Experience with Memory Management in Open Linda Systems

Ronaldo Menezes
Florida Institute of Technology
Department of Computer Sciences
150 West University Blvd
Melbourne, FL 32901
rmenezes@cs.fit.edu

Abstract

Coordination systems, in particular Linda, have established themselves as important tools for the development of applications for open systems such as the Internet.

This paper shows how to tackle a forgotten, but crucial problem in open coordination systems: memory management. As with any system which intends to be of wide use, coordination systems have to address the problems of memory exhaustion as memory is a finite resource. This paper first explores the separation between coordination and computation in order to make it clear that the problem of memory exhaustion in coordination systems *cannot* be solved using garbage collection schemes implemented at the computation language — a garbage collection scheme must exist in the coordination environment as well.

As Linda is arguably the most successful coordination system, this paper will focus on the Linda family of systems. It is expected that the solution in Linda can be adapted to other coordination systems.

1 Introduction

Linda has surely caused an impact in the computer science community. Since its proposal by Gelernter [12] the basic idea of Linda has evolved to a point of even attracting the attention of the industry. Recently two major computer companies have released commercial packages based on Linda: Sun Microsystems with JavaSpaces [30] and IBM with T Spaces [17].

The path to achieve such a recognition was not easy and it took nearly 15 years of research to convince the world of Linda's abilities. In its evolution, Linda has changed from a model aimed at parallel computing to a model aimed at distributed computing, particularly suited to *open* distributed computing.

The first implementations of Linda were confined to closed (controlled) environments [28, 27]. In the closed case most problems are of little concern as they can be dealt with at compile-time. However, the same is not true in open implementations of Linda. Any existing problem handled at compile-time in closed systems becomes a concern in the open case as solutions have to be implemented at run-time involving dynamic reconfiguration of data structures and more control over race-conditions.

Additionally, problems in closed systems are confined to the boundaries of a single computation language, which hides the idea that coordination and computation are orthogonal concepts — the computation language defines a boundary to coordination giving the impression that the computation language itself deals with coordination. In open systems the situation is rather different. Open applications can be implemented over different languages, different machines, different operating systems, that is, a complete heterogeneous environment. This heterogeneity stops computation languages from being able to deal with coordination problems since the problems break the boundaries of computation languages. Hence, coordination problems need to be tackled at the coordination level — within the coordination environment.

The aim of this paper is twofold:

1. Develop a way of keeping enough information about Linda objects within the kernel so that optimizations can be implemented based on this information.

2. Describe how the information maintained can be used to tackle the memory management problem in Linda.

Garbage collection of coordination objects cannot be performed at the computation language level. Instead, a garbage collection scheme has to be implemented so as to collect solely coordination objects. With the introduction of the multiple tuple spaces concept in Linda [13], processes became able to create tuple spaces via a primitive provided. However, in open implementations of Linda, these tuple spaces remain 'forever' in the system once they are created. This paper argues that the implementation of a garbage collection system within Linda is the only possible solution to avoid memory exhaustion when large applications are implemented in open Linda systems.

This paper is divided as follow. In Section 2 the background work is discussed. First the orthogonality argument of Gelernter and Carriero [14] is explained as this is an essential concept to understand why the memory resources at the coordination level cannot be managed by computational languages. Then a brief introduction to the Linda models is given followed by a discussion on open implementations of this model. Section 3 explains the memory exhaustion problem in Linda and why it can only be solved if more information becomes available at the kernel level. Section 4 shows how information about accessibility of tuple spaces by processes can be gathered in a distributed graph structure. Section 5 shows how to implement garbage collection using the distributed graph present at the kernel level. Section 6 presents some experimental results extracted from the implementation of the garbage collection described. Finally, Section 7 presents the conclusion of this paper.

2 Background

2.1 Computation vs. Coordination

Although the orthogonality between computation and coordination may seem clear, only recently this concept has been formally proposed. After the introduction of the Linda model [12] and its subsequent extension to the multiple tuple space model, the idea of Linda as a model exclusive for parallel computing was replaced by the idea of having Linda as a model for distributed computing. However, Linda does not represent a distributed system by itself, its implementation has to be embedded in languages like C [29], Prolog [32], ISETL [9], and others.

Gelernter and Carriero observing this characteristic argued that Linda does not deal with *computation* but only with *coordination* aspects of a distributed system — Linda is a *coordination model* [14]. The idea can be easily observed with an example: if two processes are each generating a number that will be added by a third process, coordination models are not interested in how the addition is carried out but in how the two numbers are obtained before the operation takes place — coordination is the process of managing dependencies between distributed activities. Soon after the term coordination became accepted by the computer science community, other existing models, like Actors [1] and CHAM [2] were also classified as coordination models since they, like Linda, are not concerned with *computation*.

In reality, Gelernter and Carriero, did not create the term coordination, they have only described how the coordination idea introduced by Malone and Crowston [20], could be applied on the design of distributed computer systems. They have described a distributed language in terms of coordination and computation and have shown that a coordination model (or language) when put together with a computation language form a complete distributed language.

2.2 The Linda Coordination Model

The Linda coordination model [12, 13] is based on the concept of generative communication via distributed associative shared memories. The shared memories are called *tuple spaces* and behave like bags (unordered multi-sets) of tuples. There is no direct communication between processes; all communication takes place

via tuple spaces. The model provides the processes with primitives to access the tuple spaces: storing and retrieving tuples, creating tuple spaces, and spawning other processes.

Tuples in Linda are ordered lists of valued-objects (actuals). The generative communication model is based on associative matching. The matching implemented in Linda is based on *templates* which are no different from tuples except for being able to have non-valued objects (formals) represented by `?type`. For instance the template `[?int, "Hello"]` matches the tuple `[1,"Hello"]`.

The primitives provided are[1]: `out` (stores a tuple in a tuple space), `in` (removes a tuple from a tuple space matching the template provided), `rd` (gets a copy a tuple from a tuple space matching the template provided), `collect` (moves all tuples matching the template from a tuple space to another), `copy` (copies all tuples matching the template from a tuple space to another) and `eval` (spawns a new process). By using these primitives, Linda unifies the concepts of process creation, communication and synchronization since they are all implemented as tuple space operations.

