DISTRIBUTED GARBAGE COLLECTION OF ACTIVE OBJECTS: A TRANSFORMATION AND ITS APPLICATIONS TO JAVA PROGRAMMING

BY

ABHAY VARDHAN

B. Tech., Indian Institute of Technology, Delhi, 1995

M.S., University of Illinois, 1997

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign,1998

Urbana, Illinois

To Varsha.

Abstract

Garbage collection is a valuable service which provides error-free and automatic management of system resources. A substantial amount of work has been done in the area of garbage collection in uniprocessor and distributed systems. Garbage collection techniques and algorithms have also been proposed for actor systems, but traditionally, algorithms developed for non-actor systems are considered unsuitable for actors as they do not take into account the "processing-power" inherently present in each actor. This thesis presents a *transformation* of the actor-reference graph into a *passive object* graph with the property that if a garbage collector for a general object-oriented system is run on the transformed graph,only those objects are collected which correspond to garbage actors in the original graph. A proof of correctness of the transformation is given and the costs associated therewith are discussed. The transformation enables us to reuse many algorithms already developed for other systems, to find garbage actors. This is of increased importance for distributed systems because garbage collection in such systems still faces a number of challenging issues prompting the need for investigation of new algorithms.

We demonstrate the application of the transformation technique by adapting a known garbage collector for object-oriented systems for collecting garbage actors in a distributed environment. In our scheme, local garbage collection is allowed to proceed independently on each site and a global garbage detection service collects garbage which cannot be recognized on the basis of local information alone. The issues associated with the distributed nature of the algorithm are discussed. The algorithm has been implemented on an actor system based on Java^{TM}. Implementation details are discussed and performance results are presented.

Acknowledgements

I would like to thank my advisor, Prof. Gul Agha, for his guidance and encouragement throughout the development of this thesis.

I enjoyed being a member of the Open Systems Laboratory and I appreciate the useful discussions that I had and the help that I received from my officemates, in particular, Mark Astley, Nalini Venkatasubramanian, James Waldby and Prasannaa Thati. I am indebted to Mark and James for reviewing this thesis. I would also like to thank Prof. L. V. Kale for his encouragement during the course of my Masters degree at the University of Illinois. It has also been my pleasure to have worked with Prof. Bill Dunn in the Mechanical Engineering Department.

I would like to thank all my friends who made my stay in Urbana so enjoyable. I am greatly indebted to my wife Varsha, without whom none of this would have been possible. I thank her for her love and support which kept me going and for being understanding and uncomplaining even when the long hours that I spent working on my thesis kept me away from her.

As I look back at the time I have spent at the University of Illinois, I would like to say that this period has been very constructive, educational and memorable for me.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Manually-programmed reclamation of dynamically managed store is notoriously error-prone. Because of the additional complexity of distributed systems, with the increasing use of such systems, the issue of automatic reclamation or *garbage collection* has increased in importance. Garbage collection can help improve programmer productivity and quality of the software produced. However, this benefit is not without cost. There is always some overhead associated with garbage collection. There are some applications in which this additional overhead might not be acceptable. But for a large number of applications it is indeed beneficial to have automatic garbage collection.

In this thesis, we have analyzed the problem of garbage collection in actors (the reader is referred to 2.1 for a description of what an actor is). Garbage collection of actors becomes even more important because actors consume not only memory but may also consume CPU resources.

Garbage collection of actors has been treated as a problem different from garbage collection in non-actor systems (such as traditional object oriented systems). Most algorithms developed for such systems are unsuitable for actor systems directly. In this thesis, we present a transformation that allows these algorithms to be applied to actors. The transformation also provides an elegant method to integrate garbage collection of active and passive objects in systems that support both kinds of objects. The transformation is the main contribution of the thesis. We demonstrate how a particular algorithm that is used in traditional object oriented systems can be adapted for garbage collection in actors. The environment in which the garbage collector

has been implemented is distributed. Problems related to concurrency and asynchrony of distributed systems are discussed. The transformed algorithm has been implemented in an actor system based on Java^{TM}.

1.1 Road-map to the thesis

The remainder of the thesis is organized as follows:

- Chapter 2 explains some important concepts which are used later. The actor model of computation is presented and the distinction between garbage collection in object oriented systems and actor systems is described.
- Chapter 3 reviews related garbage collection work done in uniprocessor systems, distributed systems, object oriented systems and actors.
- Chapter 4 describes a transformation that allows garbage collectors developed for other systems to work with actors.
- Chapter 5 demonstrates the application of the transformation technique to adapt a particular garbage collector for distributed object oriented system for actors.
- Chapter 6 discusses issues encountered in the implementation of the above algorithm and gives test results.
- Finally, Chapter 7 gives conclusions, a brief summary of the thesis, and suggested areas of future work.

Chapter 2

Background

This chapter provides background for concepts used later in the thesis. The model of computation in actor systems is explained and garbage collection in both non-actor and actor systems is discussed.

