A Run-Time System for WCL

Antony Rowstron[1] and Stuart Wray[2]

[1] Engineering Department, Cambridge University,
Trumpington Street, Cambridge, CB2 1PZ, UK.
aitr2@eng.cam.ac.uk
[2] Cambridge University Computer Laboratory,
Pembroke Street, Cambridge, CB2 3QG, UK.

Abstract. WCL is an inter-agent co-ordination language designed for Internet and Web based agent systems. WCL is based on shared associative memories called tuple spaces, as introduced in Linda.
In this paper we describe a novel run-time system for WCL. This distributed run-time system is radically different from traditional run-time systems supporting tuple spaces because it performs on-the-fly analysis of the usage of tuple spaces and moves tuple-space data between machines dynamically. Experimental results show that this approach provides significant speed improvements.

1 Introduction

Linda has been used with considerable success since the mid eighties [1–3] for inter-process co-ordination in the field of parallel processing. The early implementations were all "closed" in the sense that all the processes wishing to communicate had to be known at compile time. In the last few years there has been a drive to produce "open" implementations for workstations on a LAN, which do not require all the communicating processes to be known at compile time [4]. Still more recently, there have been attempts to create tuple space systems which support geographically distributed computing over the Internet. Announcements by Sun Microsystems about JavaSpaces show the level of commercial interest in the use of tuple spaces for inter-agent co-ordination over the Internet.

This recent work on tuple space systems for the Internet falls into two categories: new access primitives and new run-time systems. New tuple-space access primitives include WCL [5], BONITA [6], Sun JavaSpaces [7] and IBM TSpaces [8]. Examples of run-time systems supporting tuple spaces over the Internet are PageSpace [9] and C^2AS [10].

This paper outlines the run-time system currently being developed for WCL. All other tuple space run-time systems for Internet based co-ordination have been developed by making incremental adjustments to LAN based implementations, such as in PageSpace [9], C^2AS [10], and WWW-Paradise.

WCL is designed to provide general purpose support for geographically distributed applications – not for high performance parallel computing. For example, the run-time described in this paper will not provide high performance for

parallel image processing. The sort of applications we have been working on to date are small example applications; such as talk tools, shared white boards, asynchronous conferencing tools, event based applications, and agent based information gathering applications.

In this paper we describe a new distributed run-time system (or "kernel") used to support WCL. This run-time system builds on experience of creating LAN implementations, in particular the York Kernel II [11], but utilizes more much advanced on-the-fly analysis of tuple space usage to enable the migration of tuple spaces around a group of machines which are collectively running the kernel. This allows the system to tune itself automatically and dynamically to provide better performance, and this happens with no extra load being placed on the programmer using WCL. The kernel exploits the location transparency inherent in tuple space based co-ordination languages.

Although the system described in this paper is currently a prototype system, it demonstrates the advantages and speed of such a system. It should also be noted, that although designed to support WCL, this kernel could support any set of access primitives. It is a general purpose tuple space management system.

Section 2 briefly describes WCL, Section 3 describes the architecture of the run-time system, Section 4 shows some initial results for the kernel, and Section 6 describes the problems of transactions and how we plan to overcome them in this WCL implementation.

2 WCL

BONITA [6] was a first attempt at creating a new set of access primitives for tuple spaces that did not use *synchronous* tuple space access. The traditional Linda primitives are synchronous: the primitives block the thread of execution which performs the operation. Latency is high over the Internet, and synchronous primitives often block for significant periods. WCL was created after gaining the practical experience of using BONITA for real applications. Currently, *all* other tuple space languages use synchronous tuple space access[1]. The fundamental objects of WCL are the same as in all tuple space systems inspired by Linda: tuples, templates and tuple spaces:

Tuple A tuple is an ordered collection of typed fields. For example the tuple $\langle 10_{int}$, "Hello World"$_{str}\ \rangle$, contains two fields.

Template A template is similar to a tuple, but the fields do not need to have values. The templates: $\langle |10_{int}$, "Hello World"$_{str}\ |\rangle$ and $\langle |10_{int}, \square_{str}\ |\rangle$ will

[1] From an agents point of view primitives like `inp` and `rdp` provide *synchronous* tuple space access. The thread of execution in the agent performing the operation blocks until the tuple space has been searched for the tuple – if the tuple space is stored on the other side of the world then the time this takes can be considerable because of network latency. Hence, from an agents perspective the operation is synchronous *with respect to the tuple space*. A more detailed description is presented in Rowstron [6]

both match the tuple above. Matching is performed associatively. The number of fields in the template and tuple must be equal, the type of every field must match, and either the value of the fields must be the same, or the template field must have the "unspecified" value, represented here by □.

Tuple space A tuple space is a *logical shared associative memory* that is used to store tuples. A tuple space implements a *bag* or *multi-set*, so it can contain multiple identical tuples. There is no ordering of the tuples within a tuple space.

Agents communicate by inserting tuples into, and removing tuples from, shared tuple spaces. Note that while Linda originally had only one tuple space, used by all agents, WCL provides for many separate tuple spaces, as is usual for modern implementations. WCL does provide a global tuple space. Tuple space handles can be passed between agents in tuples inserted in tuple spaces. Co-ordination through tuple spaces provides two basic properties: temporal and spatial separation of agents. Temporal separation means that two agents can communicate even though they do not exist concurrently, because the tuple spaces through which the agents communicate can persist longer than the lifetimes of individual agents. Spatial separation means that agents can be anonymous, and other agents do not need to know their location or indeed their identity in order to communicate with them. These properties make the use of tuple spaces over the Internet ideal, and they allow the designer of run-time systems much freedom.

