Ligia: Incorporating Garbage Collection in a Java based Linda-like Run-time System

Ronaldo Menezes*
University of York
Department. of Computer Science
Heslington, York, YO10 5DD, England
ronaldo@minster.york.ac.uk

Abstract

Generative distributed open coordination systems have been so far implemented in a variety of ways. Surprisingly, no published implementation appears to address one particularly important issue for any general purpose open system — garbage collection.

This paper describes an open LINDA-like implementation called LIGIA using Java, based on communication via sockets and which includes garbage collection of tuple spaces and agents. It is also demonstrated how the garbage collection described by us in [8] is implemented and that it adds little overhead to the overall performance of the system.

Keywords: Coordination Systems, Garbage Collection, Linda, Java.

Sumário

Sistemas para coordenação de processos em ambientes distribuídos baseados em comunicação generativa têm sido implementados das mais variadas formas. Surpreendentemente, nenhuma das implementações disponíveis na literatura mostra interesse em abordar um dos pontos mais importantes no desenvolvimento de sistemas abertos: coleta de lixo.

Este artigo descreve uma implementação em Java de um modelo baseado em LINDA chamado LIGIA que inclui colleta de lixo de espaços de tuplas e agentes. O artigo também demostra como o método de coleta de lixo proposto em [8] é implementado e que este impõe um pequeno custo adicional associado ao sistema.

Palavras-Chave: Coordenação de Processos Distribuídos, Coleta de Lixo, Linda, Java.

1 Introduction

Garbage collection has long been known to be necessary in any dynamic system. We have shown that this is also the case in open LINDA-like implementations and demonstrated theoretically how it should be done [8]. However, no implementation thus far has paid the necessary attention to the issue by incorporating garbage collection capabilities.

LIGIA is a generative open system implemented *in* Java and *for* Java. It is implemented *in* Java because the communication server is implemented only using Java features and it is *for* Java because only Java classes are provided and therefore the agents (distributed processes) of the system must be Java agents. However this does not mean that the system cannot be expanded to work with agents written in other languages. In fact, the choice of using sockets in all communication was made in order to allow any agent in accordance with LIGIA's communication protocol to communicate with the servers; this is explained in more detail in Section 3.

A single server version of LIGIA is used in this paper. Garbage collection has been carefully investigated and implemented within the server as proposed by us in [8]. The goal here is to show that the garbage collector can work efficiently within the server and that the overhead added by the garbage collector is very little compared with the benefits of its implementation.

*Research supported by CAPES, Brazil, under grant 1383/95-8.

As a consequence of the garbage collection method implemented, servers become aware of the existence of agents. In other terms, servers can identify which agent is communicating with them at any time. All the decisions made by the garbage collector are based on the assumption of the agent identifiability, that is, the server can identify the agent which has generated any message.

One should note that this paper does *not* intend to propose a novel method of garbage collection or discuss the advantages of certain garbage collection methods over others. The goal and the novelty of this paper is to describe a LINDA-like implementation that tries to be more realistic by implementing *a* garbage collection method as this is essential in *any* open system but have been so far overlooked by other published implementations.

2 Background

2.1 Ligia: The Model

LIGIA is a generative distributed open coordination model based on LINDA [5, 11]. Generative due to its communication approach, distributed because agents can be distributed over the network, open because agents can join and leave the system at will and finally coordination model because it coordinates communication and synchronization of processes.

The generative communication occurs via tuple spaces. The communication is completely asynchronous — an agent *A1* that wants to communicate with another, say *A2*, has to write the message (a tuple) in a tuple space so that the other process can remove (or read) the message. The removing or reading of tuples is done using non-deterministic associative matching.

The model therefore provides primitives to add a tuple to a tuple space and to remove or read a tuple from a tuple space. There are also bulk primitives which move tuples or copy them from a tuple space to another [11]. The reading or removing of a tuple from a tuple space is based on a template which is used in the associative matching. The semantics of the primitives implemented in LIGIA are described in details in Section 3.2.

2.2 Garbage Collection

The garbage collection implemented is a mark-and-sweep method based on Dijkstra on-the-fly proposal [2] and is similar to Hudak and Keller algorithm [6] having the mark phase divided into two separate phases. As cycles are common in the structure proposed and due to the open nature of the system, mark-and-sweep poses as a good option.

