Using Agent Wills to Provide Fault-tolerance in Distributed Shared Memory Systems

Antony Rowstron
Microsoft Research Ltd.
St. George House, 1 Guildhall Street
Cambridge, CB2 3NH, United Kingdom.
antr@microsoft.com

Abstract

In this paper we describe how we use mobile objects to provide distributed programs coordinating through a persistent distributed shared memory (DSM) with tolerance to sudden agent failure, and use the increasingly popular Linda-like tuple space languages as an example for implementation of the concept.

In programs coordinating and communicating through a DSM a data structure is shared between multiple agents, and the agents update the shared structure directly. However, if an agent should suddenly fail it is often hard for the agents to make the data structures consistent with the new application state. For example consider if a data structure contains a list of active agents. In such a case, transactions can be used when adding and removing agent names from the list ensuring that that the data structure is consistent and does not become corrupted should an agent fail. However, if failure of the agent occurs after the name has been added, how does the application ensure the list is correct?

We argue that using mobile objects we can provide wills for the agents to effectively enable them to ensure the shared data structure is application consistent, even once they have failed. We show how we have integrated the use of agent wills into a Linda system and show that we have not increased the complexity of program writing. The integration is simple and general, does not alter the underlying semantics of the operations performed in the will and the use of mobility is transparent to the programmer.

1. Introduction

Using Distributed Shared Memory (DSM) as a communication and coordination mechanism for applications that are executing over the Internet is become increasingly popular. The concept of using DSM in such a framework has been explored in what could be called traditional DSM, in systems such as PerDiS[7], which quite literally provide a mechanism for moving and managing pages of shared memory between multiple machines. As well as the traditional DSM architectures there has been considerable interest in the last few years in the use of tuple space based coordination languages, with many new languages being proposed including WCL [13], PageSpace [5], TuSCon [11], MARS [2], Jada [4], TSpaces [19], KLAIM [10], Lime [12] and JavaSpaces [18]. These are all based around the original ideas contained in Linda [8, 3], where the processes which want to communicate and coordinate use a shared associative memory, or tuple space.

The domain we are interested in is systems that support asynchronous Computer Supported Collaborative Working (CSCW) applications, or in other words, systems where the DSM is persistent, and contains state that can be used after the agent that created the data has died. For the purposes of this paper we will assume that an agent is an entity (application, process, program or "agent") that is using a shared memory to communicate with other agents. The problem we aim to address is the one of sudden agent failure resulting in data structures in the memory being consistent (no dangling pointers etc.) but containing information that is incorrect, and in Section 2 we will demonstrate this problem in detail.

Traditional methods for providing fault tolerance, such as transactions, have been widely adopted in DSM systems, for example in PerDiS, TSpaces, JavaSpaces. However, with the current drive to use these infrastructures for highly distributed CSCW applications new demands are made by application programmers. These new demands include the need for more styles of fault tolerance. In the paper we describe our approach to providing a different layer of fault tolerance, which is used in conjunction with, not as a replacement for, traditional approaches.

The basic premise of the work is that if you can minimise

the distance between a piece of code and the data it is acting upon then you can provide better fault tolerance. There is nothing new per se in this, and in client-server based applications manage this by having the shared state managed by a single server (or synchronised set of servers) which copes with failure of agents and maintains the shared state, which can only be access through the server interface. In DSM systems the agents share and manipulate the data directly, and there is often no server agent that has overall responsibility for the system. Indeed, the server could well be considered as the run-time system that supports the DSM, and it is a general infrastructure that supports a general-purpose access interface. In a client-server world the server can know what to do in the case of client failure, however, in the peer-to-peer world of DSM, the general system managing the shared memory has no notion of what should be done upon agent failure.

In this paper we introduce the notion of an "agent will" that simply is a piece of code that run-time managing the DSM can execute if it detects a failure with an agent.

In the next section presence notification is described which demonstrates the need for further fault-tolerance mechanisms, and in Section 3 the concept of agent wills is described in detail. Section 4 describes Linda, a comparison of this work with other work in the Linda field, and shows how agent wills are incorporated into a working Linda based system, showing the Java API used and briefly discussing the implementation. Finally, Section 6 describes future work and then we present our conclusions.

