Comparison of Unix Communication Facilities Used In Linda

Chuck Schumann, Kenneth Landry,
and James D. Arthur

TR 92-06

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

March 6, 1992

Comparison of Unix Communication Facilities Used in Linda

Chuck Schumann
Kenneth Landry
James D. Arthur

Computer Science Department
Virginia Polytechnic Institute and State University

Comparison of Unix Communication Facilities Used in Linda

Chuck Schumann
Kenneth Landry
James D. Arthur

Abstract

The Linda system supports the specification of parallel computation, realized through concurrent processes that communicate through a shared dataspace called Tuple Space. This report presents the results of an investigative effort that focuses on a first step toward providing a distributed framework for Linda processes. In particular, we discuss the restructuring of the kernel "process" to support tuple space access through UNIX socket calls rather than through shared memory primitives based on semaphore usage. A description of the restructured system and the rationale for such restructuring is presented first. Most intriguing, however, are the latter sections that discuss the ramifications and insights gained from our particular approach to system redesign, i.e. the unnecessary serialization of tuple space access, redundant memory copies, being victimized by the UNIX scheduler.

1.0 Introduction and Background

As the computing demands in the 90's increase, so must the computing power of machines. Parallel machines and parallel programming environments are being used more as the computational limits of uniprocessor machines and conventional serial programming are being taxed. Linda is one such parallel programming environment offering the ability to write parallel programs explicitly. Linda is a coordination language rather than a complete parallel programming language. A coordination language [CARRI89 and ZENIT90] provides the primitives to create processes as well as coordinate communication among processes. By virtue of being a coordination language, Linda primitives can be introduced into many base programming languages. The original implementation, using C as its base language, exploits a preprocessor approach which transforms Linda operations into C source code. Implementing the Linda primitives in this way is especially useful when parallel systems need to be developed in multiple languages. Linda has been embedded in a wide variety of other languages, e.g. C++, Fortran, various Lisps, PostScript, Joyce, Modula-2 and soon Ada [GELER90].

The Linda approach supports process creation and inter-communication through a shared data/process repository called Tuple Space (TS). Linda provides operations to generate data tuples (`out`), to read data tuples (`rd`), and to remove them from TS (`in`). Tuple Space not only contains data tuples but also process tuples (created with the `eval` operation) which are often called "live tuples." These process tuples are instantiated and are eventually replaced by a data tuple when the instantiated process finishes executing. TS can also be used to share data structures among processes and synchronize the order of actions that processes perform.

A number of Linda applications have been written either as examples of Linda's expressivity or as "real" programs used in industry. These applications include DNA sequencing, finding primes and process lattices [CARRI88]. Moreover, Cogent has designed a Unix operating system for its XTM machine that reflects the Linda paradigm [LELER90] and has tested the system by developing a Linda-based parallel window server called PIX [LELER88].

When one discusses the Linda paradigm, two characteristics are often touted: its ease of use and its portability. From a conceptual standpoint, parallel programming within the

Linda framework is intentionally high level, which is exactly why it is so flexible and powerful. Not so surprising, however, this high-level approach is at the root of Linda's greatest criticism - its questionable performance [DAVID89]. Linda does exhibit acceptable performance on both shared and distributed memory parallel (MIMD) machines [BJORN88, BJORN89 and CARRI87], performance also suffers for full-scale local area network platforms. Since a network version of Linda is a primary goal of the Linda group at Virginia Tech, performance is critical.

The original design of Linda uses the shared memory model for the communication medium. In this approach, the kernel routines are a part of each Linda process and TS resides in shared memory. Synchronization of data access is achieved through the use of semaphores. Many Linda processes may be concurrently executing kernel code but access to TS data is serialized to insure data integrity. This approach to communication is efficient, however, care must be taken in the placement of the semaphores (for synchronization) in order to insure correct serial access. An even greater disadvantage in using shared memory is that it lacks network support by the UNIX operating system. Since the placement of Linda on a network platform is one of our primary goals, this deficiency of the shared memory approach is unacceptable.

Our current implementation of Linda uses a socket-based approach to communication between the kernel and Linda processes. In this design, the kernel is stripped out of the Linda processes and implemented as single, separate process. By removing the kernel routines and TS from the Linda process and instantiating them as a separate, but single, process the Linda processes now send and receive data from the kernel process by sending structured messages through UNIX sockets. In this model, data is sent to the kernel with the particular service to be performed prepended to the data. The kernel reads the service requested and the data on which to perform this service and then continues just as if the Linda process had made a function call to the kernel routine (as in the shared memory approach). Operations on TS behave the same as in the share memory model, the only difference in the two communication models is in the location of TS.

