A TUPLE SPACE WEB SERVICE FOR DISTRIBUTED PROGRAMMING
Simplifying Distributed Web Services Applications

George C. Wells, Barbara Mueller† and Loïc Schulé†
Department of Computer Science, Rhodes University, Grahamstown, South Africa
†Swiss Institute of Technology (EPFL), Lausanne, Switzerland

Keywords: Tuple spaces, coordination languages, distributed processing, web services, associative matching.

Abstract: This paper describes a new tuple space web service for coordination and communication in distributed web applications. This web service is based on the Linda programming model. Linda is a coordination language for parallel and distributed processing, providing a communication mechanism based on a logically shared memory space. The original Linda model has been extended through the provision of a programmable mechanism, providing additional flexibility and improved performance. The implementation of the web service is discussed, together with the details of the programmable matching mechanism. Some results from the implementation of a location-based mobile application, using the tuple space web service are presented, demonstrating the benefits of our system.

1 INTRODUCTION

The Linda coordination language for concurrent programming was first proposed by David Gelernter at Yale University (Gelernter, 1985). The small set of operations, the associative data access/retrieval mechanism and the logically shared memory space all provide a very useful simplicity and flexibility for constructing concurrent applications. On the other hand, a criticism of Linda has been that it is, at worst, inefficient, and, at best, subject to unpredictable performance (Zenith, 1992), as the simplicity of the model hides the underlying complexity of the data sharing and communication required. Furthermore, some applications may be very difficult to implement efficiently using the standard Linda communication mechanisms. Since the late 1990's there has been a resurgence of interest in the Linda coordination model, particularly in the Java community. Notably, Sun Microsystems developed the JavaSpaces specification (Freeman et al., 1999). IBM have also produced a commercial Linda implementation in Java, called TSpaces (Wyckoff et al., 1998). We believe that the simplicity of the Linda model still has much to offer, but that there are still challenges in overcoming the performance issues inherent in this approach, and extending the range of applications to which it is suited. Our previous research led to the development of a system (called *eLinda*) that provided extensions to the original Linda model, addressing some of these performance-related concerns, while attempting to retain as much of the model's simplicity as possible (Wells et al., 2004).

This paper describes the implementation of a Linda tuple space web service. This topic has also been studied by a few other groups, notably Lucchi and Zavattaro, who focused specifically on the security of a tuple space web service (Lucchi and Zavattaro, 2004), and the Advanced Information Systems Group at the Universidad de Zaragoza (Mata et al., 2004). Our new tuple space service is called the *Extended Linda Web Service* (abbreviated *eLindaWS*).

The major extension present in eLindaWS is a programmable matching mechanism designed to improve the flexibility and the efficiency of the Linda model. This is based on our previous research, but the implementation as a web service in this case leads to some unique features and potential security-related problems. The product of this research project is a simple, but highly effective coordination web service that is applicable to a wide range of Internet application areas.

To demonstrate the benefits of eLindaWS, we have implemented a number of example applications. One of these is a bioinformatics application, which shows very good performance characteristics (Mueller et al., 2007). In this paper we discuss another example application, a location-based mobile web service, which specifically illustrates the use and benefits of the programmable matching mechanisms.

2 THE LINDA PROGRAMMING MODEL

The Linda programming model has a highly desirable simplicity for writing parallel or distributed applications. As a *coordination language* it is responsible solely for the coordination and communication requirements of an application, relying on a *host language* (i.e. Java in this study) for expressing the computational requirements of the application.

The Linda model comprises a conceptually shared memory store (called *tuple space*) in which data is stored as records with typed fields (called *tuples*). The tuple space is accessed using five simple operations[1]:

`out` Outputs a tuple from a process into tuple space

`in` Removes a tuple from the tuple space and returns it to a process, blocking if a suitable tuple cannot be found

`rd` Returns a copy of a tuple from the tuple space to a process, blocking if a suitable tuple cannot be found

`inp` Non-blocking form of `in` — returns an indication of failure, rather than blocking if no suitable tuple can be found

`rdp` Non-blocking form of `rd`

Note that the names used for these operations here are the names used in the original Linda system developed at Yale. Both TSpaces and JavaSpaces use different names for the operations.

