Mobile Co-ordination: Providing fault tolerance in tuple space based co-ordination languages.

Antony Rowstron

Laboratory for Communication Engineering, Engineering Department,
University of Cambridge, Trumpington Street, Cambridge, UK
aitr2@eng.cam.ac.uk,
http://www-lce.eng.cam.ac.uk/~aitr2

Abstract. In this paper we describe the concept of *mobile co-ordination*, a general purpose approach to overcoming failure of agents when using distributed tuple spaces. We demonstrate why mobile co-ordination is better than using existing techniques such as transactions, how mobile co-ordination can provide extra functionality in the form of *agent wills*, and how the framework to provide this can be implemented in Java and can be used with multiple different tuple space co-ordination languages. Experimental results are presented to show performance gains made when mobile co-ordination is used.

1 Introduction

Early tuple space based languages, such as Linda[1], suffered from poor agent fault tolerance. Since Anderson et al.[2] first proposed the idea of using of transactions in Linda it has become widely adopted, for example in PLinda[3], Paradise[4], JavaSpaces[5] and more recently in TSpaces[6]. In this paper we advocate the use of mobile co-ordination instead of transactions.

A tuple space based system needs two fault tolerance mechanisms, one at the system level for server fault tolerance and one at the user level for application writers to provide fault tolerance at the application layer. In particular at the application layer fault tolerance is required to provide protection against the failure of an agent when the agent has removed one or more tuples from a tuple space that are required by other agents in order for the system as a whole to continue. For example, consider an agent fails whilst performing a series of tuple space operations that together created a higher level co-ordination operation as shown in Figure 1 which shows some Linda operations which increment a shared counter.

```
in("COUNTER"_string, ?x_integer);
out("COUNTER"_string, ++x);
```

Fig. 1. A co-ordination operation.

If the agent performing this co-ordination operation was to fail having performed the `in` then the incremented counter would never be inserted. This would mean that any other agents using this counter would block forever when they next try to read it.

Transactions can be used to overcome this problem, as will be shown later, however, neither the tuple space based co-ordination languages BONITA[7] or WCL[8] developed by the author used transactions, because we considered transactions a poor solution to the problem and at the time we left the problem unaddressed. Here we now address the issue and argue that in many situations resilience to failure provided by transactions is not sufficient. Our proposed solution, mobile co-ordination not only provides the kind of fault tolerance provided by transactions, without altering the underlying semantics, but it also enables the concept of an agent *will* which provides a mechanism for 'tidying' a tuple space up should an agent fail, where a traditional transaction can not be used. Mobile co-ordination is general enough to be applied to many existing tuple space based co-ordination languages that use Java as the host language.

Application fault tolerance is an important problem because the number of tuple space based co-ordination languages for use over the Internet has increased considerably in the last few years, with many companies attempting to create such languages, eg. IBM[6] and Sun[5]. The solution presented in this paper is not only novel in terms of overcoming this problem, but has been extended to introduce the novel concept of agent wills.

In the next section a description of transactions as used in tuple space based co-ordination languages is given, and the short cummings described. In Section 3 we describe the concept of Mobile Co-ordination, using a simple example, and discuss why it provides fault tolerance. In Section 4 the use of agent wills is described. In Section 5 we describe the implementation and discuss the performance issues, and show how, surprisingly, the use of mobile co-ordination is far more efficient than the use of transactions in most cases.

2 Transactions

Transactions have been used for many years in databases and Anderson et al.[2] proposed using this concept in Linda, and this was first done in PLinda and then subsequently in many co-ordination languages. Most implementations provide a similar approach to the transaction implementation. Two new primitives are added to the base tuple space access primitives, which are `start` and `commit` primitives.

The `start` primitive causes the server managing the tuple spaces to retain a copy of all tuples being removed and to hold all tuples being inserted by the agent which performed the `start`. When the `commit` command occurs all the tuples being held because the agent removed them are discarded, and any inserted tuples are actually placed into the tuple space and then become visible at that point to the other agents using the tuple space. This way, if the `commit` is never reached the inserted tuples do not appear in the system and any removed

tuples can be replaced. However, any inserted tuples do not appear to other agents until the `commit` is performed.

