Communicating Process Architectures 2000
P.H. Welch and A.W.P. Bakkers (Eds.)
IOS Press, 2000

A Comparison of Linda Implementations in Java

George Wells[a], Alan Chalmers[b] and Peter Clayton[a]
[a]*Department of Computer Science, Rhodes University, Grahamstown, South Africa*
[b]*Department of Computer Science, University of Bristol, Bristol, U.K.*
`G.Wells@ru.ac.za`

Abstract. This paper describes the implementation of an extended version of Linda in Java. The extensions have been made with a view to increasing the efficiency of the underlying communication mechanisms and the flexibility with which data may be accessed, particularly in the area of distributed multimedia applications. The system is compared with two other recent implementations of Linda for Java: JavaSpaces from Sun Microsystems, and TSpaces from IBM. The comparison is performed both qualitatively, comparing and contrasting the features of the systems, and quantitatively, using a simple communication benchmark program and a ray-tracing program to assess the performance and scalability of the different systems for networks of workstations.

1. Introduction

This paper describes an extended version of Linda[1], called *eLinda*, developed by the authors using Java[2] [1]. This system is also compared with two other recent Linda implementations in Java: JavaSpaces from Sun Microsystems [2], and TSpaces from IBM [3].

The Linda model was first proposed in the 1980's by David Gelernter [4]. This approach to distributed and parallel programming offers a number of advantages as it is based on a shared memory paradigm with a small set of simple operations to access shared data. In addition, the shared data space effectively decouples communication between processes, both with respect to time (communication is asynchronous) and location (communicating processes do not need to be aware of each other's identity or location in a multi-processor or networked system). This inherent simplicity and the decoupling of processes offer a number of benefits over systems based on message-passing, remote method invocation, etc. However, Linda has been criticised for poor performance, and until recently there has been little commercial interest in it. This has now changed with the release of JavaSpaces and TSpaces.

The extensions introduced in eLinda have been designed with a view to making some of the underlying communication issues more explicit, thus providing the programmer with a greater level of control of the communication in an attempt to address some of the efficiency concerns. JavaSpaces and TSpaces also include some extensions and differences to the original Linda model as proposed by Gelernter, aimed mainly at support for commercial applications.

---

[1] Linda is a registered trademark of Scientific Computing Associates.

[2] Java is a registered trademark of Sun Microsystems, Inc.

The rest of this paper consists of a brief introduction to the Linda programming model, followed by further details of the extensions embodied in eLinda and the motivation behind them. The implementation of eLinda in Java is also discussed. Section four presents a description of the features of JavaSpaces and TSpaces. This is then followed by results comparing the performance of the three Linda systems for a simple communication benchmark and also for a ray-tracing application.

2. The Linda Coordination Language

Linda is a coordination language for parallel and distributed processing, providing a communication mechanism based on a logically shared memory space called *tuple space*. On a shared memory multi-processor system the tuple space may actually be shared, but on distributed memory systems (such as the network of workstations used in the eLinda project) it may be distributed among the processing nodes. Whatever implementation strategy is used, the tuple space is accessed using associative addressing to specify the required data objects, stored as *tuples*. An example of a tuple with three fields is `("point", 12, 67)`, where 12 and 67 are the *x* and *y* coordinates of the point represented by this tuple.

2.1 Language Overview

As a coordination language, Linda is designed to be coupled with a sequential programming language (called the host language). The host language used in this work is Java. Linda effectively provides a library with a small set of operations that may be used to place tuples into tuple space (`out`) and to retrieve tuples from tuple space (`in` which removes the tuple, and `rd` which returns a copy of the tuple, leaving the tuple in tuple space). The latter two operations also have predicate forms (`inp` and `rdp`) which do not block if the required tuple is not present, but return immediately with an indication of failure. The specification of the tuple to be retrieved makes use of an associative matching technique whereby a subset of the fields in the tuple have their values specified and these are used to locate a matching tuple in the tuple space. For example, if a point such as that in the example above was required then the following operation would retrieve it: `in("point", ?x, ?y)`. The tuple specification here, `("point", ?x, ?y)`, is referred to as an *anti-tuple*. Any tuple with the same number and type of fields and with the string `"point"` in the first position would match this request. When a successful match is made the variables `x` and `y` are assigned the values of the corresponding fields of the matching tuple.

The original Linda model also provides for dynamic process creation by means of the `eval` operation. This is not an essential part of the paradigm, and, in fact, it has been shown that `eval` may be implemented in terms of the other operations with some support from a preprocessor [5]. Accordingly, it will not be considered any further here (none of the three Linda systems discussed in this paper include the `eval` operation). Further details of the Linda model of distributed programming may be found in [6].

2.2 Implementation Strategies

On distributed memory systems there are many possible strategies for implementing the shared tuple space (although such issues are usually transparent to the programmers and users of the system) [4] [7] [8]. Common approaches include centralised systems (all tuples are stored on a single processing node), hashing systems (the contents of tuples are used to allocate them to particular processors), and partitioned systems (tuples with a common structure are allocated to a specific processor). Another approach, and that adopted in eLinda, is to have the tuple space fully distributed, such that any tuple may reside on any processing node. In the eLinda system, this is combined with internal partitioning of the tuple space so that tuples with distinct structures are stored separately to improve the efficiency of retrieving tuples. Both JavaSpaces and TSpaces use the centralised approach.

