Linda, Tuple Spaces And Its Implementation On A Network Of UNIX* Workstations

Janakiram Cherala
Department of Computer Science
Oregon State University
Corvallis, OR 97331
ram@cs.orst.edu

A research paper
submitted in partial fulfillment of
the requirements for the degree of
Master of Science

Major Professor : Timothy Budd
Minor Professor : Bruce D'Ambrosio
Committee Member: Bella Bose

February 15, 1991

*UNIX is a registered trademark of AT&T

Contents

List of Figures

List of Tables

Abstract

In this paper we outline an implementation of Linda on a network of Unix workstations. A literature survey was done to gain a better perspective on state of the art and to learn from the experiences of other implementations. The tuple space which is central to the Linda system is implemented as multiple segments distributed on different systems. The biggest challenge in implementing Linda is management of the tuple space. We outline a mechanism for creation and management of the distributed tuple space. Linda has been embedded in C with a partial support for the underlying datatypes of C. Results of some of the test routines run on the system along with some comparative timings are provided.

1 Introduction

In writing parallel programs, the conventional choice has been between shared memory systems, which are easy to program but very difficult to program correctly, and distributed, message passing systems, which are harder to program, but safe. In either case writing programs for parallel machines has been much harder than writing programs for sequential machines. What is worse, a program written for one system is not easily portable to another. Linda was developed at Yale [ACG86] to make the task of writing parallel programs an easy and an enjoyable experience.

In this paper we outline an implementation of Linda on a network of Unix workstations. A description of the language is given as an introduction. A literature survey was done to gain a better perspective on state of the art and to learn from the experiences of other implementations. The tuple space which is central to the Linda system is implemented as multiple segments distributed on different systems. The biggest challenge in implementing Linda is management of the tuple space. We outline a mechanism for creation and management of the distributed tuple space.

The implementation was originally done on a network of Tektronix workstations. The program was later ported to a network of Sun workstations. We ran some test programs to get a feel for the performance of the system while using the Linda primitives. The timing was done for the Sun implementation.

2 Linda, the language

Linda was first defined by Gelernter [Gel82]. Linda is an explicitly parallel programming language and *not tied* to any parallel hardware architecture. Linda has been implemented on shared memory multiprocessors like Sequent Balance, disjoint memory systems such as the Intel iPSC/2 hypercube [Bjo89] and a network of VAX systems [Lei89]. Linda is not a language on its own standing. Rather, it is a set of objects and operations on those objects that are intended to be embedded into an existing language, thus producing a new language for distributed programming [Lei89].

2.1 Fundamental objects

Linda is based on two fundamental objects: *tuples* and *tuple spaces*.

2.1.1 Tuples

Tuples are ordered collections of *fields*. Fields have fixed types associated with them; the types are drawn from the underlying language. A field can be a *formal* or an *actual*. A formal field is a place-holder — it has a type, but no value. An actual field carries a value drawn from the set of possible values allowed for that type by the underlying language.

Supposing the underlying language has types **int** and **float**. Then

$< 1_{\text{int}}, 3.5_{\text{float}}, 2_{\text{int}} >$

is a tuple consisting of three fields: Two integers, 1 and 2, and a floating point value, 3.5. This tuple has only actual fields. The tuple

$< 1_{\text{int}}, \square_{\text{float}} >$

contains an integer actual, and a float formal.

Linda places no *a priori* restrictions on what types are allowed. It is determined by what the underlying language allows. So you could have records, arrays, and pointers as types.

2.1.2 Tuple space

Tuples live in *tuple space*, which is simply a collection of tuples. It may contain any number of copies of the same tuple; thus it is a *bag*, not a *set*. Tuple space is the fundamental medium of communication in Linda. Unlike in traditional message passing models of communication, in Linda all communication is three-party communication. Sender interacts with tuple space and tuple space interacts with receiver, rather than sender and receiver directly interacting with each other.

Tuple space is a global, shared object — all Linda processes that are part of the same program have access to the same (logical) tuple space.

2.2 Operations

The **out** operation inserts a tuple into tuple space.

If f is a float with value 3.5 and i an integer with value 2. The expression

out(1, f, i)

will insert into the tuple space the tuple $< 1_{\text{int}}, 3.5_{\text{float}}, 2_{\text{int}} >$ we saw earlier. The operation **out** is nonblocking. So, it inserts the tuple in to the tuple space and then returns control to the program immediately.

The **in** operation extracts tuples from tuple space. It finds tuples that *match* its arguments. **in** is a blocking operation. So, if a matching tuple is not available in the

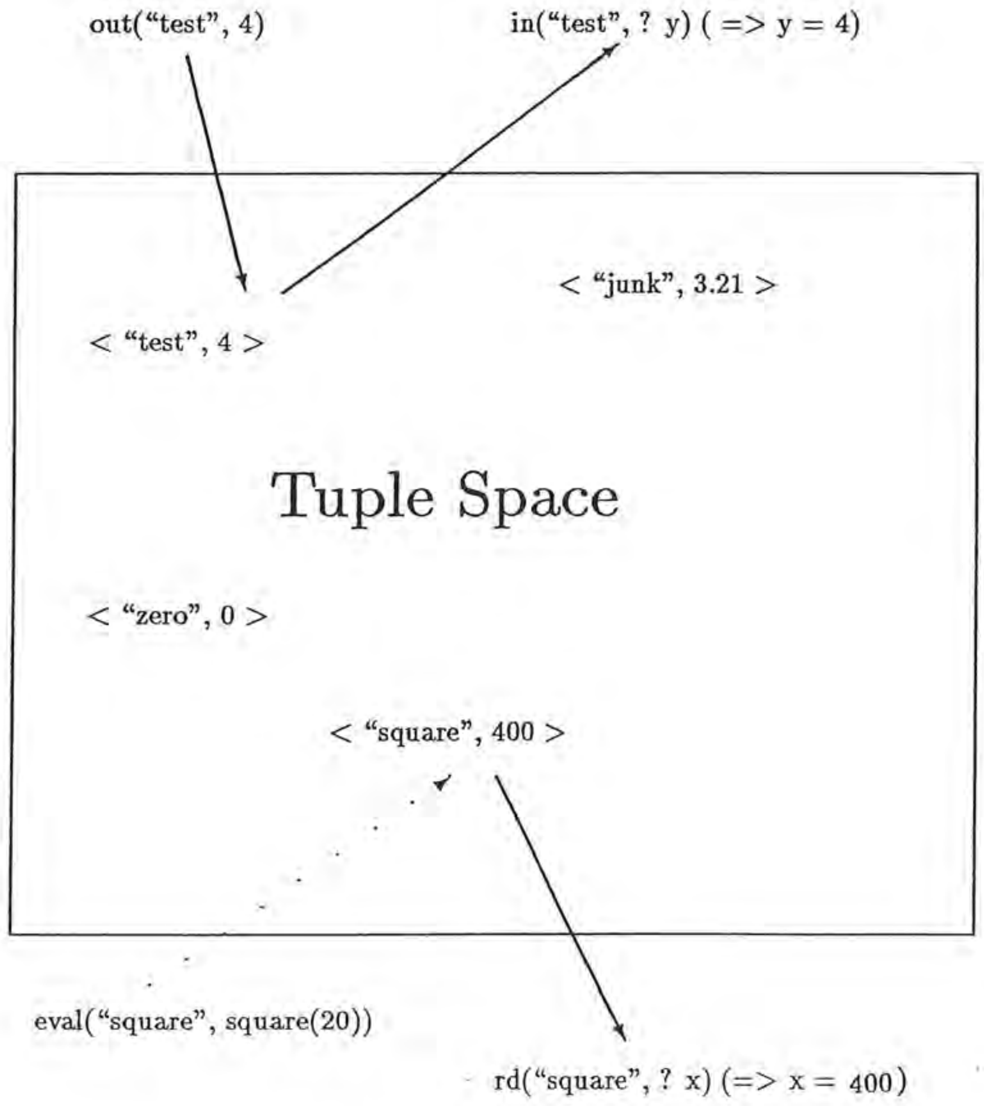

Figure 1: A Snapshot of Tuple Space

tuple space, then this operation would block till a match is found.

So, the tuple which was inserted into the tuple space can be extracted by the operation:

in(1, 3.5, 2)

The **rd** operation works very similar to **in** but it doesn't extract the tuple from the tuple space.

A successful **in** or **rd** results in variable assignments. This means that on a successful match of a template to a tuple, the formal parameters specified in the template will have actual values assigned from the tuple.

An **eval** operation is similar to the **out** operation, in that it inserts a tuple into the tuple space. The difference is in the fact that **eval** inserts an *unevaluated* tuple into the tuple space. The **eval** operation produces an active tuple. An active tuple cannot be matched by any **in** or **rd** operation. The tuple begins evaluating soon as it is created. On completion of evaluation, the active tuple becomes a passive tuple and this in turn can be matched in the usual way.

3 A literature survey

The following sections summarize some of the existing implementations of Linda.

3.1 VAX Linda

Leichter and Whiteside in their paper "Implementing Linda for Distributed and Parallel Processing" describe the implementation of Linda on a network of VAX systems [LW89]. VAX Linda allows a single application program to utilize many machines on a network simultaneously. In their implementation they demonstrate that, for suitable applications, a collection of separate machines on a network can be treated as a "virtual multicomputer." VAX Linda operates on two levels. At one level, it uses shared memory for communication among multiple processes running on a single "node" of the network. At the second level, the implementation views multiple nodes connected to the network as a "virtual multiprocessor."

