A Distributed Linda Server on a Network of Heterogeneous Processors

THESIS

Submitted in Fulfilment of the Requirements

for the Degree of

MASTER OF SCIENCE

Rhodes University

by

Graham Leslie Smith

January 1993

Acknowledgements

I would like to thank my supervisor Dr. Peter Wentworth for his ideas and guidance throughout this work. My consultations with him always left me with a clearer view of a particular situation. I would also like to thank Dave Sewry for his help with CCS. The friendly open atmosphere here at Rhodes have made the past two years pleasant ones.

Abstract

Linda is an approach to parallelism which relies on a virtual associative shared memory called tuple space. Tuple space is accessed through a small set of primitive operations and is conceptually easy to understand and manipulate. The physical implementation of a Linda tuple space may of course be completely different from the conceptual model. Rhodes has implemented versions of Linda on a ring of RS-232 joined PC's and on a cluster of T800 transputers with a single copy of tuple space on one transputer. Current research targets the implementation of a distributed Linda server on a network of heterogeneous processors. This work describes the design and implementation of a distributed Linda server. Emphasis is placed on aspects of the design which enhance portability and efficiency.

Contents

List of Figures

Chapter 1

Introduction

This work concerns the design and implementation of a distributed Linda server. It is distributed in that each node of the server is powerful enough to support Linda communication between its clients and other clients. We begin by introducing the Linda paradigm and its important features. We show the need for a distributed server and lay down the goals and requirements which guide the work.

1.1 Background

Linda is a set of language primitives that is added to an existing language to allow communication and synchronization between processes and the creation of new processes. The Linda primitives operate on a shared bag (or multiset) of data items called tuples. The bag is called **tuple space.** The implementation of the bag is the subject of this thesis. A **tuple** is an ordered collection of data items (a **field**). Each field in the tuple has a type associated with it. The choice of types is often drawn from the host language. A field can be either **formal** or **actual.** An actual field contains a data value or a function which returns a data value of a suitable type, a formal contains no data value, it points to a place where data value can be placed. It is useful to consider tuples as being one of two types, either a **tuple** or an **anti-tuple** although they are indistinguishable. Tuples represent some piece of shared information and reside in tuple space. Anti-tuples are used to extract the required tuple from tuple space. A tuple is obtained from tuple space by presenting a tuple that **matches** the one we want. A pair of tuples match if they agree in arity and each field

in the first tuple matches the corresponding field in the second tuple. Two fields match if they are of the same type and either one field is a formal and the other is an actual or both are actuals and they contain the same value.

The Linda Primitives

- **OUT**($field_1, ..., field_n$): inserts the tuple $< field_1, ..., field_n >$ into tuple space. The call returns immediately and the user can expect the tuple to become visible to other users of tuple-space in the near future.
- **IN**($field_1, ..., field_n$): blocks until a tuple matching tuple $< field_1, ..., field_n >$ is available. When a tuple becomes available it is removed from tuple-space and the **IN** returns. The place pointers in the formal fields are filled with the actual values of the returned tuple. A tuple in tuple space can satisfy only one waiting **IN** request.
- **INP**($field_1, ..., field_n$): is a predicate version of **IN**. It returns immediately with failure if a matching tuple is not available.
- **READ**($field_1, ..., field_n$): blocks until a tuple matching tuple $< field_1, ..., field_n >$ is available. When a tuple becomes available the **READ** returns. It is not removed from tuple-space. The place pointers in the formal fields are filled with the actual values of the returned tuple. There is no general agreement on the result if a number of **READ's** and **IN's** are waiting on a tuple which becomes available. One should assume that some subset of the **READ's** will be satisfied and then one of the **IN's**.
- **READP**($field_1, ..., field_n$): is a predicate version of **READ** but returns immediately with failure if a matching tuple is not available.
- **EVAL**($field_1, ..., field_n$): behaves like an **OUT** except that actual fields containing functions are not evaluated before the tuple is placed in tuple-space. A task is created on suitable processor to evaluate each unevaluated field and when each has completed the tuple becomes visible, with the functions replaced by their results.

The following small example is a framework typical of Linda programs. It is a 'Master-Worker' model, where one program is responsible for initiating the computation and gathering and co-ordinating the results of the workers:

```
master(){
        int i;
        for(i=0;i<tasks;i++) {
                OUT("task_data",i,data[i]);
                EVAL("results" ,i, sub_problem(i) );
        }
        for(i=0;i<tasks;i++) IN( "results", i, ?result[i] );
        assemble_solution(result);
}

sub_problem(i){
       IN("task_data" i, ?data);
       result =compute(data);
       return( result);
}
```

Advantages and Criticisms of Linda

Researchers have identified or collected advantages for, and criticisms against using Linda as a parallel processing paradigm [5] [18] [21] [22] [14] [12]. Some of the advantages claimed are:

- **Portability** Linda has been successfully ported to a wide range of both distributed and shared memory systems, and Linda programs can be run with little modification (implementation specific quirks) on any of these.
- **Topology independence** Linda hides the way processors are connected from the user.
- **Time uncoupling** Linda allows communication between processes disjoint in time.
- **Dynamic load balancing** The system appropriately distributes tasks as they are created.
- **Scalability** Linda allows processors to be added to the computation with no modification to the client code.

- **Algorithm design** Because of its simplicity and power, Linda is claimed to be a good tool for algorithm design.

Some criticisms levelled against Linda are:

- **No high level structures** Structures such as calling routines and queue-management filters which are often provided in other languages must be implemented by the user.
- **Efficiency** The eval mechanism is difficult to implement efficiently.
- **Linda on distributed systems** Some claim that it is difficult to implement Linda on distributed systems.
- **Tuple retrieval** Some claim that tuple retrieval is a potential bottleneck.
- **Network clogging** Linda can place massive communication overheads on network systems.
- **Unauthorised access** Current systems do not prevent unauthorised access to tuple space.

1.2 A Distributed Server

We have adopted a client/server [7] model for the project. A server daemon will run on each node that is a part of the Linda system. This distributed model allows tuple space to be spread evenly across all nodes of the system avoiding the potential bottleneck of a centralized server and tuple space. Linda programs are linked with a client library which implements the client side of the Linda operations. Each node of the server is responsible for maintaining part of the tuple space and servicing the requests of the clients on its own node. A server node also services requests from other server nodes whose clients have tuples associated with this node.

1.3 Goals of, and Requirements for a Distributed Server

This work concerns the design and implementation of a distributed Linda server on a heterogeneous network of heterogeneous processors. The design of this system was motivated by two main questions:

"How can we use as much of the hardware we have available as possible?" and
"How can we use it efficiently?"
Our hardware consists mainly of moderately powered workstations connected to a local area network. Our network is not particularly fast and our workstations are of various architectures.

What does one expect from a distributed server? What would cause us to say "This is a good/bad server"? We list some goals and requirements that drove the design. The equipment and networks that we have available played a role:

- **Use of Heterogeneous processors:** We have a number of different pieces of computer equipment available for use. They are of comparable computing power (the same order of magnitude). We have clusters of transputers. We have various models Sun workstation. We have '386' PC's running DOS. We have a couple of VAX's and a very old CYBER mainframe. To harness the power of all these different machines in a Linda server is a desirable goal.
- **Interconnectivity:** There are different types of network available for use by the server. For example the Sun workstations use TCP/IP on top of ethernet, while transputers running Helios have a number of ways of communicating with each other, including TCP/IP. The more general inter-processor communication methods are possibly less efficient than specialised communication methods. We would like efficient communication between any two processors in the server without the user having to worry about the underlying network.
- **Reliability:** Can the server survive if part of the network goes down? Will the server crash if one its nodes crashes or if some messages are lost? Will the server crash if a client program crashes? Is it possible to halt the server and continue at some later time? The server's usefulness will be increased if the answer to some or all these questions is yes. Distributed programs using the server should have no reason to blame the server for its unreliability.
- **Portability:** The server is to be run and developed in more than one environment. User programs will most likely be expected to run in each environment so it is important that it is easy for the user to write programs that will run with little trouble on all these environments. The server must be developed in such a way as to be easy to port to a new environment. The protocol for communication between elements of the server should be powerful enough to allow different implementations of server components to communicate without modification.
- **Efficiency:** The server must make good use of its resources. The most important resource is the network; the amount of data exchanged and the number of messages sent should be kept

as low as possible. Computational overhead should be kept low and memory should be used carefully. Provision should be made for load balancing.

- **Ease of Use:** The server should be relatively simple to use. It should be simple to configure and distributed programs using the server should be easy to start, preferably as easily as running a standard executable.

1.4 Thesis Outline

Firstly, a communication protocol is developed for use on reliable point to point streams. The stream may be provided in a number of different ways, allowing a pair of processors to use a less general, more efficient methods if necessary. The protocol defines all interaction between servers and between a client and a server and tries to not restrict the implementation in terms of language or internal structure. The architecture need only support the base field types i.e. 32 bit integers, IEEE-754 floating point numbers, strings and byte fields. Server communication is discussed in chapter two. The base transport mechanisms available are described, network topologies and machine architecture considerations are discussed. Finally the structure, but not the semantics of the protocol is described.

Chapter three discusses tuple-space distribution. It discusses tuple replication and shows how tuple space can be partitioned. Chapter four concerns the design of the server. It discusses various aspects of the design including portability, the role of the pre-processor and a pros and cons of a multi-threaded server. It describes the server implementation and the operation of it's components.

Chapter five uses Milner's calculus of communicating systems to model the message passing semantics of the distributed server. The message passing semantics of the distributed server has been modelled and will be shown to be equivalent to the message passing action of a generic Linda black box.

Chapter six discusses some of the research related to this work and chapter seven describes some of the enhancements and future work that can be done on the server.

This work does not cover all aspects of the server. It assumes the existence of a preprocessor which performs important tuple space analysis and syntax transformation. Process and tuple placement are not dealt with in detail.

Chapter 2

Server Communication

2.1 Introduction

Parallel algorithms can be classified by the number and complexity of the processing elements involved. The complexity of individual problems relative to the overall problem is known as the granularity of a problem. Linda is suited to coarse-grained problems. This is due to the cost of interprocess communication since, in almost all cases, a processor is able to get a significant amount of work done in the time it takes to send a message. A more fine-grained algorithm would spend most of its time communicating the results of simple computations.

The cost of interprocess communication tends to be the bottleneck in most Linda systems and so it is essential that the most efficient use is made of the available transport mechanisms. Efficient Linda systems tend to be those that use interprocess communication no more than is absolutely necessary, and so are able to work effectively on more fine-grained problems. It has been noted [5] that for many Linda applications, the task granularity can be easily modified to work better on systems which have a higher overhead due to interprocess communication.

Message passing overhead can be reduced by locating the part of tuple space that a client uses on, or close to its host machine. Broadcast messages, to locate or distribute tuples should be avoided, especially if the transport medium does not support efficient broadcasts. Network latencies tend to be high in the transport media currently being considered and so if the opportunity arises to batch small messages and send them as one message, it should be taken.

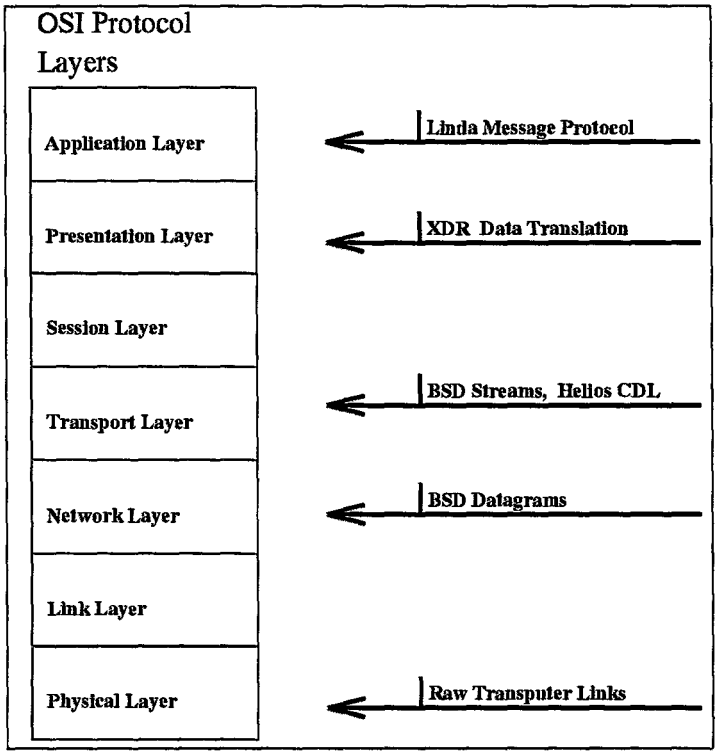

Figure 2.1: The various transport mechanisms, relative to the OSI protocol layers.

The rest of the chapter will describe the mechanisms available for communication. These mechanisms provide services at various levels. Some are more sophisticated than others. For example TCP/IP streams provide point to point reliable communication, hiding the underlying network, while dedicated transputer links provide only a raw transport medium between two pieces of hardware. When compared to the OSI model for communication, each mechanism can be placed somewhere on the layer hierarchy, providing services similar to one of the layers (see Figure 2.1). With respect to the OSI model, our goal is to provide a presentation layer interface to the nodes of the server. The chapter shows how the various mechanisms have been used to provide this service in an efficient way. The server itself may be considered to provide an application level interface to the user. It should be noted however that we do not model the OSI layers specifically: they are merely used as a reference. Ways in which various machine architectures differ are discussed, and how the communication protocol has to deal with them. Finally the communication protocol that was chosen is discussed.

2.2 Base transport mechanisms

A number of machine architectures have been considered as targets for the distributed Linda server, namely the Inmos T400 and T800 transputers, the Intel 80386 microprocessor, the Motorola 68000 range of processors and the Sun Sparc RISC chip.