Input operations specify the tuple to be retrieved from the tuple space using a form of *associative addressing* in which some of the fields in the tuple (called an *antituple*, or *template*, in this context) have their values defined. These are used to find a tuple with matching values for those fields. The remainder of the fields in the antituple are variables which are bound to the values in the retrieved tuple by the input operation (these fields are sometimes referred to as *wildcards*). In this way, information is transferred between two (or more) processes.

A simple one-to-one message communication between two processes can be expressed using a combination of `out` and `in` as shown in Figure 1. In this case `("point", 12, 67)` is the tuple being deposited in the tuple space by Process 1. The antituple, `("point", ?x, ?y)`, consists of one defined field (i.e. `"point"`), which will be used to locate a matching tuple, and two wildcard fields, denoted by a leading `?`. The variables `x` and `y` will be bound to the values 12 and 67 respectively, when the input operation succeeds, as shown in the diagram. If more than one tuple in the tuple space is a match for an antituple, then any one of the matching tuples may be returned by the input operations.

Other forms of communication (such as one-to-many broadcast operations, many-to- one aggregation operations, etc.) and synchronization (e.g. semaphores, barrier synchronization operations, etc.) are easily synthesized from the five basic operations of the Linda model. Carriero and Gelernter provide further details of the Linda programming model (Carriero and Gelernter, 1990).

3 THE IMPLEMENTATION OF ELINDAWS

The eLindaWS system is based on our previous extended Linda system, eLinda (Wells et al., 2004). A prototype web service was developed without the extensions (Wells, 2006), which showed promise in this approach, and served as the basis for the development of the full eLindaWS system presented in this paper. This system consists of two separate parts: a server-side component, providing the web service itself, and a small client library, abstracting some of the details of the use of the web service. The server side of the web service has been developed as a Java servlet, using the Apache Tomcat servlet container.

We have adopted the *REST* (Representational State Transfer) approach to web services, proposed by Roy Fielding as a reaction to the proliferation of web services standards (Fielding and Taylor, 2002). The REST approach essentially calls for the transmission of simple XML messages across the Internet, using HTTP. This approach is distinguished by its simplicity, in comparison to more conventional approaches to web service provision, such as the use of SOAP, WSDL, etc.

Using XML to transport data has the advantage

[1]A sixth operation, `eval`, used to create an *active tuple*, was proposed in the original Linda model as a process creation mechanism, but can easily be synthesized from the other operations, with some support from the compiler and runtime system, and is not present in any of the commercial Java implementations of the Linda model.

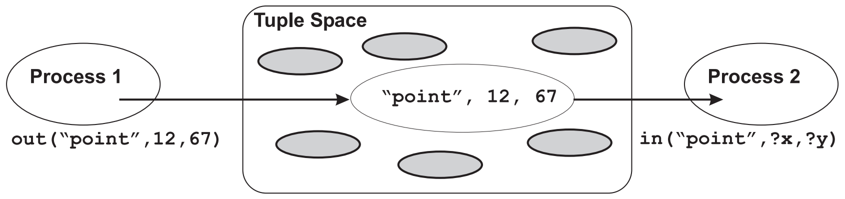


Figure 1: A Simple One-to-One Communication Pattern.

that the client and the server can be written in almost any language (Java, Perl, etc.). In our implementation, the XML document received by the server is converted into a Java data structure, which is used with a Linda-style tuple space to carry out the relevant input or output operation.

An additional benefit of using a servlet container for eLindaWS is that debugging becomes very simple. In particular, a second servlet was developed that obtains a listing of the contents of the tuple space and returns this as a conventional HTML web page. Viewing this page in a web browser produces a very useful snapshot of the current state of the tuple space. A third servlet was developed to reinitialize all tuple spaces without having to restart the Apache server, which is very helpful for testing.

The eLindaWS system has three extensions to the original Linda programming model:

1. Multiple, named tuple spaces
2. Multi-tuple input and output operations
3. A programmable matching mechanism

These extensions are briefly explained below, whereafter a more detailed description of the programmable matching mechanism is provided.

Multiple, named tuple spaces allow unrelated tuples to be stored separately, and thus provide greater efficiency.

