ELSEVIER

Science of Computer Programming 46 (2003) 137–162

Science of Computer Programming

www.elsevier.com/locate/scico

Using mobile code to provide fault tolerance in tuple space based coordination languages ☆

Antony I.T. Rowstron

Microsoft Research Ltd., 7 J.J. Thomson Avenue, Cambridge CB3 0FB, UK

Abstract

Tuple space based coordination languages suffer from poor fault tolerance to agent failure. Traditionally, transaction type mechanisms have been adopted in them to provide this type of fault tolerance. However, transactions change the semantics of the tuple space access primitives performed within them and do not provide a sufficient level of flexibility.

We propose using mobile coordination, which utilises mobile code, as an alternative mechanism for providing better fault tolerance to agent failure. The use of mobile code is transparent to the application programmer. Mobile coordination provides the same level of fault tolerance as transactions, but it also introduces the concept of agent wills. This allows coordination patterns to be performed in a fault tolerant manner which cannot be performed in a fault tolerant manner using transactions.

Mobile coordination is described in detail. The API for a prototype centralised implementation is presented. It is shown that mobile coordination provides better support and better performance than the traditional approach of using transactions. Implementation strategies for a distributed implementation are also discussed. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Linda; WCL; Tuple spaces; Mobile objects; Mobile coordination; Transactions; Mobile code; Fault tolerance; Agent wills

1. Introduction

In the last few years there have been many new tuple space based languages and systems proposed, including WCL [21], BONITA [23], PageSpace [10], TuCSoN [17], MARS [5], Jada [9], TSpaces [26], KLAIM (KLAVA) [16,3], Lime [18] and JavaSpaces [25]. All these languages or systems use coordination primitives that are

☆ Expanded version of [22] presented at Coordination'99, Amsterdam, 1999.

E-mail address: antr@microsoft.com (A.I.T. Rowstron).

0167-6423/03/$ - see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0167-6423(02)00089-8

descendants of the original primitives contained in Linda [12,8]. In Linda, agents [1] communicate and coordinate using a shared associative memory or tuple space. The new proposals differ from the original Linda because they are designed to support the use of tuple spaces in an open environment, and over a Wide Area Network (WAN). For such systems fault-tolerance is a real issue [6].

If a tuple space based system is composed of servers storing tuple spaces and agents accessing these tuple spaces, then fault tolerance mechanisms are required for both the server and the application. The server level ensures the tuple spaces are always available, and the application level ensures that, should an agent fail, the tuple spaces are left in an application consistent state. In this paper we are only interested in application level fault tolerance.

Early tuple space based languages, such as Linda suffered from poor application level fault tolerance. Since Anderson et al. [1] first proposed the idea of using transactions [13] to provide fault tolerance in Linda, transactions have become widely adopted, being used in PLinda [14], Paradise [2], JavaSpaces and TSpaces. However, it has emerged that transactions are not always ideal and the fault tolerance they provide is too limited. We propose mobile coordination as an alternative to transactions. Mobile coordination is better because it overcomes the problems of using transactions and is more efficient to implement.

Mobile coordination embodies the idea that code can be migrated transparently from the agent accessing the tuple spaces to the servers providing the infrastructure to support the shared tuples spaces. The code moved can perform arbitrary tuple space accesses and computation and returns a tuple to the agent. These servers provide safe execution environments in which the migrated code is executed. The transferred code is either executed immediately, or deferred until the server deems the agent has failed.

In the next section a brief description of Linda is presented. Section 2 describes fault tolerance at the application layer, with a detailed discussion of transactions as used in tuple space based coordination languages. The well-known problems with using transactions are then highlighted. Section 3 describes the concept of mobile coordination. Section 4 outlines a prototype Java implementation that uses mobile coordination, describing both the API and the run-time system. In Section 5 an example application is described demonstrating how mobile coordination is used to provide fault tolerance. Section 6 describes how mobile coordination could be implemented in a distributed run-time system. Section 7 discusses related work, followed by the conclusion.

1.1. Linda

Mobile coordination is *independent* of the tuple space access primitives being used, and can potentially be combined with any tuple space based coordination language. Throughout this paper the traditional Linda tuple space access primitives are used. In Linda a tuple space is a mathematical bag (a multiset), a tuple is an ordered list of typed values, and a template is an ordered list of typed values or types. Templates are

[1] Throughout the paper the term agent is used in its most general sense, and is seen as being interchangeable with the terms client or process.

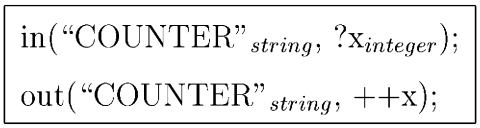


Fig. 1. Incrementing a shared counter.

matched to tuples using an associative matching process. A template matches a tuple if they have the same number of fields and the corresponding fields in the template and tuple are of the same type, and all the values in the template match their corresponding values in the tuple. Linda provides three primitives to enable access to a tuple space:

`out(tuple)` Insert a tuple into a tuple space.
`in(template)` If a matching tuple exists then return the tuple and remove it from the tuple space. If no matching tuple exists then block until a matching tuple is inserted. If several tuples match a template then one is chosen non-deterministically.
`rd(template)` Same as an `in` primitive except the matched tuple is not removed from the tuple space.

Some versions of Linda contain the `inp` and `rdp` primitives, which are non-blocking versions of `in` and `rd`, respectively [7]. These primitives either return a tuple or, a boolean value to indicate that no matching tuple was found. Also, Linda contains an `eval` primitive, which allowed an agent to spawn other concurrently executing agents. In the examples, we assume there are multiple tuple spaces and a global tuple space that all agents can access. Tuple space handles can be passed between agents in tuples.

A wide variety of access primitives have been proposed over the years supporting concepts like tuple streaming, event models and bulk movement of tuples. These tuple space access mechanisms do not impact on mobile coordination and can be used with it, so are not considered further.

2. Fault tolerance at the application layer

Transactions were introduced into Linda to protect applications against agent failure. In many scenarios an agent needs to perform a number of tuple space operations in order to move a tuple space from one application-wide consistent state to another. If these operations are not all performed, then the tuple space is left in a state where the application is no longer able to proceed. For example, consider an application that uses a master–worker style of parallelism. The single master creates tuples in a shared tuple space that describe work to be performed by the workers. A worker repeatedly removes one of these tuples, performs the work, and then inserts a result tuple into the shared tuple space. If a worker fails after removing the work tuple, but before producing the result tuple, the application will not complete as one result is missing. Another example is shown Fig. 1 which is used throughout the paper. The type of the tuple and template fields are shown as subscripts and $?x$ is used to indicate that the value of that field in the matched tuple should be written into the variable x. The

operations increment a shared counter. The tuple containing the counter is removed from the tuple space, incremented, and then reinserted.

If failure occurs after the `in` and before the `out`, the tuple representing the counter will not be reinserted. This means that any other agent using this counter will block forever when next trying to read the counter tuple. Anderson et al. [1] proposed using transactions in Linda to overcome the problems outlined here, and PLinda [14] was the first implementation to support transactions.

2.1. Transactions in tuple space based coordination languages

Most Linda-like languages incorporate transactions by introducing three new primitives; `start`, `commit` and `abort`. Sometimes the `abort` primitive is not supported. The `start` primitive is used to denote the beginning of a transaction, and the `commit` primitive is used to denote the end of a transaction. All operations performed between the `start` and `commit` primitives are executed as a transaction. The `abort` primitive causes the transaction to terminate and none of the side effects of the aborted transaction are observed. Transactions provide the ACID (Atomic, Consistent, Isolated and Durable) property. Atomicity means either all or none of the operations in the transaction are performed. Consistency means that at the end of the transaction the tuple space must be in an application consistent state. Isolation means the tuple space accesses performed are not observable by other agents until the transaction is completed.