First marking phase: After the graph has been initialised by marking all cells (representing LIGIA objects — tuple spaces and agents) as garbage, a traversal is initiated from the roots, all cells reached are marked as active (non garbage).

Second marking phase: A second phase is necessary as the nature of the graph used (as explained later) based on direct and indirect arc, makes possible that objects considered garbage by the first phase are in truth active. This is possible as active objects (mutators) can modify the graph such that something that was unreachable in the first phase becomes reachable in future traversals. This second phase then have the goal to prevent this erroneous situation. At the end of the first phase a traversal in the graph is done but this time starting from each agent node (active objects). The cells are marked as *probably active*, if the traversal reaches one of the roots, all cells marked as *probably active* by this traversal are marked as active (non garbage). On the other hand, if the traversal fails to reach one of the roots the cells marked as *probably active* by this traversal are marked as garbage.

Figure 1 depicts all four possible situations of a given object after the two marking phases. The only case where the object can be garbage collected by the sweep phase is when both marking phases fail to mark the cell (bottom right corner). If an object is considered passive, it cannot be collected. Passive object has to be a tuple space as LINDA does not have to concept of a passive agent. A tuple space is considered passive

WAS OBJECT...		...MARKED IN THE 1st PHASE?	
		YES	NO
...MARKED IN THE 2nd PHASE?	YES	ACTIVE OBJECT	ACTIVE OBJECT
	NO	PASSIVE OBJECT	GARBAGE OBJECT

Figure 1: Table showing the four possible situations of a cell after the two marking phases (based on a similar table shown in [7])

if there is no agent directly accessing it but it is linked indirectly to one or more processes, by having its handle in *UTS* for instance.

Having said this, in order to do garbage collection in Ligia, one has to show how to build a graph with the information about the usefulness of Ligia objects (tuple spaces and agents) for the system. We have shown how the necessary information for the collector to decide what is garbage is maintained within the server in a form of a hybrid graph [8] and this paper utilises this proposal.

3 Implementation Overview

3.1 Implementation Architecture

Ligia architecture is simple. The implementation uses distributed central servers as its distribution strategy [1]. Each server is multi-threaded. A new thread is created for each new request done by an agent.

From the point of view of the architecture, agents can be either single or multi-threaded as for the servers this makes no difference. In the implementation used in this paper, agents are multi-threaded. Threads are created only in some cases and the main thread of control can also communicate directly with the server. Figure 2 depicts Ligia architecture.

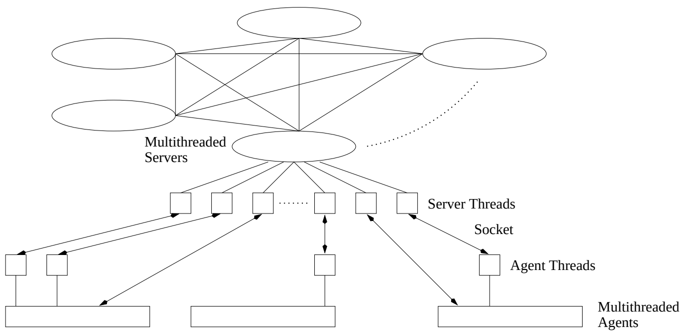


Figure 2: Structure of the Ligia system.

Although not represented in Figure 2, agents can be implemented in another language. Due to the fact that a Ligia server expects to receive objects in the sockets, agents using different languages have to either simulate the format of a serialised object (object passing through a socket) [12] or communicate with a Java translator which would mount the data in the expected format. This translator would receive data in some format, create the objects and send them serialised to the server via sockets.

JavaSpace [13] could be pointed as being similar to Ligia. However, it should be noticed that Ligia differs from JavaSpace in some important aspects:

- JavaSpace implements only non-blocking primitives whereas Ligia implements blocking primitives and uses bulk primitives as an alternative for the use of *inp* and *rdp* as the semantics of them are not precise [11].
- Ligia implements the primitive *eval* whereas JavaSpace does not.

- LIGIA implements garbage collection of tuple spaces and agents.

A more detailed comparison of the two implementations shall not be made here for it is not necessary in the context of this paper. The description given above is the full architecture of LIGIA. For this paper, a smaller version is assumed where only one central server is implemented and all the agents are Java agents.

