1.0 Introduction

The remote procedure call (RPC) and Linda paradigms are two different approaches to distributed computing. Both attempt to provide the power of distributed computation by using clean and simple methods. RPC uses procedure call and return semantics familiar to most programmers. Linda, in the original form, extends a programming language by adding three calls that interact with a conceptually shared data space.

The creators of Linda (Carriero & Gelernter 86) claim that Linda could be built on top of RPC, but that it would be inefficient. The results of this implementation support the authors claim, and provide some insights into shortcomings of RPC.

This paper will follow the development of R–Linda, starting with a definition of the problem that R–Linda attempts to solve. Next is a general description, first of RPC, and then Linda. Following this the environment and the RPC tools used for the implementation are covered. The main part of the paper discusses R–Linda itself, including a functional breakdown of the system. Some comments on the performance of the system are given in the concluding section. Lastly, there is an appendix containing performance measurements, and how they were made. The source code is available on request from the author.

Before continuing, it is necessary to define some terms that will be used in the paper. They are not based on any paper or set of papers. It should be noted that these definitions may conflict with those used in portions of the literature.

The term *parallel computing* refers to a program that concurrently utilizes more than one process and/or processor to arrive at a solution. It is not required that the program control the allocation of processes and/or processors. A *node* is a single processor. This may refer to a machine that contains multiple processors, though in this paper it refers to a single processor workstation on a heterogeneous network of computers. A computation is *distributed* if it runs concurrently on more than one node. A *heterogeneous network* is one that contains nodes of more than one machine architecture.

Using these definitions, *heterogeneous distributed processing* (or computing) refers to programs that concurrently use the processing power of nodes with different architectures on a network to arrive at a solution. In this case, processing and computing are interchangeable terms.

The *local node* is the resident node of an executing process. A *remote node* is any other node in a possibly heterogeneous, network. A *server* is a

process that performs processing (services) requested by possibly remote programs. The program requesting the service is called a *client*. It is possible for a server to be a client of other servers. If the other server is identical, it is said to be a *replicated server*.

2.0 The Problem

The goal of this project is to produce a working implementation of the Linda paradigm, which meets the following criteria :

1 - The system utilizes RPC for all Linda related interprocess communication.
2 - The Linda call semantics conform to (Carriero & Gelernter 86).
3 - Data passed in RPC calls should be minimized.

The first criterion assures that any limitations or enhancements possible with RPC will be used. For example some tasks, such as interprocess communication on the same node, are faster when RPC is not used. This prevents the use of RPC as a connection negotiator between two processes (one could use NCS to negotiate a socket between two processes).

The next criterion serves two purposes. First it reflects the state of Linda at the time of the authors claim regarding RPC. It also constrains the problem to implementing three calls with well specified semantics.

The final criterion is due to the nature of RPC calls. Any data transferred between client and server must be both copied and transmitted. Though there are ways to save time, such as not copying data on a return from a call that is specified as in data only, there is no guarantee that a particular RPC package implements them. If the amount of data is minimized, it lessens the dependance on the RPC implementation, and speeds up the call.

3.0 Remote Procedure Call (RPC)

The remote procedure call (RPC) paradigm is an extension of normal procedure call semantics. In a traditional procedure call, the currently executing routine makes a call to a procedure, and suspends (or blocks) until the subroutine has completed (Figure 1- over). There may be a transference of data involved in one or both directions.

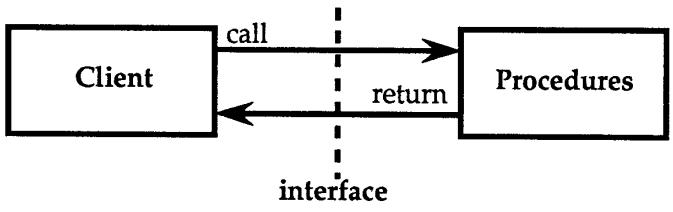

Figure 1 : Traditional Call Flow

In a remote procedure call, one process, the client, makes a procedure call to another process, the server (Birrell & Nelson 84). Unlike a regular call, the client and server may be on different machines. As with regular procedure calls, the client blocks until the call is complete. The difference is that the client must locate a server, send the data to, and receive the data from, the server. The server must process the data, and send it back to the client. An RPC package provides tools and code to support these activities.

Since one objective of RPC is to be similar to normal procedure call semantics, the client should not have to do anything special to make the remote call. For this reason, RPC packages provide facilities to generate the code that handles locating the server and converting the data to and from transmissible form (this is called marshalling and unmarshalling). The code to do this is called the *client stub*. There is also a *server stub* that provides the same functions for the server. The communication is provided by a runtime module that transmits the data. To the client program, the procedure call appears to be a traditional one, though there is actually more work being done (Figure 2).