VAX Linda uses a hash table to implement the tuple space. The tuples have a type signature, and this in turn is used for hashing into the tuple space. The second level (called the LAN level) combines multiple segments of tuple memory into a unified single tuple space. The system has to ensure that matching tuples and templates find each other. In VAX Linda tuples are handled locally, while templates are broadcast. The authors argument in favor of template broadcast is as follows. On an ethernet based network, there is bound to be packet loss and it is simple to allow the broadcast of a template to be repeated rather than that of the tuple.

The linda process interacts with a Listener process on its node. A Listener receiving a template searches its local tuple space segment. On failure, it does nothing. On success, it effectively does an **in** on the matched tuple. It then sends the matched tuple to the requesting Listener, using a virtual circuit. Once successful transmission has been completed, the tuple is deleted from the sending tuple space segment.

In this implementation, Linda is embedded into the C language and the interpreter interprets Linda-C programs and puts out C code.

3.2 Linda on a Hierarchical Multiprocessor

Here we look at an implementation by Lothar Borrmann, Martin Herdieckerhoff and Axel Klein, of Linda on Parwell-1 multicomputer, a hierarchical multiprocessor [BH 88].

Parwell-1 is a 37-node distributed memory multiprocessor developed and manufactured by pl GmbH, a Munich-based company and installed at Siemens. Its nodes consist of a Motorola processor pair 68020/68881, up to four MByte of local memory, two 32-bit bus connections and an address translation logic. The interconnection between nodes is achieved via an hierarchical bus system and the memories between the nodes of adjacent hierarchical levels. The Parwell multiprocessor is not a stand alone system, but is operated as a co-processor connected to a host station. In this implementation they have used an Apollo workstation running UNIX System V as a host station. All memory accesses are top-down. Each processor in the hierarchy has access to its own local memory as well as to the local memories of all the nodes in the subtree below it.

The implementation uses the physically shared memory as a global tuple space. In this implementation Linda is embedded into Modula-2. Tuple classes are declared statically like types and variables. A tuple class declaration states the name of that particular class and the types of each list element. These types can be any Modula-2 types like strings, open arrays, etc., but not pointers. Such a class declaration allows for type checking to be done at compile-time, reducing complexity of run-time kernel and facilitates error detection. The number of parameters in a tuple or template is restricted to a maximum of eight.

The tuple to template matching is done efficiently by utilizing the hierarchical nature of the memory. The tuples are stored locally whereas the templates migrate between nodes looking for a match.

3.3 Kernel linda

QIX is a parallel operating system based on Linda, developed at Cogent Research, Beaverton, OR. Kernel Linda [Cogent 89.16] is a version of Linda designed to be used as the interprocess communication mechanism for the operating system. The system employs a kernel that provides memory management, process scheduling, synchronization, and interprocess communication via Linda. The rest of the operating system is implemented through servers, which communicate with each other and with user processes using Linda. Cogent has developed an advanced version of Linda that supports multiple tuple spaces, persistent tuple spaces, and communication between programs written in different languages, including C, C++, FORTRAN, and PostScript. All communication is handled by Linda.

Servers are used to implement a high degree of compatibility with the UNIX operating system, while the user interface is a parallel implementation of SUN Micro Systems NeWS window system.

In Kernel Linda, a tuple to template match is done by allowing a single key in a tuple and using standard hashing techniques to search the tuple space for a match. Kernel Linda allows tuple space to be stored as a field of a tuple. In addition to supporting Linda operations, the Kernel Linda data types provide a shared object space, which can be implemented on a system that contains shared memory, or simulated on a distributed system.

3.4 Conclusion

Linda has become a popular choice for program development in the parallel environments. It has been implemented on several different systems and embedded into different languages. We have studied the implementation here on a network of VAX systems, on a hierarchical processor and as an operating system primitive. This study gives us a feel for the power and flexibility of Linda as a language and as a tool. Linda makes the task of writing parallel programs as easy as it can get. Though industry has accepted Linda by supporting various installations, we are yet to see some solid results to prove the efficacy of Linda systems. The tuple space search seems to make the overall system very sluggish and we need more efficient implementations to take care of this potential problem.

4 Multiple and persistent tuple spaces

Existing definitions of Linda [Gel85, Car87] are built around the assumption of a single tuple space though some definitions like [Lei89] and [Kl89] allow for the creation and use of multiple, persistent tuple spaces. Similarly, existing implementations create *evanescent* tuple spaces, which exist only as long as a given Linda program runs. However, it's clear that *persistent* tuple spaces might be useful objects, which would reduce the overhead of initializing the tuple space every time a program is run. In our implementation we support persistent tuple spaces for the above reason.

5 C-Linda, an embedding of linda in C

In our implementation, the Linda primitives **in**, **out**, **rd**, and **eval** are provided as C callable functions. So, the programmer writes C-Linda code as if writing a C program. The primitives interact with the Tuple Space Manager (TSM) to insert tuples, or extract tuples from the Tuple Space (TS). The primitives use the Unix socket based Inter Process Communication to communicate with the TSM. A preprocessor converts the C-Linda code to C code which is then passed on to a standard C compiler for parsing and putting out code. The standard C data types like integer, char, float, double and arrays.

6 Implementation

Here we outline the details of implementing Linda on a network of workstations running Unix Operating System.

6.1 Tuples

A tuple consists of a key field, a count field, and the actual data. The key field is a string of characters which identifies the tuple. The tuple key is primarily used for matching the template with the tuples in the TS. The count field contains the count of the number of data items that are contained in the tuple. The data itself is represented as pairs of values. The first identifies the type of the field and the next is the actual value itself. For example consider the tuple which is inserted into the TS using **out** as out("TEST", a, 10), where *a* is an integer. This would be translated internally as out("TEST", 2, INT, a, INT, 10). Here "TEST" is the tuple key, 2 is the number of data items in the tuple, INT is the data type representing the C data type integer.

6.2 Tuple space (TS)

The TS in our implementation is distributed across a network of Unix workstations. The physically distinct TS's appear as a single entity because the TSM's keep the distributed nature of the TS transparent from the application or the programmer. When an application executes an **in**, a match is tried in the local TS by the corresponding TS manager. If no matching tuple is found, then a broadcast message is sent to other managers. A matching tuple, if found in a remote TS, is sent back to the requesting application or process. So, the application doesn't know if the matching tuple was found locally or from a remote system. The TS is implemented as a hash table for quick access of a required tuple. When an **out** is encountered, the hash value is computed based on the key and the tuple is installed in the appropriate bucket.

When an **in** is encountered, the hash is computed based on the key, the corresponding bucket is scanned using the key as a filter. A full match is then attempted to find the required tuple. The first tuple that matches is removed and returned to the requesting process.

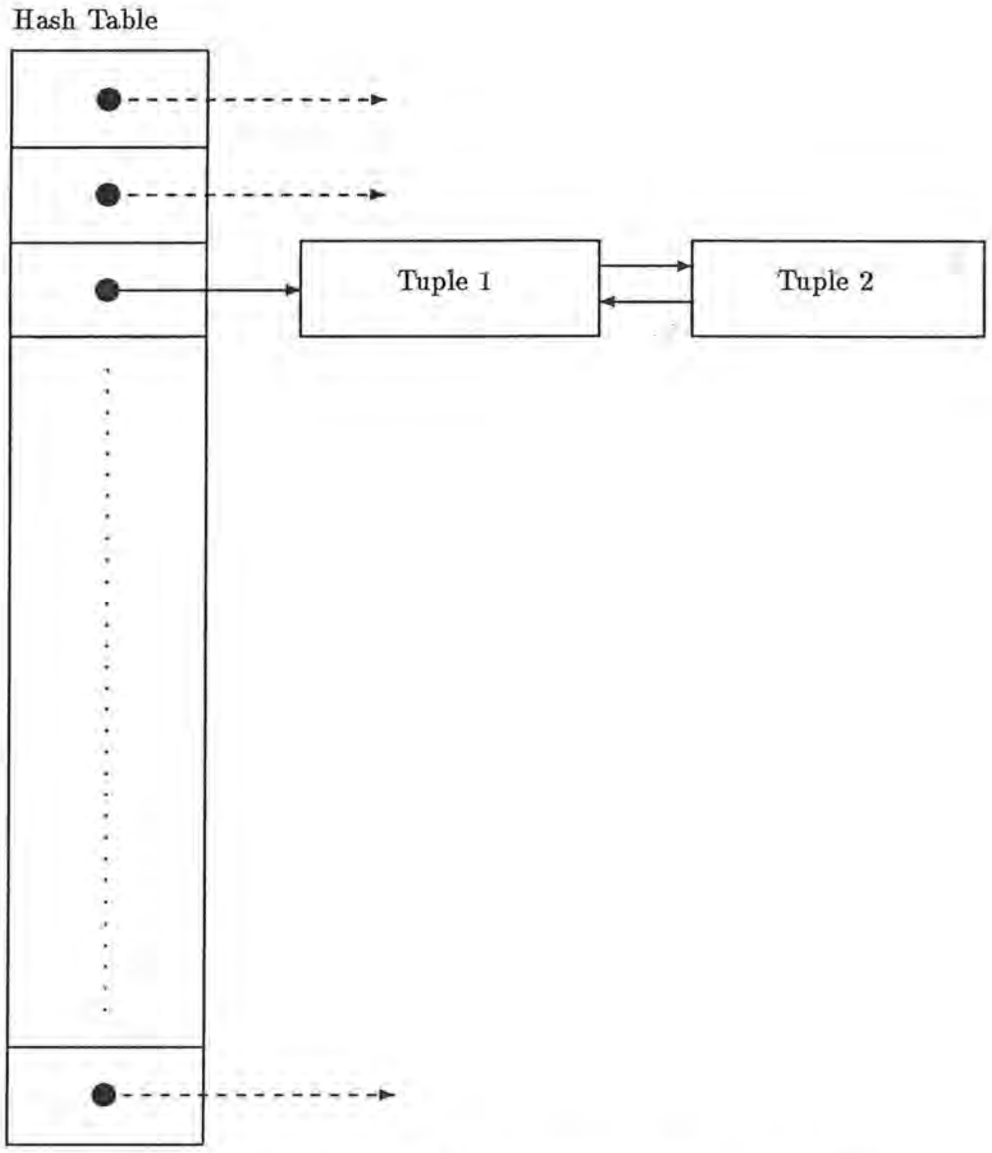

Figure 2: Hash table representing Tuple Space

6.3 Tuplespace manager (TSM)

The TSM is a daemon program, that is, it runs in the background without a controlling terminal. The TSM's on all the nodes in the network are started automatically when the system starts up. This can be arranged by putting the appropriate start-up command in the /etc/rc.local file. Once the TSM's are started, they wait in the background awaiting requests from either the C-Linda programs or from other TSM's.

