Short Running Title: WCL: A Co-ordination Language for Geographically Distributed Agents

Contact Author: Dr. Antony Rowstron

Address: Computer Laboratory, Cambridge University, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK

Phone: +44 1223 334418

FAX: +44 1223 334678

email: Antony.Rowstron@cl.cam.ac.uk

WCL: A Co-ordination Language for Geographically Distributed Agents

Antony Rowstron

Computer Laboratory,

Cambridge University,

New Museums Site,

Pembroke Street,

Cambridge,

CB2 3QG, UK

Antony.Rowstron@cl.cam.ac.uk

Abstract

In this paper a tuple space based co-ordination language, and a run-time system which supports it is described. The co-ordination language is called WCL, and it is designed to support agent co-ordination over the Internet between agents which are geographically distributed. WCL uses tuple spaces as used in Linda. WCL provides a richer set of primitives that traditional tuple space based systems, and provides asynchronous and synchronous tuple space access, bulk tuple primitives, and streaming primitives which, as a whole, provide a complete framework more suited to co-ordination over the Internet compared with the Linda primitives.

The primitives emphasise efficiency and location transparency (of data and agents) and this is exploited in the current run-time system used to support WCL. The run-time system is described in the papers and is distributed and uses the location transparency and dynamic analysis of tuple space usage to migrate tuple spaces around the distributed system. Some initial experimental results are given which demonstrate the performance gains of using the tuple space migration.

The paper motivates the inclusion of many of the primitives, and demonstrates how a well designed set of primitives provides performance and efficiency. The JavaSpace primitives are used as an example of how the choice of primitives can detrimentally effect the efficiency of the language, and exclude required co-ordination constructs.

1 INTRODUCTION

Co-ordination languages embody the concept that co-ordination should be separated from computation [Gelernter and Carriero 1992]. If two entities executing concurrently are to interact and co-ordinate, they need to be able to communicate, and a co-ordination language enables this communication and interaction. In this paper we describe a co-ordination language called WCL, together with an associated run-time system. Both have been designed to provide efficient support for different types of applications composed of geographically distributed interacting components. Some of the components interact with humans, and others interact with other components. We will refer to these components as agents.

WCL uses shared tuple spaces, as used in Linda [Carriero and Gelernter 1990]. It provides a much richer set of access mechanisms to the tuple spaces than Linda, supporting asynchronous and synchronous tuple space access primitives, bulk tuple primitives, and streaming primitives which, as a whole, provide a complete framework more suited to co-ordination over the Internet. The primitives emphasise efficiency and location transparency (of data, and agents) and concurrency transparency (ie. shared access to shared tuple spaces without interference).

The use of tuple spaces for co-ordination over the Internet was first proposed by Gelernter [Gelernter 1992], who described the concept of "mirror worlds" built using tuple spaces which would provide access to information from around the world. More recently, the PageSpace project [Ciancarini *et al.* 1996] has actually implemented an agent framework for use over the Internet using tuple spaces as the co-ordination medium. PageSpace does not allow the agents distributed over the Internet to access the tuple spaces directly, and uses the concept of personal agents in a safe 'pagespace'. Agents within the 'pagespace' co-ordinate using the Linda primitives and tuple spaces, and these agents communicate with the agents actually executing over the Internet using a different mechanism. All the work on the use of tuple spaces over the Internet to date, eg. PageSpace and C^2AS [Rowstron *et al.* 1997], has concentrated largely on using the traditional Linda primitives, and extended, or modified run-time systems from LAN based implementations. Interest in using tuple spaces as a co-ordination mechanism for co-ordination over the Internet has now reached the point where Sun Microsystems are developing a specification for a tuple space based system, called JavaSpace [Sun Microsystems 1997].

WCL does not specify any way to express computation, and computation is provided by a host language into which WCL is embedded, and WCL and the run-time system currently supporting WCL is designed to be independent of the host languages used. WCL attempts to achieve language transparency. By contrast, the JavaSpace proposal has attempted to create a tighter coupling between the co-ordination language and a particular host language, in the case of JavaSpaces' Java. In WCL this has specifically

attempted not to be done, because of the success and variety host languages that Linda has been embedded into in the last decade. Linda was also designed to be independent of the host language used, and has been successfully embedded into many different languages, including C [Carriero 1987; Leichter 1989], Fortran [Yuen *et al.* 1996], SETL [Douglas *et al.* 1995a; Hasselbring 1994], Prolog [Ciancarini 1991; Bosschere and Wulteputte 1991], and Gofer [Douglas *et al.* 1996].

The outline of this paper is as follows; Section 2 describes tuples, templates and tuple spaces. Section 3 describes the motivation for the inclusion of the chosen primitives in WCL. Section 4 briefly compares WCL to JavaSpaces. Section 5 gives an overview of a novel architecture for a distributed run-time system which supports WCL over the Internet, and some initial timings are presented which demonstrating the potential advantages of some of the architecture, and finally Section 6 discusses future work.

2 TUPLES, TEMPLATES AND TUPLE SPACES

The fundamental objects of all tuple space based co-ordination languages are tuples, templates and tuple spaces. A tuple is an ordered collection of fields, with each field having a type and a value associated with it. Eg. $\langle 10_{integer}$, "Hello World"$_{string}$, $10_{integer}$, $1.0_{float}\rangle$. A template is similar to a tuple except the fields do not need to have values associated with them, but all fields must have a type. Eg. $\langle|\ 10_{integer}$, "Hello World"$_{string}$, $10_{integer}$, $1.0_{float}|\rangle$ and $\langle|\ 10_{integer}, \Box_{string}, \Box_{integer}, 1.0_{float}|\rangle$. Templates are matched to tuples using an associative match. The tuple $\mathcal{T}$ and the template $\mathcal{M}$ are said to match when:

$$\left(\sharp(\mathcal{T}) = \sharp(\mathcal{M})\right) \bigwedge \left(\forall j \in \{1..\sharp(\mathcal{T})\} : \left(t_j(\mathcal{M}) = t_j(\mathcal{T})\right) \wedge \left((v_j(\mathcal{M}) = v_j(\mathcal{T})) \vee (v_j(\mathcal{M}) = \Box)\right)\right)$$

is true, where $t_j(\mathcal{I})$ is a function which returns the type of field number j of tuple or template $\mathcal{I}$, $v_j(\mathcal{I})$ is a function which returns the value of field number j of tuple or template $\mathcal{I}$, and $\sharp(\mathcal{I})$ is the cardinality of tuple or template $\mathcal{I}$. So, for a match to occur the number of fields in the template must be equal to the number of fields in the tuple, for every field in the tuple the type of the field must match the corresponding field type in the template, and either the value of the fields must be the same, or the template field must be a formal. Therefore, the two example templates given above will match the example tuple given above.

A tuple space is a *logical shared associative memory* that is used to store tuples. A tuple space implements a *bag* or *multi-set*, in which the same tuple may be present more than once and there is no ordering of the tuples.