Each run-time system uses a different implementation approach to transactions. Assuming only the `in`, `out` and `rd` primitives, in general, any operations performed in a transaction that remove tuples from tuple spaces require the server to retain a copy of the tuples removed. Any tuples that are inserted into a tuple space are not actually inserted but stored separately in the server. This means that the inserted tuples are not visible to other agents. When the transaction completes any inserted tuples are placed into the tuple spaces and become visible to other agents. Any tuples stored because they were removed from a tuple space are discarded. If failure is detected or the transaction is aborted, the tuples inserted in the transaction are discarded and the tuples removed from the tuple spaces are reinserted into the tuple spaces.

If the coordination language contains non-blocking primitives, such as `inp` or `rdp` then the implementation of transactions becomes more complex. It is necessary to ensure that the agents cannot observe partial results of transactions being performed by other agents. For example, assuming only two agents sharing a tuple space, an `rdp` is performed by agent one in parallel with a transaction being performed by agent two. A single tuple matches the `rdp` template and this tuple has been removed by an operation in the transaction. The completion of the `rdp` must be delayed until the transaction commits or aborts. This has to be done to ensure the isolation property of a transaction because if the `rdp` returns false and the transaction does not commit, agent one has observed a state change caused by operations within the failed transaction.

Transactions are not ideal for tuple space based coordination languages because they alter the underlying semantics of the tuple space access primitives performed within them. This is demonstrated by considering the synchronisation between two agents, as shown in Fig. 2.

Agent one	Agent two
out("syncTwo"$_{string}$);	in("syncTwo"$_{string}$);
in("syncOne"$_{string}$);	out("syncOne"$_{string}$);

Fig. 2. Example of altering the semantics in a transaction.

In Fig. 2, agent one produces a tuple that agent two consumes, and then agent two produces a tuple which agent one consumes. When not using transactions the two agents synchronise. If the two operations performed by agent one are placed within a transaction, the two agents will deadlock. The tuple inserted by agent one within a transaction cannot appear in the tuple space until the transaction commits. This cannot happen until agent two produces the tuple, but this cannot happen until agent two consumes the tuple agent one has produced. Hence, there is deadlock.

The semantics of the primitives have been altered because the outcome of the two agents is dependent on whether the primitives are performed within a transaction. This is a well-known undesirable side effect. The more complex the tuple space access primitives (such as those used in many of the new generation of tuple space based coordination languages), the more careful the programmer has to be that deadlock is not being introduced.

This problem is evident when looking at the informal semantics of the new coordination languages and systems. The semantics for the primitives are described when performed within and outside a transaction. For example, the specification describing JavaSpaces [25] provides the semantics of the primitives in its Section 2 and again in its Section 3 where the different semantics under transactions are described. In general, the problem is further compounded by descriptions of whether tuples produced within a transaction can be accessed by subsequent operations within the same transaction. In addition, yet further complexity is introduced by primitives that stream tuples to agents. This means you end up with a coordination language with very subtle behaviour and possible interactions. The reason transactions alter the semantics of the primitives is because transactions require the isolation property. However, coordination languages are about interaction not isolation.

Another problem with transactions is that they do not provide sufficient flexibility. A programmer cannot always implement a particular coordination pattern in a fault tolerance manner using transactions. For example consider implementing presence notification [11] in an application. Each of the agents requires access to a list of the currently executing agents. This list is stored in a tuple space. When an agent begins, it inserts a tuple into the tuple space containing its name. When the agent is finished the tuple is removed by the agent, and the agent terminates. Thus, the tuple space contains a list of the currently active agents. If an agent fails then the tuple containing the name must be removed. The insertion and removal of the tuple containing the name cannot be performed in a single transaction, because the tuple will not be visible outside the transaction. The insertion and removal of the tuple can be performed in

two individual transactions. However, this provides no guarantee that, if the agent fails between the transactions, the name tuple is removed.

Transactions work well in databases, but not in languages designed for coordination. In the following section we will show that mobile coordination can overcome all these problems.

3. Mobile coordination

The aim of mobile coordination is to use mobile code in order to facilitate the management of agent failure. Mobile coordination moves units of coordination from the agent to the server managing the tuple spaces. The server provides a fault tolerant execution environment for executing the code it receives from the agents. As with a transaction, the operations to be performed in a fault tolerance manner are grouped, into what we call a *coordination unit*. Fig. 1 is an example of the operations that might be placed inside a coordination unit. These coordination units are moved to the server to be executed. At the server a coordination unit can be executed either *immediately* or *delayed* until the server considers that the agent has failed. When delaying the execution of the coordination unit, the coordination unit is referred to as an *agent will.*[2] An agent will is used by an agent to tidy up any state stored in a tuple space should the agent fail. For example, in the presence notification example given in the previous section an agent will can be used to remove the name tuple should the agent fail.

A coordination unit can contain any tuple access primitives to any tuple spaces used by the agent, and can contain arbitrary computation. When an agent requests that a coordination unit is executed immediately in a fault tolerant manner, the thread of the agent performing the fault tolerance execution of a coordination unit is blocked until the coordination unit has been moved to the server and executed to completion. A coordination unit returns as its result a tuple containing any state the coordination unit it wishes.

An agent can specify that a coordination unit is an agent will and therefore not executed immediately. The agent thread is then blocked until the agent will is ready to be executed at the server. The agent can have one agent will per tuple space. Once an agent will is created, an agent can cancel it or explicitly request its execution by the server. If the agent requests the explicit execution of the agent will then the thread of the agent is blocked until the coordination unit has been executed to completion.

The only restrictions placed on a coordination unit are that it cannot perform any I/O operations except tuple space accesses, any exceptions must be handled within the coordination unit, and any side effects caused in shared objects will not be visible to the agent. Therefore, the only way for a coordination unit to pass information to the agent is via the tuple that is returned from the coordination unit. This result tuple can be used to pass result (or error information) back to the agent.

[2] The meaning of will in this context is as a legal declaration of a person's wishes regarding the disposal of his or her property or estate after death; especially: a written instrument legally executed by which a person makes disposition of his or her estate to take effect after death [taken from the Merriam–Webster Online Collegiate Dictionary].

The execution environment within the server must be fault tolerant. Once the server has accepted the coordination unit for immediate execution, it is committed to executing it in full. Therefore, mobile coordination provides the atomicity property of transactions; either all the operations are performed or none are performed. Mobile coordination does not provide isolation. The operations performed within a coordination unit interact with other agents and coordination units. Therefore, tuples removed from a tuple space are removed immediately, and tuples inserted into a tuple space appear immediately. This means that the semantics of the primitives performed within the coordination unit are not altered.

The immediate execution of a mobile coordination unit blocks the thread of computation in the agent in order to ensure that, from the application programmer's perspective there is no difference between executing the same group of operations whether or not using mobile coordination. If the agent thread were not blocked, the coordination unit could interact with the agent thread, and this would be more like an `eval`.

The underlying concept behind mobile coordination is to minimise the distance between the data and the code operating on that data. If they are in the same address space, the problem of making the code fault tolerant is considerably reduced. Whereas transactions provide ACID properties, mobile coordination provides only atomicity and consistency. When a mobile coordination unit terminates the tuple space must be in an application wide consistent state. Atomicity is provided because if any of the operations in a coordination unit are executed it is guaranteed that they will all be executed. Durability can be provided if the server manages the tuple spaces in a fault tolerant manner. Mobile coordination provides atomicity through moving the coordination unit to the server. If the agent fails once the coordination unit has been transferred to the server or the server fails, then the atomicity property still must be honoured. Consistency (as with transactions) is achieved by the programmer ensuring the result of the coordination unit leaves the tuple spaces in a consistent state.

4. Implementing mobile coordination

The prototype system is composed of a run-time system and a number of classes that provide an interface to the run-time system for the application programmer. The run-time system is described in Section 4.2 and in the next section the Application Program Interface (API) is described.