2.1 Actor systems

In the Actor model [2], the universe consists of autonomous computational agents called *actors* that encapsulate data as well as some primitive processing power to manipulate data. Each actor has a unique *mail address* which can be used to communicate with that actor. Communication between actors is asynchronous with unbounded delay but guaranteed delivery. Messages may be received in an order different than sent. Messages for an actor are buffered in a *mail queue*.

Computation in an actor system is message driven. Each actor processes messages in its mail queue one by one in the order they were queued. Processing a message involves executing a script which is called the *behavior* of the actor.

In response to a message, an actor may *create* new actors, *send* messages to actors, and *change its state* with which it responds to the next message (Figure 2.1). These actions are implemented by extending a sequential language with the following operators:

- **create** takes a behavior description and creates an actor. The operator may take additional initialization arguments.
- **send** takes the receiver's mail address and puts a message into its mail queue.

Figure 2.1: Primitive operations in the Actor model.

- **ready** changes its behavior to the one specified and prepares to process the next message in the mail queue.

The actor model provides a powerful abstraction for concurrent and distributed systems. By encapsulating a thread of control along with data, a number of synchronization problems that often plague concurrent object-oriented systems are avoided.

The graph constructed from actors as nodes and their references as edges is called the *actor-reference graph*. Mail addresses may be communicated through messages leading to dynamic changes in the actor-reference graph.

2.2 Garbage collection

Garbage collection refers to the automatic reclamation of resources which have become unusable by the user program. It is employed most commonly to reclaim dynamically allocated memory. Garbage collection has been used extensively in functional languages such as Lisp and Scheme. Object-oriented languages such as Java and Smalltalk also support garbage collection.

Although garbage collection has been applied to a variety of languages, we shall focus our attention on garbage collection for object-oriented systems and actors only. The issues involved in other types of languages are analogous to those in object-oriented systems and actors.

2.3 Garbage collection in object-oriented systems

Garbage collection of the heap in an object-oriented system can be viewed as an abstract graph problem. A directed graph is formed with objects as nodes and object references as edges. We call this graph an *object-reference graph*. A subset of objects, such as statically declared objects, input-output objects, objects on the program stack and objects referenced from the CPU registers, are considered to be non-garbage and are called *root* objects. The problem of garbage collection is reclaiming all the objects which cannot affect the user computation. This is equivalent to finding objects which are not connected to any root object by a path through the edges of the graph. The top part of Figure 2.2 shows an object-reference graph. It can be seen that objects 6, 7, 9, 10, 11 and 12 are garbage.

2.4 Garbage collection in Actors

Garbage collection in actors has some similarities with garbage collection in *passive* [1] object-oriented systems but has some important differences too. We shall make a comparison in this section.

In passive object-oriented systems, the model of computation is that of independent threads of control manipulating the objects they can refer to. Implicit in this idea is that the threads of control by themselves are always important. Thus, an important subset of the root set consists of the the objects referenced from the stacks of running threads.

In actor systems, there is no notion of an independent thread of control manipulating data. Instead, the thread of control is encapsulated in the object itself. Conceptually, there are no stacks, registers or a separate heap (an actual implementation however may still have these). The idea that all threads of control are important to the underlying computation is no longer valid; no actor would otherwise ever become garbage because each actor is referenced by the

[1] We emphasize that objects in traditional oriented-object systems are "passive" as opposed to actors being "active".

thread of control it encapsulates. To identify garbage in an actor system, we have to take a look at how an actor system performs some computation. Conceptually, a single actor called the root actor can be deemed to start the computation (if more than one actor exists at startup, one can designate a hypothetical "root" actor that directly references the startup actors). This actor may interact with the user and hence may be relevant to the computation at all times. As the computation proceeds, the root actor can create more actors that in turn may create other actors. Messages are passed from actor to actor, in response to which some progress is made in the underlying computation. Any actor that is unable to receive a message from or send a message to the root, directly or through intermediate actors, is therefore irrelevant to the underlying computation and should be regarded as garbage. This illustrates an important difference in garbage collection between actor systems and passive object-oriented systems.

2.4.1 Formal definition of garbage in actor systems

We now present a formal definition of garbage in actor systems.

The *root set* of actors is the set of actors which directly communicate with the external environment and are always considered non-garbage. An actor which is either processing a message or has messages pending in its mail-queue is called an *unblocked* actor. An actor which is not unblocked is called a *blocked* [2] actor.

If an actor A has a reference to an actor B, A is said to have a *forward acquaintance* or just an *acquaintance* to B. B is said to have an *inverse acquaintance* to A.

A blocked actor which is not connected by the recursive closure of the inverse acquaintance relation to an unblocked actor is called a *permanently blocked* actor. The set of *live* actors is then defined recursively as:

1. A root actor is live.

2. Every forward acquaintance of a live actor is live.

3. Every inverse acquaintance of a live actor which is not permanently blocked is live.

[2] We use the terms blocked/unblocked instead of active/inactive to avoid confusion with the term active as used in a system of active objects.

