Garbage Collection in Open Distributed Tuple Space Systems

Ronaldo Menezes* **and Alan Wood**
THE UNIVERSITY *of York*

Department of Computer Science
Heslington, York, YO1 5DD
England, UK
{ronaldo,wood}@minster.york.ac.uk

Abstract

This paper demonstrates the need for garbage collection in multiple tuple space ***distributed open systems****, which has* LINDA *as a major icon, and identifies problems involved in incorporating garbage collection into such systems. We concern ourselves with open implementations as the existence of a garbage collector is essential in this environment.*

The extension of LINDA *to include multiple tuple spaces has introduced this new problem as processes are now able to create tuple spaces, spawn other processes into these tuple spaces, and store tuples (data) into these tuple spaces, but are unable to delete any of the objects (tuples, tuple spaces and processes) or even decide about their usefulness.*

In this paper we begin by showing that the main problem in introducing garbage collection into LINDA *is the lack of sufficient information about the effectiveness of* LINDA *objects. We then describe techniques for maintaining a structure to be used by a garbage collection algorithm of tuple spaces.*

Sumário

Este artigo demonstra tanto a necessidade de um algoritmo de "coleta de lixo" em ***sistemas abertos distribuídos*** *baseados na extensão do modelo* LINDA *para múltiplos espaços de tuplas quanto os problemas existentes em solucioná-lo. Nossa atenção é voltada principalmente para sistemas abertos, já que nesses sistemas a existência de um algoritmo para coleta de lixo é mandatória.*

A extensão do modelo LINDA *com a inclusão de múltiplos espaços de tuplas introduziu este novo problema. Este modelo extendido possibilita que processos criem espaços de tuplas, disparem outros processos e armazenem tuplas dentro desses espaços de tuplas, mas no entanto não provê meios de se deletar objetos (tuplas, espaços de tuplas e processos) ou de ajudar na decisão de sua utilidade dentro sistema.*

Neste artigo nós demonstramos que o principal problema em se incluir um algoritmo de coleta de lixo em implementacões do modelo LINDA *é a inexistência de informação sobre a utilidade de seus objetos. Sendo assim, descrevemos técnicas para a manutenção da informação requerida por qualquer algoritmo de coleta de espaços de tuplas em* LINDA.

*Research supported by Fundação Coodenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), under grant 1383/95-8.

1 Introduction

Open systems are defined as those in which processes (or active objects) can join and leave the system at any time. More accurately, in open systems the active objects do not need to be defined before run-time. Implementations of open distributed systems need to address the problems of resource management, in particular garbage collection, in order to be of practical use. The creation of garbage, is correlated with the amount of time that a system runs. Failures due to memory exhaustion are more frequent in long-running systems. Applications for open systems are, in general, intended to have a longer running time than those for closed systems which may not need garbage collection since optimisations can often be done in order to guarantee an execution without memory exhaustion.

Recent research has shown that the overall performance of LINDA open systems can compete and in some cases beat the overall performance of LINDA closed systems [Row96]. In other terms, this shows that open systems have the potential to replace closed systems assuming the main role in the LINDA world. However, they must provide some level of memory recycling, to avoid exhaustion, so as to take this role.

Several open implementations of the LINDA coordination model have been proposed [Jen89, NS93, CSS94, ADFS94, DWR95, RW96a, Row96], but most of them have overlooked the problems related to garbage collection which may occur when open models are assumed. Those which have not, [Jen89, NS93], restrict themselves in talking about the need for a garbage collector.[1] Although these implementations suffer from this problem they have not yet shown the symptoms because they have not been used in large scale applications.

This paper focuses on the garbage collection of tuple spaces, but as processes are spawned within tuple spaces, we show that processes can sometimes be garbage collected as well. In multiple tuple space LINDA implementations, once a tuple space is created it is potentially available to all processes. Furthermore, since these implementations do not provide information about which process requires which tuple space, the proper implementation cannot use simple rules like scopes to delete them. A garbage collector is therefore made necessary in order to decide on an object's status (garbage or not) by looking at its situation in the entire system.

In this paper we show that the main problem of doing garbage collection in open LINDA implementations is that the available information about their objects is insufficient. We show how to maintain a structure with all necessary information for a garbage collection algorithm and the problems involved in such maintenance. We also propose a simple possible solution based on this structure.

We do not intend to show a novel method of doing garbage collection. Our main goal is to clarify the need for a garbage collector, and show how to solve the problems involved in maintaining a structure with the information required by *any* garbage collector. The structure we maintain uses a new idea of *bridges* (directed arcs) to avoid some of the race-condition situations.

We begin this paper by describing the multiple tuple space LINDA coordination model and the idea of garbage collection, respectively in sections 2 and 3. In Section 4 we show how the information necessary for the garbage collection is maintained. In Section 5 we define the ***out***-

[1] Kaashoek *et Al.* [KBT89], in their case study, have also pointed out the need for a garbage collector in LINDA.

termination problem and show why it must be solved. Then, in Section 6 we sketch a simple solution for garbage collection based on the proposed structure. In Section 7 we make some final remarks and give our conclusions.

2 The Linda Coordination Model

A coordination language is an embedding of a coordination model which provides operations to create processes and to support communication among them [GC92].

A Linda-based coordination language is an embedding of the Linda coordination model into some computational language, enabling the embeddings to coordinate tasks. The Linda coordination model [Gel85], which is based on the concept of tuple space communication, unifies the concept of process creation, communication and synchronisation, since all these concepts are implemented using tuple space operations.

The tuple space communication model consists of associative shared memories (tuple spaces) which are able to store *tuples*. Processes can store and retrieve tuples from tuple spaces, but are unable to communicate with each other directly. The retrieving of tuples uses an associative matching mechanism based on *templates* (anti-tuples) which, like tuples, are ordered sequences of typed objects. The templates differ from the tuples because fields can be non-valued (holes) represented by *?type* or *?typed-variable*. For instance the template *[?int, "Hello"]* matches the tuple *[1,"Hello"]*.