4.1. The Java API

The classes that the prototype system provides are: *TupleSpace*, *Template*, *Tuple* and *Formal*, and an interface specification *CoordinationUnit*. The public methods of these classes are shown in Fig. 3.[3]

[3] The *Template* and *Tuple* classes have had some constructor methods removed that allow tuples and templates with up to 20 fields to be instantiated. Also, the exceptions that can be generated are not described.

```
public class TupleSpace implements Serializable {
  TupleSpace(boolean newTS);
  public void out(Tuple tuple);
  public Tuple in(Template template);
  public Tuple rd(Template template);
  public Tuple executeSafe(CoordinationUnit obj);
  public boolean createWill(CoordinationUnit obj);
  public Tuple executeWill();
  public boolean cancelWill(); }

public class Template implements Serializable {
  public Template(Object a);
  public Template(Object a, Object b);
  public Template(Object a, Object b, Object c); }

public class Tuple implements Serializable {
  public Tuple(Object a);
  public Tuple(Object a, Object b);
  public Tuple(Object a, Object b, Object c);
  public Object getField(int number);
  public int getNumberFields();
  public String getType(int field); }

public class Formal implements Serializable {
  Formal(String classname); }

public interface CoordinationUnit {
  public Tuple coordination(); }
```

Fig. 3. API for the Java system.

The class *TupleSpace* acts as an interface to a particular tuple space, and provides the standard Linda tuple space access primitives, `in`, `out` and `rd`. When an instance of *TupleSpace* is created, the boolean value passed to it determines whether the object refers to a new tuple space or references the global tuple space. The *TupleSpace* class also provides the methods for managing the mobile coordination; these are described later. The classes *Template* and *Tuple* are self explanatory, with the constructors being used to insert the elements into the tuple or template object. Fields within the *Tuple* can then be accessed using the methods specified. The class *Formal* is used for specifying the type of formals (fields with only a type and no value) in templates.

Coordination units are created as classes that implement the interface *CoordinationUnit*. The method `coordination` contains the tuple space access operations and associated computation required. In the agent, whenever the functionality of this coordination unit is required, an instance of this class is created, and any required state is passed into the object. If this coordination unit is to be executed in a fault tolerance manner then it is passed as the parameter of the `executeSafe` method of any instance of the *TupleSpace* class. The `executeSafe` method marshals the coordination unit to the server, where the `coordination` method is invoked. The `coordination` method returns a tuple, which is then returned to the application as the result of the invoked `executeSafe` method.

An agent will is an instance of a class that implements the *CoordinationUnit* interface. Agent wills are associated with tuple spaces, with a maximum of one per tuple space per agent. If this coordination unit is an agent will, then it is passed as the parameter of the `createWill` method of the instance of the *TupleSpace* class representing the tuple space with which it is to be associated. The `createWill` method returns true to indicate that the agent will has been set successfully. If the agent has already set an agent will for that tuple space the method returns false. Two other methods are then provided to allow the agent to manage the agent will. The `executeWill` method causes the agent will associated with that tuple space to be executed, and it returns the tuple generated by the `coordination` method. The `cancelWill` causes the agent will associated with that tuple space to be discarded. If there is no agent will the method returns false, otherwise it returns true. A coordination unit passed to the server is considered part of the agent. Therefore, if an agent fails whilst a coordination unit from it is being executed at the server, the server completes the execution of the coordination unit, and then executes any agent wills associated with the failed agent.

Due to the restrictions imposed on a coordination unit, in the `coordination` method of a class that implements *CoordinationUnit* all exceptions must be handled internally, although error codes can be returned as fields in the tuple returned by the `coordination` method. Currently, if an exception is raised that is not handled during the execution of the `coordination` method then an empty tuple is returned by the `executeSafe` method. No I/O operations are permitted during the execution of the `coordination` method. However, all I/O requirements should be achievable by passing tuples through tuple spaces [15]. Also, in the prototype implementation the coordination unit objects must be serialisable and must be deterministic, given the same set of input tuples the coordination unit must produce the same tuples in the same order.

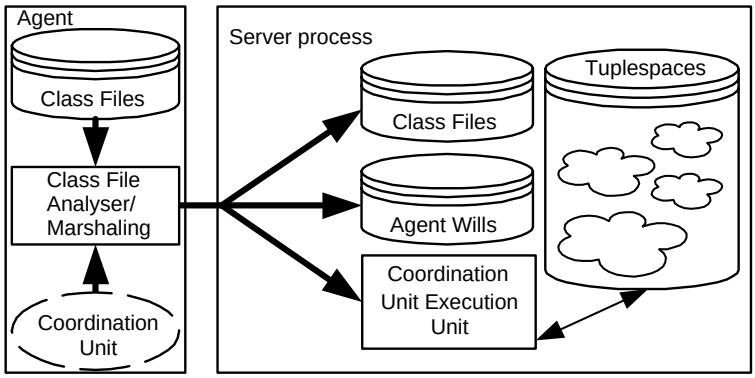


Fig. 4. Architecture of the centralised run-time system.

4.2. Centralised implementation

The run-time implementation includes a server and a library embedded within the agent. The agent library is contained within the class *TupleSpace*. The general architecture of the centralised run-time system is shown in Fig. 4.

Whenever an agent invokes a method representing a standard Linda operation, the marshalling of the request and reply to the server is managed by the agent library.

When a coordination unit needs to be moved to the server the standard Java serialisation is used. Java serialisation encodes the state of a graph of objects, using a specified object as the root of the graph. Java serialisation does not encode fields within the objects marked as transient or static. Whenever the methods `executeSafe` or `createWill`[4] are invoked the object passed as the parameter to these methods is serialised. Serialisation only captures the state of the graph of objects, not the class files (code) required to enable the recreation of the object. It is not sufficient during the de-serialisation of the graph of objects at the server to request the class files not present. This is because other class files could be required because they are instantiated during the execution of the coordination unit. If the agent has failed between passing the coordination unit and the server requiring the class files, the coordination unit cannot be executed to completion. If coordination units are not executed to completion, that means the atomicity property is not provided, and the tuple space can be left in an application inconsistent state. Also, it cannot be assumed that the required class files are already available at the server. Therefore, the agent library after serialisation analyses the class file of the object passed as the coordination unit. It determines other class files that might be required. Any class files that are not in the standard java packages are packaged with the serialised object and dispatched to the server. This means that any classes the application programmer wants can be used in a coordination unit and these can be instantiated at any time.

[4] Provided this agent has not already created an agent will for this tuple space.

The server receives the serialised object and class files. Java allows custom loaders for class files to be created, which allows the server to recreate the serialised object using the class files supplied by the agent. In Java whenever a class is instantiated a *ClassLoader* is used to load and marshal the class. At the server a subclass of *ClassLoader* is provided which manages the passed class files. For efficiency all the class files passed by an agent library to the server are cached at the server. Therefore, if an agent wishes to use the same coordination unit class a number of times there is no need to pass the class files each time, just the serialised state.

If the coordination unit is passed using the `executeSafe` method then once the coordination unit has been successfully recreated on the server, the server creates a separate thread, and the `coordination` method is invoked from within this new thread. Every coordination unit within the server is executed within its own thread to allow the coordination units to execute concurrently (coordination units can interact with other coordination units). The tuple produced as the result of the `coordination` method is returned from the server to the agent. The `executeSafe` method blocks the thread of execution that performs the method call in the agent until the result is available. Once the `coordination` method has been executed, the server discards the coordination unit as the `coordination` method can only be invoked once.

Alternatively, if the coordination unit is passed using the `createWill` the coordination unit moved is stored and associated with an agent. It would be more efficient not to recreate the object when it arrives at the server but to store the object state and class files, and then only recreate the object if required. This is not done because, if there is an error whilst recreating the object, this can be passed back to the agent and returned as a result to the `createWill` method. When *the server* believes that the agent has failed then the object is retrieved, and the `coordination` method invoked. This is done within a separate thread. In the current prototype implementation the standard system's approach of using heartbeats and timeouts is used to detect the agent failure. If a message from an agent is not received within the specified period the server assumes it has failed. The prototype allows an agent to close and open its socket to the server within that time limit. Once the server decided that an agent has failed, any future interactions with the agent result in an exception being raised in the agent.