3.2 Semantics of the Primitives

The primitives in LIGIA are implemented as methods of a class called `ConnectionRequest`. When an object of this class is created a new connection is requested to the server. When accepted a unique name is assigned to the connection. Normally there would be one connection between an agent and a server, since a second would be redundant. However there is nothing which stops an agent requesting more than one connection but from the point of view of the server (and of the garbage collector) each connection is seen as a different agent. In order for the garbage collector to collect all the existing garbage at any time, if two connections are requested, two disconnect requests will have to be made (see section 3.4). The primitives are as follows:

`Handle` *`tsc()`* : A tuple space is created in the server and a unique `Handle` object is returned. Handles instead of tuple spaces are first-class objects.

`void out(``Handle` *`h`*`,Tuple` *`tup)`*: This method takes a `Tuple` and stores it within in a tuple space whose handle is `h`.

`Tuple` *`in(`*`Handle` *`h`*`,Tuple` *`temp)`*: This method looks for a `Tuple` object stored within a tuple space whose handle is *`h`* matching the template *`temp`*. If found the tuple is removed from the tuple space and returned. This is a blocking method, that is, if no `Tuple` object is found in the tuple space the method awaits until one appears.

`Tuple` *`rd(`*`Handle` *`h`*`,Tuple` *`temp)`*: This method is the non-destructive version of the *`in`* method. If found, the tuple is read instead of removed.

`int collect(``Handle` *`h1`*`,Handle` *`h2`*`,Tuple` *`temp)`*: This method moves *all* objects of the class `Tuple` matching *`temp`* from the tuple space whose handle is *`h1`* to the tuple space whose handle is *`h2`*. The number of tuples moved is returned.

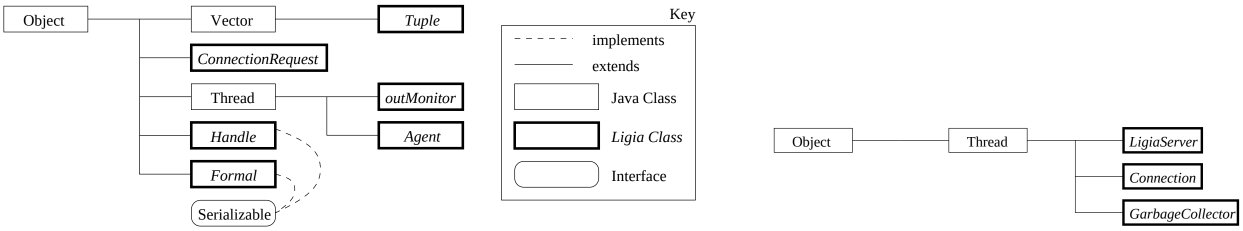
`int copy(``Handle` *`h1`*`,Handle` *`h2`*`,Tuple` *`temp)`*: This method is similar to *`collect`* but the tuples are copied instead of moved.

Besides the methods above, LIGIA also provides a method for spawning agents called *`eval`*. In LIGIA unlike LINDA, a spawned agent does not turn in a tuple within a tuple space. A spawned agent behaves exactly like a normal agent. For the context of this paper *`eval`* can be done in either ways without affecting the behaviour of the garbage collector, therefore the discussion about how *`eval`* is executed is unnecessary.

3.3 Ligia's Java Classes Description

The implementation of LIGIA using Java resulted in the creation of a series of classes and methods which include LIGIA capabilities to Java. The two most important classes are `ConnectionRequest` and `LigiaServer`. The `ConnectionRequest` class allows agents to communicate with servers. It provides agents with the primitive methods to create and access tuple spaces (see Section 3.2). Besides those methods, a method called *`leave`* necessary for the garbage collector is provided, the primitive *`leave`* communicates with the server which from there on will not accept any request from that agent. From the server's point of view that agent has terminated. The `ConnectionRequest` class together with the classes `Tuple` ,`Handle`, `Formal` and `outMonitor` form the set of classes necessary for a Java agent to coordinate via LIGIA.

From these classes it is assumed that only `outMonitor` needs further explanation as the others implement well known concepts in LINDA. Douglas *et al.* [3] have identified and solved a problem, named *`out`*-ordering. This problem consists in guaranteeing that tuples arrive in a tuple space at same order they leave an agent. The class `outMonitor` is a separate thread which monitors the execution order of the method *`out`* within the



(a) LIGIA's agent classes. (b) LIGIA's server classes.

Figure 3: Relationship between LIGIA's agent classes and Java classes and interfaces.

same agent. It also avoids other primitives being affected by this ordering, like for instance being blocked waiting for their completion. Figure 3(a) shows how LIGIA's agent classes are related to the Java classes.