There are variants of the Linda model (called nowadays the Linda family of models) such as: Melinda [16], Bauhaus [5], PageSpace [6] and Bonita [26]. Despite their differences all these models may suffer from the same problem should they be implemented in open environments: *memory exhaustion.*

2.3 Open Implementations

One might wonder why the memory exhaustion problem has not yet been felt to a great extent. The answer is simple; only recently open implementations of Linda started to appear and even the ones already available have not yet been used on the implementation of large applications. However, the increasing number of commercial open implementations of Linda — such as JavaSpaces and T Spaces — will in a short period of time make Linda a viable commercial product for developing open applications. An example of this phenomenon can be observed with Sun Microsystems' Jini [31] which includes JavaSpaces as one of its components. Yet, Jini is not "the" perfect example because JavaSpaces (coordination) is confined to a single computation language (Java).

In open implementations of coordination systems, problems that were previously solved at compile-time, are now required to be solved separately by a run-time sub-system. To make matters worse, even vital information necessary for the implementation of these sub-systems is no longer available. For instance, in a closed system, it is possible for the kernel to build at compile-time a structure containing the information about what processes are *likely* to access which tuple spaces. With this information in hand, decisions (optimizations or garbage collection for instance) can be made without interference in the execution of the system.

Generally speaking, one can say that due to the existence of enough information in closed systems the solutions for possible problems are easier. In other words, the degree of difficulty of problems in open systems is higher than the same problems for closed systems. As the majority of problems in closed systems can be dealt with at compile-time, the solutions can use stable (passive) structures containing the information. In open systems not only does the information have to be maintained (gathered) at run-time, but also the solutions have to use the information available at the right time when the information is coherent with the state of the system.

3 Memory Exhaustion

Memory management haunts every large application developer. As computers becomes cheaper, systems tend to be larger and use more of the memory resources. However, memory is a finite resource and its utilization is (and will always be) a concern.

As mentioned before, the problem has been overlooked in Linda systems. There are a number of reasons for this:

- as the first commercial implementations of Linda were closed [28, 27], compile-time optimizations at the kernel level were used to manage memory usage;

[1] `collect` and `copy` are not standard primitives but the concept of bulk primitives is well accepted for them to be included here.

- in closed system the orthogonality of coordination and computation is not observed since both concepts are mingled into one distributed/parallel language;
- large, memory demanding applications have not been implemented using closed Linda implementations because most of them demand open systems;
- open implementations of Linda are largely research prototypes where only small applications (test programs) are implemented making memory management not an issue;
- only recently companies like Sun [30] and IBM [17] started to explore the possibilities of open Linda-like implementations in the Internet context but their implementations are still to be used in the development of large open applications.

Still, the applicability of coordination systems to the Web and the benefits they can bring to the Internet are well known and accepted. Therefore, it will not be long before the existence of memory management systems becomes mandatory. Palliative solutions can be implemented but this paper looks ahead and identifies the core of the problem: lack of information within Linda kernels.

4 Information Gathering

Open implementations of the Linda model have to find means of building up information about processes, tuple spaces and tuples in the kernel. This can be done by using for instance pre-processing, using active agents for gathering information at run-time [19], or gathering the information using the messages in transit within the system [23].

The use of pre-processing can, in some cases, build an initial amount of information that allows some optimization, however, as pre-processing is normally done from a single process point of view it does not help in the solution of global problems like the problem of memory management. The solution based on active agents, although very elegant, might add undesirable overhead to the system. This paper explores how to solve the problem of memory management based on the idea of gathering information using the messages in transit between the Linda processes and the Linda kernel. It is trivial to see that most of the information necessary to build a structure that could be made available to sub-systems (such as a garbage collection) is already in transit in Linda systems. However, this information is volatile — because it is not stored, it is lost.

In order to add some persistence to the information in transit in the system, a distributed graph[2] with the information of which processes access which tuple spaces is built in the Linda kernel.

4.1 Consideration of Tuples

Before discussing how the graph is build it is worth clarifying why tuples will not be considered in the garbage collection scheme described here. One could naively think that as tuples are the basic elements being stored and retrieved in Linda, any memory management should be done at the tuple level. The reason for not doing this is that tuples in Linda are not uniquely identified. A tuple is just an ordered list of values and processes do not keep references to them. Yet, processes do keep references to tuple spaces (bag of tuples) as their names have to be known by the process if this process intends to store/retrieve tuple from them.

One solution that is worth mentioning is the idea of leases used in JavaSpaces. The basic idea is that resources can be allocated for a fixed period of time. For instance, a tuple is created with a "expiration date". Freeman *et Al.* [11] argue that leases are ideal for unreliable distributed applications where processes can fail before *resources are explicitly freed.* While leases can be used in certain cases where the behavior of the application is known, they cannot be used when the behavior is unpredictable. In asynchronous systems, processes cannot predict for how long a tuple will be required in the system. Therefore, the process cannot delete any tuple explicitly. This paper argues that not even the kernel can free resources at the tuple level because tuples are accessed associatively. Leases work in JavaSpaces because tuples are confined to a single language and are represented as Java objects, but are not a general purpose solution.

[2] Despite being called a graph, the name stands only for the pictorial representation of the data structure containing the information itself.

4.2 Using the Messages in Transit

This paper therefore concentrates in having a graph with information about tuple spaces and processes only. The construction of this distributed graph is based on two concepts that need to be introduced in the kernel, so that the information in transit does not get lost.

Process Registration/Check-out: The kernel should be aware of all processes are willing to coordinate using Linda primitives. Process registration consists of having each process communicating with the kernel so that it can assign them with a unique name. From the point of registration onwards all processes will use their internal names in all communication with the kernel.

The dual of the process registration is the process check-out where the processes inform the Linda kernel that it does not intend to use Linda facilities any longer.

Tuple Monitoring: Tuple monitoring consists of having the Linda kernel analyze all *relevant* information in transit. It is called tuple monitoring because the basic information in transit in Linda are tuples. Since processes can be identified by their names, the tuple monitoring is effective in extracting information stored within tuples.

With the implementation of the concepts above, Linda is being given more power as it becomes aware of what is happening in all communications. As processes and tuple spaces are uniquely identified, the distributed graph can be built to hold information about the system.

