Finer Garbage Collection in LINDACAP

Nur Izura Udzir, Universiti Putra Malaysia, Malaysia
Hamidah Ibrahim, Universiti Putra Malaysia, Malaysia
Silesi Demesie, Bahir Dar University, Ethiopia

ABSTRACT

As open systems persist, garbage collection (GC) can be a vital aspect in managing system resources. Although garbage collection has been proposed for the standard LINDA, it was a rather course-grained mechanism. This finer-grained method is offered in LINDACAP, a capability-based coordination system for open distributed systems. Multicapabilities in LINDACAP enable tuples to be uniquely referenced, thus providing sufficient information on the usability of tuples (data) within the tuple-space. This paper describes the garbage collection mechanism deployed in LINDACAP, which involves selectively garbage collecting tuples within tuple-spaces. The authors present the approach using reference counting, followed by the tracing (mark-and-sweep) algorithm to garbage collect cyclic structures. A time-to-idle (TTI) technique is also proposed, which allows for garbage collection of multicapability regions that are being referred to by agents but are not used in a specified length of time. The performance results indicate that the incorporation of garbage collection techniques adds little overhead to the overall performance of the system. The difference between the average overhead caused by the mark-and-sweep and reference counting is small, and can be considered insignificant if the benefits brought by the mark-and-sweep is taken into account.

Keywords: Capabilities, Garbage Collection, LINDA Coordination, Multicapabilities, Time-to-Idle

INTRODUCTION

The LINDA coordination model (Gelernter, 1985; Gelernter, 1989) offers an alternative to the conventional point-to-point communication framework with regard to coordinating and synchronising agents’ activities. The shared data space known as tuple-spaces (TSSs) provides a medium for communication and facilitates the coordination among the interacting agents—agents communicate with each other via the tuple-space where they write and retrieve data (known as tuples). The clear separation between the coordination and the computation concerns relieves the agents of the messy aspects of communication, leaving them free to concentrate their time and space for other more crucial aspects of computation. This paradigm allows for anonymous interaction between agents separated in time and space: communicating agents need not know each other’s identity, and also the data can be retrieved any time after it has been placed in the tuple-space.
In the original Linda model, there are three primitive operations which enable agents to manipulate the tuple-space: \texttt{out} to write a tuple, \texttt{rd} to perform read and \texttt{in} to read and remove the data. The tuples are retrieved associatively in a non-deterministic fashion: the retrieval of a tuple may return any matching tuple; and, if a number of agents are waiting for a tuple of the same template, a matching tuple, when available, may be given to any one of them. Interacting via the TS where there is no direct communication between them, the communicating agents are decoupled in name, space and time—they need not know each other’s identity, nor co-exist at the same time in order to communicate with each other—providing a flexible coordination mechanism suitable for open, heterogeneous systems. Linda’s popularity is shown in its commercial variants such as Sun’s JavaSpaces (Freeman, 1999) and IBM’s TSpaces (Wyckoff, 1998).

In open implementations of Linda, agents or active entities can join and leave the system at anytime: they do not need to be defined at compile time. Agents which are not executing at the same time, can communicate via tuplespaces. An agent can also store a tuple in a tuple-space to be used by some other agents at any time in the future. In general, agents running on different machines and operating systems can be compiled separately, and written in different languages. On the other hand, in closed implementations, all agents that wish to interact must be known at compile time. The information derived at compile time can be used to control the system and optimize the application in the best possible way, including when to remove which object when it becomes unnecessary. Applications for open systems are, in general, intended to have a longer running time than those for closed systems. In Linda’s open system, tuples created are never removed from the system as it is difficult to decide if the tuples are still in-use or otherwise. As memory is a finite resource, this may lead to system failure due to memory exhaustion—the system will eventually run out of memory space if unusable objects are not reclaimed. Therefore, the implementations of open Linda systems need to address the problem of memory management in order to be of practical use. One way of managing memory is by means of garbage collection (GC), which attempts to reclaim memory that is no longer used by an application.

The garbage collection mechanism has been implemented in Ligia (Menezes, 2000) to garbage collect unnecessary tuple-spaces using a graph, where reference information is maintained. However, tuples within the tuplespace cannot be garbage collected selectively: in order to perform garbage collection on tuples, information about their usefulness should be maintained for the garbage collector. This is not possible in the standard Linda as tuples cannot be individually referenced—for the purpose of maintaining the information related to their usefulness—as they are nameless. In Lindacap (Udzir, 2006; Udzir, 2007), with the introduction of multicapabilities, i.e., capabilities for classes of objects, tuples within a tuple-space can now be referenced. Therefore, the information about the accessibility of tuples can be maintained by adopting the graph structure used in Ligia. As a result, garbage collection on tuples has been implemented in order to remove unnecessary tuples with specified template and subsequently reclaim memory space.

This paper describes the deployment of garbage collection in Lindacap. The implementation is based on reference counting. Unfortunately, this technique does not apply to cyclic garbage, i.e., two or more multicapability regions referring to one another. In order to overcome this, the implementation of garbage collection in Lindacap is extended using the mark-and-sweep algorithm to allow cyclic garbage collection.

There may also be the case where some agents may still have references to certain groups of tuples (represented by multicapabilities regions) but they no longer use the tuples/groups: the agents may have forgotten about these tuples which are consuming memory space. Based on the graph, the regions will not be considered as garbage as they are agents referring to them. A mechanism should be provided.
to allow these regions to be garbage collected. Therefore, in this paper, we also describe a rather simple, but useful, additional technique, i.e., the incorporation of time-to-idle (TTI) for the purpose of reclaiming memory allocated for groups of tuples that have been ‘idle’ for a certain length of time.

RELATED WORK AND MOTIVATION

Resource management, for example, is vital in distributed systems that involve ubiquitous and persistent computing. Poor resource management, at best leads to poor performance. One resource that needs to be managed is memory, which is limited but can be reclaimed through garbage collection. Garbage collection is the process of searching and automatically reclaim unused memory cells to avoid memory exhaustion. The algorithms are based on traversing a tracing graph representing the memory to analyze and determine the cell’s usefulness (Menezes, 1997; Menezes, 2000). In order to do so, the kernel must have knowledge of which objects are being referenced by which agents.

The reference counting algorithm has been around since 1960 when it was developed by Collins (Collins, 1960)—each reference to a memory cell increases the cell’s counter, while a de-referencing decreases the counter; memory cells are reclaimed when their reference counter reach zero. This method, however, fails to reclaim cyclic structures, where the counters never reach zero (McBeth, 1963). A solution to this problem was introduced by combining tracing (mark-and-sweep) and reference counting (Friedman, 1979; Lins, 2008).

Agent registration in Ligia helps the kernel to identify any agent in the system. Tuple-spaces are also identified by their unique handle. As a result of this, a graph with reference information for each tuple-space has been proposed to maintain information about the usefulness of tuple-spaces for the garbage collector. Thus, a garbage collection mechanism has been implemented in Ligia, to garbage collect unused tuple-spaces. However, tuples within a tuple-space that are no longer used by agents cannot be removed by the garbage collector unless the whole tuple-space becomes garbage. This is particularly problematic in the case of the UTS as the default tuple-space can never be garbage collected in Ligia. In order to perform garbage collection on certain tuples, information about their accessibility should be maintained, and this can be done using multicapabilities in LINDACAP.