WCL is only a co-ordination language, and has to be embedded into a host computation language. Currently, bindings for Java and C++ have been developed. It should be noted that agents written in one language can communicate with agents written in another.

The access primitives provided in WCL are described in detail in Rowstron [5] and a detailed justification of why there are so many primitives is also given. Amongst the primitives provided by WCL there are asynchronous versions of the traditional `in`, `out` and `rd` Linda primitives and asynchronous versions of the bulk primitives `collect` [12] and `copy-collect` [13], as well as primitives supporting the streaming of tuples which also provide a simple event model. A brief overview of the WCL primitives is now presented:

First the operation used to create new tuple spaces.

tscreate() Create a tuple space and return the handle to that tuple space.

Next are the classic synchronous Linda operations, and their asynchronous equivalents.

out_sync(ts, tuple) Insert *tuple* into the tuple space *ts*. When the primitive terminates the tuple is guaranteed to be present (visible) in the tuple space.

in_sync(ts, template) This destructively retrieves a tuple which matches the *template* from the tuple space *ts*. The matched tuple is returned. The primitive is synchronous so the executing agent will block until a matching tuple becomes available.

rd_sync(ts, template) This is the same as `in_sync` except the tuple is not removed.

out_async(ts, tuple) Insert *tuple* into the tuple space *ts*. When the primitive terminates the tuple is *not* guaranteed to be present in the tuple space, but will appear in the tuple space as soon as possible.

reply_id = in_async(ts, template) This is an asynchronous destructive request for a tuple that matches the template *template* from tuple space *ts*. The primitive returns a reply identifier (*reply_id*) immediately and *does* not wait until a tuple matching the template is available (hence it is an asynchronous primitive).

reply_id = rd_async(ts, template) This is the same as `in_async` except the tuple is not removed.

The next operations provide a combined out and in.

touch_sync(ts, tuple) This primitive inserts the *tuple* into the tuple space *ts* and then attempts to destructively read it from the tuple space. This primitive is synchronous and when the primitive returns the tuple *tuple* is no longer present in the tuple space. When the tuple is inserted in the tuple space if there are other primitives blocked that could match the tuple being inserted then they compete for the inserted tuple and the winner is non-deterministic.

touch_async(ts, tuple) This primitive is the same as `touch_sync` except the primitive does not block.

Next we have operations to check on and if necessary to abandon pending asynchronous operations.

check_async(reply_id) This primitive is used to check if the tuple associated with the reply identifier *reply_id* is available. This primitive is asynchronous, so if the associated tuple is available then it is returned, but if it is not available then an empty tuple is returned. The primitive does not block.

check_sync(reply_id) This primitive is the same as `check_async` except it blocks until the result associated with *reply_id* is available.

cancel(reply_id) This primitive cancels the request associated with the reply identifier *reply_id*.

The move and copy operations provide for bulk transfer of tuples between tuple spaces. These operations are all considered to be atomic.

move_sync(ts$_{src}$, ts$_{dest}$, template) This moves all the tuples that match the template *template* from the tuple space ts_{src} to the tuple space ts_{dest}. The primitive returns a count of the number of tuples moved. If the source and destination tuple space are the same, then the 'moved' tuples are always visible.

copy_sync(ts$_{src}$, ts$_{dest}$, template) This primitive is the same as `move_sync` except the matched tuples are copied from the source to the destination tuple space. A count of the number of tuples copied is returned.

reply_id = move_async(ts$_{src}$, ts$_{dest}$, template) This moves all the tuples that match the template *template* from the tuple space ts_{src} to the tuple space ts_{dest}. The movement operation does return a count of the number of tuples moved, but unlike the synchronous version, the primitive does not block and returns a request identifier, which is used with either `check_sync` or `check_async` to actually retrieve the number of tuples moved.

reply_id = copy_async(ts$_{src}$, ts$_{dest}$, template) This primitive is the same as `move_async` except the matched tuples are copied from the source to the destination tuple space.

These last operations speed up bulk transfer of tuples to an agent and provide a basic event mechanism.

reply_id = bulk_in_async(ts, template) This primitive is used to retrieve all tuples that match the template *template* from the tuple space *ts*. The matched tuples are removed from the tuple space *ts*. The primitive returns a reply identifier *reply_id* which is then used in conjunction with the `check_async` and `check_sync` primitives to retrieve the matched tuples, one tuple at a time. The tuples can be considered as being streamed to the user agent. It should be noted that *there is no way* to determine the number of tuples returned, but this primitive can be used in conjunction with the `copy` and `move` family of primitives.

reply_id = bulk_rd_async(ts, template) This primitive is the same as the `bulk_in_async` primitive except the returned tuples are not removed from the tuple space it ts.

reply_id = monitor(ts, template) This places a monitor on a tuple space for tuples that match the template *template*. All the tuples within the tuple space *ts* that match the template *template* are streamed back to the user agent, and any tuples subsequently inserted are also sent to the user agent. The primitives returns a reply identifier *reply_id* which is used in conjunction with the `check_async` and `check_sync` primitives to retrieve the returned tuples, one tuple at a time. A `monitor` is guaranteed to match and return any tuple inserted that matches the template, regardless of whether the tuple is also matched by another primitive.

It should be noted that *reply_id* values returned by some of asynchronous primitives has no meaning other than for use with the
`check_sync` and `check_async` primitives and the *reply_id* only has any context within the agent that performed the asynchronous primitive. It can not be passed to another agent and then that agent use it to retrieve the result.