An actor which is not live is *garbage*. The property of being garbage is a stable property, in the sense that if an actor is garbage in a given state it will remain garbage in any possible state deriving from that state.

It is possible to argue that all unblocked actors should be treated as non-garbage. This might be based on the conservative assumption that any unblocked actor can affect the environment by interacting with the file system, the display unit, network and other system related units. In terms of the actor model, these system related units can be viewed as "well-known", i.e. every actor has their "mail address". However, such an approach leads to some actors being regarded as non-garbage although they can never influence the user computation. For instance, a pair of unblocked actors which are unconnected in any way to all other actors or system related units but repeatedly pass messages to each other will not be reclaimed although their existence is clearly a waste. In this thesis, we do not follow this approach. Even in our system, it is possible that an actor which is considered non-garbage is never able to influence the environment. Thus, a semantic analysis of an actor program might reveal that there is an actor A referenced only by a root actor B, but B never actually uses the reference it has for A. In this case, A is effectively garbage. Our approach does not operate at the semantic level and assumes that if an actor B has a reference to an actor A it could potentially use that reference.

Figure 2.2 shows two similar graphs. The top one is a reference graph for passive objects while the bottom one is a reference graph for a system of actors. The structure is the same in both graphs, with added information about whether an actor is blocked or unblocked being indicated in the actor graph. In both graphs, an object or an actor reachable from the root is considered live. However, object 6 is considered garbage while the corresponding actor 6 in graph (b) is live. Actor 6 has to be considered live because, being unblocked, it can send a message to actor 4 passing a self reference thereby making itself connected to the root (notice that 4 is connected to the root via 3).

We see that there are some differences between the garbage collection of passive objects and actors. These differences have led to specialized algorithms being developed for actors. In the Chapter 4 we show there is a transformation by which the actor-reference graph can be transformed to reuse a passive object garbage collection for actors.

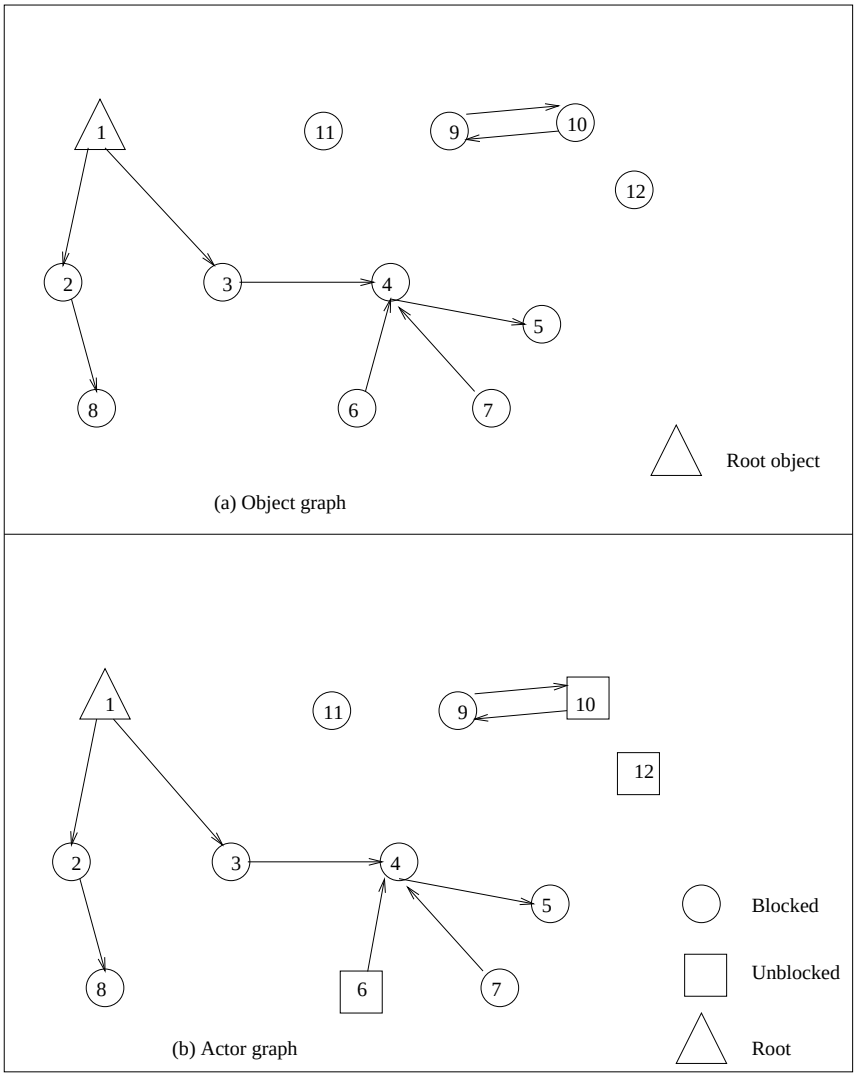

Figure 2.2: Garbage collection in passive objects and actors

Chapter 3

Related Work

In this chapter, we review various garbage collection schemes that have been published in the literature for uniprocessors, multiprocessors, passive objects and actors.

