Technical Report DHPC-128

Tuple-Space Based Middleware for Distributed Computing

K.A.Hawick, H.A.James, and L.H.Pritchard
Computer Science Division, School of Informatics
University of Wales, Bangor, North Wales, LL57 1UT, UK
{hawick,heath,l.pritchard}@bangor.ac.uk
Tel: +44 1248 38 2717, Fax: +44 1248 36 1429

Tuple-spaces provide a powerful model for managing a distributed object system. We investigate algorithmic and implementation issues for combining independent spaces into a SuperSpace that could scale and perform sufficiently well to be a basis for a global grid middleware. We have used the JavaSpaces implementation and discuss transactional integrity issues and our approach to implementing bridging modules to connect local area spaces.

Keywords: tuple-space; JavaSpace; middleware; transactions.

1 Introduction

The tuple-space model pioneered by Gelernter and co-workers [3] in the Linda programming system is an attractive one for coordinating objects across a distributed computing environment. The advent of the JavaSpaces implementation of this model [2] adds to its applicability. In this paper we describe our ideas and experiments in using JavaSpaces as the base layer for a distributed systems middleware [5] that is both scalable and dynamic in terms of resource management. We believe this approach may be useful for implementing middleware for a global computational grid [1].

The Space model provides a management structure or Space for shared objects to exist and be accessed by client programs. The JavaSpaces implementation in particular provides a useful set of tools and Java style methods that support tuple space operations on `Entry` objects. Our principle goal in this work has to to establish mechanisms for operating a wide area "space of spaces". It is important for a middleware infrastructure that local area spaces can be integrated together in a reliable and transactionally correct fashion. Our early work looked into the use of a JavaSpace for managing dynamic computer clusters [6, 7], where programs used some sort of message passing infrastructure including a Java implemented system [10]. That work was largely based around one or at most two JavaSpaces and it was possible to build bridging software components to support inter-object transactions between

the Spaces. Our present work aims at the integration of a potentially arbitrary number of Spaces, with each JavaSpace environment running on its own host computer. We are experimenting with algorithms and code structures for adding bridge modules to link the JavaSpaces.

In section 2 we review the basic concepts of a Space and the operations supported by JavaSpaces. We also discuss other Space style implementations including IBM's T-Space [8] system and Sun's GigaSpace [4] system. We have used a relatively simple global bank account applications example to explain our model and to formulate some of our experiments. We use this to explian some Space operations in section 3.

The main focus of this paper is our idea for bridging multiple Spaces together into a SuperSpace and we present a worked example of the transaction sequence for this to operate correctly in section 4. Another idea we are keen to develop is that of dynamic SuperSpaces in which the number of participants varies in time. We have conducted some work on this idea for local area spaces [6] but discuss some new ideas in section 5. Finally we present some performance measurements and estimates for scoping the scalability behaviour of a SuperSpace in section 6 and summarise our conclusions in section 7.

2 The Space Model and Implementations

Gelernter's development of the Linda implementation of a tuple-space inspired a number of research projects and has led to the development of the JavaSpaces system [2]. JavaSpaces builds upon Sun Microsystem's Jini infrastructure and makes use of several of Jini's distributed computing services [11]. Figure 1 shows the main ideas for a JavaSpace. A Space is hosted by a server computer and manages objects placed into it by client programs. A JavaSpace can manage its `‘‘Entry’’` objects persistently and certain guarantees are made about the transactional integrity of operations clients invoke on `Entry` objects. A mechanism is provided to implement transactions on compound objects or groups or objects and operations although we have found this is not entirely reliable and have been forced to consider transaction operations on groups of objects using our own additions. We discuss these ideas in section 4.

The Object mechanism used for this is to have all objects that will be managed by the Space implement the `Entry` interface specification. The JavaSpaces model allows client programs to: *write* `Entry` objects to a Space; *read* a copy of them from the space; and *take* `Entry` objects from the Space. `Entry` objects are matched by read or take operations using a pattern object with matching (public) data fields.