A tuple space is an abstraction of a shared memory. We must see tuple spaces as *bags* or *multi-sets* of tuples, where each tuple does not know anything about the existence of the others.

The multiple tuple space Linda model [Gel89] is an evolution of the single tuple space model in which processes have access to a unique tuple space representing the whole associative memory [Gel85]. In the multiple tuple space model, the associative memory is composed of a set of tuple spaces which can be created by processes in order to store data.

The model provides primitives to store, read and withdraw tuples from tuple spaces, to create tuple spaces and to spawn processes:

out(TSm,tuple): Stores the *tuple* within the tuple space *TSm*.

in(TSm,template): Withdraws a tuple from the tuple space *TSm* which matches the *template*. If there is more than one tuple which matches the *template* one is chosen non-deterministically, but if there is no such a tuple the process is blocked waiting for one.

rd(TSm,template): Same as *in* but non-destructive, that is, the tuple is copied as opposed to withdrawn.

inp(TSm,template): Non-blocking version of the *in*. If there is not a tuple to match the *template* the primitive returns *false* and the process carries on.

rdp(TSm,template): Same as *inp* but non-destructive.

n = collect(TSm,TSq,template): Bulk primitive which moves all tuples matching the *template* from the tuple space *TSm* to the tuple space *TSq* and assigns to *n* the number of tuples moved [BWA94].

`n = copy-collect(TSm,TSq,template)`: Bulk primitive which copies all tuples matching the `template` from the tuple space `TSm` to the tuple space `TSq` and assigns to `n` the number of tuples copied [RW96b].

`eval(TSm,tuple)`: The LINDA way of spawning processes. Processes are created to evaluate the elements of the `tuple` in parallel.

`handle = tsc()`: Creates a new local tuple space and assigns a unique `handle`.

Although we are considering some primitives not present in all LINDA implementations, the work described in this paper applies to the LINDA coordination model in general and does not consider any particular implementation.

In Figure 1 we define a *boundary* between LINDA and the system. This boundary is intended to separate, from the users' point of view, what is LINDA and what is not. The view is that LINDA processes (represented by squares) can only operate with tuple spaces (represented by ovals). We define a special tuple space called `I/O` which is the tuple space destination of every I/O operation so as to make this separation complete. After the data is stored within the `I/O` tuple space, some system process (represented by circles), which would be part of the LINDA kernel, have to retrieve the data and output to the correct device. Another special tuple space in our model is the *Universal Tuple Space* (`UTS`) which is the tuple space within which the main processes are spawned. Besides `UTS` and the `I/O` tuple spaces, no other LINDA object can be linked to the boundary.

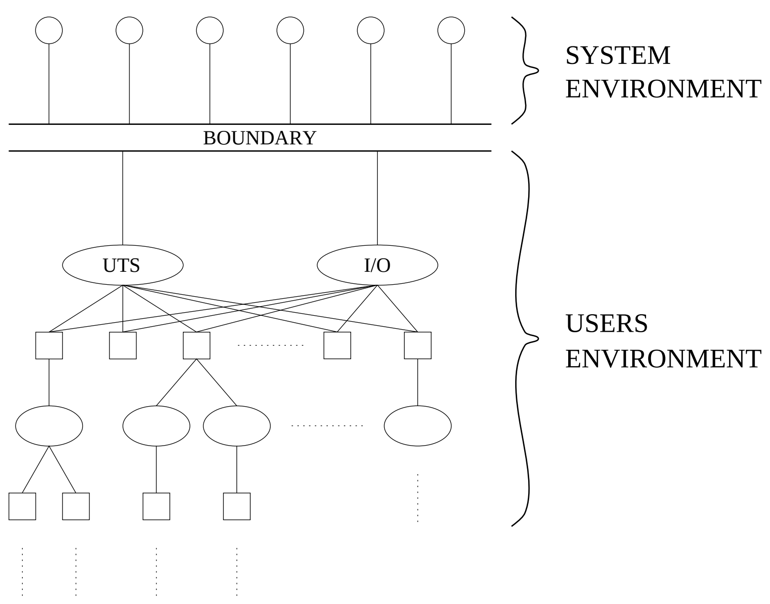


Figure 1: Boundary of the LINDA model.

Every LINDA implementation assumes different characteristics, in this paper we assume implementations having the following main characteristics:

- There is a ***UTS***.[2]
- I/O operations are done solely in terms of tuple space operations.
- Tuple spaces are *not* first-class objects, although handles (references to tuple spaces) are.
- Every tuple space is created at the same level, that is, tuple spaces cannot be created inside others.
- Every process may operate with the tuple space which contains it, that is, the tuple space specified in the ***eval*** operation which was used to spawn the process.
- The *main* processes, those spawned by an external entity, are assumed to be spawned within ***UTS***.
- We consider two different concepts in this paper in terms of accessibility of the ***UTS*** and ***I/O*** tuple spaces by processes:

  1st case: Every process has access to both of them.

  2nd case: In principle only the main processes have access to them, although their handle can be passed to other processes.

From the characteristics itemised above, the only one that is not present in any LINDA implementation yet is the one saying that I/O is done solely in terms of tuple space operations. However, most of the LINDA implementations are more restrictive than this forbidding all the processes, except the main ones, to do I/O. Still we believe that I/O must be incorporated into LINDA implementations [MW98].

3 Garbage Collection

In distributed systems, storage management should not be left to be dealt with by users. In distributed environments (including LINDA), users are unable to know whether a memory cell is being used by others in the system. As a matter of fact, if the task of reclaiming cell was left to users, the creation of *dangling references* [JL96] would be common.

Garbage collection is the process of searching and reclaiming unused memory objects automatically. These two operations (search and reclaim) are often executed in separate phases. The search phase can be considered the core of garbage collection since there is nothing special in making unused space available again.