3 Run-time System Architecture

The run-time System architecture has been designed from the beginning to use data location transparency and to support geographically distributed computing. The entire run-time system, usually running across several machines, is called the

kernel. The kernel is composed of three distinct sections; the control system, the tuple management system and the agent libraries. The current implementation is very much an experiment to discover techniques that are useful for creating tuple space support on a large scale. The initial aim has been to demonstrate that the bulk movement of tuples is a useful tool for developing such kernels. Only the agent libraries are specific to WCL, the other parts of the kernel provide a generic tuple space management system. The architecture of the kernel is designed to be independent of the host language and to support multiple host languages.

The tuple management system uses multiple tuple servers. A tuple server is designed to provide agents with an efficient data access service. It processes instructions to insert tuples, instructions to return tuples, and instructions to move tuple spaces. A separate layer, the control system, determines when a tuple space should be moved, and dispatches the instructions to the individual tuple servers. Agents connect to the control layer, and to the tuple servers as required.

The separation between the control layer and the tuple storage layer has been made so that the tuple servers can be as efficient as possible. The tuple servers are the potential bottleneck of the system, therefore the quicker they can service requests the sooner blocked agents can continue. The control role is separated from the tuple management role, and performed on different machines, and even at different geographical locations.

3.1 Tuple management system

The tuple servers are distributed geographically around the area being supported. Each tuple server manages many entire tuple spaces. To use the analogy of Douglas [4] the tuple spaces can be considered as layers of a layered cake. Most LAN based run-time systems have tuple servers managing a slice of the cake, so each tuple server manages a small part of many tuple spaces. Alternatively, some LAN based systems have all the tuple spaces managed by a single centralized server (the entire cake). In our system each tuple server manages one *or more* layers of the cake, in other words each server manages multiple entire tuple spaces. This has an advantage from the point of view of fault tolerance, as it makes it cheap to take a snapshot of a tuple space, since there is no need for global synchronization between different tuple servers.

The approach of storing an entire tuple space on one of many single servers is different from other kernels. This approach makes the system bad for 'high performance parallel processing'[2] because multiple processes can not concurrently access the data, where as when the tuples of a single tuple space are distributed over multiple servers then multiple access to different tuples which are from the same tuple space is possible. Indeed, we have been involved in the development of several such run-time systems [4, 14, 11]. However, in such run-times providing fault tolerance of tuple spaces is difficult, supporting many of the more complex access primitives is difficult (requiring global synchronization between all servers storing the tuple spaces). Across a LAN this can be achieved but if this is then

[2] Or at least no worse than for traditional single process centralized kernels!

further distributed the latency between tuple servers impairs the performance greatly - this has been demonstrated when we attempted to run the C^2AS [10] kernel across multiple sites in Europe.

However, there is nothing theoretically to stop a single tuple server in this context in fact being distributed across several machines connected by a fast LAN. In many ways the work on this kernel was an experiment to see if storing entire tuple spaces on a single server really could work and if the migration of these tuple spaces between single servers was possible and still provide reasonable performance.

Finally, it should be noted that the tuple servers are written in C++ and use sockets directly so as to have complete control over their communication.

3.2 Control system

The control system acts as a manager, with which the tuple servers register, and the control system also presents agents with their entry point to the system. Currently, the control system uses a "well known" Internet address and port. When an agent is started this is either provided on the command line or read from an initialization file. When an agent connects to the control system it tells the control system its geographical location, and is passed a handle for the universally accessible tuple space, called the *Global Tuple Space*. The geographical location of an agent is also specified either on the command line or in the initialization file.

When the tuple servers register with the control layer they specify their geographical location, and this defines the areas over which they provide the best coverage. The geographical model used is one in which named regions are completely contained within other named regions, forming a tree structure like a file system directory structure (see figure 1 for a graphical representation). Currently, the locations are fairly large and the name space is fairly shallow, so for example a location might be *Cambridge-UK-Europe*. However, the granularity of locations can be as fine as required. For example, a tuple server running on our machine might for example use the location *428-Austin-CL-CU-Cambridge-Anglia-England-UK-Europe-World*. Where 428 is the office number, Austin is the building, CL is the department, CU is Cambridge University, Cambridge is the town, Anglia the region, UK the country etc. An agent on the same machine would share the same location description. It should be noted that a tuple server can serve tuples to and accept tuples from an agent running anywhere.

The aim of the geographical descriptions is to provide a gross indication to the run-time system of where tuple spaces should be stored when being accessed by agents. Therefore, the tuple server running on a machine in Cambridge would not be a good choice to store a tuple space being used only by agents currently running in California. However, if they were all running in England it may well be a good choice. When a tuple server registers, as well as telling the control layer its geographical location, it also provides information about its processing power. This enables the control system to ensure that the load placed upon the tuple server is acceptable for its power. This information could be obtained

automatically by making the tuple servers perform tests when they initialize, and then the servers could monitor the load on the machine and reduce or increase their processing power co-efficient with the control layer as appropriate.

In the prototype control system, agents can come and go freely. Tuple servers can join at any time, but currently can not leave. In a full implementation of the system this would of course be necessary and there are no fundamental reasons why this is hard. The control system would need to be told that a tuple server wanted to leave and it could then arrange to move all tuple spaces from that tuple space server to other servers. It should be noted that an agent can re-register to provide different geographical location information. This is important if migrating agents are to be supported.