The C-Linda programs (actually the C-Linda library routines) communicate with the TSM using socket based interprocess communication (IPC). The TSM's communicate with each other over the ethernet also using socket based IPC. When a TSM is started up, a socket is created and it is then bound to a well known address. The TSM then waits at one end of the socket listening for any requests. These requests could be for inserting a tuple in to the local tuple space (TS), a request to match a template with a tuple or a request from another TSM for a tuple.

The request is in the form of a string of characters. The multiple fields of a tuple are space separated. The TSM then parses the string and determines the type of request. If the request is an **out**, then the string which represents the tuple is converted into an internal storage format and stored in the linked list representing the TS. If the request is an **in** instead, then an appropriate routine to find a matching tuple is invoked by the TSM. If a matching tuple exists in the TS, that tuple is sent to the requesting process over the socket. The tuple is then deleted from the local TS. If it is a **rd** request instead, a similar process matches the tuple, but the tuple is not removed from the TS.

If a matching tuple is not to be found in the local TS, this request is queued in a template queue for future processing. A request is then broadcast to the other TSM's for a matching tuple. Now, depending on whether a remote TSM has a matching tuple, the following possibilities arise. The broadcast sent by the local TSM is translated to an **in** request by the other TSM's. So, this request is treated like a local **in** request, but the socket address connected to this request would be a remote socket address. Supposing a matching tuple exists in one of the remote TS, then the corresponding TSM would do an **out** of that tuple to the TS of the requesting TSM. Once the tuple gets deposited into the local TS, a match can occur in the usual way when the template queue is processed. The tuple is removed from the remote TS soon as it is sent to the local TS. Now, once the match occurs, the local TSM would send back the tuple to the requesting process and remove the tuple from the local TS too. If it were a **rd** request, then the tuple is not removed.

Suppose a matching tuple is not available in any of the TS's, the template is then queued in the local template queue of all the TSM's. If a matching tuple is deposited at a later stage as a result of an **out** then a match would take place in the usual way and the tuple is sent to the blocked process. If more than one TSM has a matching tuple, then all the matching tuples are deposited by the remote TSM's into the local TS. This is not a problem because, one of the tuples is matched with the template, whereas the remaining tuples remain in the local TS and can satisfy future requests. If a remote TSM requires a tuple it sent earlier, then the local TSM will send it back.

6.4 Matching

It is easy to envison tuple space implementation on a shared memory system like Sequent Balance. But, on a network of Unix workstations with disjoint memories, where are tuples to be stored? How are they to be found?

We use an implementation called *negative broadcast* as defined in [Lei89]. In this scheme, **out's** always act only on memory local to the machine they run on, storing their tuple in the local segment of the tuple space. Conversely **in's** act globally. First, the search is made in the local TS for a match. If this fails, a broadcast request is made to all other TSM's. Any manager that can respond with a matching tuple will do so.

Negative broadcast works as follows:

- If a matching tuple exists in the local segment, it will be found quickly. Since **out**'s insert tuples into the local segment, an **in** or **rd** for a tuple **out**'ed by the same process a short while before will almost certainly be found locally.
- If there are no matching tuples locally, but matches can be found in more than one remote segment, there will be multiple tuples sent in response. One will be used to satisfy the operation initiating the request; the others will be stored in the local segment, where subsequent **in**'s or **rd**'s can find them quickly.

In figure 3, the nodes represent the Unix workstations interconnected by ethernet on the local area network. Each of the nodes has multiple processes sharing memory representing the multiple TS segments. The **TSM** shown is the daemon process on each node. At the node level an out places a tuple in the local tuple space and awakens any local process waiting for it. On an **in** or **rd**, if the local tuple segment doesn't have a matching tuple, the TSM *multicasts* the template to an address that all TSM's know. A TSM receiving a template searches its local TS for a matching tuple. On failure it does nothing; on success, it effectively does an **in** on the matched tuple from the local TS segment. It then sends the matched tuple to the requesting TSM using a socket based virtual circuit. Once successful transmission has been completed, the tuple is deleted from the sending TS segment.

As explained before, every tuple or template has a key which uniquely identifies the tuple or template. When an **in** or **rd** request is received by the TSM, the **match** routine tries to find a tuple which has the same key as the template. If no such tuple exists, the match fails immediately. Supposing such a tuple does exist, then the next field which represents the number of data items is compared. If this fails, then the TS is searched for the next matching tuple based on the key. This goes on until all the tuples in the TS are exhausted. Supposing the number of data items matches, then a comparison is made to determine if the data types match. If they do then the tuple matches the template, else it doesn't.

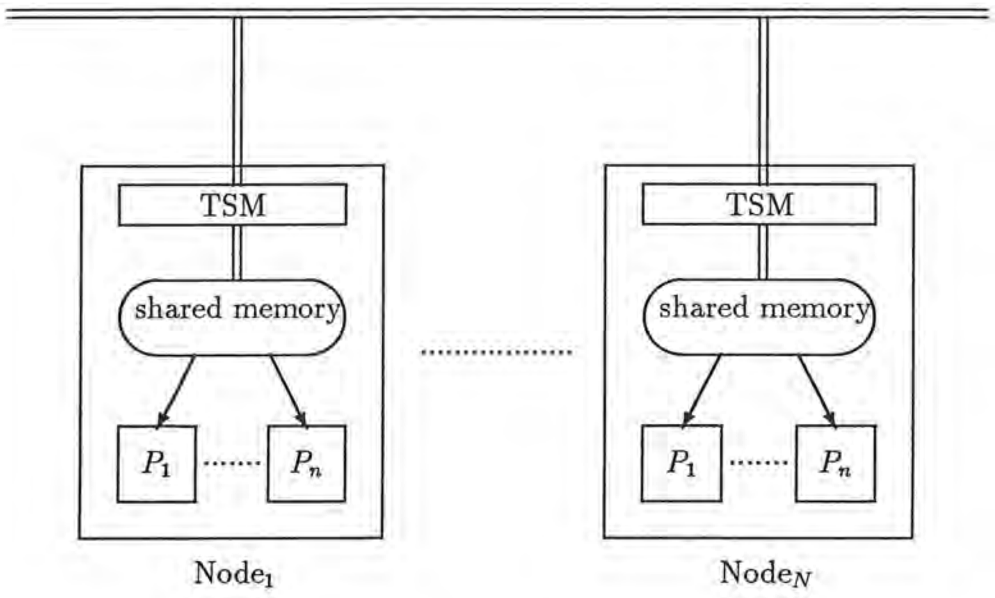

Figure 3: Tuple Space Manager's on the Network

6.5 Implementing the primitives

The section on TSM covers the implementation of **in**,**out** and **rd**. The tuple or template given as an argument to these are converted to an internal format and either stored in the TS or a search for a match is initiated. If more than one TSM is available on the local network, then the TSM's communicate with each other using socket based IPC and maintain the distributed TS. If a matching tuple is unavailable in the local TS, then a request for a tuple is broadcast on the network. If a matching tuple exists elsewhere on the net, the corresponding TSM will send the tuple back.

Implementation of **eval** is slightly different and involved. The argument to **eval** is not a tuple, but a function. This function is forked off in parallel with the parent code. When the function eventually completes execution, it returns an integer value (the current version supports functions that return integer value only). This integer value is combined with the rest of the tuple and a tuple is now inserted into the TS using **out**. This tuple can then be extracted by an eventual **in** or **rd** request in the usual way.

7 Performance and timing

The performance of our implementation of Linda is presented with respect to timings measured on some primitive operations and for a couple applications. Some basic code was timed to obtain some measurements to give us a feel for the performance on the workstations. Most of the timing routines used here are taken from [Car87] and modified to suit our requirements.

Each loop had the form:

```
for (i = 0; i < 10000 ; i++)      {

     <body>
}
```

One iteration took 1.1 μsecs for a null loop body, 1.4 μsecs for "sum += i;" as the body, and 2.3 μsecs for "foo();" as the body. The variables i and sum were declared int and foo() was the null function "{;}."

7.1 Local TS timings

The timing measurements were made for a simple form of **out** of a tuple consisting of a key and an integer field. The following code was used to time the **out** call.

```
main()
{
     int i;
     long j, k;

     j = lindatime();
     for (i = 0; i < 10000; i++)   {
          out("string", 4);
     }
     k = lindatime();
     printf("elapsed time %ld\n", k - j);
}
```

The timing routine **lindatime()** was written using the Unix system calls for measuring process time. The average execution time for five trials was 27 secs, or 2.7 msecs per **out**. Now to find out the time taken for an **in** call, a corresponding **in** was added to this program, the time for an **out-in** pair was measured and then the time for an **in** calculated from that.