Although the implementation of the concepts above is crucial for the solution of the memory management problem they are not just a commodity that is being implemented only to satisfy the garbage collection scheme. The graph is building up information on the kernel that can be made available to other optimization run-time systems.

4.3 Creating Distributed Graphs

Clearly, with the information available in the kernel after the introduction of the concepts described in Section 4.2 a data structure can be built to store the information. As said before, the intention is to build a data structure (pictorially seen as a distributed graph) with the information of tuple spaces usage by processes. Due to the distributed characteristic of this graph some rules have to be defined:

- all existing (current) tuple spaces and processes are represented in the graph;
- there is only one representation for a process and a tuple space across all distributed components of the graph;
- in order to observe the distributed components as part of a whole (connected) graph the representation of the **U**niversal **T**uple **S**pace (*UTS*)[3] is repeated in all components to serve as the connection among them;
- processes hold references to tuple spaces and tuple spaces might store references to other tuple spaces but processes do not hold references of other processes — references are represented as edges in the graph;
- for the purpose of garbage collection algorithm, edges linking processes to tuple spaces are *undirected* whereas edges linking two tuple spaces are *directed*, outgoing from the tuple space that contains the reference;
- directed edges are weighted with the number of references stored in the outgoing tuple space that contains the references;
- both directed and undirected edges are never duplicated — given two nodes there is a maximum of one undirected edge between them or two directed edges (one in each direction).

[3] It is generally the case that Linda systems have a medium of communication which is known to all processes.

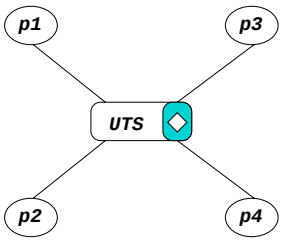

Figure 1: Simple graph situation.

Based on what has been described above, Figure 1 could represent the situation of the data structure in one of the locations (one component of the graph). The figure represents a scenario where four processes are executing and because they all are aware of the existence of *UTS* there is an edge (undirected) linking each one of them to *UTS*. The figure is equivalent to have the following adjacency matrix {{*p1* {*UTS* }},{*p2* {*UTS* }},{*p3* {*UTS* }},{*p4* {*UTS* }},{*UTS* {*p1*,*p2*,*p3*,*p4* }}}.

There are several ways to get to a particular graph configuration. The one shown in Figure 1 may have got to this configuration by having the four processes starting soon after the kernel was initialized. During the initialization the node *UTS* is created. Then when a process registers with the kernel a name is attributed to the process, a node is created representing it and the necessary edges are added, in this case the edge linking each of them to *UTS*.

For a better understanding of the garbage collection algorithm used, the nodes of tuple spaces in the graph contains also an extra field which represents the number of references from other objects (tuple spaces and processes) to this tuple space. In the *UTS* case it is set as $\Diamond$ to represent that *UTS* has infinite number of references — by default, all processes know about *UTS*.

Undirected edges are *not* weighted. Once a process (an active object) knows about a tuple space it can generate as many references (tuple space handles) as it likes. Therefore, the undirected edges are kept without a weight.

A more complex graph is shown in Figure 2. Again there are numerous ways of getting to the situation depicted. Before explaining in more detail how the structure is built it is important to understand the importance of keeping this information in the kernel. The graph in Figure 2 holds important information that is normally not available in Linda systems. For instance, it is easy to see that tuple space *ts7* is not being used by any process and that its handle is not available within any other tuple space. Basically, this tuple space is garbage and could be garbage collected — the kernel now maintains information that was not available before. Observe that although the system being considered is fully open, because tuple spaces are accessed by their names it is impossible for any process to get the handle of *ts7*. As another example suppose that both processes *p4* and *p5* terminate. Should this happen, their representations will be removed from the graph causing tuple spaces *ts3*, *ts6* and *ts9* to be unreachable to all other processes and consequently be considered garbage.

Consider the additional example shown in Figure 3. This scenario shows how the distributed graph can help on the implementation of mobility in Linda. There is a big interest on mobile computation today mainly due to the availability of the Web as a distributed network for implementation of systems that span the globe. Static techniques adapted from Local Area Networks (LANs) like RMI/RPC and Corba, although successful in fairly large Wide Area Networks (WANs) are not well adapted to the Web context [4]. If mobility is to be implemented in Linda the information provided by the distributed graph considered here is of utter importance. Figure 3 shows a simple case where process *p5* is the sole process accessing the tuple space *ts1*. The fact that *p5* is in a different location may suggest to a mobile system that *ts1* should migrate to location B or alternatively process *p5* should migrate to location A (given that this process is also not accessing other tuple spaces in location B).

The information put together in the graphs depicted so far is made available by the two concepts described in Section 4.2. First, process registration allows the kernel to create a node for the process in the graph as well as identify all messages related to this process from the moment of registration onwards. During

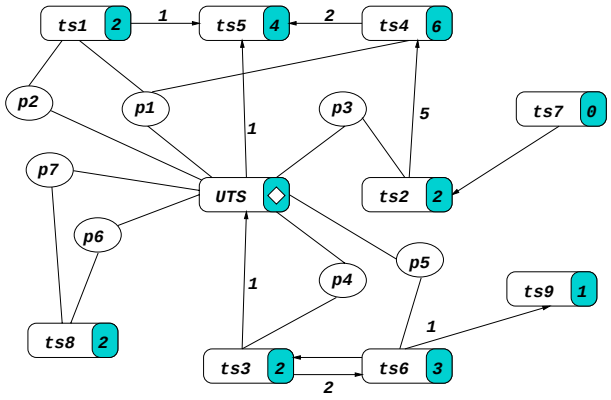

Figure 2: A more complex graph situation.