The control system generates tuple space handles (global names for tuple spaces), monitors tuple space usage and decides when tuple spaces should be moved. Every time an agent becomes capable of using a tuple space the control system is informed, and every time an agent loses the ability to use a tuple space the control system is informed. Therefore, the control system maintains an overview of all the tuple spaces, and which agents are currently using them. Although, the control system is currently centralized, the kernel has been designed to allow the control system to lag behind, thus overcoming the problem of it becoming a bottleneck. We are currently investigating the possibility of a decentralized control system. However, a centralized control system does have the advantage that it can perform garbage collection using the method of Menezes and Wood [15].

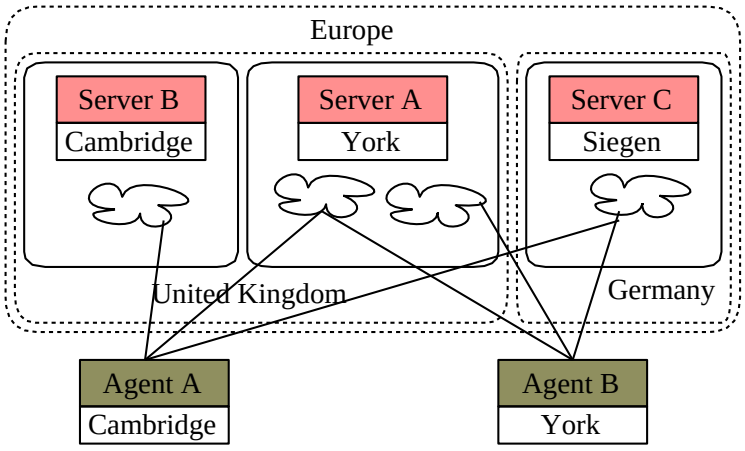

Fig. 1. Information stored in control system.

Figure 1 shows the information stored and managed in the control system. It shows three tuple servers, one in Siegen, one in Cambridge and one in York. All the servers are grouped geographically and hierarchically (eg. Europe contains

both the United Kingdom and Germany). There are also two agents, whose locations are York and Cambridge. The clouds represent tuple spaces, and their position shows which tuple server is physically looking after them.

The control system attempts to optimize the performance of the system by moving tuple spaces to the best position within the entire system. Therefore, in Figure 1 the control system analyses its usage and location information and observes that the two agents using the tuple space stored at Siegen are in fact in the United Kingdom. Therefore, it decides to move the tuple space. If both the agents were at the same physical location then the tuple space would be moved to the tuple server at that location. However, either of the tuple servers in the United Kingdom could be just as good, therefore the control system looks at the number of tuple spaces on each and then makes a decision to migrate it to either Cambridge or York. As more agents use it, the position of the tuple space may change again.

The current prototype system uses a very simple heuristic to decide when tuple spaces should be moved. It is based solely on the geographical information provided when the agents and tuple servers are started. Section 5 describes our current work on how this can be extended.

The replication of tuple spaces is not currently supported, mainly because access controls to tuple spaces have not yet been implemented. Access controls will provide information needed to sensibly decide when to replicate tuple spaces. It should also be noted that like the tuple servers, the control system is also written in C++ and uses sockets directly.

3.3 Agent library

The agent library is the part of the kernel that is embedded into each agent. These routines manage all the interaction with the other parts of the kernel, provide local tuple storage data structures, and determine where to find tuple spaces. All of this functionality is transparent to the programmer, who simply uses the high level WCL primitives. The fragment of C++ code shown in Figure 2 demonstrates the code a programmer writes to create a tuple space whose handle is stored in the variable TS1 and then inserts a tuple into the tuple space of the form [x_{int}, "D"$_{str}$].

```
TupleSpaceHandle TS1;
int x = 10;
TS1.createts();
out_async(TS1, WCL_INT(x), WCL_STR("D"), WCL_END);
```

Fig. 2. Example using WCL primitives embedded in C++.

So far, agent libraries have been written for C++ and Java. Agents written in one language can communicate with agents written in other languages. It should be noted that the Java embedding supports only the basic primitive types

in Java, rather than objects. However, the control system and the tuple management system are unaware of the types embedded within a tuple. Therefore, a language embedding could add types, provided the new 'type codes' do not interfere with existing embeddings. Currently, we control the type codes strictly, but we are considering schemes that would support the dynamic allocation of type codes managed by the control system.

The Java embedding is written in Java and can be used in either applets or stand-alone applications. If it is being used in an applet it detects this and connects to a proxy which is running on the web server which provided the applet. The proxy server is then able to forward the requests to the appropriate servers. Once the security model in Java is relaxed, the applets will be able to communicate directly with the servers.

3.4 Tuple space management and migration

In the last sections we have described the basic structure of the kernel, and now we consider briefly how the interaction between the agent library and the tuple server and control layer is managed.

Since tuple spaces migrate around the system, how does an agent know where to find a particular tuple space? Tuple spaces are identified by tuple space handles, which contain a globally unique tag and information (IP address and port number plus other information) about which tuple server is thought to be storing the tuple space. When an agent wishes to access a tuple space it looks at this information and if a connection is not already open to the appropriate tuple server, it opens a new connection and sends the requests to that tuple server. If that tuple server still holds the tuple space then the appropriate operations are performed there and the results returned to the agent. However, if the tuple space has migrated, then the tuple server will return to the agent a "forwarding address", in the form of a new tuple space handle with more up-to-date location information, but the same unique tag as before. The agent stores this and uses this new tuple space handle whenever it is presented with the superseded tuple space handle. It is not feasible to expect a tuple server to maintain a list of all tuple spaces it has ever seen for all of its life, so the tables are periodically flushed. If the tuple server receives operations on a tuple space it no longer knows about, then the agent which sent the operations is informed. That agent must then query the control system to find the current location of the tuple space. The same mechanism can be used to allow servers to be removed. If the agent can not connect to a tuple server, it can ask the control layer where a tuple space is. The control system, given any tuple space handle, can provide an up-to-date version of that tuple space handle because it controls the movement of tuple spaces, so it knows their current locations. Because of the globally unique tag within a tuple space handle it is not necessary for the control system to store all tuple space handles ever associated with a tuple space; outdated tuple space handles can always be matched by the control system to the up-to-date one.