2.0 Linda and Sockets

Given that a network based Linda environment precludes placing tuple space in shared memory, new issues are brought to light. Should TS be distributed across several nodes or located on a single host? If it is distributed, how will TS integrity be maintained? What mechanisms can be employed to send to and receive from TS? With distributed processes trying to simultaneous access TS, should there be a fairness policy in the system?

The approach we have taken is to move toward the networked implementation by establishing a series of redesigns upon which subsequent research and development can be based. Each redesign should possess several properties in order to be useful. It should:

- be a functional Linda system,
- be a test bed for studying various relationships between design and runtime characteristics of that implementation,
- be a simulation base for future developments,
- portable between BSD and USG systems,
- have minimal coupling in the method of inter-process communication; we expect that differing communication facilities will be used in subsequent redesigns,
- minimize changes to the previous working system. Once a redesign is stable then minor changes could be added before the next major redesign, and
- not include changes to the compiler/analyzer parts of the environment. One of our objectives is to maintain backward compatibility of the runtime system with the clc pre-processor.

The first redesign is to the form of a Linda environment that removed all direct access to TS from the Linda program, i.e. removing the kernel and TS primitives from the program support library and replacing the shared memory paradigm with another communications structure. The new environment is designed around the *client-server* model using UNIX sockets as the link between the Linda processes (clients) and the TS kernel (server). Using this approach allows several features of UNIX to be exploited:

- inter-process communications with sockets,
- use of the `fork` as the primary method of implementing the Linda `eval` operation, and
- liberty to experiment with different implementations of the Linda `eval` operation , e.g. using Remote Procedure Calls (RPCs) or system calls to simulate multi-node distribution.

Through UNIX sockets the programmer is able to use a file descriptor and the `read` and `write` functions to send/receive data to/from independent processes in a manner resembling file I/O. Effectively, the network is transparent to the programmer and the corresponding process is treated like a binary file available for reading or writing. However, there is a distinction. The major distinction, however, is that socket-based communication is accomplished through links and subsequently does not support forward or backward seeks. Data, once written, can not be read back. Likewise, data read from the socket can not be read a second time by rewinding the socket.

Sockets are very useful for designing *client-server* systems. In our environment there exists a repository of data (TS) and a body of code (the kernel) that manipulates this data for the many Linda processes which might be executing. A natural division of labor establishes the kernel and tuple space as a server and allows the Linda processes as clients requesting TS services from the server. This relationship is consistent with the original Linda design where the run-time kernel is supported by library calls and the compiler generates the necessary code and data structures to call the needed kernel service.

As mentioned earlier, on each redesign we desire to limit as much as possible the number of significant changes to the previous system. In this first redesign, the major difference is the removal of shared memory from the run-time system. The implications being that all the TS storage structures can remain unchanged, but that the kernel code must be heavily modified. The original implementation is intended for shared memory machines and allows the programmer to take advantage of particular architectural designs to achieve efficient execution. In particular, the use of pointers in passing data was more the rule than the exception; structures containing pointers to any piece of possibly useful data were employed. This presents a specific challenge because a structured binary data stream (no pointers) is required for the *client-server* model to be practical. This constraint is pragmatic because the kernel (server) and the Linda processes (clients) has separate address spaces, and therefore, cannot pass pointers back and forth. Subsequently, the de-referencing of pointers and imposing strict formatting conventions on the transferal of structures also becomes a major issue in the first redesign. The contrast between the liberal use of pointers for information storage and the format used for the stream socket can be seen in Figures 1 and 2, respectively.

The attempt to change as little as possible in order to implement the socket communications model encourages the continued use of data structures proven effective in the original system. The socket based system is designed to leave all run-time data structures untouched. This implies a conversion from memory organized like that shown in Figure 1 into a binary stream (Figure 2) and then back to the original structure by the receiver.