The problem with this is that the use of transactions alter the underlying semantics of the co-ordination operations they are placed around. This is shown in Figure 2.

Fragment One	Fragment Two
$\text{out}(10_{integer})$;	$\text{in}(10_{integer})$;
$\text{in}(11_{integer})$;	$\text{out}(11_{integer})$;

Fig. 2. Example of transaction problems.

The two fragments of pseudo code shown in Figure 2 are assumed to be performed on the same tuple space, and represent a trivial yet important co-ordination construct using tuple spaces. The co-ordination construct is an explicit synchronization between the two fragments. If the `start` and `commit` are placed around the co-ordination constructs in Fragment Two then this does not alter the semantics. However, if the `start` and `commit` are placed around the co-ordination constructs in Fragment One the two fragments will deadlock. The tuple inserted in Fragment One into the tuple space will not appear until after the tuple inserted in Fragment Two has been read, but this can not occur until after the tuple inserted in Fragment One appear, thus the fragments deadlock.

This altering of the semantics means that the outcome of the co-ordination operations is dependent on whether they are performed inside a transaction or outside a transaction. This is not a desirable side effect, and the primary reason why transactions were not adopted in either BONITA or WCL. It also means that a programmer can not place all the co-ordination operations required in a transaction. For example, consider a program which needs to insert a tuple into a tuple space to signify that the agent is present, and when the agent has finished it removed the tuple. An example of this can be seen in a chat tool a chat client may place the name of the user in a tuple space, so that other users can ask who is currently using the chat tools system. If the agent unexpectedly then this tuple needs to be removed. The problem with a single transaction for this is that the name tuple will never be seen by other agents, until the transaction commits (when it terminates).

To a lesser extent there is another problem, which is how long should a server wait in between the `start` and deciding that agent has died? Should this be specified by the user? Should it be assumed that the communication layer between the agent and the server has some notion of knowing when the other end has died?

Both these problems are clear if you look at the JavaSpaces[5] specification which describes the behavior of the primitives if performed as part of a transaction or not. Furthermore, the description of the basic primitives include descriptions of how they interact with transactions, and a description of tim-

ing flags used with the primitives to control how they interact with transactions. This increases the complexity of the language, and makes it from a simple model into a complex one, with very subtle behavior and interaction possible.

So, transactions alter the semantics of the co-ordination constructs performed within them, and cannot provide the level of fault tolerance required in all applications, due to the fact the tuples do not appear until the transaction commits. We believe the idea of mobile co-ordination overcomes all these problems, and furthermore is in general more efficient and faster.

3 Mobile Co-ordination

Mobile co-ordination involves the movement of co-ordination primitives that make a particular co-ordination operation to the server which stores the tuple space. A co-ordination operation is a high-level co-ordination operation composed of several tuple space access primitives. If the all the co-ordination *primitives* reach the server before any are executed, the entire co-ordination operation will be executed. In other words, this provides a 'all or none' execution of the primitives which make the co-ordination operation. The underlying assumption is that the server is reliable. This provides us with a mechanism to provide fault tolerance at an application level. By moving arbitrary (small) segments of programs containing multiple tuple space access primitives, and by ensuring that the segment is not executed until all the segment and associated state has been transfered to the tuple space server an application can provide fault tolerance.

The aim is to create a framework that supports this, without comprising the simplicity of the tuple space model. At this stage it should be noted that before creating the current implementation many other approaches were considered, including creating some sort of scripting language in which to embed the tuple space primitives. These would have been host language independent but given the current prominence of the Java language for Internet computing it seems acceptable to create a system which works only with that language.

