the modified system. If we compare this graph with Figure 7.2, we observe that execution times have decreased, but not dramatically. Specifically, execution times have decreased by an average of 7% (in the 1 processor case) to 28% (in the 5 processor case). Since rayshade's primary work is computation, we did not expect its execution time to decrease substantially by implementing the TCP_NODELAY option. We also observe that the performance penalty incurred by adding additional worker processes is smaller than in the unmodified system.
Figure 7.7. Execution times of rayshade using TCP_NODELAY
Results from cmatrix on the Modified System

Figure 7.8 shows the execution times of cmatrix on the modified system. Execution times have decreased an order of magnitude in comparison with the original times shown in Figure 7.4. Note that the scale of Figure 7.8’s vertical axis ranges from 26 to 40 seconds, while that of Figure 7.4 ranges from 50 to 350 seconds.

Figure 7.8 indicates that adding a second processor results in a decrease in execution time. However, adding more processors has almost no impact on execution times. The system is not able to use more than two processors efficiently for this particular application and problem size. We speculate that this is because the data subsystem’s components are not fully utilizing available processing power. Therefore, adding more processing power is not beneficial. The level of computation and I/O is high enough to benefit from a second processor, but more processors cannot be utilized effectively.

Additionally, cmatrix no longer exhibits speedup as a result of adding more worker processes to a given number of processors. We speculate that the level of computation is so insubstantial that adding more worker processes results in an increase in context switching activity without a corresponding increase in CPU utilization. As a result of expediting request delivery (by using the TCP_NODELAY option), worker processes no longer wait a substantial amount of time for tuples. As a side effect of TCP’s buffering scheme in the unmodified system, using many worker processes results in a larger quantity of outstanding requests per-L-Kernel in the system. Hence, TCP’s buffer fills up faster, and requests are delivered to their destinations more quickly. Since this buffering is turned off by the TCP_NODELAY option, this side effect is not observed in the modified system.
Figure 7.8. Execution times of cmatrix using TCP_NODELAY
Conclusions

The interaction between the tuple communication protocol of the data subsystem (that is, the two-packet request scheme) and the default buffering behavior of TCP tends to limit computational performance, especially under conditions of heavy I/O. However, overall performance of the distributed system can be enhanced by utilizing the TCP_NODELAY socket option, which turns off TCP's buffering.

7.3 A Discussion of Buffering and Possible Design Alternatives

Section 7.2 presents data which demonstrates that TCP's buffering scheme results in delayed delivery of requests and tuples in the distributed Linda system. We now examine the buffering algorithm more closely to determine whether it can be exploited by a modification to the design of the Linda system's tuple communications protocol.

7.3.1 TCP's Output Buffering Algorithm

As stated, the buffering scheme used by the 4.3BSD implementation of Unix [LEFS89] is designed to minimize the number of "small packets" (that is, packets of less than maximum size) sent over the network. The algorithm allows a process to write a small amount of data to a socket, and have that data sent over the network immediately in a small packet. However, subsequent small amounts of data written by the process are queued in a buffer until either of the following occurs:

- An acknowledgment is received that the outstanding small packet has been received by its destination process. In this case, any data which has been queued in the buffer is sent over the network in a new small packet.
• The process writes enough new data to the socket to result in a “maximum sized packet” being filled. Whenever the buffer has enough data to fill a maximum sized packet, that packet is sent over the network, regardless of the status of the outstanding small packet.

**Impact of Buffering on the Linda System**

If the round-trip diagram in Figure 7.6 is reexamined in the context of the buffering algorithm, it becomes clear that this buffering scheme could have a significant impact on delivery times of Linda requests. Header packets are 40 bytes in size, and the size of a data packet depends on the size of the request or tuple which it contains. In the runs of `cma-trix` presented in this chapter (in which the matrices are 50 x 50), the overwhelming majority of data packets are 268 bytes. TCP uses a maximum packet size of 1024 bytes for stream sockets. Therefore, several complete requests (header and data packets) are needed in order to fill a maximum sized TCP packet.