All these microprocessors have a Unix-like multi-user multitasking operating system available as a development platform. The facilities and abstractions offered by such an environment greatly enhance the ease of development of a Linda server, the most important of which is the operating system's ability to largely hide the differences between microprocessors, by presenting a high level common interface to the user.

2.2.1 Dedicated Transputer Links

The Helios operating system is a Unix-like parallel operating system which runs on a cluster of transputers. The resources available are arranged as files in a file-system. A resource is managed by a server and clients communicate with a server via a general server protocol. Helios eases the distribution of processes across a number of processors. It is able to automatically place processes on a suitable processor, hiding the details from the user. It provides abstractions for simple communication between different processes regardless of which processor they reside on.

Helios usually reserves all the available transputer links for its own use. Programs running under Helios usually use Helios's kernel functions for interprocess communication. However it is possible to reserve a particular transputer link for your own use. This offers little advantage if large chunks of data are being exchanged, but for small messages the Helios system overhead is much larger than the time taken to transfer a message. In the case of Linda, the messages involved are usually small so using dedicated links will be much faster. However using dedicated links makes a system extremely non-portable, since we can now only use transputers, and the Linda network has to be modified to work with every new network topology.

2.2.2 The Helios Component Distribution Language and stdio

The Helios operating system has a mechanism which is an extension of the Unix stdin/stdout/stderr, pipeline and redirection system. It allows components which execute concurrently to communicate with each other via the standard I/O ports plus a few more. Interprocess communication under this system could be expressed as:

```
fprintf(stdaux2,"hello process2");
```

The disadvantage of this high level of abstraction is that there is almost inevitably a large system overhead.

2.2.3 TCP/IP

TCP/IP is a widely used non-OSI protocol. It provides services similar to that of the transport and network layers of the OSI model. TCP/IP consists of two protocols. IP, the Internet Protocol provides services approximately equivalent to the network layer. TCP the Transmission Control protocol, provides services approximately the same as those of the transport layer.

Berkeley Sockets

BSD Unix is a common operating system and so computers using BSD Unix can be treated more or less as identical Unix virtual machines. The BSD method of interprocess and intermachine communication is a method called socket communication.

Sockets are implemented using the TCP/IP protocol and so this method is a very portable method of communication. Two types of connections are provided: a point to point stream and an unreliable datagram service.

Streams

Here the user is presented with a reliable point to point stream of data. The underlying network is hidden from the user and any errors are corrected. The disadvantage of this is that it is a little

more general than what is required. The server tends to send messages as short bursts of data at discrete intervals. The underling implementation of the stream uses a datagram packet service and the added abstraction of the stream adds inefficiencies.

Datagrams

The other service provided is a datagram service. A datagram is a block of data with a fixed maximum size. Datagrams are not guaranteed to reach their destination, neither are they guaranteed to arrive in the order in which they were sent. Messages may also be duplicated. The advantage of using this service is that is more suited to the types of messages that a Linda server will send. To use datagrams the server must provide a way of breaking large messages into datagrams and putting them back together in the correct order, and must resend datagrams that are lost.

2.2.4 User Datagram Protocol UDP

When trying to maximize performance, the usual approach is to program as close to the underlying hardware as possible, in the belief that the higher level services provided introduce unacceptable inefficiencies because they were not written with a specific goal in mind. For example writing critical code in machine code, often provides good speed-up, usually at the cost of having difficult to maintain, machine depend code. Berkeley sockets of type SOCK_RAW provide access to internal network interfaces and are only available to the super-user. They provide no error detection or correction, and do not detect packet loss, which must be dealt with by the tuple protocol.

Murakami et.al. [29] use the User Datagram Protocol to implement their Tuple Operation Suite.

2.2.5 Shared Memory

If processes wishing to communicate are on the same machine, the ideal communication is shared memory. A process only needs to inform the other process that information is available, and the other process can then use it. It is not even necessary for data to be copied. The disadvantage of this is that it is clearly not possible with processes which run on different machines and have no shared memory. One case where two processes are often on the same machine is a client and its server. In fact with the current implementation, it is always the case. Shared memory mechanisms

tend to be machine dependent (although System V Unix provides some sort of general shared memory interface), so using shared memory tends to make a server less portable. There are a number of ways in which shared memory can be used:

- First: One process can prepare a message in a block of shared memory and then inform the other process that the message is available. One must be careful to ensure that the first process is prevented from modifying the message after the second process is notified of its existence.
- Second: Shared memory can be used as a buffer to copy data from the address space of one process to the address space of the other. Shared memory is used to provide a communication stream between the two processes. The default way of communicating between two components of the server is to use a data stream, so using shared memory in this way does not add complications to the server.

Bjornson et.al. [4] indicates that although the extra work required for the second method is noticeable, it does not significantly affect the overall performance of the system. This was based on observing the performance of a particular shared memory multi-processor.

2.3 Message Passing Performance

Message Size (bytes)	TCP Streams	IP Datagrams	Transputer Raw Links	Helios streams
32	19	35	694	22
64	42	62	735	44
128	72	110	833	87
256	108	185	847	170
512	169	327	845	298
1024	248	464	858	457
2048	283	588	862	724

We decide here which mechanisms will be used and why. The table lists some of the transfer rates achieved using various communication mechanisms. The data for the transputer communication mechanisms is taken from Davies [17]. The first column lists communication rates between two Sun workstations on an ethernet LAN using TCP streams, the second lists the rates achieved using IP datagrams. The third column lists the rates achieved by using assembly language macros across

two dedicated transputer links and the last column uses the Helios system library functions to measure communication across two transputer links. Message sizes are in bytes. Transfer rates are in kilobytes per second.

The performance of IP datagrams is roughly twice that of TCP/IP streams but does not take packet retransmission and error checking into account. The performance of TCP/IP streams is not particularly bad and TCP/IP provides all of what we want , but TCP/IP is a general streams package and tuples are more like packets. So we would like to be able to override the TCP stream in specific cases where other methods are better.

Network latencies are high, so if we have more than one message to be sent to a particular host, it would be a good idea to batch them.

Linda operations generally consist of small messages rather than a long stream of data. SunOs's TCP stream implementation tries to minimize the number of messages sent (because of the latency problems mentioned above) and so waits for a short period after it has received data, in the hope that more data will arrive. This should be disabled, and implemented by the server where necessary for Linda communication streams. This is possible with the TCPNODELAY option to the **ioctl()** system call.

2.4 Network Topology

There are a number of different network topologies which may be available to a Linda server:

A Token Ring

Here the servers are connected in a ring. The token is either used to grant permission to use the ring or to grant delete access to a tuple space. Here broadcasting to all nodes is as cheap as sending a message to a particular node. Zenith [46] has developed a system which allows the addition of multiple sub-rings and claims that it is scalable to an interesting number of nodes.

A Richly Connected Cluster

To have every processor connected to every other would be ideal if it were available, unfortunately the number of connections increases with the square of the number of nodes. The modules of the Linda server (other than the transport module) assume that it has a richly connected cluster to work with, but the server designer must of course be aware that this is not the case.

A Heterogeneous Network

Here there are connections of varying capacity, fast links between subsets of processors, some processors are far removed from the rest of the network, and there is often no direct route between processors. Great care must be taken when placing workers an the parts of the tuple space which it will use. Most formal analyses assume a homogeneous network of identical processors.

2.5 Machine architecture considerations

Machines differ in the way they store data.

Some architectures store integers with the least significant byte in the lowest part of memory, these are termed 'little-endian' architectures. Others store the most significant byte first, these are termed 'big-endian' architectures. Also some architectures place restrictions on the way data is stored, the addressing mechanism only allowing the storage on data on four-byte aligned boundaries for example.

Linda programs communicate with each other by placing some of their data into tuples and placing them into tuple space , or retrieve tuples from tuple space and extract their data. It is important that this is done efficiently. So we want the Linda data-types to be as close as possible to those of the Linda program. Also it is desirable to choose the internal representation of a tuple so that it can be manipulated easily on the architecture on which the Linda program is going to be run.

We have chosen four data types to make up Linda tuples, namely integers, floating point numbers, strings and byte arrays. We do not have structured data types, since we consider the task of breaking up a structured type into its component parts best suited to a preprocessor, one whose

other tasks would be to allow tuple space operations to be represented in the true Linda syntax regardless of the underlying language, and to partition the tuple space. All this begs the question: 'What type of integer or floating point?'. In the past, most Linda implementations, and all here at Rhodes have been targeted at a specific architecture. So the answer would be: 'The type of integer or floating point number that that machine uses.' In this project however, one of the goals is to have a heterogenous collection of processors work together on a Linda program.

We have chosen to use 32 bit integers, since it is the size preferred by most of out target architectures. The only remaining decision would be to decide on some internal representation. Floating point numbers are more difficult to deal with. Floating point operations can differ in many subtle ways from implementation to implementation, with the precision of the number only a single aspect. We need to specify exactly how floating point operations will be performed, and how they will be stored internally. Fortunately such a specification already exists.

2.5.1 Sun Microsystem's XDR protocol for external data representation

The problem of machine independent data representation is not a new one and so it is unsurprising that people have already had to tackle the problem. The XDR protocol for external data representation forms the foundation of the remote procedure call protocol. It defines a way of representing a range of base data types in a machine independent way and in particular its integers are 32 bit integers and its floating point numbers follow the IEEE 754 standard. For this reason we have decided to follow the XDR protocol in our representation of the base data types.

2.6 The Linda Message Protocol

This section describes the format that we have chosen for all messages that pass between servers and between clients and servers. The message format used by the transputer version of Linda at Rhodes [44] was used as a base for developing this protocol. It specifies how a server should appear to other components of the system, trying to leave the implementation of the components as free as possible. They are not limited to a particular language or a particular structure. For example it is quite possible to write a library in Pascal or Modula 2 to allow Pascal-Linda or Modula 2-Linda programs to interact with C-Linda programs. The server may be written with a multi-threaded structure instead of the monolithic structure of the current implementation.

It is designed for use on a reliable point-to-point stream. i.e. It does not deal with error correction or routing. This would be the task of a lower level protocol. We identified the types of messages that pass between components of the system and saw that they fell into two categories: those which exchanged tuples (e.g an OUT() message from a client) and those which exchanged control information (e.g. initiating connections between servers and opening and closing tuple partitions). All messages are in one of these two forms and the protocol has a type field which stores exactly what type of message it is. The components of the messages (e.g. bytes, byte streams, integers, floating point numbers) are stored according to the XDR protocol. The meaning of the Linda messages and how they are used by the server are discussed later when we describe the operation of the server.

2.6.1 Structure

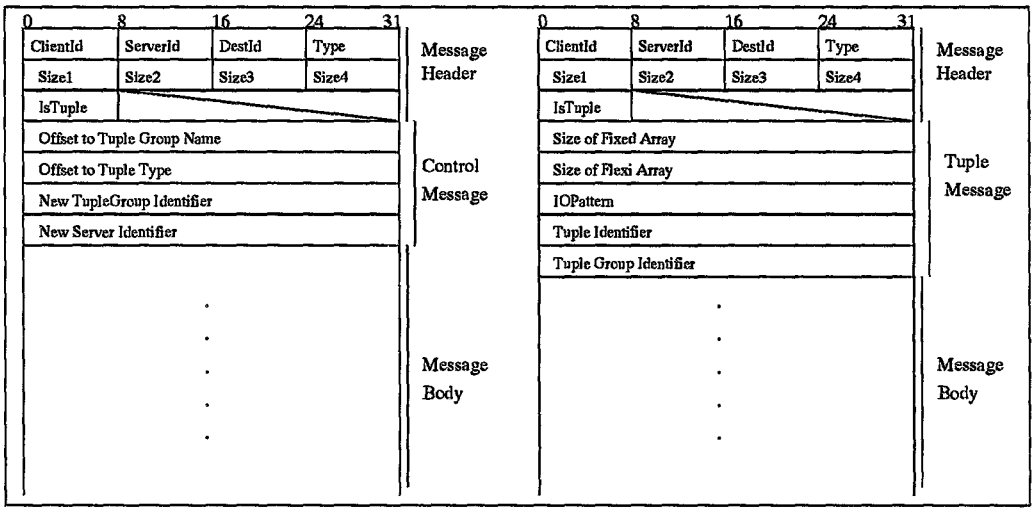

Figure 2.2: The Structure of a Linda Control and Tuple Message.

A message is a linear sequence of bytes, transmittable via stream read or write operations. The same format is used for tuples and templates. The message consists of two parts: a header, followed either by a control message, or a tuple message. A tuple message consists of a header, a fixed-size vector, and a flexible-size vector. The protocol is shown as a block diagram (see Figure 2.2), and as C type definitions (see below).

```
typedef struct RhodaMsgHdr {
```

```
        byte ClientId;   /* the name of a client (unique together with serverid)*/
        byte ServerId;   /* the server associated with the client  */
        byte DestId;     /* the destination of the message */
byte Type;
byte s1,s2,s3,s4;
byte IsTuple;
} RhodaMsgHdr;
```

The type field is a byte value representing the type of message. The symbolic name for the message is one of the following: **mIN, mOUT, mREAD, mINP, mREADP, mEVAL, mHELLO, mHALT, mBYE, mRECONNECT, mOPEN, mCLOSE, mDELETE, mDODELETE, mAPPROVEDOPEN, mQUERYOPEN, mINr, mREADr, mINPr, mREADPr** or **mCOMPOSITE**

```
typedef struct ControlMsg {
        char *TGName;    /* a word offset into this message holding a */
        char *TupleType; /* tuple group name and type signature */
        word TGId;       /* a tuple group identifier */
        word ServerId;
} ControlMsg;
```

```
typedef struct Tuple {
       word FixedSize;   /* Size of fixed array         */
       word FlexiSize;   /* Size of flexi array         */
       word IOPattern;   /* 32 bits of polarity data         */
       word Signature[3];/* Encoded arity and type signature
       word TupleId;     /* Unique tuple id.          */
       word TGId;        /* id of the TupleGroup to which this tuple belongs */
       word Fixed[Tuple.FixedSize];  /* ptr to fixed array in this structure */
       byte Flexi[Tuple.FlexiSize];  /* ptr to flexi array in this structure */
}Tuple;
```

The 'fixed' array of words holds one item per tuple field: integers and reals are held directly in this array, and for each string or byte block, the fixed array holds the size of the flexi field. The size of the array for the fixed objects is bounded by the maximum number of fields (32 in this case) times the maximum size of any fixed-size field (an 8 byte real), so in practice it is a small array and intermediate storage provides no problems.