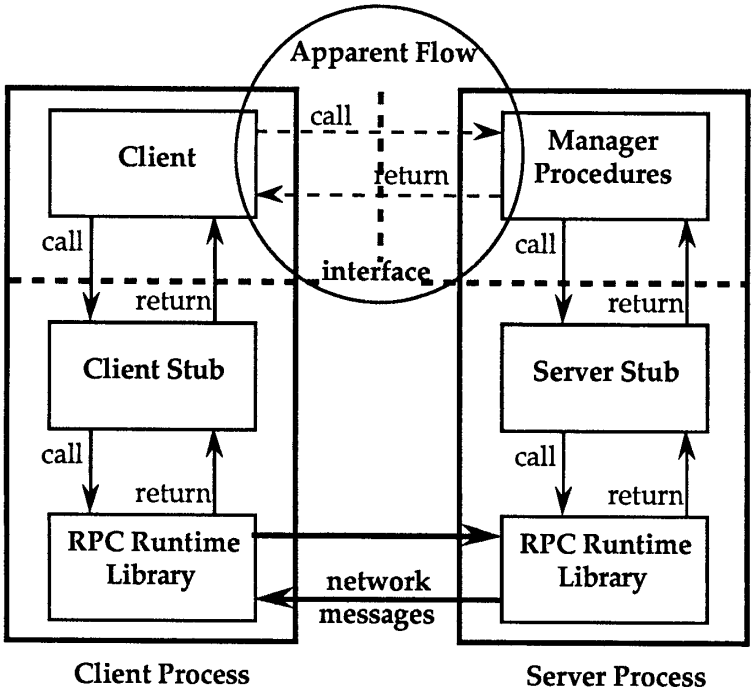

Figure 2 : Remote Procedure Call Flow

It is also possible to communicate between differing architectures with differing data representations such as ASCII and EBCDIC. Either the generated stubs or the RPC runtime library will convert between the formats. In order to do this, the types of data must be known in advance. This is done by making an *interface definition*, which declarativly states what calls will be made. This definition is used to create the stubs used by the client and server.

There must be some form of naming resolution, so that a client can find a particular server. This is usually done by an RPC runtime facility. The server registers with the RPC runtime library, and when a call is made by a client for a service (or interface), the runtime package finds if and where that service is offered. This is called *binding* to a server.

An important point to note is that the server does not know how to contact the client. This is analogous to having no scope in a procedure call, that is a procedure can not access variables known to the caller. Another implication of this will be covered in the implementation section.

4.0 Linda

Linda is a language designed to support distributed processing and distributed data structures (Carriero and Gelernter 86). At the heart of Linda is a conceptually shared memory space through which all processes communicate. The medium of communication is the *tuple*, a set of arguments which may be actual or formal in nature. Thus the name for the shared memory is *tuple space* (TS).

Linda provides four operations on tuples : *out*, *in*, *read*, and *eval*. These are used respectively to post a tuple to TS (out), match tuples and delete them from TS (in), match tuples without withdrawing (read), and cause a program to start execution in TS (eval). The authors have admitted to limited success with the eval primitive (Carriero and Gelernter 88).

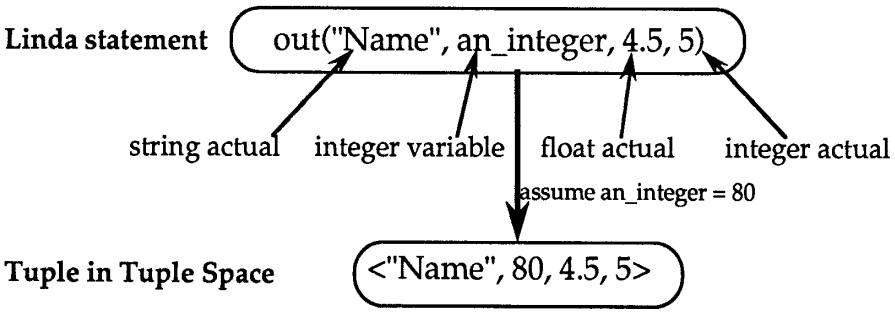

Figure 3 : Tuples

Figure 3 gives an example of an out call and the resulting tuple. In this case, there were no formal parameters given, however it is possible to do so. Formals are mainly used with the in() and read() calls. Since the behavior of the calls is important to R–Linda, they are described here.

On an out() call, the resulting tuple is immediately posted to TS and the process continues execution. In other words, there is no blocking of the process making the out() call.

An in() call is more involved. The tuple created by the in() call will attempt to match with an already existing tuple in TS. If this is possible, the in–tuple will combine with the matched tuple (thus instantiating the in–tuple formal parameters), and will be returned to the calling process. This will also delete the matched tuple from TS. If no matching tuple is present, the calling process suspends until a match is found. Read() works the same way as in(), except it does not delete the matched tuple.

The matching process is important since it is the method of information exchange between Linda processes. The match must occur at three levels, the number of arguments of the tuples, the argument types, and actual values. This means that the value parameters in an in-tuple or read-tuple, must match the value for the out-tuples, and the type of a formal must match the type of a value. When a match occurs between a formal and a value, the formal receives the value (Figure 4).