Garbage collection of tuples in JavaSpaces (Freeman, 1999) uses leasing in which tuples within the space are associated with a limited time frame (i.e., leases). As a result, the garbage collector agent can remove these tuples after a certain period of time elapses (when the lease expires). The drawback of this approach is that it is difficult to estimate in advance how long a particular object will be of use. They vary according to applications, for instance, we might want to reclaim all objects generated by a particular agent, or to clean up any object left over a particular protocol.

Object (tuple) retrieval in SecOS (Bryce, 1999) is different from LINDA. Matching does not require the number of fields of the template and the target tuple to be equal. Thus, an agent can use an empty template to retrieve any tuple without gaining access to its contents. As a result, tuples can be deleted periodically by the garbage collector agent. Since tuples are deleted randomly without considering any tuple’s usage information, the garbage collector agent cannot selectively delete unnecessary tuples. In a later version of SecOS (Vitek, 2003), each tuple is tagged with some information (similar to a lease) before being inserted into the space. The garbage collector agent can use this information and remove the tuple with particular tags. However, tagging of tuples is performed by agents voluntarily but not automatically by the system.

In Scope (Merrick, 2001), a streaming mechanism is used to perform an input operation upon all tuples in a given scope without affecting the other agents which are using the same tuples. This streaming mechanism can be used to access fresh tuples in the relevant
scope and these tuples can be tagged with a scope representing the time-slices in which their leases expire. A sibling agent can use these tags to search for tuples whose lease has expired, and remove them.

Garbage collection in shared memory systems has also been discussed in XMem (Wegiel, 2008), where unused shared objects are identified and reclaimed, using conventional tracing approach. Their more fine-grained tracing garbage collection is an improvement as compared to the coarse-grained block-based reference counting garbage collection of Microsoft’s Singularity (Fähndrich, 2006).

In LINDACAP, tuples within a tuple-space are grouped together based on a specified template and can be referred to using a multi-capability with a unique identity. This allows the kernel to keep track of the tuples usability for the garbage collector. Therefore, garbage collection of tuples has been implemented in LINDACAP to remove selective tuples (a multicapability region) rather than removing the whole tuple-space (Udzir, 2006; Udzir, 2007; Udzir, 2008). The implementation is based on the reference counting technique, in which the links that reference each multicapability region is maintained, and the storage for these regions can be reclaimed when the reference counter is decreased to zero. Unfortunately this approach has two problems. First, the reference counts for multicapability regions that are part of a cycle (i.e., two or more multicapability regions that refer to one another) will be non-zero. Thus, this cyclic garbage cannot be garbage collected. In order to overcome these problems, a tracing based technique called the mark-and-sweep method, which is based on Dijkstra’s on-the-fly proposal (Dijkstra, 1978) is also implemented.

**LINDACAP**

LINDACAP extends Linda with a capability-based control mechanism to provide a more refined control for open distributed systems without losing the flexibility of Linda. Objects visibility to agents in LINDACAP are defined by the capabilities (similar to tickets) they hold. Capabilities can provide information not only on a particular object, but also on which methods of the object an agent is permitted to invoke. Specific information about an agent’s ‘knowledge’ is potentially very useful and can be manipulated in a variety of ways.

However, unlike access control lists (ACLs), capabilities can only be applied to named objects, such as tuple-spaces, but not the nameless tuples. To overcome this, we have introduced multicapabilities (Udzir, 2006; Udzir, 2007), i.e., capabilities for a class of objects: whereas a permission in a uni-capability allows an action on the object it refers to, a multicapability allows the action to be performed on an element of the class. Throughout this paper, we shall use the term ‘capability’ to refer to capabilities in general, and the terms ‘uni-capability’ or ‘multicapability’ accordingly when referring to a specific class.

Extending uni-capabilities which are pairs of object identifier obj and rights ([obj, [i, r, o]]), each multicapability is a tuple of an identifier, the template of tuples it refers to, and a set of rights, i.e., [u, t, p], where u will be used as a unique (unforgeable) identifier to a collection of objects of the specified template t, and p is a set of operations allowed on an element of the group. For example, a multicapability for a template of two integers is [a, {Int, ?Int}, {i, r, o}], where i, r and o are permissions to perform destructive and non-destructive read, and write, respectively.

The capability data (unique identifier, reference/template, and permissions) are assumed to be securely encapsulated in the capability and only interpretable by the kernel when the capability is presented for verification.

Consequently, there are two kinds of capability in LINDACAP: a uni-capability (TS-capability) that holds the set of operations permitted on the tuple-space it refers to, and a multicapability which holds the set of operations to be performed on a group of tuples (multicapability region) within the tuple-space. Every tuple-space and tuple operation requires two-level capabilities: a TS-capability for the
target tuple-space, and a multicapability for a specified template (a formal for a tuple pattern).

Whenever an agent requests for a new capability, the kernel returns a unique capability with full rights. Therefore, tuples of the same pattern may be referred to by different multicapabilities. For example, if an agent makes two separate requests for a multicapability for the template \( \langle \text{str}, \text{int}, \text{int} \rangle \), it will get two multicapabilities, each different from the other (i.e., identified by different tags), although they correspond to the same template. If one of the multicapabilities is given to another agent, the second agent can only ‘see’ tuples in the multicapability group it holds, but not those in the other group. Hence, multicapabilities can provide a partitioning of a tuple-space—enabling certain operations to be performed on a tuple of a specific group, but not on one of another group, even though both groups have the same template.

In LINDACAP, communications between two or more agents can be performed by passing capabilities among them. Requesting a capability, even for the same template will not allow an agent to retrieve data produced by another agent: they need to use the same capability to access the same (set of) data. Hence the capability needs to be passed from the producer to the other agent via a tuple-space. Capabilities can be elements of tuples and stored within tuple-spaces. These tuples can be retrieved by another agent that has access to the tuple-spaces. The agent will then be able to access the objects referred to by these capabilities. The universal primordial capability,

\[
cc = [8, \langle ?cap \rangle, \{r, o\}],
\]

for a capability type is used to pass capabilities (encapsulated within tuples) via tuple-spaces, or at least via the universal tuple-space (UTS). For this purpose, a TS-capability (with full rights) for the UTS, and \( cc \) are given to each agent whenever it connects to the kernel.