The Space model has proved relatively popular with a set of useful applications and components discussed on the Web [9]. IBM have also made available a product using the tuple space model. Their TSpaces [8] system makes available a number of communication services to a Java program.

There has not been quite as widespread an uptake of the model as might have been expected however perhaps due to performance concerns about Linda, shared memory systems and JavaSpaces. We believe these concerns are being addressed with new implementations such as Sun's GigaSpaces system [4]. Furthermore the memory resources that are required to run a JavaSpace are no longer such a relatively large fraction of the memory available on

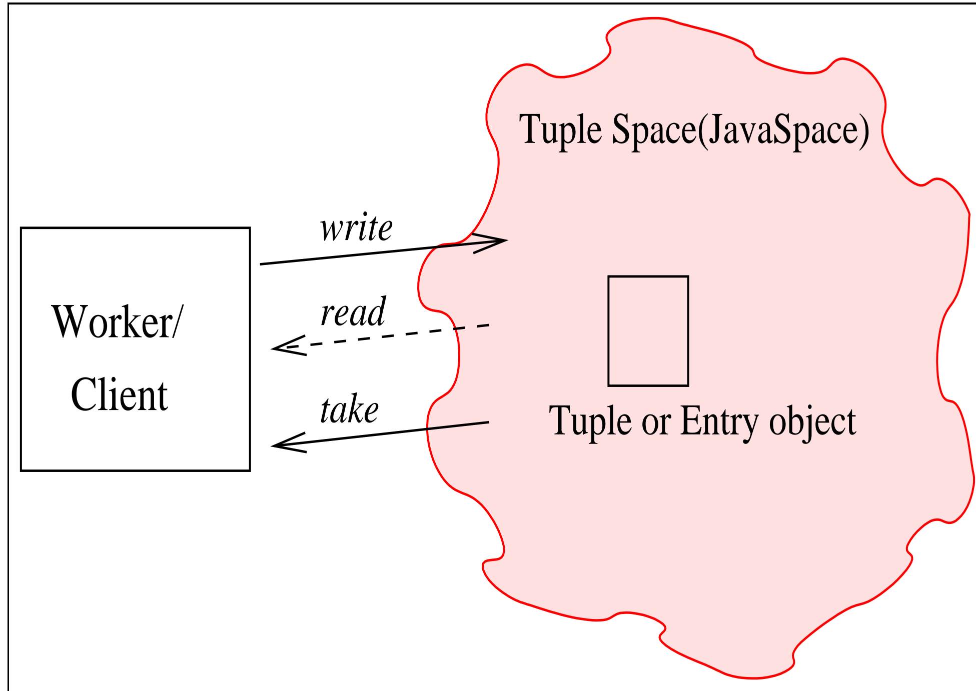

Figure 1: Primitive operations supported by a JavaSpace. The client program can *read*, *write* or *take* `Entry` objects that are managed by the Space.

typical desktop computers and it is therefore much more feasible at the time of writing to use the JavaSpace system as a basis for a distributed or wide-area middleware. We discuss this issue further in section 7.

The key attributes of the tuple-based model are that objects can be shared using asynchronous communications with buffering and persistence automatically managed. Essentially, data residing in the shared objects can be decoupled from mutator programs. Our goal is to extend the functionality of existing implementations to take account of the unreliability and integrity issues that arise in a wide area distributed environment.

3 An Application Example

A classic example application that we have found useful in guiding our design and in testing our implementation is that of a (simplified) global banking system. We consider a (large) number of bank accounts and a progression of account transactions moving sums of money around. The system is successful if financial transactions all complete and if no money is lost from the system. This model is useful in our investigation of a a SuperSpace of separate Spaces. Individual banks can be represented as independent Spaces and the overall banking system by the SuperSpace. Individual customers making financial transactions are represented by client worker programs which issue object operations.

In the subsections below we present short fragments of “Java pseudo-code” illustrating the key components of our banking example.