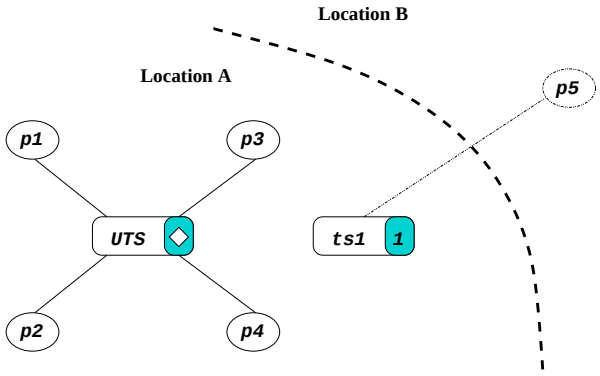

Figure 3: A simple case of distributed graphs.

the registration some links can also be created to existing tuple spaces if the process is spawned receiving handles as parameters.

Tuple monitoring is used to keep the graph updated. It behaves as an independent process that basically listens to the messages being transmitted between the Linda kernel and processes and vice-versa. Although all messages in transit are potentially important, it is possible for messages to be identified so that only three types of messages are considered:

- messages requesting process registration;
- messages storing or retrieving tuples containing handles;
- messages requesting process check-out;

Although process registration and check-out is dealt with separately, it is up to the tuple monitoring to identify the message as a registration or a check-out request. The messages are marked when they are mounted at the Linda process side. This flagging does not impose big overhead to the system as it occurs when the tuple is packed to be sent to the server.

The tuple monitoring updates the graph in several ways. For every message storing a tuple that contains a handle (via `out`, `copy` or `collect`) a directed edge has to be created in the graph to represent a *dependency* between the tuple spaces involved — if a tuple space `tsx` contains a handle of `tsy`, any process with access to `tsx` has potential access to `tsy` by retrieving its handle. If a directed edge already exists linking the two tuple spaces, its weight is modified accordingly since the weight represents the number of handles of one tuple space stored inside another. The other messages monitored are the ones retrieving handles from

a tuple space (*in*, *rd*, *copy* and *collect*)[4]. If a handle is removed from a tuple space the weight has to be updated to reflect the change, and if the weight of the directed edge gets to zero the edge is removed altogether. The retrieval of handles can grant a process access to a tuple space, therefore not only are the necessary directed edges updated but, if necessary, an undirected edge between the process and the tuple space has to be created.

4.4 Avoiding Race-Conditions

The operations that update the graph must be done with care in order to avoid race conditions. Observe for instance a case with primitive *in*. Suppose that in the scenario depicted in Figure 2 process *p5* removes the tuple containing the handle of *ts9* from *ts6*. If a garbage collection scheme is using the graph to decide on what is garbage and what is not, a race condition might occur between the garbage collector and the update of the graph. When process *p5* removes the handle it automatically gains access to *ts9*, however if the order of updating the links is not observed tuple space *ts9* can be considered garbage: if the edge linking *ts6* to *ts9* is removed before one linking *p5* and *ts9* is created.

One of the characteristics of the update method used is that garbage is only created by process termination. Removal of directed edges does *not*, in any circumstance, generate garbage. The edges in the graph represent knowledge of a particular tuple space, if one is removed due to some process operation another will be created (or modified in terms of weight). References to a particular tuple space are never lost due to execution of Linda primitives.

On the other hand, process termination is an operation that can generate garbage as references may be lost. Race conditions between garbage collection and graph update is a well-known problem in distributed systems and has been solved using various approaches developed through the years [15, 3, 25]. The adaptation of these models to the Linda case would avoid the race-condition. Yet, this paper uses a different approach and instead of adapting one of these (more complex) algorithms, uses a simple solution and avoids the race conditions by guaranteeing that at any time the graph is consistent with the system situation.

Douglas *et Al.* [10] have described the *out*-ordering problem showing that this is a problem that must be solved in any Linda-like coordination system. The problem consists of guaranteeing that the order of execution of a sequence of *out*s to a tuple space *tsx* is the same order that these tuples appear in *tsx*. The solution for avoiding race conditions in the graph can be seen as an extension of this problem where some ordering is also imposed to the check-out (termination) message sent by Linda processes.

4.5 Termination ordering

Termination ordering consists of guaranteeing that no message will arrive at the kernel coming from a process whose check-out message has already been processed by the kernel.

From the point of view of garbage collection, race conditions are removed if the system guarantees the above. At first this may sound restrictive as ordering is the least attractive operation in distributed systems. However, the experiments shown in Section 6 show that this solution imposes little overhead to the system.

The argument that can be used to show that little overhead is added by this extra ordering is based on the fact that only the primitive *out* needs to be considered in the ordering. Other primitives always return something to the process therefore ordering already exists as part of the semantics of the primitive. For instance, suppose that a primitive *in* is the last primitive executed before the process checks out. As *in* returns a tuple, a check-out operation immediately following this *in* primitive would not even be executed at the process side until the primitive *in* is completed, thus guaranteeing the ordering of execution. The primitive *out* is the only one that does not return a value and consequently the only one that needs to be considered in the ordering with termination (check-out).

5 Garbage Collection

Garbage collection in Linda is implemented at the kernel level for this is the only way to guarantee that the scheme would not be restricted to boundaries of computation languages. The solution implemented aims at

[4] Bulk primitives, *collect* and *copy*, both store and retrieve tuples from tuple spaces.

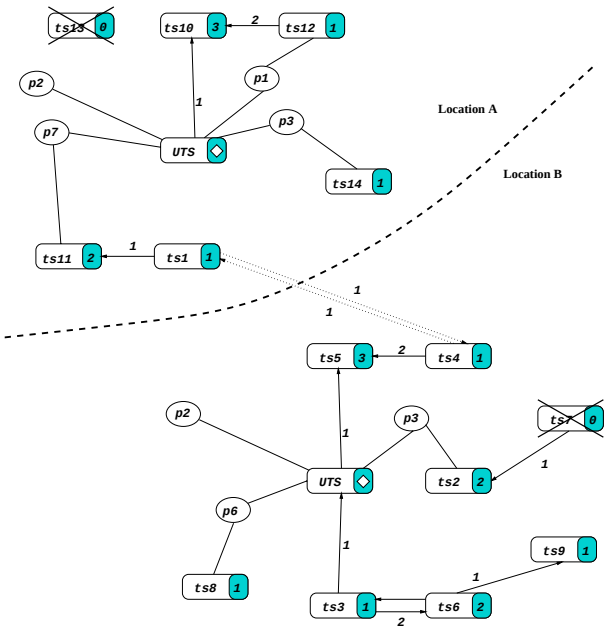

Figure 4: Reference counting identifies two tuple spaces as garbage.

finding garbage tuple spaces. However this section also looks at what would be required to garbage collect active process as well as tuple spaces

5.1 Finding Garbage Tuple Spaces

Given that the information is now available in the kernel, the implementation of a garbage collection scheme for finding tuple spaces which are no longer necessary in the system is simple. The most popular garbage collection algorithms available are variations of one of the two basic methods: Reference Counting[7] and Mark-and Sweep[21].