A detailed explanation of how the framework is created in Java is given in Section 5. However, an overview from an application developers perspective is given now. The developer creates a class that contains the tuple space access primitives which they wish to be performed as a single co-ordination operation. In the agent an instance of that class is created, and any necessary information is initialized and stored within the created object. The programmer then simply passes this object to a method associated with the object providing access to the tuple spaces. This manages the moving of the object (its state) and the necessary class files to the tuple space server. The class that contains the mobile co-ordination code must implement an interface *MobileCoordination* which specifies there is a method called `coordination()` in the object. The tuple server server calls this method. The interface also specifies that the method `coordination()` must return an object of the type *Tuple*. This is passed back to the agent. More detail about the implementation is provided in Section 5.

The fault tolerance is provided, because from the server's point of view, it either receives the class file, and the object state in its entirety or it does not receive them at all. If the socket fails before both are received then the server does not (indeed can not) create the object because it catches the exception and terminates the operation. If the server receives all the information correctly it recreates the object and calls the `coordination()` method, thereby performing the mobile co-ordination segment. The tuple that this method returns is passed back to the agent. If the socket has died in between the code starting and ending it does not matter, and the result is 'lost', however, the high-level co-ordination operation will have been executed in its full. Hence it provides either 'all or none' execution of the mobile co-ordination segment.

It should be noted that moving the co-ordination operations to the tuple space server does not alter the semantics, because (currently) the agent thread performing the co-ordination operation is blocked until the result from the migrated code has been returned. This means that it is similar to any other blocking primitive in Linda. It should be noted that it is quite possible to execute the mobile co-ordination operation at the agent, by simply calling the `coordination()` method – however this will not provide the fault tolerance. The reason why the semantics of the tuple space access primitives performed in the mobile co-ordination code is because the system provides 'all or none' execution of the operations *not* atomicity. Other agents are able to interact with and use the tuples that the mobile co-ordination generates. Indeed, the mobile co-ordination section can generate tuples and then subsequently consume them[1]. It is the fact that the tuple space access primitives that compose the co-ordination operation are not atomically executed that allows the semantics not to be altered.

The mobile co-ordination code is expected to handle all exceptions that occur. If an exception is raised which is not handled by the code then an empty tuple is returned to the agent which performed the operation. There are currently no restrictions on the operations that can be performed in the code. However, it makes no sense to perform any I/O operations. Objects can be instantiated within the code once it is executed, however, in order to guarantee the 'all or nothing' approach the class files must be available locally. However, when the object is first migrated, pre-instantiated objects can be transfered.

In order to demonstrate fully how mobile co-ordination performs a simple example is used which is based loosely on the idea of a talk tool. The talk tool does not use a chat server but instead is peer-to-peer, with each talk tool client manipulating the conversation directly in the tuple space. A full example of such a tool using a tuple space language is given in Rowstron[7]. Each client places a tuple in a tuple space with the name of the user in it then it starts to add lines to the conversation and then when finished the tuple containing the users name is removed.

[1] At its limit the entire agent could be migrated, however, this is neither necessary nor wanted as it would increase the load at the tuple space server and reduce performance of the system (we discuss this further in Section 7.)

The necessary routines to manage the mobile co-ordination are, in the current implementation, added to the class *TupleSpace*. As well as being able to perform the normal Linda operations (`out`, `in` and `rd`) using the class a number of new methods are added; `executeSafe`, `createWill` and `cancelWill`. It should be noted that `executeSafe` is not equivalent to an `eval` operation in Linda.

Figure 3 shows the main program. A very simple centralized tuple space kernel has been used in this example with the Linda style access primitives of `in`, `rd` and `out`. These operations return objects which are instantiations of *Tuple*, and the class *Tuple* provides a method called `getField` which returns the object stored in the tuple at position specified. Line *2* shows the creation of a tuple space. Lines *5* and *6* create a will for the agent, which will be discussed in the next section. Lines *8* and *9* create an instance of the class InsertLine, which is our high-level co-ordination operation, and insert the necessary values in it. Line *10* causes the object to be migrated to the server, and the result is returned. If the user had wished to execute this locally all they would have done is replace Line *10* with:

```
Tuple t = InsertLine.coordination();.
```