Passing capabilities through tuple-spaces is just like passing any ordinary data, using the standard input/output primitives. For example, passing an arbitrary capability \( \text{cap1} \) entails writing the capability into a tuple-space,

\[
ts\text{-}out(\text{cc}<\text{cap1}>);
\]

With the universal primordial capability, \( cc \), this tuple may be retrieved by another agent,

\[
ts\text{-}rd(\text{cc}<?\text{cap}>) ;
\]

Agents may duplicate, or make restricted copies, or even combine capabilities (in sum or subtraction operations) in their possession. There are two types of operations: unary and binary. Unary operations involve a single capability, where a copy of a capability, while having the same unique identifier as the original, may have a restricted template and/or rights. However, it is not possible to add rights, nor to generalize the template in a multicapability. For example, a copy of multicapability \( [a, \langle \text{int}, 3 \rangle, \{i, r, o\}] \) may have one or more rights removed; and its template restricted to \( \langle 56, 3 \rangle \) — or any integer value as its first element—but not generalized to \( \langle \text{int}, \text{int} \rangle \) as the second element has been specialized to integer 3 and cannot be altered.

Binary operations ’combine’ two (or more) capabilities in an agent’s possession. We present in this paper two simple binary operations: sum and subtraction.

The sum operation (+) produces a multi-capability referring to the template of either multicapability, and represents permissions if any one of the multicapabilities grants that permission. A tuple produced using the sum of two multicapabilities can be retrieved using either multicapability, provided the action is permitted by the multicapability used in the attempted retrieval. To further elaborate on this operation, let us consider the following multicapabilities:

\[
cl = [a, \langle \text{int}, \text{char} \rangle, \{i, r, o\}]
\]
\[
c2 = [b, \langle 3, \text{char} \rangle, \{r, o\}]
\]

Writing a ‘sum’ tuple of these multicapabilities, e.g.,
produces (into tuple-space \textit{tst}) a tuple that can be accessed using either \textit{c1} or \textit{c2}, or both multicapabilities. However, as the multicapabilities grant different rights, then any agent who has \textit{c1} (or both multicapabilities) can read or remove the tuple, while those with only \textit{c2} can only read it.

Reading using the sum operation,

\begin{verbatim}
  tsl.out((c1+c2)<1,'a'>);
\end{verbatim}

enables the reader to search both groups ‘simultaneously’. Without sum, the reader must perform two separate reads, e.g.,

\begin{verbatim}
  tsl.rd((c1+c2)<3,'a'>);
  tsl.rd((c1<3,'a'>);
  tsl.rd((c2<3,'a'>);
\end{verbatim}

with the risk of being blocked on the first group if no matching tuple is available, before it has the chance to search the second group (where the tuple might exist). Therefore, using sum in a read operation reduces the probability of being blocked by half. Indeed, what is more, possible deadlocks can be avoided—a sequence of two input operations may deadlock, whereas the sum (which is equivalent to a parallel combination of two input operations) will only block until a tuple becomes available in either group.

The subtract operation (−) restricts a capability relative to another. For instance, if we have a third multicapability,

\begin{verbatim}
  c3 = [γ, (3, ?char), (i)]
\end{verbatim}

an output operation using \((c1 - c3)\) will write a tuple of template \(\langle ?int, ?char \rangle\), excluding any tuple whose first element is integer 3. This tuple will be written into group \textit{c1}, and can only be accessed by agents holding \textit{c1}, but not accessible with \textit{c3}.

An input operation using \((c1 - c3)\) would yield a tuple matching the template of \textit{c1}, except those matching the template of \textit{c3}. Since the permission(s) in \textit{c3} is also subtracted from \textit{c1}, non-destructive reading is disallowed.

The combinatorial (binary) operations offer richer possibilities for capabilities to be manipulated to provide a finer control on objects’ visibility to agents in open systems.

Multicapabilities in \textit{LINDACAP} can be used to implement some applications which are not feasible in the standard \textit{LINDA} model, such as garbage collection of tuples. Garbage collection requires the knowledge of the references to objects, meaning that the objects must be uniquely identifiable, which tuples are not—they are referred to using associative pattern matching, not by names. Multicapabilities enable selective groups of tuples to be referenced and consequently garbage collected, rather than garbage collecting the whole tuple-space as in Ligia.

\section*{Garbage Collection of Tuples}

Garbage collection is the process of deallocating memory spaces previously allocated to objects but the objects are no longer in use. Most garbage collection algorithms are implemented using directed graphs (Dijkstra, 1978) which are traversed to locate unused nodes (representing memory cells allocated to objects). Although there are a number of garbage collection algorithms for distributed systems, we focus on two approaches: reference counting and tracing.

In reference counting, each memory cell (node) allocated to an object is associated with a counter of the number of references to itself. The counter is increased every time a reference is made to it from a ‘new’ subject, and decreased whenever a reference to it is severed/deleted. The object is considered garbage when its reference count reaches nil, and the memory can be reclaimed.

The tracing approach involves searching the graph and marking all nodes that are reachable from the root (or roots of the subgraphs) (Dijkstra, 1978), hence all unmarked nodes are considered garbage: nodes that are not reachable from the roots are not accessible by any ‘living’ node, and therefore are unnecessary.
A tracing algorithm is usually described based on colours: all nodes are initialized as garbage (white), and those accessible from the root is marked black (not garbage). The basic tracing algorithm is called “mark and sweep”, where the garbage collector traverses the graph of references and marks each object it encounters, and then sweeps the graph to remove unmarked objects.

The implementation in Ligia, however, was restricted to tuple-spaces: we can either garbage collect, or keep, the whole tuple-space, but not selectively garbage collect certain tuples within the tuple-space. The main problem in introducing garbage collection for tuples is the lack of sufficient information about their “usage”. This information can be maintained if we can reference a particular tuple, or a group of tuples of a certain type—something that is not possible in LINDA. While tuple-spaces have unique identities, tuples (and templates) are anonymous: they are referred to by values instead of names. Thus it is difficult to employ garbage collection on tuples without modifying the model: giving unique names to tuples will certainly break one of the fundamental characteristics of LINDA, i.e., associative retrieval. However, with the introduction of multicaabilies, this can be avoided as multicapabilities enable the system to reference a collection of nameless tuples to perform garbage collection on them, by garbage collecting the multicapability regions themselves. As multicapabilities provide a means to refer to tuples, we can perform garbage collection on the tuples (Figure 1).

In fact, this mechanism is better than Ligia in the sense that it provides a finer control over the system as we can now selectively garbage collect only a certain region specified by a given multicapability, rather than the whole tuple-space. In Menezes’s Ligia, the UTS (including its contents) can never be garbage collected, therefore any tuple put into it will persist in the system forever until explicitly removed, or the system terminates (Menezes, 2000). The number of tuples in the UTS may grow, and some may become unusable over time, thus consuming valuable memory space.

With multicaabilities, it becomes possible to garbage collect some of these tuples as we can specify region(s) in the UTS to be garbage collected—without having to remove the whole UTS—thus providing a finer control over the system.

**Garbage Collection in LINDACAP**

Our earlier papers on LINDACAP (Udži, 2007; Udži, 2006; Udži, 2005) discuss why capabilities are suitable for open systems, mainly due to their “distributivity”. Multicapabilities have been implemented in a distributed manner in the sense that the kernel does not need to maintain any information pertaining to the capabilities. However, in order to implement garbage collection, the kernel must keep track of the capabilities (reference and permission on objects, including groups of objects, i.e., multicapability regions) against the agents holding them. For example, to determine if an object is garbage or not, the kernel must know which agent has the reference (capability) to the object: if no agent has the reference to the object, then it is garbage and can be reclaimed.