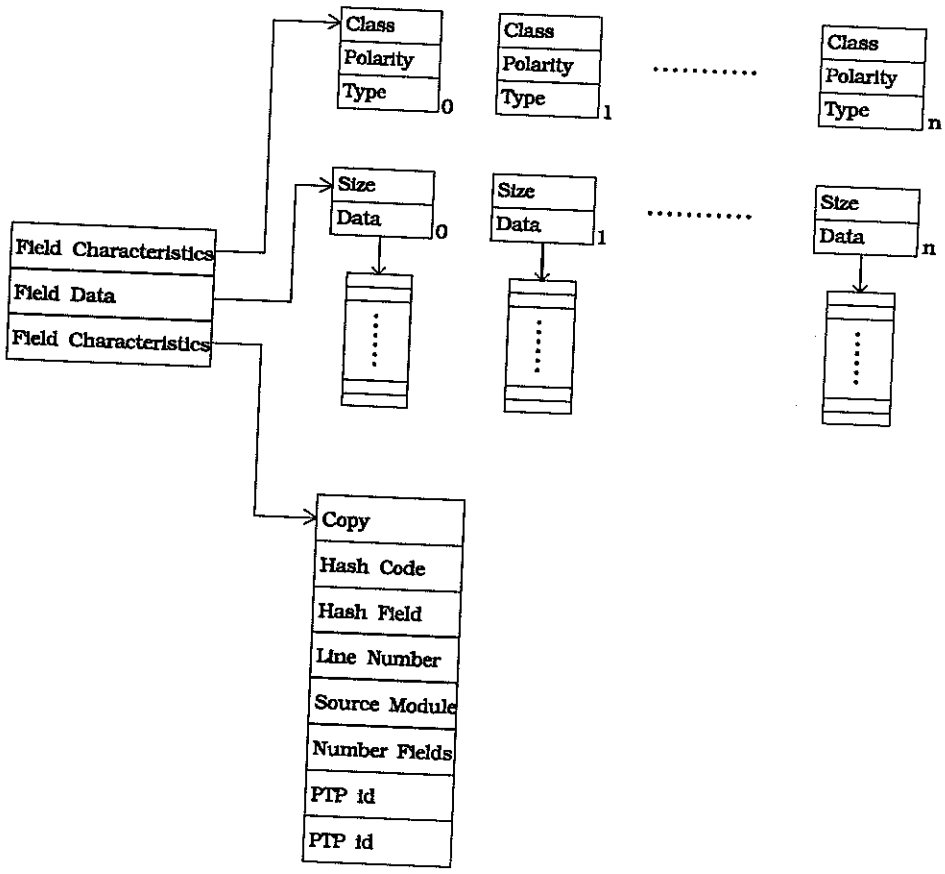


Figure 1. Data structure for a tuple.

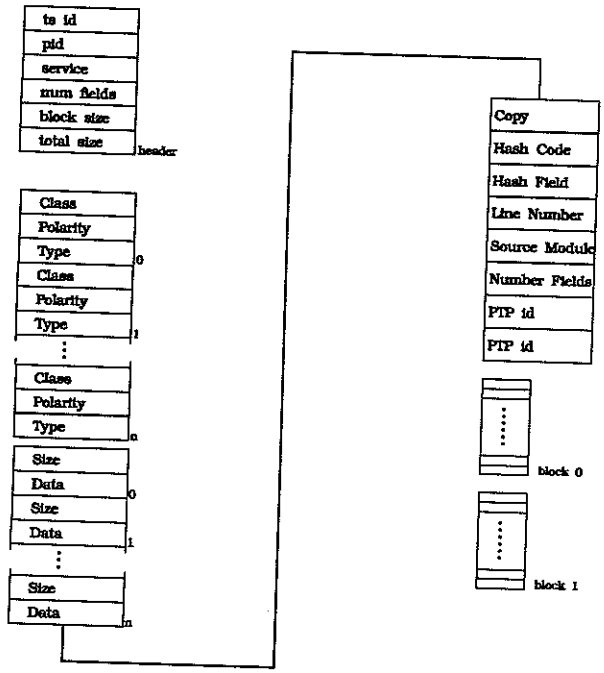


Figure 2. Binary stream for a tuple in a socket .

Figure 3 shows the transformation that a tuple takes in its path between a Linda process and the kernel.

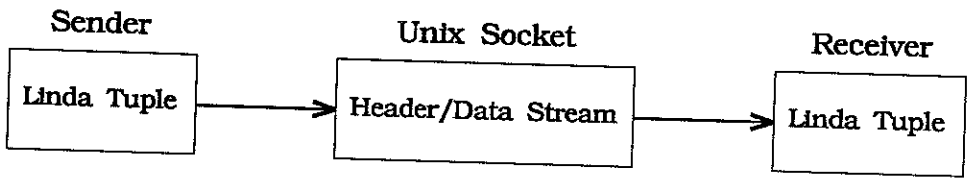


Figure 3. Tuple transformation when transferred between a Linda process and the kernel.