In WCL it is assumed that the tuple space structure is flat. Each tuple space is not related to any other tuple space. Also, tuple spaces are not considered first class, and subsequently, handles to tuple spaces are used. A handle can be considered as a pointer or reference to a tuple space and can be passed to other

agents through a tuple space. The current implementation of WCL creates a number of universal, "well known", tuple spaces upon startup which are available to all agents.

3 THE WCL CO-ORDINATION LANGUAGE

What is novel about WCL? It supports asynchronous tuple space access in a clean and efficient manner. BONITA [Rowstron and Wood 1997] was the first and only other tuple space based co-ordination language to use asynchronous tuple space access, and provided a poor set of access primitive, which has been addressed in WCL. It is the first tuple space based co-ordination language to support the streaming of tuples. This is supported through the use of the `touch_sync`, `touch_async`, `bulk_in_async` and `bulk_rd_async` primitives. Although the primitives of `bulk_in_async` and `bulk_rd_async` are loosely based on the primitives of a similar name in Objective Linda [Kielmann 1996] they provide a different semantics, and within WCL the need for extra local data structures has been removed. Within WCL it has been explicitly recognised that there are two very different semantics for the insertion of tuples into tuple spaces and this has been addressed. The co-ordination constructs supported in WCL are very different from in traditional Linda systems, and WCL represents one of a few attempts to provide extra functionality to tuple space systems specifically for use in high latency environments. We will now consider these issues in more depth.

The access primitives used in WCL are presented in Tables 1 and 2. Table 1 presents the basic access primitives, that provide both asynchronous and synchronous single tuple operations. Table 2 provides a description of the bulk primitives, and those primitives which enable the streaming of tuples. The exact *syntax* of the primitives is left to the person creating the embedding of WCL in a particular host language, as with the Linda primitives. To date the WCL primitives have been embedded in Java and C++, and the header files and class descriptions for these embedding are shown in Appendix A.

3.1 Asynchronous access primitives

A synchronous access primitive is one where the primitive does not terminate *until* a result is available from a tuple space. The agent performing the primitive and the tuple space must synchronise in order for a tuple to be returned. All the Linda primitives are synchronous. An asynchronous primitive is one which allows an agent to perform further co-ordination constructs or computation in parallel with the tuple space access, hence from an agents point of view the operation is asynchronous. There is no need for the agent and the tuple space to synchronise in order for the agent to retrieve a tuple.

Initially, the motivation behind using asynchronous access primitives was to provide better perfor-

Table 1: Basic tuple space access primitives - both asynchronous and synchronous.

Primitive	Description
tscreate()	Create a tuple space and return the handle to that tuple space.
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.
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.
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.
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).
rd_async(ts, template)	This is the same as `rd_async` except the tuple is not removed.
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.
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.
cancel(reply_id)	This primitive cancels the request associated with the reply identifier *reply_id*.

mance given a high latency network (eg. the Internet). Experience with profiling Linda systems [Rowstron *et al.* 1995; Douglas *et al.* 1995b; Rowstron and Wood 1996a] on LAN networks showed that even in these environments latency was a significant cost when performing tuple space operations. This led to the conclusion that time costs associated with synchronous primitives were not due to bottlenecks in the implementations but quite simply the time taken to send and receive messages over networks.

In order to demonstrate this let us consider the primitives `in_sync` and `rd_sync`. These primitives are synchronous and therefore block the thread of execution. In general the time that agent is blocked for can be described by a set of time cost categorisations (from the agent's point of view):

$$T_{Pack} + T_{SendRequest} + T_{Queue} + T_{Process} + T_{Block} + T_{SendReply} + T_{Unpack}$$

where T_{Pack} is the time taken to package the template; $T_{SendRequest}$ is the time taken to communicate the

request; T_{Queue} is the time the request waits before being serviced; $T_{Process}$ is the time taken to process the request, package the reply tuple; T_{Block} is the time spent waiting for a matching tuple if one is not available; $T_{SendReply}$ is the time taken to communicate the reply; and T_{Unpack} is the time taken to unpack the reply message.

Table 2: Streaming and bulk primitives

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

The `in_sync` and `rd_sync` primitives will therefore block an executing thread not just because a

matching tuple is not present but simply because of the latency and overhead costs. Asynchronous access primitives allow the time $T_{SendRequest} + T_{Queue} + T_{Process} + T_{Block} + T_{SendReply}$ to be performed in parallel with computation in the user agent that is performing the primitive, thereby allowing either the pipelining of tuple space access or the parallelisation of computation and communication. A simple example of pipelining tuple space access is shown in Figure 1[1)], and a simple example of performing computation and tuple space access in parallel is shown in Figure 2.

```
int rqid1 = ts1.in_async(new Template(new Integer(10)));
int rqid2 = ts2.in_async(new Template(new Integer(11)));
Tuple T1 = check_sync(rqid1);
Tuple T2 = check_sync(rqid1);
```

Figure 1: Pipelining tuple space access using asynchronous tuples.

```
int rqid1 = ts1.in_async(new Template(new Integer(10)));
perform_calc();
Tuple T1 = check_sync(rqid1);
```

Figure 2: Parallelising tuple space access and computation.

Bonita [Rowstron and Wood 1997] demonstrated that simple asynchronous access primitives could be provided directly to the programmer. However, it provided complex and difficult set of access primitives, whose functionality was heavily overloaded, in order to minimise the number of access primitives. WCL attempts to draw on the simplicity of access primitives, like Linda did, but at the expense of increasing the number of primitives.

Asynchronous primitives also provide the ability to perform different co-ordination constructs. Figure 3 shows an example of a new co-ordination construct using the C++-WCL embedding. Although this uses polling, it is all local polling within the agent, the checks sees if the result has arrived back from the remote run-time system, rather than contacting the remote run-time system.

3.2 The ordering of tuple inserts

Since the inception of Linda the question of whether the insertion of a tuple is a blocking or non-blocking operation has been a vexing one. Most implementations treat the insertion as non-blocking which has implementational advantages (from a performance perspective). Therefore, the tuple is not guaranteed to be visible to other agents when the primitive returns.

[1)] It should be noted that access to different tuple spaces can be pipelined.

```
int rqid1 = in_async(ts1, WCL_INT(10), WCl_END);
int rqid2 = in_async(ts1, WCL_INT(11), WCL_END);
while(!finish) {
  if (check_async(rqid1, WCL_INT(10), WCL_END)
    finish = do_op_1();
  if (check_async(rqid2, WCL_INT(11), WCL_END)
    finish = do_op_2();
}
```

Figure 3: A non-deterministic choice construct.

The problem with having a non-blocking or blocking insertion first became an issue when the two primitives `inp` and `rdp`[2)] were proposed by Carriero [Carriero 1987] and incorporated into the first implementations of Linda. There are a number of perceived "semantic problems" associated with these primitives which are used as the primary reason for their removal [Leichter 1989], replacement [Butcher *et al.* 1994] or, when implemented, behaviour which can potentially lead to unintentional program behaviour [Scientific Computing Associates 1995]. The example code fragments shown in Figure 4 written using the WCL primitives demonstrate the problem. Let us assume that the tuple space `ts1` is only accessible by the code fragments shown. Producer 1 and 2 both place two tuples into the tuple space `ts1`. Consumer 1 performs an `in_sync` primitive on one of the tuples and then performs a `move_sync` operation on the other. Consumer 2 performs two `in_sync` operations to retrieve both the tuples.