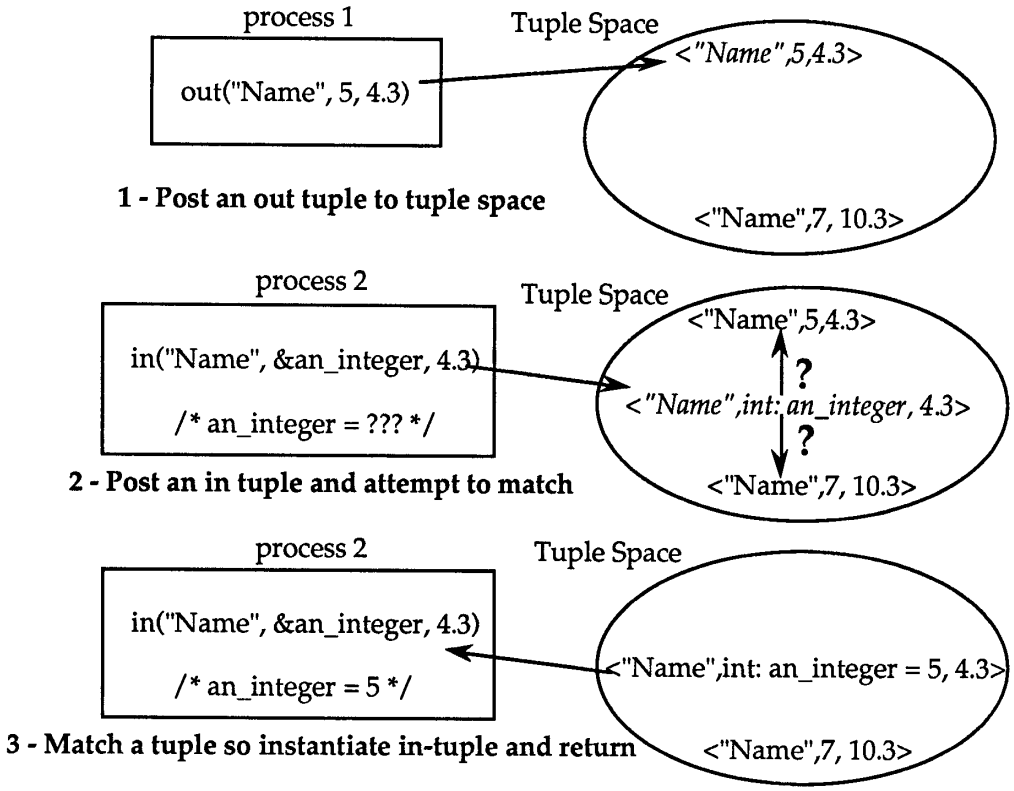

Figure 4 : Matching in Linda

It was mentioned that the tuple space provides a conceptually shared memory. It is conceptual because TS does not necessarily reside on one machine. In fact, it is recommended that TS be distributed amongst many machines, resulting in many local tuple spaces (TSl). The union of these TSl is the global TS. Thus, the physical memory that comprises the tuple space may not reside on the same machine as the processor that is accessing it.

For this project, the important points to note are as follows :

1 - The semantics of the out(), in(), and rd() calls :
    out : posts a tuple to tuple space and does not block
    in : blocks until a tuple is matched, deletes the matching tuple
    rd : blocks until a tuple is matched, does not delete tuple
2 - The matching process requires a match on all three levels
    i) the number of arguments is identical
    ii) the argument types are the same
    iii) values match, and formal types match value types
3 - Tuple space may be distributed among many smaller local TS

5.0 The Tools

R–Linda is built as a C library and a server routine which use the network computing system (NCS). The major constraints for R–Linda are directly attributable to the semantics and operation of NCS, so a description of it is given.

The aim of NCS is to provide a "set of tools for heterogeneous distributed computing" (Apollo 87). In terms of the description of RPC given above, NCS consists of a runtime library and an interface definition compiler (NIDL). The runtime library handles all binding and connections, as well as the transmission of data between the client and the server. It also performs any data translations that are needed

The NIDL compiler generates the client and server stubs from an interface definition. This consists of an interface identifier that is unique across space and time, type definitions, and procedure call declarations. The type of data transmitted can be arbitrarily complex. However, if a structure contains pointers, routines to convert the structures to and from transmissible forms must be provided. The routines end up copying data and managing the memory associated with both the transmittable and non-transmittable structures. Thus if you wished to transmit a linked list, it would first have to be converted into an array at the transmitting, and converted back into a linked list at the receiving end.

NCS also provides a set of routines for generating unique identifier numbers. The identifier consists of the current time, the node id, and a reserved field. The identifiers are used by a client to locate a server. This is done by another part of NCS called the *location brokers*. Their function is to maintain a database of all active servers and their current addresses. There are two types of locations brokers, local and global. The local broker (llbd) runs on every node that uses NCS. It maintains the database of servers present on the local node and global brokers. The global location broker (glbd) maintains a list of all llbd. Both types of brokers may contain information on remote servers. Each time a call is made by a local client, the binding information for the remote server is kept by the location broker that found the binding. The brokers are accessed through the NCS runtime library.