Then Lines *11* and *12* read back the tuple inserted by the mobile co-ordination operation. Finally, in Line *14* the will is canceled, and this will be described later.

```
1  public class TestMobile {
2    TupleSpace gts = new TupleSpace("ts host",8989);
3
4    public TestMobile() {
5      MyWill theWill = new MyWill(gts,"Antony Rowstron");
6      gts.createWill(TheWill);
7      gts.out(new Tuple("Antony Rowstron"));
8      InsertLine mobileCoord = new InsertLine();
9      mobileCoord.insertData(GTS,"Antony","Hello I have joined!");
10     Tuple t = gts.executeSafe(mobileCoord);
11     t = gts.rd(new Template((Integer)t.getField(0),"Antony",
12                       new Formal("java.lang.String")));
13     gts.cancelWill();
14   }
15
16   static public void main(String args[])  new TestMobile();
17 }
```

Fig. 3. Example - Main Class.

Figure 4 shows the high-level co-ordination operation. In Figure 1 in Section 1 the example used is a global counter stored in a tuple space. This is the same idea used to store and manage the conversation in the talk tool. Therefore, the counter

tuple is removed, incremented and reinserted, and the necessary line of text is inserted. Lines *4* and *5* allow the object to be set up with the tuple space, and the text for the tuple to be inserted. Lines *7* to *14* provide the method required for *InsertLine* to implement *MobileCoordination*. This removes the counter tuple (line *9*) increments the value and reinserts it (line *11*), then inserts the line we wanted to add (line *12*) and returns a tuple containing the value of the line we inserted (line *13*).

```
1  class InsertLine implements MobileCoordination, Serializable {
2    TupleSpace ts; String name, Text;
3
4    public void insertData(TupleSpace ts, String name, String text) {
5      this.ts = ts; this.name = name; this.text = text; }
6
7    public Tuple coordination() {
8      Tuple count; int cnt;
9      count = ts.in(new Template("C",new Formal("java.lang.Integer")));
10     cnt = ((Integer)count.getField(1)).intValue();
11     ts.out(new Tuple("C",new Integer(cnt+1)));
12     ts.out(new Tuple(new Integer(cnt),name,text));
13     return new Tuple(new Integer(cnt));
14   }
15 }
```

Fig. 4. Example - Mobile Co-ordination Class.

This has shown how the mobile co-ordination works, and has also demonstrated that it is not complicated. It should be noted that the object that is sent to the server is not strictly migrated, it is replicated. A copy of the object and its state remain at the agent.

4 Using mobile co-ordination for Agent Wills

We have already demonstrated how mobile co-ordination can be used to provide the sort of fault tolerance that traditionally is provided by transactions in tuple space based systems. In the above example we are guaranteed not to loose the counter tuple due to agent failure. However, we have not dealt with how the tuple containing the users name can be removed on agent failure (Figure 3, line *7*).

Using mobile co-ordination we have also added the concept of an agent will for an agent. An agent will is a set of tuple space access primitives that are executed when the *tuple space server* managing the tuple spaces decides that an agent owning the will has failed. Currently, a loose interpretation of this is taken, in so far as we allow a will for *every tuple space* (handle) that the agent has. This appears to allow more flexibility than simply allowing one. An agent

has the ability to cancel (or execute) a will it has associated with a tuple space explicitly.

In order to demonstrate this consider the example we used in the last Section. In Figure 3 line *5* the 'will object' is instantiated and its internal values are passed to it. Line *6* sets the will in operation. This is achieved by passing it to the tuple space server/manager, which then stores it until required. Finally, line *13* cancels the will. However, it should be noted that it is often the case where the programmer actually wants the will explicitly to be executed (as in this case) to remove the name tuple.

```
1 class MyWill implements MobileCoordination, Serializable {
2   TupleSpace ts; String name;
3
4   public MyWill(TupleSpace ts, String name) {
5     this.ts = ts; this.name = name; }
6
7   public Tuple coordination() {
8     ts.in(new Template(name)); return null; }
9 }
```

Fig. 5. Example - The Will Class.