2.3 Advantages and Disadvantages of Linda

The small set of operations, the associative retrieval mechanism and the shared tuple space all combine to provide a very useful simplicity and flexibility for constructing distributed applications. On the other hand, a criticism of Linda has been that it is, at worst, inefficient, and, at best, subject to unpredictable performance [9], as the simplicity of the model hides the underlying complexity of the data sharing and communication required. In order to try to overcome some of these problems, while retaining much of the essential simplicity of the Linda approach, some extensions to the original Linda model are proposed in eLinda.

3. The eLinda System

The extensions that have been made in the eLinda system provide the programmer with a somewhat higher level of control over the communication mechanisms than is normally the case. This has been done with a view to improving the performance, and to making the performance issues more explicit (aiding predictability).

Usually some form of program source preprocessor is used with Linda systems to translate the Linda operations into the actual forms used by the host language. Currently a preprocessor is not provided for eLinda and so all interaction with the system takes place using the standard Java function calling and parameter passing mechanisms. However, the examples given below all make use of a simplified syntax (referred to as the *ideal syntax*) such as might be supported by a preprocessor.

3.1 The Extensions in eLinda

The extensions in the eLinda system take three forms. The first is an additional form of output operation, which provides the programmer with a greater degree of control of the underlying network communication. The second is a mechanism to allow customised searching algorithms to be integrated into the eLinda system efficiently. Lastly, support for multimedia data types has been added. Each of these topics will now be discussed in further detail.

3.1.1 Explicit Broadcast Communication

Two types of output operation are now provided to reflect explicitly a choice of optimised internal tuple space communication strategies. These are a "point-to-point" mechanism (using non-replicated data) and a "broadcast" mechanism (using replicated data). This contrasts with the existing Linda mechanism where data is written to tuple space using a single instruction (`out`), but is then read using one of two methods (`in` or `rd`, or the equivalent predicate forms). In effect, the use of `in` implies a form of exclusive point-to-point communication, in that one process places a tuple into tuple space, which is then removed by another. Similarly, the use of `rd` suggests a form of shared, or broadcast (read-only), communication, as several processes may obtain copies of the tuple.

To allow the programmer to take advantage of this behaviour and the fact that the tuple space is distributed across all the processors, a new output operation, `wr`, has been added in eLinda. This operation broadcasts the tuple throughout the processor network, whereas `out` simply places a single tuple in the local tuple space. These mechanisms provide the programmer with the necessary facilities to express shared, read-only access to data (`wr–rd`), or exclusive, delete/modify access (`out–in`). It should be noted that this usage is not enforced by the system. For example, it may occasionally be necessary to update data that is otherwise shared in a read-only fashion. In such a case a tuple would be broadcast using `wr`, accessed using `rd`, and then removed for updating using `in`. This would result in a performance penalty, as all the broadcast tuples would have to be deleted. Similarly, `rd` may be used to retrieve a tuple placed in tuple space using `out`, but a search of all the processors involved in the computation may be required to locate it.

3.1.2 The Programmable Matching Engine

The second extension is to provide a *programmable matching engine* (PME) for the retrieval of tuples, allowing the use of more flexible criteria for the associative addressing of tuples. For example, in dealing with numeric data one might require a tuple which has a value that is “close to” some specified value (possibly using fuzzy set membership functions). In a graphical context, with tuples representing the objects in an image, one might require a tuple corresponding to an object lying within a specified area of the image. Such queries can usually be expressed using the standard Linda associative matching methods, but will generally be quite inefficient. For example, the application might have to retrieve all tuples of the required type, select one of interest and then return the rest to tuple space. When the tuple space is fully distributed, as it is in the eLinda system, searching for a tuple may involve accessing the sections held on all the processors in parallel. This problem is handled efficiently in eLinda by distributing the matching engine so that

Table 1: The `ProgrammableMatcher` Interface

```
public interface ProgrammableMatcher
    { /** This function compares one anti-tuple with a list of tuples.
        * This is needed for all operations, but particularly for non-
        * blocking operations (i.e. rdp and inp).
        */
      public Tuple matchList (AntiTuple a, TupleIterator t)
          throws MatcherException;

      /** This function compares one anti-tuple with one tuple.
        * This is needed only for blocking operations (i.e. in and rd)
        * where tuples may come in one at a time (of course, it can
        * also be used by the matchList function).
        * If a matcher is never to be used in a blocking operation this
        * can simply return false.
        */
      public boolean match (AntiTuple a, Tuple t)
          throws MatcherException;

    } // interface ProgrammableMatcher
```

network traffic is minimised. For example, in searching for the "largest" tuple, each section of the tuple space would be searched locally for the largest tuple and that returned to the originating process, which would then select the largest of all the replies received. The syntax currently used to specify the matcher which is to be used is `op.matcher`, where `op` is one of the eLinda input operations and `matcher` specifies the customised matching routine to be used[3]. In addition, the extended syntax `?=` is used to identify which field (or fields) is to be used by the non-standard matching routine. For example, the command `in.maximum("point", ?=x, ?y)` specifies an `in` operation using a matcher which will find the tuple with the maximum *x* value. Omitting the matcher specification causes the system to use the "standard" technique of matching for strict equality.