The multi-tuple output operation, called `outMany`, is used to minimize network traffic and to accelerate the execution time of applications. If a client application wants to add several tuples to a tuple space at the same time, the use of `outMany` allows a single XML document to be sent, containing all the tuples. Likewise, the multiple input operations, which have corresponding names and functionalities to the single input operations (`inMany, inpMany, rdMany, rdpMany`), return all tuples that match the given antituple. In the same way as the `outMany` operation, this minimizes the network traffic.

The programmable matching mechanism allows the use of more flexible criteria for locating tuples. This is particularly useful in situations where a global view of the tuples in a tuple space is required. The programmable matching mechanism allows the necessary computation to be moved out of the application into a specialized matcher, which is executed on the server, both minimizing the network traffic and maximizing the parallelism of a distributed application.

3.1 Programmable Matching

With the standard Linda associative matching methods, a tuple can only be retrieved if it corresponds *exactly* to an antituple. Moreover, the matching process can only be done with one tuple at a time. While this basic matching mechanism is simple to implement and very useful in many cases, it is problematic in certain situations, especially those where a global view of all tuples in a tuple space is required. The programmable matching mechanism allows for the use of very flexible criteria for matching tuples with antituples and provides for a global view of the tuples during the matching process.

For example, consider a scenario where a tuple is required that has a numeric field, the value of which is the closest to some specified value (i.e. not necessarily equal to the given value). Such queries can be expressed using the standard Linda associative matching methods, but will generally be very inefficient. In the example above, the application would have to retrieve all of the eligible tuples using the `inMany` operator, work through them to locate the one with the required closest value and then put all other tuples back in the tuple space using the `outMany` operator. This will result in a large volume of network traffic (this aspect is even worse if the additional multiple-tuple operations are not available, as our analysis has shown that the network throughput is the main performance bottleneck). Furthermore, the parallelism of the system is reduced as the application keeps all tuples for the period required to determine which one is required, possibly blocking other clients.

Using the programmable matching mechanism, this problem is handled efficiently by allowing a client to install a new matcher on the server, thus defining its own tuple matching strategy. For example, when searching for the closest tuple, a client can install a specialized matcher capable of performing this matching operation, and then perform an input operation using the new matcher. Since the matching is done on the server, the network traffic and the impact on the parallelism of the system are minimized.

Specialized matchers may also perform aggregate operations, where a tuple is returned that summarizes or aggregates information from a number of tuples. For example, a matcher could calculate the sum of numeric fields.

In general, the use of the programmable matching mechanism will simplify application development, in addition to the performance benefits outlined above. This is particularly true where a pre-written matcher is available, as we envisage that any complete implementation of our system would include a library of common matching operations, providing a useful set of generic matching facilities. More specialized matchers would have to be written as part of the development of the application for which they were required. Such matchers could then be added to the library of existing matchers for future use, if appropriate. It is also conceivable that writing specialized matchers could become a service provided by an entity separate from the application development team, possibly as a commercial service.

3.1.1 Design and Implementation of the Programmable Matching Mechanism

The programmable matching mechanism is designed to be simple and as easy as possible for a programmer to master. To begin with, the process can be separated into two sub-processes:

1. The installation of a new matcher in the server
2. The use of a specialized matcher by a client

Installation of a New Matcher. There are two candidate approaches to installing a matcher on the server. One solution is that a client compiles the code and sends the executable code to the server. This solution is interesting but implies that the client must send complex, binary objects through the network. This complicates the system architecture and the XML documents used. It potentially also increases the volume of network traffic (compiled code is generally larger than source code, particularly when encoded for transmission as text). For these reasons, this solution was discarded.

The second possibility, and the one that we have chosen, is to send the source code to the server and to let the server perform the compilation. In this approach, the client sends a textual message containing the matcher code to the server. The server then performs the compilation and informs the client whether the compilation was successful or not.

From the client's perspective, installing a matcher requires obtaining the source code for the matcher and sending it to the server. From the perspective of the programmer, writing matchers, while not a trivial operation, is not overly complex. A library is provided that allows the programmer to access the tuples in the tuple space at a lower level of abstraction than usual, and care needs to be taken to preserve the semantics of the Linda tuple retrieval operations.

While lines of code are a notoriously poor indication of complexity, they can give an approximate indication of the difficulty of writing a customized matcher. For example a matcher to find the minimum of a numeric field can be written in only 37 lines of Java code.