Both the Producers achieve the same final outcome, two tuples inserted into tuple space `ts1`. The outcome of Consumer 2 is independent of which Producer is used. This is because the primitives used in Consumer 2 *block* for a matching tuple to arrive. The outcome of Consumer 1 *is* dependent on which of the two insertion fragments is used. Producer 1 uses the `out_sync` which guarantees that the tuple is visible when the `out_sync` completes. Therefore, when Consumer 1 performs the synchronisation with Producer 1 using the ⟨"DONE"⟩ tuple, Consumer 3 is guaranteed to move the other tuple, and `x` will be set to 1. This is because the tuple ⟨10⟩ *must be visible*, because the `out_sync` was used. However, Producer 2 uses the `out_async` primitive which does not guarantee that the tuple will be present when it terminates. Therefore, the Consumer 1 and Producer 2 will synchronise on the ⟨"DONE"⟩ tuple, but the `move_sync` may yield a result of 0 or 1, because the first tuple may not yet be visible to other agents.

A formal analysis of three different semantics for the Linda `out` primitive has recently been completed by Busi et al. [Busi *et al.* 1997], which shows formally that `out` ordering is necessary. The tuple insertion ordering problem is used as the justification for including the `out_sync` primitive in WCL. One justification for also providing an asynchronous version `out_async` is simply to provide better performance. If a programmer

[2)]These primitives return either a tuple or a value to indicate that a matching tuple was not available.

Producer 1
`ts1.out_sync(new Tuple(new Integer(10)));` `ts1.out_sync(new Tuple(new String("DONE")));`
Producer 2
`ts1.out_async(new Tuple(new Integer(10)));` `ts1.out_async(new Tuple(new String ("DONE")));`
Consumer 1
`ts1.in_sync(new Template(new String ("DONE")));` `int x = ts1.move_sync(ts2, new Template(new Integer(10)));`
Consumer 2
`ts1.in_sync(new Template(new String ("DONE")));` `ts1.in_sync(new Template(new Integer(10)));`

Figure 4: Out ordering example.

knows that tuple insertion ordering is not required then better performance will be achieved using the asynchronous version, regardless of the underlying implementation. By making the two different styles of primitive explicit the confusion over the semantics of the primitive disappears. It should be noted that the ordering is from a *single agents perspective*, and therefore does not introduce a global state.

Some may consider the inclusion of an `out_sync` may mean that implementation strategies for the run-time systems can not use replication for tuples and tuple spaces. This is not the case, the replication of a tuple occurs and the `out_sync` primitive blocks until all the replications are completed. There are many ways that the replication could be introduced into a run-time system, and it is up to an implementor to ensure that the way chosen ensures this.

3.2.1 All tuples

Within several of the definitions of the WCL primitives the words *all tuples* are used. The understanding of what *all tuples* means is closely linked to the insertion ordering issue and is an issue of predictability. Let us consider the `move_sync` primitive, which moves all matching tuples from one tuple space to another. Figure 5 shows two pieces of code, a Producer and a Consumer. The Producer is split into two parts, with Part 1 inserting a 100 tuples into a tuple space and then places a tuple into the tuple space. Part 2 places another 100 tuples into the tuple space and places another tuple into the tuple space. The Consumer blocks awaiting a tuple that matches the template $\langle|$"DONE"$|\rangle$, and then performs a `move_sync`.

Assuming that no other agents are using the tuple space represented by the handle `ts1`, the value of `x` in the Consumer will be in the range of 100 to 200. This is because communication through a tuple space is *asynchronous*. Part 1 of the Producer and Consumer have *explicitly* synchronised using the tuple

Producer
Part 1

`for (int j = 0; j < 100; j++) {`
`  ts1.out_sync(new Tuple(new Integer(j)));`
`}`
`ts1.out_async(new Tuple(new String("DONE")));`

Part 2

`for (int j = 0; j < 100; j++) {`
`  ts1.out_async(new Tuple(new Integer(j)));`
`}`
`ts1.out_async(new Tuple(new String("DONE2")));`

Consumer
`ts1.in_sync(new Template(new String("DONE")));` `int x = ts1.move_sync(ts2, new Template(new Formal("integer")));`

Figure 5: Example of what 'all tuples' means.

⟨"DONE"⟩. Part 1 of the Producer inserts 100 tuples using the synchronous insertion primitive, therefore, when the synchronisation between the Producer and Consumer occurs, the Consumer can it predict that all the tuples inserted are present. Hence, the value of `x` in the Consumer can never be less than 100. However, because there is no explicit synchronisation between Part 2 of the Producer and the Consumer, the Consumer can not predict *how* many more tuples have been inserted. Therefore, between 0 and 100 could have been inserted, so the value of `x` in the Consumer could lie between 100 and 200.

Subsequently, the issue as to what happens if an `out_sync` is performed in parallel with a `sync_move` primitive is not an issue, the inserted tuple is either included or not included. Because of the nature of the *asynchronous communication between agents* provided by tuple spaces, it makes no difference because the agent can not predict that the tuple will be inserted. This means that the agent performing the primitive must be able to deal with the tuple either being present or not being present. In our experience, it is normal, when using the primitives which manipulate 'all tuples' that the programmers use explicit synchronisation, in order to ensure that the agent can predict that the tuples it requires are present within the tuple space.

It should be noted that even if an explicit synchronisation between the two fragments in Figure 5 was introduced using the ⟨"DONE2"⟩ tuple instead of the ⟨"DONE"⟩ tuple, the value of `x` in Fragment 2 would still be in the range of 100 to 200. This is because the second set of primitives are inserted using the asynchronous insert primitive, as described in the previous section on insertion ordering. It should be noted that the use of the asynchronous insert for the ⟨"DONE"⟩ tuple is fine.

3.3 Bulk primitives

The bulk primitives of WCL: `move_sync`, `copy_sync`, `move_async`, and `copy_async` are based on the synchronous primitives `collect` [Butcher *et al.* 1994] and `copy-collect` [Rowstron and Wood 1996b] proposed for inclusion in Linda. These are called the bulk primitives because they manipulate more than one tuple at a time. These primitive move multiple tuples from one tuple space to another *tuple space*.

The addition of bulk primitives to tuple space based system has been justified as either providing extra performance, often involving less communication than if they were not provided, or overcoming the inability to express certain co-ordination constructs using existing Linda access primitives, for example the *multiple* `rd` *problem* [Rowstron and Wood 1996b].

To demonstrate the multiple `rd` problem, consider a tuple space containing a number of tuples with each containing two fields representing peoples names such as $\langle \text{“Joe”}_{str}, \text{“Bloggs”}_{str} \rangle$. This tuple space is shared among many agents that may require access to the tuples. How would all the tuples representing people whose surname is *Bloggs* be retrieved by an agent?