The `createWill` method blocks the thread of execution in the agent until the agent will is initialised and ready to be executed. An agent will can be created within a coordination unit being executed on the server. If the agent performs a `cancelWill` then the agent library informs the server to disable the agent will. If the agent performs an `executeWill` then, if the agent will exists for the tuple space, the server executes the coordination unit as for an `executeSafe`. The agents thread is blocked until the server has executed the coordination unit and the resulting tuple is returned.

There is no garbage collection of the coordination units. It is possible to have a coordination unit that performs an operation that blocks on a tuple space access where there will never be a tuple inserted that matches the template of the blocked primitive. In addition, for performance reasons an instance of *TupleSpace* is able to detect whether it is at the server or at the agent. It is able to modify its behaviour accordingly so if it is at the server it accesses the data structures storing the tuples directly, and if it is at the agent it marshals the instructions to and from the server.

In order to provide the atomicity property the server has to provide a fault tolerance execution environment. The assumption is that the server has a persistent fault tolerant backing store that it can use, and uses a check pointing and replay method as used in PLinda [14]. Periodically the tuple spaces are check pointed to the backing store. This involves the tagging of all requests to and from an agent with globally unique identifiers. Whenever an update to a tuple space is performed this is logged on a backing store. Should the server fail the agents can reconnect to it when it is resurrected. The agent library handles this transparently and the user's agent perceives no loss of connectivity, even when a coordination unit is being executed on the server.

When a server receives a coordination unit from an agent, either it receives all the class files and the serialised object state in its entirety, and then it successfully recreates the object, or it does not receive them. If the agent fails before all the information has been received, the server does not recreate the object. When the server has received a serialised object and set of class files, it places this information in a log on the backing store. When the coordination unit is executed all tuples that are consumed and produced by the coordination unit are logged on the backing store. When it completes the server attempts to return the result tuple to the agent. If the agent has failed between the `coordination` method starting and ending it does not matter, and the result is discarded. The coordination unit will have been executed in its entirety, and the state of the tuple space will be consistent. This provides the atomicity property of using mobile coordination.

If the server fails, the information stored on the backing store can be used to reinitialise a new instance of the server. The tuple spaces are recreated, and the log of updates to each tuple space is replayed. The coordination unit objects are recreated, using the information stored on the backing store. As the coordination units are deterministic, the `coordination` method is re-invoked, and the tuples it requests are taken from the stored log of its previously input tuples until the log is empty. Any tuples produced are checked against the log of previously produced tuples, and if matching are discarded. As the coordination unit is deterministic, the same tuples should be produced in the same order with the same input sequence of tuples. Once all the logged tuples have been consumed or reproduced, direct interaction with the tuple spaces starts again.

5. An example application using mobile coordination

In order to demonstrate fully how mobile coordination is used a simple example is presented which is based loosely on the idea of an instant messenger that provides a persistent conversation. A conversation is composed of a sequence of lines of text, and stored in a tuple space managed by the tuple space server. It is therefore persistent, so that at anytime anyone can join in, and review the conversation up to the current time. Each instant messenger agent manipulates the conversation directly in the tuple space with each of the lines of the conversation being stored in a single tuple. Each tuple contains a sequence number, the name of the user that generated the line, and the text. In Fig. 1 an example of a counter stored in a tuple space was shown. This

```
1  public class TestMobile {
2    TupleSpace gts = new TupleSpace(false);
3
4    public TestMobile() {
5      gts.executeSafe(new InsertName(gts, "Ant"));
6      InsertLine coordUnit = new InsertLine();
7      coordUnit.insertData(gts,"Ant","Hello I am here!");
8      Tuple t = gts.executeSafe(coordUnit);
9      t = gts.rd(new Template((Integer)t.getField(0),"Ant",
10                      new Formal("java.lang.String")));
11     gts.executeWill();
12   }
13
14   static public void main(String args[])  new TestMobile();
15 }
```

Fig. 5. Example—Main class for the instant messenger agent.

is used in the instant messenger to generate the sequence number of the next tuple. A full example of an instant messenger agent using a tuple space based coordination language is given in [20]. We assume that each instant messenger agent places a tuple in a tuple space with the name of the user in it when the agent starts (referred to as the name tuple), it can then add lines to the conversation, and display the other lines added to the conversation. Finally, when the user is finished the agent removes the name tuple. This is a simple asynchronous computer supported cooperative working tool.

In this example, the name tuple is being used to provide presence notification [11]. So, each instant messenger agent should maintain, on the screen, a list of the other agents that are active. When someone starts an instant messenger agent their name is added to the list, and when they terminate their instant messenger agent their name is removed.

Fig. 5 shows the main part of the instant messenger agent. Line *2* retrieves the handle to the globally shared tuple space. Line *5* creates an instance of *InsertName* (which implements the *CoordinationUnit* interface), passing to the constructor a tuple space handle and the user's name, and then executes the coordination unit in a fault tolerance manner. From the perspective of the programmer the performing of the `executeSafe`

```
1  class InsertName implements CoordinationUnit, Serializable {
2    TupleSpace ts; String name;
3
4    InsertName(TupleSpace ts, String name) {
5      this.ts = ts; this.name = name;}
6
7    public Tuple coordination() {
8      MyWill theWill = new MyWill(ts,name);
9      ts.createWill(theWill);
10     ts.out(new Tuple(name));
11     return new Tuple("okay");
12   }
13 }
```

Fig. 6. Example—A coordination unit to insert the name and set the agent will.

is similar to replacing Line *5* with:

```
(new InsertName(gts, "Ant")).coordination();
```

except this does not provide fault tolerance execution of the operations performed in the method `coordination`. Lines *6* and *7* create an instance of the class *InsertLine*, which is another coordination unit, and passes it the parameters it requires (a tuple space handle and a user name). Line *8* executes the coordination unit in a fault tolerant manner. Line *9* reads the tuple inserted in the coordination unit executed in the previous statement. Line *11* explicitly executes the agent will.

Fig. 6 shows the coordination unit used to insert the name tuple into the tuple space. Lines *8* and *9* create an agent will for the instant messenger associated with the tuple space in which the name tuple is inserted. Line *11* returns a tuple, which in this case is not used by the main class.

Fig. 7 shows the coordination unit that acts as the agent will for the instant messenger agent. Upon instantiation the constructor is passed a tuple space handle and user name. When the agent will is executed the name tuple is removed from specified tuple space.

Fig. 8 shows the coordination unit used to insert a line into the conversation. Lines *4* and *5* allow the object to be initialised with the tuple space, user name, and the text for the tuple to be inserted. The insertion is performed in Lines *7–14*. First the counter tuple is retrieved (Line *9*), then incremented and reinserted (Line *11*). Then the tuple representing the new line in the conversation is inserted (Line *12*). A tuple containing the value of the line inserted (Line *13*) is returned as the result.

```
1 class MyWill implements CoordinationUnit, 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     return ts.in(new Template(name));}
9 }
```

Fig. 7. Example—A coordination unit to act as an agent will.

In Fig. 6 the agent will for the instant messenger agent is set inside another coordination unit. Even thought the agent will is set inside the coordination unit it is still associated with the instant messenger agent. Therefore, the agent will is only executed if the agent is considered to have failed. Setting the agent will inside the coordination unit that inserts the name tuple is convenient, because we know that if the name tuple is inserted the agent will is also created. If the agent is considered to have failed whilst the coordination unit setting the will and inserting the name tuple is executing, the coordination unit will run to completion before the agent will is executed.

A possible alternative is, the instant messenger creates the agent will and then inserts the tuple, without using mobile coordination. However, if the agent failed between the setting of the agent will and the insertion of the name, then the agent will would have potentially consumed another user's name tuple if there was another user with the same name, or blocked on the name tuple. If it blocked, then next time an instant messenger agent was started by a user with the same name, the name tuple would be removed by that agent will. By creating the agent will and inserting the name tuple in the same coordination unit these problems are avoided.