To install a new matcher on the server, a client simply sends the name and the source code for the matcher, using an XML document with the following format:

```
<eLinda>
  <Matcher name="MyMatcher">
    <Source type="string">
      ...
    </Source>
  </Matcher>
</eLinda>
```

The server parses the XML document and forwards the information to the *Matcher Manager*. The Manager then compiles the matcher, and informs the server whether the source code was compiled correctly or not. The server then encodes any error messages or an indication of success and finally sends this information back to the client.

Using a Specialized Matcher. Once a specialized matcher is installed on the server, it becomes trivial to use it. The client must simply specify which matcher it wants to use by indicating the name of the matcher. The XML schema used includes an attribute, `matcher`, indicating the name of the matcher that should be used. If no matcher name is specified by the client application, the `matcher` attribute is set to `DefaultMatcher`. The following XML document gives an example of an input tuple sent together with a matcher name:

```
<InputTuple oneResult="true"
            matcher="MyMatcher"
            tupleSpace="t1"
            blocking="true"
            destructive="true">
  <TupleData>
    <Field type="string">point</Field>
    <Field type="int" />
    <Field type="int" />
  </TupleData>
</InputTuple>
```

Once the matcher is installed, the process of using a specialized matcher is almost completely transparent to the client, which only needs to specify the name of the matcher. This mechanism helps maintain the simplicity of eLindaWS, while providing considerable flexibility and enhanced performance.

Parameterization of Matchers. Ideally, the number of matchers installed should be minimized as far as possible. For example, a client might want to retrieve tuples containing a certain set of values, where the set of values varies for each run. It would be impractical and inefficient for the client to install a new matcher each time with the actual set of values and it would be resource consuming for the server. This observation leads to the conclusion that some matchers may require parameterization. The eLindaWS system supports this mechanism through a simple extension to the basic mechanism outlined above, whereby a list of parameter values can be passed to the matcher, along with the antituple specification, for input operations. This greatly increases the flexibility of eLindaWS and the reusability of specialized matchers.

Error Handling. With the programmable matching mechanism, any client can install its own matcher on the server. This implies also that any client can install a badly designed matcher that compiles but that doesn't execute correctly, generating a run-time exception. This requires additional error detection and transmission to the client application. Since eLindaWS may be used by people with widely varying skill levels in many contexts, we implemented the error-handling to be configurable, with a number of different "levels". For example, if the server is used by developers who need detailed information in order to debug their matchers, the server can be configured appropriately. On the other hand, if the application is to be used by non-technical users, the server can be configured to return minimal information about any errors. This is done by using the servlet deployment descriptor mechanism.

Security Issues. Security was not a major initial focus of this project, however, the problem of security had to be considered. The eLindaWS system allows a client to install a specialized matcher by sending the matcher code to the server. If there is no compile-time error, it installs the matcher and allows client applications to use it. This feature increases the flexibility of the system, but also opens a major security loophole, as a matcher may attempt to modify system variables, or to write or read files that it should not access.

There are two possible solutions to this problem that were considered. The first was the use of the Java security package, `java.security`. This package provides classes that implement a configurable, fine-grained access control security architecture. This package also supports the generation and storage of cryptographic public key pairs, as well as a number of exportable cryptographic operations including those for message digest and signature generation. Finally, this package provides classes that support signed/guarded objects and secure random number generation.

The second possible solution involved the use of a Security Manager. This is a single Java object that performs runtime checks on potentially dangerous methods. Code in the Java library consults the Security Manager whenever such an operation is attempted. The Security Manager can veto the method call by generating a `SecurityException`. Decisions made by the Security Manager are defined in a policy file describing the rights of all packages. Each virtual machine can have only one Security Manager installed at a time, and once a Security Manager has been installed it cannot be uninstalled (except by restarting the virtual machine).

While the Java security package offers a highly flexible, sophisticated and complete set of security mechanisms, a light-weight solution was better suited to our needs. As explained above, the only requirement was to make sure that the specialized matcher could not write or read files on the server or access system variables. Accordingly, we configured the Security Manager to give no rights to the package containing the matchers but unlimited rights to the rest of the server packages.