The actual process of binding to a server can be done automatically by the client stub, or manually by the client. The binding must be done manually if the client accesses more than one replicated server. A bound handle gives the client all the information needed to communicate with a server. The reverse is not true. A server has no way of accessing the client through NCS calls. It is possible for the client to send sufficient information for the server to contact the client (a UNIX socket address), but this contact would not use any of the NCS facilities.

6.0 R–Linda : The Implementation

R–Linda has two parts, a library containing all the client code, and a server that administers tuple space on a local node. A program which uses R–Linda does not do any binding, it merely makes out(), in(), and rd() calls. The figure on the next page illustrates the steps needed to post a tuple into tuple space using R–Linda. It gives the operations at the R–Linda level and the corresponding data.

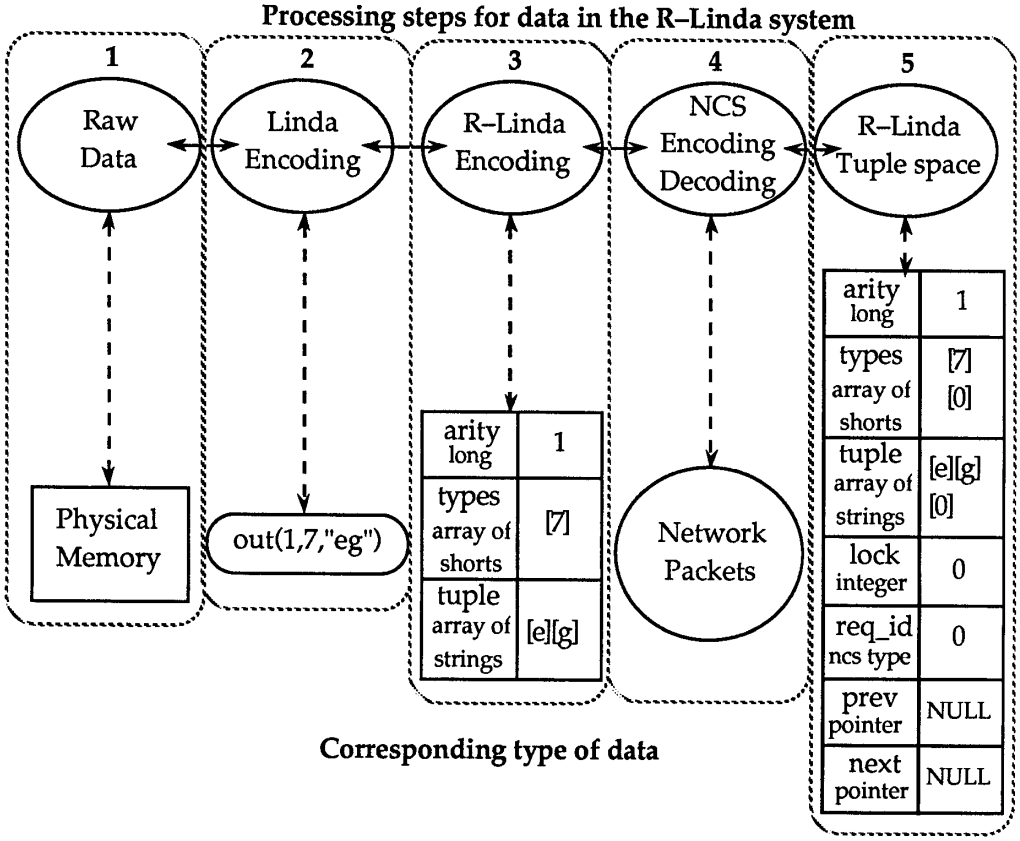

Figure 5 : Flow of Data in R–Linda

In designing R–Linda to run on NCS, two major constraints came to light. Firstly, the data transmitted between the client and the server (step 4 above) needed to be as simple as possible. This is due to the NIDL compiler, since for any reasonably complex type, a lot of processing is involved (See Appendix A). A further constraint was introduced by the nature of the Linda calls (step 2 above).

The number of arguments of a tuple (the *arity*) is not known in advance. This requires using the C varargs library. Varargs lets you decode a variable argument lists, but you must know the number of arguments and type of each argument before decoding it. To do this, the first argument in the varargs list is the arity of the tuple expressed as a C long integer. The elements of the tuple are expressed as pairs of one short integer and the actual tuple argument. The short integer identifies the type of the following argument. An out() in R–Linda has the form :

out(arity, type_arg_1, arg_1, type_arg_2, arg_2, ..., type_arg_n, arg_n) ;

The other calls are similarly defined. In order to be consistent with C, a formal is passed as a pointer (i.e. if *x* is an integer, it would be passed as *&x*). The type argument is a number from 0 to 19, with all pointers having a type number greater than 9.