3.1 Account object

The bank account is the `Entry` Object primitive that will be shared across our system. We are interested in the case where a large number of tuple objects are involved and we can investigate this through varying the number of bank accounts in the experiment. A good test of the transactional integrity and correctness of our experimental model is that no money leaks from the system.

```
import net.jini.core.entry.Entry;

public class Account implements Entry {
        public Integer accountNo;
        public Integer value;

        public Account() { ... }

        public Account(int accountNo, int value) {
                this.accountNo = new Integer(accountNo);
                this.value = new Integer(value);
        }
}
```

3.2 Account Creation

An interesting feature of the JavaSpaces implementation is the potential use of the leases mechanism provided by the Jini software. Leases allow a well defined interface to what is always a difficult tradeoff decision involved in writing distributed systems code. How does one incorporate time out information and how does one choose the timeout values. The Java/Jini Leases mechanism does not solve this problem but does provide a useful way to address the tradeoff. We use the `Lease.FOREVER` constant value to denote that the bank account `Entry` object should be left in the Space indefinitely. The code is surrounded by a try/catch clause in recognition of failure exceptions that may be thrown by the code accessing the managing Space.

```
try {
        Account newAccount = new Account(accountNo, value);
        space.write(newAccount, null, Lease.FOREVER);
}
```

3.3 Transaction Worker

We employ an agent called the Transaction Worker to establish bridging links between Spaces in our SuperSpace. The Transaction Worker acts as a portal for client programs to find the particular `Entry` object (bank account) they require. The Transaction Worker can encapsulate various cache management algorithms to decide which Space in the SuperSpace to

query first. It is also a suitable location for encapsulating various transactional integrity algorithms. We show here just the skeleton code for managing the account locks.

```
public void getLockRequests {

        space.read(requestLock);
        workerID = requestLock.ID;
        accountNo1 = requestLock.account1;
        accountNo2 = requestLock.account2;
}

public void getAccount {
        space.read(accountNo1);
        space.read(accountNo2);
        .......
        transObject newTransactionObject =
             new transObject(account1.accountNo,account1.value,
                             account2.accountNo, account2.value)
        space.write(newTransactionObject);
        space.take(accountNo1);
        space.take(accountNo2);
}
```

3.4 Client Worker

We show here a short skeletal client worker that is attempting to transfer money between two accounts. It employs a transaction object that can be used to implement various algorithms to ensure transactional integrity around the calculations and transfers. We are forced to implement this ourselves in the absence of a working transactions system in the JavaSpaces implementation.

```
requestLock newLock =
        new requestLock(ID, account1, account2);
space.write(newLock);

transObject newObject = new transObject();
newObject.ID = workerID;
space.read(transObject);
........
perform calculations
........
space.write(results);
space.take(transObject);
```