`LigiaServer` is a multi-threaded class which deals with all the requests coming from distributed agents. `LigiaServer` spawns thread objects of the class `Connection` for each agent thread. Also a thread object of the class `GarbageCollector` is spawned. Therefore the object of the class `LigiaServer` has as its sole job to spawn the `Connection` and `GarbageCollector` objects. The class `Connection` has methods to deal with the request of the agents. These methods are at a lower level than the primitives used by the agents. All the communication is based in sockets. The `GarbageCollector` object is a thread with very low priority and its full description is given in Section 3.4. Figure 3(b) shows LIGIA's server classes in relation to Java classes.

3.4 Garbage Collector

LIGIA is the first generative communication open system to include garbage collection capabilities. In open systems, garbage collection is necessary since compile-time analysis is not available. In any generative communication open implementation problems due to a lack of garbage collection can appear. Consider for instance the toy program in Figure 4.

```
(1) public class Spawner {
(2)   public static void main (String args[]) {
(3)     ConnectionRequest a = new ConnectionRequest();
(4)     for (;;) {
(5)       a.eval(Creator); } } }

(6) public class Creator extends Agent {
(7)   public void run() {
(8)     ConnectionRequest a = new ConnectionRequest();
(9)     Handle tsh;
(10)    tsh = a.tsc(); } }
```

Figure 4: A toy program composed of two agents. The `Spawner` keeps indefinitely spawning `Creator` agents which only create a tuple space and terminate.

One could argue that this sort of program does not happen in real world applications, however the idea expressed can easily appear hidden between the thousands of lines of some real world application. The idea is that tuple spaces are created and never reclaimed and therefore sooner or later a memory exhausting exception will happen. With the garbage collection implemented in LIGIA, programs like this do not cause any harm as the tuple spaces are reclaimed when they become unnecessary to the system.

In order to be able to create a graph, one should first make the LINDA server (kernel) aware of agent identity. In the generative communication systems so far, the kernel had no knowledge of agent identity — a server received requests from *some* agents and filled them. In order to create a structure useful to a garbage collection method, the kernel should be made aware of agents identity in order to identify which agent is executing which primitive at any time.

By looking again to the program in Figure 4 one can see that since Java is an object oriented language, an object has to be created in order to allow the agent to have access to the primitive methods (see lines `(3)` and `(8)`). Since this will have to happen before any communication between the agent and the server, an agent registration was implemented in the constructor of the class. This means that when an agent creates an object to access their methods, a communication between the agent and the server takes place. This communication sets a name to the agent (given by the server). This name is guaranteed to be unique and will be used in all following communications.

The use of the structure above requires a restriction on the implementation. For security reasons, ConnectionRequest objects should not be used as tuple elements. If ConnectionRequest objects are allowed as tuple elements, another agent would be able to retrieve this object and consequently get access to the other agent's unique name and make use of it in future communications. The problem is that the garbage collector does not expect to receive any request from an agent after a *leave* primitive was received.

It should be noticed however, that the restriction is not due to a failure of the garbage collection method but is due to the fact that in object-oriented languages all the methods should reside within classes therefore making object creation necessary [10].[1] In an imperative implementation a link to a library could be used instead of the object creation making the restriction disappear as no object would exist to be passed.

Another important point for the garbage collector is to know when an agent has finished using LIGIA resources. For this the primitive *leave* was implemented. Unfortunately this point relies on the user. It is up to them to make sure every agent has a *leave* method invocation. If forgotten the garbage collector would not be able to collect properly as it will assume the agent is still active. One should notice that the *leave* method would make an ideal Java object finalizer. However, Java does not guarantee that finalizers methods are executed since the Java interpreter can exit before its garbage collector starts — finalizers are called by the Java garbage collector [4]. At this point the program in Figure 4 has to be implemented in LIGIA as the code in Figure 5

```
(1) public class Spawner {
(2)   public static void main (String args[]) {
(3)     ConnectionRequest a = new ConnectionRequest();
(4)     for (;;) {
(5)       a.eval(Creator); }
(*)     a.leave(); } }

(6) public class Creator extends Agent {
(7)   public void run() {
(8)     ConnectionRequest a = new ConnectionRequest();
(9)     Handle tsh;
(10)    tsh = a.tsc();
(*)     a.leave(); } }
```

Figure 5: Program which creates tuple spaces indefinitely

The lines shown with (*) were the ones included. Observe that the complexity of writing the program is not modified with the exception that the user is required to remember to use the primitive. In the case that the *leave* is not executed, the garbage collection will still be able to do some garbage collection. A conservative view of garbage collection is taken and therefore nothing will be collected erroneously but some objects which are garbage can be left in the system.