The 'flexi' array holds the actual string or byte blocks of data and can be potentially huge.

The structure does not include a checksum since the protocol was designed for use in a reliable point to point stream.

2.6.2 IOPattern

The iopattern defines the polarity of the tuple fields - each 0 bit in the iopattern denotes a formal field, each 1 bit denotes an actual field. Only the least significant 'arity' bits of the iopattern are used - the others are padding bits with don't-care values. The bits are ordered so that the least significant bit corresponds to the first tuple field, the next bit to the next field, and so on. Tuples are restricted to 32 fields. Both tuples and templates may contain formal fields.

2.6.3 Type Signature

Only four types are currently supported:

```
"i"  integer  a 32 bit integer
"r"  real     64-bit IEEE 754 format floating point number
"s"  string   arbitrary length array of char, with leading count field.
"b"  byte[]   arbitrary length array of byte, with leading count field.
```

The three words comprising the signature hold information about the 'arity' of the tuple (its number of fields) and their types. These words have the following contents:

```
word arity_mask;    /* Least significant 'arity' bits are 1    */
```

```
word scalar_mask;    /* 1 bit in each position if int or double */
word sub_type_mask; /* int/double or string/byte[] flags */
```

The scalar mask tells whether a count field is present or not. If the type is scalar, the sub type mask denotes integer (0) or double (1). If the type is non-scalar, the sub type mask denotes string (0) or byte array (1)

2.6.4 Key

The unique `TupleId` id can be used to extract the creator information, (top 8 bits) and a unique sequence number (24 bits).

2.6.5 Data

No information is stored in the fixed or flexi array for any formal fields.

Each actual field in the tuple has an entry in the fixed array, denoting either the item itself (integer or float), or its size in the flexi array.

```
The storage is as follows:
     integer field:           stored in one word of the fixed array.
     double field:            stored in two words of the fixed array.
     string or byte field:  count is stored in one word of the fixed array.
                              Data is stored in count successive bytes of the
                              flexi array, and followed by zero or more
                              alignment bytes.  The null byte at the end of
                              a string is stored in the tuple.
```

2.7 Summary

In this chapter we have discussed various transport mechanisms for IPC and chosen TCP/IP for its generality. We allow for other base IPC mechanisms and allow for local optimizations where another mechanism proves to be more efficient. We have defined a Linda message protocol which relies on a point to point reliable data stream. The components of the server e.g, the server nodes and client programs, use only Linda messages to communicate with each other.

Chapter 3

Tuple Space Distribution

This chapter discusses tuple replication, the various tuple placement schemes and tuple space partitioning. The trade-offs of localized or replicated partitions are weighed against each other. Partition and process placement are not dealt with.

Tuples must be placed somewhere, the best place is one which gives the fastest average transaction speed i.e. create a tuple, store it in tuple space, ask for it and then retrieve it. This depends on the message speed and the number of messages sent. In the systems we are dealing with message are expensive to send and there is a relatively high latency. Processors are not all equally powerful and the networks connecting them are not all equally fast. This chapter will discuss the various tuple placement schemes and compare them with respect to the number of messages sent and the cost of the individual messages

3.1 Tuple replication

An important decision to make is whether to keep multiple copies of tuples. There are advantages and disadvantages to replicating tuples and to keeping only a single copy of a tuple. We discuss the various tuple replication schemes.

3.1.1 No Replication

Here a particular tuple has only one instance in the entire tuple space. It may be stored at a predetermined place or at the place where it was created (by an OUT()). The advantage of this is that the OUT() operation is usually cheap, but READ() and IN() are relatively expensive since the tuple must be found by sending messages to all possible locations, although it is possible to limit this by using a hashing scheme.

hash distribution

Bjornson et.al. [5] and Tonsig [41] use tuple hashing in their Linda implementations on a iPSC/2 Hypercube and a cluster of transputers respectively. The iPCS/2 scheme can be summarized as follows:
A preprocessor partitions tuples into disjoint sets, the sets falling into 2 different classes: those which have a key that can be computed and those which do not. Tuples from a set in the first class are stored evenly across all nodes in the network in a distributed hash table. Tuples from a set in the second class are all stored on a single node hashed on the class key. (The class key essentially encodes the structure of the tuple, i.e. its arity and field types, rather than the contents of the tuple)

3.1.2 Full Replication

If a tuple is replicated on every node, the read operation READ() will be very cheap but IN() will be expensive because we must ensure that only one IN() of a tuple will succeed. OUT() requires a broadcast message and will be expensive unless the underlying network can provide broadcasts cheaply (A token ring for example). It also takes much more memory to store a particular tuple, since there must be one copy per node.

3.1.3 Partial Replication

This offers some advantages of both the above but also some of their disadvantages. Less memory is used than full replication, while READ's are faster than the non-replicated scheme without hashing.

Partial distribution can be described better as one instance of a class of distribution methods called uniform distribution. [4]

Uniform Distribution

A tuple generated at a particular node is sent to a predetermined set of nodes and a request for a tuple also generates a search at a (possibly different) predetermined set of nodes. This scheme will always find a tuple if its available if, for all nodes, the set of search nodes intersects with at least one node of the placement set of every other node. If each placement set contains only one node, no tuple replication occurs. If all nodes are in every placement set, full replication occurs. One particular implementation [5] has its server nodes configured in a grid of busses, with the placement set defined as the node's row and the search set defined as the node's column.

3.2 Disjoint Tuple Spaces

It is often the case that different classes of tuples are mutually exclusive and so permission to delete a tuple need not lock the entire system. Some tuples are read by only certain workers and so parts of tuple space can be stored close to the workers which use them. This can improve the efficiency of large Linda systems by giving them the performance of smaller systems.

One of the goals of this project is to produce a Linda system which runs on a heterogeneous network. It is conceivable that there may be subnets of tightly linked processors, for example a cluster of transputers running Helios. If we have an application that makes particularly heavy use of the Linda network, it may be a good idea to partition tuple space so that the tuples that will be used in that application only get stored and searched for on nodes in that subnet. In order for this to be effective, we must ensure that EVAL processes are placed effectively.

Another example at the other end of the scale, certain 'trivially parallel' applications (e.g. rendering, ray-traced animations) can produce good speedup even on extremely slow networks. (e.g. a Linda network consisting of the local area network of a number of universities). If one partitions tuples by the subnet on which their generators run, by default, we would not incur the cost of using the slow links, but in certain cases, using the entire network might be feasible.

3.2.1 Partioning Criteria for Tuple Space

There are a number of ways in which tuple space can be partitioned. The analysis is usually performed by a pre-processor at compile time. The following are some important partitioning criteria [4] [10]:

- **By Arity:** Two tuples which differ in the number of fields cannot possibly match.
- **By Type Signature:** Tuples must agree exactly in type for each of their fields for matching to be possible. A formal field in this implementation has a type, so partitioning by type signature is possible.
- **By Considering Constant Fields:** If a field is always a constant, the precompiler can use the values to partition tuple space. Partitioning by constant fields limits the possibility of using Linda as a persistent store, since a later Linda program may create a tuple where a particular field is no longer constant. In order to make use of constant fields, we have to process a collection of Linda programs as a whole, and not allow them to interact with other programs.
- **By Application:** It is usually not desirable to have the data of one application visible an modifiable by another application, and so it is important to be able to separate applications from one another.

The last two partitioning criteria bring to the fore an issue in our perception of Linda programs, namely persistence of data: "Do tuples exist and have meaning past the run-time of a Linda program?".

If one considers tuple space to exist only for one application and only for the runtime of that program, one is able to perform significant optimisations in one's internal representation of tuple space. A major thrust of the YALE Linda project has been to develop such optimisations. These optimizations rely on facts such as "this tuple field uses only constants" or "only these nodes use this class of tuple" and to make such statements, one has to know about all Linda programs at compile time. This does not eliminate Linda as a persistent storage mechanism though.

The approach to this project has been to have a server daemon which accepts connections from clients and manages tuple spaces specified by clients, which is why the issue of persistence was

brought up: It requires extra work to remove this potentially useful feature. But in order to make best use of a precompiler and allow multiple applications to use the server at one time, it is neccesary to make use of partitioning by application, in order to protect applications from one another.

3.3 The strategies decided upon for this Project

- **replication:** For the moment we have decided that the merits of tuple replication do not outweigh the disadvantages and so this implementation stores a tuple in only one place.
- **distribution:** All tuples in a partition are stored on the same machine, although different partitions can be distributed across machines. It is possible to distribute tuples in some partitions across machines using a hashing scheme and this is a proposed future enhancement.
- **partitioning:** Tuple partitioning is performed by a pre-processor, which hides partitioning information from the user. The server relies on the preprocessor to perform the partitioning. The server manages partitions as a collection of files.
- **placement:** We do not analyse the Linda program to decide the best placement of tasks and partitions. We currently place a partition on the processor of the task that first opens it. This does not seem to be a bad first choice of location.

Chapter 4

Server Design

This chapter discusses aspects of the server design. It discusses the role portability plays, the pros and cons of a multi-threaded server are considered. The role of a preprocessor and some aspects of the X Window System protocol are also discussed. Finally, the structure of server and its major components are described.

4.1 Portability

Portability is a desirable quality , especially when one wants a server to run on a number of different architectures, working together as a single program. It is possible to write portable code, but one must take care in a number of areas:

- **Language:** The target language should preferably be available on all target environments and be language with a well defined specification. Ansi-C fits this description, and is the most common language on Unix-like operating systems, which are our primary targets and so it was our language of choice for the project. Note however that a single language is not strictly neccesary, one could conceivably implement the server in a number of different languages and in different ways if one pays strict attention to the final point. We shall explore this idea briefly in the last chapter.
- **Operating System:** A powerful operating system eases one's task greatly. It provides many useful high level features (e.g. process and job control, light-weight processes, high

level message passing routines, the hiding of the computer architecture and many others). The target operating systems for this project were a number of different flavours of Unix, and possibly MS-Dos. It is useful to choose the 'least common denominator' when it comes to features to use since the utility of a particular feature will probably be negated if it can only be used on part of the system.

- **Code:** Care must be taken to avoid platform specific features which are often present. In the case of C, where platform specific features must be used, one can write code for each individual platform and let the pre-processor choose the relevant code at compile time. One platform specific library, which may well have been used if it were widely available was Sun's lightweight process library. The design of the server would have been significantly different if the library had been widely available.
- **A well defined interface between nodes and modules:** One way to implement a server on several platforms is to have a different, machine specific implementation for each platform, each conforming to strict specifications. This is possible in this particular design, because the operation of the server is relatively simple and straightforward.

4.2 The Mythical Pre-Processor

An important part of the system is a pre-processor. It is a good place for analysing and optimizing the Linda Program. The existence of the pre-processor effects the design as it allows the server to present an interface which was closer to its internal structure. In fact, the transformations performed by the pre-processor, partially dictate the structure of the server. For this reason we discuss briefly some of tasks suited to a pre-processor that have been identified from the literature [44] [15] [4] [13] [18] [10] and how they impact the design of the server. The pre-processor used at Rhodes performs only some of these functions and the current server deals only with these, others are either invisible to the server or require minor modifications. Others require additional functionality in the server.

Functions of the preprocessor:

- **partitioning:** tuple space can be separated into disjoint partitions as described in chapter 3. The server was written on the assumption that this has already been done and so the system calls in the client library are of the form:

```
TSOperation(TupleSpaceIdentifier,Field1,...Fieldn);
```

and not:

```
TSOperation(Field1,...Fieldn);
```

and internally the server operates as a maintainer of a collection of files, each file being a tuple space partition.

- **storage mechanism:** Tuple space partitions can be classified by the formal/actual signatures of the tuple space operations that use them [24]. For example pre-processor analysis can yield fields which can be used as hash keys or detect that two processes use tuple space as a kind of message stream. As a result of this, some partitions need only be stored as a queue of tuples, others can be stored in a hash table, while others need an exhaustive search in order to locate a tuple. The type of storage needed needs to be communicated to the server when the partition is created. The current version of the server however uses only the general storage method.

- **tuple batching:** A large portion of the time taken to send a message in the current environment is due to latencies rather than data transfer rates. This makes trying to reduce the number of messages exchanged between components an important optimization. Messages of type mCOMPOSITE have a structure similar to tuples with a type signature of a number of byte arrays and allows a number of messages to be sent as one message, when a pre-processor decides that this is possible. The current version of the server does not implement composite messages.

- **syntax modification:** The syntax of the host language may not allow us to express tuple space operations as we would like. Modifying the syntax to some form convenient to the server is a task suited to the pre-processor, and should not significantly effect the operation of the server. for example the server does not handle complex data types, since the best way of dealing with these would be to have the pre-processor break complex types into their component parts.

- **program transformation:** There are cases where tuple space operations can be changed or re-arranged to produce a more efficient program [13]. This should be possible without effecting the server in any way.