2. Presence notification

To demonstrate the general shortcoming with current DSM systems we consider the problem of "presence notification" [6] in a CSCW application. Presence notification is characterised as the need for an agent to know that another agent currently exists. For example, given a shared whiteboard application and the assumption that at some point in time there is more than one user sharing the same whiteboard it is quite conceivable that the application should maintain, on each user's screen, a list of people who are currently sharing that whiteboard. When someone starts the application their name is added to the list, when they close the application their name is removed. This list would be an example of presence notification.

If writing such an application using a client-server paradigm a server is written that manages the shared state, and the server can detect when it thinks an agent has failed and then remove the name from the list and propagate the change. The server knows what to do when the agent has been considered to fail, because the application programmer who created the server created the code to perform the necessary operations on the data structure. However, when using a DSM system the shared state would be stored by the underlying infrastructure providing the DSM, and the application writers create an agent that manipulates and monitors this shared state accordingly.

In a DSM system, although it is possible for the underlying infrastructure to detect that an agent has stopped responding, it is not easy for the agents to decide that another agent has failed. How this can be detected varies on the particular attributes of the DSM implementation. For example, in most tuple space based DSMs this is difficult/impossible to detect. This is compounded because multiple applications may share the same data structure, and then one application may not understand what to do to the data structure when another applications agent fails. This is especially true as there may be only one agent executing when the failure occurs, and therefore there are no other agents active. This is possible because many DSM systems provide persistence of data structures (all tuple space languages support persistence of tuple spaces) therefore; other agents will use the shared data structures in the future.

The problem is that the infrastructure supporting the DSM has no concept of what an application would like done when an agent fails (becomes disconnected, dies or whatever). In the shared whiteboard application example there is no real problem with the whiteboard. Whenever an update to the whiteboard is performed the program uses some fault tolerance mechanism (such as transactions) to update the whiteboard. If failure occurs in the middle of the transaction then the transaction aborts and the whiteboard is left in a consistent state. However, the presence notification data structure needs to be updated to reflect the agent has died. In other words, the shared data structure that contains the name of the user needs to have the name removed. But if the failure was unexpected how can the agent who died update the shared state[1]?

In order to overcome this problem we have examined the use of agent wills. The concept of agent wills is now described.

3. Agent wills

In order to overcome this we propose using mobile objects and to allow an agent to pass to the run-time supporting the DSM a piece of code that should be executed *should the system* think that the agent has failed. In this context, a mobile object is considered to be the code and internal state of the object, but not the execution point (in other words you can not pass a thread to the run-time system). The run-time stores the passed object, and this is called the "agent will". If the agent terminates normally then the will is discarded,

[1]The failure could have even occurred whilst the agent was trying to remove the name from the data structure.

but if the run-time decides that the agent has failed it executes a method within the object. It should be noted that the agent must decide when the agent has failed. The mechanism used to decide when failure has occurred will most likely be network dependent, and communication method specific. The object that is passed to the run-time system, implements an interface. The Linda embedding that is described in the next section is shown in Figure 1. The run-time system calls the method `will`.

If the will is executed it is able to update any shared data structures as though it was a normal agent. It can therefore, remove a name from a shared data structure, or even insert some state to indicate that the agent has disappeared rather than terminated normally. The application programmer creating the agent that dies is free to choose what operations are to be performed on the shared data structures.

The aim is that the application programmer should not perceive that the mobile code is being used, and the semantics of the operations performed in the will should be the same as though the operations are performed within the agent. Although the application programmer will hopefully not perceive this they are providing "small servers" that slot into the main server. The small server only provides one service which is called automatically on failure.

```
1 interface AgentWill{
2   public void will(); }
```

Figure 1. The Java interface for a will object.

Having provided an overview of what an agent will is, let us now consider the addition of agent wills to a Linda system. Before describing the addition, we will provide a brief overview of Linda, and some background work on Linda systems.