Garbage collection algorithms are based on a *tracing graph* representing the memory. By performing a traversal of the graph or analysing its cells (memory locations) it is possible to decide on their usefulness.

Generally speaking, there are three basic ideas for searching for garbage:

Mark-and-Sweep: A search is performed in the graph starting from special nodes (roots). A node is still useful if it is reachable from the roots but garbage otherwise [McC60].

[2] Some implementations call it Global Tuple Space (***GTS***).

Reference Counting: Each cell has a count of the number of nodes pointing to it, consequently every cell with counter 0 (zero) is garbage [Col60].

Reference Listing: Each cell has a list of cells pointing to it. If a cell's list is empty, it is garbage [SDP92].

As we said before, all garbage collection ideas are based on either a data structure, or on counting/listing the references for a cell. However, LINDA provides neither a data structure of its tuple spaces nor references to them.

We shall not go into the details of these searching algorithms for it should be clear that our problem is to maintain the necessary information to be used by a searching algorithm and not to choose a suitable searching algorithm for LINDA. For good surveys on garbage collection see [Coh81, PS94, Wil94, JL96].

4 Supporting Garbage Collection Within Linda

The goal of this section is to describe how we intend to maintain in LINDA the information required by any garbage collector. Existing LINDA implementations do not provide any information that can be directly used by a garbage collector. Thus, we propose a way to maintain a data structure (a graph) with all information necessary to a garbage collector.

In Section 2 we described how every LINDA process is (implicitly) linked to a tuple space since LINDA's way of spawning processes requires such a link. Additionally, we know that every tuple space, except ***UTS*** and ***I/O*** which represent the boundary of the model (Figure 1), has to be created by a process.

These two characteristics will be used to construct one graph representing processes and tuple spaces where, in principle, processes are linked only to tuple spaces and vice-versa (although this will be modified by the introduction of *bridges* in Section 4.1). The idea behind the graph is that the tuple spaces linked to a process ***Pn*** represent those to which ***Pn*** has access; and the processes linked to a tuple space ***TSm*** represent those with knowledge of ***TSm***. This graph structure follows the common tracing graph concept as used in most garbage collection methods: every object unable to reach the boundary by following its links is garbage. Since the only objects linked to the boundary are ***UTS*** and ***I/O*** we can say that an object is garbage if it is unreachable from both ***UTS*** and ***I/O*** and cannot reach all of them.

The structure maintains the invariant that once an object is unreachable it cannot become reachable again, in other words, once an object becomes garbage it cannot became useful again. This invariant guarantees that no erroneous collection occurs.

Although in this paper performance issues are not being considered, one can argue about where such graph is supposed to be stored. For now we can assume that such structure can be maintained in the LINDA kernel, but it should be clear that this assumption should be revisited when implementing a solution since there are other possibilities including even having this structure stored inside a tuple space.

4.1 The Basic Structure

As we said in Section 2, we consider in this paper two different cases of Linda models with different concepts. Despite the differences, we can describe rules for constructing the graph structure that apply for both cases:

1. There are two nodes representing the *UTS* and the *I/O* tuple spaces.
2. All main processes are linked to both *UTS* and *I/O* nodes.
3. There is a unique node representing each tuple space.
4. There is a unique node representing each process.
5. A process has to be linked to every tuple space of its knowledge. A process can acquire knowledge of a tuple space in different ways:
   - When it is spawned into one. Every process knows about the tuple space it is spawned into.
   - When it creates a tuple space.
   - When it receives a handle as a parameter when it is spawned.
   - When it retrieves a handle (as an element of a tuple) from a tuple space.
6. A tuple space has to be linked to another directly using a direct arc (*bridge*) if the handle of one is stored in the other.

The direction of the bridges will depend on the *strength* of the class of the tuple spaces involved in the storing. For our purposes there are two classes of tuple spaces:

Strong class: Composed of *UTS* and *I/O* only.

Weak class: Composed of all the other tuple spaces.

In the following sections we explain in detail each of the points of graph construction, and how the classification above may be used to define the direction of the bridges.

4.2 Main Processes

In both models considered in this paper, the main processes have to be linked to both *UTS* and *I/O* nodes. The link to the *UTS* node represents the assumption that every main process is spawned into *UTS*; and the link to the *I/O* node represents the assumption that at least the main processes are able to do I/O.

Figure 2 shows the graph situation when two main processes are running in a Linda environment. The part just added in the graph is showed in dashed lines.

The processes' unique names assigned to the nodes are given by the kernel when the processes register themselves. The Linda kernel must have knowledge of all processes in the system and this is achieved by having a process registration scheme that occurs just before a process starts to run.

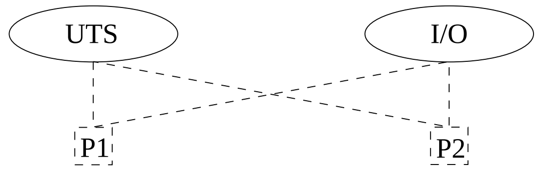


Figure 2: Two main processes, *P1* and *P2*, running in a LINDA environment.

4.3 Creation of Tuple Spaces

As described in Section 4.1, every creator process has knowledge of its created tuple spaces; or in other words, every tuple space is liked to its creator. Figure 3 shows the graph situation just after the creation of two tuple spaces by process *P1* and one tuple space by process *P2* (starting from Figure 2).

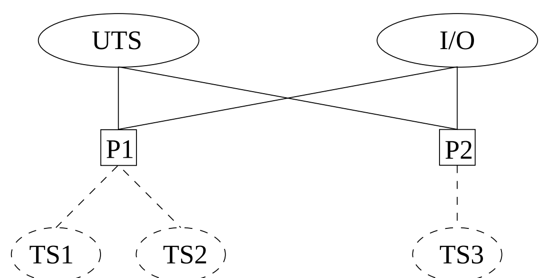


Figure 3: Creation of tuple spaces. A tuple space has to be linked to its creator.