- **tuple updates:** It has been suggested [13] that operations which are intended to modify a tuple and expressed something like:

```
IN("variable",?value);
OUT("variable",new_value);
```

should be identified and replaced with a new operation to modify a tuple inside tuple space. This operation will probably require less than half the number of messages as before. This would require changes in the server to cope with this.

4.3 The Server Structure

We discuss the internal structure of the server, comparing threaded and non-threaded servers.

4.3.1 The X Window System

The X window protocol was investigated, while the Linda server was in it's design phase and a number of interesting points were noted, some of which influenced the Linda server design:

- The X server was designed to be robust with respect to client failures. In this respect it never trusts the client to provide the correct data. Also it is able to continue with other work if it has to wait for input from a particular client.
- X tackles the byte ordering problem by providing two connection ports, a little endian port and a big-endian port. Clients always send in their own ordering and the server translates only if neccesary (i.e. only one of the ports). The alignment problem is solved by requiring that blocks are multiples of 32 bits and each 16 and 32 bit quantity within a block is 16 and 32 bit aligned respectively. This means that the XDR protocol is not used.
- Some events are unavoidably synchronous. Round trip time (server to client to server) plays an important part in deciding whether such an event is necessarily implemented within the server e.g. mouse tracking. X was designed to work in an environment where the round trip time is between 5 and 50 milliseconds.
- X protocol events are variable length with a length field.
- X is a single-threaded server that polls its clients rather than a multi-threaded server.

4.3.2 Threaded vs. Non-threaded Servers

Conceptually the operation of a server is to wait on a number of I/O streams, to read instructions and data from the streams as it becomes available and to act on the instructions, modifying the internal data structures as neccesary and to send data down certain streams as neccesary.

There are two important ways of handling this situation, either a monolithic server maintaining all connections or a multi-threaded server, where each thread maintains a single connection. One design decision was whether or not to use a threaded server. Experience with the single tuple space transputer version showed that a multi-threaded server had many desirable qualities. The idea was to have one thread for each connection. The operation of the thread was relatively straightforward, access to global structures was guarded by locks in relevant places, and having a thread wait on a message did not really affect the operation of the rest of the server.

On the other hand a non-threaded server can be much more complicated. A single process has to deal with a number of different connections, each at a different point in its interaction with the server. The state information associated with each connection has to be stored somewhere, while dealing with other connections and restored when control returns to first connection and one ends up writing a thread multitasking package. Waiting unnecessarily on a message delays the entire server and can easily lead to deadlock when it is distributed with instances of the server on different machines. Therefore it would appear that a threaded server is a much more efficient and elegant way of writing a Linda server. Unfortunately thread implementations are not all that common, and where they do exist they are not standardised in any way. This is a significant argument against the use of threads if having a Linda server run on a heterogeneous collection of processors is one of the goals of the design.

Therefore the possibility of a non-threaded server was re- examined. It was noted that the interaction between the client and server was relatively simple, the client sending a message, (representing the Linda primitives (OUT(), IN() etc.)) and blocking waiting for a reply. If a server was to break up its operation into transactions as follows, things might be a little less complicated:

- The transaction begins with the receipt of a message from one of the server's connections.
- The server does some processing and sends any necessary operations which are guaranteed not to block (e.g a reply to a client or a message to the courier, because we can test if it is waiting.)

- Any possibly blocking messages are queued to a courier, which is a separate process to the server daemon, [25] whose task is simply to receive messages from the server and try and deliver them.

The messages sent to the courier are not essential for the operation of the server and could easily be discarded with the drawback that some client would hang on one of its operations. Messages from other servers are treated much the same as messages from clients.

An advantage of this approach is that it allows the server to remain largely stateless. A client is dealt with in a single transaction beginning with the receipt of a single message. This transaction does not depend on previous transactions. Some 'state' information is kept though in the form of IN or READ operations pending delivery. When the time comes to deliver these messages, the server is able detect whether the client is blocked waiting for them and if not, flag an error. The server then is able to remain resilient to client failure.

Figure 4.1 shows a sample Linda network. A server maintains connections with its own clients and other servers. These connections are all treated in the same way, and the receipt of a message from one of them begins a transaction.

4.4 Server Implementation

This section describes the structure of the server. For this implementation we chose a non-threaded server based on transactions similar description above. The section shows the breakdown of the server into logical and conceptual modules and how they relate to each other. See *Figure 4.2* for an illustration of the relationship between modules.

4.4.1 The Connect Module

This module is responsible for starting up the Linda server daemons on the different machines. For each machine, it will invoke the relevant copy of the server on that machine and establish links with its neighbours. Each server consults a configuration file containing details about it's neighbours. The server initiates connections between servers lower than itself in the list and accepts connections from servers higher than itself in the list. Our current network allows direct connections between

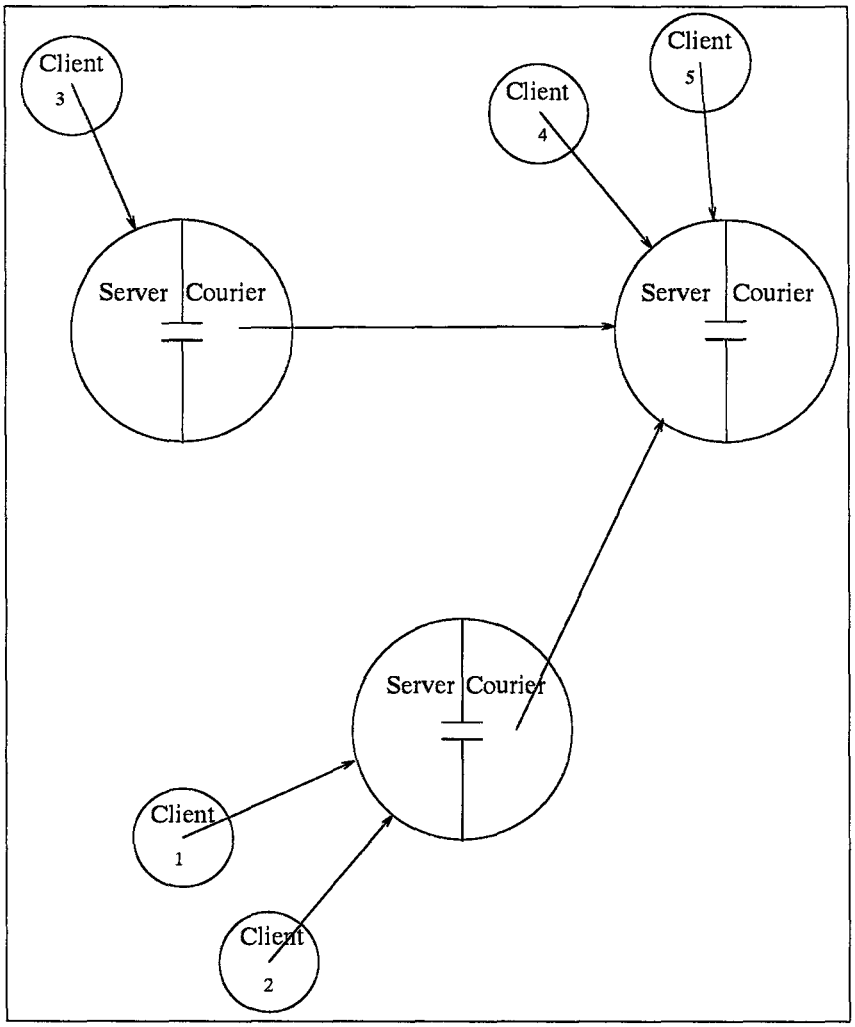

Figure 4.1: A Linda Network showing, Server nodes, clients and some transport connections .

each node in the system and so we have not implemented indirect connections, to implement this the entry for the indirect node, would contain the name of an immediate neighbour who is to be the first link in the route to the destination. The server is able to detect messages not destined for itself and will be made to pass them on, according to its own tables.

4.4.2 The Transport Module

This module controls the transport mechanism between servers. It defines a number of communication primitives which are used by the rest of the server. The implementation of these primitives

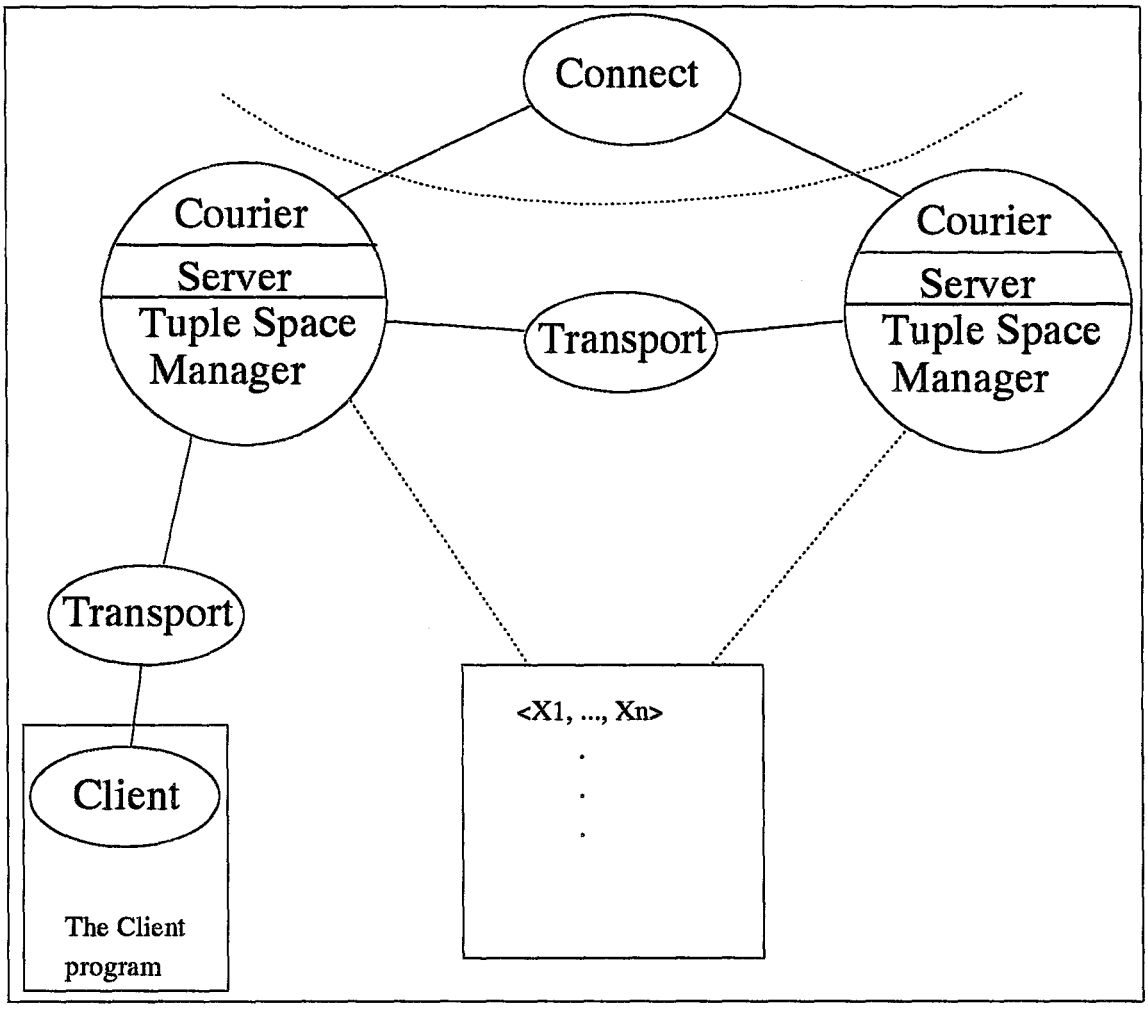

Figure 4.2: The relationship between server modules.

may be different for different nodes in the server. Different communication mechanisms might be used depending on the location of the source and destination of the message. For example to pass a message between a client and its server on the same machine, the message may simply be a notification that a piece of shared memory is available with the body of the message in it. The only way a primitive appears different to the user is in the speed of its execution.

4.4.3 The Tuple Space manager

This module maintains the distributed tuple space and provides functions for use by the rest of the server. The tuple space manager of each node may store part of the tuple space and knows where

to look for any tuple. The rest of the server is unaware of the physical structure or the distribution of tuple space. It presents a tuple to the tuple space manager which either stores it or adds to the list of messages which need to be delivered, or requests a tuple which the tuple space manager either returns immediately or adds to the list of messages for delivery.

The Messages which may be received and the Server's response

We list here the different types of messages which may be received and the server's response to them:

- **mIN, mOUT, mREAD, mINP, mREADP, mEVAL:** If this message is from a client, the server will consult its own database if it maintains the relevant tuple partition and possibly send a reply to some of its own clients and generate some messages for the courier. If it does not maintain the tuple space, it will it will pass the message on to the relevant server via the courier. It puts its own name in the from field. If this message is from a server it will consult its database and possibly reply to some clients or generate reply versions of this message for delivery by the courier to other servers.
- **mINr, mREADr, mINPr, mREADPr:** The server has previously sent one of the above messages to another server. The only response to this is to these messages is to reply to some of its clients.
- **mHELLO** This message is exchanged when the system is started. A server node greets all nodes above it in the configuration file, and accepts greetings from all servers above it in the file. The courier has not yet been started and we do not want two servers simultaneously sending messages to each other.
- **mHALT:** This message is sent to a blocked client, telling it to terminate.
- **mBYE:** This message is sent by a server to other server nodes, informing them that it is terminating.
- **mRECONNECT:** This message will be sent to a client, telling it that it should attempt to connect directly to another server.
- **mOPEN:** This message is sent by a client, attempting to open a new tuple partition. A **mQUERYOPEN** message is sent to the master server node to enquire whether the partition

has already been opened by another node. The master node keeps a track of how many clients are using a connection.