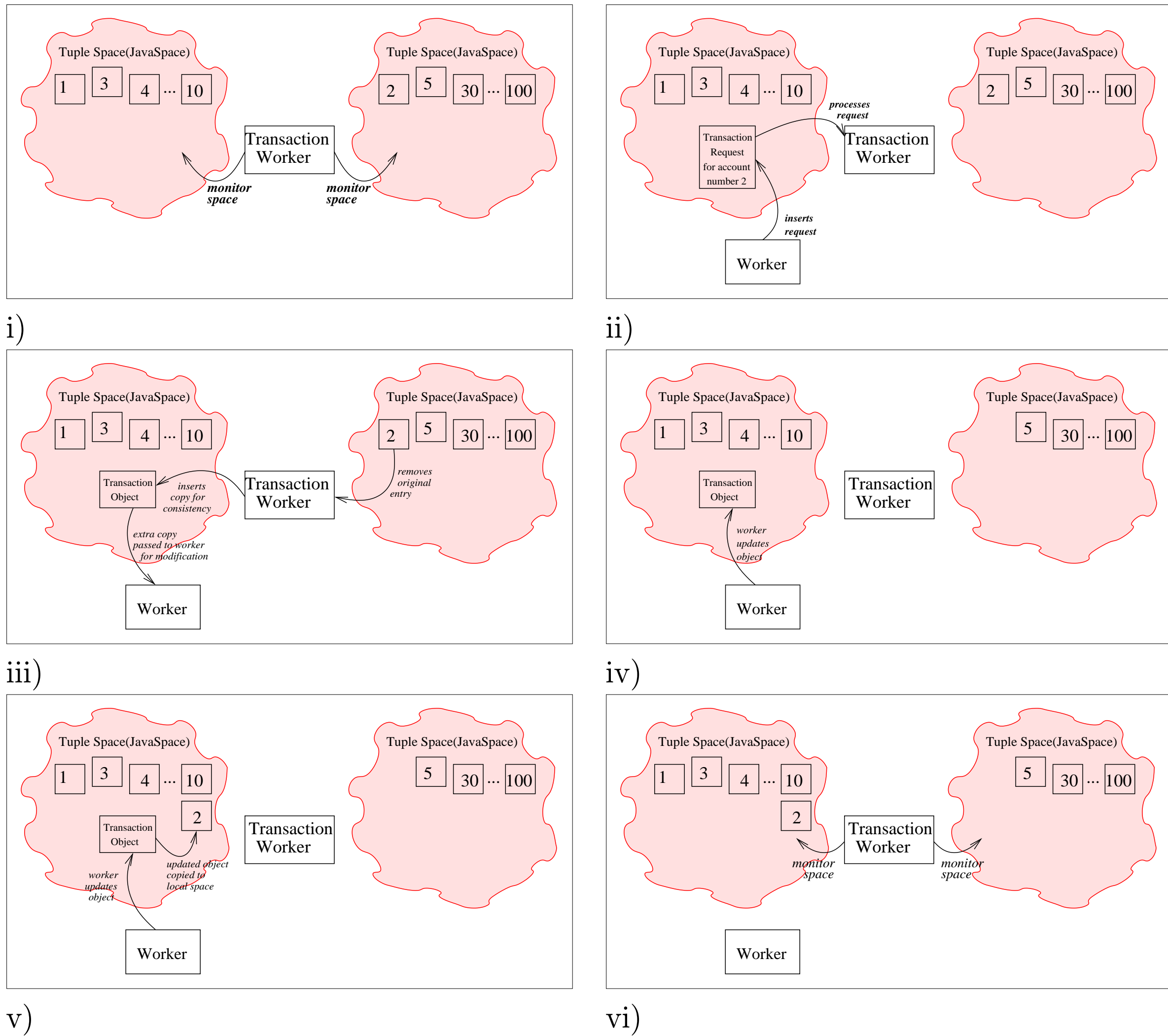

Figure 2: Sequence of operations involved when a client Worker requests write access to an account held in a remote Space. i) The Transaction Worker monitors all distributed spaces; ii) a request is made and the Transaction Worker notices; iii) the Transaction Worker removes the original account from the Space and places a temporary copy in the Space local to the Worker; iv) the worker updates the temporary copy; v) the updated temporary copy is converted into a 'real' account; vi) the Transaction Worker resumes monitoring the distributed spaces.

4 Bridging Spaces

Figure 2 shows in i) that two separate spaces have been started. These spaces have been populated with objects. Each object contains an account number (by which it is identified) and the value of the account, i.e. the balance. The objects have been placed in no specific order within the spaces. The transaction worker (TW, as described in previous section) knows of both spaces and waits for a request from a worker in either space.

In figure ii) a worker has been loaded. This worker knows only of the one space, this would be whichever responded to it first at load time. The worker wishes to access account number 2 so writes a request object into the space stating its own ID and the number of the account(s) it requires. The transaction worker takes this request from the space, it then looks at each space to find the account (iii). Once the transaction worker has found the account it reads a copy of the account and places that in the space the worker is using, it then removes the original entry to avoid other transactions on that account. The worker is then able to read the the temporary transaction object (iv), which is an exact copy of the original and perform the necessary calculations to the account.