The scheme implemented in Linda is based on a two-phase garbage collection. The first phase uses reference counting to find tuple spaces that are clearly not required within the system. Tuple spaces that have their counter set to *zero* are wasting memory space and can be removed. If the counter is set to zero no process or other tuple space has a reference to this tuple space — this is the most basic case of garbage collection. Figure 4 shows a scenario where two tuple spaces, `ts7` and `ts13`, can be removed by the garbage collector since their counters are zero.

The choice of reference counting is partially due to the fact that it can be easily implemented in distributed systems and partially due to the cost associated with running the algorithm being low [18]. Reference counting generally suffers from race-condition problems in the update of the counter. In fact, this is the primary reason why extensions for this solution have been used in distributed systems [3, 8, 25]. The elimination of all possible race-conditions (as explained in Section 4.4) at the Linda system level makes reference counting appropriate to be used in Linda systems.

However, reference counting methods do not always find all garbage; they do not work well in collecting cyclic garbage structures. These structures appear when garbage nodes keep references to each other making the counter greater than zero causing the reference counting algorithm to identify these nodes as still being required in the system. Despite this drawback the cost of the algorithms justifies the option of implementing a scheme in two phases. The phases are implemented in such a way that the reference counting runs more frequently and ideally collects most of the garbage.

The lack of a killer application in Linda makes it difficult to know whether reference counting will collect most of the garbage. Still, based on patterns that appear frequently in coordination systems, it is likely that

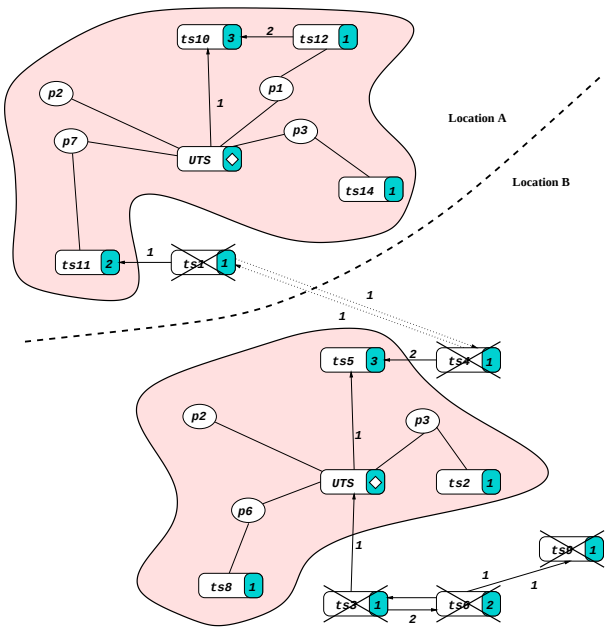

Figure 5: Marking identifies five tuple spaces as garbage and the sweep step collects them.

tuple space will be either short lived or remain in the system for a very long time. The idea is that reference counting could easily garbage collect short lived tuple spaces as they become unnecessary in the systems.

Despite the expectation that reference counting will collect most of the garbage tuple spaces, a second phase is nevertheless important so that cyclic garbage is collected. Mark-and-sweep is able to collect cycles because it does not decide on the need of an object based on its counter. Instead, mark-and-sweep traverses the graph marking everything that can be reached from special nodes called *roots*. After the graph is traversed nodes that could not be reached are considered garbage. In Linda *UTS* can be nominated as root because it represents the boundary to the *outside world* (non-Linda environment). Processes, being active objects, could also act as *roots* in the search but because they are all linked to *UTS* this is not necessary — if a node is reached from a process this node is also reached from *UTS*.

The mark-and-sweep has been implemented using a concept based on sets. In the marking step, nodes reached from the roots are included in a set that represents the nodes alive (not garbage). In the end of the marking the nodes left outside the set are considered garbage and can be collected by the sweep step. By performing a marking in the scenario depicted in Figure 4 more garbage can be found, as shown in Figure 5.

It should be clear why the tuple spaces identified as garbage (crossed) in this phase are indeed garbage. Take for instance the case of tuple spaces *ts3*, *ts6* and *ts9*. Although *ts3* contains a reference to *UTS*, the traversal that started from the *UTS* cannot reach any of these tuple spaces — the search respects the direction of the edges. Also there are no processes accessing them, in fact if there were, the search from *UTS* would have reached the tuple spaces.

The mark-and-sweep phase itself can be further divided into two stages: a local and a global mark-and-sweep. Given that the representations of tuple spaces in the graph contain some information about the location of the objects with access to a particular tuple space, local and global mark-and-sweep can be used. In the scenario shown in Figure 5, *ts3*, *ts6* and *ts9* would be collected by a local mark-and-sweep since none of them has links to other locations whereas *ts1* and *ts4* would be collected by a global mark-and-sweep, as this would have the view of the whole distributed graph.

5.2 Considering Active Processes

When extending the concept of memory management to resource management, can the graph help on garbage collecting processes that are only wasting processor time? The answer is *no*.

Since all processes are linked to *UTS* they can always side-effect other processes. Surely if this condition is relaxed by not having all processes aware of the existence of *UTS*, the graph would provide information so that garbage collection of active processes may be done.

It should be noticed that the concept of garbage collection of processes here does *not* include garbage collection of deadlocked processes which are for instance blocked (via *in* or *rd*) waiting for a tuple. This should not be dealt with by a garbage collection scheme, instead the semantics of the primitive should foresee and deal with this situation.

UTS as a tuple space has handles that can be passed to other processes via tuple spaces. Therefore, the assumption that all processes know by default about *UTS* can be considered too general and unnecessary.