Initially, the answer would appear to be the repeated use of the `rd_sync` primitive. However, the template $\langle | \Box_{string}, \text{“Bloggs”}_{str} | \rangle$ will only work if there is a single tuple which matches the template. If all the names of an entire family are in the tuple space, or there are several unrelated people with the same surname stored in the tuple space, the repeated use of a `rd_sync` will not work. The semantics of `rd_sync` mean that if more than one tuple matches a template the tuple returned is chosen non-deterministically. The traditional Linda approach is to use semaphore tuples and destructively read all the tuples, process the required ones, and return them all to the tuple space and release the semaphore. However, this is both expensive (in terms of communication and server load) and leads to sequential access of the tuple space. The other approach is to add to each tuple a unique field, and then again check each tuple stored in the tuple space.

As tuple spaces are used in less closed environments, such as the Internet, the likelihood of storing information as tuples in tuple spaces, where each tuple is not unique, is high. For example the storing of names and email addresses in a tuple space. Therefore, the need to perform this efficiently, in terms of communication usage and run-time system load, is important. Indeed, in [Gelernter and Zuck 1997] whilst describing a system built on top of Linda called LifeStreams, they refer indirectly to this problem in a footnote. The LifeStreams application is specifically developed for use over the Internet.

WCL provides support for overcoming the multiple `rd` problem in two different ways, depending on the type of co-ordination required. The first way is using the `copy_sync` and `copy_async` primitives. The alternative way is using the new streaming primitives.

In this section bulk primitives were described which moves tuples from one tuple space to another tuple space. Others have proposed the inclusion of primitives which return tuples to a special local data structure within the agent performing the operation, such as in Bauhaus Linda [Carriero *et al.* 1994] and Objective Linda [Kielmann 1996]. The local data structures within the agent require new access mechanisms. In both Bauhaus Linda and Objective Linda the data structure takes the from of a multi-set, or in other words a tuple space. Therefore, the tuples are moved from a tuple space, to a special tuple space which requires special access mechanisms and operations (such as counting the number of elements in it).

3.4 Streaming primitives

The concept of 'bulk moving' tuples to an agent is attractive for several reasons, which we will describe in this section. WCL provides support for 'streaming' tuples, in the form of the `monitor`, `bulk_in_async`, `bulk_rd_async`, `touch_async` and `touch_sync` primitives. WCL does not provide special local data structures, but uses the mechanisms already required by the primitives that provide the asynchronous access of tuple spaces. Not only does this remove the need for special local data structures but it also has the advantage that at an implementational level the run-time system maintains control of the tuples which are being returned to the agent, and therefore, can transfer the tuples at an appropriate rate. This means that there are no memory management issues for the programmer concerning whether a local data structure can be created in memory to store all the returned tuples – this is taken care of by the run-time system. If this is not managed by the run-time system this is a particular problem for the programmer as the number of tuples being returned is not known, and the memory required to store these tuples is also not known.

To demonstrate the `bulk_rd_async` primitive consider the example in Figure 6. This represents a solution to the multiple `rd` problem, not using the `copy_sync` or `copy_async` primitive. The `move_sync` is used to count the number of matching tuples, and then these are streamed to the user agent, which knows how many to expect. Multiple agents can perform this operation concurrently.

The `monitor` primitive is an interesting primitive, because this primitive is used to create a continuous stream of tuples to an agent. A `monitor` returns all tuples in the tuple space which match the given template (in a similar manner to a `bulk_rd_async`) and any tuples *inserted* into that tuple space. To terminate a `monitor` the `cancel` primitive is used. It should be noted that the `monitor` primitive *returns* a copy of the tuple inserted into the tuple space which matches the template, not just an indication that a matching tuple has been inserted. Hence, the `monitor` primitive provides a stream of tuples back to the agent.

The `monitor` primitive provides extra functionality to WCL as it can be used as a mechanism for a simple form of event distribution within WCL. Eg, if e-mail is being stored in a tuple space, a mail agent

```
Tuple tuple;
tuple = ts1.move_sync(ts1, new Template(new Formal("string"), new String("Bloggs")));
int rqid = ts1.bulk_rd_async(new Template(new Formal("string"), new String("Bloggs")));
int count = ((Integer) Counter.GetField(1)).intValue();
for (int loop = 0; loop < count; loop++) {
  Tuple result = ts1.check_sync(rqid)
  use_tuple(result);
}
```

Figure 6: Using the bulk primitives to overcome the *multiple rd problem.*

can watch for e-mail arriving simply by using the `monitor` primitive, and would be informed whenever a piece of email arrives. In other words the inserted e-mail raises an event. This functionality of the `monitor` primitive is further enhanced by adding the `touch_sync` and `touch_async` primitives. These allow a tuple to be inserted and removed, and subsequently provides a simple mechanism for an agent to raise an event. This extra functionality to WCL that will be demonstrated using the simple example of a tea-timer organiser.

The tea-time organiser is an application that enables a group of people (in this case distributed over several corridors) to organise and co-ordinate their visits to the tea room where they can then make either a cup of tea or coffee. Although the application is not particularly geographically distributed it demonstrates very well the use of the `monitor` primitive to allow agents to monitor each others state stored in a tuple space.

Each person runs a *tea agent* which enables them to signal to other people what their ‘state’ is, and to observe the state of the other people. A person can signal that they are going to the tea room or have returned from the tea room by pressing buttons in the tea agents window. Although this example is simple, it demonstrates well the co-ordination constructs.

A simple way to create the tea-time organiser is to create a shared tuple space into which the tea agents place tuples representing their state. In this simple case the state is either *in the tea room* or *not in the tea room.* Whenever a person starts a tea agent, it inserts a state tuple into the shared tuple space, containing the user’s name and the *not in the tea room* state. When a person indicates to the tea agent that they are going to the tea room, the tea agent removes their state tuple and changes the state to *in the tea room.* The state tuple is changed by destructively removing the tuple and inserting a new state tuple with updated state value. When the tea agent starts it displays a list of all people represented by a state tuple with their current state. As other tea agent change their states, these changes are reflected on the tea agent display.

The state tuples in the shared tuple space at any one time represent the *global state* of *most* of the tea agents, and if these tuples can be examined allows any one tea agent to determine who is using the tea-time organiser. It should be noted that the tuple space does not necessarily contain the entire global state because tuples are removed to be updated. Therefore, if a tea agent is updating their state tuple it will not be in the tuple space.

The implementation of such a scheme using the standard Linda primitives is not possible, but using the `monitor` primitive in WCL it is possible. When a tea agent is started it performs a `monitor` on the shared tuple space. This has the effect of returning all the current state tuples and any ones that are inserted in future. In the description of the tea-time organiser it was noted that it is possible for the shared tuple space not to contain all state tuples as one or more of the tea agents may be updating their state tuple. However, this does not matter because any tea agent updating their state tuple will have to reinsert the state tuple. When this occurs the `monitor` primitive will match the inserted tuple and return it to the appropriate tea agent. When a tea agent receives the tuple it is able to check the persons name locally and discover whether the tuple represents the state of as yet unseen person or if it is an update of an existing person's state. The tea agent can then update it's display accordingly. Therefore, the fact that all the state tuples are not present in the tuple space all the time is not important.