Once the worker has completed its alterations to the account, it writes the account back into the space it is using (iv). It then removes the transaction from the space. The transaction has now been completed and the transaction worker waits for the next request by a worker (vi).

If there were to be workers connected to just the one space all the accounts would gradually migrate from the other space to the local space, due to the worker always writing back to their own space. This would help increase access times if the accounts were held locally. However if workers were to be loaded on the other space accounts would simply move between the two. This moving of accounts does not have a discernible slowing effect due to the already random nature of the location of objects. We are investigating the costs should the pattern of transactions cause the account to be in continual (thrashing) migration.

At all times during the transactions a copy of the account is stored within the space, this is done to ensure no loss in the event of failure. The space is persistent, so if the space were to go down it could be restored with all of the objects that had populated it. Transaction objects can then be used to restore the space to a consistent state should a failure occur.

The transaction worker or agent can bridge two or more spaces but can also act in cooperation with other instances of the TW. Critical aspects of the TW are the algorithms and approaches it uses to search for the requested object.

Polling is one approach that the TW can use to discover request objects created by the workers and when locating account objects. This same process can also be done using an event driven method. The TW can contain some intelligence to order the spaces it will search/manage.

The transaction worker acts essentially as an object portal to the SuperSpace. There are important issues associated with how well connected the SuperSpace is. The SuperSpace is a graph of Spaces, each managed in the normal way. Participating Space nodes of the SuperSpace are connected via TW agents. The SuperSpace can be fully connected in a number of ways.

5 Dynamic Space Participation

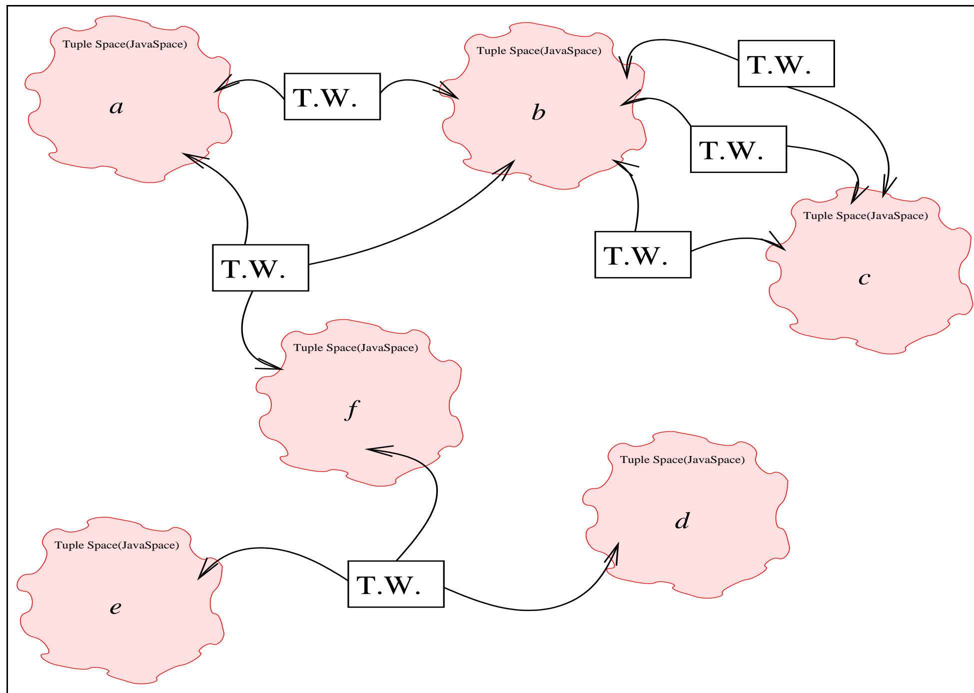

Figure 3: Use of multiple Transaction Workers to fully connect the Graph of Spaces in the SuperSpace, and to provide additional load balancing for some client workers.