Any high level co-ordination operation stored in a separate class can be used as an agents' will. There is no difference between a will class an a normal co-ordination operation that is to be performed fault tolerantly. Figure 5 shows the Class used as the will. All it does is remove the tuple containing the users name. It should be noted that it returns a null pointer. It could return a tuple, but given that the connection to the agent has been lost in order for it to be executed there is little point. However, currently it is possible for the agent to request the server to execute the will. When this is done, the result is returned to the user agent. This would be achieved by replacing line *13* in Figure 3 with the line:

```
Tuple tup = gts.executeWill();.
```

It should be noted that if the will was not canceled explicitly then the will would be executed anyway when the agent terminated. This means the need for an explicit execution command may be unnecessary.

To summarize, a will is considered as a high-level co-ordination operation that is only executed if the connection between the agent and the server (in the servers opinion) has failed (or if the agent explicitly asks for it to be executed).

5 Implementation

In this section we present a very brief overview of how the mobile co-ordination system is implemented. The implementation is relatively straight forward and

uses the many unique features of Java. In essence, whenever a class is instantiated a *ClassLoader* is used to load the class. At the server end the implementation subclasses the ClassLoader to provide a NetworkClassLoader which gets the class (and the object state) from the agent. Therefore, to load a class the agent provides to the server a socket name and a port number. The agent creates an instance of a serializable class that contains the class file code and the serialized state of the object to be migrated. This is then transfered to the tuple space server. It should be noted that the agent keeps a record of which class files have been passed to the tuple space server, and if the current object is an instantiation of a class whose code has already been passed to the server, then the class file is omitted. This saves on the amount of information passed to the server.

In order for the server to be able to call the necessary methods in the migrated object, the migrated object must implement an interface (currently called *MobileCoordination*). All this specifies is that there should be a method called Coordination() in every object passed.

In order to allow the co-ordination operations in the migrated object to be the same as those used locally, an object which is an instantiation of the TupleSpace class (this holds information about where the tuple space is stored) is able to detect whether it is at the server or at the agent using a hack. This means a TupleSpace object is able to modify its behavior accordingly, if it is at the server then it accesses the data structures directly, and if it is at the agent is passes the instructions through a socket to the tuple space server where they are performed and then the results returned.

6 Performance

Initial thoughts on the mobile co-ordination led us to believe that although the use of mobile co-ordination would provide fault tolerance it would be more expensive (time wise) than using transactions. In the best case it may be possible to piggyback the transaction start and transaction end messages on other messages, therefore requiring no real overhead.

However, it turns out that using mobile co-ordination can be many times more efficient. In general, if more than one tuple read/removal operation is performed then it is more efficient to use mobile co-ordination. In order to demonstrate this, consider a simple program which inserts using (an unordered) `out` a number tuples each containing a counter and a random number. The code to insert the tuples was written as a high level co-ordination operation which could be performed as a mobile co-ordination operation. The inserted tuples are then read using `rd` and the random numbers summed. Again, this was created as a mobile co-ordination operation. The agent then executed the mobile co-ordination both remotely (providing fault tolerance) and locally (not providing fault tolerance). Hence we are able to compare the effect (in the absence of faults) of migrating the mobile co-ordination.

Figures 6, 7 and 8 show the results for when between 1 and 200 tuples are inserted, both over a LAN and over a WAN. In the case where the mobile co-

ordination is enabled the size of the class file transferred for the Insert operations is 784 bytes with the object state being a further 187 bytes. For the adding component the class file size is 887 bytes with the object state being a further 176 bytes. Therefore this is approximately 1K to be transferred. The results for a LAN were gathered over a 10 MB/sec Ethernet, using a Pentium Pro 200 MHz PCs running Linux. The WAN results were gathered using an Indy Workstation at York, UK to a Pentium Pro 200 MHz PC Linux running computer at Cambridge.