The current kernel is scalable and could support thousands of tuple spaces and agents, but performs best when there are a small number of agents using

each tuple space. Because a tuple space is stored on a single tuple server at a particular location all the access for that tuple space must be sent to that tuple server. However, if several 'popular' tuple spaces are stored on a single tuple server, one or more of them can be moved to other tuple servers, so as to balance the load between them. It is safe to say that this kernel supports more agents, and provides better access, over the whole system, than existing LAN based implementations extended for use over the Internet.

The migration of a tuple space is not as simple as it may at first seem; often there will be operations pending on a migrating tuple space and care must be taken that these complete correctly. Another concern is that there are primitives in WCL that must be applied to all the tuples in a tuple space rather than a subset. A problem is introduced because an agent can be informed that a tuple space has been moved to a new tuple server, and then contact that tuple server, *before* the messages containing the tuple space is received by the next tuple server. Therefore, some primitives have to be queued because otherwise it is possible to introduce race conditions. The tuple space handle has information encoded in it that allows a tuple server to know whether the tuple space handle represents a new tuple space, or whether a tuple space handle represents a tuple space being moved. Therefore we ensure that commands to a new tuple space are never queued awaiting the arrival of non-existent tuples.

4 Experimental results

The aim of the prototype is to investigate the migration of tuple spaces, and we now present some experimental results which demonstrate the effectiveness of tuple space migration in kernels for Internet based co-ordination. The experimental results were obtained by running the kernel across three sites; Cambridge University (UK), the University of York (UK) and the University of Siegen (Germany). At Cambridge the tuple server was run on a Linux PC workstation, at Siegen on a DEC Alpha workstation and at York on a SGI Indy workstation.

The results shown in Figure 3 show the time taken to insert (`out_sync`) and retrieve (`in_sync`) 2500 tuples from a tuple space. The columns show the location of the agent performing the insertion or retrieval. The rows show where the tuple space was being stored at the start of the test. For the retrieval timings two times are given, marked (*"disabled"* and *"enabled"*). The row marked *"disabled"* shows the time when the migration of tuple spaces was disabled, and the row marked *"enabled"* shows the time when it was enabled. It should be noted that the retrieval timings include the time taken to register with the system, so this ensures that the times represent the time to move the tuple space as well as retrieve the tuples. This is important because if the time taken to fetch and move the tuples was greater than just fetching them then there is no point in migrating the tuples. When the migration is enabled the final location of the tuple space will be the same location as the agent performing the retrieval operation. The speedup (number of times faster) obtained when using tuple space migration is shown on the row below the two retrieval results.

Obviously, the results are highly dependent on network load, and the results presented here were gathered early on a Sunday morning, when network traffic in Europe was low. It should also be noted that the insertion timings are for synchronous insertion.

Tuplespace Location	Type of Operation	Agent Location		
		York	Cambridge	Siegen
York	2500 `out_sync`	6.79	60.44	114.55
	(disabled) 2500 `in_sync`	11.35	56.31	115.56
	(enabled) 2500 `in_sync`	11.35	2.94	9.73
	Speedup	-	19.2	11.9
Cambridge	2500 `out_sync`	63.11	1.50	106.16
	(disabled) 2500 `in_sync`	62.64	2.32	112.91
	(enabled) 2500 `in_sync`	11.65	2.32	8.40
	Speedup	5.4	-	13.4
Siegen	2500 `out_sync`	108.21	110.23	4.51
	(disabled) 2500 `in_sync`	120.22	112.23	7.57
	(enabled) 2500 `in_sync`	11.86	3.77	7.57
	Speedup	10.1	29.8	-

Fig. 3. Insertion (`out_sync`) and retrieval (`in_sync`) timings.

These results demonstrate that the use of bulk movement of tuple spaces within the kernel gives a large increase in performance. This performance increase is due to reduced communication across the Internet, the ability to tune packet sizes to an optimum size for routing through the Internet, and the much lower latency of performing local operations compared to operations at a remote site. It should be noted that the results presented here do not deal with concurrent access to tuple spaces by more than one agent – this is because of the difficulty of setting up experiments over multiple sites that test this repeatably enough to present results in this paper. The tests of concurrent access to a tuple space by multiple agents does show a speedup when using migration. Indeed, this demonstrated that some care might have to be taken to prevent continuous migration between tuple servers. However, the nature of most tuple space applications makes this unlikely anyway, and a "resistance" to migrating too often can be added in the control layer.

It should be noted that the migration of a tuple space may result in faster access for some agents accessing the tuple space, whilst providing slower access for other agents. However, the aim is that on *average* all the agents can access the tuple space quicker. If this is not the case then the migration of a tuple space can be regarded as a failure.

5 Migration control experiments

Some consideration has been made as to how the simplistic heuristic of when to move a tuple space can be extended, for example by adding dynamic checking of communication latency between agents and tuple servers and by using historical data. Our initial investigations have concentrated on using the "historical" load on the Internet as a way to predict when and where network load will again be high. This was motivated by an observation from using the Internet that at some times of the day access to many countries is far slower than at other times of the day. For example, from the UK access in the afternoon to USA is very slow.