To introduce some concepts common for all garbage collection algorithms, we start by considering uniprocessor garbage collection.

3.1 Uniprocessor garbage collection

The problem of garbage collection in uniprocessors has been well studied. An excellent survey of uniprocessor garbage collection techniques is given by Wilson [40]. Algorithms for garbage collection for both uniprocessors and distributed systems are described by Jones in [14].

We now discuss different categories of uniprocessor garbage collection algorithms.

3.1.1 Reference counting

Reference counting is based on the idea that if an object is no longer being referenced by any *other* object then it is clearly unreachable from any root object (in fact it is unreachable from *any* object). For each object, a count of the number of objects which hold a reference to it is maintained (this count is called the *reference count* for that object). When an additional reference to this object is created the reference count is incremented and when a reference to this object is deleted the reference count is decremented. The object is identified as garbage as soon as the reference count falls to zero.

The advantage of this scheme is that it is simple to implement, identifies garbage incrementally (*i.e.* execution of the algorithm is interleaved with the execution of the user program), and an object is identified as garbage as soon as it becomes garbage. A major disadvantage is that this scheme is unable to collect cyclic structures. There is also considerable processing overhead as the reference count has to be updated frequently. Memory has to be reserved for every object to hold the reference count.

3.1.2 Mark-sweep

Mark-sweep relies on a global traversal of all live objects to determine which objects are garbage. The algorithm proceeds in two stages: a mark phase and a sweep phase. In the mark phase, a depth-first or a breadth-first search is performed on the object-reference graph starting from all the root objects and a *mark* bit is set on all the objects encountered. The mark phase is followed by the sweep phase in which objects which are left untouched by the mark phase are reclaimed or "swept" (such objects are clearly unreachable from any root).

Mark-sweep has the advantage that it can reclaim cyclic structures also. There is no processing overhead for manipulating pointers. The disadvantage is that the algorithm is not incremental and unless special procedures are applied, the user process has to be paused while marking is being done.

3.1.3 Copying collector

A copying collector is similar to a mark-sweep collector except that it also compacts the heap. Copying collectors divide the heap equally into two *semi-spaces*, one of which contains the current data and the other contains obsolete data. The collector starts by flipping the roles of the two spaces. Then all live objects in the old semi-space, *Fromspace* are copied to the other semi-space, *Tospace*, by tracing from the root set in *Fromspace*. The garbage objects remain in *Fromspace* and the entire memory associated with them is made available at the next collection time.

3.1.4 Generational garbage collection

In generational garbage collection, objects are segregated on the basis of their age into two or more regions of the heap called *generations*. Different generations can then be collected

at different frequencies with the youngest generation being collected most often. This takes advantage of the observation that most objects die young and so the efforts of the collector are concentrated in the region where most benefit is likely to be achieved. Different policies can be used to *tenure* an object from a young generation to an old generation.

3.2 Distributed garbage collection

In distributed systems, object references may cross site boundaries. This makes garbage collection in a distributed system much harder than in a uniprocessor system. Hosts must be synchronized across the network, termination detection of the collector becomes non-trivial and network partitions and host crashes have to be taken into consideration. Schemes have been presented in the literature to solve this challenging problem while achieving efficiency, scalability and fault-tolerance. We first discuss the methods based on reference counting followed by tracing based collectors and finally we review the algorithms that fall into neither category.

3.2.1 Reference counting based distributed garbage collection

Compared to reference counting in uniprocessors, additional problems arise in distributed systems because of the fact that communication between hosts has to be done through messages which might not be delivered in causal order. Therefore naive versions of reference counting would have race conditions which might lead to incorrect garbage collection. To solve this, many algorithms have been presented.

3.2.1.1 Weighted Reference Counting

Bevan [7] and Watson and Watson [39] independently proposed the use of Weighted Reference Counting (WRC). In this scheme, each reference to an object is associated with a weight instead of a simple count of 1 as in the case of reference counting. A newly created object is given an arbitrary *total weight* to start with and the first reference to this object has this as the weight. When an existing reference is copied, its weight is reduced by the amount "given" to the new copy. On reception of such a copy, the recipient uses the weight coming with the message as the weight of the reference. When a reference is destroyed, a message is sent to the referred-to

object asking it to decrement its *total weight* by the amount that the destroyed reference held. When the *total weight* reaches zero, the object can be reclaimed. The invariant maintained at all times is that the sum of the weights contained in all the references to a particular object is equal to the *total weight* residing in that object.

$$TotalWeight = \sum PartialWeights$$

The main problem with WRC (apart from the inherent shortcomings of reference counting) is that there is a possibility of underflow of the weights. Dickman [11] proposed an optimized version of WRC which makes it resilient to message loss and handles the problem of underflow.