4. Case study: Linda

Linda [8, 3] was the first tuple space based co-ordination language. In the nomenclature of Linda a tuple space is a mathematical bag (a multiset allowing for repetition of members), a tuple is an ordered list of typed elements (actuals), and a template is an order list of formals and actuals. Tuples are inserted and removed from a tuple space using an associative matching process. A template matches a tuple if they have the same number of fields, and all the actuals in the template match the actuals in the tuple and the formals in the template match the type of the corresponding actuals in the tuple. Basic Linda provides three primitives to enable access to a tuple space:

out(tuple) Insert a tuple into a tuple space.

in(template) If a tuple exists that matches the template then remove the tuple and return it to the entity performing the `in`. If no matching tuple is available then the primitive blocks until a matching tuple is available.

rd(template) If a tuple exists that matches the template then return a copy of the tuple to the entity that performed the `rd`. If there is no matching tuple then the primitive blocks until a matching tuple is available.

It should be noted that if multiple tuples in a tuple space match a template then the returned tuple is chosen non-deterministically. Although the original implementation only supported a single global tuple space, current tuple space based coordination languages support multiple tuple spaces in one form or another. In these implementations normally tuple space handles can be passed between agents in tuples. Also, the new generation, such as WCL, support a wide variety of access primitives, and support concepts like tuple streaming, event models, bulk movement of tuples and so forth. However, the exact primitives that a language supports and the styles of interaction with a tuple space are of no real consequence to the whether they can use the agent will concept. Any access primitives or coordination patterns that can be used by an agent can be used in the agent will.

Most of the new generation of coordination languages use a variety of mechanisms to provide fault-tolerance not found in the original Linda. Most languages that address fault-tolerance (such as TSpaces and JavaSpaces) use a transaction-based method of providing fault tolerance. The transaction model adopted provides fault tolerance for updating a data structure, where at the end of the transaction the data structure within the tuple space will be consistent. WCL uses the concept of mobility to provide the fault-tolerant properties which transactions address in traditional tuple space languages [14], and this work described in this paper is an extension of this work.

There are several other systems which look at the use of mobility within Linda systems, such as KLAIM [10], Lime [12], TuCSoN [11] and MARS [2]. Most of these systems attempt to expose the mobility of agents and to make it a feature that the application programmer explicitly uses. One of the main concepts of agent wills is that there is no concept of location. The application programmer has no notion of where or how an agent will is stored or executed.

There are two Linda based systems of particular interest at this point, both of which use reactive tuple spaces: TuCSoN and MARS. Reactive tuple spaces were first introduced in TuCSoN and they allowed sequences of non-blocking tuple space operations to be inserted into a tuple space. These sequences of operations are created using a logic language. The reactions are first class and persistent, so can be inserted and removed. These reactions were fired whenever a certain tuple operation in performed, such as tuple in-

sertion, tuple removal or the use of a particular primitive. The operations which fire the reactions are all high level, not as low as agents failing, joining and so on. MARS extends the concept further incorporating mobility of agents. Agents can only access the tuple spacs which is physically present at the same location as the agent and therefore the agents migrate between one tuple space and the next in order to access them. Location is therefore explicit in MARS. The tuple spaces in MARS are reactive as well, and again the reactions are attached to tuple space access events as in TuCSoN. The reactions only refer to one tuple space, and in MARS are Java based.

There are many similarities but also many differences between the work described here and MARS and TuCSoN. It is guaranteed that an agent will is only ever executed once which is when an agent is thought to have died, whereas in MARS a reaction is persistent. MARS allows only administrator agents to add and remove (any) reactions, where as any agent can have a will but only they can manipulate their own will. The event that triggers a reaction is tuple space accessing whereas no tuple space access affects a will. A reaction in MARS could not be configured to fire when an agent dies and a reaction could not remove itself which is what would be required for the systems to provide an agent will. Different agents will probably not use the same agent will. Even, if "agent death" was added as an event it is unclear that each agent could configure its own agent will, because all reactions that reacted to "agent deaths" would fire. Also, an agent will can refer to multiple tuple spaces not just a single tuple space. There is no explicit notion of location with the will and so the programmer has no notion of where the will is being stored or managed. In the current Java implementation a will is associated with a tuple space, however, it can access any tuple space, and in future a new mechanism for managing agent wills may be added which is completely independent of the tuple spaces. Also, wills are presented as a general-purpose solution to the problem in DSM systems as a whole; Linda is used to demonstrate the concept. Indeed, it should be feasible to implement this on top of (as a service) many current Linda implementations (although this has not been attempted) without altering the underlying implementation at all.