Similarly to the processes case, the tuple spaces' unique names are given by the kernel upon the creation of them. However in this case, the users are aware of these names for they are tuple space handles.

4.4 Spawning of Processes

The LINDA model provides a primitive (*eval*) to spawn processes. In the graph, spawned processes must be represented individually since the garbage collection has to decide on the usefulness of a process individually.

In the multiple tuple space model, the spawning of processes requires a containing tuple space to be defined in the primitive. Again, as explained in Section 4.1, every process knows about its containing tuple space and therefore they are linked in the graph.

Suppose that starting from the situation in Figure 3, process *P1* spawns processes *P3* and *P4* into *TS1* and *UTS* respectively, and process *P2* spawns process *P5* into *TS3*. Figure 4*(a)* shows the situation considering the 1st case in terms of accessibility to *UTS* and *I/O* and Figure 4*(b)* considering the 2nd (as defined in Section 4.1).

From now on the differences of the two cases start to appear in the graph since processes other than main processes are being represented.

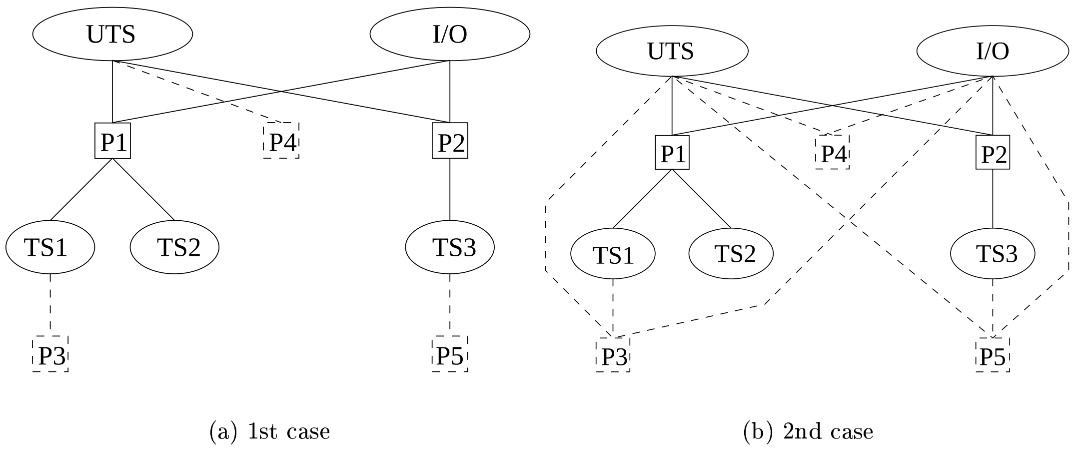


Figure 4: Spawning of processes with different assumptions in terms of accessibility to *UTS* and *I/O* tuple spaces.

4.5 Further Ways of Processes to Acquire Information

In addition to the creation of tuple spaces and the spawning operations, there are two other ways a process can acquire information about a tuple space as enumerated in Section 4.1. The next two sections give more details about these operations.

4.5.1 Passing Handles as Parameters

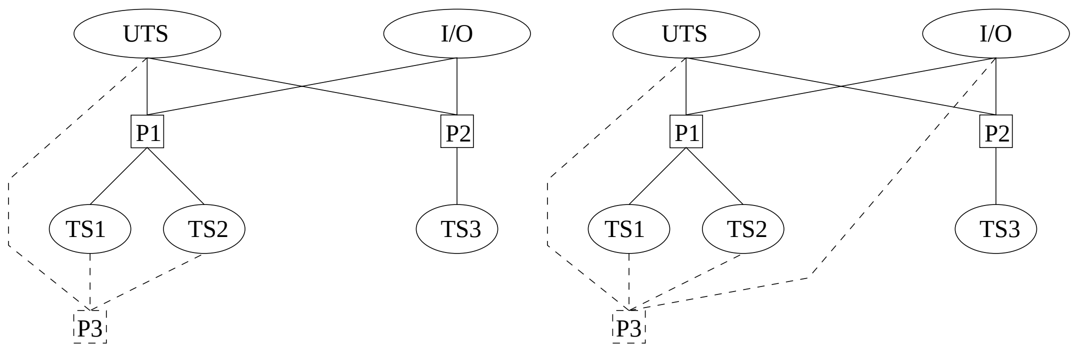
Tuple space handles are first-class objects and can therefore be passed as parameters. For instance, suppose the description of a process *F* below:

```
void F (tsh my_ts)
{
   int number;
   in(my_ts,[?number]);
   do_something();
   number++;
   out(my_ts,[number]);
}
```

The process above receives the handle *my_ts* as a parameter. Then when it is spawned it starts knowing about the tuple space referred to by *my_ts*.

Depicting a similar situation, Figure 5 shows the graph, in both cases, when process *P1* spawns a process *P3* into *TS1* passing the handle of *TS2* and *UTS* (starting from the situation in Figure 3).

It should be noticed here that with the 2nd case, all spawned processes already know about *UTS* and *I/O*, consequently passing the *UTS* handle does not alter the graph.



(a) 1st case (b) 2nd case

Figure 5: Spawning of *P3* into *TS1* passing the handle of *TS2* and *UTS*.

4.5.2 Passing Handles Through Tuple Spaces

Handles, like any other first-class object, may be stored within tuple spaces. Once a handle of *TSm* is stored into a tuple space, it may be retrieved by another process which then becomes able to access *TSm*. This makes it possible to pass information about local tuple spaces.

Starting again from the situation described in Figure 3 where process *P1* has created two local tuple spaces, it is perfectly possible to reach a situation where process *P1* writes the handle of *TS2* into *UTS* (executing for instance, `out(UTS,[TS2])`) and *P2* retrieves the information from *UTS* (executing for instance, `in(UTS,[?handle])`, where `handle` is a variable of the type `tsh`) becoming able to access *TS2*.