Such customised matchers are provided by writing a class that implements a Java interface specifying the necessary methods that must be provided for a matcher. The interface for the programmable matching engine is shown in Table 1.

Writing such a matcher is not a trivial operation. Various library calls are provided to allow the matcher to interact directly with the eLinda system (e.g. retrieving tuples from tuple space, replacing unwanted tuples, deleting local and remote tuples, broadcasting requests to other processors and subsequently retrieving the results of such requests, etc.). However, this interaction allows the programmer to access the tuples in tuple space at a lower level of abstraction than usual, and care needs to be taken to preserve the semantics of Linda tuple retrieval operations. As an indication of the complexity of writing a customised matching algorithm, a matcher to find the closest numeric value for a particular field (or fields) takes approximately 190 lines of (extensively commented) Java code. Another matcher, that returns a total of numeric tuple fields, is written in 175 lines of code.

The programmable matching engine concept has some aspects in common with the recent adoption of *mobile agents* [10]. Effectively, a customised matcher written for the eLinda programmable matching engine is a form of mobile agent that is distributed on a network to find a matching tuple (or tuples). This provides the same performance advantage as for mobile agents, namely that the need to move large amounts of data across the network is minimised by doing the processing where the data is to be found rather than centrally.

3.1.3 Support for Multimedia

The third distinctive feature of eLinda is its support for multimedia data. Tuples in eLinda may contain any of the eight primitive data types supported by Java (`int`, `char`, `double`, `float`, `byte`, `short`, `long` and `boolean`) as well as standard Java `String` objects. Furthermore, any Java object[4] may be added to a tuple (making use of the polymorphism supported by Java), although this limits the type checking that can be performed by the eLinda system. In this way the eLinda system attempts to provide the maximum possible functionality for general-purpose applications.

To this basic functionality has been added the ability to use a `MultiMediaResource` object in an eLinda tuple. This class acts as a wrapper to the underlying Java Media Framework (JMF) multimedia resource. In particular the implementation of the `MultiMediaResource` class provides support (transparent to the application programmer) for any necessary buffering of data, fetching or streaming of multimedia data across the network, etc. Multimedia applications are not considered further in this paper. Further details of this aspect of eLinda may be found in [1].

---

[3] Again, this is the *ideal syntax*, not the actual Java code.

[4] The only restriction is that the object must be serializable.

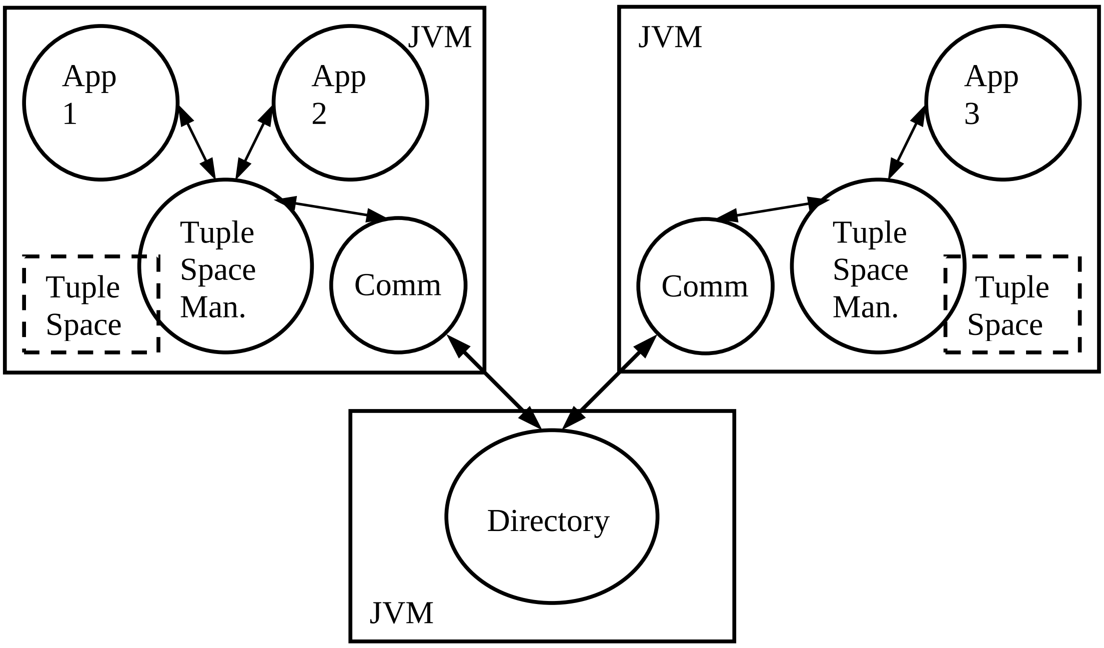

Figure 1: The Structure of the eLinda System

3.2 The Implementation of eLinda on a Network of Workstations

A version of eLinda has been implemented using Java. The current structure of the eLinda system (simplified somewhat) is shown diagrammatically in Figure 1, which illustrates a relatively small configuration with only three application processes. As indicated in the diagram, parts of the applications may be multithreaded, and in general there may be many more application threads/processes than shown here. The Tuple Space Managers (TSMs) are implemented by a Java class, which is responsible for controlling access to the tuple space. The “Comm” components shown in the diagram are lightweight threads that are responsible for handling the communication and buffering requirements.