5. Incorporating agent wills in Linda

5.1. Java API

In order to demonstrate the concept of agent wills we have implemented the system using our Java embedding of Linda. In order to demonstrate the concept of agent wills the Java language is a natural choice as it supports object serialisation and dynamic loading of class files which can be easily extended to support loading of the object code and current state across a network. The important point here is the use of a Virtual Machine, which allows heterogeneous machines to exchange executable code, and the ability to dynamically load code.

One of the aims of the system was to make the mechanisms providing the fault tolerance as transparent as possible. The application programmer should not need to worry about how the fault tolerance is provided. Indeed, it should be noted that in Linda the use of transactions alters the meaning of some of the primitives performed within the transaction. Mobile objects may also provide an alternative to the use of transactions in Linda based systems [14].

In order to show the API and how it is incorporated into the Java embedding of Linda consider the example code shown in Figures 2 and 3. They show an application that creates and uses an agent will and the object that acts as the agent will respectively. It should be noted there is no explicit mention of location anywhere in the two fragments of code.

In Figure 2 the class *TupleSpace* provides the interface to a single tuple space managed by the run-time, providing the methods `out`,`in`, and `rd` as would be expected, and also two extra methods `createWill` and `cancelWill`. A new object has to be created for each tuple space that is to be accessed, and the syntax of the constructor for the *TupleSpace* class has been simplified slightly for clarity. In the current implementation an agent can add a will to any tuple space, therefore it is potentially possible for a single agent to have multiple wills. This does not seem to be a problem and in many ways provides a more natural way of incorporating the concept into a Linda system. There is no reason why there could not be a single will attached to no particular tuple space.

A will is attached to a tuple space by calling the `createWill` method of the object that is acting as the gateway for accessing the tuple space. Once an object has been made a will any alterations to its internal state will not be reflected in the will that the system stores. If the internal state is updated then the object will have to be resubmitted as the will for this to be reflected in the will the system is storing. The `createWill` method requires an object that implements the AgentWill interface. The `AgentWill` interface is shown in Figure 1 and when the server wishes to execute a will it calls the method *will*. If there is already a will in place, currently, the will is cancelled and the new will is made the will. It may make sense though to raise an exception at this point and allow an agent to decide what they wish to do.

In Figure 2 line 9, the `cancelWill` method is called to remove the will. The `cancelWill` method means that an agent will can be put in place for short periods of time, or for the duration of a program. The lines 5-8 in Figure 2 insert a tuple into the tuple space with a name in it, then

```
1  public class TestWill {
2    TupleSpace gts = new TupleSpace(‘‘localhost",8989);
3
4    public TestWill() {
5      MyWill theWill = new MyWill(gts,"Antony");
6      gts.createWill(TheWill);
7      gts.out(new Tuple("Antony"));
.......  // do something
8      t = gts.rd(new Template("Antony"));
9      gts.cancelWill();
10   }
```

Figure 2. Example - TestWill Class.

does something and before terminating removes the tuple containing the name. If run-time decides that the agent has died somewhere in between inserting the tuple and removing the name tuple then the will is executed. It should be noted that when the `createWill` method completes the will is *guaranteed* to be in place and usable. It should also be noted that although a will is notionally attached to a tuple space the will can access any tuple space that it has access to (the handles for the tuple spaces would have been passed in prior to the object being made a will), and can perform any of the Linda operations it wishes to.

For efficiency reasons we have also added a method called `executeWill` to the *TupleSpace* class. This causes the will to be executed, and blocks the current thread of execution until that has happened. This may seem a natural enough extension, but the implications are large, as a new style of programming becomes viable, which may or may not want to be encouraged (the movement of computation to the server may not want to be encouraged).

Figure 3 shows the will class. When it is instantiated the parameters it requires are passed in which are a tuple space handle and a string. All the `will` method does is to remove a tuple from the tuple space. The tuple it will remove is the name tuple, and thereby, even if the agent becomes disconnected before it can tidy the name state this agent will performs the task for it.

There is a restriction made on the object that is migrated; it cannot perform any I/O operations except through tuple spaces (in other words it can insert tuples that other agents read and then print or whatever but it can not perform I/O operations itself). In the current implementation if the will does perform I/O operations these are performed through the console of the server. We used to place a restriction that all instantiations of classes that are not in the standard JDK must be made *prior* to the object being passed as the parameter to a `createWill` method. This was because obviously the remote server cannot go back to request the class code when the agent is being executed. However, this restriction has been removed. Whenever an object is to be migrated, the class file is analysed and all potential class files are detected and packed with the object.