The header block prepended to the data stream is necessary for two reasons. First, in the original system Linda processes can designate the kernel service to be performed by making a direct function call to the supporting library routine. In the new system, the header block contains an identifier indicating the service requested. Second, the number of fields in the tuple must be known before the receiver can properly convert the data stream into the correct structure which also must be stored in the header for the new system. Other information being currently placed in the header is for debugging and run-time efficiency.

Choosing Unix sockets as the communication medium has several distinct advantages.

Portability. As stated earlier, we wish to insure portability between BSD and USG which is achieved through the use of sockets.

Reliability. The sockets used are stream sockets which are guaranteed to be reliable where as data grams sockets, for example, are not.

Network Support. With sockets, all the networking details are, for the most part, handled by the sockets library routines. The only details the programmer needs to address are how to initiate the client-server connection.

Simplification of Linda kernel code. Because TS has been moved from shared memory into a distinct kernel process, access to TS data no longer requires synchronization with semaphores. This provides for a more maintainable and robust system with respect to future extensions.

Reduction in Linda program size. Since the kernel contains all TS data and management code each Linda process is smaller. This makes for faster process creation and reduced disk swapping. As a result only one kernel exist, whereas originally each process had its own copy of the kernel code and shared only the data in TS and the semaphores controlling its access.

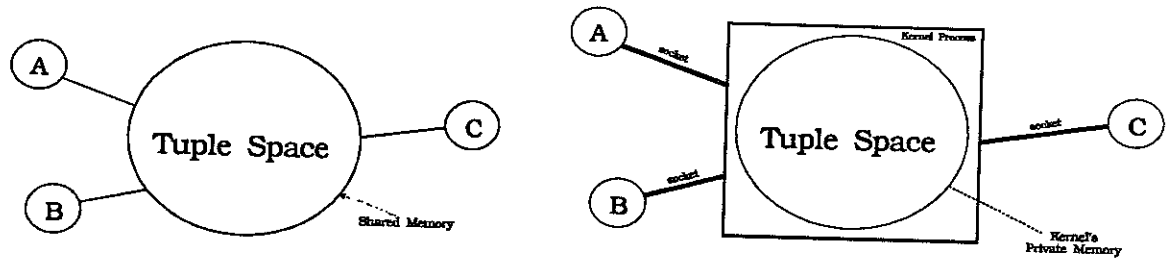
Easy prototyping. Inter-process communications can be easily implemented using sockets. Using sockets provided a proven inter-process communication model that could be easily implemented and easily replaced allowing other mediums to be tested.

The use of sockets in implementing the communication medium between the Linda kernel and Linda processes has a number of advantages as seen above. However, This approach does incur certain penalties which are described in the following sections.

3.0 Serialization of TS Access

The problem of unnecessary serialization is a direct result of centralizing the kernel routines and TS into a single process. That is by implementing TS using private memory in a separate kernel process and placing all kernel services in the kernel process, any use of Linda operations (and hence TS access) is necessarily serialized (see Figure 4). In this model all processes requesting kernel services send their requests to the kernel via UNIX sockets and the wait. These blocked processes continue to queue up until the kernel is scheduled to run by the UNIX operation system, at which time, they are processed one at

a time until they have all been serviced or until the UNIX scheduler decides to suspend the kernel process. Although this course grain serialization is operationally equivalent to the shared memory model on a uniprocessor (in terms of total kernel service time), it compares unfavorable on multiprocessor machines.



(a) Share memory approach (b) Socket approach

Figure 4. Serialization of TS access.

This serialization is also disadvantageous in that many concurrent kernel services do not require mutual exclusion. Since TS is partitioned into regions based on the communication patterns of the Linda program (determined at compile time), in most cases the need for synchronization is at a minimum (i.e. each region of TS can be accessed at the same time). The current socket implementation does not exploit this fact.