Having a (centralized) list maintained by the kernel undermines the distributed properties of capabilities. This would seem like a retrograde step—contradicting the ideal notion of capabilities need not be maintained by the kernel—but some carefully weighted decisions need to be made: do we want a system with distributed capabilities, or do we want a system that provides garbage collection and a deadlock detection mechanism? It is obvious that garbage collection, at least, is vital for persistent systems: if systems are meant to exist for a long period of time, there has to be some kind of scheme to ensure that the system does not run out of resources, such as memory. Of course, there is no guarantee that memory exhaustion can be fully prevented—as memory is a finite resource—but a system without some kind of mechanism to reclaim unused memory spaces would be disastrous, as memory exhaustion would be unavoidable. In the end, the advantages...
of these applications far outweigh the need to have a distributed capability management.

In LINDACAP, each multicasability region (or rather the data structure representing it) holds a list of references to it, with the first entry being the agent that created the region in the first place. Every time a multicasability region is referenced (by an agent, or an object, or another multicasability region), a new reference is added to the list specifying the identity of the agent/object/multicasability. Conversely, an entry in the reference list is deleted when a reference to the multicasability is removed. Eventually, if there are no more references left, the multicasability region is garbage collected.

In this implementation phase, the kernel keeps a list of all capabilities that have been passed through it under the following circumstances:

- Every time an agent requests and gets a new multicasability, or creates (and gets the capability for) a new TS, or
- Every time a tuple containing a capability is retrieved by an agent.

The tuple monitoring mechanism reported in (Menezes, 1998) helps the kernel keep track of the (TS) references being passed within the system. LINDACAP also incorporates tuple monitoring, extended to capabilities being passed as tuple elements, which will be essential for garbage collection. To perform garbage collection of tuples, the graph structure used in Ligia (for tuple-spaces) has been adapted in LINDACAP (as described in (Udzir, 2008)) to keep track of the usability of tuples (multicasability regions) in the system. The nodes in the graph are made of agents (depicted as rectangles) and multicasability regions (ellipses). The root of the graph is the default multicasability cc where agent nodes linked to when they connect to the system. Each multicasability node is linked to the agent nodes holding a reference to it, and each agent must be linked to all multicasabilities it holds. Whenever a multicasability (in a tuple) is written into another multicasability region, then a bridge (directed edge) is created between the two nodes, and the weight of the bridge is a counter representing the number of tuples containing the first multicasability that exist in another region.
Graph Implementation

The kernel can use the information supplied by the multicapabilities to keep track of the usage of tuples in the system, eventually removing unusable tuples to reclaim memory space. Although the garbage collection mechanism for TSSs (adopted from Ligia) has been incorporated in LINDACAP, the discussion in this section focuses on the garbage collection mechanism with regard to tuples (and multicapability regions).

As capabilities are first-class values, it is possible to remove unused multicapability groups. It is more practical and efficient to re-create a new multicapability later when needed, rather than retaining the old unused ones which have the probability of not being used ever again.

The graph data structure used in Ligia can be adapted for this purpose. In the graph to implement tuple-garbage collection in LINDACAP, the nodes represent the agents and multicapability groups, connected by edges. Each multicapability node is tagged with a counter field to represent the number of references to them. The basic structure of the graph is as follows:

1. There is a node representing the primordial capability $cc$, which is the root of the graph. This node is used as the starting point of the tracing phase of mark-and-sweep to decide if a given node is garbage based on its accessibility from this starting point.

As mentioned above, the universal multicapability $cc$ is the default multicapability for a capability type tuple/template—when an agent connects to the kernel (or a LINDACAP server), it is given a copy of the universal multicapability $cc$ to enable them to share information among them. With regard to the graph creation, a node representing the $cc$ multicapability is created upon the start of a LINDACAP server. As the root of the graph, and therefore the starting point for the tracing phase, this node is never garbage collected as its counter field is set to infinity. $cc$ will continue to exist as long as the LINDACAP server lives.

2. There is a unique node representing each agent, which is created whenever an agent connects to a LINDACAP server. Since all agents automatically get access to the $cc$ multicapability, all agent nodes are linked to the $cc$ node.

In order to keep track of tuples usability to perform garbage collection in LINDACAP, agents have to be registered upon their connection to the LINDACAP kernel. Thus, LINDACAP is able to identify the agents whenever they execute a primitive that could alter the graph. During agent registration, an agent requests for, and receives a unique identifier from the kernel. From there on, the agent uses this identifier when executing the Linda primitives. The use of this unique identifier is transparent to the agents themselves, as the identifier is assembled in the messages sent to the kernel.

The graph structure is updated: the link is created between the node representing the agent and $cc$ node. Figure 2a shows the graph situation when two agents connect and register themselves to the kernel.

3. There is a unique node representing each multicapability.

When an agent made a request to the LINDACAP kernel for a new multicapability, the kernel will return a unique multicapability for the specified template, different from any other multicapability, even those for templates of the same pattern. LINDACAP provides a primitive to create a multicapability object. The newcap primitive returns a multicapability object with a unique identifier which has the default full rights for in, rd, and out operations. At this point, the graph is updated accordingly. The kernel can identify the creator of multicapability as the message requesting the creation will contain the agent identifier. Using tuple monitoring (Menezes, 1998) the kernel can extract this
Figure 2. A graph representation of the references/capabilities in LINDACAP

(a) Agents connecting to the kernel. Each has reference to cc.

A1: lindacap.connect();
A2: lindacap.connect();

(b) Agent A1 creates new multicapability. mc1's reference count is set to 1; it is currently known to A1 only.

A1: mc1 = newcap(<...>);

(c) A1 puts tuple <mc1> into cc, twice. The reference count is now 3:
2 tuples <mc1> in region cc, and 1 held by A1.

A1: out( cc<mc1> );
out( cc<mc1> );

(d) A2 reads tuple <mc1> from cc. The reference count is now 4:
2 tuples still in region cc, and 1 each held by A1 and A2.

A2: rd( cc?<cap> );
(assuming no other capability in cc)

The agent and the node representing the multicapability (Figure 2b).

(It is important to see the difference between creating a node representing a multicapability in the reference graph here, and the actual physical creation of the multicapability region (representing the collection of tuples) referred to by that multicapability, which is created when the first tuple is outputed using the multicapability.)

4. The links between agents and multicapabilities are always via direct edges.
   - Whenever an agent connects to the kernel, an edge is created between it and the cc node (Figure 2a).
   - Whenever an agent creates a new multicapability, an edge is created between

5. A multicapability node must be linked to its creator (agent) node.
6. An agent node must be linked to all mult-capabilities it knows (see Section ‘Keeping track of capabilities’ for more details on how an agent can acquire knowledge of a mult-capability).