Assuming exactly the operations above, the situation just after *P2* withdraws the handle of *TS2* from *UTS* is depicted in Figure 6.

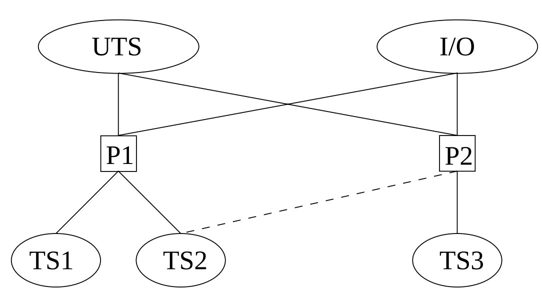


Figure 6: Process *P2* withdraws a handle and acquires knowledge of *TS2*.

4.6 Constructing Bridges

Assuming the same operations described in section 4.5.2, if process *P1* terminates before process *P2* gets the handle of *TS2*, any garbage collector could collect *TS2* before the actual retrieving of *TS2* by *P2* as the graph would not represent the real situation.

Since LINDA implementations are asynchronous environments and the passing of handles between processes through tuple spaces is not an indivisible operation, we cannot guarantee race-condition free operations.

Therefore, in spite of what was described in the previous sections, it is still necessary to make a further modification in the graph to avoid a race-condition between the garbage collector and the algorithm that maintains the graph.

In the context of garbage collection if a handle of *TSm* is stored into a tuple space *TSq* we have a dependency between these tuple spaces in the sense that *TSm* is only garbage if *TSq* is garbage. To represent such a property in the graph we create a *bridge* linking these two tuple spaces. This bridge is a direct arc and is called bridge because it shortcuts the original path(s) between these tuple spaces.

A bridge has to be created following certain rules. Assuming that the handle of *TSm* is being stored into *TSq*, we have:

- If *TSm* and *TSq* are from different classes (as defined in Section 4.1) then the bridge is always from the tuple space belonging to the stronger class to the one belonging to the weaker. This means that every time that either *UTS* or *I/O* is involved in the operation of storing handles, the bridge is always outgoing from them.

- If *TSm* and *TSq* are from the same class, then the bridge goes from the containing tuple space to the contained. This conveys the correct idea that every process with access to tuple space *TSq* has *potential* access to *TSm*, and therefore *TSm* cannot be collected unless *TSq* becomes garbage.

Every bridge has an attribute that we will call *width* meaning the number of corresponding handles stored in the containing tuple space. Every bridge is created having width 1 but may become wider or narrower depending on the operation executed (storing or withdrawing). The width of a bridge from *TSm* to *TSq* is the number of handles of *TSq* stored into *TSm*.

Figure 7 shows the graph situation, starting from Figure 3, after all the following operations with handles:

- A handle of *TS2* is stored by process *P1* into *TS1*. The bridge goes from *TS1* to *TS2* since they are from the same group.

- A handle of *TS1* is stored by process *P1* into *UTS*. The bridge goes from *UTS* to *TS1* since *UTS* is the stronger tuple space.

- A handle of *UTS* is stored by process *P2* into *TS3*. The bridge goes from *UTS* to *TS3* since *UTS* is the stronger tuple space.

- Another handle of *UTS* is stored by process *P2* into *TS3*. The bridge becomes wider; its width goes to 2.

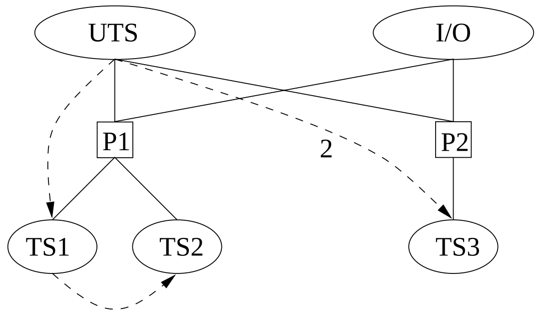


Figure 7: Creating of bridges according to the strength of the tuple spaces.

4.7 Breaking Links

To complete our description we must show how arcs are deleted from the graph. First of all let us consider the simplest case: deletion of undirected arcs. When a process terminates, its representation in the graph is deleted together with all arcs with an end linked to it. Figure 8 shows in dotted lines everything deleted when process *P1* terminates (starting from the situation in Figure 6).

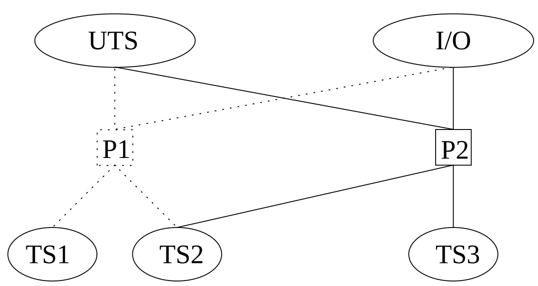


Figure 8: Process *P1* terminates; its representation and the arcs linked to it are deleted.

As every tuple space has the potential of being a global object, tuple space *TS1* in Figure 8 cannot be deleted by the process *P1* when it terminates because no process is able to decide on the usefulness of a tuple space — this would have to be done by a garbage collector.