The implications of TCP’s buffering scheme are numerous:

1. When a Linda process writes a request to the socket connected to its L-Kernel, the header packet is sent to L-Kernel immediately. But the corresponding data packet is not sent until acknowledgment of the header packet’s arrival is received. L-Kernel cannot receive the header packet and send an acknowledgment until it is scheduled to use the CPU. Therefore, the Linda process will not send the data packet until (1) L-Kernel has run, and (2) the Linda process is again scheduled to run.

2. When L-Kernel is scheduled to use the CPU, it receives and acknowledges the header packet, and then forwards it to TS-Man. It then attempts to read and
forward the data packet. But the Linda process has not yet transmitted the data packet, so L-Kernel blocks. This implies that L-Kernel cannot service requests from other Linda processes which may be available.

(3) When TS-Man receives a header packet on an L-Kernel connection, it assumes the corresponding data packet is forthcoming, and attempts to read it. Since delivery of the data packet from the Linda process to the L-Kernel is delayed, that data packet will not yet be available to TS-Man. Therefore, TS-Man blocks on its attempt to read the data packet. The implication is that TS-Man cannot read requests which may be available from other L-Kernels until the awaited data packet arrives.

(4) Similar blocking behavior will occur when TS-Man sends responses back to L-Kernel and the Linda process. That is, L-Kernel will block when it attempts to read the data packet from TS-Man because TS-Man has not yet sent the data packet. The Linda process will block when it attempts to read the data packet from L-Kernel because L-Kernel has not yet received the data packet from TS-Man nor sent it to the Linda process.

(5) As the total number of processors increases, the number of connections between TS-Man and L-Kernels increases, decreasing the negative impact of the buffering algorithm accordingly. For example, TS-Man can send a “small packet” to each L-Kernel at once. When L-Kernel blocks, it will be rescheduled to run again sooner, because there are fewer Linda processes competing for the CPU. Overall execution time decreases as a result.

(6) As the total number of worker processes on a given processor increases, the total number of outstanding (not yet completed) requests increases proportionally. When there are more outstanding requests it becomes more likely that TS-Man
will be able to send a larger amount of data (header and data packets from subsequent requests) after the initial header packet to an L-Kernel is acknowledged. Also, as the number of outstanding requests increases, TS-Man is more likely to fill up maximum sized packets with the header and data packets from those requests. These packets can be sent immediately, without waiting for acknowledgment of an outstanding small packet. Again, the result is that overall execution time decreases.

7.3.2 A Discussion of Three Design Alternatives

Based on the implications described above, this section presents three potential modifications to the Linda system’s mechanism for sending requests and tuples, and discusses the potential effects of each. The intent is to determine whether the Linda system’s request sending/receiving protocol could be modified to exploit the default TCP behavior. TCP’s buffering scheme is implemented for the specific purpose of minimizing the number of small packets transmitted over the network. Therefore, we reason that defeating this scheme (with the TCP_NODELAY socket option) may not be the most effective method for enhancing the performance exhibited by the unmodified system.

Padding Small Packets to Maximum Size

One cause of the original system’s relatively poor performance (in comparison with that of the modified system which uses the TCP_NODELAY option) is the fact that header and data packets are not optimally sized for immediate transmission by the TCP code. We conjecture that performance of the system can be enhanced by padding all request packets to the size of a maximum sized TCP packet.
To demonstrate that the size of request packets is crucial in the unmodified Linda system, test "client" and "server" programs were constructed which send packets to each other through a stream socket connection. The default TCP buffering scheme was used. The client process sends two packets over the network to the server process. The server reads the two packets and then writes both of them back to the client. Finally, the client reads the two packets from the server. This process was repeated for 100 iterations.

When the size of the packets was set to 1024 bytes (which is the size of a "maximum sized" TCP packet), 100 iterations took approximately 0.8 seconds. However, when the packet size was reduced to 1023 bytes, 100 iterations took 18.7 seconds. This vast difference in execution time is due to the fact that the 1023 byte packets are considered "small" by the TCP algorithm. Hence, the client's TCP code sends the first packet, but then waits for an acknowledgment from the server before actually transmitting the second packet. The server does not send the first packet back to the client until it has read this second packet. Also, the server does not send the second packet back to the client until it receives an acknowledgment from the client indicating receipt of the first packet. On the other hand, when the packets are 1024 bytes, each is sent immediately to the destination process, and neither process waits for acknowledgment before sending the second packet.