3.1.2 Applications of the Programmable Matching Mechanism

The examples of new matchers given during the preceding discussion have been in the domain of numeric applications. These are convenient for the purposes of the discussion as they are simple, easily explained and easily understood. However, it would be incorrect to believe that the programmable matching mechanism was only useful for numeric problems — it is just as applicable to textual or other problems. Some examples that emphasize the generic nature of the programmable matching facilities are:

- A string matcher could match string fields using some alphabetic measure of "closeness", or even approximate homophonic matching.
- A spatial matcher could compare coordinates to locate a tuple corresponding to a point in some area or space — see the example in Section 4.
- A matcher could be written to locate tuples with fields corresponding to a date or time in some range of temporal values.

4 THE MOBILE, LOCATION-BASED APPLICATION

This application demonstrates the benefits of the programmable matching mechanism in eLindaWS. However, it also examines the suitability of using eLindaWS for location-based systems, and its applicability to the field of mobile web applications. The application is that of locating desired facilities close to the user's current position.

A *midlet* is a Java program for embedded devices, which conforms to the *Mobile Information Device Profile* (MIDP) standards (Sun Microsystems, 2007). MIDP is a set of J2ME APIs that define how software applications interface with mobile communication devices. Several different toolboxes for developing wireless applications are available, e.g. the *Sun Java Wireless Toolkit*, and the *NetBeans Mobility Pack*. To write our midlet application, we used the former toolkit.

4.1 Structure of the Application

Our example application enables a user to obtain information about facilities within a certain range of the user. We restricted our prototype application to the range of the town of Grahamstown, South Africa, but it could be trivially extended to include other areas by adding to the data set used. The user first specifies whether they are looking for information about restaurants or other places of interest. In the former case, they can enter the type of food and the price range they are looking for, as well as the desired proximity of the restaurant. If they are looking for other facilities, they can specify the type of facility, the quality of service, and the desired proximity. This information is then used to construct an antituple, which is sent to the eLindaWS server. The server then uses a specialized matcher to perform the spatial matching required to locate the desired facilities. A set of the matching tuples is then returned to the mobile device. This is used by the application to display all matching facilities, and the user is then able to consult the details of any of these facilities or to make another request. This interaction is illustrated by the screenshots in Figure 2.

4.2 Development Tools and Issues

The Sun Java Wireless Toolkit (Sun Microsystems, 2006) is a state-of-the-art toolbox for developing wireless applications that are based on the J2ME Connected Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP), and is designed to run on cell phones, personal digital assistants and other small, mobile devices. The toolkit includes the emulation environments, performance optimization and tuning features, documentation, and examples that developers need to bring efficient and successful wireless applications to market quickly.