3.2.1.2 Indirect reference counting

Piquer [28] proposed Indirect Reference Counting as an alternative to naive reference counting. In this algorithm, each remote reference for a non resident object has two fields associated with it. One is a counter and the other is a pointer. A counter is also associated with the actual object being referred to. When a remote reference is created, the counter associated with the object is incremented and the pointer in the remote reference points directly to the object. When a reference is duplicated, the pointer of the newly created reference is made to point to the reference it was copied from and the latter's counter is incremented. Thus all the remote references denoting an object form a "diffusion tree" rooted at the actual object. When a remote reference is deleted a decrement message is sent to the parent of the reference (in the diffusion tree) and its counter decremented. If a remote reference is no longer locally reachable but is not a leaf in the diffusion tree, it is not discarded until its counter is zero (after all its children have been deleted). Once the diffusion tree gets pruned to the root of the tree and the counter of that object falls to zero, the object can be reclaimed.

3.2.1.3 Reference listing

Reference listing is similar to reference counting except that instead of maintaining a count of the references held for an object, the actual list of objects which hold the references is kept. This improves resilience to message and space failures at the cost of some memory overhead.

Shapiro *et al* [35] and Birrel [8] use this technique for distributed garbage collection.

3.2.2 Hybrid techniques

Reference counting and reference listing are unable to collect cyclic garbage but they do not require synchronization among processors and are scalable. Therefore hybrid schemes have been proposed that take advantage of the scalable nature of reference counting and try to augment it so that cyclic garbage can also be collected.

Some techniques propose running a distributed tracing algorithm at a low frequency in addition to the reference counting techniques to reclaim cyclic garbage. Bishop [9] proposes an object migration technique where objects suspected of belonging to a garbage cycle are migrated to a single node where they can be collected by a cyclic local garbage collector.

Rodriguez-Rivera [32] proposes an algorithm based on reference listing augmented with back tracing. For an object suspected to be garbage, the algorithm recursively traces back the references to find the transitive closure of objects that reach the suspect. If this closure does not contain any root, the objects in the closure are recognized as garbage. A similar approach is followed by Maheshwari [25].

Rodriguez [31] suggests an algorithm based on reference listing with partial tracing in order to collect cyclic garbage. Tracing is initiated at an object suspected to be a part of a garbage cycle. A mark phase identifies the processes which will form a group to collaborate in order to reclaim distributed cyclic garbage. A scan phase then finds out whether any member of the marked subgraph is reachable from outside the subgraph. Finally a sweep phase reclaims any inaccessible cycles.

3.2.3 Tracing-based distributed garbage collection

The major problem with distributed tracing is in synchronizing the distributed mark phase with the distributed sweep phase. All the hosts must come to an agreement that the mark phase has completed before the sweep can be begun. This makes this scheme difficult to scale. It is also not resilient to network partitions and host crashes as it relies on the cooperation of all the processors.

Augusteijn [4] presents an algorithm based on an incremental three-color mark-sweep algorithm applied to a distributed object system. Juul [15] proposes another distributed version of

the incremental mark-sweep algorithm. The protocol to determine the termination of the mark phase is described as a two-phase commit protocol.

Ladin and Liskov [19] propose an algorithm that relies on a logically centralized global garbage detection service. The service may be physically replicated for higher availability. Each site maintains a list of objects that have been referenced from other sites. These are treated as roots for local garbage collection. Information about these objects and related information, is passed to the global garbage detection service. Messages to the service are time stamped with vector clocks so that the global garbage detector can form a consistent cut of the distributed system. The local garbage collector periodically queries the global garbage collection service about the elements of its alleged roots that are no longer remotely reachable.

Hughes [13] describes an appealing algorithm where mark bits are replaced by timestamps. The key idea is that a garbage object's timestamp remains constant whereas a non-garbage object timestamp increases monotonically. A global threshold is computed and any object whose timestamp is lower than the threshold is recognized as garbage.

Fessant *et al* [21] present an algorithm based on Hughes' algorithm in a simplified form that makes fewer assumptions about the distributed system. They use a logical clock to timestamp events. A reference-listing-based algorithm is used to collect acyclic distributed garbage and the timestamp algorithm can be invoked to collect the cyclic part of the garbage.

Lang, Queinnec and Piquer [20] suggests combining reference counting and mark-sweep in order to perform garbage collection within groups. A group is a dynamic collection of spaces which are built by *group negotiation*. Mark-sweep is carried out within the group and reference counting is used to collect objects whose references cross groups. Scalability of the garbage collector is achieved through a hierarchy of nested groups.

Ali [26] presents an algorithm that allows each processor to mark-sweep its own heap independently. At the end of a local garbage collection, the processor informs all other processors of the remote pointers that it retains, so that they may treat the corresponding objects as roots in their own local garbage collection. This algorithm is unable to collect global cycles.