The diagram illustrates the system in terms of Java Virtual Machines (JVMs). These may be running on separate network hosts or may be allocated to common hosts in any desired configuration, thus allowing for simple load-balancing. The communication between JVMs makes use of the TCP/IP network protocol, implemented by the Java `Socket` class. Within an individual JVM, communication between separate threads of execution is implemented using shared buffer data structures to decouple the execution of the threads.

The “Directory” process is used to direct network messages between the cooperating processes. This helps to centralise the communication and network configuration issues—each TSM needs to connect to just one Directory handler. There may be many Directory handlers in a large network. Each Directory is fully interconnected with all others, giving a maximum of three network hops to send a message from any part of the system to any other.

4. Other Linda Implementations

As has already been mentioned, both Sun Microsystems and IBM have recently released Linda implementations in Java. This section describes their features and compares them with eLinda and the original Yale Linda model. It is worth noting that both JavaSpaces and TSpaces make use of the object-oriented features of Java (inheritance, polymorphism and interfaces) to negate the need for a preprocessor.

4.1 JavaSpaces

JavaSpaces [2] is a complex product and relies heavily on a number of other technologies from Sun. It forms part of the Jini system for networking heterogeneous systems [11] and so makes extensive use of the Jini API. Network support is provided by the Java RMI (Remote Method Invocation) protocol. Furthermore distribution of classes to clients is handled by the HTTP protocol. This means that before a JavaSpaces application can be started the following list of services must be running:

- an HTTP server (a minimal one is provided with the Jini/JavaSpaces release)
- an RMI activation server (part of the standard RMI software bundled with Java)
- a Jini lookup service (alternatively the RMI registry service can be used, but this is discouraged)
- a Jini transaction manager
- a JavaSpaces server

Most of these services (and any application programs) also require extensive setting of command line parameters, further adding to the overall complexity of using JavaSpaces. Applications are also required to run a security manager, whether security checking is required or not.

JavaSpaces supports the following operations (the names differ from the original names used by Yale, but essentially the same set of functions is provided): `write` (output), `read` (non-destructive input) and `take` (destructive input), and also predicate input forms: `readIfExists` and `takeIfExists`. Tuples are created by the programmer from classes that implement the Jini `Entry` interface, and only public fields that refer to objects are considered.

Tuples are transmitted across the network using serialisation, however JavaSpaces uses a non-standard method of serialisation in that only *public* fields of classes are serialised. Furthermore, multiple references to the same object cause multiple copies to be serialised. Matching of tuples with anti-tuples (templates) is done using byte-level comparisons of the data, not the conventional `.equals()` method. Matching can make use of object-oriented polymorphism for matching sub-types of a class.

The Yale Linda model is extended in JavaSpaces to provide support for *transactions* (a number of tuple space operations can be grouped into a single transaction and rolled back if any one step cannot be completed successfully), and *leases* (tuples can be given an expiry date after which they will automatically be removed from the tuple space). Both of these extensions are relevant to commercial software, but do not address any of the performance issues inherent in the Linda model. A centralised tuple storage approach is used and this may become a performance bottleneck in large systems.

4.2 TSpaces

TSpaces is a product of the IBM Alphaworks research division. In their words it is intended as “the common platform on which we build links to all system and application services” [12]. The TSpaces implementation is fairly simple and all that is required is that a single server process be running on the network. This server makes use of a textual configuration file and provides a useful web interface for monitoring and configuration purposes. Applications wishing to make use of the TSpaces service need only know the hostname of the computer running the server.

TSpaces offers a large number of operations. The basic Linda operations are provided (again different names are used): `write` (output), `read` (non-destructive, predicate input), `take` (destructive, predicate input), and non-predicate input forms (`waitToRead` and `waitToTake`). Note the rather confusing way in which the standard forms are predicates and the alternatives non-predicates. There is a `delete` operation that will simply delete a matching tuple from the tuple space. There are also operations for the input and output of multiple tuples: `scan`, `countN`, `consumingScan`, `deleteAll`, `multiWrite` and `multiUpdate`. Lastly there are operations that specify tuples by means of a "tuple ID" rather than the usual associative matching mechanisms: `update`, `readTupleById` and `deleteTupleById`.

TSpaces transports tuples across the network using standard Java object serialisation. Tuples are simply objects consist of a number of `Field` objects (or `FieldPS` objects which preserialise to a byte array to allow the server to work with unknown classes). The associative matching process then uses `Field` objects with a class type for a wildcard (e.g. `String.class`). This restricts tuples to containing objects, not primitives for matching purposes. Matching is performed using the standard `.equals()` method (and in some cases the `.compareTo()` method). Matching can be done using so-called *indexed tuples* (fields are named; ranges of values may be included; AND and OR operations are supported), and XML queries (Extensible Markup Language, a specification for structured documents produced by the World Wide Web Consortium [13]). Tuples may have an expiration time set, providing similar functionality to the lease mechanism in JavaSpaces, and there is transaction support. Furthermore, access control is provided (based on user names, passwords and groups).