7. The links between mult-capability regions are done using labelled directed edges, called bridges. When a tuple containing a mult-capability, say $mc_2$, is deposited into a mult-capability region, say $mc_1$, then a bridge is created between them in the reference graph, directed from $mc_1$ to $mc_2$. (Again, the bridge created is between the nodes in the graph, not between the physical regions, as the $mc_2$ region might not be created yet—no tuple output into it yet). The weight of the bridge is a counter representing the number of mult-capabilities $mc_2$ (in tuples) that exist in $mc_1$. If there are two capabilities $mc_2$ within $mc_1$, the weight is 2 (Figure 2c).

The counter is decreased every time the tuple containing mult-capability $mc_2$ is removed (via in or inp) from $mc_1$, but remains the same if the mult-capability is only read (rd or rdp). (Note that, except in certain cases where a new mult-capability for a capability type is explicitly created, a capability tuple can only be seen using the primordial capability cc which does not allow destructive read, i.e., in or inp. However, with the sum operation (see Udzir, 2006; Udzir, 2007, such a tuple can be removed).

To explain the graph structure in Figure 2:

Figure 2a: Agents $A1$ and $A2$ connects to the kernel, and gets the default primordial capability $cc$. In the graph, an edge is created between the $cc$ node and each node representing each agent.

Figure 2b: Each mult-capability node is associated with its reference counter: the counter is set to 1 when the mult-capability is created, implying that it is held by one agent—its creator. Therefore, in the diagram, when agent $A1$ creates a new mult-capability $mc_1$, an edge is created between node $A1$ and node $mc_1$, with the reference count 1 indicating there is 1 reference to $mc_1$, i.e., by $A1$.

Figure 2c: When the mult-capability is written (in a tuple) using another mult-capability, e.g., $cc$ (i.e., written into region $cc$), a bridge is created from node $cc$ to node $mc_1$, indicating there is a reference to $mc_1$ from within $cc$. The reference counter associated with $mc_1$ is increased to indicate two references to it (from $A1$ and now from $cc$). When a second tuple containing $mc_1$ is written into $cc$, as in Figure 2c, this reference counter is increased to 3 (meaning two references (in tuples) in $cc$ and one reference from $A1$). The label on the bridge from node $cc$ to node $mc_1$ is also updated to 2, indicating two references (tuples) to $mc_1$ in region $cc$.

Figure 2d: When agent $A2$ reads a tuple containing $mc_1$, a new edge is created between $A2$ and $mc_1$, indicating that $A2$ now has a reference to $mc_1$. The reference counter for $mc_1$ is increased to 4: one reference from $A1$, one reference from $A2$, and two references (in tuples) from within $cc$. Note that, the references/tuples in $cc$ remains the same because $A2$ only reads the tuple containing $mc_1$ (as cc does not allow inp). If the capability tuple were removed—assuming the same diagram as in Figure 2d except that $cc$ is replaced with a different capability for capability type (e.g., $c2$) which allows in—then the bridge label will be decreased to 1 indicating there is only one tuple left containing $mc_1$ in the region $c2$. The reference counter on the other hand will remain the same, i.e., 3, as the reference has only been moved from the region $c2$ to the agent $A2$.

Graph Algorithms

The algorithm for the graph construction, i.e. when an agent performs an out operation is:

The pseudocodes for updating the graph when agents retrieve tuples from mult-capability regions are given below. When an agent executes a rd operation:
When an agent executes an in operation, the pseudocode is:

### Keeping Track of Capabilities

We have established that the kernel needs to maintain some kind of information regarding the references—which agent knows about which object in order to perform garbage collection. There are three ways for an agent to acquire a capability (either for a tuple-space or a group of tuples), and how the kernel may keep the information it needs:

1. The agent creates a tuple-space (therefore obtaining the TS-capability in return), or requests a new multicapability for some template of tuples. The kernel can simply record the agent's identity against the newly created capability.

2. The agent has retrieved a tuple containing a capability for the object. To obtain this information, it is necessary for the kernel to implement tuple monitoring (Menezes, 1998)—which has also been extended to monitor tuples containing multicapabilities—to enable the kernel to keep track of capabilities being passed as tuple elements.

3. The agent has been spawned by a parent agent, and the capability has been passed from the parent. As discussed in (Jacob, 2000), this is a rather complicated case, and the solution relies on process registration and 'deregistration' (Menezes, 1998), which have been incorporated in the LINDACAP implementation—all newly spawned agents must register themselves and the capabilities they hold, and must 'unregister' with the kernel before terminating.

Termination ordering assures that the termination message is the last message (from a given agent) to arrive in the kernel (Menezes, 2000). Agent termination is an operation that can generate garbage, as capabilities may be deleted, which would result in the loss of references to some objects. Therefore, termination ordering should also be observed to avoid race conditions.

### Garbage Collection using Reference Counting

In the graph, each multicapability node is attached with a counter field indicating the number of references to the multicapability region, either references held by agents, or in tuples.

---

**Algorithm 1.**

```
IF agent A1 outs a tuple into mcl THEN
  IF NOT Linked(A1, mcl) THEN
    Link(A1, mcl);
  IF the tuple contains multicapability mc THEN
    IF the agent outs to cc THEN
      Link(cc, mc);
    IF NOT Linked(A1, mc) THEN
      Link(A1, mc);
  out(tuple);
```

**Algorithm 2.**

```
IF agent A1 reads a tuple in mcl using a specified template THEN
  IF NOT Linked(A1, mcl) THEN
    Link(A1, mcl);
  IF the tuple contains multicapability mc THEN
    IF NOT Linked(A1,mc) THEN
      Link(A1, mc);
  rd(template);
```
within another multicapability. As illustrated in Figure 2 this counter is increased every time a new reference is made to the multicapability region, i.e., when:

- A copy of the tuple containing the multicapability is written into a TS, or
- A capability tuple is read by an agent.

A rd operation does not ‘move’ a tuple to the retrieving agent, but rather it returns a duplicate copy of the tuple—hence increasing the reference counter. In on the other hand, removes a tuple to be given to the retrieving agent. Therefore, an in operation does not increase the counter value.

Naturally, a node (representing a multicapability region) is deleted (i.e., garbage collected) if and when the reference (counter) to it becomes nil. The counter is decreased every time a reference to the multicapability is deleted, i.e., the edge(s) connected to the node is/are deleted. This transpires when:

- A multicapability is revoked, or
- A tuple containing a multicapability referring to the region is deleted, or
- An agent holding the multicapability dies.

Any one of these circumstances decreases the reference counter in the multicapability node. The multicapability region is considered garbage when the counter reaches zero. Being the default multicapability, the cc (root) node is never garbage collected: its reference counter is \( m = \infty \).

**Garbage Collection for Cyclic Garbage**

Garbage collection based on reference counting does not garbage collect multicapability regions that are part of a cyclic structure. Passing of multicapabilities in LINDACAP allows the creation of cyclic structures between multicapability regions. In this paper, garbage collection based on the mark-and-sweep algorithm (Dijkstra, 1978) has been implemented to remove cyclic garbage using the graph structure in LINDACAP (as described in (Udzir, 2008)) to keep track of the usability of tuples in the system. Mark-and-sweep has two separate phases:

**Marking phase:** After the graph has been initialized by marking all objects (multicapability regions) as garbage, traversal is initiated from the root; all objects it encounters are marked as non garbage.