The process of decoding a tuple (step 3 above) consists of looping for the number of arguments and scanning the varargs list into an appropriate type. There is no need to encode a tuple since the caller already knows the form. The only requirement is to instantiate formal parameters, which consists of assigning the matching value to the address of the formal.

The next decision is the format of the tuples used in the RPC calls (step 3 and 4 above). Three factors influenced this : the transmitted data should be small, the number of arguments is variable, and both the type and value (or address) of each argument is needed. Ignoring the first factor, the best solution is a structure that contains the arity, a list of shorts for the types, and a list of strings for the tuple. But, as noted, the lists would be inefficient. This leads to the use of arrays instead of lists. In the actual procedure calls, a tuple is represented by three parts, an arity, an array of shorts for the types, and an array of strings for the arguments.

The disadvantage of using arrays is that an upper limit must be placed on the length of tuples and the size of the arguments. Also, the whole array must be copied. However, if lists were used, it would need to be copied into an array, then the array would need to be transmitted to the receiver, and the receiver would also need to copy the array back into a list. So the method chosen does save time.

Figure 6 (over) gives an example of how R–Linda would look in a network. There are *** parts in an R–Linda system. The user client program uses the R–Linda facilities to run a Linda program. This user program communicates using the R–Linda library (rlclient in the figure). This library establishes contact with linda servers and provides the functionality needed to make the Linda calls. The server part (rlserver) maintains tuple space. It also performs matching. Both the client and the server may communicate

with remote servers. This is done using NCS which makes remote procedure calls both on the local node, and to remote nodes. Thus when the rlclient makes a call to rlserver, that all goes through NCS.

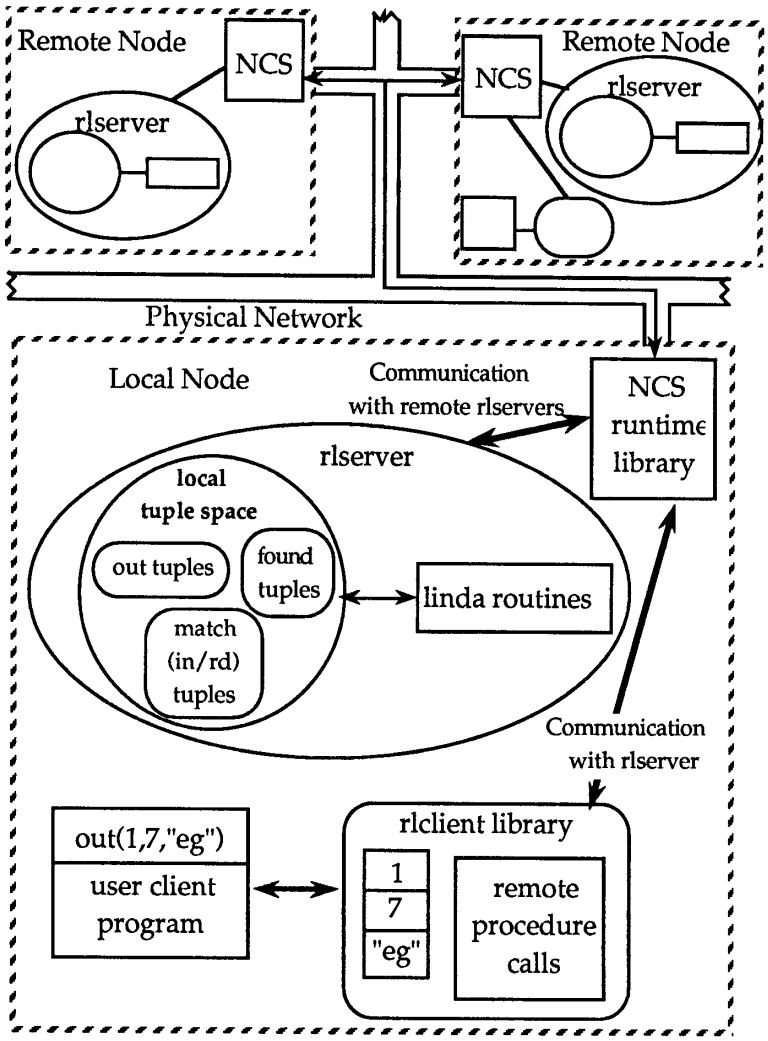

Figure 6 : The Structure of R–Linda on a Network

The second major constraint imposed by NCS involves the semantics of remote procedure calls. As noted above, the client binds to a server then performs the procedure call. The server services the call and returns any values to the client. The client blocks during the call, and the server can not process other requests until it has completed the current one. There is no provision for the server to bind to the client. Thus it can not suspend the client call and continue service at a later time. The server must execute the entire procedure before it can service another call. Though NCS provides

facilities for concurrency in the server, they are minimal (a maximum of 10 simultaneous calls), and require very careful checking that the code is fully re-entrant.