Another issue is that the agent may not have failed, just that the connection was transient. In this case, if the run-time has been over pessimistic about when an agent has lost connection and the agent attempts to perform more operations the current run-time implementation raises an exception in the agent. This means the agent can deal with it – which usually means they create another will and change back the state (in this example reinsert their name tuple). In the example of the presence notification, the other users will see the user disappear and then reappear. In the same example, another approach is to have the will remove the name tuple and insert another name tuple, but indicating in the tuple that the user has disappeared. The application then displays this information to the user, so a user knows when someone has purposely left the system, and when a fault has caused the user to leave the system. These are issues that the application programmer can decide upon.

5.2. Implementation

The current implementation uses a centralised server to provide support for the tuple spaces. A centralised server means all the tuple spaces are managed within a single process, but the agents can reside anywhere. Although many of the current WAN targeted tuple space based coordination languages use centralised servers some use distributed kernels, for example WCL [17]. In this section we provide an overview of how we implemented the current implementation. At the end of section we discuss how we intend to extend this for a decentralised kernel.

The main server is written in Java. When a will is passed to the server, the server creates an instance of the object passed as the will, which must provide an interface *AgentWill*. When the agent that sent it fails the will method of this object is called. When the object acting as the will is passed to the server, the server uses a new subclass of *ClassLoader* that attempts to find the class files locally (ei-

```
1 class MyWill implements AgentWill, Serializable {
2   TupleSpace ts; String name;
3
4   public MyWill(TupleSpace ts, String name) {
5     this.ts = ts; this.name = name; }
6
7   public void will() {
8     ts.in(new Template(name));}
9 }
```

Figure 3. Example - The MyWill Class.

ther on disk or in a cache) and then remotely from the agent who provided the will. This is relatively simple and is the same method that is used in many Java based mobile object systems. The *ClassLoader* has been extended to determine which class files it may require in the future for a given (set of) class files. And retrieves these from the agent if required.

The wills are executed within a thread, with one thread being created for each will. This means that the wills can block whilst accessing a tuple space, without causing the kernel to stop, or the processing of other wills to stop. Indeed, in the current implementation a will could block forever.

As a slight aside, from an optimisation point of view, the *TupleSpace* objects detect whether it is running at the server or whether it is running remotely[2]. If it is running remotely then it opens up the necessary sockets to the server and if it is running under the same JVM as the server then it detects this, and is able to access the data structures used to store the tuples directly. This is important, because the migration and local execution of wills could increase the computational load on the server. This in turn would lead to worse performance of the server, and therefore of the system. However, by accessing the data structure directly many of the overheads of accessing a tuple space are removed [16]. This means that although running the will on the same JVM as the server increases the computational load, the server load is reduced so it appears roughly equivalent to performing the tuple space accesses remotely.

Figure 4 demonstrates the architecture of the Linda system used. The agent and the server process either run on the same or on different machines to the agent processes (normally a different machine). The server process maintains and manages all the tuple spaces; a cache of class files that the agent wills can use; and the agent wills. Although, the current implementation creates an active object as soon as the will arrives, another approach would be to only create the object when it was actually required. This means a will that is added and then removed incurs less of an overhead.

[2]Exploiting *transient* data types in Java.

5.3. Server fault tolerance

We have described how we provide fault tolerance for the application, but can this be implemented in a fault tolerance manner in the run-time system? Although not implemented in the current implementation, providing fault tolerance for the run-time system should be possible in a cheap and efficient manner. There is no need to check point the running code on the server, and this is good because this would require either modifications to the JVM being used or transformations of the class files. The first option is not acceptable and the second option would be time consuming if performed on the fly.

Instead, the server will checkpoint the tuple spaces as done in many Linda implementations [1] and the ClassLoader should create a local copy (on a persistent store) of all class files retrieved over the network and original state of the will should also be stored (on a persistent store). When the will is executed all tuples consumed and produced by the will should be stored and checkpointed as the tuple spaces are. If a total failure of the server should occur, the tuple spaces can be recreated; instances of all the wills can also be re-instantiated in their initial state. And the will method of the object called. When the will attempts to retrieve a tuple the stored tuples are re-returned and the produced tuples are discarded if they have already been produced. Once all the stored tuples have been used then normal operation can continue. This *must* be performed before any other operations on the tuple space are allowed in order to ensure that this is semantically correct. This also assumes that the will, given the same tuples in the same order, produces the same tuples in the same order.