Figure 3 shows two key ideas for utilising our model in a wide area middleware. Firstly our Transaction Worker (TW) component is designed so that multiple instances of it can be used to bridge a subset of the SuperSpace's Spaces. This can enhance the performance by load balancing client transactions amongst several TW's. Secondly, it is important for wide area system that the various participating Spaces form a fully-connected Graph (or SuperSpace).

A new Space can join the SuperSpace if a TW agent is established to bridge it to any one of the existing Space participants. There are of course some interesting algorithmic approaches needed to propagate knowledge of the joining (or departing) space to the SuperSpace. Building on our dynamic cluster management ideas [6] we are investigating various resource discovery algorithms for ensuring full connection in the case where some Spaces dynamically join or depart the SuperSpace.

6 Scoping JavaSpace Performance

Figure 4 shows the results of a simple experiment to time the write, read and take operations for a JavaSpace running on a Sun Blade 2000 system. The most significant feature of the plot is that the curves are not linear. They do appear to be piece-wise linear however and we are investigating the possible effects of the various knee-points. As one might expect, the write operation is the most expensive and the read is least expensive. Removing operations from the Space using the take operation is intermediate in cost.

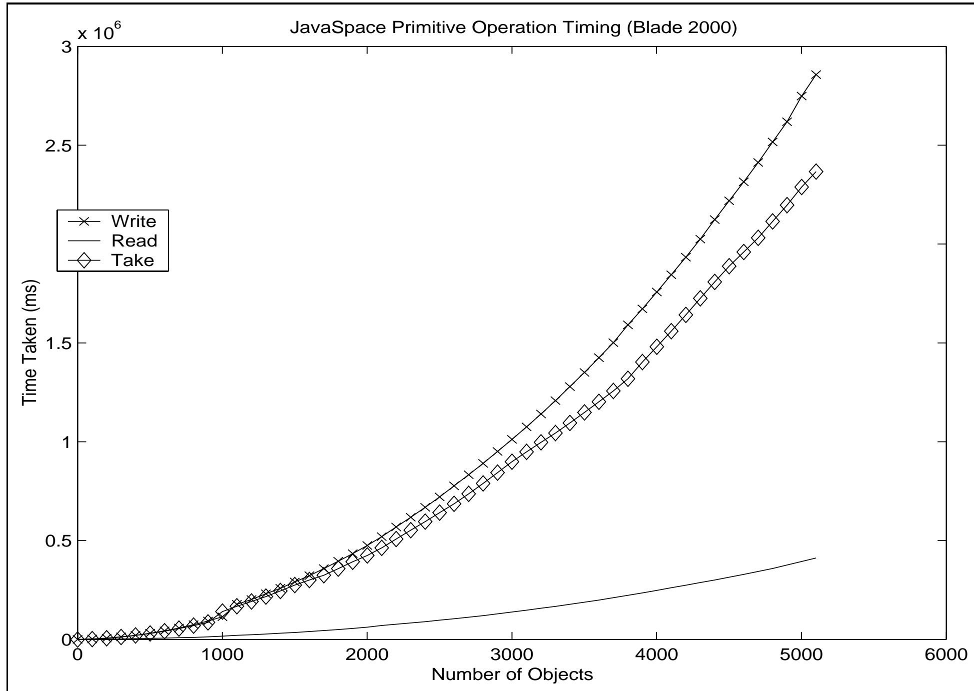

Figure 4: Timings of JavaSpace primitive operations against number of `Entry` objects involved.

These results emphasise that object management using the Space model comes at some considerable performance cost. These operations are not as cheap as for example simple message passing ones would be. They also reflect the fact that the JavaSpace system is implemented using Java Remote Method Invocation (RMI) technology - which adds considerable overheads to the communications. Nevertheless we believe that the cost is acceptable for the convenience of using the Space model and for the transactional integrity of the primitive operations.