4.0 Redundant Memory Copy

Redundant memory copy operations are inherent in the use of UNIX sockets. First of all, the socket `write/read` sequence requires several copy operations just to deliver data from a sender to the receiver process. In general, the sender calls the `write` function with a file (socket) descriptor. This transfers (copies) the data to a system buffer for that socket. The receiving process calls a "read" function whereby the data is transferred from the system buffer to private memory owned by the receiving process. In Linda, data being sent requires that it be preceded by a header block (approximately 32 bytes) telling the receiver what is being sent and how to reconstruct the correct tuple structure at the receiver end. Thus, for every service requested of the kernel, the tuple data (including the header) must be copied three times, in the case of a uniprocessor and four

times if communicating over a network. This compares poorly with the shared memory model which only requires that data for a tuple be copied once for a kernel operation.

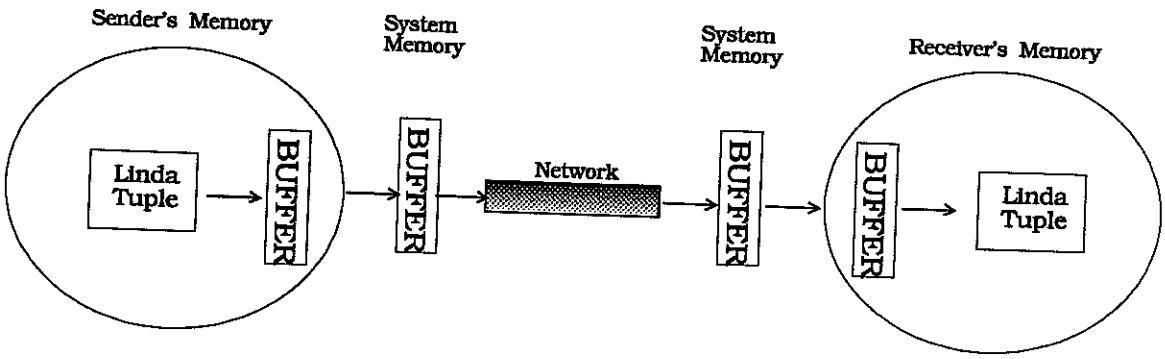


Figure 5. As a tuple moves between a process and the kernel, many buffer copies are made.

Figures 1 and 2, respectively, show the structure of the data used in the kernel and also the structure of data that is used in sending tuple information through the socket. Since the data structures in the Linda process and in the kernel were not modified in this redesign, all transactions with the kernel require a transformation from the Linda tuple structure shown in Figure 1 to a structured binary data stream show in Figure 2 before sending the tuple out the socket. The receiver must then reconstruct from the structured binary data stream. Each transformation (from Figure 1 to Firgure 2) requires a series of copy operations to de-reference the pointer fields and construct the tuple as a stream of contiguous bytes. The transformation has the same number of copy operations when the operation is done in reverse.

To summarize, the typical `rd` operation involves the following copying operations, none of which are needed in the original version:

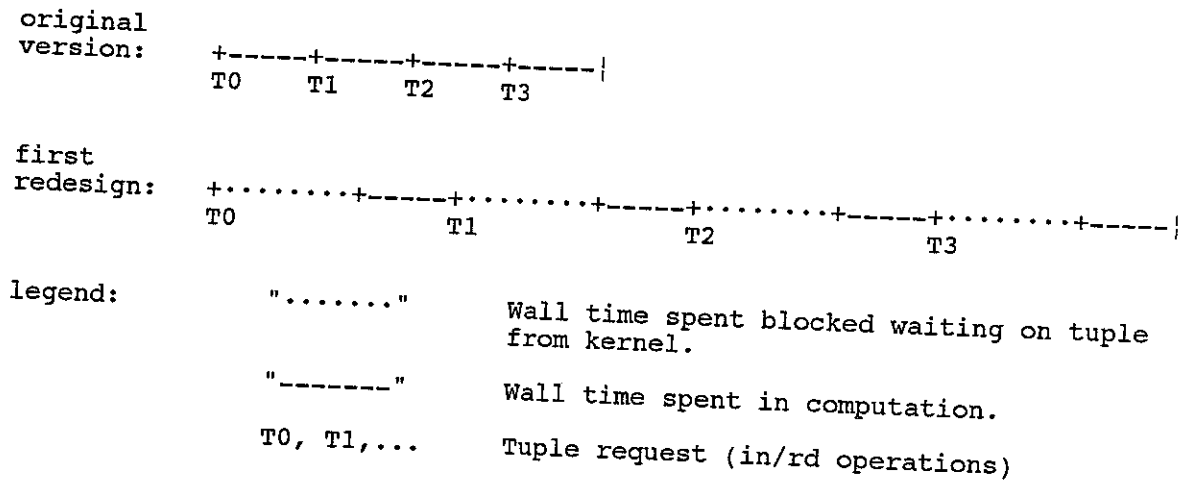
1. convert from the Linda tuple structure to stream buffer
2. the socket write call copies from stream buffer to system buffer
3. the sender copies data from system buffer out the network
4. the receiver node copies data into system buffer
5. socket read call copies data from system buffer to local stream buffer
6. receiver converts from stream buffer to the Linda tuple structure

Moreover, most of these copy operations have associated memory allocation (`malloc`) overhead at runtime to provide storage space for the buffers. Together this represents a considerable degree of additional memory manipulation to the system.