In Fig. 5 the agent will is explicitly executed. This means that the name tuple is removed. In this case, the instant messenger agent could have simply terminated, and then the server would have automatically executed the agent will. If the agent had not wished this to happen then it would need to explicitly cancel the agent will. However, cancelling the agent will and then explicitly removing the name tuple is not equivalent to executing the agent will. The agent will could be cancelled and then the agent fail before removing the name tuple. This would lead to the name tuple being left in the tuple space. Removing the name tuple first, then cancelling the agent will also does not work. If the agent fails after removing the name tuple but before cancelling the agent will, then the agent will would be executed with similar consequences as setting

```
1  class InsertLine implements CoordinationUnit, 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. 8. Example—A coordination unit to insert a conversation line.

the will before inserting the name tuple. Managing this name tuple in a fault tolerance manner cannot be achieved using transactions.

The coordination unit shown in Fig. 8 ensures that the conversation stored in the tuple space is kept in an application consistent state. Because of the atomicity property of mobile coordination, the coordination unit is executed either until termination, or no operations are executed. Therefore, either the counter is updated and a line inserted or a line is not inserted and the counter is not changed. The use of mobile coordination and agent wills means that this agent can be executed in a fault tolerant manner. If the agent fails the shared tuple spaces are left in an application consistent state. The name tuple representing the instant messenger agent is not present, and the conversation tuples are left in a consistent state.

This example has demonstrated the API for a prototype Linda implementation using mobile coordination. Although mobile code is used to provide the fault tolerance, from an application programmer's perspective there is no concept of mobility. However, the program has to be structured to reflect the units that are required to execute in a fault tolerance manner.

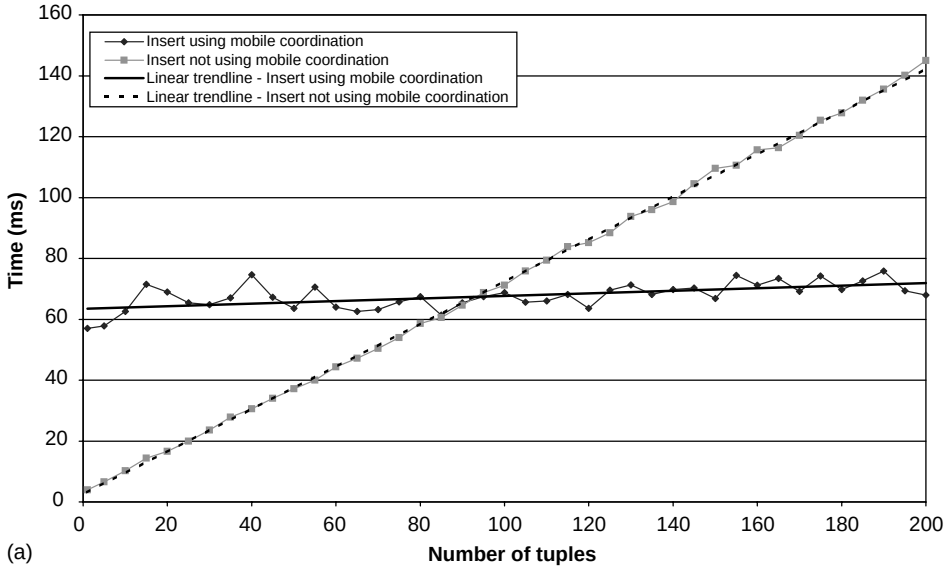
5.1. Performance of the centralised version

The performance of mobile coordination in a centralised run-time system has proven to be good and provide better performance than using transactions. In order to demonstrate this, an example program was created that inserts a number of tuples into a tuple space and then performs a summation operation over those tuples. The insertion stage inserts a specified number of two integer tuples. The value of the first field is incremented with each generated tuple, and the second field is a random integer. The insertion phase also inserts a tuple containing a single integer, which contains the number of tuples inserted. The second stage of the program reads the tuple containing the number of tuples inserted, and then reads, using the `in` primitive, all the tuples and sums the random numbers contained within them. The sum of all the random numbers is then used by the program.

The code for each stage of the program was written as separate coordination units. The coordination units can be executed either in a non-fault tolerant manner without migrating the coordination unit, or in a fault tolerant manner migrating the coordination unit. To execute them in a non-fault tolerant manner the `coordination` method is invoked within the agent, and to execute in a fault tolerant manner the coordination unit is passed as a parameter to an `executeSafe` method. This allows comparison (in the absence of faults) of the performance when using and not using mobile coordination. The implementation does *not* add any overheads to the primitives when the mobile coordination is not used.

Figs. 9–11 show the results when between 1 and 200 tuples are inserted and summed, both over a Local Area Network (LAN) and over a Wide Area Network (WAN). In the case where the mobile coordination is used the size of the class files transferred for the insert coordination unit is 784 bytes with the object state being a further 187 bytes. For the summation coordination unit the class files size is 887 bytes with the object state being a further 176 bytes. Therefore, in both cases the size of the code and state being transferred is about one kilobyte. The LAN experimental results were collected using a 10 MB/s Ethernet, using two Pentium Pro 200 MHz PCs running Linux. The WAN results were gathered using the tuple space server running on an SGI Indy Workstation at York University in the United Kingdom and the agent on a Pentium Pro 200 MHz PC computer running Linux at Cambridge University, in the United Kingdom. For all the results, the load on the machines was low and several batches of results were gathered over periods of low network activity and averaged. On the Linux machines the JVM used came from the Blackdown Java-Linux port of JDK 1.1.6. On the SGI workstations the JVM used came from a port of the Sun JDK 1.1.5.

All the graphs have linear trend lines added to demonstrate the underlying trend that the results show. In all cases, the time taken to perform the coordination unit using mobile coordination increases slowly with the number of tuples being inserted or summed. This is seen by looking at the trend lines in the graphs. The time taken to analyse and transfer the coordination units state and class code, and for the server to reconstruct the coordination unit is independent of the number of tuple space accesses. This is constant at about 175 ms for a WAN (with the server running on a JVM on



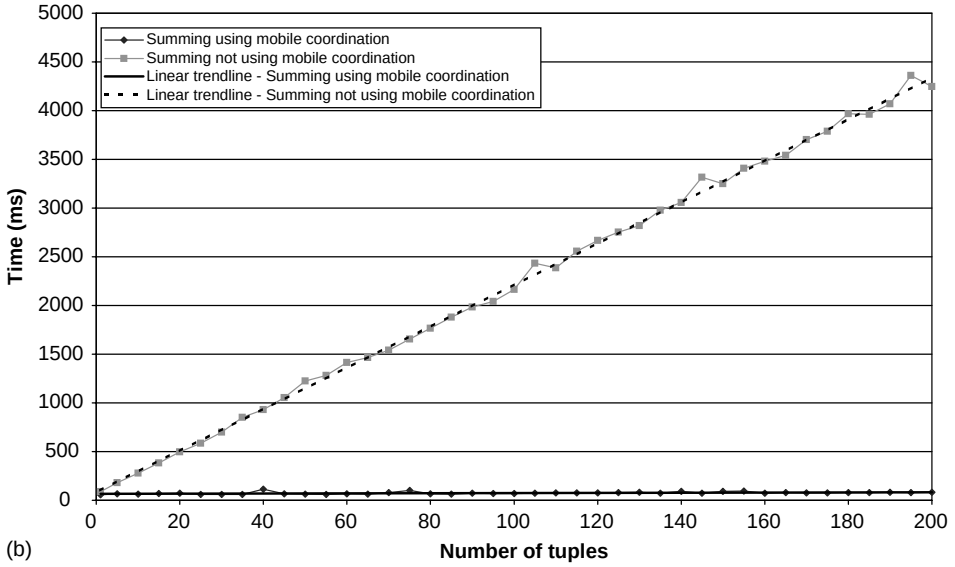
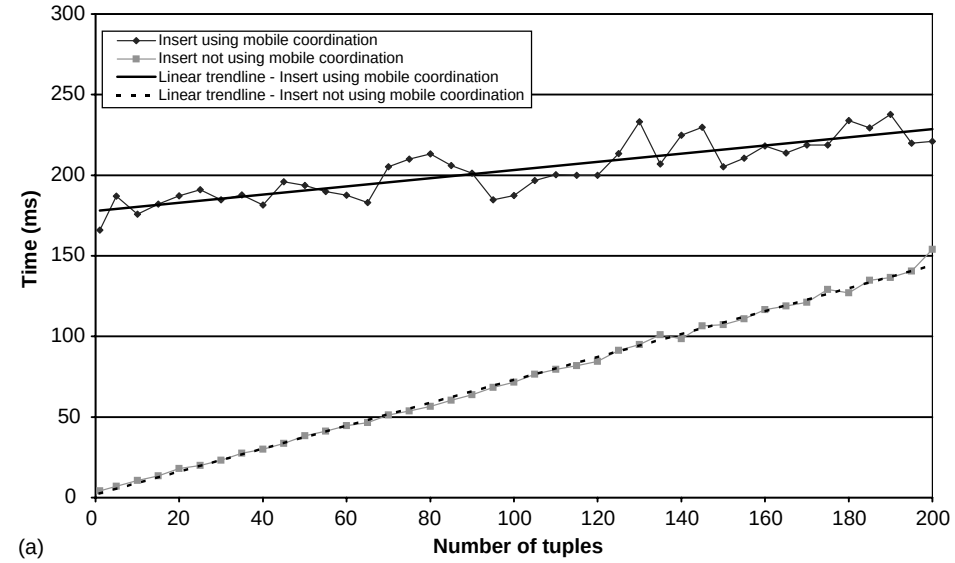


Fig. 9. The effect of using mobile coordination and not using mobile coordination over an LAN: (a) Insert using mobile coordination and not using mobile coordination over an LAN; (b) summation using mobile coordination and not using mobile coordination over an LAN.

an SGI Indy) and 60 ms for an LAN (with the server running in a JVM on a Pentium Pro 200 MHz PC).

Figs. 9(b) and 10(b) show the results for summing the tuples. They show that it is more efficient to use mobile coordination than not to use it. Indeed, the speed up



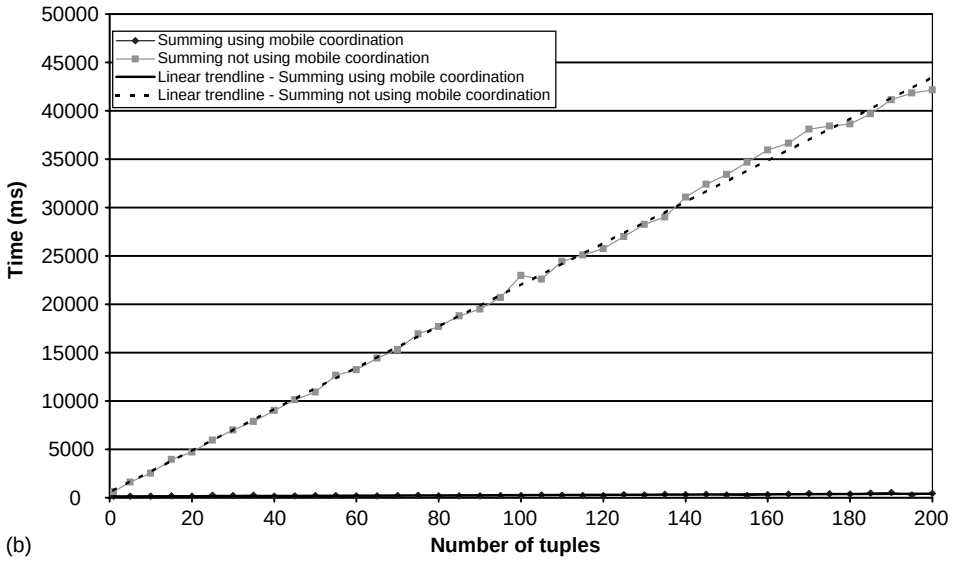


Fig. 10. The effect of using mobile coordination and not using mobile coordination over a WAN; (a) Insert using mobile coordination and not using mobile coordination over a WAN; (b) summation using mobile coordination and not using mobile coordination over a WAN.

of using mobile coordination for summing 1 tuple is 1.6 times faster than not using mobile coordination, for summing 5 tuples it is 2.7 times faster and for summing 200 tuples it is 52.4 times faster. It should be noted that the minimum number of tuples that the summation coordination unit will read is 2 (the counter and one tuple).

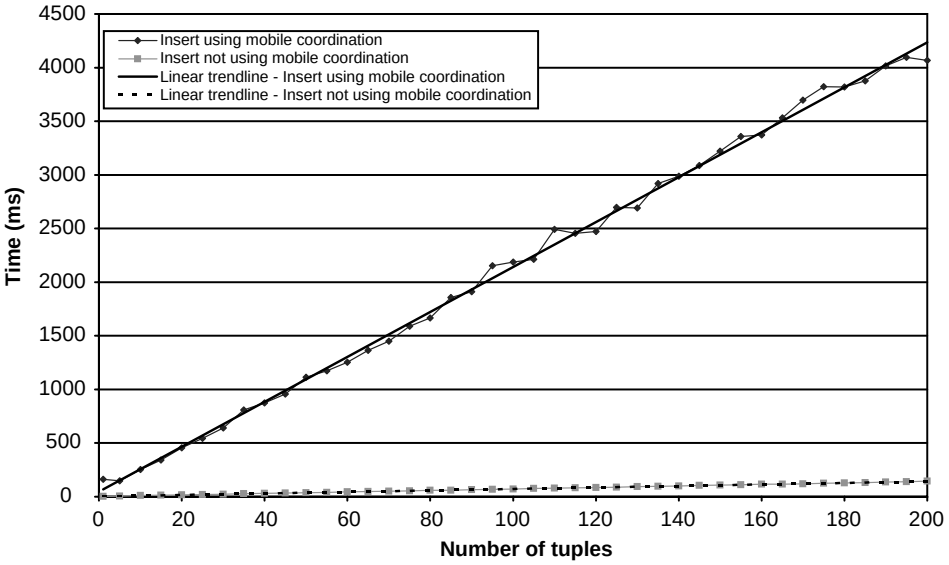