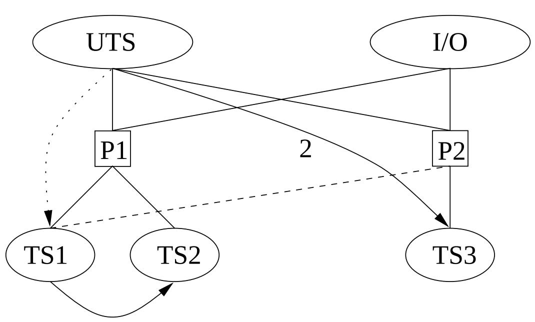
Having described the case above, the only point remaining now is the deletion of bridges. In principle a bridge linking two tuple spaces has to remain while the dependency between these two tuple spaces remains. Two cases have to be considered:

- A handle is withdrawn using the primitive *in*. If a process *in*s a handle, the bridge has to become narrower. If its width becomes zero then the bridge disappears (it is deleted).
- More care is necessary when a *collect* is used as *collect* is a bulk primitive. Consider the tuples which match the *template* when a *collect* is executed. It is clear that each one of these tuples would match the same *template* if it were used with an *in* primitive. Hence, for the purpose of updating the bridge's width, we could view *collect* as a sequence

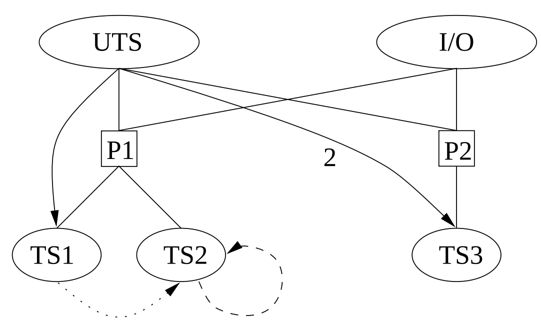
of *in*s in the source tuple space followed by a sequence of *out*s to the destination tuple space.[3] One important characteristic is that we assume that the primitive *collect* is indivisible and therefore it is not possible for a tuple space to be garbage collected in the middle of a *collect* execution.

Observe that in both cases above if an *in* is executed in a tuple space *TSm* we have only to look at the outgoing bridges of *TSm* and only one incoming bridge (if there is such a bridge) which represents the storing of *UTS* handles into *TSm*.

Starting from the situation shown in Figure 7, Figure 9*(a)* shows a very simple case where process *P2* *in*s the handle of *TS1* from *UTS* and Figure 9*(b)* shows the result of an execution of a *collect* by the process *P1* from *TS1* to *TS2*. Observe that the *collect* execution generated in Figure 9*(b)* a self-bridge in the tuple space *TS2*. This self-bridge can be deleted by the garbage collector or even not created since it is clear that every tuple space has an implicit bridge to itself.



(a) Process *P2* executed *in(UTS,[?tsh])* and got *TS1*



(b) Process *P2* executed *collect(TS1,TS2,[?tsh])*

Figure 9: Modification in the bridges.

5 The *out*-termination Problem

In the context of garbage collection, the termination[4] of processes plays the principal role since this is the only way of generating garbage. One of the main requirements of our proposed garbage collection method is what we call *out-termination ordering*. The problem here is to guarantee the two basic properties below:

- The LINDA kernel is aware of processes' termination. We are not considering how this can be done but it must be clear that it is possible. For instance, a new primitive could

[3] We can make a corresponding assumption with *copy-collect*.

[4] Because the term termination can mean different things we should make clear that the termination occurs after the last instruction of a process — we are not considering external intervention.

be introduced or the language where the LINDA is embodied could inform the kernel. In either case, we assume that a termination message is sent to the kernel.

- The termination message and the instructions ***out***, ***collect***, ***copy-collect*** and ***eval*** have to be delivered to the kernel in the order in which they occured. This may be guaranteed in a similar way as done by Douglas *et Al.* [DWR95] when describing the ***out***-ordering problem.

Despite being named ***out***-termination, the ***out*** primitive is not the only one affected by this problem, any primitive with the capacity to store information in a tuple space must be included in the ordering. We show now why these primitives have to be delivered in temporal order with the termination.

Consider the process described below. The last instruction is one of the primitives after the brace depending on the case we describe. Notice that the temporal order is: primitive, then termination.

```
P1:
   :
   out(UTS,[TS2]);
   eval(UTS,[TS2]);
   collect(TS1,UTS,[?tsh]);
   copy-collect(TS1,UTS,[?tsh]);
end;
```

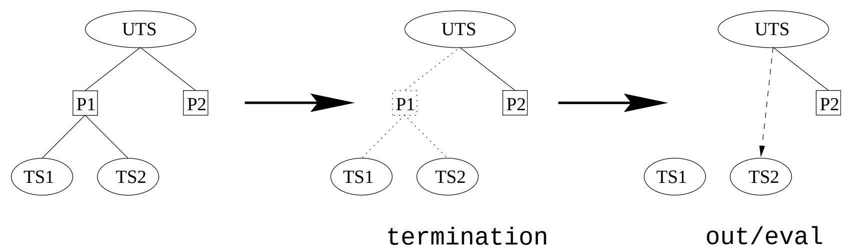
Assuming the primitive is either ***out*** or ***eval***, Figure 10*(a)* shows the situation in the graph when this temporal order is not maintained and Figure 10*(b)* shows the correct situation.

It must be noticed that in the case shown in Figure 10*(a)* the garbage collector may erroneously collect ***TS2*** even though it is still required. Hence, the ***out***-termination problem applies to the ***out*** and ***eval*** primitives.

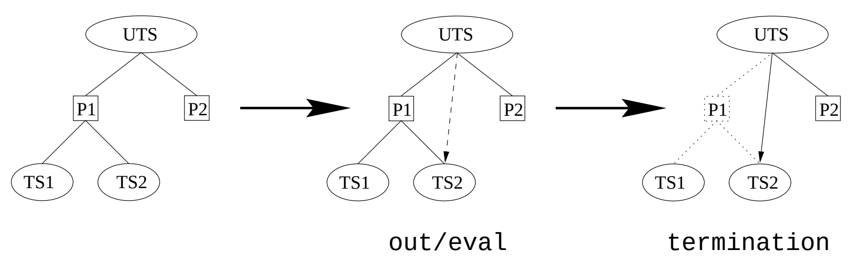
Considering now the primitive ***collect***, Figure 11*(a)* shows the situation in the graph when this temporal order is not guaranteed and Figure 11*(b)* shows the correct situation. Since a similar situation to the ***out*** case is shown in Figure 11*(a)* the ***out***-termination problem applies for the ***collect*** primitive.