5.0 Victim of the Unix Scheduler

The last deficiency of the socket based model appears to be the most costly - the Linda parallel environment falling victim to the Unix scheduler. With the shared memory model, because each process effectively has its own copy of the kernel, calls to the appropriate kernel routine can be serviced during the calling process' own time slice. This is not the case when TS is a separate process and communication with the kernel occurs with Unix sockets.

To accomplish the same service request is in the shared memory version of Linda a process must wait for the kernel to execute before a matching tuple is returned. This minimally extends the completion time for any process communicating with the kernel by a product of the time slice and the number of processes executing. The time lines below illustrate this for a Linda process which acquires four tuples and performs some computations after each tuple arrives.



This time line shows that in the original system tuple requests are granted without any time delay allowing the process to continue with its computation. In the first redesign the

computation continues after the request is received by the kernel, the kernel finds a match and sends it to the process, and finally the process is unblocked and reads the tuple from the socket. Although this is a simple TS `rd` operation, it illustrates that at a minimum the Linda process must block for a complete cycle of time slices of all other competing processes before it can receive a tuple from TS. In the shared memory model, this time is not wasted since the entire kernel operation is performed under the scheduled time slice for the Linda process performing the operation and not by a centralized kernel process.

In order to determine the total effect of the UNIX scheduler on process completion, it should be compared to other delays with tuple operations. In the example above the socket overhead and the time spent by the kernel finding a match is considered negligible. Experience has shown that this time (overhead and matching) is indeed small in comparison to the time spent waiting for the kernel to execute. Also, as the number of processes in the system increases this time skew becomes more prominent.

6.0 Conclusion

The migration from a shared memory approach to a socket approach for kernel/process communication in Linda is necessitated by the ultimate goal of placing Linda on a network platform. The decision to use sockets as apposed to another method of communication is made primarily for portability since this is one of the few (reliable) communication primitives for networks. The redesign using a separate TS and kernel does pose problem however. In particular, the serialization of TS access, redundant memory copies, and the detrimental effect of the UNIX scheduler are all problems inherent in the use of a separate kernel process with socket communication. The recognition of these problems is not only significant for Linda but for parallel programming in general when separate processes are involved with socket communication.

REFERENCES

[BJORN88] R. Bjornson, N. Carriero, D. Gelernter and J. Leichter, "Linda, the Portable Parallel," *Research Report YALE/DCS/RR-520*, January 1988.

[BJORN89] R. Bjornson, N. Carriero, and D. Gelernter, "The Implementation and Perfomance of Hypercube Linda," *Research Report YALEU/DCS/RR-690*, March 1989.

[CARRI87] N. Carriero, "Implementation of Tuple Space Machines," *Research Report YALEU/DCS/RR-567* (PhD thesis), December 1987.

[CARRI88] N. Carriero and D. Gelernter, "Applications Experience with Linda," *Proc. ACM Symp. Parallel Programming*, July 1988.

[CARRI89] N. Carriero and D. Gelernter, "Coordination Languages and their Significance," *Yale Tech Report,* YALEU/DCS/RR-716, July 1989.

[DAVID89] C. Davidson, "Technical Correspondence on *Linda in Context,*", Communications of the ACM, Vol. 32, No. 10, October 89, pp.1249-1252.

[GELER90] D. Gelernter, "Ada-Linda: Motivation, Informal Description and Examples," *Yale Tech Report.*

[LELER88] Wm. Leler, "PIX, the latest NeWS," *Cogent Technical Report,* Cogent Research, November 1988.

[LELER90] Wm. Leler, "Linda Meets Unix," *IEEE Computer*, February 1990, pp.43 - 54.

[ZENIT90] S. E. Zenith, "Linda Coordination Language; subsystem kernel architecture (on transputers)," *Research Report YALEU/DCS/RR-794*, May 1990.