Fig. 11. Insert (with `out` ordering) using mobile coordination and not using mobile coordination over an LAN.

The operations performed within a coordination unit at the server can access the data structure directly; therefore the cost of communication is removed for the tuple space access. At the server the time taken to access the data structure is small compared with network latency, and so 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 running Linux it takes approximately 0.11 ms to read a tuple from the data structure storing the tuple space within the server. Measuring the network latency (using the `ping` tool) between the machine at Cambridge and the machine at York for 22 packets during the night, provides the results: min time is 11.2 ms, average is 13.7 ms and the max is 24.7 ms. Nine tuple space primitives can be performed in less than 1 ms locally, so it is clear why mobile coordination provides better performance, given the network latency context. For the summation coordination unit using mobile coordination with the server running on such a machine (in this case the LAN results) one would expect the increase in time between summing one tuple and 200 tuples to be approximately 22 ms. The experimental results show an increase of 25 ms. Furthermore, the tuple space accesses are sequential, the next tuple space access operation can only begin when the current one completes. This means the network latency is incurred sequentially, rather than being pipelined. As will be seen, in the case of the unordered `out` the communication latency is pipelined, as effectively the next primitive can be performed before the tuple has reached and been processed by the server.

Figs. 9(a), 10(a) and 11 show the performance for a coordination unit that inserts the tuples. Fig. 9(a) would suggest that not using mobile coordination is better if less than 100 tuples are to be inserted running over an LAN. Fig. 10(a) would suggest

the mobile coordination would always provide worse performance for inserting 200 or less tuples over a WAN. Figs. 9(a) and 11 show the same operation (insertion over an LAN). Fig. 11 shows the results when using an `out` primitive that provides *out* ordering. An unordered `out` primitive guarantees no ordering between the tuples inserted from the same agent. To make an `out` ordered you need an acknowledgement message. For more information see Busi et al. [4], where they show that, for tuple space languages supporting non-blocking tuple space access primitives, an ordered `out` is required. Because the results in Figs. 9(a) and 10(a) use an unordered `out` the times represent the time taken to package the tuples up and insert them into a socket buffer. Given the size of tuples used, the buffers within the TCP/IP stack will be sufficient to buffer all the tuples. This means the time taken reflects the time taken for the agent to complete the insertion into the socket, *not* for the tuples to appear in the tuple space. This also means the results, when not using mobile coordination, are similar regardless of whether an LAN or a WAN is being used.

The results not using mobile coordination shown in Fig. 11 closely resemble the results in Fig. 10(b) for summing over an LAN. This is unsurprising, as to make an `out` ordered, an acknowledgement is required. The results in both these figures demonstrate the impact that network latency has when the agent has to interact with a tuple space server in a synchronous manner. The implementation approach of `out` ordering in this implementation is simplistic compared to the kernel described in [19].

We conclude that if more than one tuple space access is to be performed in a coordination unit, and these accesses require either a tuple to be returned, or an acknowledgement (e.g. using an ordered `out`), mobile coordination provides better performance than not using mobile coordination. If a single tuple space access is required, there is no benefit in using mobile coordination. Normally there will be at least two operations performed in a single coordination unit, because there is no benefit of performing a single tuple space operation in a fault tolerance manner.

The example program used coordination units of a size, in terms of class files size and state, that was felt to be representative of coordination units used in real-world applications. The performance of using mobile coordination will be influenced by the network bandwidth and latency characteristics, the size of the code and state being transferred, and the number of times the same coordination unit class files are required.

We have also attempted to determine whether the computation load at the server is increased by using mobile coordination. When a coordination unit is moved to the server then the costs of marshalling the individual tuple space accesses are removed, but there is the cost of executing the coordination unit within the server. We have determined that in the Java prototype implementation the predominant cost is in serialising and de-serialising the objects. The following results were obtained on a 400 MHz Pentium II computer running Windows 2000 and using Microsoft JVM 5.00.3240. The average time cost of serialising the tuples used in this example is 0.55 ms, the average time cost of de-serialising the templates is 0.63 ms, de-serialising the insertion coordination unit is 0.53 ms, and de-serialising the summation coordination unit is 0.54 ms. The time taken to execute the coordination units is dependent on the number of operations they perform. For a tuple space access performed by the agent, which requires a template and a tuple, the time required to marshal them is more than 1 ms. The overhead

of executing the coordination unit is approximately 0.11 ms, plus the time required for the user computation and tuple space operations. Whether a tuple space access is performed from an agent or from within a mobile coordination unit the same cost of manipulating the tuple storage data structure will be the same for both. Therefore, it appears that the mobile coordination imposes less of a computational load on the server. However, it must be remembered that the current implementation is a prototype and it may be possible to generate more efficient marshalling code than being currently used. This remains one of the questions for future research.

5.2. Comparing the performance of mobile coordination and transactions

In order to compare mobile coordination and transactions, we consider how transactions are implemented. Two primitives are used, one to indicate the beginning of a transaction, and one to indicate the end of a transaction, as outlined in Section 2.1. The cost of performing each tuple space access primitive is the same whether it is performed within or outside a transaction. Each primitive involves dispatching a request to the server, and if appropriate, waiting for a reply from the server. In the best case, the begin and end transaction messages can be piggybacked on the first and last tuple space access of the transaction. The piggybacking of the end transaction message on the last tuple space access message requires compile-time analysis. Even in the best case, a set of operations performed within a transaction will provide the same performance as the same operations performed outside a transaction. We conclude, therefore, that the performance of mobile coordination is indeed better than transactions.