The behavior of this test client and server mimics that of the components of the Linda system, and emphasizes the critical nature of packet size in the Linda system. We speculate that if all Linda system packets were always the size of maximum sized TCP packets, its behavior would be similar to that of the modified system which uses the TCP_NODELAY option. However, since all packets would be much larger than necessary, packet padding would require even more network bandwidth than the TCP_NODELAY system. If other active processes also required significant network bandwidth, contention for the (potentially saturated) network could lead to even worse performance.
What if only header packets were padded to the maximum TCP packet size? Such a modification would cause a request’s header packet to be sent immediately as a full-sized TCP packet. The request’s data packet would then be sent immediately as a “small packet”. A subsequent request’s header packet sent on the same socket would also be sent immediately (since it is padded to maximum TCP packet size). However this new request’s data packet would be queued until an acknowledgment of the first request’s data packet is received. If a third request is written by the process while the second request’s data packet is still queued, then the queued data packet and part of the third request’s header packet would be sent, since the new header packet would provide more than enough data to fill a TCP packet. The implication is that the third request’s header packet would become fragmented by the sender’s TCP software. It is unclear whether the overall behavior of such a modification would compare favorably to the results obtained by implementing the TCP_NODELAY option.

In general, packet padding does not seem like an effective approach, since it results in large quantities of non-data (that is, padding) being sent over the network. It would appear that using the TCP_NODELAY socket option is a more reasonable response to the delays caused by TCP’s default buffering scheme.

Sending Header and Data as a Single Packet

One of the reasons using the TCP_NODELAY option has such a substantial impact on overall execution times is that the Linda system sends requests as two distinct packets. What if the header and data packets of each request were written as a single packet?

- The Linda process would send each request in its entirety to the L-Kernel. Therefore, L-Kernel would not block on attempts to read a request’s data packet.
• Similarly, TS-Man would send each response to L-Kernel as a single packet, rather
than as separate header and data packets. As a result, L-Kernel would not block on
attempts to read data packets from TS-Man.

• When a Linda process sends a tuple to Tuple Space with the OUT operation, it writes
the request to the socket and continues executing. It does not block because it does not
need to wait for a response. If the process generates a large quantity of OUTs all at
once, TCP’s buffering scheme would result in more efficient use of available network
bandwidth. Depending on their size, multiple tuples could be sent to Tuple Space in
single TCP packets.

• If TS-Man responds to multiple requests from Linda processes on the same machine,
the buffering algorithm will again increase network efficiency, but without the side
effect of causing L-Kernel to block on attempts to read data packets.

We speculate that modifying the distributed system to send requests as single packets
could substantially increase the overall performance of the system. This modification pre-
vents components of the data subsystem from blocking on attempts to read data packets
which have not yet been sent. We suspect that this blocking behavior is the primary factor
causing inflated execution times in the unmodified system. In addition, by sending single-
packet requests (rather than using the TCP_NODELAY option) the TCP buffering scheme
can still be used to send multiple requests in single packets.

It may be possible and advantageous to use a combination of the single-packet request
modification and the packet padding modification to create an optimal request protocol.
The Linda code which writes packets to the socket would need to somehow determine
when padding would be beneficial and implement it in only those cases. Further research
is needed to determine what types of applications would exhibit better performance using single-packet requests and packet padding (rather than the TCP_NODELAY option).

**Handling Data Packets Independently**

We suspect that the primary cause of poor performance in the original system is the blocking behavior of L-Kernel and TS-Man when data packets are not available. What if these components could receive (and send) header and data packets as separate entities? That is, what if each packet is sent and received separately, rather than as two parts of a single request?

- When L-Kernel determines (by invoking the `select()` call) that a header packet is available, it would read and forward that header, but would not assume that the corresponding data packet was forthcoming. L-Kernel would have to call `select()` again to determine that the data packet was available. Therefore, L-Kernel *would not block* because of the delayed arrival of a data packet. Rather, it could continue to service other available requests.