To enable a tea agent to terminate a third possible state that a tea agent can be in is added; *leaving*. When a person indicates to the tea agent that they wish to leave, it removes their state tuple and then uses the `touch_async` primitive to insert a new state tuple, with the state set to *leaving*, then performs a `cancel` for the `monitor`, and terminates. The `touch_async` primitive inserts the new state tuple, and then removes it. This means that any tea agents monitoring the shared tuple space will receive a copy of the inserted tuple, but the tuple will not reside in the tuple space. When a tea agent receives a tuple with the state *leaving* it removes the associated person from it's display.

There are other approaches to managing the interaction between the tea agents, but these require the state tuples to be stored within complex structures within the tuple space. One such approach would be to create an ordered queue of state tuples. Tea agents would add their new state to the end of queue. However, these tuple structures are larger and more complex to manage than the method described. Also, in a closed implementation the control of the tuple structures is not such a problem, because the run-time system will terminate. However, in the open implementations envisaged for use over the Internet, the run-time system will not terminate often (if ever). Therefore, a tuple space may exist for years, not hours or days, and the control of the tuples stored within it is important, and often complex.

The tea-time organiser has demonstrated how the `monitor`, `touch_async` and `touch_sync` primitives in WCL can be used, which enabled multiple agents to easily create and monitor a global state. This

co-ordination construct saves the use of complex tuple structures to be created and managed within tuple spaces to achieve a similar effect.

4 WCL VERSUS JAVASPACE

JavaSpace [Sun Microsystems 1997] is a draft specification for a tuple space based co-ordination language. Unlike WCL, there is apparently is no run-time system that supports JavaSpaces. The JavaSpace specification distances itself from Linda, but fundamentally uses many similar concepts, including tuple spaces and similar access primitives. The specification describes four basic operations that can be performed on tuple spaces. A *non-blocking* `read` (a Linda `rdp`) and a *non-blocking* `take` operation (a Linda `inp`), a write operation (a Linda `out`) and a `notify` operation. We will now contrast some of the design decisions taken in JavaSpaces with those taken in WCL.

One of the problems of including access primitives that are *synchronous* and non-blocking is shown in the fragment of Java code in Figure 7. The `take` operation takes a template provided as an EntryRep object (the other parameters provide support for atomic transactions and identity tagging). If no matching tuple is found the `take` primitive returns `null`. The `while` loop will cause the code to block until a tuple matching the template encoded within `erep` is found. The problem with the code fragment is that it polls the run-time system continuously. This dramatically increases the traffic passed over the Internet, and places a *considerable load* on the server storing the tuple space. Every time a `take` is performed a message is sent to the server. The server must decode and interpret the incoming message, then search the tuple space and finally return a message to the agent executing the operation. All this will then be immediately repeated. Even if blocking primitives are added to JavaSpace [Arnold 1997], experience has shown that it is not easy to stop programmers writing code using a particular style of co-ordination if it can be done.

```
JavaSpace space = getSpace();
EntryRep erep = new EntryRep(demo);
while (space.take(demo,null,null) == null);
```

Figure 7: Polling in JavaSpaces.

An equivalent piece of code can not be written using the WCL primitives. Having seen how non-blocking primitives can effect the performance of LAN based tuple space system, the WCL primitives were designed not to allow explicit server polling. However, in WCL a form of polling can be performed, using the `in_async` and `check_async` primitives. However, the communication to the server is restricted to one message to perform the `in_async` and then the `check_async` is a *local* operation which checks to see if the tuple has been returned from the tuple server. This provides the functionality required, and reduces the

communications and server load.

The JavaSpace specification states that they will provide some sort of atomic transaction capabilities within JavaSpaces. Atomic transactions are used in both PLinda [Anderson and Shasha 1991] and Paradise [Scientific Computing Associates 1996], which are fault-tolerant LAN based Linda implementations. The use of transactions means that a number of tuple space access primitives are grouped together, and either all the operations are performed or none of the operations are performed. This means that the run-time system must store the tuples it has dispatched to you, and store the tuples that you insert. If the transaction fails, the removed tuples are reinserted and inserted tuples are thrown away. Having examined transactions we decided that they are not well suited in their current form for geographically distributed computing. This is because the server needs to attach timeouts to the transactions to decide when they have failed, and because we feel there is a better solution to the problem, which is described in Section 6.

JavaSpace provides the `notify` primitive which supports basic event distribution, and is very different from the `monitor` primitive in WCL. The `notify` primitive *does not* return tuples that are already in the tuple space which match the template and the `notify` primitive *does not return* the matching tuples inserted. It simply, returns a message to say that a matching tuple has been inserted. These have several side effects, one is that the `notify` primitive can not be used in the same way as the `monitor` primitive in the tea-time organiser example. Secondly, an agent can be notified that a tuple has been inserted, but it does not guarantee that the agent can retrieve that tuple. The tuple may either have been destructively removed by another agent already, or there may be multiple tuples in the tuple space which match the template. When there are multiple tuples that match the template the `read` operation will pick one non-deterministically, not necessarily the one which was inserted and generated the message back to the agent. This is the *multiple* `rd` *problem* which can not be overcome using the primitives provided in the JavaSpace specification.

5 RUN-TIME SYSTEM

WCL was first implemented as an extension to the Cambridge Collaborative Agent System (C^2AS) [Rowstron *et al.* 1997], and this was used as a platform to evaluate the expressiveness of WCL. In the same way that there are many ways of implementing a Linda run-time system, there are many ways in which a WCL run-time system can be implemented. We now consider a novel run-time system architecture, which from the beginning has been designed with the goals of exploiting data location transparency, and to support geographically distributed computing using the WCL primitives. The current implementation is a prototype system, aimed at investigating some techniques that such kernels will use, in particular the migration of tuple spaces for efficiency.

The entire run-time system is called the *kernel*. The kernel is composed of three distinct sections; the control system, the tuple management system and the agent libraries. The tuple management system uses multiple tuple servers. A tuple server is designed for efficient data access, receiving tuples, requests for tuples and instructions to migrate tuple spaces. A separate layer, the control system, determines when a tuple space should be migrated, and dispatches the instructions to the individual tuple servers. The separation is made so that the tuple servers can be as efficient as possible. It should be noted that the control layer and tuple servers can reside on different machines, anywhere on the Internet. Indeed, it is advantageous not to run a tuple server on the same machine as the control layer.

5.1 Tuple management system

The tuple servers are distributed geographically around the area being supported. Each tuple server manages many complete tuple spaces. To use the analogy of Douglas [Douglas *et al.* 1995b] 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. In such a scheme each tuple server manages a small part of many tuple spaces. In our system the each tuple server manages one *or more* layers of the cake, in other words 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 synchronisation between different tuple servers.