6. Distributed implementation

We are predominantly interested in the centralised version. However, for completeness, we have considered how mobile coordination could be implemented in a distributed server system, such as the one described in [24]. In this run-time system each individual server is responsible for one or more *entire* tuple spaces.[5] There is an assumption that disconnection between the servers is only transient and it is assumed that if a server fails it will be resurrected.

Mobile coordination could be implemented in several ways in such a distributed run-time. One approach, given that the servers provide a fault tolerant execution environment for coordination units as in the centralised implementation, is to provide a different robust mechanism for inter-server tuple space requests. If a coordination unit executing on server one requires a tuple from a tuple space stored on server two, then server one would send a request for the tuple to server two, and log the request on a local permanent store. Server two would log the request on a permanent store, then service the request, and then send the result to the first server, logging the result has been dispatched. If server one does not receive the result within a timeout period,

[5] There is no fundamental reason why each server could itself not be distributed over a cluster of workstations.

server one would reissue the request to server two. Server two would check the logs whether it had received the request and if not would service the request. Otherwise, if it had sent the result then it would find it in its log and reissue the reply.

This approach works because the system guarantees to resurrect the coordination unit in case of failure. However, it is a heavy weight approach adding an overhead to all the tuple space accesses. Another approach could be used that builds on the ideas of mobile coordination, and extends it to work between the servers.

As with the centralised implementation the coordination unit is transferred to a server. In the distributed implementation the coordination unit is moved to the server which is managing the tuple space that is represented by the object on which `executeSafe` is invoked. Each server provides a fault tolerant execution environment for coordination units. The coordination unit execution begins, but when a tuple space, which is not stored on the server, is accessed then the coordination unit is migrated to the server storing the tuple space. The object state and code should be migrated in a robust manner, as outlined in the previous distributed architecture description. Ideally, strong migration would be used, so the program counter, execution state, object state and class files would be transferred. In a Java implementation this is non-trivial, requiring either a specialised JVM or a program transformation (similar to the technique used in KLAVA [3]). If strong migration was not used the same effect could be achieved by transferring the *original* object state, class files, and the log of consumed and produced tuples. Once at the new server the coordination unit is recreated and the `coordination` method invoked. The tuples in the consumed and produced tuple log can be provided to the coordination unit until no more are available. The next tuple produced or consumed should be an operation on a tuple space stored on this server. Thereby the coordination unit is replayed until the point where the tuple space on the local server is accessed. This would be less efficient than using strong migration.

The two implementation strategies outlined here are different and both provide possible solutions to how mobile coordination could be implemented in a distributed run-time system. Which performs better is a complex issue and depends on the number of tuple spaces accessed, the distribution of tuple spaces over the servers, the number of tuples accessed per tuple space, in what order the tuple spaces are accessed, and how much computation the system coordination unit performs. Detailed analysis of a distributed implementation remains an open question.

7. Related work

Many coordination languages and systems incorporate transactions, including PLinda [14], Paradise [2], JavaSpaces [25] and TSpaces [26]. The approach taken in all these is similar to the approach described by Anderson et al. [1]. However, there are two Linda based systems of particular interest, both of which use reactive tuple spaces: TuCSoN [17] and MARS [5]. Reactive tuple spaces were first introduced in TuCSoN and they allowed sequences of non-blocking tuple space operations to be inserted into or appended on to a tuple space. These sequences of operations are created using a logic language and are referred to as reactions. These reactions are first class and

persistent, so can be inserted and removed from a tuple space. These reactions are fired whenever a certain tuple operation is performed, where these operations are a tuple insertion, particular primitives being performed, etc. The operations that can trigger a reaction are all tuple observation related, such as tuple insertion rather than agent failure. MARS extends the concept further incorporating mobility of agents. Agents can only access the tuple space at the location where they are and therefore migrate between one tuple space and the next in order to access them. The tuple spaces in MARS are reactive as well, and again the reactions are attached to tuple space access events. The reactions only refer to one tuple space, and in MARS are Java based.

There are many differences between mobile coordination and MARS and TuCSoN. Initially considering agent wills, it is guaranteed that an agent will is only executed once, when an agent dies, whereas in MARS a reaction is persistent. MARS allows only administrator agents to add and remove reactions, whereas any agent can have an agent will and only they can manipulate their own agent will. The event that triggers a reaction is tuple space accessing, whereas an agent will is only triggered by a perceived failure of the agent by the server. A reaction in MARS could not be configured to fire when an agent dies, and a reaction could not remove itself, which would be required for the systems to provide an agent will. In mobile coordination, different agents do not use the same agent will, each agent creates its own agent will.

Mobile coordination allows individual agents to perform personal coordination units in a fault tolerant manner. Reactive tuple spaces add general new properties to tuple spaces that are invoked by any agent performing a particular operation. Mobile coordination allows access to arbitrary tuple spaces, whereas reactions can access only a single tuple space. Indeed, it should be feasible to implement mobile coordination on top of (as a service) many current Linda implementations (although this has not been attempted) without altering the underlying implementation at all. Reactive tuple spaces require a lower-level intervention.