**Sweeping phase:** Unmarked objects are freed, and the resulting memory is made available to the executing program.

Figure 3a depicts a possible graph situation at some point during the execution of a LINDACAP system. A cyclic reference exists between mc1 and mc2 when a tuple containing mc1 is written in mc2, and vice versa. When agents A1 and A2 terminate, the reference counters for mc1, mc2, mc3, and mc4 are decreased by 1. The reference counting takes effect and identifies and collects the objects (shown in dotted lines in Figure 3b), i.e., all links to/from both A1 and A2, and node mc4, whose counter is now zero.
The regions mc1 and mc2 are now garbage as they are no longer referenced by any agent but they are not garbage collected as each of their reference counter is 1: they are cyclic garbage. Therefore, in this case, the garbage collection using reference counting is incomplete as it does not collect cyclic garbage.

This problem can be solved using the mark-and-sweep technique. Firstly, the graph is initialized by marking all nodes (regions) as garbage (white), then the graph is traversed starting from the root (cc node) and every node in the path is marked non-garbage (black) to identify this node as being in-use. At the end of the first step, all unmarked (white) nodes are considered as garbage—as they are not accessible from the root—and are ‘swept’. Figure 3c depicts the graph after the tracing: all the nodes left unpainted are considered garbage and are collected, including regions mc1 and mc2.

**Time-to-Idle (TTI) Technique**

Both reference counting and mark-and-sweep techniques cannot garbage collect multiscapability regions as long as the agent that holds the reference to these regions exist in the system. If these regions are no longer in use by the agent, they should be removed from the system but the agent may forget to delete them. These regions are consuming memory resources, which may cause memory exhaustion. To overcome this problem, a time-to-idle (TTI) technique is proposed.

TTI is the period for which the object is guaranteed to stay idle (not in use). When a link is made to a multiscapability object, TTI is added to the current time and the resulting expiration date is attached to the object. To guarantee that the object will continue to exist, it should be refreshed before it expires. This can be done when an object is referenced. When the object is about to expire, the kernel can send a notification message to all agents possessing a reference (capability) to this object at a specified time before the object is about to be garbage collected, and any agent wanting to use the object must refresh it. The TTI for a multiscapability region is chosen automatically by the system when the object is created. Every object is treated equally: the same TTI value is chosen for every object in the system.

**Implementation**

We implemented the LINDACAP system with garbage collection using reference counting and mark-and-sweep separately, together with the time-to-idle technique. The system was implemented using Java (JDK 1.5.0). The particular choice of Java was based on the idea that Java provides the support to run the system in heterogeneous environment.

The implementation of LINDACAP uses distributed central server as their tuple-space distribution strategy (Carriero, 1986), in which the kernel is distributed among several machines (Figure 4), which would result in the distribution of workload among the servers. The communication among servers occurs via sockets. LINDACAP implementation has the following characteristics:

- There is a public universal tuple-space (UTS) that is accessible by all.
- All agent communications are done solely in terms of tuple-space operations.
- Every tuple-space, with the exception of the UTS, is explicitly created by agents in the system.
- Tuple-spaces in the kernel are flat structured: every tuple-space is created at the same level, that is, tuple-spaces cannot be created inside others.
- Capabilities are first class objects.
- There is a single type of capabilities, referred to as ?cap in templates.

The architecture is divided into two basic systems: the server and agent systems (Figure 4).

**Server System**

Each server is multithreaded: a new thread is created for each request made by an agent.
Figure 3. Tuples (multic和平pability regions) being garbage collected (b) reference counting and (c) tracing (mark-and-sweep)

(a) Initial Scenario

(b) GC by reference counting (after A1 and A2 died)

(c) GC by tracing (mark-and-sweep)
The server is composed of the following basic components:

1. Agent Connection

This component is responsible for dealing with all communications with agents. It is implemented by the Connection Java class which keeps reading the socket for requests. When an agent requests a connection, the server assigns a name to the agent (AgentId): this name is guaranteed to be unique and will be used in all ensuing communications. Basically, a thread is normally created when the message is received from an agent that has never communicated with this server before.

The message sent from the agent system is attached with the AgentId, which identifies the agent sending the message: this is very important for the garbage collector which needs to know which agents are executing each operation. The TupleMonitoring method which is implemented in the agent connection module extracts information for the garbage collector from the message sent by the agents. If the multicapabilities are involved in the operation, the garbage collection structure is updated accordingly. When an agent disconnects, the corresponding connection is closed and the Socket object is collected by the Java garbage collector.

2. Garbage Collector

In the server system classes GCReferenceCounting, GCMarkAndSweep, and GCIdleCollector are implemented, which extend the class thread. When the first object of the class LindacapServer is created, a GCIdleCollector object is also created. The method sleep in the class thread is used to stop this thread for a determined period of time to reduce the resource consumption by the garbage collector. The execution of this thread removes multicapability regions with expired TTI and reclaims the memory space. Prior to the expiration of a TTI of a multicapability region, a notification message will be sent to all agents that has access to this region, as a reminder for them to refresh the TTI if the region is still needed.

GCReferenceCounting is executed when an agent holding the reference terminates. The execution of the GCReferenceCounting invokes relevant methods and these methods linearly search for nodes (multicapability regions) whose ReferenceCounter is zero, remove these multicapability regions, and reclaim the memory space. The other garbage collection thread is GCMarkAndSweep, which is executed when there is a possible existence of a cyclic garbage: two or more multicapability regions that refer to one another. The execution of this object involves several separate steps. First, the color variable in all the multicapability objects set to 0 (marked white). After this initialization, the marking phases starts. Starting from cc, the color of all multicapability regions which are reachable from the root cc, is set to 1 (marked black). Reaching the end of the graph, it pro-
ceeds to remove all nodes whose color variable is still 0 (i.e., white). The thread method sleep is used to avoid both these garbage collection objects from consuming resources. Therefore, the sleep method is used to control the execution of the object, which would help to reduce resource consumption.

**Agent System**

The agent system is composed basically of one component, i.e., the Communication System, which is responsible for providing all the agents with coordination capabilities. An agent has to request a connection to the LindacapServer and only then can it use the primitives. When the request is accepted, a unique name (AgentId) is assigned to the agent and a communication channel using a socket object is established.

**Algorithms**

The pseudocodes for the mark-and-sweep and the time-to-idle techniques are given below. For mark-and-sweep, the pseudocode is:

The pseudocode for Time-To-Idle (TTI) is given below:

A simple and naive strategy for implementing garbage collection is to run the garbage detection algorithm every time the reference to an object has been decreased to zero/nil; and one reason for a reference to be deleted is when the agent holding the reference dies. It is also known that performing garbage collection can be an expensive operation (in terms of kernel load). Therefore, it is more efficient to garbage collect only when needed, i.e., when there is insufficient memory space available. Even though this strategy involves a larger amount of work to be carried out at one time compared to the former, garbage collection is likely to be performed less often.