Finally, we consider the primitive ***copy-collect***. As shown in Figure 12 the situation is quite similar to the ***collect*** case. The only difference in this case is that the bridge from ***TS1*** to ***TS2*** remains. Although the bridge remains, if the temporal order is not guaranteed both ***TS1*** and ***TS2*** will be garbage collected. Another point not related to the ***out***-termination problem but shown in Figure 12*(b)* is that after ***P1*** terminates, the tuple space ***TS1*** is garbage for it is not reachable from the roots. The direction in the bridges guarantees this.

It is easy to see that the order of the primitives ***in***, ***rd*** and ***tsc*** are not significant. Since these primitives are blocking primitives, the next instruction is only executed after a response from the kernel.



(a) Situation when the temporal order is changed.



(b) Situation when the temporal order remains.

Figure 10: Situation showing that the *out*-termination problem applies for the primitives *out* and *eval*.

6 A Solution for Garbage Collection

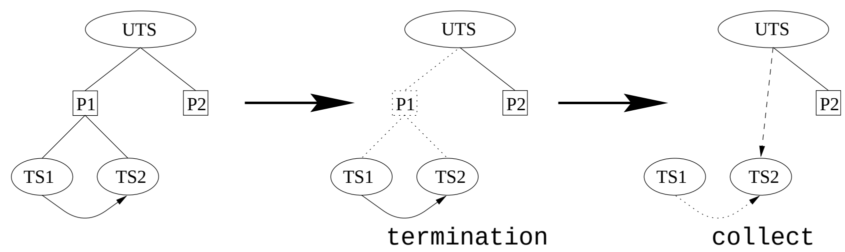
The concept of reachability from the roots (*I/O* and *UTS*) in the proposed graph, makes it possible to find garbage in LINDA-like systems. The rule is very simple: an objects is garbage if it is unreachable from the roots and cannot reach all of them. From time to time a traversal in the graph is made and all reachable nodes are marked; the garbage cells are those not marked at the end of the traversal.

Observe that if this traversal in the graph has to be executed concurrently with the LINDA processes, we should use a variation of the marking algorithm as proposed by Dijkstra *et Al.* [DLM+78].

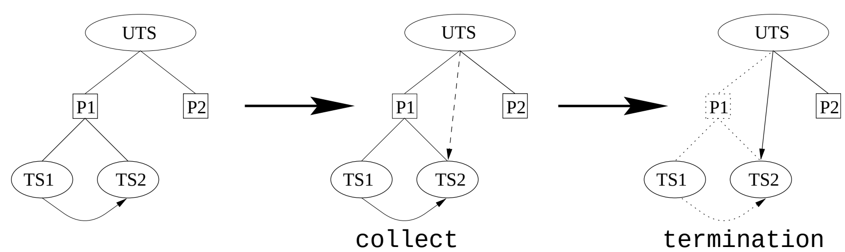
The reason for proposing the use of an algorithm based on mark-and-sweep is due to the possibility of cycles in the graph, and mark-and-sweep works well with cycles.

7 Conclusion

Implementation is always a concern. At first glance, the proposal of a non-distributed solution for garbage collection may appear strange as distributed systems usually require distributed solutions. The LINDA case is different, though. The central elements of our proposed garbage collection scheme are tuple spaces and in LINDA a single tuple space might not be stored in



(a) Situation when the temporal order is changed.



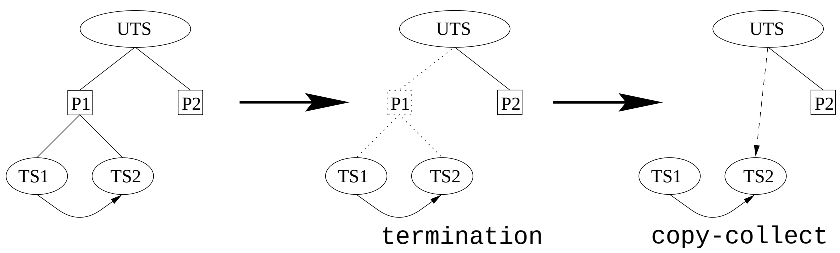
(b) Situation when the temporal order remains.

Figure 11: Situation showing that the *out*-termination problem applies for the primitive *collect*.

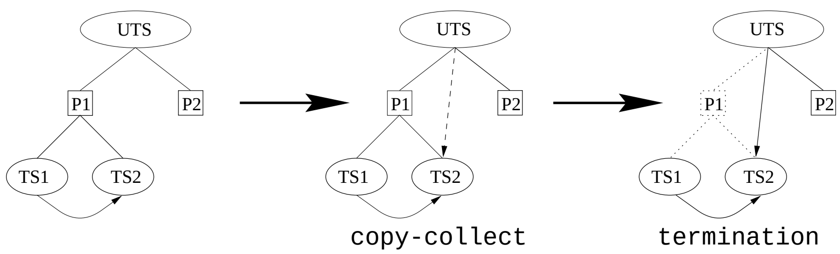
a single location. Tuple space implementations are generally done by having their contents distributed and therefore a single tuple space might be partitioned. Since this is the case, it is not absolutely clear that the application of an existing distributed garbage collection method (assuming non-partitioned objects) would produce better performance. If we distribute our graph in order to use some distributed garbage collection algorithm, we will be forced to arbitrarily choose a location for the nodes representing tuple spaces, but the question then is "how can we make a *realistic* choice since tuple spaces are partitioned?". In addition to this, the LINDA processes (the mutators in the garbage collection algorithm) may move from one location to another and we might also be unable to choose proper locations to the nodes representing them.