In order to garbage collect processes the scheme described in Section 5.1 may be used but processes have now to be considered as roots of the graph since not all of them are linked to *UTS*. As active objects, even if they are not linked to *UTS*, they can get access to *UTS* by retrieving its handle. This has to be considered in the mark-and-sweep phase which needs to search the graph starting from the processes nodes as well as *UTS*. However, *UTS* remains the center of the algorithm. In the end of the search a given node *x* (process or tuple space) is garbage if:

1. it is not contained in any set generated during the search; or,
2. it is contained in a set but *UTS* is not an element of it.

The creation of the sets during the marking is very simple. If the search has started from a node y, all nodes reached in this search will be elements of the set Y and they are marked as reached by y. Given a set Y and a set Z, if $Y \cap Z \neq \emptyset$ then Y and Z have to merge forming a single set.

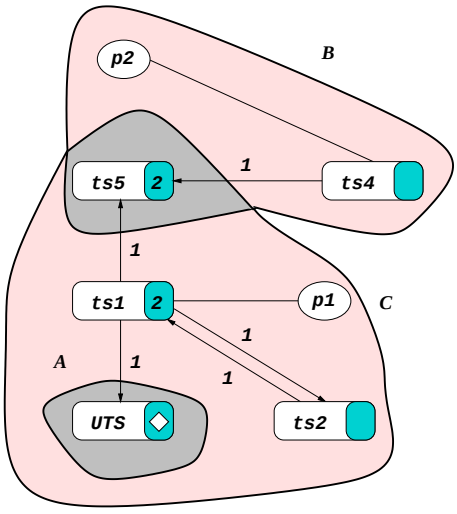

Figure 6: Scenarios showing how sets merge during the marking.

Suppose the two scenarios in Figures 6 and 7 which are now feasible given that not all processes are linked to *UTS*. Figure 6 shows three sets A, C and B that were generated by the marking phase starting from *UTS*, *p1* and *p2* respectively. In this figure none of the objects is garbage because the sets can be merged forming one single set that contains *UTS* as one of its elements. Using the property described above, because $B \cap C = \{ts5\}$, they merge forming one single set, say D. Now $D \cap A = \{UTS\}$ means that A and D also merge. In the end of all merge operations a single set given by $\{UTS, p1, p2, ts1, ts2, ts4, ts5\}$ is formed. This set represents the nodes that are *not* garbage.

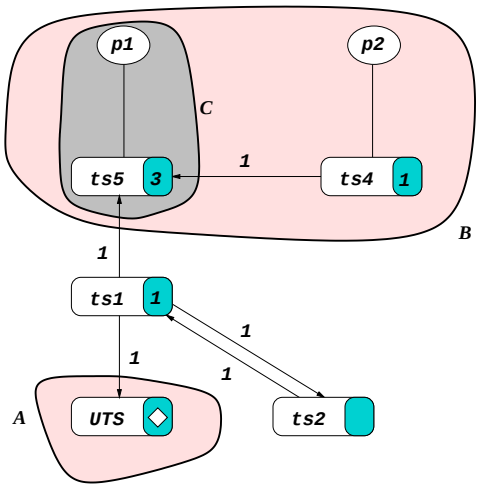

Figure 7: Scenarios showing how sets merge during the marking.

Figure 7 shows how the marking phase can find garbage nodes. Similar to the case in Figure 6 sets B and C merge because they have elements in common. However, the set resulting of the merge, say D, does not have an element in common with A, that is, $D \cap A = \emptyset$. Therefore in the end of the marking the following applies:

- *ts1* and *ts2* are garbage because they do not belong to any set;
- *p1*, *p2*, *ts5* and *ts4* are also garbage because *UTS* is not an element of the set they belong, D;
- *UTS* will be the only node left in the graph.

5.3 The I/O Side-Effect

Given the method described so far one question remains: what happens if a process is not linked to *UTS* but intends to output some results? This in fact might be a very common situation where a process *px* starts, creates a tuple space *tsx*, spawns another process *pz* passing the tuple space handle of *tsx*, and terminates. *pz*, in turn, will access information in *tsx* and display the result. Surely it is undesirable to garbage collect processes doing I/O even though they are not linked to *UTS* in any way.

Menezes and Wood [22] have shown that I/O is a coordination problem and must be addressed within the coordination model. The authors described how devices and files can be abstracted in terms of tuple spaces allowing Linda processes to deal with them using Linda primitives. In the context of garbage collection this can be represented by adding a tuple space that represents I/O within the model. If a process has the handle of this *I/O* tuple space this process is able to do *I/O* operations or in other words abstract devices and files as tuple spaces.

This slightly modifies the concept of garbage defined in Section 5.2. The modification is due to the existence of a new tuple space that is also root of the graph: *I/O*. A node that is element of a set that does not contain *UTS* but contains *I/O* as element is *not* considered garbage. The practical reason for this is that a node that can reach the *I/O* tuple space can in the Linda model affect the outside world (non-Linda environment) by doing I/O operations.

Figure 8 shows the case where an *I/O* tuple space exists. In the scenario shown, although *p6* and *p7* are not linked to *UTS* in any way, they are not garbage because the definition of garbage has to consider the reachability from the *I/O* tuple space in the same way it considered for *UTS* — the set formed from a search from either *p6* or *p7* will include *I/O* as an element.

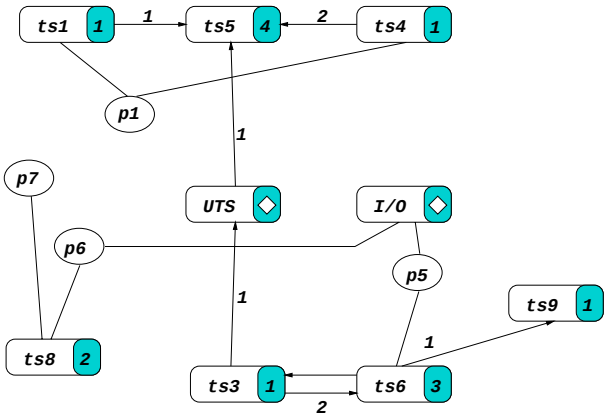

Figure 8: Graph showing the *UTS* and the *I/O* tuple space where not all process have their handles.