- **mQUERYOPEN:** This message is sent to the master node asking permission to open a partition.
- **mAPPROVEDOPEN:** This message is received by a server node from the master node approving the opening of a tuple partition. It contains node on which the partition is stored.
- **mCLOSE:** This is sent by a client, letting the server know that the connection on which the message arrived is no longer needed. It forwards this message to the master node.
- **mDELETE:** This message is sent by a client, telling the server to delete a tuple partition. The partition will be deleted when the last client has closed its connection. The server knows this when it receives a **mDODELETE** message from the master node.
- **mDODELETE:** This message is received from the master node telling the server to delete a partition.
- **mCOMPOSITE:** This is a composite message. Its data is a tuple whose fields are byte arrays. Each field contains a message. The server completes one transaction loop for each message in the composite message. This message is currently not used.

4.4.4 The Server module

This is the main program for the server daemon. It establishes links with clients who wish to use tuple space and it services the requests of the clients on the machine on which it runs. It is also responsible for invoking active tuples who are to work for another client.

The server is divided into two main sections, the server proper and a courier which is another process which performs message transfers which might block, on behalf of the server.

The Server Transaction loop

The server works as follows: it receives a message from one of its connections, either a client or another server (they are treated in the same way), it performs the processing it can without

performing a possibly blocking message transfer and queues all the blockable messages for the courier. The blockable messages are not essential to the operation of the server, although a client may delay indefinitely if a message pertaining to it is lost or delayed.

The Courier

The courier [25] is a separate process, connected to the server with a Unix pipe, which is started with the server, whose task is to deliver messages on behalf of the server. Its operation is to notify the server that it is ready, accept a message and then try to deliver it. If the delivery fails, the message is kept for possible later delivery. The server is able to detect if the courier is ready to accept a message and if it is not, the message will be queued for later delivery.

Messages to a process on another machine take much longer to deliver than messages between local processes and the receiver may not be ready to accept the message immediately because it busy with other operations. So the server will have to delay local operations in order to send the message. Another more important reason for having a courier is that the receiver and the sender may be delayed indefinitely, trying to deliver messages to each other. The courier allows the server to avoid deadlock by allowing the server to not send messages that will block indefinitely in delivery.

4.4.5 The Client Library

This is a collection of functions which are linked in to the client program. The only functions which the client can see and use are the Linda primitives IN(), OUT(), EVAL() etc. It also has functions for opening, closing and deleting tuple space partitions, which can be used by the pre-processor.

4.5 Implementation of Eval

The current server has a weak EVAL implementation. It uses the fork() system call to spawn a new process which executes a program whose name is the first field of the tuple to be EVAL'ed. Because this is so slow, we follow the operation of the transputer implementation [44] and invoke the eval process only once per processor. Once started, the process repeatedly accepts active tuples for evaluation. For example:

EVAL("task", $field_1, field_2(), field_3()$);

will be converted by the preprocessor to:

OUT("task-in", $field_1, field_2(), field_3()$);

and will start a process which looks like:

```
while(True){
        IN("task-in",field,?field_2,?field_3);
        r1=field_2();
        r2=field_3();
        OUT("task",field_1,r1,r2);
}
```

Chapter 5

Properties of the Server

5.1 Using CCS to Model the Message passing operation of Rhodes' Linda

This chapter shows how the message passing semantics of Rhodes Linda can be modelled using Milner's process calculus. The operation of the server and the client are modelled at various levels of abstraction, one model showing the interaction amongst nodes of the server, using anonymous messages, the other showing the operation of the server as a single entity, but with the meaning of the messages taken into account.

Milner's calculus is a tool for modelling the message passing operation of communicating processes. We give a brief informal overview of the calculus here, but the reader unfamiliar with the calculus is recommended to consult one of the texts on the subject [34] [42]. Other work using CCS is described in [8], [31] and [36]. The basic 'process' in the calculus is an agent. An agent is defined by describing how it changes state when it sends or receives a message. These 'messages' are termed actions. An action can either be an input action or an output action which is designated by an overbar. There is a special third type of response which is described shortly. For example:

$$Process1 \stackrel{\text{def}}{=} \texttt{request}.Process1' + \overline{\texttt{response}}.Process1$$

$$Process1' \stackrel{\text{def}}{=} \overline{\texttt{response}}.Process1$$

$$User \stackrel{\text{def}}{=} \overline{\texttt{request}}.User1 + \texttt{response}.User$$

$$User1 \stackrel{\text{def}}{=} \texttt{response}.User$$

Process1 is an agent which initially behaves like **Process1** and can either receive a `request` message and then behaves like **Process1'** or send a $\overline{\texttt{response}}$ message and continue behaving like **Process1**.

Agents can be composed to form a new agent. The new agent can accept and send the messages of its parts. There is a third special action (written as τ) which is used to designate the atomic exchanging of messages internal to the system, resulting in a state change. For example the system:

$$User|Process1$$

can receive the message `request` and then behave like:

$$User|Process1'$$

We write this as:

$$User|Process1 \stackrel{\text{request}}{\rightarrow} User|Process1'$$

It can also for example exchange messages internally and change state, which is written as:

$$User|Process1 \stackrel{\tau}{\rightarrow} User1|Process1'$$

We can restrict the external messages that the system can exchange. But we cannot restrict the τ message. For example the system:

$$(User|Process1)\backslash\texttt{request}$$

cannot receive external requests. We can also relabel messages (but not the τ message). So we could write :

$$(User|Process1)[\texttt{request1/request}]$$

to indicate that a system that can receive `request1` messages and treat them as `request` messages.

The most important idea in the calculus is the notion of 'observational equivalence' to express the idea that two systems are to all intents and purposes the same if they appear to some external observer to behave the same way with respect to message passing. A proof of observational equivalence is the presentation of a relation (termed a bisimulation), pairing possible states in the two systems which is best described by giving the formal definition [34]. The $P \xrightarrow{\alpha} P'$ indicates a state change as a result of the α action. The $P \stackrel{\alpha}{\Rightarrow} P'$ indicates an α action as well as arbitrarily many τ actions :
A binary relation $S \subseteq P \times P$ over agents is a (weak) bisimulation if $(P, Q) \in S$ implies, for all $\alpha \in Act$ (the actions of P and Q),
(i) Whenever $P \xrightarrow{\alpha} P'$, then for some $Q', Q \stackrel{\alpha'}{\Rightarrow} Q'$, $(P', Q') \in S$
(ii) Whenever $Q \xrightarrow{\alpha} Q'$, then for some $P', P \stackrel{\alpha'}{\Rightarrow} P'$, $(P', Q') \in S$

We use CCS as a tool for simplifying the potentially complex interaction amongst server nodes, to clarify the operation of the server and to help convince ourselves that the server is deadlock free.

We are not interested in knowing what was passed in a message. When we have a message A $\rightarrow$ B we can assume that any necessary information that was at A at the time of the message was sent to B. What is important is whether a message was passed at all when B needed some information that A had. For example when showing that our blackbox Linda definition is equivalent to the distributed version, we must show that if a client C is expected to have some information after the receipt of a message, the sender must have that information. To be more specific if a tuple group is stored on another processor, we cannot simply have the client send to its host processor and have the host reply with a message, we must show that there was a sequence of messages started after the host received the client request, which reached the tuple store processor (possibly via other processors) and that a sequence of messages arrived from the tuple store to the host before the reply message was sent. We say that a host does not cache tuples in its memory if is not a tuple store for that tuple type.

5.1.1 The Operation of the Client

The basic operation of a Linda client is to send a message to a server, to wait for a response if one is neccesary, and then to do some computation and go back and send another message. The messages requiring a response are the messages representing IN, READ, READP, INP and OPENTG. The

messages not requiring a response are OUT, EVAL and CLOSETG. We omit some message types in our model for simplicity. The server's response to them with respect to message passing is the same as one of the messages which are dealt with. The client can be described as working as follows:

WHILE (not finished)
send a message to server
IF response neccesary **THEN**
wait for response
do some work
END

which in the syntax of CCS is:

$$Client \stackrel{\text{def}}{=} \overline{\texttt{rq}}.Client + \overline{\texttt{rq1}}.\texttt{rp}.Client$$

Note that some information has been lost i.e. **IF** cond **THEN** A **ELSE** B has been loosened to A **OR** B can happen. This simplification is made because we are interested in the modelling the client from the view of the messages it sends and receives, rather than its internal operation. Properties true of the more general model should be true of the more specific case which is more difficult to work with. If the message contents are considered, a refined version of the client can be written as:

$$\begin{aligned} Clientm \stackrel{\text{def}}{=}\ & \overline{\texttt{out}}(t).Clientm + \\ & \overline{\texttt{in}}(t).\texttt{inr}(t').Clientm + \\ & \overline{\texttt{inp}}(t).\texttt{inpr}(t').Clientm + \\ & \overline{\texttt{read}}(t).\texttt{readr}(t').Clientm + \\ & \overline{\texttt{readp}}(t).\texttt{readpr}(t').Clientm + \\ & \overline{\texttt{opentg}}(t).\texttt{opentgr}(t').Clientm + \\ & \overline{\texttt{eval}}(t).Clientm + \\ & \overline{\texttt{closetg}}(t).Clientm \end{aligned}$$

As a first trivial example of bisimilarity which we use to justify the use of the simplified client, we take the above definition and class the messages into two groups, those that require a response, and those that do not. We relabel those in the first group to `rq1` and the second to `rq`. The resulting

equation, written:
Clientm[rq/out,rq/eval,rq/closetg,rq1/in,rq1/inp,
rq1/read,rq1/readp,rq1/opentg,rq/closetg,rp/inr,rp/inpr,rp/readr,rp/readpr,rp/opentgr]
is bisimilar to *equation 1.* (It is also syntactically equal because amongst the algebraic laws for manipulating expressions is the rule $E = E + E$)

We write this out in full and simplify:

$$\begin{aligned} Clientm \stackrel{\text{def}}{=} \; & \overline{\texttt{rq}}.Clientm + \\ & \overline{\texttt{rq1}}.\texttt{rp}.Clientm + \\ & \overline{\texttt{rq1}}.\texttt{rp}.Clientm + \\ & \overline{\texttt{rq1}}.\texttt{rp}.Clientm + \\ & \overline{\texttt{rq1}}.\texttt{rp}.Clientm + \\ & \overline{\texttt{rq1}}.\texttt{rp}.Clientm + \\ & \overline{\texttt{rq}}.Clientm + \\ & \overline{\texttt{rq}}.Clientm \end{aligned}$$

which is equal to:

$$\begin{aligned} Clientm \stackrel{\text{def}}{=} \; & \overline{\texttt{rq}}.Clientm + \\ & \overline{\texttt{rq1}}.\texttt{rp}.Clientm + \end{aligned}$$

5.1.2 Message Passing amongst Distributed Server Nodes

This section models the message passing amongst the nodes of the server and with the clients of a server node. As with the first example of the client, messages which produce the same message passing effect in another process are mapped to a single message. A single node of a server works as follows:

WHILE(not finished)
receive a message from somewhere
IF(it was a client message) **THEN**

send a message to zero or more clients who
should be blocked waiting **OR**
queue some messages for the courier
either a request message to another server **OR**
a reply message to another server
ELSE IF(it was a server message) **THEN**
send a message to some clients who
should be blocked waiting **OR**
queue some messages for the courier
ELSE IF(it was a server reply) **THEN**
send a message to some clients who
should be blocked waiting
END IF
WHILE (the courier is ready)
send the courier a message from the queue
END
END

which can be modelled in CCS as follows. `rq` is a request from a client, `srq` is a request from a server and `srp` is a reply from a server.

$$Server \stackrel{\text{def}}{=} \texttt{rq}.Server' + (*\ \texttt{a client request}*)$$
$$\texttt{srq}.Server'' + (*\ \texttt{a server request}*)$$
$$\texttt{srp}.Server'''(*\ \texttt{a server reply}*)$$

$$Server' \stackrel{\text{def}}{=} Server + (*\ \texttt{an OUT or EVAL not wanted}*)$$
$$\overline{\texttt{srp}}.Server' + (*\ \texttt{OUT satisfies non-local client}*)$$
$$\overline{\texttt{rp}}.Server' + (*\ \texttt{OUT satisfies local client}*)$$
$$(*\ \texttt{local IN or READ is satisfied}*)$$
$$\overline{\texttt{srq}}.Server(*\ \texttt{request cannot be satisfied locally}*)$$

$$Server'' \stackrel{\text{def}}{=} Server + (*\ \texttt{Non-local request cannot be satisfied (IN READ)}\ *)$$
$$(*\ \texttt{Non-Local ; OUT not wanted}*)$$

$$\overline{\texttt{rp}}.Server'' + (* \texttt{Non-Local OUT satisfies local client} *)$$
$$\overline{\texttt{srp}}.Server''(* \texttt{Non-Local OUT satisfies remote IN READ} *)$$

$$Server''' \stackrel{\text{def}}{=} Server + (* \texttt{client has been satisfied some other way} *)$$
$$\overline{\texttt{rp}}.Server'''(* \texttt{A reply has satsfied some clients} *)$$

A problem encountered when trying to model a system is the question of at what level of abstraction to model. If the model is too detailed proofs become longer and more difficult and the chances of the model containing errors and incorrectly modelling the system increases. If the model is too simple it may not be powerful enough to show the properties we want or have simplifying assumptions which cause the model to behave different from reality. We want enough abstraction to clearly show the operation of the server, but enough detail to convince ourselves that the operation of the model does indeed reflect something about the actual server. We can formally and rigorously prove any number of desirable properties about our model.