Some mechanisms may be adopted to increase the efficiency of the algorithms, e.g. the Jump_stack data structure by Lins (Lins, 2000); or the 'critical link' concept (pointer operation that creates a cycle) (Lins, 2008) where the mark-scan algorithm will be run whenever an attempt is made to delete a pointer to the cell pointed by the critical link. These algorithms have been adapted in shared memory architectures (de Aratij Formiga, 2007) as well as in distributed environments (Lins, 2006; Lins, 2003). Baker et al. (2009) propose lazy pointer stacks, which perform accurate garbage collection in such uncooperative environments, with the implementation of a real-time concurrent garbage collection algorithm.

---

**Algorithm 4.**

```
IF agent A1 terminates THEN
    FOR (all mei, is_Linked(A1, mei)) DO
        Un_Link(A1, mei);
    // Mark-and-Sweep Algorithm
    // Initialization step
    FOR (all mei in the system) DO
        Mark(white);
    // mark phase starts
    FOR (all mei, referenced by ec) DO
        // all multicapability regions that are
        // directly or indirectly accessible by
        // the ec will be marked black (non-garbage).
        Mark(black);
    // sweep phase starts
    FOR (all mei, is_Mark(white)) DO
        Sweep();
```
Algorithm 5.

IF an Agent get access to mcel THEN
   add(TTI,mcel);
FOR every interval of time CHECK
FOR(each mci in the system) DO
   IF (TTI of mci is about to expired &&
      IsNotNotifiedYet(mci)) THEN
      FOR(all Aj, Is_referred(mci,Aj) DO
        Send(Notification Message);
        ChangeStatus(mci,notified);
   ELSE IF (TTI value is expired and isNotified (mci)) THEN
      GarbageCollect(mci);

EXPERIMENTAL RESULTS

The reason for garbage collection is to avoid memory exhaustion. As discussed in earlier sections, Ligia (Menezes, 2000) only performs garbage collection on TSs, and therefore does not garbage collect tuples in the UTS. This can lead to disastrous consequences. Multicapabilities enable some of these tuples to be garbage collected, as has been discussed in the previous section. Experiments have been carried out to demonstrate this.

These experiments compared two capability systems for memory exhaustion: one incorporates the garbage collection mechanism for tuples, whereas the other does not. The characteristics for the systems are:

1. Both systems deployed garbage collection of TSs, based on Ligia.
2. All interactions are via the UTS, which is not being garbage collected—the TSs garbage collection mechanism (Menezes, 2000) cannot be performed on UTS. Therefore, we can be certain that these experiments only concern garbage collection on tuples, and not on TSs.
3. Each agent requests their own multicapability with no capabilities being passed among the agents, which implies that all the agents used different multicapability regions. Thus, these regions cannot be referenced by any other agents, and are considered garbage when the agents creating them die.

Results for GC using Reference Counting

The experiments involved running a group of agents in limited memory space, where each agent requests a new multicapability, outs a number of tuples into the UTS using the newly acquired multicapability, and then dies after explicitly deleting the multicapability. The agents’ code snippet is given below.

As expected, the server with no tuple-garbage collection eventually ran out of memory, whereas the server with garbage collection did not encounter the same problem, in fact, was able run indefinitely.

Figure 5 depicts the results of the experiments: with (a) 5000 tuples outed per iteration, (b) 7000 tuples, and (c) 10000 tuples. Each graph shows that without garbage collection on tuples, the system crashes (runs out of memory) sooner than its GC-enabled counterpart, i.e., after approximately 67000 tuples have been deposited in the UTS, whereas there is no such concern with the system with garbage collection on tuples.

When garbage collection on tuples is not present, tuples written by previous agents are left accumulating in the UTS (in effect, stored in the kernel); this adds considerable overhead to the system. Although none of the tuples would be used again, they are consuming the memory resources—causing the system to crash if the memory is not reclaimed (by the garbage collector). With tuple-garbage collection incorporated, we have a cleaner structure, hence better performances for agents—thus, tuple-GC
Algorithm 6.

```c
cap = newcap<int,int>();
for (t = 0; t < 10000; t++)
    UTS.out(cap<1,t>);
del cap;
```

Figure 5. Memory exhaustion problem

(a) Increment of 5000 tuples per iteration

(b) Increment of 7000 tuples per iteration

(c) Increment of 10000 tuples per iteration

provides the system with a major advantage in terms of reliability.

Of course, no garbage collection mechanism can completely eliminate the memory exhaustion problem, as memory is a finite resource—there is always a possibility of completely consuming the resource without creating any garbage.

Incorporating the garbage collection mechanism incurs some overhead. Again, the two LINDACAP systems are compared: both incorporate capabilities, but only one has the garbage collection mechanism on tuples. Figure 6 illustrates that no significant increase in overhead (measured by completion time for each agent) is imposed by the garbage collection algorithm.

As can be seen from the graph, the larger the number of tuples output, the less overhead (proportionally) incurred. Comparing the time taken by each system (with, and without tuple-GC) to write a certain number of tuples, in Figure 6a, the time difference in outputting 100 tuples is 0.001 second (only 1.5% increase in
overhead), whereas writing 1000 tuples produces a difference of 0.023 second (i.e., 6%, which is the same time difference as in Figure 6b). In Figure 6b, where tuples are written in increments 1000, the system with garbage collection on tuples took 0.24 second longer (7%) to write 10000 tuples, compared to the one without tuple-GC.

Results for GC using Mark-and-Sweep

The main reason for garbage collection is to overcome memory exhaustion. In order to show the occurrence of memory exhaustion, an experiment was performed with limited memory size (the heap size is limited to 12517376 kb) where agents repeatedly write tuples until the server exhausts its capacity to store the tuples. In these experiments, three LINDACAP systems were compared for memory exhaustion: the first two systems incorporate garbage collection mechanisms for tuples based on reference counting and mark-and-sweep, respectively, whereas the third one does not incorporate any garbage collection mechanism. In addition, the time-to-idle technique is implemented in the first two systems. Figure 7 depicts the experiments that cause memory exhaustion. In Figure 7a an agent repeatedly writes $n \times 3000$ tuples in each iteration, where $n$ is the iteration number.

In the system without garbage collection on tuples, the server crashed (runs out of memory) when an agent was about to out 15000 tuples at a time, whereas both the other systems (with tuple-garbage collection) tolerate the system failure, because unused tuples are deleted and memory is reclaimed by the garbage collector. Figure 7b depicts similar results for 5000 tuples increment in each iteration, when the memory crashed during the attempt to write 20000 tuples in the system without garbage collection.

Figure 8 shows the overhead caused by the tuple garbage collection schemes, comparing the three systems described above. Figure 8a shows the case where the agent stores tuples with the number of tuples are increased by 5000 in every iteration, and the average overheads measured due to the reference counting and mark-and-sweep techniques are around 6.92% and 8.33%, respectively. Figure 8b shows the case where with 10000 tuples increment in every iteration, and the average overheads measured are around 6.59% for reference counting and 8.67% for mark-and-sweep.