For instance, a process *P1* located in *A* is mutating the graph located in *B* and a process *P2* located in *B* is mutating the graph located in *A* and therefore every communication is remote. In our proposed implementation if *P1* and *P2* are in different locations, we assume that the single graph is stored in either of the two locations and therefore one process will have local communication. However, we are currently working on a distributed implementation, and our intention is, making use of LINDA characteristics, to propose a novel form of distributed structure. Such a proposal will have to take into account all LINDA's properties in order to active good performance.

Regarding the restrictions we have imposed in Section 2, two in particular will need further



(a) Situation when the temporal order is changed.



(b) Situation when the temporal order remains.

Figure 12: Situation showing that the ***out***-termination problem applies for the primitive ***copy-collect***.

exploration. The assumption that all I/O operations are done via tuple spaces is not well understood. We believe that this is a feasible assumption and we are currently working on a proposal for it — some early results are presented in [MW98]. The other interesting point is having tuple spaces themselves as first class objects. We are still to address the effects of this on our proposal for garbage collection but we expect that the concept of bridges may be extended to deal with this situation in addition to have only tuple space handles as first class objects.

The last point we intend to address is the problem of deadlock of processes in LINDA. Even though our method often garbage collects deadlocked processes, in some cases the problem remains. We intend to use the structure described in this paper to find out about process deadlock in a LINDA environment.

This paper has shown how a structure containing the information required by a garbage collector may be maintained asynchronously in LINDA systems. Two different LINDA models were used to show examples in this paper.

8 Acknowledgements

The authors would like to thank Antony Rowstron for his useful comments on the LINDA side of this work. Ronaldo Menezes would also like to thank CAPES for their current financial support.

References

[ADFS94] Henning Andersen, Jan M. Due, Peter D. Fabricius, and Flemming Sørensen. Ariadne: A Further Development. Technical report, University of Aalborg, Institute for Electronic Systems - Department of Mathematics and Computer Science, 1994.

[BWA94] Paul Butcher, Alan Wood, and Martin Atkins. Global Synchronisation in c Linda. Concurrency: Practice and Experience, 6(6):505–516, 1994.

[Coh81] Jacques Cohen. Garbage Collection of Linked Data Structures. Computing Surveys, 13(3):341–367, September 1981.

[Col60] G. E. Collins. A Method for Overlapping and Erasure of Lists. Communications of the ACM, 31(9):1128–1138, 1960.

[CSS94] João Carreira, Luis Silva, and João Gabriel Silva. On the design of Eilean: A Linda-like library for MPI. In IEEE, editor, 2nd Scalable Parallel Libraries Conference, October 1994.

[DLM+78] 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.

[DWR95] Andrew Douglas, Alan Wood, and Antony Rowstron. Linda Implementation Revisited. In Patrick Nixon, editor, Proc. of the 18th World Occam and Transputer User Group, pages 125–138. IOS Press, April 1995.

[GC92] David Gelernter and Nicholas Carriero. Coordination Languages and their Significance. Communications of the ACM, 35(2):96–107, February 1992.

[Gel85] David Gelernter. Generative Communication in Linda. ACM transactions in programming languages and systems, 1:80–112, 1985.

[Gel89] David Gelernter. Multiple Tuple Spaces in Linda. In Eddy Odijik, M. Rem, and Jean-Claude Sayr, editors, Proc. of PARLE 89, pages 20–27. Springer-Verlag, June 1989. Lecture Notes in Computer Science.

[Jen89] Keld K. Jensen. Towards a Multiple Tuple Space Model. PhD thesis, Aalborg University, Institute for Electronic Systems - Department of Mathematics and Computer Science, August 1989.

[JL96] Richard Jones and Rafael Lins. Garbage Collection — Algorithms for Automatic Dynamic Memory Management. Wiley, 1996.

[KBT89] M. Frans Kaashoek, Henri E. Bal, and Andrew S. Tanenbaum. Experiences with the Distributed Data Structure Paradigm in Linda. In Proceedings of the Workshop on Experiences with Distributed and Mulitprocessor Systems, pages 175–192, Fort Lauderdale, FL, USA, October 1989. USENIX Association.

[McC60] John McCarthy. Recursive Functions of Symbolic Expressins and Their Computation by Machine. Communications of the ACM, 3(4):184–195, April 1960.

[MW98] Ronaldo Menezes and Alan Wood. Coordination of Distributed I/O in Tuple Space Systems. Submitted to HICCS '98, 1998.

[NS93] Brian Nielsen and Tom Sørensen. Implementing LINDA with Multiple Tuple Spaces. Technical report, Aalborg University, January 1993.

[PS94] David Plainfossé and Marc Shapiro. A Survey of Distributed Garbage Collection Techniques. Technical Report Research Project 6360, INRIA, November 1994.

[Row96] Antony Rowstron. Bulk Primitives in LINDA Run-time Systems. PhD Thesis, Department of Computer Science – University of York, 1996.

[RW96a] Antony Rowstron and Alan Wood. An Efficient Distributed Tuple Space Implementation for Networks of Workstations. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, Euro-Par'96, volume 1123 of Lecture Notes in Computer Science, pages 510–513. Springer-Verlag, 1996.

[RW96b] Antony Rowstron and Alan Wood. Solving the LINDA Multiple `rd` Problem. In Paolo Ciancarini and Chris Hankin, editors, Proceedings of 1st. International Conference - COORDINATION 96, pages 357–367, 1996. Published in Lecture Notes of Computer Science Vol. 1061.

[SDP92] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP Chains: Robust, Distributed References Supporting Acyclic garbage Collection. Technical Report 1799, INRIA, November 1992.

[Wil94] Paul R. Wilson. Uniprocessor Garbage Collection Techniques (Long Version). Submitted to ACM Computing Surveys. Also as a Technical Report, University of Texas, 1994.