It should be noted that Figures 6(a) and 8 show the same operation (Insertion over a LAN) however Figure 8 shows the use of an `out` primitive which provides *out ordering*[2][9]. The provision of *out ordering* means that more messages need to pass between the server and the client. However, the approach taken in the implementation in naive, compared to the kernel described in Rowstron[9]. Therefore, Figure 8 represents the worst case. It should also be noted that Figures 6(a) and 7(a) the insertion time seems to be independent of whether a LAN or a WAN is being used. This is because there are no acknowledgements being returned. Therefore, the time represents the time taken to insert that number of tuples into a socket, which, given the size of the data being sent can probably store it in local buffers.

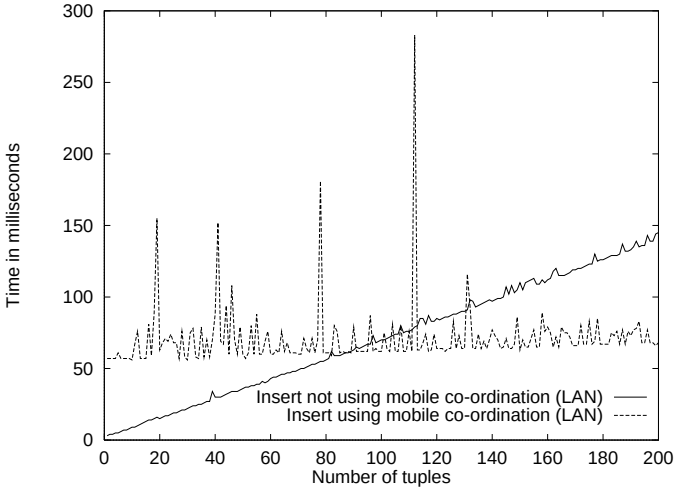
In all the cases the use of mobility appears to take an almost constant time, regardless of the number of operations performed. This can be explained by considering the time tuple space access can be performed on the server and across the network. On a Pentium Pro 200 MHz PC running Linux it takes approximately 110μS to read a tuple from a tuple space in the worst case. The time taken to send a message (containing the code and data) over a network, and then for the Java Virtual Machine to unpack and instantiate the class is constant regardless of the number of operations performed. The time difference of performing 1 or 200 tuple space operations is insignificant.

The results show that if more than one single tuple space operation requiring a tuple to be returned is to be performed (eg. not just a single or multiple `out` primitives), then the mobile co-ordination will provide better performance. In this case, because the add routine read the number of tuples inserted from a tuple in the tuple space and then read that many tuples the mobile co-ordination is always quicker.

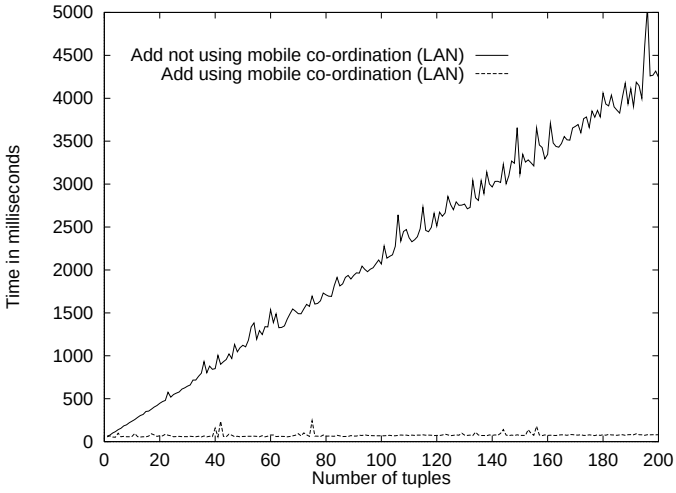
This conclusion may seem a little surprising, but if we consider the Maximum Transmission Unit (MTU) for TCP/IP is 1500 bytes[3], and therefore, for sending a 1K packet has much the same effect as sending a 200 byte packet. Given the use of synchronous tuple space access primitives (like `in` and `rd`) the agent has to block computation whilst the result tuple is returned. Therefore, the second tuple space access operation can only begin when the first one terminates. In the case of a mobile co-ordination segment, the operations at the server can access the data structure directly, therefore the cost of communication is removed

[2] This is where the `out` primitive is implemented in such a way that it requires an acknowledgment from the server storing the tuple before the next `out` is performed.