New commands can also be added to the TSpaces system relatively easily. The rich set of operations and complex matching criteria provides a facility similar to the Programmable Matching Engine concept in eLinda. This still requires further investigation to assess the differences in functionality.

The current implementation of TSpaces also makes use of a centralised server model, which may become a performance bottleneck.

5. Performance Comparison

This section presents quantitative results for two sets of performance measurements: a simple communication benchmark, and a ray-tracing application. These results are compared and discussed for the three Linda systems.

5.1 Round-trip Communication and Setup Time

The first benchmark test that was run was a simple program where one process sent a time-stamped tuple to another process, which then returned the data to the original process. This allowed the minimal round-trip communication time to be measured (the average of 50 trips). This program was also used to measure the setup time for each of the Linda systems (i.e. the time taken from starting the program until the necessary connections to the tuple space had been established and communication could commence).

This program was run first on a single machine (Pentium II, with 160MB RAM) with version 1.2 of the Java SDK, under Windows '95, giving the results show below. The two setup times are for the two separate systems (client and server).

System	Average Round-trip Time (ms)	Startup Time (ms)
eLinda	797	660/650
TSpaces	74	380/390
JavaSpaces	176	11750/11480

The same program was then run between two machines across a 10MB Ethernet network. The first machine was the one described above, and the second was a Pentium with 64MB of memory, also running under Windows '95.

System	Average Round-trip Time (ms)	Startup Time (ms)[5]
eLinda	653	710/3460
TSpaces	54	380/3790
JavaSpaces	279	11590/17960

It is interesting to note from this data that both eLinda and TSpaces ran faster on a network than on a single machine. This indicates that the overheads of context-switching on the Microsoft Windows platform is relatively high. However, JavaSpaces performed considerably worse on a network than on a single machine, suggesting that the network

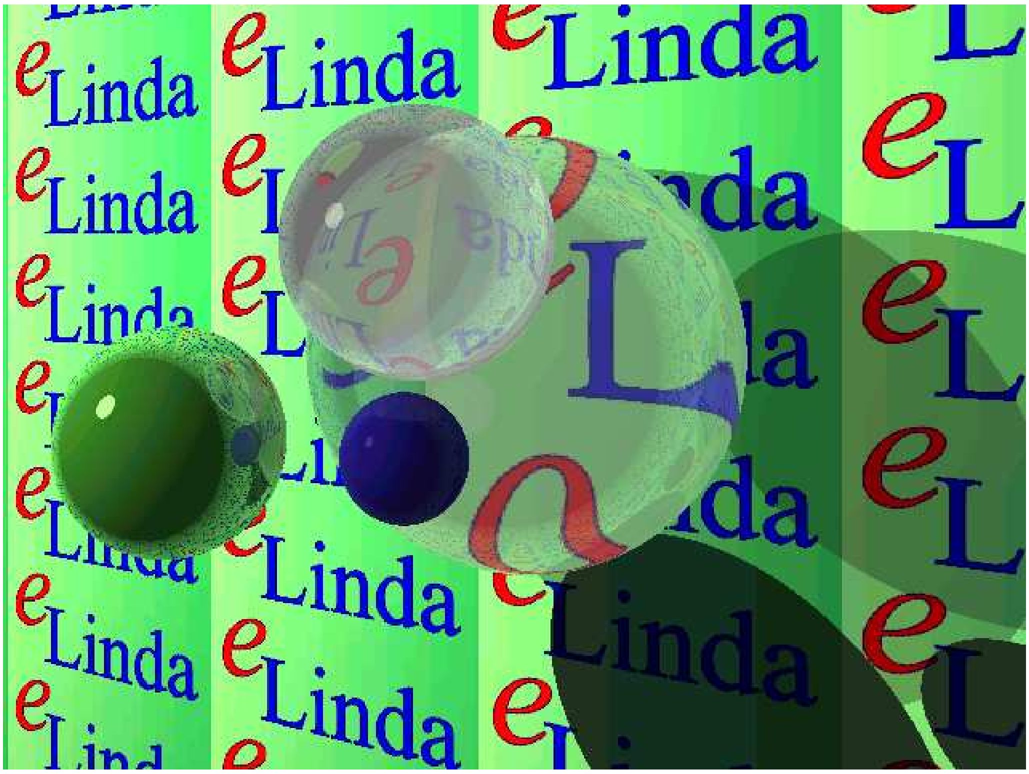

Figure 2: The Scene used in the Ray-tracing Program

[5] Startup on the remote machine required manual intervention to terminate a dial-up networking dialog, hence the extended times.

protocols used by JavaSpaces are not very efficient. This may be due to the fact that JavaSpaces makes use of a non-standard means of serialising objects for transmission across a network, whereas both eLinda and TSpaces use Java's standard object serialisation mechanisms.

While these results are interesting for comparing the three systems, they are focussed solely on communication with little data content, and no computational requirements—an unlikely situation in practice.