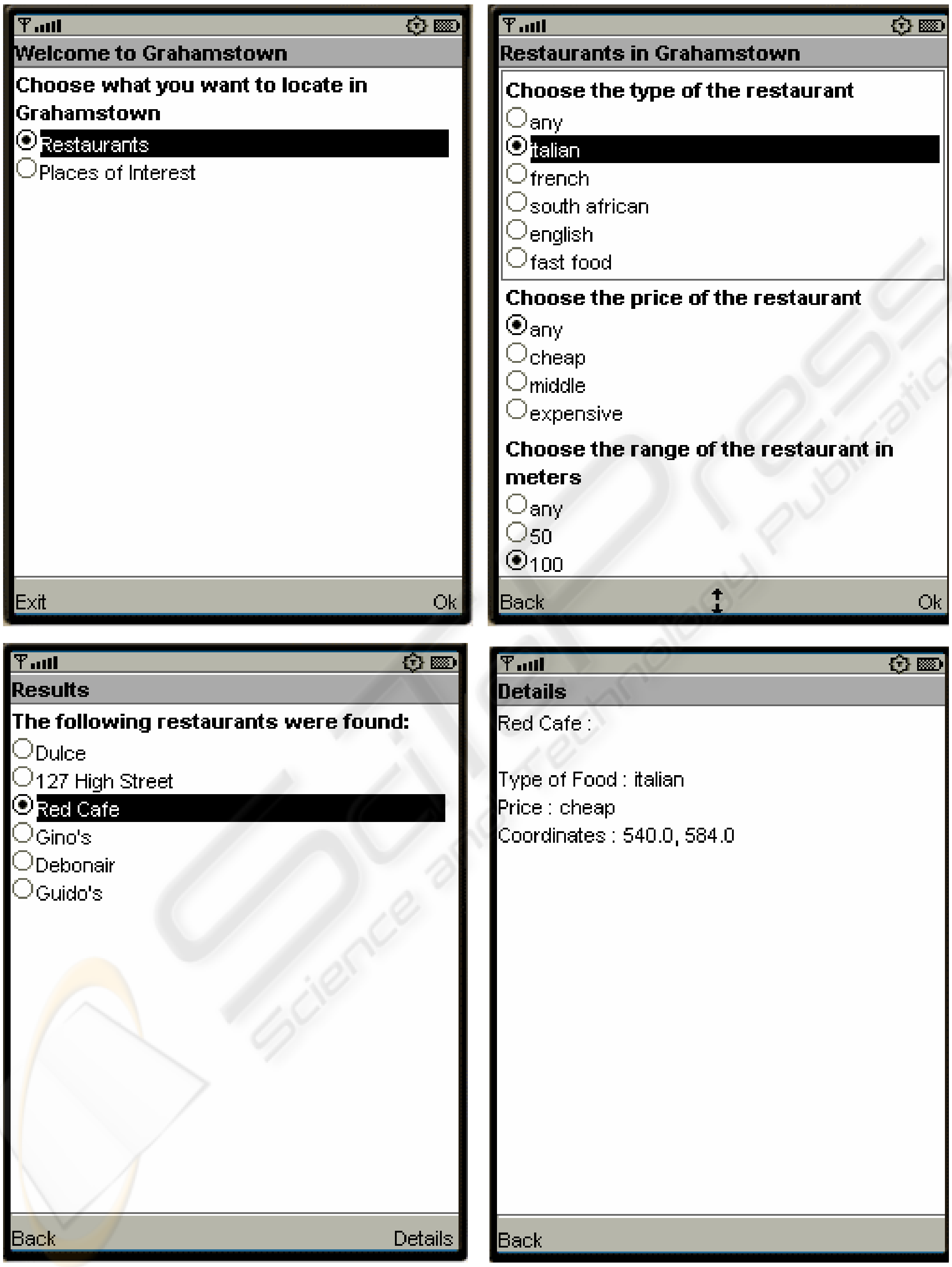
To implement the interface of our location-based midlet application, we used the cellphone emulator which is contained in the J2ME Wireless Toolkit. We chose this toolkit because it supports HTTP communications as well as an XML parser. We had only a few changes to make to adapt the Java client library for this environment. The most significant change was that the SAX XML API had to used, rather than the StAX API, used in the rest of the system, but which is not supported by the J2ME Wireless Toolkit.

4.3 Discussion

This application demonstrates several beneficial characteristics of the eLindaWS. First, it shows that eLindaWS is suitable to be used for location-based applications. Each tuple corresponds to a facility at a given location, and the coordinates are simply two additional fields inside the tuple. The nature of the standard associative matching mechanism simplifies the construction of queries that take into account user-specified criteria such as the type of food and price-range.

Secondly, this example demonstrates the use of a programmable matcher to implement the spatial matching required by this web service, and the parameterization of the matcher in order to specify the user's location and desired proximity of the results. The details of this mechanism are illustrated in the XML fragment below. This shows the specification of the specialized matcher (named `LocationMatcher` in this application), and the parameters specifying the radius of the search, and the user's current location to be used by the specialized matcher. A standard Linda application would have to retrieve all the tuples corresponding to the desired facilities, and then perform the spatial filtering itself in order to locate ones within the required radius.

```
<InputTuple matcher="LocationMatcher"
            tupleSpace="LocationTuple">
  <Parameters>
    <Field type='double'>100</Field>
    <Field type='double'>500.0</Field>
    <Field type='double'>400.0</Field>
  </Parameters>
  <TupleData>
    <Field type='string'>Restaurant</Field>
    <Field type='string'/>
```



Welcome to Grahamstown

Choose what you want to locate in Grahamstown

[✓ Restaurants]
Places of Interest

Exit Ok

Restaurants in Grahamstown

Choose the type of the restaurant

any
[✓ italian]
french
south african
english
fast food

Choose the price of the restaurant

[✓ any]
cheap
middle
expensive

Choose the range of the restaurant in meters

any
50
[✓ 100]

Back Ok

Results

The following restaurants were found:

Dulce
127 High Street
[✓ Red Cafe]
Gino's
Debonair
Guido's

Back Details

Details

Red Cafe :

Type of Food : italian
Price : cheap
Coordinates : 540.0, 584.0

Back

The top two screens show the input specifications entered by the user. The lower left screen shows the summary of the results returned by the server, and the lower right screen shows the full details of the selected result.