We have discussed a centralised run-time system in this section. The implementation of this in a distributed kernel such as the one described in Rowstron [17] is quite feasible. There is an underlying assumption that if a particular server fails, then the sever will restart itself. We describe here the approach we are taking. Figure 5 shows the architecture. Each server is responsible for one or more entire tuple spaces.

In order to explain what happens lets us consider the

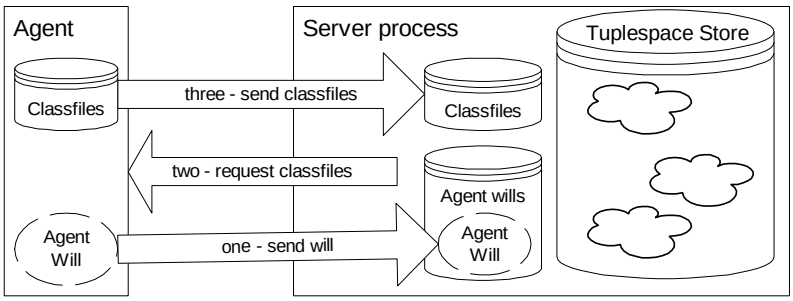

Figure 4. Architecture of the centralised system.

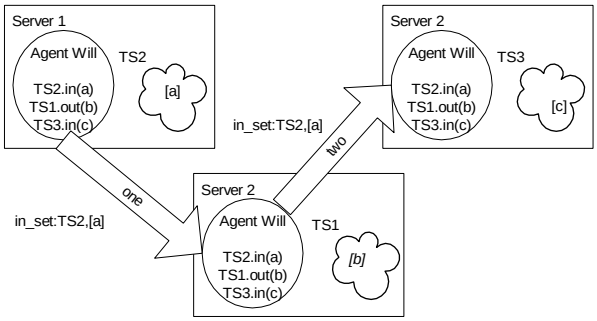

Figure 5. Architecture of the distributed system.

example in the figure. The agent will is attached to tuple space *TS2* initially. The agent will performs three operations: *TS2.in(a); TS1.out(b); TS3.in(c);*. The basic concept is that each time an agent will accesses a tuple space that does not reside on the local server, the agent will is moved to the correct server. So, in the example, the first statement of the agent will is executed, an *in*. The code is then moved to the location of *TS1*. The transfer of the state and classfiles is achieved in a reliable manner, using logs on a persistent store. However, the agent was in the middle of a method when the mobility became necessary. In order to migrate an executing code one would need to either augment the classfile or alter the JVM. However, because agent wills should be small we take an alternative approach. We create an *in_set* for the agent will and insert in the *in_set* a record of all the tuples consumed (and the order they were consumed) and a counter of the number of tuples produced by the will. When the agent will is moved, it is restarted, and the tuples in the *in_set* are returned as the results of the `in` and `rd` primitives, until the *in_set* is empty. The output tuples are discarded until the number of tuples produced matches the passed counter, which should incidently be when an operation that is local to the current server is performed. Any tuples consumed by the agent will are added to the *in_set* (in this case there are no tuples to be added as a tuple is inserted into the tuple space). When the next operation is performed, the agent will has to be migrated again, and this again transfers the state, code and the *in_set*. Again the agent will is executed, and the *in_set* is used to feed the agent will until all the entries in the *in_set* have been used once.

This approach provides a reliable, and fault tolerance approach to the problem, with low communication overhead. The same approach can be used to implement mobile coordination as described in Rowstron [15].

6. Future work

We are going implement the system using a distributed run-time system for Linda, using the method described in the last section. We also are currently considering two other issues: how much computation we allow a will to perform and how long, once it has been invoked, it should be allowed to run. We do not want agents to make their wills too computationally expensive. A will should, in general, perform a small number of accesses to the shared data structure then terminate. Wills that calculate pi to millions of decimal places are unacceptable, as they will degrade the performance of the servers managing the tuple spaces. Also, they can perform operations that block for example, awaiting a tuple. How long should a Will be allowed to exist? It is probable that the tuple garbage collection techniques described in Menezes et al. [9] can be extended to support garbage collection wills associated with tuple spaces as well.