In our case we wish to model the message passing operation of the server and ignore the internal workings of the server nodes and the message content as much as possible. The above model is possibly too simple. The system we want to model consists of a few server nodes and some clients, which would be written as:

$$System \stackrel{\text{def}}{=} Client|Client|Client|(Server|Server\backslash[\texttt{srq}, \texttt{srp}])$$

for example. The type of response it expects from the client is too simple. It expects the client to behave like

$$Client \stackrel{\text{def}}{=} \overline{\texttt{rq}}.Client + \texttt{rp}.Client$$

An important point to note here it that the server only sends messages that it knows will not block. Messages which may block are given to a courier to deliver, which will eventually deliver them because it is delivering them to a server which will also not take the risk of blocking. However, when we model the server below we model the message passing between server as synchronous

communication, the two servers mutually decide to exchange a message, and they do so in co-operation. They cannot both decide to send messages to each other. This is not realistic because in reality two server could quite easily try simultaneously to send messages to each other and block in the process. The reason we do this is because we have taken the courier into account in our model.

Rather than add the courier to the model, we try to justify the use of the synchronous messages. Let us introduce our goal; two processes synchronously exchanging messages:

$$A \stackrel{\text{def}}{=} \mathtt{b}.A + \overline{\mathtt{a}}.A$$
$$B \stackrel{\text{def}}{=} \mathtt{a}.B + \overline{\mathtt{b}}.B$$

$$EasyComm \stackrel{\text{def}}{=} (A|B)\backslash[\mathtt{a},\mathtt{b}]$$

Now let us consider two processes communicating over a real communication medium. We introduce the notion of an 'ether' which accepts messages from anywhere, delivers them and sends an acknowledgement:

$$Ar \stackrel{\text{def}}{=} \overline{\mathtt{sa}}.\mathtt{ack_a}.Ar + \mathtt{ra}.\mathtt{ack_{ar}}.Ar$$
$$Br \stackrel{\text{def}}{=} \overline{\mathtt{sb}}.\mathtt{ack_b}.Br + \mathtt{rb}.\mathtt{ack_{br}}.Br$$

$$Ether \stackrel{\text{def}}{=} \mathtt{sa}.\overline{\mathtt{rb}}.\overline{\mathtt{ack_{br}}}.\overline{\mathtt{ack_a}}.Ether + \mathtt{sb}.\overline{\mathtt{ra}}.\overline{\mathtt{ack_{ar}}}.\overline{\mathtt{ack_b}}.Ether$$

$$RealComm \stackrel{\text{def}}{=} (Ar|Br|(Ether|Ether|Ether|Ether))\backslash [\mathtt{sa},\mathtt{sb},\mathtt{ra},\mathtt{rb},\mathtt{ack_a},\mathtt{ack_b},\mathtt{ack_{ar}},\mathtt{ack_{br}}]$$

We can see straight away that it is possible for both processes to try to send to each other and to deadlock. Now let us add a courier to each of the two processes:

$$Ac \stackrel{\text{def}}{=} \overline{\mathtt{sca}}.\mathtt{ack_{ac}}.Ac + \mathtt{ra}.\mathtt{ack_{ar}}.Ac$$
$$Bc \stackrel{\text{def}}{=} \overline{\mathtt{scb}}.\mathtt{ack_{bc}}.Bc + \mathtt{rb}.\mathtt{ack_{br}}.Bc$$

$$Ca \stackrel{\text{def}}{=} \mathtt{sca}.\overline{\mathtt{ack_{ac}}}.\overline{\mathtt{sa}}.\mathtt{ack_a}.Ca$$
$$Cb \stackrel{\text{def}}{=} \mathtt{scb}.\overline{\mathtt{ack_{bc}}}.\overline{\mathtt{sb}}.\mathtt{ack_b}.Ca$$

$$Ether \stackrel{\text{def}}{=} \mathtt{sa}.\overline{\mathtt{rb}}.\overline{\mathtt{ack_{br}}}.\overline{\mathtt{ack_a}}.Ether + \mathtt{sb}.\overline{\mathtt{ra}}.\overline{\mathtt{ack_{ar}}}outack_b.Ether$$

$$CourierComm \stackrel{\text{def}}{=} (((Ac|Ca)\backslash[sca, ack_{ac}])|((Bc|Cb)\backslash[scb, ack_{bc}])|(Ether|Ether|Ether|Ether))\backslash [\mathtt{sa}, \mathtt{sb}, \mathtt{ra}, \mathtt{rb}, \mathtt{ack_{ar}}, \mathtt{ack_{br}}, \mathtt{ack_a}, \mathtt{ack_b}]$$

This is becoming complicated, but we can still see that allowing courier processes to deliver messages on behalf of another process, we can avoid deadlock.

In order to avoid deadlock, we would like to take into account the different states a client could be in (blocked waiting for a reply or computing), but we want to avoid identifying individual clients and servers if possible because this would require a great deal of state information to be kept in the model, making it more difficult to work with. So instead we keep a count of the number of clients that are in which state, or more specifically, which the number of blocked clients. Because client are anonymous, it allows servers to be anonymous and so we do not have to specifically direct a message from one server to another and from one specific server to one specific client.

Can you do this and still have a realistic model? In our case we have no choice. The direction of messages to a specific destination is an intrinsic part of the working of the server, which is exactly what we are trying to avoid modelling. We are trying to model the message sending and receiving of a single server node, not a complete simulation of the whole system.

We write the revised model as follows:

$$Server \stackrel{\text{def}}{=} \mathtt{rq}.Server' + \overline{\mathtt{up}}.\mathtt{rq1}.Server' + \mathtt{srq}.Server'' + \mathtt{srp}.Server'''$$

$$Server' \stackrel{\text{def}}{=} Server + \overline{\mathtt{srp}}.Server' + \overline{\mathtt{down}}.\overline{\mathtt{rp}}.Server' + \overline{\mathtt{srq}}.Server$$

$$Server'' \stackrel{\text{def}}{=} Server + \overline{\texttt{down}}.\overline{\texttt{rp}}.Server'' + \overline{\texttt{srp}}.Server''$$

$$Server''' \stackrel{\text{def}}{=} Server + \overline{\texttt{down}}.\overline{\texttt{rp}}.Server'''$$

$$Counter(i) \stackrel{\text{def}}{=} \texttt{up}.Counter(i+1) + \mathbf{IF}\, i > 0 \,\mathbf{THEN}\, \texttt{down}.Counter(i-1)$$

$$Counter \stackrel{\text{def}}{=} Counter(0)$$

and we would express an example system as:

$$System \stackrel{\text{def}}{=} Client|Client|Client|Server|Server|Counter\backslash[\texttt{srq},\texttt{srp}]$$

The counter is not part of the real system. There is no process, exchanging messages with all server nodes, keeping track of the number of blocked clients. The counter represents the internal working of each server, where it keeps track of clients who are waiting for messages.

5.1.3 The Server model is Deadlock Free

We now show that the model of the server is deadlock free. We start off by showing that the distributed server (we use a system with 2 servers as an example) is bisimilar to a non-distributed server specified as follows, the three states representing the number of servers that have received messages and are dealing with them:

$$SServer \stackrel{\text{def}}{=} \texttt{rq}.SServer1 + \overline{\texttt{up}}.\texttt{rq1}.SServer1 +$$

$$SServer1 \stackrel{\text{def}}{=} SServer + \mathtt{rq}.SServer2 + \overline{\mathtt{up}}.\mathtt{rq1}.SServer2 + \overline{\mathtt{down}}.\overline{\mathtt{rp}}.SServer1$$

$$SServer2 \stackrel{\text{def}}{=} SServer1 + \overline{\mathtt{down}}.\overline{\mathtt{rp}}.SServer2$$

$$Counter(i) \stackrel{\text{def}}{=} \mathtt{up}.Counter(i+1) + IFi > 0THEN\,\mathtt{down}.Counter(i-1)$$

And we show that the system $SServer|Counter(0)$ is bisimilar to the system $Server|Server|Counter(0)$. To do so we show the relation **S** which is the required bisimulation. For each pair where *Server'* 's appear, add pairs for all possible combination of *Server' ,Server"* and *Server"'* in the place where *Server'* appears. Where a pair containing $Counter(i)$ appears, replace it with all pairs with i replaced with the all integers greater than 0:

$$\begin{array}{rl}
SServer|Counter(0), & Server|Server|Counter(0) \\
SServer|Counter(i), & Server|Server|Counter(i) \\
SServer1|Counter(0), & Server'|Server|Counter(0) \\
SServer1|Counter(i), & Server'|Server|Counter(i) \\
SServer1|Counter(0), & Server|Server'|Counter(0) \\
SServer1|Counter(i), & Server|Server'|Counter(i) \\
SServer2|Counter(0), & Server'|Server'|Counter(0) \\
SServer2|Counter(i), & Server'|Server'|Counter(i)
\end{array}$$

It is now a relatively simple matter to inspect the system : $Client|Client|SServer|Counter(0)$ (or with any other number of clients) and see the that it does not block where we do not expect it to.

5.2 Summary and Discussion

In this chapter we have seen how CCS can be used as an aid to understanding the working of the server. It proved to be a useful tool for thinking about how communicating systems operate. The value of CCS was not so much in the model itself as in the process of writing it down. There are a great many different models at various levels of abstraction and we have tried to model them in isolation rather than place them all in one large unmanageable model. We do not consider this a proof that the server is deadlock free, since we have no way of formally tying the model with reality. We can happily claim that the model is deadlock free though. There are some important aspects of the server which we have not modelled. We have not dealt with the courier in detail and how it effects the system in terms of efficiency or deadlock for example. What we have omitted is a realistic model of communication between servers plus a courier which is bisimilar to the `srq`, `srp` communication pairs.

Chapter 6

Related Research

In this chapter we describe some of the work being done on Linda at other institutions.

6.1 Other Linda Implementations

6.1.1 POSYBL

POSYBL (a PrOgramming SYstem for distriButed appLications) [37] is an implementation of the Linda model for Unix networks. It is built on top of Unix sockets and remote procedure call conventions and is made up of processes called Tuple Managers. There is one Tuple Manager per processor. User processes use remote procedure calls to the local Tuple Manager to read, write or delete tuples. If the request cannot be satisfied by the local Tuple Manager, it is broadcast to all the Tuple Managers. Up to 25 user processes can run together on each node.

POSYBL differs from the original Linda model in a few ways. It does not have a preprocessor to determine the size and type of the fields in a Linda call and so the user has to provide this with simple field description functions. There is also no way to determine which tuple field to use as a hash key and so they have adopted the convention that the first field of a tuple must be an actual. The EVAL operation is different in that rather than specifying a function to call, an executable file to be invoked is specified. The READP and INP functions have not been implemented because their cost would be prohibitive due to the nature of the implementation.

6.1.2 NUE-Linda

Murakami et.al. [29] have defined a Tuple Operation Protocol Suite (TOPS) for the NUE-Linda computation model. It was designed for interoperability between many kinds of processors. TOPS is a four layered protocol on top of UDP/IP. It provides functions such as a bi-directional remote procedure call facility by piggyback operation, embedded flow control, multiple tuple-space operations and multiple presentation protocols. UDP was chosen as a transport because of its low overhead and high-speed processing. The four protocol layers are described briefly starting with the lowest:

Bi-directional RPC

Bi-RPC is based on the Stop-and-Wait method. An agent sends a request packet and waits for an acknowledgement. If a timeout occurs, the packet is resent. Acknowledgements may be piggybacked on packets arriving from the destination.

Tuple Operation Protocol

TOP provides basic tuple operation facilities such as `in` and `out`. It is designed to be independant of the Bi-RPC protocol and may utilize other transport layer primitives like Sun RPC for example.

Tuple Space Management Protocol

The task of the tuple space management protocol is to provide the same view of a clustered multiple tuple space as on a single tuple space. It coordinates the replication of tuples and deleting copies of tuples, and locating tuples.

Multiple Presentation Protocols

The final protocol level allows multiple implementations to co-exist, each with its own coding rules or data formats. As an example, their implementation of NUE-Linda on the TAO/ELIS system

utilizes both its proprietary coding rule and ASN.1 of OSI. The multiple presentation protocol scheme provides data transparency and interoperability between different implementations.

6.1.3 Prolog-D-Linda

Prolog-D-Linda [39] is an embedding of Linda in SICStus Prolog. It has a distributed tuple space and uses unification and Prolog style deduction in the tuple space. Prolog-D-Linda implements tuple space as a collection of Prolog clauses in the Prolog database. Both Prolog rules and facts can exist in the tuple space. The OUT operation adds tuples using Prolog's `assertz` database operation and the IN removes tuples using `retract`. The READ operation interrogates the tuple space using the Prolog query mechanism. Tuple matching is thus generalised to unification. The EVAL operation differs from the original Linda paradigm. It is used to start a new Prolog environment containing specified clauses and evaluating a specified Prolog query. The evaluation of the query may cause tuples to be inserted into tuple space.

The system runs on Sun SPARC workstations running SunOS 4.1.1 and uses Suns network file system. It consists of controller process used to start the server processes and manage terminal IO and a number server processes each controlling a part of tuple space. One server is designated as an eval-server which starts new clients on the various machines. Communication between clients and server and between servers is via internet domain stream sockets.

Linda operations in the client are translated into requests which are passed to the appropriate server. The server is determined by consulting a configuration program which takes a tuple and returns the correct server. The requests are serviced by the server by evaluating them as Prolog queries and are thus simply queries on prolog procedures which implement the required operations.

Having rules in tuple space presents some problems. If a deductive database is to be used, it is neccesary to have all the required rules stored in the same server. Deductive rules must be evaluated by the server and so prevents the server from servicing other requests while it is computing. A more serious problem is that the server may enter infinite deduction, stopping all other clients. A proposed solution to this, however, is to restrict the deductive database to be hierachical. A deductive database greatly enhances the power of the system and so it is worthwhile pursuing solutions to these problems.