So, the program to test **in** is as below.

```
main()
{
    int i;
    long j, k;

    j = lindatime();
    for (i = 0; i < 10000; i++)   {
        out("string", 4);
    }
    for (i = 0; i < 10000; i++)   {
        in("string", 4);
    }
    k = lindatime();
    printf("elapsed time %ld\n", k - j);
}
```

The average execution time for five trials was 73 secs, or 7.3 msecs per **out-in** pair. So, for a simple **in** the timing was 4.6 msecs. The **rd** version of the above code needs to **out** only once. That code resulted in a timing of 4.4 msecs per **rd**.

7.2 Multiple TS timings

In the following timings we have the TSM's on multiple systems communicating with each other to find matches to queued tuple templates. The first measurement is the basic transaction time — from an **out** on one system to an **in** on another.

The first process on system A executes:

```
main()
{
    int i;
    long j, k;

    j = lindatime();
    for (i = 0; i < 10000; i++)   {
        out("string", 4);
    }
    in("done");
    k = lindatime();
    printf("elapsed time %ld\n", k - j);
}
```

The second process on system B (started before the first process) executes:

```
main()
{
    int i;

    for (i = 0; i < 10000; i++)   {
        in("string", 4);
    }
    out("done");
}
```

The results were 76 secs total execution time, or 7.6 msecs per **out-in** transaction.

The next test measures round-trip time, simulating a series of RPC calls between two processes.

The pair of processes are:

```
ping()
{
    int i;
    long j, k;

    j = lindatime();
    for (i = 0; i < 10000; i++)   {
        out("a", 4);
        in("b", 2);
    }
    k = lindatime();
    printf("elapsed time %ld\n", k - j);
}

pong()
{
    int i;

    for (i = 0; i < 10000; i++)   {
        in("a", 4);
        out("b", 2);
    }
}
```

The time per iteration was 32.18 msecs, which reduces to 16.09 msecs per transaction. This transaction took longer time due to the fact that there is a tighter synchronization between the two processes, thus serializing most of the work.

Test routine/ Linda system		Cogent	Sequent	S/Net	Net Linda
Basic - Null body	μsecs	2.4	7	5.19	1.1
Basic - sum += i	μsecs	3.7	9	7.37	1.4
Basic - foo()	μsecs	5.6	20	14.19	2.3
Simple out	millisecs	0.22	0.09	1.03	2.7
Simple in	millisecs	0.41	0.11	0.97	4.6
Simple rd	millisecs	0.41	0.09	0.46	4.4
Multiple in/out	millisecs	2.1	0.1	1.54	7.6
RPC	millisecs	3.1	0.37	1.99	12.09

Table 1: Comparative Timings

7.3 Comparative timing analysis

The table 1 summarizes the timings for the example routines shown in the previous section. The Cogent timings were obtained by running the test routines on a Cogent Research XTM workstation with 4 processors and 16 MB memory. The Sequent timings were obtained by running the routines on a Sequent Balance 2000 with 28 processors and 32 MB memory. The compiler used was SCA Linda compiler version 2.0. The timings for the S/Net implementation are from [Car87]. The timings for the Net Linda implementation were obtained on a network of Sun4 workstations.

The best timings were obtained on the Sequent implementation. This is because the SCA Linda compiler uses a preprocessor to determine the patterns of tuple usage. This assists in using efficient mechanisms while searching the tuple space for a matching tuple. The next best timings were obtained on the Cogent implementation. As described in the literature survey section, the Cogent implementation uses some hardware mechanisms to make the Linda system efficient. The Cogent XTM workstation has a 32-bit parallel LindaBus shared by all the processors. LindaBus is used to transmit short messages between processors in an efficient manner. It also has a dedicated 32-bit microprocessor on the mother board responsible for controlling the communication between the processors. This allows a transmission of 64 simultaneous messages using a crossbar switch. Our implementation uses socket based IPC mechanisms of Unix for communication which has a lot of overhead and is not known to be terribly fast. We also do not have the benefit of a preprocessor to analyze the tuple space usage. So, the timings obtained compare favourably with the other implementations.

8 Porting to Sun workstations

The original implementation of C-Linda was on a network of Tektronix 4315 workstations running version 3.1 of tek OS. We then decided to port the C-Linda code to a network of Sun-4 SPARC† workstations running version 4.0.3 of SUN OS. The porting itself was trivial requiring minimal modifications in the socket based code. The performance gains were significant probably because of better networking software on the SUNs.

9 Concluding remarks

The implementation of Linda in a distributed environment has been addressed before but managing the tuple space efficiently was a challenging task. We have outlined the mechanism of implementing the Linda primitives, the tuple space manager and the tuple space itself. Our implementation of Linda on a network of Unix workstations is a minimal system in the sense that only a subset of C datastructures have been supported. Also, the implementation of **eval** is very limiting on what functions can be passed to it as a parameter. We haven't provided for fault tolerance in the system; that is there is no backup mechanism in the case of one of the tuple space manager's failure in the midst of a program execution. A preprocessor to analyze the tuple space usage during the compilation phase might improve the performance of the system.

†SPARC is a trade mark of Sun Microsystems, Inc.

References

[ACG86] Ahuja, S., Carriero, N., and Gelernter, D., *Linda and Friends*, IEEE Computer, Vol.19, No.8, pp. 26-34 (Aug. 1986).

[BCGL87] Robert Bjornson, Nicholas Carriero, David Gelernter, and Jerrold Leichter. *Linda, the Portable Parallel*, Research Rreport RR-520, Yale University Department of Computer Science, January 1988.

[BH88] Lothar Borrmann, and Martin Herdiecherhoff, *Parallel Processing Performance in a Linda System*, Siemens AG, Corporate Research and Technology, 1988.

[Bjo89] Robert Bjornson. *Experience with Linda on the iPSC/2*, Technical Report RR-698, Yale University Department of Computer Science, March 1989.

[Car87] Nicholas John Carriero, Jr. *Implementation of tuple space machines*, Technical Report RR-567, Yale University Department of Computer Science, December 1987.

[CG88] Nicholas Carriero, David Gelernter, *Linda in Context*,

[Gel82] David H. Gelernter, *An Integrated Microcomputer Network for Experiments in Distributed Computing*, PhD thesis, State University of New York at Stony Brook, 1982.

[Gel85] David H. Gelernter, *Generative Communication in Linda*, ACM Transactions on Programming Languages and Systems, Vol. 7, No.1. January 1985, Pages 80-112.

[Kl89] *Kernel Linda Specification: Version 4.0*, Cogent Research, Inc. Technical Note 89.17.

[Lei89] Jerrold Sol Leichter, *Shared Tuple Memories, Shared Memories, Buses and LAN's - Linda Implementations Across the Spectrum of Connectivity*, Technical Report TR-714, Yale University Department of Computer Science, July 1989, also PhD thesis.