5.2 Control system

The control system acts as a manager, with the tuple servers registering with it, and it also presents agents with their entry point to the kernel. Currently, the control system uses a well known Internet address and port. When an agent connects to the control system it tells the control system its' geographical location, and is passed a handle for a well known tuple space, called the *Global Tuple Space*. This registration is transparent to the user and to the programmer. However, the user has to supply the well known Internet address and the geographical position of the agent.

When a tuple server registers, it tells the control system information about its geographical position, power and the area which it would like to cover. The control system supports dynamic agent registration and deregistration. Tuple servers can dynamically register, but currently in the prototype system they can not terminate. It should be noted that an agent can re-register to provide different positional information. This is important if migrating agents are to be supported.

The control system generates tuple space names (handles), monitors tuple spaces and decides when

they should migrate. 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 centralised[3)], the kernel has been designed to allow the control system to 'fall behind', thus overcoming the problem of it becoming a bottleneck.

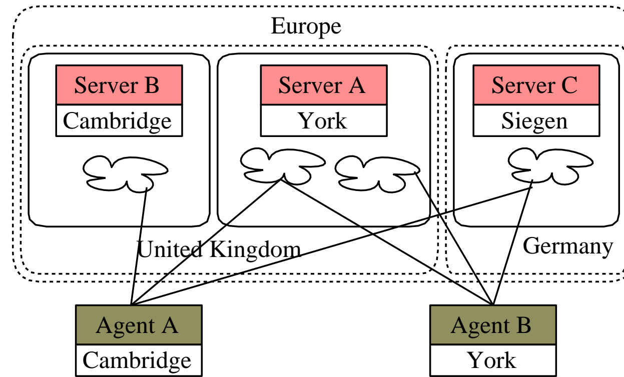

Figure 8: Information stored in control system.

Figure 8 shows the information stored and managed in the control system. It shows three tuple servers, one in Siegen (Germany), one in Cambridge (UK) and one in York (UK). 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 optimise the performance of the system by moving the tuple spaces to the best position within the entire kernel. Therefore, in Figure 8 the control system analyses the information and observes that the two agents using the tuple space stored in Siegen are in fact in the United Kingdom. Therefore, it migrates the tuple space. If both the agents were at the same geographical location then the tuple space would be migrated there. In this case either tuple server would be just as good, therefore, the control system looks at the current loading and capability of each of the tuple servers and makes a decision as to which the tuple space is migrated. It should be noted, that if more agents joined, then the tuple space may migrate again.

5.3 Agent library

The agent library is the part of the kernel that is embedded into the agent. These routines manage all interaction with the other parts of the kernel, provide local tuple storage, and determine where to find tuple

[3)] We are currently developing a decentralised version of the control system.

spaces. All of this functionality is transparent to the programmer, who simply uses the WCL primitives.

5.4 Tuple space migration in the kernel

Since tuple spaces migrate around the system, how does an agent know where to find a particular tuple space? An agent uses a handle to refer to a particular tuple space. This handle contains a unique tag, plus information about which tuple server is storing the tuple space. When an agent wishes to access a tuple space it looks at this information and if a communication channel is not already present to the appropriate tuple server it creates a communications channel and sends the requests to that tuple server. If that tuple server still maintains the tuple space then the operations are performed there. However, if the tuple space has migrated, then the tuple server will return a new tuple space handle to the agent, containing more up-to-date information. The agent stores this and uses this new tuple space handle whenever it is presented with the superseded tuple space handle.

The current kernel is scalable and can 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 go to that tuple server. However, if several 'popular' tuple spaces are stored on a single tuple server, one or more of them will be moved to other less busy tuple servers. This kernel supports more agents, and provides better access, over the whole system, than existing LAN based tuple space implementations for Internet based co-ordination.

5.4.1 Experimental results

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 PC workstation, at Siegen on a DEC Alpha Workstation and at York on a SGI Indy workstation.

The results shown in Figure 9 show the time taken to insert and retrieve 2500 tuples from a tuple space, where the average size of a tuple was 100 bytes. The columns represent where the agent performing the insertion or removal was located. The rows represent where the tuple space was being stored. In many of the retrieval timings two times are given, the top timing is when the tuple space is not migrated and the lower timing is when the tuple space is migrated. It should be noted that the retrieval timings include the time taken to register with the system, this ensures that the times represent the time to move the tuple space as well as retrieve the tuples. 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 should have been

low. It should also be noted that the insertion timings are for synchronous insertion.

	2500 Insertions			2500 Retrievals		
	York	Cambridge	Siegen	York	Cambridge	Siegen
York	6.79	60.44	114.55	11.35	56.31	115.56
					2.94	**9.73**
Cambridge	63.11	1.50	106.16	62.64	2.32	112.91
				11.65		**8.40**
Siegen	108.21	110.23	4.51	120.22	112.23	7.57
				11.86	**2.77**	

Figure 9: Insertion and retrieval timings.

These results demonstrate that the migration of tuple spaces within the kernel produces a large performance increase, under some circumstances. 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 to a remote site.

5.5 Host Language Integration

As already stated, the WCL primitives are embedded into a host Language. Currently, they have been embedded into C++ and Java. The syntax of the Java embedding can be seen in Appendix A, Tables 3 and 4. The Java embedding provides support for both stand alone Java programs and Applets. The syntax of the C++ embedding can be seen in Appendix A, Table 5.

Agents written in C++ can communicate with agents written in Java. Tuples created by a Java agent can be read by a C++ agent which means that there must be some sort of mapping between types in the two languages. In the current implementation the following Java objects are supported: Integer, Character, String, Double and TupleSpace. In C++ the following types are supported: int, char, char *, double and TupleSpaceHandle. Each of the host language libraries encodes the different types in to a standard format, therefore, a char * in C++ is will be translated into the tuple encoding, and then a String object will be created when a Java agent reads it in. If a Java TupleSpace object is placed in a field of a tuple then the tuple space handle is placed in the tuple, not the object itself. It should be noted that tuple space handles can be passed between agents written in both languages through tuples in tuple spaces. Therefore, an agent written in Java can create a tuple space and pass it to an agent written in C++.

There is currently no support for the placing of actual Java objects into a tuple. However, given

object serialisation operations in Java, it should be possible to support this. We have two concerns about this though; tuples containing representations of Java objects can only be read by agents written in Java. Secondly, given that this is an open implementation and multiple people will be using it, we need to ensure uniqueness of fields. For example, if two independent people create two independent classes called Set and then insert tuples containing objects of this class into the same tuple space, how do we ensure that they each retrieve tuples with the correct Set object? The specific matching of Linda with Java is considered in detail in [Ciancarini and Rossi 1997].

There is no explicit support for the creation of agents within WCL, or within the implementation. An agent has to be explicitly started, but they can run anywhere in the world. Therefore, an agent could be started by downloading an Java applet over the Internet, or by explicitly executing an agent on a particular machine.