We set up an experiment to use the UNIX ping tool every fifteen minutes to measure the time taken for a round-trip to a number of sites. Each time 10 packets were sent, and the results then analyzed. Figure 4 provides the average packet round trip time from a machine at the Computer Laboratory at Cambridge University to a number of other computers around the world. The results are shown for only three days, starting 00:00 on the 20th May 1998 GMT, and the last reading is at 00:00 on the 23rd May 1998 GMT. This represents three days during the working week. We monitored many sites during the time, and sites were chosen because of their geographical location and because of the routers that were 'normally' chosen to reach them (we detected this using the UNIX `traceroute` tool).

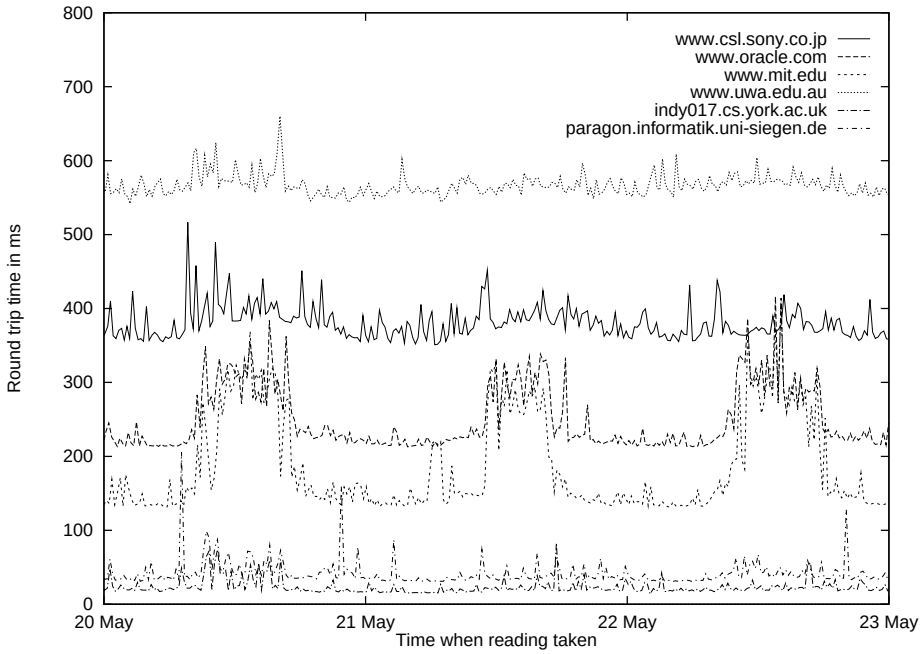

Fig. 4. Round trip time for a packet.

What is interesting is that the round trip time to the machines in the USA increases noticeably when America goes to work, and drops again when working day finishes. The traces for the weekends interestingly show no such increase. What is also interesting is that there is no such peak for machines located outside the USA. The latency remains constant.

This was interesting because we had expected to see latency increase whenever a country was at work! Ping uses ICMP packets, so there is no retransmission of lost packets, as there would be for the TCP connections used by the WCL system. Therefore, we next examined the number of lost packets. Figures 5 and 7 show respectively the results for the sites outside Europe and the results for the two sites in Europe (used in the previous experiment).

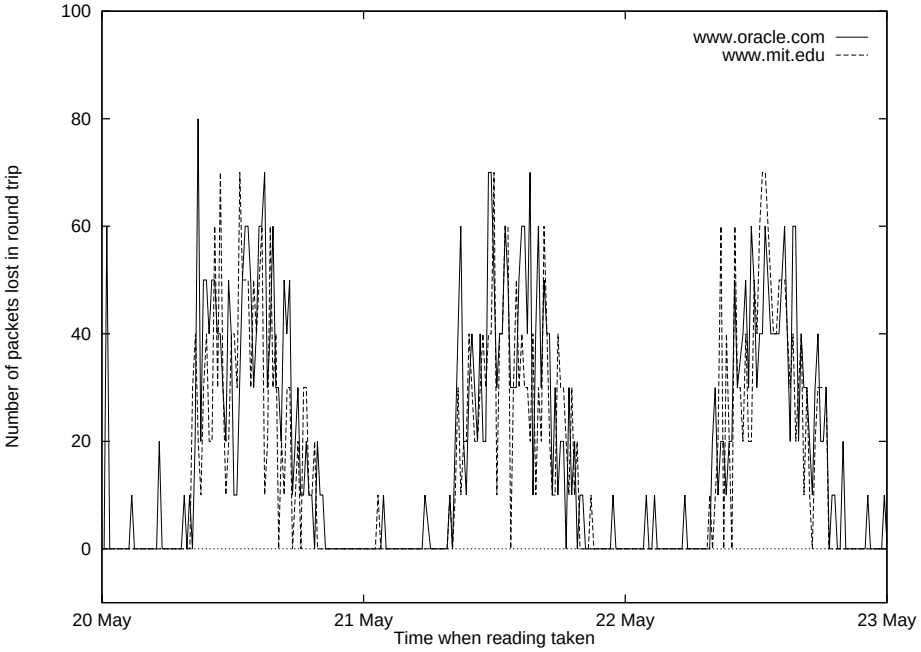

Fig. 5. Percentage of packets lost to North America.

It is interesting that the number of packets to all countries outside Europe rises when the USA is working. By contrast, the number of packets lost in Europe is very small.