[LW89] Jerrold S. Leichter, and Robert A. Whiteside, *Implementing Linda For Distributed and Parallel Processing*, Yale University Department of Computer Science, February 1989.

```
/*
 *      Include file Linda.h
 *
 *      Author: Janakiram Cherala
 *      Date  : March 1990
 *
 *      This is the header file for netlinda
 *      Contains defines and other declarations
 */

#include <stdio.h>
#include <string.h>
#include <syslog.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <net/if.h>
#include <netdb.h>

#define GUB(x)              (0)

#ifdef DEBUG
#undef DEBUG
#define DEBUG(x)            (printf x, fflush(stdout), 1)
#define DEBUGGING           (1)
#else
#define DEBUG(x)            (0)
#define DEBUGGING           (0)
#endif DEBUG

#define TRUE      1
#define FALSE     0
#define QUEUE     1
#define DONTQUEUE 0
#define LOCAL     0
#define REMOTE  1
#define LINDAPORT 1098              /* port at which to listen for requests */
#define TUPLELEN 1024
#define TUPLEKEYLEN 20
#define BUFLEN 1024

typedef struct TupleSpace {
        char    tuple_key[TUPLEKEYLEN];
        int     tuple_comps;
        struct  TupleData     *tuple_data;
        struct  TupleSpace *next;
        struct  TupleSpace *prev;
} TUPLESPACE, *TSPOINTER;

typedef TSPOINTER *DICTIONARY;

typedef struct TupleData {
        int data;
        int type;
        struct TupleData *next;
} TUPLEDATA, *TDPOINTER;

typedef struct TupleQueue {
        int operation;
        int site;
        int dequeue;
        struct sockaddr_in from;
        TSPOINTER tuple;
        struct TupleQueue *next;
} TUPLEQUEUE, *TQPOINTER;

#define TSNULL ((TSPOINTER)NULL)
#define TDNULL ((TDPOINTER)NULL)
#define TQNULL ((TQPOINTER)NULL)

#define ARGSIZE 20
```

```
#define MAXTUPLESIZE 1024
#define DATA "Initializing the sockets."

/*
 * it is important that these values start from 0 because I use them as
 * indices into an array for formatting the tuples
 */

#define  IN       0
#define OUT       1
#define RD        2
#define RQST      3
#define DEQUEUE 4

#define BLANKSTRING " "
#define INSTRING "in"
#define RDSTRING "rd"
#define OUTSTRING "out"
#define RQSTSTRING "rqst"
#define DEQUEUESTRING "deq"
#define ACKSTRING "ack"
#define COMPCHAR ";"

/* C type declarations
 * it is important that these values start from 0 because I use them as
 * indices into an array for formatting the tuples
 */

#define CHAR      0
#define INT       1
#define FLOAT     2
#define DOUBLE  3

/* set the value of TYPEOFFSET to the maximum assigned datatype value +1 above.
 * This value is used as an offset during matching in the match_tuple
 * function - see listener.c
 */

#define TYPEOFFSET 4

/* The corresponding address values for the data types
 */

#define CHARPTR          (CHAR+TYPEOFFSET)
#define INTPTR           (INT+TYPEOFFSET)
#define FLOATPTR         (FLOAT+TYPEOFFSET)
#define DOUBLEPTR        (DOUBLE+TYPEOFFSET)
#define FUNPTR           10

/* for broadcast */
#define MSG_REQUEST      ((char)       0)
```

```
/*
 *    A C-Linda library
 *
 *    Author:  Janakiram Cherala
 *    Date:    Feb 26 1990
 *
 *    This library supports the following calls
 *
 *    init_linda()
 *        This function creates sockets and sets up the communication
 *        with the linda daemon
 *
 *    in (argument_list)
 *        try to get a matching tuple if possible, else hang
 *        remove the tuple from tuple space if successful
 *        argument_list is the formals to match
 *        eg:  in (i, f, 2)
 *
 *    out (tuple)
 *        put the tuple into tuple space, return immediately
 *        eg:  out (1, 3.5, 2)
 *
 *    eval (function)
 *        evaluate the function and out the resulting tuple to
 *        tuple space
 *
 *    rd (argument_list)
 *        try to get a matching tuple if possible, else hang
 *        do not remove the tuple from tuple space if successful
 *        just assigns the values of actual parameters to formals
 *        eg:  rd(i, f, 2) would assign the matched tuples's
 *        actual parameter values to i and f
 *
 */

#include <varargs.h>
#include "linda.h"

static char rcsid[] = "$Header: /u2/ram/linda/RCS/linda.c,v 2.8 90/07/28 12:06:04 ram Exp Locker: ram $";

char *type_str[] = {"%c", "%d", "%f", "%lf"};
char *prim_string[] = {INSTRING, OUTSTRING, RDSTRING};

int inited = FALSE;
int sock;
struct sockaddr_in name;
struct hostent *hp, *gethostbyname();
int dummy;

init_linda()
{
        struct servent *sp;
        char hostname[32];

        inited = TRUE;              /* init has been called before */
        sp = getservbyname("brd_test", "udp");
        if(sp == NULL)    {
                perror("unknown service\n");
                exit(1);
        }
        sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock < 0)         {
                perror("opening datagram socket");
                exit(1);
        }
        if(gethostname(hostname, sizeof(hostname)) == -1)            {
                perror("error getting hostname\n");
                exit(1);
        }
        hp = gethostbyname(hostname);
        if (hp == 0)          {
                fprintf(stderr, "%s: unknown host0", hostname);
```

```
			exit(2);
		}
		bcopy(hp->h_addr, &name.sin_addr, hp->h_length);
		name.sin_family = AF_INET;
		name.sin_port = sp->s_port;
}

in (va_alist)
va_dcl
{
		va_list ap;
		char		tuple[TUPLELEN];

		if(inited == FALSE)
			init_linda();
		va_start(ap);
		make_tuple(IN, ap);
		if(recvfrom(sock, tuple, TUPLELEN, 0, &dummy, &dummy) < 0)
			perror("receiving datagram packet");
		DEBUG(("in: tuple received %s\n", tuple));
		if(strcmp (tuple, ACKSTRING) == 0)		{
			if(recvfrom(sock, tuple, TUPLELEN, 0, &dummy, &dummy) < 0)
				perror("receiving datagram packet");
			DEBUG(("in: tuple received %s\n", tuple));
		}
		va_start(ap);
		assign_values_to_variables(ap, tuple);
}

rd (va_alist)
va_dcl
{
		va_list ap;
		char		tuple[TUPLELEN];

		if(inited == FALSE)
			init_linda();
		va_start(ap);
		make_tuple(RD, ap);
		if(recvfrom(sock, tuple, TUPLELEN, 0, &dummy, &dummy) < 0)
			perror("receiving datagram packet");
		DEBUG(("rd: tuple received %s\n", tuple));
		if(strcmp (tuple, ACKSTRING) == 0)		{
			if(recvfrom(sock, tuple, TUPLELEN, 0, &dummy, &dummy) < 0)
				perror("receiving datagram packet");
			DEBUG(("rd: tuple received %s\n", tuple));
		}
		va_start(ap);
		assign_values_to_variables(ap, tuple);
}

out(va_alist)
va_dcl
{
		va_list ap;
		char		tuple[TUPLELEN];

		if(inited == FALSE)
			init_linda();
		va_start(ap);
		make_tuple(OUT, ap);
		if(recvfrom(sock, tuple, TUPLELEN, 0, &dummy, &dummy) < 0)
			perror("receiving datagram packet");
		DEBUG(("out: tuple received %s\n", tuple));
}

eval(va_alist)
va_dcl
{
		va_list ap;

		if(inited == FALSE)
			init_linda();
```

```
        va_start(ap);
        make_tuple(OUT, ap);
}

make_tuple(primitive, ap)
int primitive;
va_list ap;
{
        char *tuple_comp, *ptr;
        char tuple[TUPLELEN];
        char tmp[TUPLELEN];
        int     tuple_component;
        long int tuple_long;
        double tuple_double;
        int     i, count;
        int     fork_id = 0;
        int     type=0;
        int     type_flag=0;                            /* flag to tell us */
                                                        /* we are reading a C type */
        int     (*fun_ptr)();

        tuple[0] = NULL;
        strcat (tuple, prim_string[primitive]);
        strcat (tuple, BLANKSTRING);
        tuple_comp = va_arg(ap, char*);                 /* tuple key */
        strcat (tuple, tuple_comp);
        strcat (tuple, BLANKSTRING);
        DEBUG(("%s: tuple key %s\n", prim_string[primitive], tuple_comp));
        tuple_component = va_arg(ap, int);              /* count */
        count = tuple_component*2 ;
        sprintf(tmp, "%d", tuple_component);
        strcat(tuple, tmp);
        strcat (tuple, BLANKSTRING);
        type_flag = 1;
        for(i = 0; i < count ; i++)             {
                if(type_flag)           {
                        type_flag = 0;
                        tuple_component = va_arg(ap, int);
                        type = tuple_component;
                        if(tuple_component == FUNPTR)
                                tuple_component = INT;
                        sprintf(tmp, "%d", tuple_component);
        GUB(("data type %d\n", tuple_component));
                }
                else    {
                        type_flag = 1;
                        switch(type)            {
                        case CHAR:
                                tuple_component = va_arg(ap, int);
                                sprintf(tmp, type_str[type], tuple_component);
        GUB(("data value %d\n", tuple_component));
                                break;
                        case INT:
                        case INTPTR:
                        case CHARPTR:
                        case FLOATPTR:
                        case DOUBLEPTR:
                                tuple_component = va_arg(ap, int);
                                sprintf(tmp, type_str[INT], tuple_component);
        GUB(("data value %d\n", tuple_component));
                                break;
                        case FUNPTR:
                                DEBUG(("executing a function"));
                                fun_ptr = (int (*)()) va_arg(ap, int *);
                                if( fork_id = fork()) {
                                        tuple_component = (*fun_ptr) () ;
                                        sprintf(tmp, type_str[INT], tuple_component);
                                        exit(0);
                                }