6 FUTURE WORK

There are two main areas where WCL and the run-time system need further work; security and fault-tolerance. Security within WCL will be introduced by expanding the underlying tuple space model, as done in Paradise [Scientific Computing Associates 1996]. This will involve the addition of read-write controls on tuple spaces, and potentially tuples. Security at an implementation level will be provided by encoding the packets passing between the kernel and the agents, when requested. It is possible to envisage that a tuple space can be created as 'secure' which means all operations to and from it will be encoded.

Currently we are working on adding fault tolerance to the tuple servers, using snapshots and insertion and removal logs. The current run-time system can cope with agents dying unexpectedly, but when an agent dies we currently can not make consistent any 'global' state that the agent had removed or was updating. For example, if there was a tuple being used as a counter between many agents, and one removes it and dies without reinserting the counter tuple, no other agents will ever be able to access the counter tuple, and the application that these agents were part of will probably deadlock. As already mentioned, some fault-tolerant LAN implementations use atomic transactions to overcome this problem. We are currently experimenting with the use of 'mobile co-ordination'. The actions that need to be performed atomically on a tuple space are migrated to the tuple servers and then executed there. We provide a very simple and limited computational model to allow small amounts of computation to be performed in the migrated co-ordination. Initial work has shown that this approach results in less bandwidth usage, less latency and *less* computation on the tuple servers. It also has the advantage that using the same mechanism we can provide a 'will' to a tuple space, which is executed only if the agent disconnects from a tuple space unexpectedly. For example, consider the

tea-time organiser example, we could have provided a 'will' that said: remove my state tuple, touch with a state tuple containing *leaving*. If the agent dies, the tuple space is disconnected from unexpectedly, so the 'will' is executed and the other tea-agents know that the tea agent has terminated. This work is still in its infancy, but it shows great hope of being very effective.

Another issue, given that kernel will execute potentially for a great length of time, is garbage collection of tuple spaces. We are not intending to explicitly look at garbage collection, however, some garbage collection schemes already exist for tuple space based system [Menezes and Wood 1997].

7 CONCLUSIONS

WCL is a tuple space based co-ordination language. It has been designed to support co-ordination in truly distributed computing environments, and to be independent of the host language used. The current run-time system has been briefly described, and uses a novel approach to supporting tuple spaces for use over the Internet.

WCL is a lower level co-ordination language compared to Linda. This is due to the fact that the Linda primitives are not good for use in high latency networks, and that they do not support all the co-ordination constructs that are required for a more open co-ordination style. Is WCL too complex? We have had many users of WCL start to write usable programs within minutes of being introduced to WCL. The underlying concepts are easy, and in our opinion compared to the complexities of systems such as CORBA and Java's RMI, WCL can not be considered as too complex.

WCL has been used in a number of applications. Many of the applications use a peer-to-peer paradigm rather than a client-server paradigm. For example, it is possible to create a talk tool that uses no servers, and the text is still available even if all the talk tools are terminated, and restarted because of the persistence of tuple spaces. Also, these applications have demonstrated the flexibility of WCL in supporting applications where the number of participating agents change dynamically.

Even though we have working systems and have been able to evaluate the functionality of WCL, a number of areas remain where significant work is still required; primarily on the security aspects and on the run-time system.

8 ACKNOWLEDGEMENTS

I would like to thank Dr. Andy Hopper and the Olivetti and Oracle Research Laboratory, Cambridge, UK for financially supporting this work. I would like to thank Dr. Stuart Wray for his help and advice on

improving this paper, the reviewers of this paper, the previous BONITA paper and the C^{2}AS paper. I would also like to thank Andrew Douglas for his help and advice over the last years.

REFERENCES

Anderson, B. and D. Shasha (1991), "Persistent Linda: Linda + Transactions + Query Processing," In *Research Directions in High-Level Parallel Programming Languages*, J. Banâtre and D. Le Métayer, Eds., LNCS 574, Springer Verlag, pp. 93–109.

Arnold, K. (1997), "Private Communication," .

Bosschere, K. D. and L. Wulteputte (1991), "Mutli-Prolog: Implementation on an 88000 shared memory multiprocessor," Technical Report DG 91-19, University of Ghent, Belgium.

Busi, N., R. Gorrieri, and G. Zavattoro (1997), "Three semantics of the output operation for generative communication," In *Coordination Languages and Models (Coordination'97)*, D. Garlan and D. Le Métayer, Eds., LNCS 1282, Springer-Verlag, pp. 205–219.

Butcher, P., A. Wood, and M. Atkins (1994), "Global Synchronisation in Linda," *Concurrency: Practice and Experience 6*, 6, 505–516.

Carriero, N. (1987), "Implementation of Tuple Space Machines," Ph.D. thesis, Yale University, New Haven, CT, USA, YALEU/DCS /RR-567.

Carriero, N. and D. Gelernter (1990), *How to write parallel programs: A first course*, MIT Press.

Carriero, N., D. Gelernter, and L. Zuck (1994), "Bauhaus Linda," In *Object-Based Models and Languages for Concurrent Systems*, P. Ciancarini, O. Nierstrasz, and A. Yonezawa, Eds., LNCS 924, Springer-Verlag, pp. 66–76.

Ciancarini, P. (1991), "PoliS: a programming model for multiple tuple spaces," In *Proceedings of the sixth International Workshop on Sofware Specification and Design*, pp. 44–51.

Ciancarini, P., A. Knoche, R. Tolksdorf, and F. Vitali (1996), "PageSpace : An Architecture to Coordinate Distributed Applicat ions on the Web," *Computer Networks and ISDN Systems 28*, 7-11, 941–952.

Ciancarini, P. and D. Rossi (1997), "Jada: Coordination and Communication for Java agents," In *Mobile Object Systems: Towards the Programmable Internet*, J. Vitek and C. Tschudin, Eds., LNCS 1222, Springer-Verlag, pp. 213–228.

Douglas, A., N. Röjemo, C. Runciman, and A. Wood (1996), "Astro-Gofer: Parallel Functional Programming with co-ordinating processes," In *Euro-Par'96*, L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, Eds., LNCS 1123, Springer-Verlag, pp. 686–693.

Douglas, A., A. Rowstron, and A. Wood (1995a), "ISETL-LINDA: Parallel Programming with Bags," Technical Report YCS 257, University of York, York, UK.

Douglas, A., A. Wood, and A. Rowstron (1995b), "Linda implementation revisited," In *Transputer and occam developments*, P. Nixon, Ed., Transputer and occam Engineering Series, IOS Press, pp. 125–138.

Gelernter, D. (1992), *Mirror worlds*, Oxford University Press.

Gelernter, D. and N. Carriero (1992), "Coordination Languages and their Significance," *Communications of the ACM 35*, 2, 97–107.

Gelernter, D. and L. Zuck (1997), "On what Linda is: Formal description of Linda as a Reactive System," In *Coordination Languages and Models (Coordination'97)*, D. Garlan and D. Le Métayer, Eds., LNCS 1282, Springer-Verlag, pp. 187–204.