[3] For IPv6 the *minimum* size is 576 bytes.

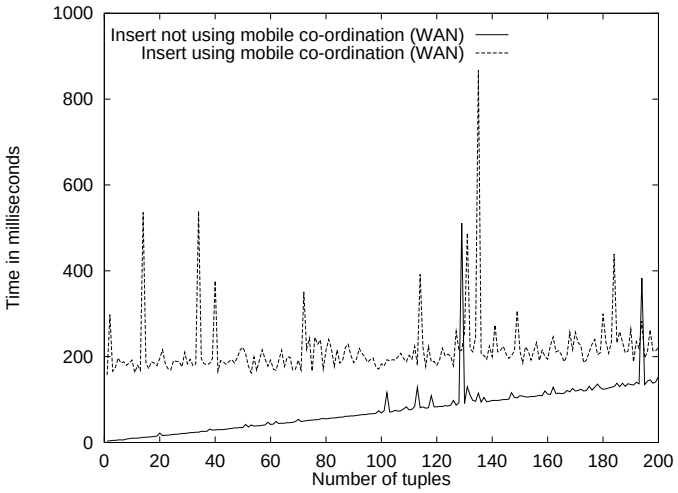


(a) Insert using mobility and not using mobility over a LAN.

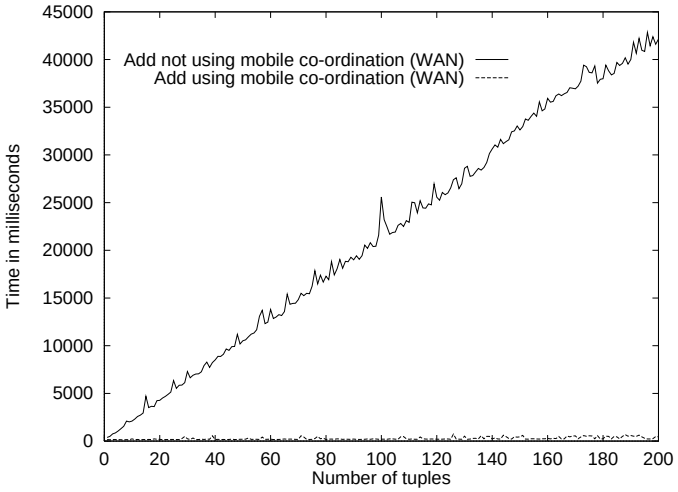


(b) Add using mobility and not using mobility over a LAN.

Fig. 6. The effect of using mobility and not using mobility over a LAN.



(a) Insert using mobility and not using mobility over a WAN.



(b) Add using mobility and not using mobility over a WAN.

Fig. 7. The effect of using mobility and not using mobility of a WAN.

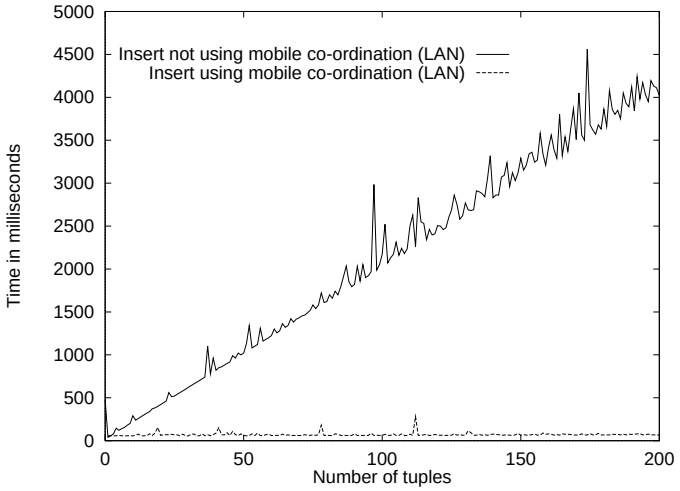


Fig. 8. Insert (with `out` ordering) using mobility and not using mobility over a LAN.