3.2.4 Other algorithms

Schelvis [33] proposed a comprehensive global garbage collection algorithm based on time-stamp packet distribution. In this algorithm, a table called an *entrance table* is kept of all the local

objects that are remotely referenced. These are included in the root set for local garbage collection. An *entrance graph* is formed with nodes being the entrance nodes (the entries in the entrance table) and edges being the set of local paths from some entrance node to a remotely referenced object. Local garbage collection on different hosts proceeds independently while a global garbage detection strategy tries to reclaim the entrance nodes which have become garbage. For global garbage detection, packets are asynchronously and repeatedly sent to each remotely referenced object. Each packet contains some information about the connectivity of the node from which it comes, so that it is possible to recognize garbage once sufficient information has been collected. The algorithm does not require any synchronization among different processors but is still able to collect all garbage. We have used this algorithm in our implementation of global garbage collection. The details are presented in Section 5.7.

Louboutin [23, 24, 22] presents an interesting algorithm which is able to collect all distributed garbage by tracking causal dependencies of relevant mutator events. Each host performs its local garbage collection independently and builds a *global root graph* which is the *entrance graph* in Schelvis' terminology described in the previous paragraph. Each global root (entrance node in Schelvis' terminology) maintains a log of timestamps of certain events received from adjacent global roots. These events, called *log-keeping events*, are related either to creation or deletion of an edge in the global root graph. A *path-history* of a log-keeping event is defined as a subset of its causal history containing only those events responsible for the creation of existing paths to the object in question. Knowing the path history of events it is possible to identify garbage in the global root graph. A *lazy log-keeping* mechanism is used to prevent race conditions between different control messages. Timestamps called *dependency vectors* are propagated along the paths of the global root graph to characterize the path histories of the log-keeping events. An edge-destruction event might result in a path history which shows that the given global root has become unreachable from any root. The algorithm is reactive, incremental, scalable, does not require synchronization among processes, and is able to collect cyclic garbage.

However, some implementation issues are unclear in the given algorithm. The algorithm requires that log-keeping events be recorded for each global root. But the global root graph is reconstructed only after local garbage collection completes at some host. The actual mutator events happening in non-global-root objects in that host are not tracked. It is unclear how the log-keeping events relevant to the global root graph are recognized.

3.3 Actor garbage collection

As we saw in Chapter 2, the problem of garbage collection of actors is slightly different from garbage collection for "passive" objects. Traditionally, it has been felt that special algorithms must be devised for garbage collecting actors. In Chapter 4 we show it is possible to transform the actor-reference graph so that a "passive" object collector can work on it. But before we describe that, we review the existing algorithms that have been formulated for actors.

A formal definition of garbage actors was first given by Kafura [17]. The basis of the algorithms presented in his work is marking actors with three different colors that have the following meanings:

- **White**: It has not been shown this actor can communicate with a root actor
- **Gray:** This actor is blocked but can communicate with a root actor if it can become unblocked which has not yet been shown.
- **Black:** This actor is non-garbage.

Nelson [27] formulated certain coloring rules to identify actor garbage. These are given in Algorithm 3.1.

1. All actors are colored white, with the exception of root actors which are colored black.
2. Repeat the following rules until no more markings are made:
   (a) Color black all acquaintances of black actors.
   (b) Color black all inverse acquaintances of black actors if the inverse acquaintance is not blocked.
   (c) Color gray all inverse acquaintances of black actors if the inverse acquaintance is blocked.
   (d) Color black all inverse acquaintances of gray actors if the inverse acquaintance is not blocked.
   (e) Color gray all inverse acquaintances of gray actors if the inverse acquaintance is blocked.
3. Actors that are black are not garbage; all other are garbage.

Algorithm 3.1: Nelson's Coloring Rules

Based on these rules, Kafura presents two algorithms, *Push-Pull* and *Is-Black* algorithms. In the *Push-Pull* marking algorithm, there are two coroutines, a Pusher and a Puller, to move actors between black, gray and white sets. The Pusher operates on the white set and tries to push a white actor into the gray or black set depending on whether it has an acquaintance which is black or gray. The Puller pulls acquaintances of black actors out of the gray or white set into the black set. In the algorithm *Is-Black*, one rule repeatedly colors black the acquaintances of black actors. A second rule does a depth first search from all active actors for a black actor. If a black actor is found, then the originating actor is colored black. Both these algorithms have a space complexity of Order(N) and time complexity of Order(N^2) where N is the number of actors in the system.

Washabaugh [38] has described a real-time and distributed version of actor garbage collection based on the *Push-Pull* and *Is-Black* algorithms. In a more recent paper, Kafura *et al* [16] have described in detail a distributed version of the Push-Pull algorithm. Local garbage collection at different hosts is allowed to proceed independently. A distributed global collector is invoked for collecting garbage not recognizable by the local collectors. The collector uses previously available algorithms for taking consistent snapshots in a distributed system and for detecting termination. FIFO delivery is assumed between any two nodes of the network and inverse acquaintances are not explicitly maintained. The garbage collector is able to collect cycles of active actors which cannot send any message to any live actor.

