An Extended Version of Linda for Distributed Multimedia Applications

George Wells[a] Alan Chalmers[b] Peter Clayton[a]

[a]*Department of Computer Science, Rhodes University, Grahamstown, South Africa,*
`G.Wells@ru.ac.za`, `P.Clayton@ru.ac.za`
[b]*Department of Computer Science, University of Bristol, Bristol, U.K.,*
`alan@compsci.bristol.ac.uk`

Abstract

This paper describes the implementation of an extended version of Linda, providing a form of shared memory communication for distributed processing on a network of workstations. The extensions to the Linda paradigm 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 has been implemented using Java, and some language-specific aspects of this implementation are also discussed.
Keywords: *Linda, distributed multimedia, coordination languages, Java, mobile agents*
Computing Review Categories: *C.2, D.1, D.3*

1 Introduction

This paper describes the implementation of an extended version of Linda[1] called *eLinda*. The Linda approach offers a number of advantages for parallel and distributed programming 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. The extensions introduced in *e*Linda 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. An integral part of this is the distribution of complex matching (searching) operations. The application area that is of interest to us is distributed multimedia and so support for multimedia data types has also been added to *e*Linda.

The rest of this paper consists of a brief introduction to the Linda programming model, followed by further details of the extensions embodied in *e*Linda and the motivation behind them. The implementation in Java[2] and some interesting aspects of this are also presented. The paper concludes with an overview of some sample applications that are to be used to assess the benefits of the extensions to the Linda model.

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 the tuple space may actually be shared, but on distributed memory systems (such as a network of workstations) it is usually distributed among the processing nodes. We will not consider shared memory systems further in this paper. Whatever the implementation strategy, the tuple space is accessed using associative addressing to specify the required data objects, or *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.

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

[1]Linda is a registered trademark of Scientific Computing Associates.
[2]Java is a registered trademark of Sun Microsystems, Inc.

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 (indeed, it has been shown that `eval` may be implemented in terms of the other operations with some support from a preprocessor[4]) and will not be considered any further here. Further details of the Linda approach may be found in [2].

On distributed memory systems there are many possible strategies for implementing the shared tuple space (such issues are usually transparent to the programmers and users of the system) [**REF**]. Two common approaches are hashing systems where the contents of tuples are used to allocate them to particular processors, and partitioned systems where tuples with a common structure are allocated to a specific processor. A third approach, and that adopted in this work, is to have the tuple space fully distributed, such that any tuple may reside on any processing node. In *e*Linda, 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.

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[6], 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.

3 The *e*Linda System

The extensions that have been made in the *e*Linda 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). These extensions have been made with multimedia applications in mind, but also considering the requirements of other distributed applications.

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 *e*Linda 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 *e*Linda

The extensions 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 *e*Linda system. 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 implementation 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 and it 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, a new output operation, `wr`, has been added in *e*Linda, which will broadcast the tuple throughout the processor network, whereas `out` will place a single tuple in 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* 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, where the tuples represent 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 *e*Linda system, searching for a tuple may involve accessing the sections held on all the processors in parallel. This problem is handled efficiently in *e*Linda by distributing the matching engine so that 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 *e*Linda 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.

A particularly useful feature of Java for the implementation of the programmable matching engine is the *interface* mechanism provided by the language. This allows the system designer to specify the mechanisms which must be provided by the user in order to implement a matching algorithm, without dictating the implementation. The interface for the programmable matching engine is shown in Table 1.

The writer of a matching algorithm may implement it in any class, which may or may not be part of a particular application. An object of this class must implement the `ProgrammableMatcher` interface (and thus the two required matching functions described in the code above) and it can then be passed to the *e*Linda system and be used by it.

Writing such a matcher is not a simple operation. Various library calls are provided to allow the matcher to interact directly with the *e*Linda system.

The programmable matching engine concept has something in common with the recent adoption of *mobile agents*[3]. Effectively, a customised matcher written for the *e*Linda programmable matching engine is a form of mobile agent that is distributed on a network to find a matching tuple (or tuples).

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

3.1.3 Support for Multimedia

The third feature of *e*Linda is its support for multimedia data. Tuples in *e*Linda may contain any of the 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 *e*Linda system. In this way the *e*Linda system attempts to provide the maximum possible functionality for general purpose applications.

To this basic functionality has been added the ability to use a `Multimedia` object in an *e*Linda tuple. This class acts as a wrapper to the underlying Java Media Framework (JMF) multimedia resource. In particular the implementation of the `Multimedia` class provides support (transparent to the application programmer) for any necessary buffering of data, fetching or streaming of multimedia data across the network, etc.

As an example, in order to search for and present a multimedia resource something like the following might be written using the *e*Linda extensions:

```
Multimedia m;
if (inp("Movies", "Star Wars", ?m))
  m.play();
else
  System.out.println("Unavailable");
```

The support for multimedia has not been fully implemented at this stage. The design of this part of the system relies on the multimedia facilities implemented in the Java Media Framework (JMF). This is an API that defines support for various differing types of audiovisual media, including stored and streamed media (for example, live video feeds).