                                break;
                        case FLOAT:
                                tuple_long = va_arg(ap, long);
                                sprintf(tmp, type_str[type], tuple_long);
        GUB(("data value %f\n", tuple_long));
                                break;
                        case DOUBLE:
                                tuple_double = va_arg(ap, double);
                                sprintf(tmp, type_str[type], tuple_double);
        GUB(("data value %lf\n", tuple_double));
                                break;
                        }
                }
                strcat(tuple, tmp);
                strcat (tuple, BLANKSTRING);
        }
        va_end(ap);
        if( type != FUNPTR || ( type == FUNPTR && fork_id ))            {
                if( sendto(sock, tuple, strlen(tuple), 0, &name,
                        sizeof(name))      < 0)
                        perror("sending datagram message");
        }
}

assign_values_to_variables( ap, tuple)
va_list ap;
char *tuple;            /* matched tuple */
{
        char    *token, *tmp_str;
        char    *tuple_comp;
        int     i, tuple_component, type, count;
        int     int_data;
        char    char_data;
        double  double_data;
        int     *pitmp;
        char    *pctmp;
        double  *pdtmp;

        token = strtok(tuple, BLANKSTRING);             /* key */
        tuple_comp = va_arg(ap, char*);                 /* key */
        count = atoi(strtok(NULL, BLANKSTRING));                /* count */
        tuple_component = va_arg(ap, int);
        for ( i = 0; i < count; i++ )           {
                type = atoi(strtok(NULL, BLANKSTRING));
                if(type == CHAR)                {
                        tmp_str = strtok(NULL, BLANKSTRING);
                        char_data = tmp_str[0];
                }
                else if (type == INT)           {
                        int_data = atoi(strtok(NULL, BLANKSTRING));
                }
                else if (type == FLOAT || type == DOUBLE)               {
                        double_data = atof(strtok(NULL, BLANKSTRING));
                }

                tuple_component = va_arg(ap, int);              /* get type */

                if(tuple_component == INTPTR)           {
                        pitmp = va_arg(ap, int*);
                        *pitmp = int_data;
                }
                else if(tuple_component == CHARPTR)             {
                        pctmp = va_arg(ap, char*);
                        *pctmp = char_data;
                }
                else if(tuple_component == FLOATPTR ||
                                tuple_component == DOUBLEPTR)   {
                        pdtmp = va_arg(ap, double*);
                        *pdtmp = double_data;
                }
                else
                        va_arg(ap, int);
        }
        va_end(ap);
}
```

```
/*
 *      Listener module for Net Linda
 *
 *      Author: Janakiram Cherala
 *      Date:   Feb 26 1990
 *
 *      Description:
 *
 *          This is the daemon process that is started on each node
 *          of the network when a Linda program is executed.
 *          When the Linda program executes any of the linda library
 *          calls, they are internally translated to calls to the
 *          listener. Here, socket calls are used for communicating
 *          with the linda process as well as the other listeners.
 *
 */


#include <signal.h>
#include "linda.h"


static char rcsid[] = "$Header: /u2/ram/linda/RCS/listener.c,v 2.15 90/07/28 12:06:18 ram Exp Locker: ram $";


char *convert_tuple_to_string( );
TSPOINTER parse_tuple(), match_tuple();
void insert_tuple(), queue_tuple(), process_queue(), request_tuple();
void remove_tuple(), dequeue_tuple(), send_broadcast_msg(), send_matched_tuple();
void send_dequeue_msg();

char *type_str[] = {"%c", "%d", "%f", "%lf", "%d", "%d", "%d"};
char *operation_string[] = { INSTRING, OUTSTRING, RDSTRING, RQSTSTRING, DEQUEUESTRING };
TSPOINTER tuple_space[102];
TQPOINTER tuple_queue;
int sock, port;
struct sockaddr_in from;
int fromsize = sizeof(from);

main()
{
        int         cntrlc_handler();
        int         length,
                    request,
                    rqst;
        struct sockaddr_in name;
        struct servent *sp;
        char        buf[BUFLEN],
                    Stuple[TUPLELEN], tmpstr[TUPLELEN];
        TSPOINTER Ttuple, Mtuple;
        char        *Ptuple;

#ifdef DAEMON
        disassociate_terminal();
#endif
#ifdef CNTRLC
        signal(SIGINT, cntrlc_handler);
#endif
        sp = getservbyname("brd_test", "udp");
        if(sp == NULL)      {
                    syslog(LOG_ERR, "listener: unknown service: %m");
                    exit(1);
        }
        port = sp->s_port;
        sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock < 0)           {
                    syslog(LOG_ERR, "listener: opening datagram socket: %m");
                    exit(2);
        }
        name.sin_family = AF_INET;
        name.sin_addr.s_addr = htonl(INADDR_ANY) ;
        name.sin_port = sp->s_port;
	if (bind(sock, &name, sizeof(name)))		{
		syslog(LOG_ERR, "listener: binding datagram socket: %m");
		exit(3);
	}
	length = sizeof(name);
	if (getsockname(sock, &name, &length))		{
		syslog(LOG_ERR, "listener: getting socket name: %m");
		exit(4);
	}
	while (TRUE)	{
		bzero(buf, BUFLEN);
		if(recvfrom(sock, buf, BUFLEN, 0, &from, &fromsize) < 0)
			syslog(LOG_ERR, "listener: receiving datagram packet: %m");
		if(strcmp(buf, ACKSTRING) == 0)
			continue;
		/* separates the buffer into type of tuple (request)  & the tuple Stuple */
		request = process_request( buf, Stuple );
		switch( request )		{
		case	IN:
		case	RD:
			Ttuple = parse_tuple(Stuple);
			if( Mtuple = match_tuple( Ttuple ) )		{
				send_matched_tuple(Mtuple);
				if(request == IN)		{
					remove_tuple(Mtuple);
				}
				free( Ttuple );
			}
			else	{
				request_tuple( request, Stuple );
				/* queue the template as a local request */
				queue_tuple( request, LOCAL, Ttuple);
			}
			break;
		case	OUT:
			Ttuple = parse_tuple(Stuple);
			insert_tuple(Ttuple);
			break;
		case	RQST:
		/* this is a request from another listener for a tuple */
			strcpy(tmpstr, Stuple);
			/* to get rid of the request ID */
			rqst = process_request(tmpstr, Stuple);
			Ttuple = parse_tuple(Stuple);
			if( Mtuple = match_tuple( Ttuple))		{
				Ptuple = convert_tuple_to_string(Mtuple);
				buf[0] = NULL;
				/* out the matched tuple to another TS */
				strcpy( buf, OUTSTRING );
				strcat( buf, BLANKSTRING );
				strcat( buf, Ptuple );
				if(sendto(sock, buf, BUFLEN, 0, &from, fromsize) < 0) {
					syslog(LOG_ERR, "listener: sending packet: %m");
				}
				send_dequeue_msg(Mtuple);
/*
 *	Actually the tuple should be removed only if the request is an IN
 *	but there is a problem with duplicate tuples, if I don't delete it
 *	even for rd requests.  So, let us delete it, but get it back if
 *	required here locally
				if(rqst == IN)
 */
					remove_tuple(Mtuple);

			}
			/* queue the template as if it were a local request */
			else
				queue_tuple( rqst, REMOTE, Ttuple);
			break;
		case	DEQUEUE:
			Ttuple = parse_tuple(Stuple);
			dequeue_tuple(Ttuple);
			break;
			}
			process_queue();
			buf[0] = NULL;
			/* send acknowlegement */
			strcpy(buf, ACKSTRING );
			if ( sendto( sock, buf, BUFLEN, 0,
				(struct sockaddr *) &from, fromsize ) < 0 ) {
					perror( "sendto" );
					exit(5);
			}
		}
}

#ifdef CNTRLC
cntrlc_handler()
{
		DEBUG(("Cleaning up the tuple space\n"));

		while(tuple_space)		{
			printf("tuple in TS %s\n", convert_tuple_to_string(tuple_space));
			tuple_space = tuple_space->next;
		}
		exit(6);
}
#endif CNTRLC

char *
convert_tuple_to_string( tuple )
TSPOINTER tuple;
{
/*		This function converts the given tuple into a string form which
 *		can then be conveniently manipulated, for eg. to print or to
 *		send over the socket
 */
		int i;
		TDPOINTER tmp_data;
		char *string, tmp_string[64] ;

		string = (char *) calloc(1, TUPLELEN );
		if( string == (char *) NULL)		{
			syslog(LOG_ERR, "listener: insufficient memory");
			exit(7);
		}
		string[0] = NULL;
		strcpy( string, tuple->tuple_key);
		strcat( string, BLANKSTRING);
		sprintf(tmp_string, "%d", tuple->tuple_comps);
		strcat( string, tmp_string);
		strcat( string, BLANKSTRING);
		tmp_data = tuple->tuple_data;
		for(i = 0; i < tuple->tuple_comps; i++)		{
			sprintf(tmp_string,"%d", tmp_data->type);
			strcat( string, tmp_string);
			strcat( string, BLANKSTRING);
			sprintf(tmp_string,type_str[tmp_data->type], tmp_data->data);
			strcat( string, tmp_string);
			strcat( string, BLANKSTRING);
			tmp_data = tmp_data->next;
		}
		return(string);
}

/*
 *		void dequeue_tuple(TSPOINTER Ttuple)
 *
 *		This function removes the Ttuple from the template queue. This
 *		is necessary to ensure that more than one tuple is not matched
 *		per template.
 */
void
dequeue_tuple(Ttuple)
TSPOINTER Ttuple;
{
	int match = FALSE, i;
	TSPOINTER tuple;
	TQPOINTER tmp_tq, remove_q, prev_q;
	TDPOINTER tuple_data, tmp_data;
	char buf[BUFLEN];

	DEBUG(("dequeue_tuple: dequeue tuple %s\n", convert_tuple_to_string(Ttuple)));
	tmp_tq = tuple_queue;
	prev_q = tmp_tq;
	while( tmp_tq )	{
		/* see if key matches */
		tuple = tmp_tq->tuple;
		if( (strcmp(tuple->tuple_key, Ttuple->tuple_key) == 0) &&
			/* number of components same? */
			(tuple->tuple_comps == Ttuple->tuple_comps))		{
			match = TRUE;
			tuple_data = tuple->tuple_data;
			tmp_data = Ttuple->tuple_data;