A hierarchical distributed garbage collection algorithm is described by Venkatasubramanian *et al* [36, 37]. An approach similar to mark-sweep is followed with specialized marking rules which are formulated according to the definition of garbage in actors. A fixed network topology is assumed. A broadcast to all hosts initiates the process of garbage collection. A snapshot of the global system is formed and messages in the network sent before the initial broadcast are accounted for by sending "bulldoze" messages across the network. Actors are partitioned into different generations to take advantage of the temporal locality of garbage. The distributed system is partitioned into clusters which are organized hierarchically to avoid the bottleneck of computation and resource management. The mark-sweep procedure followed in the algorithm is reproduced in Algorithm 3.2. A slight change in terminology has been made from Venkatasubramanian's algorithm. Instead of using the terms *touched, suspended* and *untouched*

for different possible states of actors, we use the terms black, gray and white respectively to remain consistent with Kafura's terminology.

A garbage collector for active objects in an MPP environment has been described by Kamada [18]. In this work, all active objects are considered live. An interesting feature of their algorithm is the use of a centralized agent which handles many problems related to synchronization and detection of termination of various phases of the garbage collection.

Puaut [29, 30] presents an algorithm comprising of independent local collectors loosely coupled to a global collector. The global collector is a logically centralized service that maintains a graph which is a merge of subgraphs sent by the local garbage collectors. The local collectors send a vector timestamp along with the message containing the subgraph relevant to their host. Based on these timestamps the global collector can decide if a consistent vision of the global distributed graph has been formed, and proceeds to garbage collect only in the case that consistency has been achieved.

In a recent paper, Dickman [12] presents an interesting algorithm called the *Partition Merging algorithm* (PMA). A key idea of the algorithm is that all actors reachable from an unblocked actor (including the unblocked actor itself) have the same garbage status. Moreover, if the set of reachable actors from an unblocked actor overlaps with the set of reachable actors from another unblocked actor then actors in both sets have the same garbage status. The main action of PMA is to form sets of actors having the same garbage status due to their being reachable from one or more unblocked actors. The algorithm proceeds in such a way that each such set can be traversed by an Eulerian cycle[1]. Once all the sets have been formed, each Eulerian cycle is traversed to see if any actor in the set is reachable from a root actor. The use of Euler cycles enables the traversal to be completed in linear time. If there is at least one such actor, the entire set is live otherwise the entire set is garbage. Both the time and space complexity of the algorithm are found to be $O(N + E)$, where N is the number of actors and E the number of edges in the actor-reference graph.

[1] An Euler cycle in a connected directed graph is a cycle that traverses each edge of the graph exactly once.

```
for every actor A in the root set do
    set status of A to black
    for every forward acquaintance facq of actor A do
        scavengeForward(facq,A)
    endfor
    for every inverse acquaintance iacq of actor A do
        scavengeInverse(iacq,A)
    endfor
endfor

procedure scavengeForward(actor, sender)
begin
    if actor is not black then
        mark actor black
        for every forward acquaintance facq of actor A do
            scavengeForward(facq,A)
        endfor
        for every inverse acquaintance iacq of actor A do
            scavengeInverse(iacq,A)
        endfor
    endif
end

procedure scavengeInverse(actor, sender)
begin
    if actor is unblocked then
        if actor is not black then
            mark actor black
            for every forward acquaintance facq of actor A do
                scavengeForward(facq,A)
            endfor
            for every inverse acquaintance iacq of actor A do
                scavengeInverse(iacq,A)
            endfor
        endif
    else
        if actor is not gray then
            mark actor gray
            for every inverse acquaintance iacq of actor A do
                scavengeInverse(iacq,A)
            endfor
        endif
    endif
end

NOTES: Initially all actors are white.
After marking, all actors which are white or gray are reclaimed.
```

Algorithm 3.2: Garbage collection algorithm by Venkatasubramanian

Chapter 4

Transformation of the actor-reference graph

4.1 Introduction

This chapter describes a transformation of the actor-reference graph to a graph of passive objects. The transformed graph has the property that if a garbage collection algorithm which works for passive objects is applied to it, exactly those objects are recognized as garbage which correspond to garbage actors in the original actor-reference graph.

A substantial amount of research has been done in the area of garbage collection in functional, procedural and object-oriented languages in both uniprocessor and distributed systems. However, it has been commonly believed that, because of the "activity" associated with actors (recall that each actor has an encapsulated thread of control), the algorithms for garbage collection in these systems are not applicable directly for a system of actors. Therefore specialized algorithms have been formulated for actors. We describe a transformation of the actor-reference graph which captures all the necessary information peculiar to actors and makes it possible to use any garbage collection algorithm for passive objects to garbage collect actors. The numerous garbage collection algorithms already available for passive objects can then be adapted to work for actors.

The organization of the chapter is as follows. Section 4.2 explains the transformation of the actor-reference graph and illustrates it with a few examples. The proof of validity is given

in section 4.3. Finally, the cost of transformation and optimization issues are discussed in section 4.4.

4.2 Transformation of the actor-reference graph

In this section we describe the transformation of the the actor-reference graph into a "passive" object-reference graph while maintaining certain invariants.