6 Experimental Results

The system described in the previous sections was implemented using Java. All the concepts described were implemented in a Linda-like kernel called Ligia [24].

The first experiment, described in Figure 9, aims at showing that garbage collection is in fact avoiding early memory exceptions due to the existence of unnecessary information in the memory.

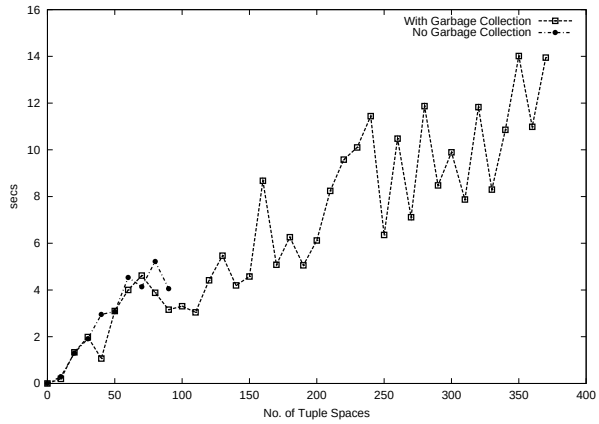

Figure 9: Avoiding early memory exhaustion.

The experiment was performed using several Linda processes that start, create n tuple spaces and terminate. For each new process starting, the number of tuple spaces created is incremented by 10. The Java Virtual Machine was configured so that memory exhaustion happened when around 450 tuple spaces were stored in memory (10+20+30+...+90). Figure 9 shows that if the garbage collector is not active, the memory exhaustion problem happens when a process is about to create 100 tuple space because there are already 450 tuple spaces stored in memory. When the garbage collector is active the exception does not happen at the same point because the garbage tuple spaces are removed from memory when the processes terminate. With garbage collector active the exception occurs when a process start and tries to create 370 tuple spaces.

Although the capacity of the memory is around 450 tuple spaces the exhaustion happens when 370 are being created due to the concurrent nature of the garbage collector. The garbage collector is running concurrently with the execution of Linda processes which means that some garbage may still be in the memory when the exception happens. This behavior could be avoided by implementing an exception handling system which calls the garbage collector and only if it cannot free any memory the exception takes place. However,

the system used was implemented in Java and the *OutOfMemoryError* is one of the exception that cannot be handled in Java.

Overhead is one of the biggest concerns when implementing a garbage collection scheme. Figure 10 demonstrates the raw overhead of the garbage collector described previously.

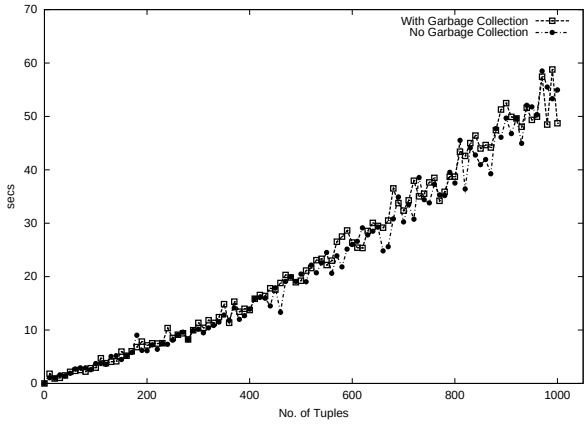

Figure 10: Worst case overhead added by the garbage collector.

The experiment shown in Figure 10 was performed by having a Linda process storing tuples in a single tuple space. The process loops and in each interaction its increases the number of tuple being stored by 10. Because all tuples are being stored by a single process into a single tuple space, no garbage is created until the process terminates. Therefore the overhead observed in Figure 10 accounts for the tuple monitoring and the cost of the garbage collector. Process registration/check-out is not considered because it only happens once per process; it adds a startup time which does not have influence on the process total running time.

The average overhead observed in Figure 10 was around 5% which is bellow what Jones and Lins have argued as acceptable in terms of overhead due to garbage collection [18]. Yet, this is the worst case of the overhead observed because the system is not benefiting from any improvement done by the garbage collector. It is natural to believe that if a system generates an excessive amount of garbage, the execution of the garbage collector can improve the performance of the system by keeping the memory tidy.

Figure 11 shows a scenario which is the opposite of the case in Figure 10. The process loops similarly to the previous case, but instead of storing the tuples in a single tuple space the process spawns another process which in turn creates a tuple space, stores the tuples and terminates — garbage is created.

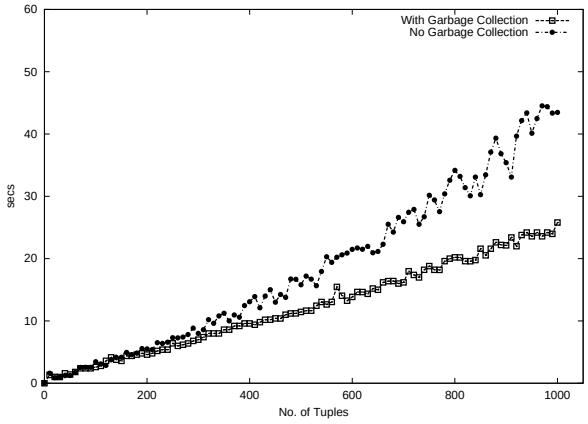

Figure 11: Garbage collector improving the system performance.

Although the results shown in Figure 11 may be surprising, it should be noticed that the situation is the best case for the garbage collection because, as opposed to the case in Figure 10, all tuple spaces created become garbage very quickly. Without garbage collection, the data structure of tuple spaces and tuples become very 'heavy' and operations take longer to complete. When the garbage collector is running and collecting all the garbage, the structure is most of the time 'clean', and operations take less time to complete.

One should expect that in the average case the overhead added by the the garbage collection scheme together with information gathering system is minimum. Several factors account for these results; first the garbage collector runs as a low priority thread which tries to use the processors idle time; second, the tuple monitoring only monitors tuples which contains handles and from experience it can be said that operations involving handles are not the most common; finally, process registration and check-out only adds four extra messages (two for registration and two for check-out) between the given process and the kernel, and in the long run this adds practically nothing to the total execution time of a process.