5.2 A Ray-Tracing Application

The JavaSpaces distribution includes an example ray-tracing program, using the JavaSpaces facilities. This was ported to the other two Linda systems and augmented to time the process of ray-tracing a simple image with four spheres, a textured plane and a single light source in a scene of 640x480 pixels. Figure 2 shows the rendered image. No attempts were made to optimise the original program—it was simply used as a means of measuring the relative efficiency of the three Linda systems.

This was again run on a single machine first (with the characteristics described above), with one worker process and then again with two. This test was also performed for various different sizes of the tasks (expressed in pixels). The 50x50 task size resulted in 130 tasks being created, 100x100 in 35 tasks and 200x200 in only 12 tasks. The following table gives the results of this test. The times reported are the total time taken to render the entire scene in each case.

System	**Task Size**	**One Worker (seconds)**	**Two Workers (seconds)**
eLinda	50x50	89.75	98.43
	100x100	82.17	89.86
	200x200	77.99	80.96
TSpaces	50x50	95.29	90.88
	100x100	81.51	80.36
	200x200	76.79	77.33
JavaSpaces	50x50	102.54	104.74
	100x100	86.30	87.49
	200x200	81.29	82.11

These tests were then repeated on a network of 133MHz Pentium PC's with 32MB of memory, running under Windows NT 4.0. These results were also collected using version 1.3 of the Java SDK which was recently released and which shows considerable performance improvements over version 1.2 (speedups of approximately four times for the ray-tracing application have been measured).

These results are graphed in Figures 3, 4 and 5 for a task size of 200x200 and various numbers of workers. Again the times reported are those for the rendering of the complete scene. However, in this case one machine was dedicated to running the "system" processes (tuple space server and any other required supporting software), another ran the controlling process, and the workers were each run on separate machines (so, for example, four workers corresponds to a total of six machines being used). Due to networking limitations a maximum of nine workers could be run in this environment. Furthermore, due to its memory requirements for the system processes, a maximum of eight workers could be used with JavaSpaces. Results were also obtained for 50x50 and 100x100 task sizes, and showed a similar pattern. It should be noted that there is some statistical variation in the results due to the fact that the work is distributed randomly to the workers—the results shown are the average of five runs.