This does not pose problems for the out() call, since it is non-blocking, but this is not true of the in() and rd() calls. If the server blocked until a match occurred, it would not be able to process other client calls, which would prevent it from receiving a matching tuple. The server would need to suspend the current client call and queue it for later resumption which is not possible using NCS or traditional RPC. Since the server can not queue blocked clients, another method of blocking the client program is needed. One possibility is to have the server fork off processes to deal with waiting clients. This is not reasonable since common tasks in Linda cause a large amount of traffic in tuple space (Carriero & Gelernter 88). Thus some other method for blocking the client is needed.

In order to achieve this, one should realize that only the users client process, the one making the in() or rd() call, needs to block. The actual library process can simulate blocking semantics for the users process. This can not be done with a single call, since this would block both the client library and the server. Since a call to a server blocks until completion, the only alternative is to busy wait for the matching tuple. That is, to check at regular time intervals if the server has found a matching tuple.

It is clear that continual transmission of a tuple is inefficient, as is the match checking that the server must perform. To circumvent this problem, a scheme of request identifiers is used. An in() or rd() tuple that is not immediately matched from the local tuple space is assigned a unique identifier (using the facilities provided by NCS) which can be used by the busy wait call to check if a match has been made. Thus there is less to transmit, and only one element to search on.

The last decision involving the client is how to match tuples on remote Linda servers. There are two main ways to do this, replicate the out tuples or replicate the in/rd tuples. The literature for Linda recommends replicating the in/rd tuples. This is due to the nature of the in call, which deletes the matching out tuple. The fact that only one rd() can match an out tuple, requires the ability to lock off the out tuple. If the out tuples are replicated, this becomes complex, and requires confirmation from all servers before an out tuple is matched.

If the out space is local, and the in/rd space is distributed, the match will occur at the out tuples home server. This makes both the delete and match protocols simple. When a match occurs, a local server needs to check if the match is still needed on the home server of the in/rd tuple. Using this scheme requires that all remote servers are informed when an in/rd tuple needs to be matched, which is done only once. In the replicated out tuple

scheme, a transmission is made to all servers at least twice, once to post the out tuple, and once to confirm the delete.

It remains to decide whether the client or server will transmit the match request. The client is used, since it will be doing less work than the server. If one considers that a remote match is only checked for after local matches have failed, it is clear the the client would be in a blocked state. That is, it would be checking for a match at regular time intervals. This checking needs processing by the server, which would be binding to, then sending match requests, to all remote servers. It makes more sense for the client to send the match requests since it is already in a block phase and would otherwise be busy-waiting.

These constraints led to the client library having the following flow of control for out(), in(), and rd() statements.

```
out() :
   convert users client out to linda call structures
   if not already bound to server
      bind to server
   post the tuple to the server

in() :
   convert users client in to linda call structures
   if not already bound to server
      bind to server
   post the tuple to the server
   if no match occurred
      if not already go remote servers
         get remote servers
      for the number of remote servers
         post match request on remote server n
      while no match
         wait for time interval
         check for match
   for the number of remote servers
      deregister match request
   instantiate users original in tuple

rd() :
   (the same as in)
```

The server provides the actual tuple space and the matching of tuples. The decision to make the out tuples local and the in/rd tuples replicated, lead to two distinct tuple spaces. One, called the local tuple space (local_ts) is composed strictly of out tuples. The other is composed strictly of in/rd tuples and is called the match tuple space (match_ts) since the tuples resident in it are attempting to match something. The match space contains tuples from the local node and from remote nodes.

The next problem is how to assure that a match works properly. That is, if an in/rd tuple from a remote node matches a local out, other tuples must be prevented from matching that out tuple until the remote in/rd tuple has had a chance to confirm the match. This is especially important for in tuples, since they delete the out tuple that they match. To accomplish this, a lock is provided for an out in the local_ts data structure. Since there is no concurrent programming involved, the lock is simply an integer flag.

More subtle than matching out tuples, is matching in/rd tuples. Since match_ts is possibly replicated, it needs to be checked on every post of an out tuple on a local node. If a match occurs, a lock on the in/rd and matched out tuple is required until the remote node has queried if the match is still needed.

Finally, when a match is made, the client program needs a way of claiming the correct tuple. This is provided by a third tuple space, called found tuple space (found_ts). Posting to this space can occur from remote Linda servers since a match may be made on a remote host. This requires sending the tuple data to the call that posts the match. Thus posting remotely to this space requires a data copy, though searching it uses the request identifier.