3.2 The Implementation of *e*Linda on a Network of Workstations

A preliminary version of *e*Linda has been implemented using Java. The current structure of the *e*Linda 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 shared hosts in any desired configuration, thus allowing for simple load-

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

```
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
```

Table 1: The `ProgrammableMatcher` Interface

balancing. The communication between JVMs makes use of the TCP/IP network protocol, implemented by the Java `Socket` class. Within a 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.

The decision to use the TCP/IP protocol for communication between the JVMs was taken for efficiency reasons. Several other communication mechanisms exist, notably RMI (Remote Method Invocation) and CORBA, both of which support remote objects and method calls. While the use of these higher level protocols would simplify the structure and implementation of the system, they rely on TCP/IP for their underlying communication needs and so introduce their own additional processing overheads.

4 The Sample Application Areas

Several different application areas are being investigated in order to evaluate the usefulness of the *e*Linda system and to assess its efficiency.

4.1 Video-on-demand

This project involves using the *e*Linda system to handle a pilot video-on-demand application. It will allow research into the most efficient ways of transferring video data across a network and how the *e*Linda system should best support such multimedia data types. This would include provision for both “live” video streams and stored video. Commercial video servers are extremely expensive (costing as much as $14m), and this project is intended to show how similar functionality could be provided by a network of several cheap machines making use of *e*Linda for the coordination and communication.

4.2 Computer-Based Education

The second application area that we intend to explore is the use of *e*Linda to present multimedia CBE materials on the World Wide Web. This will involve setting up some pilot lessons to demonstrate the feasibility of the proposals embodied in the *e*Linda project. In addition it will illustrate the general-purpose applicability of *e*Linda, allowing simple, efficient distribution of multimedia resources and providing control and coordination (e.g. of student-teacher interaction in a WAN education environment).

4.3 Search Engines

One of the strengths of the Linda paradigm is the way in which work can easily be distributed among workstations on a network. This is likely to be especially useful for applications such as distributed search engines (e.g. searching the World Wide Web). This project will enable an assessment to be made of the programmable matching engine concept, particularly with respect to its usefulness, but also its efficiency.

5 Future Work and Conclusions

Part of the design of the communication mechanisms in *e*Linda was specifically intended to allow experimentation with some of the concepts from the Data Diffusion Machine (DDM). The DDM is a virtual shared memory archi-

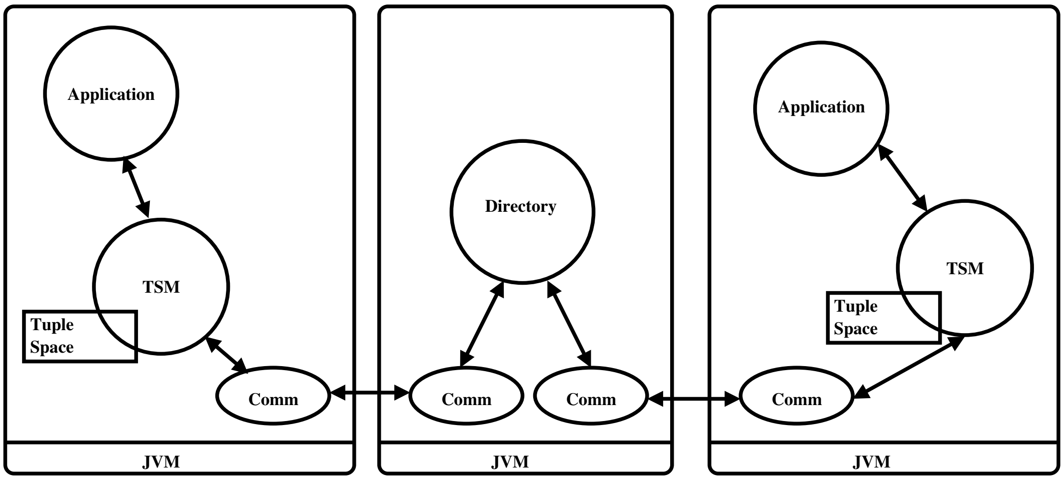

Figure 1: The Structure of the *e*Linda System

tecture with a hierarchical, associative addressing mechanism developed at the University of Bristol[5]. This is intended to exploit the fact that the Linda tuple space is essentially a virtual shared memory with an associative searching/retrieval mechanism. The application of the communication principles proposed by the DDM research team may lead to further efficiencies in the communication mechanisms used by the *e*Linda system.

Virtual shared memory approaches to parallel and distributed programming are very appealing due to their inherent simplicity. When implemented on distributed memory systems (such as networks of workstations) a major concern is the efficient handling of the underlying communication. The *e*Linda project is intended to demonstrate that reasonable efficiency can be achieved without unduly compromising the simplicity of the Linda model, in particular in the area of distributed multimedia applications.

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 and Dimension Data.

References

[1] J. P. Banâtre and D. Le Métayer, editors. *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

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

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

[4] S. Hupfer, D. Kaminsky, N. Carriero, and D. Gelernter. Coordination applications of Linda. In Banâtre and Métayer [1], pages 187–194.

[5] Sanjay Raina. *Emulation of a Virtual Shared Memory Architecture*. PhD thesis, University of Bristol, 1993.

[6] S. Ericsson Zenith. A rationale for programming with Ease. In Banâtre and Métayer [1], pages 147–156.