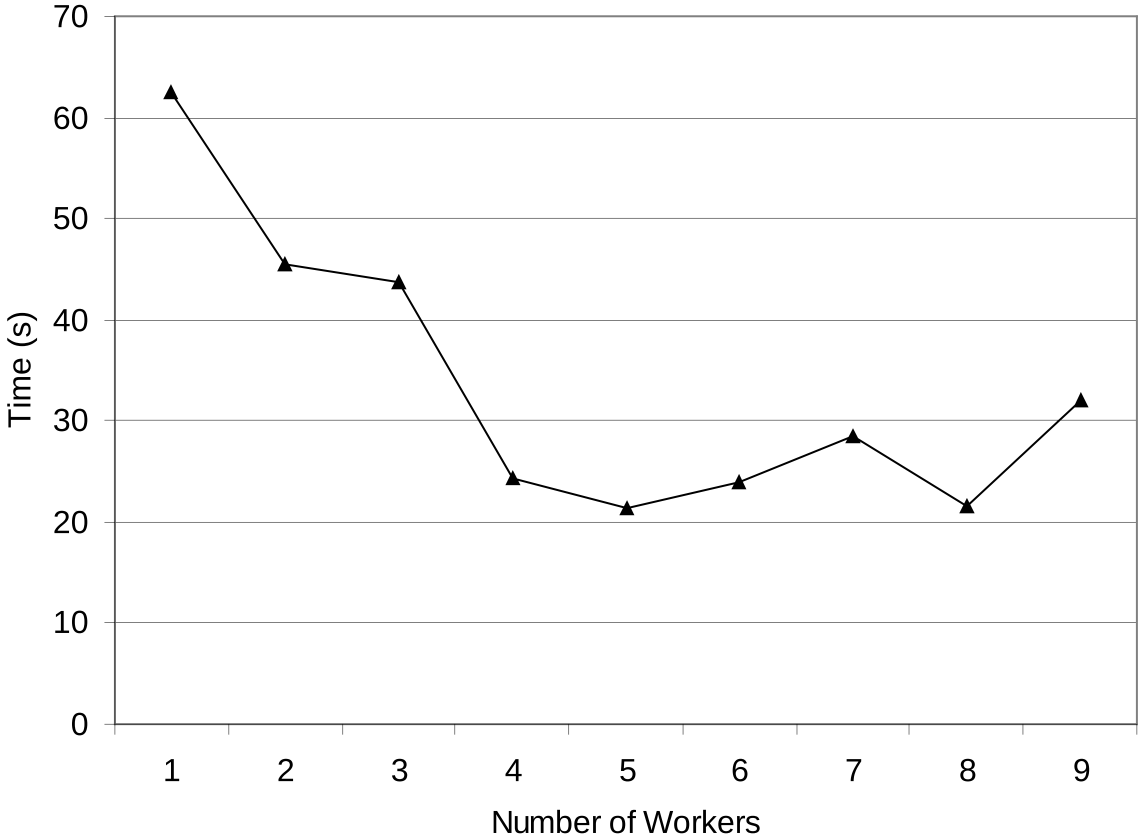

Figure 3: Ray-tracing Results for eLinda

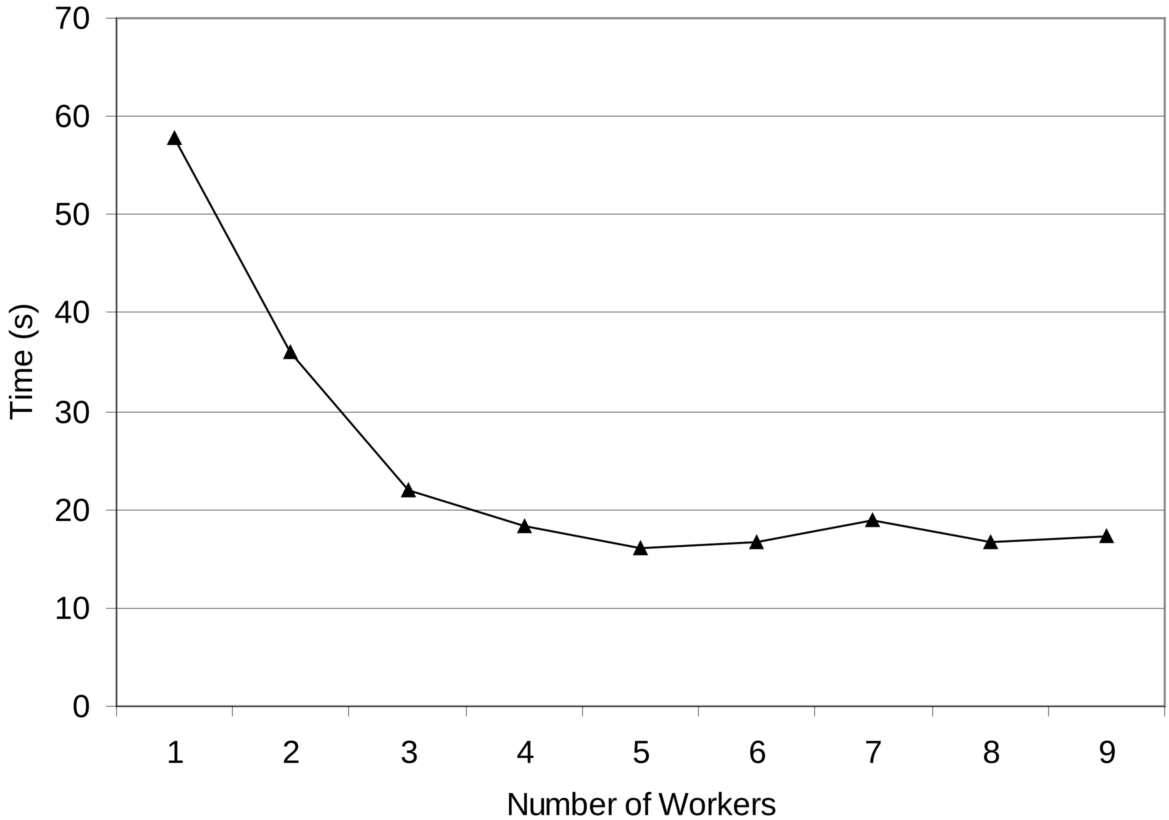

Figure 4: Ray-tracing Results for TSpaces

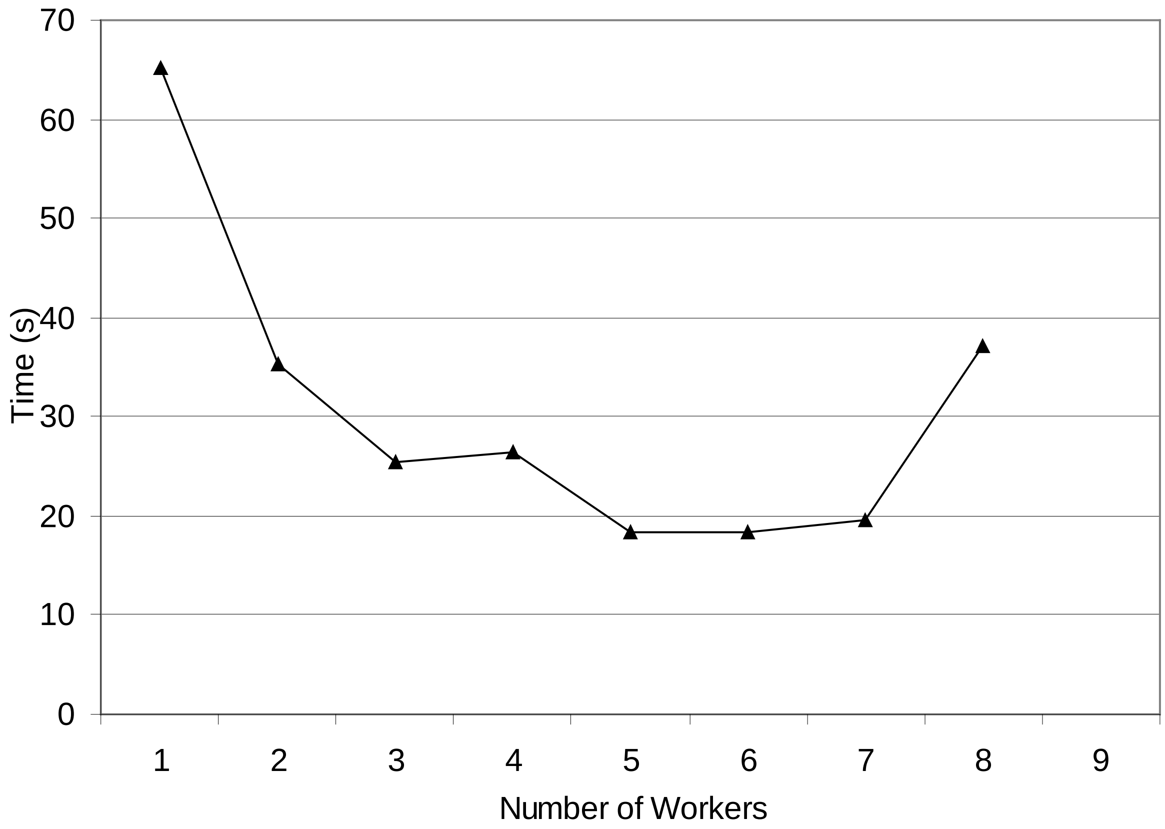

Figure 5: Ray-tracing Results for JavaSpaces

6. Future Work and Conclusions

It is clear from the reported results that the implementation of JavaSpaces is rather inefficient. Allied with this is the fact that the setup of JavaSpaces is extremely complex and often presents great difficulties. On the other hand TSpaces is a comparatively efficient implementation of the Linda model of communication. While the efficiency of the eLinda system is poor when measuring only communication, it is close to that of TSpaces for the ray-tracing application. Additionally, it may be that the distributed nature of the eLinda tuple space will result in greater scalability for larger networks of processing elements. However, all of the systems, but particularly TSpaces, do show performance benefits for the ray-tracing application.

Further work, beyond the scope of this paper, will focus on the use of these three systems in the area of distributed multimedia applications, and also on the degree of overlap between the facilities of the Programmable Matching Engine in eLinda and the tuple-matching extensions provided by TSpaces.

More generally, it appears that there is a resurgence of interest in the Linda model of distributed/parallel programming, with significant industry backing. The decoupled nature of the communication in Linda appears to be ideally suited to the current trend towards "ubiquitous computing" requiring communication between many different types of devices (mobile telephones, portable computing devices, digital assistants, multimedia entertainment devices as well as traditional computers). However, the long-standing concerns about the efficiency of the Linda model must be addressed if it is to succeed in the market-place, and the performance of the JavaSpaces implementation is particularly concerning in this regard.