7 Discussion and Conclusions

Our experiences in using JavaSpaces have been that at least 60MBytes of RAM is typically necessary. In practice we have found that an entry level of 128MBytes is needed to run a JavaSpace without interfering with other system processes. This was problematic in recent years due to desktop system limitations. Now, however, with the increasing availability of memory and desktop systems frequently having well in excess of this amount, it is now feasible to consider the use of JavaSpaces as a common component of middleware.

We also note that the storage limitations of the JavaSpace is bound by the default memory limit of the underpinning Java Virtual Machine. Explicit intervention is therefore required to increase the available stack and heap memory.

Our experience shows that the space model is a useful one for building wide area object management systems. However, we note there are still some significant limitations in the current Java tuple-based products. We have particularly missed the support for transac-

tional integrity of compound operations. In consequence we have experimented with various transactional protection algorithms including multi-phase locking and optimistic concurrency control. As it turns out, this approach has proved necessary to bridge transactional operations between multiple spaces. We also observe some performance limitations of the JavaSpace system but these seem quite acceptable as a tradeoff in view of the additional capabilities provided.

The concept of a space of spaces opens up the possibility of joining objects existing on systems in different physical locations. Multiple spaces also offers the ability to group objects into appropriate spaces for ease of access. By using a variation of our Transaction Worker we believe that a dynamic super space environment can be constructed to allow participating spaces to join or depart at arbitrary times.

The ability to use multiple Transaction Workers between spaces will increase the overall throughput of the system. Using these custom Transaction Workers offers the option for further experimentation with the transactional algorithms.

Acknowledgements

The Distributed and High Performance Computing Group is a collaborative venture between the University of Wales, Bangor and the University of Adelaide in South Australia.

References

[1] Ian Foster and Carl Kesselman, editors. “The Grid: Blueprint for a New Computing Infrastructure”, Morgan Kaufmann Publishers, Inc., 1999. ISBN 1-55860-475-8.

[2] Eric Freeman, Susanne Hupfer and Ken Arnold, “JavaSpaces Principles, Patterns, and Practice”, Pub Addison Wesley, 1999, ISBN 0-20130955-6.

[3] David Gelernter. “Generative Communication in Linda”, In *ACM Transactions on Programming Languages and Systems*, vol 7(1) pp. 80-112 January 1985.

[4] GigaSpaces. “GigaSpaces Cluster White Paper.”, Available http://www.j-spaces.com/ Last visited October 2002.

[5] K.A. Hawick, H.A. James, A.J. Silis, D.A. Grove, K.E. Kerry, J.A. Mathew, P. D. Coddington, C.J. Patten, J.F. Hercus, and F.A. Vaughan, “DISCWorld: An Environment for Service-Based Metacomputing,” *Future Generation Computing Systems (FGCS)*, 15:623–635, 1999.

[6] K.A.Hawick and H.A.James “Dynamic Cluster Configuration and Management using JavaSpaces”, *Proc IEEE Cluster 2001, Newport Beach, USA, 8-11 Oct 2001.* also DHPC Technical Report DHPC-103.

[7] K.A.Hawick, D.A.Grove, P.D.Coddington and M.A.Buntine Commodity Cluster Computing for Computational Chemistry In *Internet Journal of Chemistry*, 2000, 3, 4., http://www.ijc.com/ also DHPC Technical Report DHPC-073

[8] IBM. "TSpaces Intelligent Connectionware," Available from http://www.almaden.ibm.com/cs/Tspaces/faq.html Last visited October 2002.

[9] JavaWorld. JavaWorld Home Page. Available from http://www.javaworld.com Last visited October 2002.

[10] J.A. Mathew, H.A. James and K.A. Hawick, "Development Routes for Message Passing Parallelism in Java," *Proc. ACM 2000 Java Grande Conference*, San Francisco, June 2000, pp 54-61.

[11] Sun Microsystems. "JavaSpaces Technology," Available from http://java.sun.com/products/javaspaces/ Last visited October 2002.