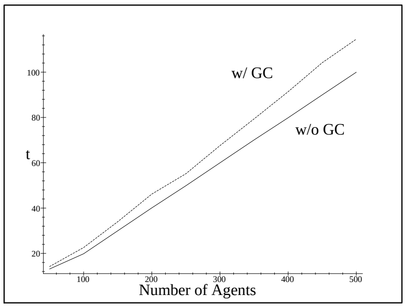
Up to this point little overhead has been included. An overhead of $2m + c$ where m is the size of a message and c is the constant time taken by the server to compute a unique name for an agent. A very tiny overhead is added in each message due to the adding of agent name — the message is slightly longer. Finally no overhead is added by the primitive *leave* in the Java case. *leave* is the primitive responsible for the solution of the *out*-termination problem [8]. The solution consists in waiting for the completion of all primitives in an agent and then terminate. In Java, the main thread only terminates when all the children threads have terminated. Therefore by adding the primitive *leave* the *already existing* overhead is just moved and the LIGIA becomes in control of terminations. Figure 6 shows from where the overhead was moved and where it lays now.

```
public class example_of_leave {
  public static void main (String args[]) {
    ConnectionRequest a = new ConnectionRequest();
    Handle tsh = a.tsc();
    for (int i=0;i<100;i++) {
      a.out(new Integer(i)); }
    a.leave();                               <--- ...to here
  }                                          <--- Overhead moves from here...
}
```

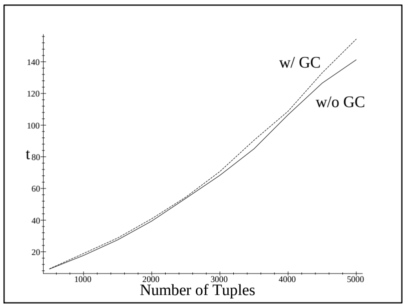
Figure 6: The *leave* implementation waits for the completion of the thread outMonitor and therefore Java does not have to wait at the end of main()

Another source of information for the graph construction is the contents of the tuples involved in the primitive methods. The reason is mainly due to the possibility of having tuple space handles being passed from agent to agent. The server has to be aware of this and the only way of making it know is to have a

[1] Static access to methods is not considered as a new *private* data structure is needed for each connection requested.



(a) Overhead imposed by the process registration.



(b) Overhead imposed by the tuple analyser.

Figure 7: Analysis of the overhead added by the garbage collection implemented

tuple analyser which searches for handles within tuples or templates; if it does find a handle, an action has to be done in the graph. One could argue that this will include a lot of overhead in the system perhaps making the system impractical to use. However the results in Section 4 show a different scenario.

At this point the concept of orthogonality between coordination and computation needs to be emphasised. Java as a computational language provides a garbage collection of its own but this garbage collector does not (and cannot) interfere with coordination objects which compose the LIGIA model. In specific terms the garbage collection method that exists within a Java Virtual Machine can only collect objects within the context of the Virtual Machine. The garbage collection provided in the LIGIA implementation collects objects (LIGIA objects) which reside outside any virtual machine or any computation language context.

4 Results

It has not been possible to formally compare the performance of LIGIA with the performance of other LINDA implementation mainly due to the difficulty of forming a framework from where conclusions can be drawn. LIGIA differs from other systems in several aspects and a fair comparison have to take into account all the differences not only in the model implemented but also the choices made during the implementation. Informally, it has been observed that Java itself make the system much slower around ten times slower (compared with implementations in C for instance), but this is the price of architecture independence. It is believed that if one chooses to a use native code Java compilers the performance of the system would be comparable with systems written in C for instance.

Garbage collection works in LIGIA as a thread with low priority. This causes the thread to stay asleep during most of the execution. The aim is to make this thread include as little overhead as possible by using idle times of the processors. Indeed, the overhead is very little as the results show. Two types of overhead were identified. One is due to agent registration and the *leave* primitive and the other is due to the tuple analyser.