			for(i=0; i<tuple->tuple_comps; i++)		{
				if((tuple_data->type == tmp_data->type) &&
					(tuple_data->data == tmp_data->data))
					match = TRUE;
				else if(tuple_data->type ==
						tmp_data->type+TYPEOFFSET)
					match = TRUE;
				else	{
					match = FALSE;
					break;
				}
				tuple_data = tuple_data->next;
				tmp_data = tmp_data->next;
			}
			if(match == TRUE)	{
				remove_q = tmp_tq;
				prev_q->next = tmp_tq->next;
				tmp_tq = tmp_tq->next;
				free(remove_q);
			}
		}
		prev_q = tmp_tq;
		tmp_tq = tmp_tq->next;
	}

}

disassociate_terminal()
{
	int i;
/* disassociate the listener from the controlling terminal
 */
	for(i = 0; i < 3; i++)
		close(i);
	open("/", O_RDONLY);
	dup2(0, 1);
	dup2(0, 2);

	i = open("/dev/tty", O_RDWR);
	if(i >= 0)	{
		ioctl(i, TIOCNOTTY, 0);
		close(i);
	}
}

/*
 *	int get_ifc( int sock )
 *
 *	  Put the IFCONF data structure into the ifc structure and return
 *	the number of ifreq structures it contains.
 */

static struct ifconf ifc;

int
```

```
get_ifc( sock )
int sock;
{
     static char ifc_buffer[BUFSIZ];

     ifc.ifc_len = sizeof(ifc_buffer);
     ifc.ifc_buf = ifc_buffer;
     if ( (ioctl( sock, SIOCGIFCONF, (char *) &ifc )) < 0 ) {
          perror( "get ifconf" );
          exit(1);
     }
     return ifc.ifc_len / sizeof(struct ifreq);
}

hash( key )
char *key;
{
          int i, sum = 0;
          int len;

          len = strlen(key);

          for( i = 0; i < len ; i++ )
                    sum += key[i];

          return(sum % 101 );
}

/*
 *        void insert_tuple( TSPOINTER Ttuple)
 *
 *        return value: nothing
 *
 *        insert the tuple pointed to by Ttuple in the tuple space
 *        this is a result of an "out" or "eval" request in to the tuple space
 */

void
insert_tuple( Ttuple )
TSPOINTER Ttuple;
{
          int bucket;
          TSPOINTER old_header;

          DEBUG(("insert_tuple: inserting tuple %s\n", convert_tuple_to_string(Ttuple)));

          bucket = hash(Ttuple->tuple_key);
          old_header = tuple_space[bucket];
          tuple_space[bucket] = Ttuple;
          tuple_space[bucket]->next = old_header;
          /* create the backward link for easy deletion */
          if(old_header)
                    old_header->prev = tuple_space[bucket];
}
/*
 *        TSPOINTER match_tuple( TSPOINTER tuple)
 *
 *        return value:  pointer to the matched tuple in tuple space
 *
 *        Try to find a match of the template tuple in the tuple space
 *        If match occurs, return a pointer to the matched tuple
 *        else return a NULL pointer
 */

TSPOINTER
match_tuple(tuple)
TSPOINTER tuple;
{
          int match = FALSE, i;
          TSPOINTER tmp_ts;
          TDPOINTER tuple_data, tmp_data;
          char buf[BUFLEN];
	DEBUG(("match_tuple: tuple %s\n", convert_tuple_to_string(tuple)));

	tmp_ts = tuple_space[hash(tuple->tuple_key)];
	while( tmp_ts )	{
		/* see if key matches */
		if( (strcmp(tuple->tuple_key, tmp_ts->tuple_key) == 0) &&
			/* number of components same? */
			(tuple->tuple_comps == tmp_ts->tuple_comps))		{
			match = TRUE;
			tuple_data = tuple->tuple_data;
			tmp_data = tmp_ts->tuple_data;
			for(i=0; i<tuple->tuple_comps; i++)		{
				if((tuple_data->type == tmp_data->type) &&
					(tuple_data->data == tmp_data->data))
					match = TRUE;
				else if(tuple_data->type ==
						tmp_data->type+TYPEOFFSET)
					match = TRUE;
				else	{
					match = FALSE;
					break;
				}
				tuple_data = tuple_data->next;
				tmp_data = tmp_data->next;
			}
			if(match == TRUE)		{
				return( tmp_ts );
			}
		}
		tmp_ts = tmp_ts->next;
	}
	/* if you are here, then no match has occured */
	return(TSNULL);
}

/*
 *	TSPOINTER parse_tuple( char *tuple )
 *
 *	return value: pointer to tuple space structure
 *
 *	parse the tuple and return a pointer to a structure of type
 *	TupleSpace
 */

TSPOINTER
parse_tuple(tuple )
char *tuple;
{
	char *tmp_str;
	int i, type;
	char *tuple_key;
	char tmptuple[TUPLELEN];
	TSPOINTER store_in;
	TDPOINTER tmp;

	DEBUG(("Parse_tuple: parsing %s\n", tuple));
	tmptuple[0] = NULL;
	/* make a copy so that strtok doesn't screw up the input string */
	strcpy( tmptuple, tuple);
	store_in = (TSPOINTER) calloc(1, sizeof(TUPLESPACE));
	if( store_in == TSNULL)	{
		syslog(LOG_ERR, "listener: insufficient memory");
		exit(8);
	}
	tuple_key = strtok(tmptuple, BLANKSTRING);
	strcpy(store_in->tuple_key, tuple_key);
	store_in->tuple_comps = atoi(strtok(NULL, BLANKSTRING));
	store_in->tuple_data = (TDPOINTER) calloc(1, sizeof(TUPLEDATA));
	if(store_in->tuple_data == TDNULL)		{
		syslog(LOG_ERR, "listener: insufficient memory");
		exit(9);
	}
	tmp = store_in->tuple_data;
```

```
        for(i = 0; i < store_in->tuple_comps; i++)                {
                /* first get the size from type of data */
                type = tmp->type = atoi(strtok(NULL,BLANKSTRING));
                if(type == INT || type == CHARPTR
                || type == INTPTR || type == FLOATPTR || type == DOUBLEPTR)
                        tmp->data = atoi(strtok(NULL, BLANKSTRING));
                else if(tmp->type == CHAR) {
                        tmp_str = strtok(NULL, BLANKSTRING);
                        tmp->data = tmp_str[0];
                }
                else if(tmp->type == FLOAT)
                        tmp->data = atol(strtok(NULL, BLANKSTRING));
                else if(tmp->type == DOUBLE)
                        tmp->data = atof(strtok(NULL, BLANKSTRING));

                tmp->next = (TDPOINTER) calloc(1, sizeof(TUPLEDATA));
                if(tmp->next == TDNULL)   {
                        syslog(LOG_ERR, "listener: insufficient memory");
                        exit(10);
                }
                tmp = tmp->next;
        }
        return( store_in );
}

/*
 *      void process_queue( void )
 *
 *      return value: nothing
 *
 *      process the queue of pending requests and see if we can match
 *      any of the templates with the available tuples in local Tuple
 *      Space
 */

void
process_queue()
{
        TQPOINTER head_q, prev_q,   remove_q;
        TSPOINTER Mtuple;
        struct sockaddr_in tmp_from;

        DEBUG(("processing queue\n"));
        head_q = tuple_queue;
        prev_q = tuple_queue;
        tmp_from = from;

        while(tuple_queue)            {
                from = tuple_queue->from;
                if(Mtuple = match_tuple(tuple_queue->tuple))            {
                        /* send the matched tuple to the requesting process */
                        send_matched_tuple(Mtuple);
                        /* send a dequeue message to the other listeners so that
                         * they can remove the template from the queue of
                         * waiting templates
                         */
                        if(tuple_queue->dequeue == TRUE)
                                send_dequeue_msg(Mtuple);
                        if(tuple_queue->operation == IN ||
                                tuple_queue->site == REMOTE)
                                remove_tuple(Mtuple);
                        remove_q = tuple_queue;
                        prev_q->next = tuple_queue->next;
                        tuple_queue = tuple_queue->next;
                        if( head_q == remove_q )
                                head_q = tuple_queue;
                        free(remove_q);
                }
                else        {
                        prev_q = tuple_queue;
                        tuple_queue = tuple_queue->next;
                }
        }
        from = tmp_from;
        tuple_queue = head_q;
}

/*
 *      process_request( char *buffer, char *tuple)
 *
 *      return value:       type of request - an integer
 *
 *      Buffer is the string received on the socket by the listener.
 *      This function analyses the buffer, determines the type of request,
 *      and stores the request string (which is a tuple) in the string
 *      tuple.  It returns the type of request in request.
 *
 *      A typical tuple would appear as
 *              in TEST 2 0 2345678 1 10
 *         ^^^^ key
 */

process_request( buffer, tuple)
char *buffer;                   /* input with identifying request */
char *tuple;                    /* output without the identification */
{
        char    tmp[TUPLELEN];
        char    *token;
        int     request;

        strcpy( tmp, buffer );
        DEBUG(("process_request:  %s\n", buffer));
        token = strtok(buffer,BLANKSTRING);
        if (strcmp(token, INSTRING) == 0)           {
                strcpy( tuple, strtok(NULL, "\n") );
                request = IN;
        }
        else if (strcmp(token, OUTSTRING) == 0)     {
                strcpy( tuple, strtok(NULL, "\n") );
                request = OUT;
        }
        else if (strcmp(token, RDSTRING) == 0)      {
                strcpy( tuple, strtok(NULL, "\n") );
                request = RD;
        }
        else if (strcmp(token, RQSTSTRING) == 0)            {
                strcpy( tuple, strtok(NULL, "\n") );
                request = RQST;
        }
        else if (strcmp(token, DEQUEUESTRING) == 0)         {
                strcpy( tuple, strtok(NULL, "\n") );
                request = DEQUEUE;
        }
        else    {       /* this came from other listener */
                strcpy(tuple, tmp);
                request = OUT;
        }
        return( request );
}

/*
 *      void queue_tuple(int what, int site, TSPOINTER tuple)
 *
 *      return value: nothing
 *
 *      store the unmatched tuple templates in a queue for future matching
 *      what is the operation (in, rd, eval, etc) that was requested and