6.1.4 The Linda Program Builder

The Linda program builder [6] is a development environment for Linda programs. It is a structured editor that builds Linda programs under the user's supervision. It contains templates which the user fills in according to his needs. It speeds up program development by providing a significant part of the application framework. Because the application is built in this way, the Program Builder has valuable high level information about the application available for optimisations which would be impossible for a pre-processor to obtain. Because it knows what the programmer wants to do, it is able to fuse and re-arrange tuple operations, to provide better performance.

The user has high level structures at his disposal but the final program produced by the program builder is still a Linda program with its simple semantics, avoiding a very high level language with its complex structure, and over-specific features. The final program retains all the portability of a Linda program, able to run with no modification on all platforms with Linda systems.

6.1.5 Piranha

Piranha [6] is a project at YALE aimed at utilising the idle time of workstations on their local area network. A Piranha program is a Linda program structured as a bag or ordered collection of tasks. These tasks are used to perform the computation, the more tasks running, the faster the computation is completed. Workstations are configured to join the computation when they become idle, taking tasks to execute. They leave the computation when their user needs them (notifying this by either typing on the keyboard or putting load on the cpu).

The Linda Program builder supports the creation of Piranha programs. The framework it presents consists of three functions : *feeder, piranha* and *retreat.* A Piranha program does not use the EVAL operation to create tasks but instead creates a varying pool of processes automatically, each running the *piranha* function. Initially one *piranha* process is created for each idle workstation, when new workstations are added , new *piranha* processes are created. The *retreat* function is called when the workstation is needed and it makes a note in tuple space of any unfinished work. The *feeder* function initialized, coordinates and finalizes the computation.

6.2 New Preprocessor Optimizations

An important area of research in the development of Linda systems is that of preprocessor optimization. Carreiro et.al. [13] have classified a range of optimizations which can be applied to Linda programs based on new analyses.

Generalized tuples concentrate on tuples which are related by a common consumption or production site, attempting for example to merge a series of Linda operations into fewer operations. For example a common operation to apply a function to a shared variable e.g:

```
IN("counter", ?i);
OUT("counter", f(i));
```

can be replaced by the preprocessor by a call to update the "counter" in a single function e.g:

```
i= update_f("counter");
```

More sophisticated analysis is required make this possible in general. The state information that `f` accesses is needed, as well as information on how computationally intensive `f` is. Some special cases for example incrementing a shared variable are possible however since the state information it needs is only the value `i` and it is not computationally intensive.

Field sub-context analysis attempts to relate tuples by values shared in their fields. The following example was presented as an illustration:

```
MASTER:
for (i=0; i < N; ++i) OUT("worker_info", i, N);
for (i=0; i < dim; ++i) OUT("col", i, col_vecs[i]);

WORKER:
IN("worker_info", ?id; ?N);      /* Who am I? */

for (i = id; i < dim; i += N) /* Collect my columns. */
        IN("col", i, ? local_vecs[local_index++]);
```

The influence of `id` and `N` can be traced and allows us to determine the column vectors which are bound for a particular worker. The relevant "col" tuples can then be generalized into a single tuple.

EVAL context attempts to determine the number of instances of an operation which are running concurrently, possibly allowing broadcast mechanisms to distribute data to nodes, while *Task characterization* attempts to quantify the nature of a process that is to be EVAL'ed, trying to identify the local and global data that it references, and its run time. This gives us better information to be able to do better process placement.

6.3 Summary

We have presented some of the work that is being done on Linda, much of which presents ideas for future work on this server. The utilization of idle workstations by Piranha leads us to investigate how our server can be used to utilize idle time in this way. The deductive tuple space in Prolog-D-Linda is a powerful, but difficult enhancement to the Linda paradigm. The Linda Program Builder is a valuable environment for developing Linda programs and providing high level information to the pre-processor and work on preprocessor optimization is an important and potentially fruitful area of research.

Chapter 7

Enhancements and Future work

This work does not cover all aspects of the server. It assumes the existence of a preprocessor which performs important tuple space analysis and syntax transformation. Process and tuple placement are not dealt with in detail and not all of the implementation for the various target architectures have been completed. We describe here some of the enhancements to the current server that are planned.

7.1 Other Transport Mechanisms

The server currently uses TCP/IP streams as the base transport mechanism because of its generality and it's availability on the target platforms, but there are cases where another mechanism may be better. The server has been designed to allow other mechanisms to be used in the specific cases where there is a mechanism better than TCP/IP

Shared Memory

Communicating processes on the same machine should achieve much better communication if they used shared memory as a transport rather than TCP/IP. A client and its server is usually on the same machine and so this is an ideal place to use shared memory.

Helios Raw Links

For small messages, dedicated transputer links achieve significantly faster communication than the built-in Helios functions. Only some of the links can be taken over since care must be taken to leave Helios with enough connectivity to run. Experiments with an older version of Helios were problematic. Removing transputer links from Helios made an already unstable operating system even less stable. The current of Helios version appears to be more stable and once the work on the server under Helios is complete, a second attempt at utilizing dedicated links may be worthwhile.

UDP Datagrams

It may be worthwhile investigating whether using UDP datagrams are better than TCP/IP. The current protocol assumes reliable streams. Rather than changing the existing tuple protocol, another layer should be added to deal with packetization, error checking and retransmission, giving the appearance of a stream.

7.2 A Better Eval

Two ideas present themselves as ways to improve the performance of the EVAL operation. The first is to keep dynamic load information and place new tasks according to the current load information and the tuple partitions that the EVAL task will use. This can be used to develop a better heuristic for task placement than the current one.

The second is to improve the process creation time for an eval task by making use of an EVAL daemon using a light-weight process library. The library will be created by the preprocessor at run-time and will run on processors prepared to run EVAL tasks. Instead of starting a new process using fork() and running an executable, the daemon will start a new thread to perform the EVAL computation. It is important however to make this daemon optional since light-weight process libraries are only available for some of the target platforms.

EVAL processes are spawned the first time a particular tuple needs to be evaluated and run indefinitely, waiting for new requests. Is has been noted that the system will eventually be running many EVAL processes, only some of which will be doing useful work. Provision must be made for

deleting idle EVAL processes.

7.3 A Different Server Implementation, Working with the Existing Implementation

A multi-threaded server node may well prove to be more efficient than the single threaded, transaction server in existence. This server runs on a number of different platforms and presents a relatively simple interface to the other server nodes. The interface it presents is well defined and it should be possible to write a multi-threaded server which conforms to this interface, on machines where an efficient threads implementation exists.

We may consider the possibility of a server node, seen by the rest the system as a single node, presenting the standard interface, but in reality it runs on a shared memory multi-processor, for example. It remains to be seen whether it practical to look at things this way and whether efficient applications can be written on such a system.

7.4 Analysing the Message Passing Semantics of the Linda Server

A more detailed modelling of the message passing of the Linda Server may be illuminating. The current model treats the underlying network as a richly connected cluster as presented by the Linda transport layer. Taking the real network topology into account, with the real network latencies and bandwidth could reveal interesting insights into tuple placement and distribution.

7.5 Persistence in Linda Programs

The nature of the server is such that extra work is required to remove the tuples of a program after it has finished execution. Making use of tuple space as a persistent store will add important functionality to the server. It will allow programs to use Linda as a data store, allowing programs to store data in a form which is closer to their internal structure, thus reducing the overhead of file handling. Some work is required to realize this. Firstly applications must be protected from each other since it is not desirable to have a program in one application accessing the data of another. To

do this we will add an application identifier to a Linda programs, either at a tuple partition level or at a tuple level. Another important reason for this separation is that we need to know about all the programs that are part of application in order to perform the preprocessor analysis, which decides on tuple partitioning and special cases based on the whole picture. We cannot subsequently add to the application without re-analysing the system. We will also have to provide features for data protection and integrity, since a user will not trust valuable data to the memory of a continually running Linda daemon.

We also need to allow the user to specify what data she would like to continue to exist after a particular execution has completed.

7.6 Preprocessor Optimizations

Currently all tuple partitions are stored in the same way. If our preprocessor is able to provide information about the nature a particular partition we will be able to use better storage methods in certain cases. For example the preprocessor may inform us about which fields can be used as a hash key or let us know that the partition need only be stored as a simple semaphore [24].

Some of the new optimization strategies [5] described in the previous chapter look promising. Realizing them would require significant analysis by the preprocessor.

The system does not handle structured data types, since we decided that this was best handled by a preprocessor, breaking down structured types into their components. This is a useful enhancement to the system and will be considered sometime in the future

7.7 Additional Tuple Space Functionality

We can add to the usability of the system by adding a number of administrative functions.

Operation on Whole Tuple Groups

Hupfer [28] extends the functionality of the Linda model by allowing the Linda operations to operate on whole tuple space partitions rather than simple on tuples, creating a "MetaLinda", where whole tuple spaces can be added and removed from a tuple space. Some of this functionality would add to the usability of the current system.

Some additional functions might be:

- **tuple group migration**, taking a whole tuple group and placing it on another processor, possibly closer to the programs using the tuple group,
- **server suspend and resume** allowing us to run the server on a user's workstation, using the processor only when it is idle, the ability to dump an entire suspended server to disk might be desirable, since some of our workstations will be set up to run either DOS or Unix, necessitating the shutdown of the server daemon.
- **tuple group dump and restore**, allowing tuple groups to be dumped to disk, either for backup purposes or to allow work to be continued at a later date or
- **statistics** on various aspects of the server, the amount of memory used by a tuple group, the number of tuples it contains or the number of operations performed on the tuple group.

Client Reconnect

We currently expect a client and its server to run on the same machine, but there is nothing stopping us from allowing a server to accept a client from another machine. This is desirable because a the number of messages sent per operation can be reduced if a client connects to the processor on which the tuple space it uses resides. Messages to clients on other machines will have to be given to the courier for delivery. Allowing connections from other machines permits the use of processors running DOS since there is only one 'process' allowed under DOS. These processors could either run a client or be a server node. A problem with DOS, however, is that we have to start all processes on all DOS machines manually when we wish to run a program. This limits the ease of use of the system.

Chapter 8

Conclusion

8.1 The Goal and Requirements revisited

We consider once more the goals and requirements we had at the beginning of the project to see how we have fared:

- **Use of Heterogeneous processors:** The server does not rely on a particular architecture. The initial implementation was done on a network of Sun SPARC processors running SunOS 4.1.1 because of the good development environment this operating system provides. It was then ported to a 68030 processor running SunOS to eliminate any architecture dependencies that crept in. We have just completed a port to an 80306 processor running 386BSD Unix to eliminate some of the dependencies made on SunOS features. The original development platform, a cluster of transputers running Helios was abandoned as a development environment at an early stage because of its instability. We have recently acquired a later version which is much more stable. There is also now a TCP streams package for this system. We are now almost ready to run the server on this system. Work on allowing clients to run on DOS machines is planned. The number of architectures on which the server runs is a fair indication that it does not rely on a particular machine architecture.

- **Interconnectivity:** TCP was designed to provide interconnectivity and network hiding and so provides a good default communication mechanism. The server does not rely on TCP though and was designed to allow other communication mechanisms to be used.

- **Reliability:** The server is robust with respect to client failure. It can detect if a client has died and will clear the connection. If a server node goes down the other servers will detect it and not use it in future. There is a problem when a client or server dies though because programs which are currently using the client or have part of their tuple space on the server node will in all likelihood fail. We do not keep 'backup' tuple spaces on other servers and we have no way of recovering a program whose client has terminated abnormally. So a Linda program running on this server is not as robust as a regular executable because the chance of some external reason for failure increases with the number of processors used by the program.

- **Portability:** The server currently runs on a number of Unix-like systems and porting the system to new platforms while it was under development proved to be relatively straightforward. Running the system on new Unix-like platforms should be much simpler. The server is written in Ansi-C which is available on a larger number of platforms. Running the server under other operating systems (e.g. VMS) might require us to implement a transport layer using TCP/UDP. We are satisfied with the portability of the system as it allows us to make use of almost all our computing hardware.

- **Efficiency:** The server attempts to make good use of the resources we have available. The server makes fair use of the available memory, keeping only one copy of a tuple and both the server daemon and the client library are relatively small. We reduce the amount of message passing needed by designing the server in such a way as to avoid the necessity of broadcast messages. The number of messages Linda operation is not optimal, with six messages sometimes required to complete a transaction. Two of these are messages to the courier which are relatively fast. We spread the processing load among the processors by allowing tuple partitions to exist on any server.

- **Ease of Use:** The server is relatively simple to use. The server daemon needs to be run on every node and client programs need to be distributed but this is relatively straightforward using the tools on the Unix operating system. The server daemons need only be started once and this task can be done by the system administrator.

We have developed a distributed Linda server which runs on a heterogeneous network of processors, and have designed a communication protocol for tuple exchange and server operation. It hides the details of the server implementation and architecture dependencies of the host processors. TCP/IP was used as a transport mechanism because of its generality but allows other transport mechanisms to be used where neccesary. We avoid the bottleneck of using a single node to hold tuples by allowing

tuple space partitions to be stored on any server node. Message passing is reduced by designing the server so that broadcast messages are not neccesary. The structure of the server is such that it does not make use of platform specific libraries such as a light weight process library and so is easily portable to new platforms. We made use of Milner's process calculus to model various aspects of the message passing semantics of the server. It proved to be a useful tool in understanding the operation of the server.

Linda is a simple and easy to understand parallel programming paradigm. It is able to make good use of the capabilities of the hardware given a suitable Linda implementation. The goal is to have an implementation which will work well for all applications. This work was purposefully optimistic in its design and has wide scope for improvement and future development and will hopefully serve as a basis for future work. We currently do not see the need to start again from scratch when new developments arise. The server takes into account the various implementation strategies which can be used and the realities of the available hardware and tools and makes good use of our current hardware.