- Similarly, TS-Man could read header packets from more than one connection, but defer servicing a request until a request's data packet arrived. As a result, TS-Man would not block on attempts to read data packets, but would continue to service other available requests.

This scheme would require several other modifications:

- When a packet arrives on a connection, the receiving process must determine whether it is a header or data packet (so that it can determine how many bytes to read). This could be accomplished by tagging each packet with a packet-type variable, and then
peeking at the socket stream to read the packet type.

- When a data packet arrives, the receiving process must identify the header packet with which the data packet is associated, so that it can find out how many data bytes to read from the socket. Essentially, the data packet would have to include its own "header" information to help the receiving process associate the header and data packets.

The implication is that separating header and data packets into "separate requests" doesn’t solve the problem. The method which would have to be used to read a given data packet involves peeking into the socket stream. If processes can peek at the data waiting to be read, then there is really no reason to use header packets. The only reason for separating requests into header and data packets is that data packets (which contain tuples) vary in size, and the size of a given data packet can easily be transferred to the receiving process by transmitting it in a fixed length header packet. Therefore, we do not believe that modifying the system to treat header and data packets independently is an advantageous endeavor.

Final Consideration

We expect that if the nodes of the Linda system were connected by a slower Wide Area Network rather than by a fast Ethernet, defeating TCP’s buffering scheme could have a significant negative impact on execution times of Linda applications. Therefore, research which identifies a better solution is needed to further the development of the distributed Linda system.
8.0 Summary and Future Work

The research presented in this thesis focuses on the design and preliminary investigation of a system for distributed Linda process execution. This chapter summarizes the distinctive aspects and characteristics of the system's design and reiterates the important findings of the investigation. Finally, several areas for possible future work are discussed.

8.1 Summary of Contributions

The primary contributions of the research presented in this thesis are as follows:

- the design and implementation of a remote Linda process instantiation mechanism which is consistent with the Linda paradigm and the semantics of Linda's EVAL operation, and which provides for true parallel execution of Linda processes on a network of computers,

- the division of Tuple Space and the Tuple Space Manager program into separate, distributable processes which exploit concurrency by providing for parallel access to the various logical partitions of Tuple Space, and

- the investigation and analysis of the system's execution profile which has revealed important characteristics of the distributed Linda system which may tend to limit computation performance under certain conditions.

Each of these contributions is revisited below.
8.1.1 The Distributed EVAL Mechanism

The distributed Linda system presented in this thesis provides an environment for concurrent process execution on a network of computers. The design of this environment has been based on two previous implementations of Linda. In particular, the distributed Linda system avails itself of the interprocess communication facilities present in the previous systems to communicate active tuple transformation specifications to new remote Linda processes.

More specifically, the distributed system employs Tuple Space and Linda's OUT and IN operations in the implementation of a distributed version of the EVAL operation. When a Linda process encounters a call to the EVAL operation, it places the specified active tuple and the corresponding evaluation parameters in Tuple Space using the OUT operation. The Linda system then effects the instantiation of a new process, possibly on a different machine from the process which originally invoked the EVAL. This new process retrieves the active tuple and evaluation parameters with the IN operation, and transforms the active tuple into a passive tuple, which it then places back in Tuple Space.

The distinguishing characteristic of our solution is that the EVAL is naturally redefined in terms of Linda's existing OUT and IN operations, rather than implemented by a whole new set of protocols. Two particular problems which arise in considering the design for a remote version of the EVAL operation are that an EVAL-ing process must have some way of communicating portions of both its code space and its data space to the EVAL-ed process. By using Linda's OUT and IN operations, the EVAL-ing process can share data with the EVAL-ed process by way of Tuple Space.
This design provides flexibility for future enhancements to the Linda system, including evaluation of active tuples based on tuple characteristics (execution requirements) and available resources.

8.1.2 Split Tuple Space

Tuple Space and its Tuple Space Manager (TS-Man) process have been "split" into separate processes. Each new TS-Man process manages a separate "partition" of Tuple Space, where partitions are based on tuple content and usage patterns. Since the various TS-Man processes can execute independently, each can be assigned to run on a separate machine of the Linda-LAN.