Therefore, there are considerable differences between mobile coordination and reactive tuple spaces. They are designed to solve different problems using different techniques.

8. Conclusion

In this paper, we have demonstrated the concept of mobile coordination and shown how it can be used to provide fault tolerance in tuple space based coordination languages. It has been shown whilst mobile coordination requires the application programmer to construct programs in a certain way, the actual mechanics of how the fault tolerance is provided (mobile code) is transparent to the application programmer. The ideas of mobile coordination are applicable to any tuple space based language. An example implementation has been given that uses Java and the standard Linda primitives. The same technique has been incorporated into the coordination language WCL. There are no technical reasons why other tuple space based coordination languages and systems could not incorporate the technique as well.

Mobile coordination does not alter the semantics of the primitives performed within the coordination units. The use of agent wills provides a level of fault tolerance that cannot be achieved using transactions. Mobile coordination provides better performance than using transactions and, in general, better performance than not executing a set of operations in a fault tolerant manner.

The work described here presents a novel and successful approach for providing fault tolerance in tuple space based coordination languages. The full evaluation of whether or not the computation load at the tuple space server is reduced; and the issue of developing and evaluating a distributed run-time system that supports mobile coordination; are both left as future work.

Acknowledgements

I would like to thank Dr. Alan Wood for allowing me access to his computing facilities at York University, Dr. Stuart Wray for his many long conversations on the subject of mobile coordination, the reviewers of the conference version of this paper, the reviewers of this paper for their helpful and enlightened comments, and my colleagues at Microsoft Research who read or commented on this work, especially Kurt Geihs.

References

[1] B. Anderson, D. Shasha, Persistent Linda: Linda + transactions + query processing, in: J.P. Banatre, D. Le Metayer (Eds.), Research Directions in High-Level Parallel Programming Languages, Lecture Notes in Computer Science, Springer, Berlin, Vol. 574, 1991.

[2] S.C. Associates, Paradise: User's Guide and Reference Manual, Scientific Computing Associates, New Haven, Connecticut, USA, 1996.

[3] L. Bettini, R. De Nicola, G. Ferrari, R. Pugliese, Interactive Mobile Agents in X-KLAIM, in: P. Ciancarini, R. Tolksdorf (Eds.), Proc. 7th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 1998, IEEE Computer Society Press, Stanford, pp. 110–115.

[4] N. Busi, R. Gorrieri, G. Zavattaro, Three semantics of the output operation for generative communication, in: P. Ciancarini, C. Hankin (Eds.), Coordination Languages and Models, Proc. Coordination '97, Lecture Notes in Computer Science, Vol. 1282, Springer, Berlin, 1997, pp. 205–219.

[5] G. Cabri, L. Leonardi, F. Zambonelli, MARS: a programmable coordination architecture for mobile agents, IEEE Internat. Comput. 4 (4) (2000) 26–35.

[6] L. Cardelli, Wide area computation, in: J. Wiedermann, P. van E. Boas, M. Nielsen (Eds.), Automata, Languages and Programming, Proc. 26th Internat. Colloq. ICALP '99, Lecture Notes in Computer Science, Vol. 1644, Springer, Berlin, 1999, pp. 10–24.

[7] N. Carriero, Implementation of tuple space machines, Ph.D. Thesis, Yale University, 1987, YALEU/DCS/RR-567.

[8] N. Carriero, D. Gelernter, Linda in context, Comm. ACM 32 (4) (1989) 444–458.

[9] P. Ciancarini, D. Rossi, Coordinating Java agents over the WWW, World Wide Web J. 1 (2) (1998) 87–99.

[10] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, A. Knoche, Coordinating multigent applications on the WWW: a reference architecture, IEEE Trans. Software Eng. 24 (5) (1998) 362–366.

[11] M. Day, Presence and instant messaging via HTTP/1.0: a coordination perspective, in: P. Ciancarini, A. Wolf (Eds.), Coordination Languages and Models: Coordination99, Lecture Notes in Computer Science, Vol. 1594, Springer, Berlin, 1999, pp. 417–418.

[12] D. Gelernter, Generative communication in Linda, ACM Trans. Program. Languages Systems 7 (1) (1985) 80–112.
[13] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, Los Altos, CA, 1993.
[14] K. Jeong, D. Shasha, Persistent Linda 2: a transaction/checkpointing approach to fault-tolerant Linda, in: Proc. 13th Symp. on Reliable Distributed Systems, Dana Point, CA, Oct. 25–27, 1994.
[15] R. Menezes, A. Wood, Incorporating input/output in Linda, in: Proc. 31st Ann. Hawaii Internat. Conf. on System Sciences, Vol. VII, IEEE CS Press, Big Island, Hawaii, USA, 1998, pp. 216–225.
[16] R.D. Nicola, G. Ferrari, R. Pugliese, KLAIM: a kernel language for agents interaction and mobility, IEEE Trans. Software Eng. 24 (5) (1998) 315–330.
[17] A. Omicini, F. Zambonelli, Coordination for internet application development, Autonomous Agents Multi-agent Systems 2 (3) (1999) 251–269.
[18] G. Picco, A. Murphy, G.-C. Roman, Lime: Linda meets mobility, in: D. Garlan, J. Kramer (Eds.), Proc. 21st Internat. Conf. on Software Engineering (ICSE '99), ACM Press, Los Angeles, USA, 1999, pp. 368–377.
[19] A. Rowstron, Bult primitives in Linda run-time systems, Ph.D. Thesis, Department of Computer Science, University of York, 1997.
[20] A. Rowstron, Using asynchronous tuple space access primitives (BONITA primitives) for process co-ordination, in: P. Ciancarini, C. Hankin (Eds.), Coordination Languages and Models, Proc. Coordination '97, Lecture Notes in Computer Science, Vol. 1282, Springer, Berlin, 1997, pp. 426–429.
[21] A. Rowstron, WCL: a web co-ordination language World Wide Web J. 1 (3) (1998) 167–179.
[22] A. Rowstron, Mobile co-ordination: providing fault-tolerance in tuple space based co-ordination languages, in: P. Ciancarini, A. Wolf (Eds.), Coordination Languages and Models: Coordination 99, Lecture Notes in Computer Science, Vol. 1594, Springer, Berlin, 1999, pp. 196–210.
[23] A. Rowstron, A. Wood, BONITA: a set of tuple space primitives for distributed coordination, in: HICSS-30, Vol. 1, IEEE CS Press, Silver Spring, MD, 1997, pp. 379–388.
[24] A. Rowstron, S. Wray, A run-time system for WCL, in: H.E. Bal, B. Belhouche, L. Cardelli (Eds.), IEEE Workshop on Internet Programming Languages, Lecture Notes in Computer Science, Vol. 1686, Springer, Berlin, Chicago, USA, 1998, pp. 78–96.
[25] Sun Microsystems, Javaspaces specification version 1.0, Final Specification, January 1999.
[26] P. Wyckoff, S. McLaughry, T. Lehman, D. Ford, TSpace, IBM Systems J. 37 (3) (1998) 454–474.