Appendix 1

We present here a short tour of the system as it now stands. We show the format of the configuration file, how the system is started. We show a sample execution using an eight queens program and a 'ping-pong' program.

8.1.1 The configuration file

```
# Linda server node descriptor file        G.L.S.  17/02/92
#
# Format:  1 line for each node, in increasing order of id
#
# Line Format: nick:id:address:class:machine_type
#         nick matches gethostname();
#         id: integer server identifier
#    class: s slave, m master
#     lines beginning with # are comments as are lines with
#    insufficient colons.
alpha:10:alpha.ru.ac.za:m:BSD
beta:20:beta.ru.ac.za:s:BSD
clayton:30:clayton.ru.ac.za:s:BSD
gamma:40:gamma.ru.ac.za:s:BSD
delta:50:delta.ru.ac.za:s:BSD
#Hyaena:60:hyaena.ru.ac.za:s:BSD
#hippo:60:hippo.ru.ac.za:s:BSD
#cc190.ru.ac.za:70:cc190.ru.ac.za:s:BSD
```

```
#braae.ru.ac.za:70:braae.ru.ac.za:s:BSD
#
```

8.1.2 Start-up

The server is started by starting up a daemon on each node of the system which can be done by a shell script similar to te following:

```
rsh alpha   -l g91s9798 'cd 1; rl'&
rsh beta    -l g91s9798 'cd 1; rl'&
rsh gamma   -l g91s9798 'cd 1; rl'&
rsh delta   -l g91s9798 'cd 1; rl'&
rsh clayton -l g91s9798 'cd 1; rl'&
```

8.1.3 The PingPong Example

We run the pingpong program on **alpha** and a tuple partition is created which resides on **alpha**. The program simply stores and then removes tuples from tuple space and gives a rough idea of the number of operations that can be achieved per second. The body of the program after processing looks as follows:

```
void pingpong()
{ int i,b,e;
  tg1 = OpenTS("test10","i",M_Passive,1024,malloc);
  printf("begin\n");
  b=clock();
  for (i=0;i<100; i++)
  {
       OUT(tg1,~0,1);
       IN(tg1,~0,1);
  }
  e=clock();
  printf("end %f\n",200/(((float)(e-b))/(1000)));
```

```
  printf("begin\n");
  b=clock();
  for (i=0;i<100; i++)
  {
       OUT(tg1,~0,1);
  }
  for (i=0;i<100; i++)
  {
       IN(tg1,~0,1);
  }
  e=clock();
  printf("end %f\n",200/(((float)(e-b))/(1000)));
  sleep(200);
  CloseTS(tg1);
/*  DeleteTS(tg1);*/
}
```

It produces the following output. The results are in operations per second:

```
alpha:/home/g91s9798/1> pingpong
begin
end 351.493835
begin
end 440.528625
alpha:/home/g91s9798/1>
```

This is reasonable since simple message passing over TCP/IP streams can achieve around 1400 message transfers per second and Linda operations usually require two messages per operation.

When we run the program from **beta**, the tuple partition that the program uses already exists and resides on **alpha** forcing the server to move messages across the network. We get the following results:

```
beta:/home/g91s9798/1>pingpong
```

```
begin
end 10.004502
begin
end 9.913751
```

This is quite disappointing even considering there is now a network involved.

As an experiment we modify the client library so that clients always connect to a specified host, rather than their local server, eliminating the intermediate server but still using the network. We get these results:

```
beta:/home/g91s9798/l>pingpong
begin
end 402.414490
begin
end 431.965454
beta:/home/g91s9798/l>
```

Which is a lot better. This leads us to believe that implementing the mRECONNECT command is not only a desirable extension to the server, but virtually essential. These results are not unrealistic, the results of the pingpong program vary with about 30 messages per second every time it is run and the number of small messages per second over TCP/IP is roughly the same regardless of whether the processes participating are on the same machine or the same ethernet segment.

8.1.4 The Queens Example

Now let us try the queens problem. It is taken from the test suite for the Transputer Linda implementation. It is a master/worker model. Note how replies from the worker have been batched to try and reduce the impact of high network latencies:

```
/*
 *  Queens: Boss process
 */
.
```

```
/*Code omitted*/
.
int issafe(int q, int col, int dist)
/*********************************/
/* Is q safe with respect to column col of the board?  Q is dist away
   from the place where q is about to be placed.  If it is safe,
   try column col-1.
*/
{/*Code omitted*/}

int placeQueens(int num_placed)
/*****************************/
{ int q, pos;

  for (q=0; q < boardSize; q++)
    if (issafe(q,num_placed-1,1))
    {  pos = num_placed;
       board[pos++] = q;
       if (pos < bossLimit)
           placeQueens(pos);
       else
           OUT(partials,~0,boardSize,pos*sizeof(int),board);
    }
}

int solutions [] =
  { 0, 1, 0, 0, 2, 10, 4, 40, 92, 352, 724, 2680, 14200, 73712, 365596, 0, 0,
    0, 0, 0, 0 };

int main(void)
/************/
{ int q, count,i,j;
  partials  = OpenTS("partials","ib",M_Passive,128,malloc);
  completed = OpenTS("completed","b",M_Passive,128,malloc);
```

```
/*Code omitted*/

  count = 0;
  do
  {  q = sizeof(boardbuf);
     IN(completed,0,&q,boardbuf);
     for(i=0;i<q/(boardSize*sizeof(int));i++)
            printboard(++count,&boardbuf[i*boardSize]);
  }
  while (count < solutions[boardSize]);
  if(count!=solutions[boardSize])printf("error: too many solutions\n");
  printf("\nTotal number of solutions = %d ", count);
  if(INP(completed,0,&q,boardbuf))printf("error: too many solutions\n");
  CloseTS(partials);
  CloseTS(completed);
  CloseTS(evalGroup);
  return(0);
}

/*
 *  Eight Queens ... Worker
 */

/*Code omitted*/

void placeQueens(int num_placed)
/*****************************/
{ int q, pos,i;

  for (q=0; q < boardSize; q++)
    if (issafe(q,num_placed-1,1))
    {
       pos = num_placed;
```

```
        board[pos++] = q;
        if (pos < boardSize)
            placeQueens(pos);
        else {
           if(curbuf<BBUFSIZ){
               bcopy(board,&boardbuf[curbuf++*boardSize],boardSize*sizeof(int));
           }else{
               OUT(completed,~0,pos*sizeof(int)*curbuf,boardbuf);
               curbuf=0;
           }
        }
    }
}

int main(void)
/*************/
{
  completed = OpenTS("completed","b",M_Passive,128,malloc);
  evalGroup = OpenTS("w_queens","",M_Passive,128,malloc);
  while (1) {
     j = sizeof(board);
     IN(partials,0,&boardSize,&j,board);
     num_placed = j / sizeof(int);
     placeQueens(num_placed);
     if(curbuf){
               OUT(completed,~0,boardSize*sizeof(int)*curbuf,boardbuf);
               curbuf=0;
     }
  }
}
```

The serial version of this program can solve the 10 queens problem in about a second. We do not expect to better this with all the overheads of Linda, but we are able to keep up. Three worker processors connected directly to the node holding their partition plus one processor for the master process, are able to compute the 720 solutions in about one second. For 11 queens the serial version

takes about 8 seconds. The same Linda setup as before takes 3 seconds to complete. The serial version takes 54 seconds to calculate the 14200 solutions to the 12 queens problem, while the Linda version takes 16 seconds. Using 5 machines with workers on all of them we complete the 12 queens problem in 8.5 seconds.

Although this is not a proper benchmark, it does seem indicate that the server is a viable product with its own distinct advantages.

Bibliography

[1] S. Ahuja, N. Carreiro, and D. Gelernter. Domestic parallelism - linda and friends. *Computer (USA)*, 19(8):26–34, Aug. 1986.

[2] G. R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–91, Mar. 1991.

[3] R. Bjornson. Experience with linda on the ipsc/2. Technical Report YALEU/DCS/RR-698, YALE, Mar. 1989.

[4] R. Bjornson, N. Careiro, D. Gelernter, and J. Leichter. Linda, the portable parallel. Technical Report DCS/RR-520, YALE, Feb. 1988.

[5] R. Bjornson, N. Carreiro, and D. Gelernter. The implementation and performance of hypercube linda. Technical report, YALE, Mar. 1989.

[6] R. Bjornson, D. Gelernter, T. Mattson, D. Kaminsky, and A. Sherman. Experience with linda. Technical Report DCS/TR-866, YALE, Aug. 1991.

[7] B. H. Boar. *Implementing Client/Server Computing - a Strategic Objective.* McGraw Hill Inc., 1992.

[8] G. Bruns. A language for value-passing ccs. Technical report, Sept. 1992.

[9] N. Carreiro and D. Gelernter. The s/net's linda kernel. *Operating Systems Review,*, 19(5):54–71, Mar. 1895.

[10] N. Carreiro and D. Gelernter. Tuple analysis and partial evaluation strategies in a linda precompiler. Technical report, YALE, May 1989.

[11] N. Carreiro, D. Gelernter, and J. Leichter. Distributed data structures in linda. In *13'th Symposium on Principles of Programming Languages*, volume 13, pages 236–242, Jan. 1986.

[12] N. Carriero and D. Gelernter. Linda in context. *Comminucations of the ACM*, 32(4):323–356, Apr. 1989.

[13] N. Carriero and D. Gelernter. New optimization strategies for the linda precompiler. In Wilson [30], chapter 5, pages 74–84.

[14] P. Clayton. An overview of parallel programming paradigms for sharing information in distributed systems. In *Concurrent Computing 89, Pretoria*, Nov. 1989.

[15] P. Clayton, E. P. Wentworth, G. C. Wells, and F. de Heer Menlah. An implementation of linda tuple space under the helios operating system. *SA Computer Journal*, pages 3–10, Mar. 1992.

[16] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP Volume II.* Prentice Hall, 1991.

[17] I. Davies. *The Helios Parallel Operating System.* Prentice Hall, 1991.

[18] F. K. de Heer-Menlah. Analyzing communication flow and process placement in linda programs on transputers. Master's thesis, Rhodes University, Oct. 1991.

[19] T. Durham. Communicating linda's message to the world. *Computing*, pages 32–33, Oct. 1989.

[20] N. Gehani. *Concurrent Programming.* ATandT Bell Laboratories, 1988.

[21] D. Gelernter. Generative communication in linda. *ACM TOPLAS*, 7(1):80–112, 1985.

[22] D. Gelernter. Getting the job done. *Byte*, 13(12):301–308, Nov. 1988.

[23] D. Gelernter. The metamorphosis of information management. *Scientific American*, pages 54–61, Aug. 1989.

[24] N. C. D. Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3):323–359, Sept. 1989.

[25] W. M. Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. *Software Practice and Experience*, 11(5):435–466, 1981.

[26] F. Halsal. *Data Communication, Computer Networks and OSI.* Addison Wesley, 2 edition, 1988.

[27] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[28] S. C. Hupfer. Melinda: Linda with multiple tuple spaces. Technical Report DCS/RR-766, YALE, Feb. 1990.

[29] K. ichiro Murakami, Y. Amagai, O. Akashi, and H. G. Okuno. Tops: A tuple operation protocol suite for nue-linda computation model. In *Proc. of JWCC-6*, July 1991.

[30] Linda-like systems and their implementations. Edinburgh Parallel Computing Centre, June 1991.

[31] K. K. Jensen. The semantics of tuple space and correctness of an implementation. Technical report, YALE, jensen-keld@cs.yale.edu, 1990.

[32] T. J. LeBlanc and E. P. Markatos. Shared memory vs. message passing in shared-memory multiprocessors. Technical report, University of Rochester, Apr. 1992.

[33] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. *ACM SIGPLAN Notices*, pages 276–284, Sept. 1988.

[34] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[35] J. Powell and N. Garnett. Helios performance measurement. In *Helios Technical Reports*, chapter 22. Distributed Software Limited, Feb. 1990.

[36] S. R. Saunders. The specification and design of a nondeterministic data structure using ccs. Technical report, Division of Computer Science, Hatfield Polytechnic, June 1988.

[37] G. Schoinas. Posybl: Implementing the blackboard model in a distributed memory environment using linda. In Wilson [30], chapter 8, pages 105–117.

[38] M. Sloman and J. Kramer. *Distributed Systems and Computer Networks.* Prentice Hall International, 1988.

[39] G. Sutcliff and J. Pinakis. Prolog-d-linda: An embedding of linda in sicstus prolog. Technical Report 91/7, The University of Western Australia, geoff@cs.uwa.oz.au, 1991.

[40] A. S. Tannenbaum. *Computer Networks.* Prentice Hall International Inc., 2 edition, 1988.

[41] J. H. Tonsig. A linda implementation for transputers. Submitted in partial fulfillment of the requirements for the degree b.ing. (elektron), University Of Pretoria, Nov. 1989.

[42] D. Walker. Introduction to a calculus of communicating systems. Department of Computer Science, University of Edinburgh, Feb. 1987.

[43] E. Wentworth. Parallelism via linda - a transputer implementation, status report. Technical report, Rhodes University, cspw@alpha.ru.ac.za, June 1990.

[44] P. Wentworth. Prototyping a linda system. In *Concurrent Computing 89, Pretoria*, Nov. 1989.

[45] R. A. Whiteside and J. S. Leichter. Using linda for supercomputing on a local area network. Technical Report DCS/TR-638, YALE, June 1988.

[46] S. E. Zenith. Linda coordination language; subsystem kernel architecture (on transputers). Technical Report DCS/RR-794, YALE, 1990.