By using traceroute to these addresses and to other randomly chosen class C addresses we were able to investigate where the packet loss was occurring. Traffic from Cambridge to sites in the USA travel from external-gw.ja.net across the Atlantic to teleglobe.net and thence via other Teleglobe routers to other service providers. Unsurprisingly, most of the delay is introduced on the link across the Atlantic. However, hardly any packets were dropped on our side of the Atlantic

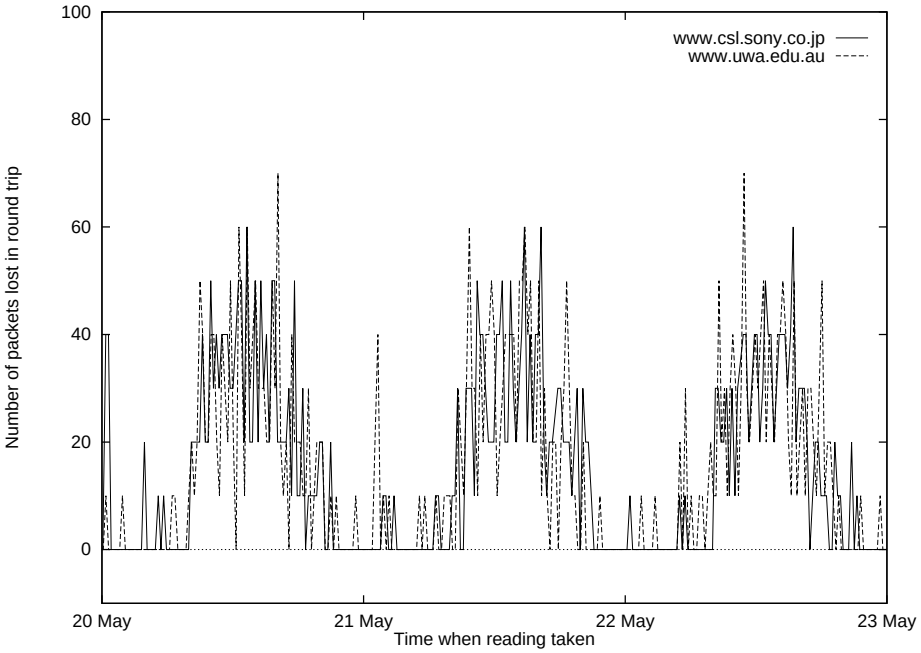

Fig. 6. Percentage of packets lost going through North America.

— most of the lost packets were dropped by telegloble.net or other Teleglobe routers.

Although geographical position provides a strong indication of latency and traffic capacity between sites (if for no other reason than the speed of light!), we have come to the conclusion that in many ways the *service providers (who provide the routers) define the true geography of the Internet.* Rather than using physical geography we need to build an "image" of how the Internet is interconnected, and the times when certain key routers are liable to be busy in order to determine when a it is a good time to move a tuple space.

Future work on taming the current wild nature of the Internet will dramatically affect how one should both measure network capacity and how to use these measurements to decide when to migrate data. For example, end-to-end bandwidth negotiation and traffic-smoothing gateways would both have a big impact in this area.

6 Further work

The existing prototype needs improvement in a number of areas before it is sophisticated and robust enough for wider use:

Surviving agent failure The kernel can survive agent failure, but agent failure is still a problem for applications. Agents must remove tuples from a

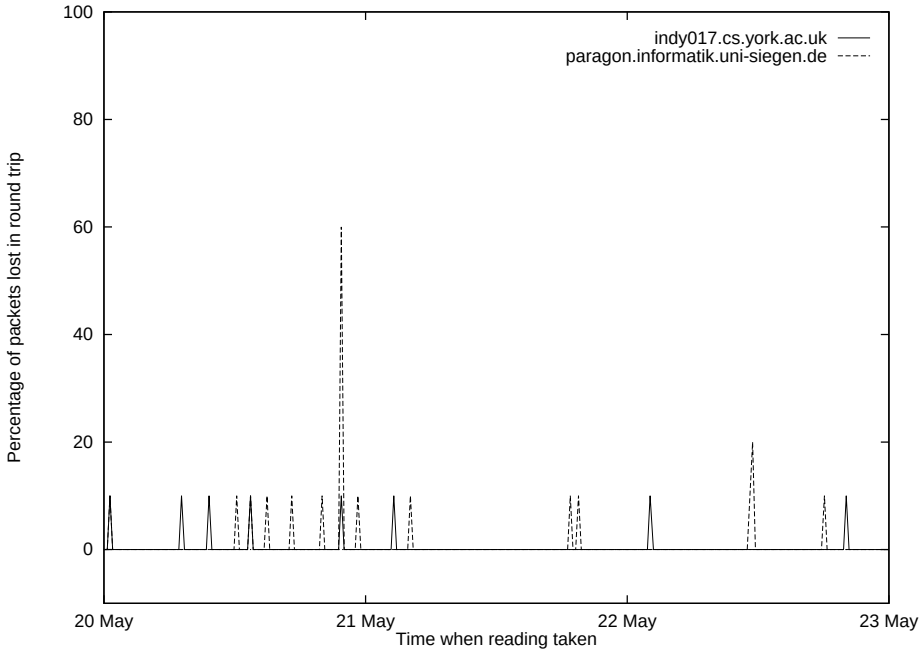

Fig. 7. Percentage of packets lost to Siegen and York.

tuple space, for example to increment a counter or claim a lock. If an agent terminates unexpectedly while holding such a tuple, then the whole application may deadlock.

Some implementations [16, 17, 7] offer solutions to this problem based on transactions, but none of them are completely satisfactory.