The following are the names of the routines from the interface definition file, along with pseudo-code for their actions.

```
linda_out :
   checks for a match in match_ts
   if found
      get remote server
      if still_needed
         post_found on remote
      post local_ts
   post local_ts

linda_in & linda_read :
   generate unique identifier     /* unique id */
   check for a match in local_ts  /* returns matching tuple
                                     if it exists */
   if not found
      post match_ts

linda_match_in & linda_match_read :
   check for a match in local_ts
   if not found
      post match_ts
   else
      get remote server
      if still_needed
         post match_ts
      else
```

```
put matched out back into local_ts  /* in only */
```

```
linda_check_found_in & linda_check_found_read :
   check found_ts for match on required id
   return true if found else return false

linda_get_in & linda_get_read :
   get tuple from found_ts based on required id

is_needed :
   check match_ts for still required using required id
   return result

linda_found :
   post tuple to found_ts

deregister_tuple :
   remove in/rd tuple of required id from match_ts
```

The final decision involved matching. The section on Linda describes matching as occurring on three levels, arity, argument types, and values. The match is checked with all tuples in a tuple space. A tuple space consists of an array (whose length is the maximum arity) of pointers to tuple structures. A tuple structure (Figure 5) contains the array of types, the array of tuple strings, the arity, a lock, a unique identifier, and a pointer to the next and previous items. If it is an in/rd tuple there a flag for in or read, and an entry for the *home* node of the tuple, where home is defined as the originating node of the match request.

lock	Integer. 1 if a lock is on
id	An NCS generate unique identifier
types	An array of shorts
tuples	An array of strings
next	Pointer to the next tuple in the tuple space
prev	Pointer to the previous tuple in the tuple space
home	The home node of this tuple
is_read	A flag that is 1 for a rd or 0 for an in.

Figure 7 : A Tuple Space Data Structure

Given the above structure, the match algorithm is as follows :

```
match (ts, t, m)
tuple_space ts ;  /* the tuple space to check */
tuple_rec   *t ;  /* the tuple to find a match for */
tuple_rec   **m ; /* a match if one is found */
{
   if (tuples of arity t.arity in ts)
      while (tuples of arity t.arity in ts)
         for (t.arity)
              if ~(t.type[arity] match ts.type[arity])
                   *m = NULL
                   return to caller
         for (t.arity)
              if ~(t.tuple[arity] match ts.type[arity])
                   *m = NULL
                   return to caller
         /* found a match, check lock */
         if (t.lock)
              *m = NULL
              return to caller
         else
              *m = current ts pointer
              return to caller
   else
      *m = NULL
      return to caller
}
```

This is not strictly how the calls are made, but the algorithm is the same.

This algorithm attempts to do the least amount of work to fail a match. The first check, for tuples of the required arity in tuple space, is the first level of matching between tuples, identical arity. The next stage checks for type matching. This is implemented as a look up table accessed by the out and in/rd type value. The final stage compares the actual strings if the particular type is not a formal. It makes no sense to compare the tuple strings if one type is a formal.

7.0 Conclusions

The R–Linda system currently runs on Apollo computers that support NCS. In order to check the performance of the system in relation to the original Linda, the ping/pong test was run (See Appendix A). The Linda kernel achieves between 720-770 pairs per second in a simple in/out test. R–Linda only achieves 3 pairs per second and a rate of 24 RPC calls per second.

This is very low compared to the kernel version, and one might attribute it to inefficient code. In order to isolate for this, a skeleton version of the code

that did no processing was used (See Appendix A). The pairs per second only increased by 0.81, which means that the inefficiency is due to NCS. To confirm this, the skeleton was used with no delay time for the busy wait, resulting in 7.17 pairs per second (57 RPC calls per second).

It is clear that the pairs per second could be increased if the busy waiting were eliminated. This requires an extension to RPC, namely the ability of the server to obtain the information necessary to queue the client and service it at a later time.

Two additions are required for this. Firstly, a method of obtaining a 'client handle'. This handle would need to contain enough information for the RPC runtime library to establish contact with the suspended client process on the correct node.

The other addition is a method for the server to inform the RPC runtime library that it has suspended processing of the current client call and is ready to process other requests. Under the usual RPC scheme, a server responds to a request, processes the request, and returns the result. The return contains an implicit 'ready for next request' statement by the server. If client suspension were added, there would need to be an explicit 'ready for next request' in addition to the implicit one.

As for the original goals of the project, they have been satisfied. The system utilizes NCS (RPC) exclusively for all R–Linda related communications. The semantic of the out(), in(), and rd() statements are those given in (Carriero & Gelernter 86) using the busy wait methodology. And an attempt was made to minimize transfer of data using the required identifier scheme.

In reference to the prediction that Linda would not be efficient on top of RPC, this has been confirmed by the findings of the project. Referring to Appendix A, it was found the NCS (RPC) accounts for 77 % of time time to process an in/out pair.

NCS implementation of Linda
– Logically sound – Inefficient in 3 ways 1 – Data copying 2 – Data processing for transmission 3 – Unsuitablity of RPC semantics

R–Linda is logically sound. That is, the semantics of the Linda calls is preserved, and the behavior is the same as Linda. However it is not usable for heterogeneous distributed processing. In order to make R–Linda feasible, NCS would need to change in three ways. Firstly, the amount of data copying would need to be reduced. This is possible by using text compression and prediction techniques. Such a method would increase problem 2. The unsuitability has been discussed above, and involves adding two calls to NCS. One to get a handle to a client, and the other to inform the NCS runtime library that the server is ready to service more calls.