Hasselbring, W. (1994), "Prototyping Parallel Algorithms in a set-orientated language," Ph.D. thesis, University of Essen, Germany.

Kielmann, T. (1996), "Designing a coordination model for open systems," In *Coordination Languages and Models, Proceedings of Coordination '96*, P. Ciancarini and C. Hankin, Eds., LNCS 1061, Springer-Verlag, pp. 267–284.

Leichter, J. (1989), "Shared tuple memories, shared memories, buses and LAN's – Linda implementations across the spectrum of connectivity," Ph.D. thesis, Yale University, New Haven, CT, USA, YALEU/DCS/TR-714.

Menezes, R. and A. Wood (1997), "Garbage Collection in Open Distributed Tuple Space Systems," In *Proceedings of 15th Brazilian Computer Networks Symposium - SBRC'97*, pp. 525–543.

Rowstron, A., A. Douglas, and A. Wood (1995), "A distributed Linda-like kernel for PVM," In *EuroPVM'95*, J. Dongarra, M. Gengler, B. Tourancheau, and X. Vigouroux, Eds., Hermes, pp. 107–112.

Rowstron, A., S.Li, and R. Stefanova (1997), "C^2AS: A System Supporting Distributed Web Applications Composed of Collaborating Agents," In *Proceedings of Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, IEEE CS Press, pp. 127–132.

Rowstron, A. and A. Wood (1996a), "An efficient distributed tuple space implementation for networks of workstations," In *Euro-Par'96*, LNCS 1123, Springer-Verlag, pp. 510–513.

Rowstron, A. and A. Wood (1996b), "Solving the Linda multiple `rd` problem," In *Coordination Languages and Models, Proceedings of Coordination '96*, P. Ciancarini and C. Hankin, Eds., LNCS 1061, Springer-Verlag, pp. 357–367.

Rowstron, A. and A. Wood (1997), "Bonita: A set of tuple space primitives for distributed coordination," In *Proceedings of 30th Hawaii International Conference on System Sciences*, volume 1 - Software Technology and Software, IEEE CS Press, pp. 379–388.

Scientific Computing Associates (1995), *Linda: User's guide and reference manual*, Scientific Computing Associates, New Haven, CT, USA.

Scientific Computing Associates (1996), *Paradise: User's guide and reference manual*, Scientific Computing Associates, New Haven, CT, USA.

Sun Microsystems (1997), "JavaSpace(tm) Specification, Revision 0.4," http://chatsubo.javasoft.com/.

Yuen, C., M. Feng, and J. Yee (1996), "BaLinda suite of languages and implementations," *Journal of Software Systems 32*, 251–267.

Appendix A

Tables 3 and 4 show the class descriptions for the Java embedding of WCL, and Table 5 shows the include file for the C++ embedding of WCL. These provide an insight into the syntax of each embedding.

Table 3: Java Tuple, Template and Formal Class Definition

```
public class Tuple {
  // Public Constructors
    public Tuple();
    public Tuple(Object a);
    public Tuple(Object a, Object b);
    public Tuple(Object a, Object b, Object c);
    public Tuple(Object a, Object b, Object c, Object d);
    public Tuple(Object a, Object b, Object c, Object d, Object e);
    public Tuple(Object a, Object b, Object c, Object d, Object e, Object f);
  // Public Instance Methods
    public int TupleLength();
    public Object GetField(int n);
}
```

```
public class Template {
  // Public Constructors
    public Template();
    public Template(Object a);
    public Template(Object a, Object b);
    public Template(Object a, Object b, Object c);
    public Template(Object a, Object b, Object c, Object d);
    public Template(Object a, Object b, Object c, Object d, Object e);
    public Template(Object a, Object b, Object c, Object d, Object e, Object f);
}
```

```
public class Formal {
  // Public Constructors
    public Formal(String name);
}
```

Table 4: Java TupleSpace Class Definition

```
public class TupleSpace {
  // Public Constructors
    public TupleSpace(String TS);
    public TupleSpace();
  // Public Instance Methods
    public synchronized void InitWCL();
    public void MakeGTS();
    public synchronized void createts();
    public void out_async(Tuple t);
    public void out_sync(Tuple t);
    public void touch_async(Tuple t);
    public void touch_sync(Tuple t);
    public Tuple in_sync(Template t);
    public int in_async(Template t);
    public int bulk_in_async(Template t);
    public int bulk_rd_async(Template t);
    public Tuple rd_sync(Template t);
    public Tuple move_sync(TupleSpace ts2, Template t);
    public int move_async(TupleSpace ts2, Template t);
    public Tuple copy_sync(TupleSpace ts2, Template t);
    public int copy_async(TupleSpace ts2, Template t);
    public int rd_async(Template t);
    public int monitor(Template t);
    public void cancel(int rqid_r);
    public Tuple check_sync(int rqid_r);
    public Tuple check_async(int rqid_r);
    public void close();
}
```

Table 5: C++ Library Header file

```
class TupleSpaceHandle {
  public:
    ~TupleSpaceHandle();
    TupleSpaceHandle(void);
    char *GetTupleSpace(void);
    void SetTupleSpace(char *);
  private:
    char *TS;
};
```

```
class RequestIdentifier{
  public:
    ~RequestIdentifier();
    RequestIdentifier(TupleSpaceHandle *, int, char);
    int GetReqIdTag(void);
    TupleSpaceHandle *GetReqIdTS(void);
  private:
    TupleSpaceHandle *TS;
    int Tag;
};
```

```
void wcl_init(int, char *[]);
void wcl_exit();
TupleSpaceHandle *createts(void);
void out_sync(TupleSpaceHandle *, ...);
void in_sync(TupleSpaceHandle *, ...);
void rd_sync(TupleSpaceHandle *, ...);
void out_async(TupleSpaceHandle *, ...);
RequestIdentifier *in_async(TupleSpaceHandle *, ...);
RequestIdentifier *rd_async(TupleSpaceHandle *, ...);
void touch_sync(TupleSpaceHandle *, ...);
void touch_async(TupleSpaceHandle *, ...);
void cancel(RequestIdentifier *);
void check_sync(RequestIdentifier *, ...);
int check_async(RequestIdentifier *, ...);
void check_sync(RequestIdentifier *, int *);
int check_async(RequestIdentifier *, int *);
int move_sync(TupleSpaceHandle *, TupleSpaceHandle *,...);
RequestIdentifier *move_async(TupleSpaceHandle *, TupleSpaceHandle *,...);
int copy_sync(TupleSpaceHandle *, TupleSpaceHandle *,...);
RequestIdentifier *copy_async(TupleSpaceHandle *, TupleSpaceHandle *,...);
RequestIdentifier *bulk_in_async(TupleSpaceHandle *, ...);
RequestIdentifier *bulk_rd_async(TupleSpaceHandle *, ...);
RequestIdentifier *monitor(TupleSpaceHandle *, ...);
TupleSpaceHandle *GTS = (TupleSpaceHandle *) NULL;
```