7. Conclusions

We have identified and described a type of operation (person notification) that is common in CSCW applications that cannot easily be supported when using a DSM as the communication medium between agents. This is because the applications are written in a peer-to-peer style with the agents interacting directly with shared data structures.

We have proposed the use of mobile objects that we call agent wills that provide the ability for an agent to “tidy”

the shared data structures, should the agent become disconnected from the underlying infrastructure. The will object has the ability to outlive the agent which created it.

We have shown that such as system has been implemented on top of a Linda implementation, and that it works. Also, it has been shown that this has been implemented in a fashion that makes then underlying method used to provide this transparent to the programmer. By making it transparent it does not increase the complexity of the program for the programmer.

We finally conclude that the use of agent wills in distributed shared memory systems is novel and provides a higher level of fault tolerance, and supports the use of a peer-to-peer style of distributed program.

Acknowledgements

I would like to thank all those who have discussed the ideas of agent wills, especially Stuart Wray. I would also like to thank my colleagues at Microsoft Research for their comments on the idea of mobile coordination and specific comments on this paper, including Marc Shapiro and Cedric Fournet.

References

[1] B. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In *Research Directions in High-Level Parallel Programming Languages*, LNCS 574, 1991.

[2] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A programmable coordination architecture for mobile agents. *To appear IEEE Internet Computing*, 0(0):0–0, 2000.

[3] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[4] P. Ciancarini and D. Rossi. Coordinating Java agents over the WWW. *World Wide Web Journal*, 1(2):87–99, 1998.

[5] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Coordinating multiagent applications on the WWW: A reference architecture. *IEEE Transactions on Software Engineering*, 24(5):362–366, 1998.

[6] M. Day. Presence and instant messaging via HTTP/1.0: A coorindation perspective. In P. Ciancarini and A. Wolf, editors, *Coordination Languages and Models: Coordination99*, volume 1594 of *Lecture Notes in Computer Science*, pages 417–418. Springer-Verlag, 1999.

[7] P. Ferreira, M. Shapiro, X. Blondel, O. Fambon, J. Garcia, S. Kloosterman, N. Richer, M. Roberts, F. Sandakly, G. Coulouris, J. Dollimore, P. Guedes, D. Hagimont, and S. Krakowiak. Perdis: Design, implementation, and use of a PERsistent DIstributed Store. Technical report, QMW TR 752, CSTB ILC/98-1392, INRIA RR 3525, INESC RT/5/98, 1998.

[8] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[9] R. Menezes and A. Wood. Garbage Collection in Open Distributed Tuple Space Systems. In *Proceedings of 15th Brazilian Computer Networks Symposi um - SBRC'97*, 1997.

[10] R. D. Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[11] A. Omicini and F. Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-agent Systems*, 2(3):0–0, 1999. Special Issue on Coordination Mechanisms and Patterns for Web Agents.

[12] G. Picco, A. Murphy, and G.-C. Roman. Lime: Linda meets mobility. Technical Report Technical report WUCS-98-21, Washington University, Department of Computer Science, St. Louis, Missouri, 1998.

[13] A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1(3):167–179, 1998.

[14] A. Rowstron. Mobile co-ordination: Providing fault-tolerance in tuple space based co-ordination languages. In P. Ciancarini and A. Wolf, editors, *Coordination Languages and Models: Coordination99*, volume 1594 of *Lecture Notes in Computer Science*, pages 196–210. Springer-Verlag, 1999.

[15] A. Rowstron. Mobile co-ordination: Providing fault tolerance in tuple space based co-ordination languages. In *To appear Coordination'99*. Springer Verlag, 1999.

[16] A. Rowstron and A. Wood. BONITA: A set of tuple space primitives for distributed coordination. In *HICSS-30*, volume 1, pages 379–388. IEEE CS Press, 1997.

[17] A. Rowstron and S. Wray. A run-time system for the web co-ordination language. In *IEEE Workshop on Internet Programming Languages*, 1998. Chicago, USA.

[18] Sun Microsystems. Javaspace specification, revision 0.4. Final Specification., 1997.

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