Because TS-Man serializes all requests for access to Tuple Space, Tuple Space is a potential bottleneck of the distributed Linda system. By separating TS-Man into separate processes, each managing a separate area of Tuple Space, we provide the capability for parallel access to the different partitions of Tuple Space, thereby reducing contention for TS-Man's services.

8.1.3 Investigation and Analysis of Results

The investigation which is presented in Chapter 7 has resulted in the identification of some important characteristics of the distributed system.

- Distribution of Linda processes over multiple Linda-LAN processors can enhance computational performance dramatically. Specifically, we have demonstrated that certain applications (e.g., the rayshade ray-tracing program) are able to make effective and efficient use of a multiple processor Linda-LAN.
• Distribution of computational (and I/O) work to multiple Linda processes does not necessarily increase overall performance of the distributed Linda system. In particular, we have shown that applications such as rayshade which are computationally intensive do not make effective use of multiple processes on each processor.

• A distributed system composed of many processors will not necessarily exhibit significantly better performance than a system of fewer processors. We have shown this to be true when the I/O intensive cmatrix application is run with more than 2 processors.

• The default buffering scheme of TCP (as implemented in the 4.3BSD Unix operating system) which is designed to increase network efficiency is not necessarily appropriate for use in the distributed Linda system. We have demonstrated that our system's two-packet request protocol does not interact favorably with TCP's buffering. Therefore, either the buffering should be turned off, or the request protocol should be redesigned.

8.2 Future Work

The work presented in this thesis sets the stage for a wide variety of areas for further research. Several of these topics relate to potential enhancements to the design of the distributed Linda system. Other topics suggest that further study of the system's performance would be beneficial.

Future work on the distributed system's design

• Further investigation is needed into the impact of L-Kernel's scheduling policy. Currently, L-Kernel always services its TS-Man socket(s) first, then services its Linda process sockets. The L-Process sockets are serviced in reverse order of connection.
That is, the last Linda process to connect to L-Kernel is always the first to be serviced. This policy could result in underutilized and/or underserved Linda processes.

- It may be beneficial to eliminate L-Kernel from the system. That is, if Linda processes were to communicate directly with TS-Man, then we could do away with all the complexity associated with L-Kernel's behavior. On the other hand, if every Linda process had a stream socket connection to TS-Man, then TS-Man could easily end up with too many stream sockets. An alternative communications mechanism (such as datagram sockets) would probably need to be investigated. TS-Man's socket servicing policy would also need to be examined to ensure that all connections are given adequate attention. Selection of a communications mechanism other than reliable stream sockets could imply that the communications interfaces between components would need packet-sequencing, error-detection, and error recovery software.

- Further investigation is needed into the design alternatives presented in Section 7.3. Specifically, an empirical study of the impact of padding request packets to maximum TCP size, and of coalescing header and data into single packets could yield results which might lead to a better design of the Linda system's tuple request protocol.

Future Investigation of the Distributed System's Behavior

- The impact of distributing Tuple Space needs to be examined. In particular, no work has been done to quantify the possible benefits of the Split Tuple Space version over the Single Tuple Space system.

- The impact of Linda-LAN configuration should be investigated. Although we suspect that the various Linda-LAN components will exhibit better performance when each is provided with its own processor, we need to investigate optimal configurations when
the number of Linda-LAN processors is limited.

- A wider selection of Linda applications should be executed on the distributed Linda system, to provide a broad picture of the system's capabilities. In particular, research is needed to determine what types of Linda applications benefit most from being executed in the distributed environment, especially when using the Split Tuple Space version of the system.
References and Bibliography


References and Bibliography 112


Appendix A.  Start-up and Shutdown of the Linda-LAN Environment

A.1 System Start-up

The distributed Linda-LAN is started in two distinct stages. The control subsystem is started once, and can remain running indefinitely. The data subsystem is started when a user wants to execute a Linda program, and is shut down after the program completes. In addition, the executable file that represents a user's Linda program must be distributed to the Linda-LAN before program execution.