As a final word, I would like to suggest some possibilities for further work. The first thing to try is using the domain distributed services (DDS) protocol ports with the servers. Though this would not speed up operations to a kernel level, it is faster than the IP protocol. Another idea would be to implement Linda on top of message passing and compare the results with R–Linda. It would also be useful to implement Linda on top of UDP, which would require implementing a method for locating Linda servers. The simplest way to do this would be using well know ports.

8.0 Bibliography

Apollo Computer Inc. (1987). *Network Computing System (NCS) Reference.* System Documentation No. 010200, Revision 00.

Birrell, A.D. and Nelson, B.J. (1984). *Implementing remote procedure calls.* ACM Transactions on Computer Systems, Vol. 2, No. 1, February. Pps 39-59.

Carriero, N. and Gelernter, D. (1986). *The S/Net's Linda kernel.* ACM Transactions on Computer Systems, Vol. 4, No. 7, May. Pps 110-129.

Carriero, N. and Gelernter, D. (1988). *Applications experience with Linda.* Proceedings of the ACM/SIGPLAN Parallel Programming : Experience with Applications, Languages and Systems. New Haven, Connecticut, July 19-21. Pps 173-187.

Appendix A : Testing R–Linda

To have a good comparison with the original Linda, a test given in the (Carriero & Gelernter 86) paper was used. This consisted two programs which used in/out pairs. They are reproduced from the paper below :

```
PING :
   count = 0 ;
   while (TRUE) {
      in("ping") ;
      if (++count == LIMIT) break ;
      out("pong") ;
   }
   print elapsed time ;

PONG:
   while (TRUE) {
      out("ping") ;
      in("pong") ;
   }
```

These are used to test the number of in/out pairs per second that the kernel can process. In the case of R–Linda, the same programs were used (with the modified in/out formats) with a LIMIT of 500 calls. The test was performed on a single node (DN 4500, 68030 processor) with the server using the IP communications protocol. Each of the processes was run in a separate window (ping, pong, rlserver). The three processes accounted for approximately 99 % of the nodes total processing during the test runs.

Tests were made with a full implementation of R–Linda using various delay times for the busy wait. Only one of these sets of figures is given below (the fastest with a delay of 0.125 seconds). Measurements were also made with an R–Linda skeleton for a 0.125 second delay and no delay. The skeleton program is R–Linda with all the processing stripped from the client and server sides. That is, the out/rd/in do no conversion, they just make the call. In the case of in, it makes a linda_in, then only one linda_check_in, and finally a linda_get_in (rd is the same). The server does not processing, it merely returns a single function value if needed. The table below summarizes the results :

Delay time (sec)

Time for 500 calls (sec)

Trial #	.125	.125 (skeleton)	0 (skeleton)
1	163	131	67
2	174	125	73
3	161	126	61
4	173	134	66
5	163	129	58
6	169	133	78
7	175	133	87
8	159	135	85
9	164	124	60
10	163	141	62
Avg.	166.4	131.1	69.7
pairs/sec	3	7.17	3.81
calls/sec	18	22.9	43

Table 1 : Ping/Pong Testing Values

The other measurement taken was the time to do a null remote procedure call. This was done to isolate for the time that R–Linda actually takes. In the above table the calls/sec entry gives the approximate number of remote procedure calls per second. It is estimated that 500 in/out pairs will take 3000 remote procedure calls on average (2000 best case, at least 4000 worst case). This is based on the following breakdown :

500 ping/pong cycles best case (always matches) :
1000 linda_out calls
1000 linda_in calls

500 ping/pong cycles worst case (no matches) :
1000 linda_out calls
1000 linda_in calls
1000 linda_check_in calls
1000 linda_get_found_in calls

Since the processes are running concurrently, the actual sequence of calls is non-deterministic. Tracing the sequence is not useful since it would both add time, and change the sequence of processing. The 3000 figure reflects the fact that check calls effectively take longer due to the busy waiting. The actual

check call time for the server is very short compared to the linda_out and linda_in calls. The 3000 calls figure is conservative.

The test was performed under the same circumstances, except only two processes were used (a client and a server). It was found that 3000 remote procedure calls take 25.2 seconds, which is 119 calls per second.

Using these figures the time that R–Linda takes for processing (including busy waits) and the time that NCS takes for data transmission can be estimated :

R–Linda processing = R–Linda time – skeleton time
= 166.4 - 131.1
= 35.3 seconds

Data transmission = skeleton time (no wait) – 3000 null call time
= 69.7 – 25.2
= 44.5

With these figures we can now get a percentage breakdown of the time taken per R–Linda ping/pong pair transaction :

Total processing time for a single pair = 166.4 / 500 = .33 seconds

R–Linda processing per pair = 35.3 / 500 = 0.07 seconds 21 %

Data transmission per pair = 44.5 / 500 = 0.089 seconds 27 %

NCS overhead per pair = .33 – 0.07 – 0.089 = 0.171 52 %

This means NCS is responsible for 77 % of the time taken to process one pair. Of this, it is possible to work on the data transmission by using more efficient representations.