7 Conclusion

This paper proposed an efficient way of gathering information in open Linda-like systems which can be used as the source of information for optimizations. The paper has explored how garbage collection can be implemented using the distributed data structure containing the information gathered at run-time.

Due to the way the data structure is maintained a simpler garbage collection can be used as race-conditions are dealt with at the level of the data structure construction. The results have shown that the garbage collection does not add overhead to open Linda systems, and more importantly, the information gathering system does not have influence on its overall performance.

It has been argued that this solution is expected to work well under different assumptions. One case that is worth mentioning are models in which processes may not maintain references to tuple spaces as they can use a lookup services to "discover" tuple spaces. It is important to realize that "something" must have a reference to the tuple space if it is still required. The solution proposed can easily adapted to consider lookup services as active processes with references to tuple spaces.

There is one case where the solution in this paper cannot be used. In systems such as IBM T Spaces [17], processes get access to an existing tuple space by "creating" a tuple space with the same name as the existing one. The reason the solution described in this paper cannot be used is that in systems such as T Spaces, garbage collection is not possible under the open system assumption. In fact, this "guessing" of tuple space names should be avoided because it removes the idea of local tuple spaces. Any tuple space in the systems can be accessed by other processes if its name is "guessed" by these processes. This concept compromises the security of data in the system and should be avoided.

References

[1] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[2] G. Berry and G. Boudol. The Chemical Abstract Machine. In ACM, editor, *Proceedings of the seventeenth annual ACM symposium on Principles of programming languages*, pages 81–94, New York, NY, USA, 1990. ACM Press.

[3] D. I. Bevan. Distributed Garbage Collection using Reference Counting. In W. J. Bakker, , L. Nijman, and P. C. Treleaven, editors, *Proc. of Parallel Architectures and Languages Europe*, pages 176–187. Springer-Verlag, June 1987. Lecture Notes in Computer Science, Vols. 258 and 259.

[4] L. Cardelli. Wide Area Computation. *Lecture Notes in Computer Science*, 1644, 1999.

[5] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 66–76. Springer-Verlag, Berlin, 1995.

[6] P. Ciancarini, A. Knoche, R. Tolksdorf, and F. Vitali. PageSpace: An Architecture to Coordinate Distributed Applications on the Web. In *Proc. of the 5th Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE '96)*, pages 340–345. IEEE Computer Society Press, June 1996.

[7] G. E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 31(9):1128–1138, 1960.

[8] H. Corporaal, T. Veldman, and J. V. D. Goor. An Efficient Reference Weight-based Garbage Collection Method for Distributed Systems. In *Proc. of the PARBASE-90 Conference*, pages 463–465. IEEE, 1990.

[9] A. Douglas, A. Rowstron, and A. Wood. ISETL-LINDA: Parallel Programming with Bags. Technical Report YCS-257, University of York, April 1995.

[10] 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.

[11] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, Reading, MA, USA, 1999. The Jini Technology Series.

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

[13] D. Gelernter. Multiple Tuple Spaces in LINDA. In *Proc. of PARLE 89*, pages 20–27. Springer-Verlag, 1989.

[14] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):96–107, February 1992.

[15] J. Hughes. A Distributed Garbage Collection Algorithm. In J.-P. Jouannaud, editor, *Proc. of ACM Conference on Functional Programming Languages and Computer Architecture*, pages 256–272, 1985. Lecture Notes in Computer Science, Vol. 201.

[16] S. C. Hupfer. MELINDA: LINDA with Multiple Tuple Space. Technical Report YALE/DCS/RR-766, Yale University, February 1990.

[17] IBM Corporation. *T Spaces Programmer's Guide*, 1998. Eletronic version only. http://www.almaden.ibm.com/cs/TSpaces/.

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

[19] C. A. Knoblock and J. L. Ambite. Agents for Information Gathering. In J. Bradshaw, editor, *Software Agents*. AAAI/MIT Press, Menlo Park, CA, 1997.

[20] T. W. Malone and K. Crowston. What is Coordination Theory and How Can it Help Design Cooperative Work Systems? In *Proceedings of the conference on Computer-supported cooperative work, CSCW '90*, pages 357–370, 1990.

[21] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM*, 3(4):184–195, April 1960.

[22] 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 216–225, Big Island, Hawaii, USA, 1998. IEEE Computer Society.

[23] R. Menezes and A. Wood. Garbage Collection in Linda using Tuple Monitoring and Process Registration. In *Proc. of the 10th International Conference on Parallel and Distributed Computing and Systems*, pages 490–495, Las Vegas, Nevada, USA, 1998. Acta Press.

[24] R. Menezes and A. Wood. Ligia: A Java based Linda-like Run-time System with Garbage Collection of Tuple Spaces. Technical Report YCS-304, Department of Computer Science, University of York, September 1998.

[25] J. Piquer. Indirect Reference-Counting: A Distributed Garbage Collection Algorithm. In E. Odijik, M. Rem, and J.-C. Sayr, editors, *Proc. of Parallel Architectures and Languages Europe*, pages 150–165. Springer-Verlag, June 1991. Lecture Notes in Computer Science, Vols. 365 and 366.

[26] A. Rowstron and A. Wood. BONITA: A Set of Tuple Space Primitives for Distributed Coordination. In H. El-Rewini and Y. N. Patt, editors, *Proc. of the 30th Hawaii International Conference on System Sciences*, volume 1, pages 379–388. IEEE Computer Society Press, January 1997.

[27] Scientific Computing Associates Incorporated. *Paradise – User's Guide and Reference Manual*, 1994.

[28] Scientific Computing Associates Incorporated. LINDA: An Introduction and Example. Available from Scientific Computing Associates Incorporated, One Century Tower, New Haven, CT 06510-7010, U.S.A, 1994.

[29] A. H. Sherman. *C-Linda Reference Manual.* Scientific Computing Associates, Inc., New Haven, Connecticut, 1990.

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

[31] Sun Microsystems, Inc. *Jini Architecture Specification*, January 1999. Revision 1.0.

[32] G. Sutcliffe and J. Pinakis. PROLOG-LINDA: An Embedding of LINDA in muPROLOG. Technical report, The University of Western Australia, 1991.