Control subsystem start-up

Figure A.1 depicts the sequence of events involved in the start-up of the control subsystem. The Linda-LAN network system administrator starts the L-Man process manually. L-Man creates a server socket for accepting connections from L-Coms, and then spawns (through the Unix rsh facility) an L-Com process on each machine whose name is found in a list of participants. The argument list for the L-Com process includes the name of the machine running L-Man and the socket port at which L-Man is to be contacted. L-Com uses this information to establish a socket connection to L-Man. Once L-Man has accepted connections from all of the L-Coms it has instantiated, the control subsystem of the Linda-LAN environment is considered operational.

Distributing the user's executable Linda program to the Linda-LAN

Although not strictly part of system start-up, the executable file that represents the user's Linda program must be distributed to all the machines on the Linda-LAN before the program can be executed on the distributed system (see Figure A.2) A special utility program called ldist has been written to help accomplish this task. The ldist program (which
Linda-LAN system administrator starts L-Man

L-Man creates L-Com server socket

L-Man spawns L-Coms on participating machines via. rsh

L-Coms establish socket connections to L-Man

Figure A.1. Startup of the Control Subsystem
Figure A.2. Distribution of the Linda program to the Linda-LAN
accepts the name of the executable file as its sole parameter) copies the executable file to a location accessible by the Linda system, establishes a socket link to its L-Com, sends a distribute executable command and the name of the executable file to L-Com, and then waits for an indication of completion from L-Com. L-Com forwards the distribute executable command and the Linda program’s name to L-Man. L-Man notifies all L-Coms to obtain a copy of this file (using the Unix rcp facility) from the machine which originated the command. Each L-Com notifies L-Man when it has obtained a copy of the executable file. After L-Man receives such notification from all L-Coms, it returns an indication of success to the originating L-Com, which immediately forwards this notification to the ldist program. On receiving the completion indication, ldist indicates to the user that his Linda program has been successfully distributed to the Linda-LAN, and then exits.

Data subsystem start-up

Figure A.3 shows the steps involved in starting the data subsystem of the Split Tuple Space version of the Distributed Linda System (described in Chapter 5). The data subsystem is not started until a user indicates, by running a utility program called lexec, that he wishes to execute a Linda program on the Linda-LAN. The lexec program accepts the name of the Linda program as an argument, and sends an execute request command and the Linda program’s name to the L-Com on the user’s workstation. L-Com forwards this information to L-Man. On receiving the execute request command, L-Man decides whether to grant the request (the current implementation allows only a single Linda program to run at a time). If the request is denied, L-Man notifies the originating L-Com, which notifies the lexec program.

However, if the execute request is granted, L-Man first spawns (by calling rsh) the Eval TS-Man process. The Eval TS-Man first creates a server socket for accepting connections
Sequence of events:

(1) lexec sends execute request to L-Com
(2) L-Com forwards request to L-Man
(3) L-Man spawns Eval TS-Man
d of other TS-Men
(4) Eval TS-Man creates L-Kernel server socket
(5) Eval TS-Man creates TS-Man server socket
(6) Eval TS-Man spawns the other 3 TS-Men
(7) TS-Men create L-Kernel server sockets
(8) TS-Men connect to Eval TS-Man
(9) Eval TS-Man connects to L-Man
(10) L-Man sends startup command to L-Coms
(11) L-Coms spawn L-Kernels
(12) L-Kernels connect to Eval TS-Man and obtain addresses
d of other TS-Men
(14) L-Kernels connect to other TS-Men
(15) L-Kernels create server socket(s)
(16) L-Kernels connect to L-Coms
(17) L-Coms notify L-Man that L-Kernels have started
(18) L-Man notifies originating L-Com that system is up
(19) L-Com notifies lexec that system is up

Figure A.3. Startup of the Data Subsystem
from the L-Kernels, and then creates a server socket for accepting connections from the
other TS-Man processes. The Eval TS-Man then spawns (using rsh) the other three
TS-Man processes (to manage the Counting Semaphore, Hash, and Queue partitions of
Tuple Space). Each of these TS-Men creates a socket for accepting connections from the
L-Kernels, and then connecta to the Eval TS-Man. Once the Eval TS-Man has accepted
connection requests from all three of the other TS-Men, it connects to L-Man and informs
L-Man of the socket address at which it will accept L-Kernel connections.