As can be seen from the average overhead, a slight overhead is added by the garbage collection techniques using mark-and-sweep and time-to-idle, and the difference between the average overhead caused by the mark-and-sweep and reference counting is small and can be considered insignificant if the benefits

Figure 6. Overhead added by tuple garbage collection scheme
brought by the mark-and-sweep is taken into account.

Incorporating the TTI garbage collection technique in a system imposes some overhead. In order to show the overhead due to TTI technique, two LINDACAP systems are compared. Both systems have garbage collection mechanism using reference counting or mark-and-sweep algorithm, but only one has TTI garbage collection mechanism. Figure 9 depicts the experimental results where reference counting is incorporated on both systems but only one has TTI. In Figure 9a, where agents store tuples in 5000 increments, the average overhead imposed due to TTI was around 3.35%. The result was almost similar for the experiment with 10000 tuples increment, i.e., 4.28% (Figure 9b). As expected, incorporating TTI imposes significant overhead on a system that already has garbage collection using reference counting algorithm.

In case of Figure 10, mark-and-sweep is implemented on both systems instead of reference counting. The average overhead due to TTI was 3.62% for 5000 tuples increments in each loop (Figure 10a), and 5.05% for 10000 tuples increments (Figure 10b). According to the above results, the incorporation of TTI technique adds a slight overhead on the performance of the system.

**Garbage Collection Memory Usage**

When any garbage collection algorithm is implemented in any system, some memory overhead is expected. In order to show the memory usage by the garbage collection algorithms, an experiment was performed where an agent creates a number of multi-capability regions and therefore increasing the number of nodes in the graph. In this experiment, two LINDACAP systems were compared: the first system incorporates a garbage collection mechanism based on reference counting whereas a mark-and-sweep algorithm is implemented in the second system. In addition, the time to idle technique is implemented on both systems. The memory usage due to each garbage collection algorithm was collected.

In Figure 11a an agent loops and outs $n \times 100$ multi-capability regions in each iteration, where $n$ is the iteration number. The system with mark-and-sweep garbage collection algorithm uses more memory resource compared to the one with reference counting. This shows the fact that in tracing-based algorithms such mark-and-sweep, unreachable (garbage) objects are not reclaimed immediately when they become unreachable. This dramatically increases the memory overhead and degrades system performance. This can be a problem in a system that interacts with a human user or that must satisfy real-time execution constraints. Figure
Figure 8. Overhead added by the tuple garbage collection schemes

Figure 9. Overhead added by the incorporation of TTI in a system with tuple garbage collection based on reference counting

11b depicts the same case but where the increment was 500 regions per iteration.

**FINER GARBAGE COLLECTION**

With capabilities, a finer garbage collection mechanism can be achieved. Ligia only incorporates object/TS references, but LINDACAP can manipulate the reference and the associated permissions (in the capability) for finer control: rights can be encoded in the edges. Consider an example as illustrated in Figure 12. A tuple containing a capability for ts2 is emitted into ts1, forming a bridge from ts1 to ts2. ts2 itself contains a capability tuple for ts3, creating another bridge from ts2 to ts3.

In a non-capability system, if there exists at least one agent, A1 that has the reference for ts1, then as long as A1 is alive, none of the three TSs can be garbage, as there is always a possibility that A1 might retrieve the tuple (ts2) in ts1, and subsequently gaining access to ts3 via ts2. (Here it is assumed that no intervention occurs resulting in the tuples being removed except by A1.) In LINDACAP on the other hand,
the rights in the capability for ts1 held by A1 is relevant: if the capability only grants out permission, then A1 cannot read the tuple (ts2) from ts1, and therefore cannot access ts2, and subsequently ts3, too. Hence, ts2 and ts3 can be garbage collected. In the case of ts1 itself, although ts1 is not garbage—since A1 still has the capability for it and may write into it—any reference/capability within it is garbage. In fact any tuple within it is garbage. The tuple space ts1 can be regarded as a “null”, where anything can be written into it, but they will not be retained—much like a dumping ground—and therefore can be flagged as such, for optimization purposes, to ensure that tuples stored in it will not increase memory usage. A null capability—i.e., a capability that does not grant anything, and holding such a ‘null’ capability is useless for the agent’s access to the object—can be useful for garbage collection. For instance, if the kernel knows that all tuples containing the capability for an object (a TS or a group of tuples) are null, and no other agent holds a copy of the capability, then it knows that the object is garbage, as no operation can be performed on it, therefore the object can be reclaimed.
CONCLUSION

Capabilities represent the information on 'who knows about what operation on a certain object'. The more the kernel knows of the system's behaviour, the better, more optimised coordination can be achieved, thus increasing the system's efficiency. The extra information, supplied by the capabilities given to the agents, can provide the facility to create a finer level of control in distributed systems.

Having a finer control over coordination aspects also means obtaining a better, more efficient decentralised resource management. We have demonstrated how multicapabilities—which extends capabilities to apply to a group of un-named objects—can contribute towards improving one important aspect in managing resources in a distributed way: garbage collecting unused tuples (or a specific region of a TS).

In this paper, we have presented the garbage collection technique implemented in LINDACAP using reference counting. However, this technique alone does not garbage collect unused tuples which are part of a cyclic structure. Therefore, we have also incorporated the mark-and-sweep garbage collection algorithm (Dijkstra, 1978) as a solution to this problem. This technique allows the system to remove unusable tuples in a cyclic structure and reclaim the memory space. In addition, a time-to-idle (TTI) technique has also been proposed to reclaim the memory storage for multicapability regions that are being referred to by agents but no longer in use.

The performance results indicate that the incorporation of garbage collection techniques adds a slight (almost insignificant) overhead to the overall performance of the system. And also the difference between the average overhead caused by the mark-and-sweep and reference counting is small. This is not such a big loss compared to a possibly more disastrous consequence, such as a memory crash, especially in open, persistent systems.

ACKNOWLEDGMENT

The authors are grateful to Dr. Alan Wood of University of York, UK, for his helpful comments in the earlier stage of this work.

REFERENCES


Copyright © 2010. IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
Nur Izura Udzir is a Senior Lecturer at the Faculty of Computer Science and Information Technology, Universiti Putra Malaysia (UPM). She received her Bachelor of Computer Science (1996) and Master of Science (1998) from UPM, and her PhD in Computer Science from the University of York, UK (2006). She is a member of IEEE Computer Society. Her areas of specialization are access control, secure operating systems, coordination models and languages, and distributed systems. She is a member of the Information Security Group at the faculty.

Hamidah Ibrahim is currently an Associate Professor at the Faculty of Computer Science and Information Technology, Universiti Putra Malaysia. She obtained her Ph.D. in Computer Science from the University of Wales Cardiff, UK in 1998. Her current research interests include distributed databases, transaction processing, and knowledge based systems.

Silesi Demesie graduated from Addis Ababa University with his Bachelor of Computer Science in 2005. He received his Master of Computer Science degree in 2009 from Universiti Putra Malaysia (UPM), majoring in Distributed Computing. He is currently working in the Faculty of Engineering, Bahir Dar University, Ethiopia, and currently a member of the System Design and Development case team there.