for the tuple space access. At the tuple space server the time taken to access the data structure is very small compared with the time taken to perform the communication, and therefore multiple tuple space operations can be performed in apparently the same time as a single remote tuple space operation. On a Pentium Pro 200 MHz PC runing Linux it takes approximately 110μS to read a tuple from a tuple space in the worst case. Therefore, 9 tuple space primitives can be performed in under 1ms. Using the Unix `ping` tool between the machine at Cambridge and the machine used at York provides the following results (for 22 packets); Min time is 11.2 ms, average is 13.7 ms and the max is 24.7 ms. This puts the time to perform nine operations in the full context of expected network latency.

7 Future work

This paper describes the use of mobile co-ordination to provide fault tolerance. The work to date has concentrated on using as single tuple space server. Without doubt multiple servers will have to be used in real implementations. However, transactions have similar problems, and we could make restrictions on tuple spaces which could be accessed within the co-ordination operations migrated. This is an area which we are still considering.

Also, all the performance figures are given from the point of view of the agent. This does not consider extra tuple space server load. Some of this extra load will be offset by reducing the need to construct and deconstruct packets of tuples being sent to the agent. Some control of the amount of CPU time a migrated co-ordination operation can consume needs to be added, otherwise there is the potential for abuse. This is another area still under consideration.

8 Conclusion

In this paper we have demonstrated how the concept of Mobile co-ordination can be used to provide fault tolerance in tuple space based co-ordination languages. Mobile co-ordination provides the same fault tolerance that the use of transactions in many tuple space languages provides, but without the drawback of altering the semantics of the primitives. Also, the same basic technique allows agents to register 'wills' with the tuple space system, that are executed if the agent dies. This has been shown in this paper to extend the fault tolerance support. An example program has been used to demonstrate that the addition of mobile co-ordination does *not* increase the complexity of the language, and therefore, does not extend the load placed on the programmer using it.

The ideas of of mobile co-ordination are applicable to any tuple space based language which uses Java. The demonstration uses a traditional Linda implementation, but the same system has been used with WCL. It can easily be introduced into either TSpaces or JavaSpaces.

It is also interesting that the mobile co-ordination uses the ideas of mobile objects to achieve something that is not feasible without the use of mobile objects (wills).

In the last year the importance of tuple space based co-ordination languages has become very visible with many companies announcing systems which utilize tuple space technology. The need to solve the few remaining drawbacks of such systems has driven this work. The work described here presents a novel approach to providing fault tolerance in one of the best known classes of co-ordination language.

Acknowledgements

I would like to thank Prof. Andy Hopper and the Olivetti and Oracle Research Laboratory for funding me. I would also like to thanks Dr. Alan Wood at York University for allowing access to his computing facilities. I would also like to thank Dr. Stuart Wray for his many long chats on the subject of mobile co-ordination.

References

1. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
2. B. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In *Research Directions in High-Level Parallel Programming Languages*, LNCS 574, 1991.
3. K. Jeong and D. Shasha. Persistent Linda 2: a transaction/checkpointing approach to fault-tolerant linda. In *Proceedings of the 13th Symposium on Fault-Tolerant Distributed Systems*, 1994.
4. Scientific Computing Associates. *Paradise: User's guide and reference manual.* Scientific Computing Associates, 1996.

5. Sun Microsystems. Javaspaces specification. Avaliable from Sun Microsystems WWW Site (http://java.sun.com/products/javaspaces), 1998.
6. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. TSpaces. To appear in *IBM Systems Journal*, August, 1998.
7. A. Rowstron. Using asynchronous tuple space access primitives (BONITA primitives) for process co-ordination. In *Coordination 1997*, pages 426–429, 1997.
8. A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1998.
9. A. Rowstron. *Bulk primitives in Linda run-time systems.* PhD thesis, Department of Computer Science, University of York, 1997.