On receiving this connection information from the Eval TS-Man, L-Man sends a com-
mand to all the L-Coms, instructing each to spawn an L-Kernel on its machine. L-Man
also forwards the hostname and address of the Eval TS-Man’s server socket to the
L-Coms. Each L-Com passes this information to its L-Kernel as a parameter.

L-Kernel establishes several socket connections. First, it connects to the Eval TS-Man. As
part of the initialization between L-Kernel and the Eval TS-Man, L-Kernel is informed of
the hostnames and socket addresses of the other three TS-Man processes. L-Kernel uses
this information to connect to the other TS-Men. Next, L-Kernel creates a server socket on
which it will accept connections from Linda processes on its machine. Finally, it connects
to L-Com, so that it can receive commands (such as program exit, program abort, create
new Linda process) sent from L-Man through the L-Coms.

Once this initialization activity is completed, L-Kernel notifies L-Com that it is ready for
Linda program execution. L-Com forwards this notification to L-Man, and L-Man notifies
the originating L-Com. Finally, this L-Com returns an indication of success to the lexec
program. The system is now ready for Linda program execution.
A.2 Shutdown of the Linda-LAN environment

Data subsystem shutdown

Just before a Linda program exits, it sends a program exit request to its L-Kernel. L-Kernel forwards the request to L-Com, and L-Com forwards it to L-Man. L-Man then sends a shutdown program command to all its L-Coms and to the Eval TS-Man.

The L-Coms forward the shutdown command to their respective L-Kernels. When L-Kernel receives the shutdown command, it does the following:

1. kills any Linda processes which may still be connected to L-Kernel,
2. sends an EXIT request to each of the four TS-Man processes,
3. closes all its sockets, and then
4. exits.

When the Eval TS-Man receives the shutdown command from L-Man, it does the following:

1. closes its socket connection to L-Man,
2. forwards the shutdown command to the other three TS-Men,
3. closes its socket connections to the other TS-Men,
4. continues processing requests from connected L-Kernels until an EXIT request has been received from each L-Kernel (closing L-Kernel sockets as EXIT requests are received),
5. closes its socket for accepting connections from L-Kernels, and
6. exits.
When each of the three (non-Eval) TS-Men receives the shutdown command from the Eval TS-Man, it performs the following actions:

1. closes its socket connection to the Eval TS-Man,
2. continues processing requests from connected L-Kernels until an EXIT request has been received from each L-Kernel (closing L-Kernel sockets as EXIT requests are received),
3. closes its socket for accepting connections from L-Kernels, and
4. exits.

Once all the L-Kernels and TS-Man processes have exited, only the control subsystem remains active.

*Control subsystem shutdown*

The control subsystem is shut down manually by the Linda-LAN administrator by executing the utility program *kill*, which sends a *shutdown Linda-LAN* message to L-Man. L-Man forwards the shutdown command to its L-Coms, causing them to exit, and then L-Man itself exits.
Vita

Patrick G. Robinson was born on August 2, 1962, in Birmingham, Alabama, where he lived until the age of 12. He graduated from Jackson Preparatory School in Jackson, Mississippi in 1980. He received a Bachelor of Science degree in Mathematics from Wheaton College, in Wheaton, Illinois, in the Spring of 1984.

Patrick began working for Virginia Tech's Department of Anaerobic Microbiology in 1985, doing applications programming in the areas of bacterial identification through pattern recognition. In the Fall of 1986, he entered the Master's program in Computer Science at Virginia Tech, and began taking courses on a part-time basis. He began working for Extension Information Systems at Virginia Tech in 1988, while continuing work toward his degree. His Master's thesis, "Distributed Linda: Design, Development, and Characterization of the Data Subsystem" was completed in May of 1994.

Patrick is a member of Upsilon Pi Epsilon. His research interests include distributed processing and network performance issues.

[Signature]

Patrick G. Robinson