Acknowledgements

This work has been carried out under the auspices of the Distributed Multimedia Centre of Excellence in the Departments of Computer Science at Rhodes University and the University of Fort Hare, with funding from Telkom SA, Lucent Technologies, Dimension Data and THRIP.

References

[1] G.C. Wells *et al.*, An Extended Version of Linda for Distributed Multimedia Applications, *SAICSIT '99*, `http://www.cs.wits.ac.za/~philip/SAICSIT/SAICSIT-99/papers_ideas.html`, November 1999.

[2] Eric Freeman *et al.*, *JavaSpaces Principles, Patterns, and Practice*, Addison-Wesley, 1999.

[3] IBM, *TSpaces*, `http://www.almaden.ibm.com/cs/TSpaces/index.html`.

[4] David Gelernter, Generative Communication in Linda, *ACM Transactions on Programming Languages and Systems*, 7(1) (January 1985), pp. 80–112.

[5] S. Hupfer *et al.*, Coordination Applications of Linda, in J.P. Banâtre and D. Le Métayer (eds.), *Research Directions in High-Level Parallel Programming Languages* (Lecture Notes in Computer Science, 574), Springer-Verlag , 1992, pp. 187–194.

[6] Nicholas Carriero and David Gelernter, *How to Write Parallel Programs: A First Course*, The MIT Press, 1990.

[7] Sudhir Ahuja *et al*., Linda and Friends, *IEEE Computer*, 19(8) (August 1986), pp. 26–34.

[8] Nicholas Carriero and David Gelernter, The S/Net's Linda Kernel, *Operating Systems Review*, 19(5) (March 1985), pp. 54–71.

[9] S. Ericsson Zenith, A Rationale for Programming with Ease, in J.P. Banâtre and D. Le Métayer (eds.), *Research Directions in High-Level Parallel Programming Languages* (Lecture Notes in Computer Science, 574), Springer-Verlag , 1992, pp. 147–156,

[10] J. Conde, *Mobile Agents in Java*, `http://wwwinfo.cern.ch/asd/rd45/white-papers/9812/agents2.html`, December 1998.

[11] Sun Microsystems, *Jini Connection Technology*, `http://www.sun.com/jini`.

[12] IBM, *The TSpaces Vision*, `http://www.almaden.ibm.com/cs/TSpaces/html/Vision.html`.

[13] World Wide Web Consortium, *Extensible Markup Language (XML)*, `http://www.w3.org/XML`.