Two programs were implemented and their results analysed so as to observe the overhead added by the inclusion of the garbage collection in the two cases above. They were executed using a Silicon Graphics Indy and a Silicon Graphics O^2. The agents executed in an Indy while the LIGIA server was placed in the O^2.

The case of agent registration a program which does spawning of agents is executed. An agent is spawned, creates a tuple space and terminates (as in Figure 5). When an agent terminates the garbage collector can collect the tuple space the agent has created. The results are depicted in Figure 7(a).

As expected, the use of garbage collection imposes some overhead. However the overhead is small and becomes insignificant if the benefits brought about by the garbage collector are taken into account. In the case of agent creation depicted in figure 7(a) the average overhead was 13.19%.

For the case of tuple analysis a program which does *out* of tuples of size $5 * n$ where n is the average size of an object of one of the classes representing some primitive types (`Integer`, `Float` and `Double`). The results are depicted in Figure 7(b). The overhead added by the tuple analyser is 4.3% of the total time in the case of Figure 7(b).

Jones *et al.* [7] argue that an acceptable garbage collector should impose at most 20% overhead in the system and the experiments shown in Figure 7 shown a total overhead which stays below this number. The experiments above were also done by having the LIGIA server placed on a Pentium 166MHz and on a Silicon Graphics Indy. Although the figures were different the results were similar in terms of percentage of overhead added by the garbage collector.

5 Conclusion

Garbage collection is not yet a fully investigated subject in generative systems. It has been shown how to implement garbage collection in a particular generative system called LIGIA and proven that garbage collection implementations should not impose big overhead to the system. It has been found that the overall cost of the garbage collection method implemented was little compared with the benefits brought by its inclusion in the system.

Despite the work done in this paper, there are still plenty of areas to be explored. The implementation of of LIGIA is not yet optimal. A multi-server version is being implemented using a distributed garbage collection version of the method implemented in this paper, this version will also include distributed I/O via tuple spaces [9]. This will make the garbage collector even more powerful in deciding what to collect. For the context of garbage collection, subgraphs with I/O accessibility (one of the processes of the subgraph has I/O capability) cannot be collected.

The garbage collection for the multi-server version of LIGIA will also have to be able to deal with movement of tuple spaces within the servers. The main advantage of a multi-server implementation is the possibility to use load-balancing techniques having tuple spaces moving around as needed and the garbage collection method has to be optimised for this.

References

[1] N. Carriero and D. Gelernter. The S/Net's Linda Kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, 1986.

[2] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly Garbage Collection: An Exercise in Cooperation. *Communications of ACM*, 21(11):966–975, November 1978.

[3] A. Douglas, A. Wood, and A. Rowstron. LINDA Implementation Revisited. In P. Nixon, editor, *Proc. of the 18th World Occam and Transputer User Group*, pages 125–138. IOS Press, April 1995.

[4] D. Flanagan. *Java in a Nutshell – A Desktop Quick Reference for Java Programmers.* O'Reilly & Associates, Inc., 1996.

[5] D. Gelernter. Generative Communication in LINDA. *ACM transactions in programming languages and systems*, 1:80–112, 1985.

[6] P. Hudak and R. Keller. Garbage Collection and Task Deletion in Distributed Applicative Processing Systems. In *Conference Record of the 1982 ACM Symposium on List and Functional Programming*, pages 168–178. ACM Press, Pittsburgh, PA, August 1982.

[7] R. Jones and R. Lins. *Garbage Collection — Algorithms for Automatic Dynamic Memory Management.* Wiley, 1996.

[8] R. Menezes and A. Wood. Garbage Collection in Open Distributed Tuple Space Systems. In *Proc. of 15th Brazilian Computer Networks Symposium - SBRC'97*, pages 525–543, São Carlos - São Paulo - Brazil, May 1997.

[9] R. Menezes and A. Wood. Coordination of Distributed I/O in Tuple Space Systems. In *Proc. of 31st Hawaii International Conference on System Sciences*, volume VII, pages 216–235, Big Island, Hawaii, USA, 1998. IEEE Computer Society.

[10] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, Inc, 2nd edition, 1997.

[11] A. Rowstron. *Bulk Primitives in* LINDA *Run-time Systems.* PhD Thesis, Department of Computer Science – University of York, 1996.

[12] Sun Microsystems, Inc. *Java Object Serialization Specification*, February 1997. Revision 1.3. JDK 1.1 FCS.

[13] Sun Microsystems, Inc. *JavaSpace Specification*, June 1997. Revision 0.4.