All the implementations take a similar approach. Two new commands are added, START and COMMIT. The START command causes all tuples inserted to be held, and all tuples removed by the co-ordination commands in between the START and the COMMIT to be held. When the COMMIT occurs any inserted tuple appear in the tuple space. This way, if the COMMIT is never reached the inserted tuples do not appear in the tuple spaces and any removed tuples can be replaced in the tuple spaces.

The problem with extra locking operations like START and COMMIT is that they alter the underlying semantics of the co-ordination fragments they are placed around. This is demonstrated by the code fragments given in Figure 8.

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

Fig. 8. Example of transaction problems.

The two fragments of pseudo code shown in Figure 8 are assumed to be performed on the same tuple space, and represent a trivial but yet important co-ordination construct using tuple spaces. This construct is an explicit synchronization between the two fragments. If the START and COMMIT are placed around the co-ordination construct in Fragment two then this does not alter the semantics. However, if the START and COMMIT are placed around the co-ordination construct in Fragment one then Fragment two will deadlock. The tuple inserted in Fragment one into the tuple space will not appear until after the tuple inserted in Fragment two has been read, but this can not occur until after the tuple inserted in Fragment two appears in the tuple space, thus the Fragments deadlock.

In order to overcome this problem we have used the novel approach of migrating "co-ordination fragments" from agents to tuple servers, which we have called mobile co-ordination [18]. These fragments will contain WCL operations with Java. This technique has also improved latency since fewer round-trips are necessary, and it reduces the bandwidth needed for similar reasons.

Surviving tuple server failure The development of a fault tolerant kernel will be facilitated by our approach of storing a tuple space on a single tuple server, because this makes it easy to take a snapshot of a tuple space, as in the LAN-based PLinda [16]. It will be necessary to replicate the tuple space over perhaps multiple sites.

Control system The control system is centralized, and for reliability and load sharing this needs to be changed to a distributed version. Also we need to work on the algorithms and techniques used to control when tuple spaces are moved. This will include taking account of network topology, capacity and recent and historic load.

Tuple and tuple space management We need to add access control mechanisms to tuple spaces and potentially tuples. This will also allow us to consider garbage collection of tuple spaces and tuples.

7 Conclusions

In this paper we have described a novel implementation of a distributed kernel supporting tuple spaces, which can be used for the co-ordination of geographically distributed agents over the Internet. The kernel uses tuple space usage analysis to transparently move tuple spaces to the tuple servers which can best support the dynamically changing access patterns of the agents.

Recent interest in tuple space based systems for the Internet and recent product announcements shows the importance that tuple spaces may play in the future. If the true potential of the tuple space paradigm is to be fulfilled the development of run-time systems which support them in a scalable and efficient manner must be developed. We are currently addressing this, and the current prototype implementation represents an initial step in the right direction.

The work on implementations has revealed a weakness both in the current prototype and the WCL languages, in their inability to support fault tolerance.

This is to be addressed with the concept of mobile co-ordination. This should be seen as complementary to the mobile tuple space concept presented in this paper.

Some implementors of tuple space systems for wide area use have suggested that centralized implementations are preferable to distributed ones. However, the experimental results from our prototype distributed kernel show the potential performance gains from bulk data migration. In some cases we have observed a 30-fold speed increase. These figures argue compellingly for distributed tuple space run-time systems.

Acknowledgements

We would like to thank Prof. Andy Hopper and the Olivetti and Oracle Research Laboratory for funding this work. We would also like to thank Alan Wood (York University) and Thilo Kielmann (Siegen University) for allowing us to use their computing resources. We would also like to thank Sheng Li for his advice.

References

1. N. Carriero. *Implementation of Tuple Space Machines*. PhD thesis, Yale University, 1987. YALEU/DCS/RR-567.
2. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
3. D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
4. A. Douglas, A. Wood, and A. Rowstron. Linda implementation revisited. In *Transputer and occam developments*, pages 125–138. IOS Press, 1995.
5. A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1998.
6. 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.
7. Sun Microsystems. Javaspace specification, revision 0.4. Unpublished beta draft specification., 1997.
8. Peter Wyckoff, Stephen McLaughry, Tobin Lehman, and Daniel Ford. TSpaces. To appear in *IBM Systems Journal,* August, 1998.
9. P. Ciancarini, A. Knocke, R. Tolksdorf, and F. Vitali. PageSpace: An architecture to coordinate distributed applications on the web. In *5th International World Wide Web Conference*, 1995.
10. A. Rowstron, S. Li, and S. Radina. C^2AS: A system supporting distributed web applications composed of collaborating agents. In *WETICE*, pages 87–92, 1997.
11. A. Rowstron. *Bulk primitives in Linda run-time systems*. PhD thesis, Department of Computer Science, University of York, 1997.
12. P. Butcher, A. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.
13. A. Rowstron and A. Wood. Solving the linda multiple `rd` problem using `copy-collect`. *Science of Computer Programming*, 31(2-3), July 1998.

14. A. Rowstron and A. Wood. An efficient distributed tuple space implementation for networks of workstations. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96*, volume 1123 of *Lecture Notes in Computer Science*, pages 510–513. Springer-Verlang, 1996.
15. R. Menezes and A. Wood. Garbage Collection in Open Distributed Tuple Space Sys tems. In *Proceedings of 15th Brazilian Computer Networks Symposi um - SBRC'97*, 1997.
16. B. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In *Research Directions in High-Level Parallel Programming Languages*, LNCS 574, 1991.
17. Scientific Computing Associates. *Paradise: User's guide and reference manual.* Scientific Computing Associates, 1996.
18. A. Rowstron. Mobile co-ordination: Providing fault tolerance in tuple space based co-ordination languages. In *To appear Coordination'99*. Springer Verlag, 1999.