Figure 2: Screen-Shots of the Mobile, Location-Based Application.

```
      <Field type='string'>French</Field>
      <Field type='string'>cheap</Field>
    </TupleData>
  </InputTuple>
```

Thirdly, while this is a simple demonstration, it demonstrates the potential practical applicability of eLindaWS. In particular, it would be a cost-effective solution to this problem in the South African context, where cellphone usage is far more widespread than general Internet usage, and mobile data rates are considerably cheaper than voice or SMS rates.

5 CONCLUSIONS

Our previous research has demonstrated both the desirable simplicity and the improved performance that can be obtained by extending traditional Linda systems with programmable matching facilities. The research presented in this paper demonstrates that these advantages can be extended to the field of distributed web service applications.

The facility-locating midlet application demonstrates that eLindaWS is suitable for use in mobile, location-based applications, and illustrates the use of a parameterized programmable matcher. While it is only intended as a proof-of-concept prototype, it functions in a realistic way, and could potentially be the basis for a useful and cost-effective commercial application, particularly in our South African context.

Future work to improve eLindaWS is likely to concentrate on optimization, in particular:

- The optimization of the creation and parsing of the XML documents (Ranganath et al., 2006)
- Improving the communication protocol used to send the XML documents, especially for LAN configurations
- Preprocessing the tuples before storing them in a tuple space, e.g. indexing of the tuples, sorting of the tuples, or other methods

ACKNOWLEDGEMENTS

This work was supported by the Distributed Multimedia Center of Excellence in the Department of Computer Science at Rhodes University, funded by Telkom, Business Connexion, Comverse, Verso Technologies, StorTech, Tellabs, Amatole Telecommunication Services, Mars Technologies, Bright Ideas Projects 39 and THRIP. Financial support was also received from the National Research Foundation.

REFERENCES

Carriero, N. and Gelernter, D. (1990). *How to Write Parallel Programs: A First Course*. The MIT Press.

Fielding, R. and Taylor, R. (2002). Principled design of the modern web architecture. *ACM Trans. Internet Technology*, 2(2):115–150.

Freeman, E., Hupfer, S., and Arnold, K. (1999). *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley.

Gelernter, D. (1985). Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112.

Lucchi, R. and Zavattaro, G. (2004). WSSecSpaces: a secure data-driven coordination service for web services applications. In *SAC '04: Proc. 2004 ACM Symposium on Applied Computing*, pages 487–491, New York, NY, USA. ACM Press.

Mata, E., Álvarez, P., Bañares, J., and Rubio, J. (2004). Towards an efficient rule-based coordination of web services. In *IBERAMIA 2004*, volume 3315 of *Lecture Notes in Artificial Intelligence*, pages 73–82. Springer Verlag.

Mueller, B., Schulé, L., and Wells, G. (2007). Using a tuple space web service for parallel processing in bioinformatics. In *Proc. First Southern African Bioinformatics Workshop*, pages 34–37. Wits University.

Ranganath, V., King, A., and Andresen, D. (2006). Automatic code generation for LYE, a high-performance caching SOAP implementation. *International Conference on Semantic Web and Web Services, Las Vegas*.

Sun Microsystems (2006). Sun Java Wireless Toolkit. `http://java.sun.com/products/sjwtoolkit`.

Sun Microsystems (2007). Mobile Information Device Profile (MIDP). `http://java.sun.com/products/-midp`.

Wells, G. (2006). A tuple space web service for distributed programming. In Arabnia, H., editor, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2006)*, pages 444–450. CSREA Press.

Wells, G., Chalmers, A., and Clayton, P. (2004). Linda implementations in Java for concurrent systems. *Concurrency and Computation: Practice and Experience*, 16:1005–1022.

Wyckoff, P., McLaughry, S., Lehman, T., and Ford, D. (1998). T Spaces. *IBM Systems Journal*, 37(3):454–474.

Zenith, S. (1992). A rationale for programming with Ease. In Banâtre, J. and Métayer, D. L., editors, *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*, pages 147–156. Springer-Verlag.