 *      tuple is the pointer to the template.  The site information is stored
 *      in the queue so that some site specific processing can be done later.
 */

void
queue_tuple(what, site, tuple)
int what;
int site;
TSPOINTER tuple;
```

```
{
        TQPOINTER tmp_queue, tmp;

        DEBUG(("queue_tuple: queueing tuple %s\n", convert_tuple_to_string(tuple)));
        tmp_queue = tuple_queue;
        tmp = tuple_queue;

        while(tuple_queue)          {
                tmp = tuple_queue;
                tuple_queue = tuple_queue->next;
        }

        tuple_queue = tmp;
        tmp = (TQPOINTER) calloc (1, sizeof(TUPLEQUEUE));
        if(tmp == TQNULL)           {
                syslog(LOG_ERR, "listener: insufficient memory");
                exit(11);
        }
        tmp->next = TQNULL;
        tmp->operation = what;              /* type of tuple in/out/eval */
        tmp->site = site;                   /* local or remote request  */
        if(site == REMOTE)                  /* should a dequeue broadcast be sent */
                tmp->dequeue = TRUE;
        else
                tmp->dequeue = FALSE;
        tmp->from = from;                   /* where did the request come from */
        tmp->tuple = tuple;                 /* note that this tuple that we
                                             * point to here needs to be freed
                                             * when the match occurs later
                                             */

        if(tuple_queue)
                tuple_queue->next = tmp;
        else
                tuple_queue = tmp;
        if(tmp_queue)
                tuple_queue = tmp_queue;
}

/*
 *      void remove_tuple( TSPOINTER tuple)
 *
 *      return value:       nothing
 *
 *      remove the tuple from tuple space - this is done when a match occurs
 *      as a result of an "in" request

void
remove_tuple( tuple )
TSPOINTER tuple;
{
        int i;
        TSPOINTER tmp;
        TDPOINTER tmp_data, data;

        DEBUG(("remove_tuple: removing tuple %s\n", convert_tuple_to_string(tuple)));

        if(tuple == tuple_space)            {
                tmp = tuple;
                if(tuple_space->next)
                        tuple_space->next->prev = TSNULL;
                tuple_space = tuple_space->next;
        }
        else    {
                tmp = tuple;
                tuple->prev->next = tuple->next;
                if(tuple->next)
                        tuple->next->prev = tuple->prev;
        }
        data = tuple->tuple_data;
        for(i=0; i<tuple->tuple_comps; i++)         {
                tmp_data = data->next;
                free(tmp);
                data = tmp_data;
```

```
		}
		free(tmp);
}
*/

void
remove_tuple( tuple )
TSPOINTER tuple;
{
		int i;
		int bucket;
		TSPOINTER tmp;
		TDPOINTER tmp_data, data;

		DEBUG(("remove_tuple: removing tuple %s\n", convert_tuple_to_string(tuple)));
		bucket = hash(tuple->tuple_key);
		if(tuple == tuple_space[bucket])			{
				tmp = tuple;
				if(tuple_space[bucket]->next)
						tuple_space[bucket]->next->prev = TSNULL;
				tuple_space[bucket] = tuple_space[bucket]->next;
		}
		else		{
				tmp = tuple;
				tuple_space[bucket]->prev->next = tuple_space[bucket]->next;
				if(tuple_space[bucket]->next)
						tuple_space[bucket]->next->prev = tuple_space[bucket]->prev;
		}
		free(tuple);
}

void
request_tuple(what, tuple )
int what;
char *tuple;
{
/*
 *		This function broadcasts a request for the given tuple using
 *		the socket sock. The request is directed at servers on the named
 *		port on all machines in the local INET network.
 */

	static char data[TUPLELEN];

	DEBUG(("request_tuple:  request %s\n", tuple));

	data[0] = NULL;
	strcat(data, RQSTSTRING);
	strcat(data, BLANKSTRING);
	strcat(data, operation_string[what]);
	strcat(data, BLANKSTRING);
	strcat(data, tuple);
	send_broadcast_msg(data);
}

void
send_broadcast_msg( data )
char *data;
{
	struct sockaddr_in sin;
	struct sockaddr dst;
	int off = 0, on = 1;
	int n;
	int data_len = strlen(data) + 1;
	int i= 0;
	struct hostent *host, *gethostent(), sethostent();
	char hostname[64];

#ifdef SO_BROADCAST
	struct ifreq *ifr;

	/* Set socket for broadcast mode */
```

```
if ( (setsockopt( sock,SOL_SOCKET,SO_BROADCAST,&on,sizeof(on) )) < 0 ) {
    perror( "set socket option for broadcast" );
    exit(1);
}

/* Initialize broadcast address and bind socket to it */
sin.sin_family = AF_INET;
sin.sin_port = port;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
bind( sock, (struct sockaddr *) &sin, sizeof(sin) );

for ( n = get_ifc(sock), ifr = ifc.ifc_req; n > 0; --n, ++ifr ) {

    /* Only deal with AF_INET networks */
    if ( ifr->ifr_addr.sa_family != AF_INET ) continue;

    /* Use the current address by default */
    bzero( (char *) &dst, sizeof(dst) );
    bcopy( (char *)&ifr->ifr_addr, (char *)&dst, sizeof(ifr->ifr_addr) );

    /* Get the flags */
    if ( ioctl( sock, SIOCGIFFLAGS, (char *) ifr ) < 0 ) {
        perror( "get ifr flags" );
        exit(1);
    }

    /* Skip unusable cases */
    if ( !(ifr->ifr_flags & IFF_UP) ||                      /* If not up, OR */
         (ifr->ifr_flags & IFF_LOOPBACK ) ||                /* if loopback, OR */
         !(ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTOPOINT)) )
        continue;

    /* Now, determine the address to send to */
    if ( ifr->ifr_flags & IFF_POINTOPOINT ) {
        if ( ioctl( sock, SIOCGIFDSTADDR, (char *) ifr ) < 0 ) {
            perror( "get ifr destination address" );
            exit(1);
        }
        bcopy( (char *) &ifr->ifr_dstaddr,
               (char *) &dst,
               sizeof(ifr->ifr_dstaddr) );
    }
    if ( ifr->ifr_flags & IFF_BROADCAST ) {
        if ( ioctl( sock, SIOCGIFBRDADDR, (char *) ifr ) < 0 ) {
            perror( "get ifr broadcast address" );
            exit(1);
        }
        bcopy( (char *) &ifr->ifr_broadaddr,
               (char *) &dst,
               sizeof(ifr->ifr_broadaddr) );
    }

    /* ... make sure that the port number is OK */
    if ( dst.sa_family == AF_INET ) {
        struct sockaddr_in *sa_in = (struct sockaddr_in *) &dst;
        sa_in->sin_port = port;
        DEBUG(( "Request address = %s (%d)\n",
                inet_ntoa(sa_in->sin_addr), ntohs(port) ));
    }

    /* ... and send the request */
    DEBUG((" Send to %d [%2d] %s\n",sock,data_len,show_data(data,data_len)));
    if ( sendto( sock, data, data_len, 0,
                 (struct sockaddr *) &dst, sizeof(dst) ) < 0 ) {
        perror( "sendto" );
        exit(1);
    }
}

/* Reset socket option to normal operation */
if ( (setsockopt( sock,SOL_SOCKET,SO_BROADCAST,&off,sizeof(off) )) < 0 ) {
    perror( "reset socket option for no broadcast" );
    exit(1);
    }

#else !defined(SO_BROADCAST)
    /* Simulate broadcast if you ain't got it */
    gethostname(hostname, 64);
    while ( (i++ < 30) && (host = gethostent()) != NULL ) {
        /* Only deal with AF_INET networks */
        if ( host->h_addrtype != AF_INET ||
             *host->h_addr == 127    /* LOOPBACK */ || (strcmp(host->h_name, hostname) == 0)) {
            GUB(("Host %s has address family %d & first %d\n",
                    host->h_name, host->h_addrtype,
                    (unsigned char) *host->h_addr ));
            continue;
        }

        /* Start building the address from scratch */
        bzero( (char *) &dst, sizeof(dst) );

        {
            struct sockaddr_in *sa_in = (struct sockaddr_in *) &dst;

            /* Copy the host address into the destination address */
            bcopy( (char *) host->h_addr,
                    (char *) &(sa_in->sin_addr),
                    host->h_length );

            /* ... make sure that the port number is OK */
            sa_in->sin_family = host->h_addrtype;
            sa_in->sin_port = port;
        }

        /* ... and send the request */
        /*
        GUB(( "Send request to %s at %s %d\n", host->h_name,
                inet_ntoa(*(struct in_addr *)host->h_addr), ntohs(port) ));
        GUB((" Send [%2d] %s\n",data_len,show_data(data,data_len)));
        */
        if ( sendto( sock, data, strlen(data), 0,
                        (struct sockaddr *) &dst, sizeof(dst) ) < 0 ) {
            perror( "sendto" );
            exit(12);
        }
    }
#endif SO_BROADCAST
}

/*
 *        send the dequeue msg to all the listeners
 */

void
send_dequeue_msg( Ttuple )
TSPOINTER Ttuple;
{
        char *buf;
        char fub[TUPLELEN];

/* discovered a bug, if a deque_message is sent while the communication is
between two servers, then there is confusion and a proper match doesn't
happen.  had to disable this by returning from this routine without doing
anything.  Now things work fine.  Need to investigate this though
*/

/* an effort to fix a bug - ram - 07/17/90 */
 return;

        buf = convert_tuple_to_string(Ttuple);
        DEBUG(("send_dequeue_msg: (%s) \n", buf));
        strcpy(fub, DEQUEUESTRING);
        strcat(fub, BLANKSTRING);
        strcat(fub, buf);
        free(buf);
        send_broadcast_msg(fub);
```

```
}

/*
 *      send the matched tuple from the local tuple space to the requesting
 *      process at the socket from
 */

void
send_matched_tuple( Ttuple )
TSPOINTER Ttuple;
{
        char *buf;

        buf = convert_tuple_to_string(Ttuple);
        DEBUG(("send_matched_tuple: (%s) \n", buf));

        if(sendto(sock, buf, BUFLEN, 0, &from, fromsize) < 0)
                syslog(LOG_ERR, "listener: sending packet: %m");
        free(buf);
}

show_data()
{
/*      dummy function
*/
}
```