Intuitively, because an actor might be unblocked and might change the reference graph spontaneously we see added complexity in garbage collection of actors. Recall that an actor is unblocked if its mail queue is non-empty or if it is processing a message. To capture the state of being blocked or unblocked, we conceptualize an object associated with each actor called the actor's *Mailqueue* We transform the actor-reference graph into a passive object graph using the following rules:

1. For every actor, a, in the actor-reference graph there is a corresponding object $CO(a)$ and a Mailqueue object $MQ(a)$ in the object-reference graph.

2. If an actor a is blocked, the $CO(a)$ is unconnected to $MQ(a)$, whereas if a is unblocked, $MQ(a)$ has a reference to $CO(a)$.

3. If and only if an actor a has a reference to an actor b, then $CO(a)$ has references to both $CO(b)$ and $MQ(b)$; and the $MQ(b)$ has a reference to the $MQ(a)$. Figure 4.1 illustrates this invariant.

4. If r is a root actor then $MQ(r)$ has a reference to $CO(r)$.

5. The root set in the object-reference graph consists of the Mailqueue objects corresponding to the root actors.

6. Whenever an object $CO(a)$ is recognized as garbage, $MQ(a)$ is also removed from the graph.

Once we have formed the Mailqueues for every actor and established the above mentioned invariants, we get a graph in which the notion of an actor being blocked or unblocked is automatically captured in whether or not an actor's CO has a reference from its MQ object. We

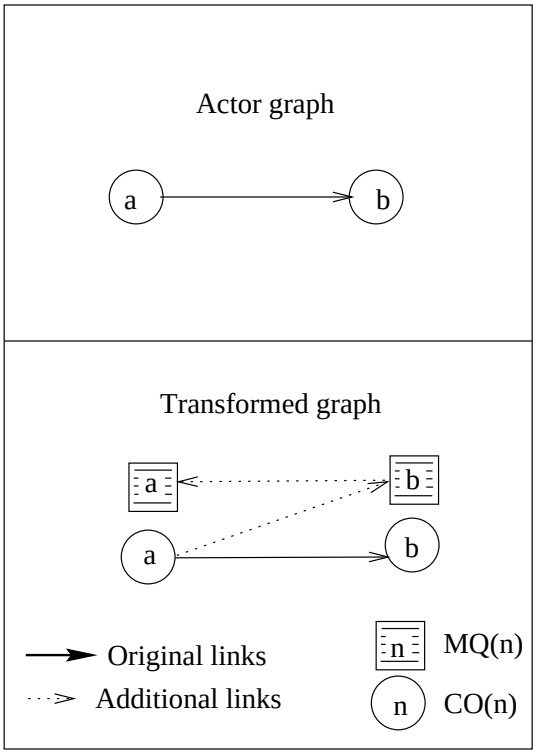

Figure 4.1: Transformation of an acquaintance in the actor-reference graph

will describe how to determine in an asynchronous distributed system whether or not an actor is blocked in Section 5.5. The entire graph (the MQs and the COs) can now be regarded as a graph of passive objects for the purpose of garbage collection. If a garbage collector for passive objects is executed on this transformed graph it will be able to recognize exactly those objects as garbage which correspond to garbage actors in the original actor-reference graph. The proof of this correspondence is given in the next section. Before describing the proof in detail, we illustrate the transformation in Figure 4.2. The top part of the figure shows the original actor-reference graph while the bottom one shows the transformed graph. In the transformed graph all objects are considered passive. COs for 6, 10 and 12 which correspond to unblocked actors are shown to have a reference from their Mailqueues. Looking at this graph we can see that a garbage collector for passive objects would regard COs for 1, 2, 3, 4, 5, 6, 8 and their MQs as live and all others as garbage. A look at the original actor-reference graph shows that it is exactly actors 1, 2, 3, 4, 5, 6 and 8 that are live. Of special interest is $CO(6)$ in the transformed graph. Because $CO(6)$ has a reference from $MQ(6)$ which is reachable from $MQ(1)$ (the root), it is correctly identified as being live.

4.3 Correctness of the transformation

We first give a definition and prove some lemmas.

Definition 4.1. *The set of all actors in the reflexive and transitive closure of the acquaintance relationship (actor* i *has an acquaintance relationship with actor* j *if* i *has* j *as an acquaintance) with an unblocked actor* a *is called the reachability set* R_a *of* a. *In particular,* a$\in R_a$. *We call* a *the* head *of* R_a. *The reachability set is defined only for unblocked actors.*

Lemma 4.1. *Consider the set* R_a *with head* a. *All the actors in* R_a *have the same status, i.e. either all elements of* R_a *are garbage or all are live.*

Proof. If any actor $b \in R_a$ is live, that implies that a is live since there exists a path from the unblocked actor a to a live actor b (a could send a message to b with a self reference thereby making itself reachable from b). But then all other actors in R_a must also be live since they are reachable from the live actor a.

It is possible for all actors in R_a to be garbage. This can happen if the subgraph formed by elements of R_a is disconnected from any other actor in the system. □

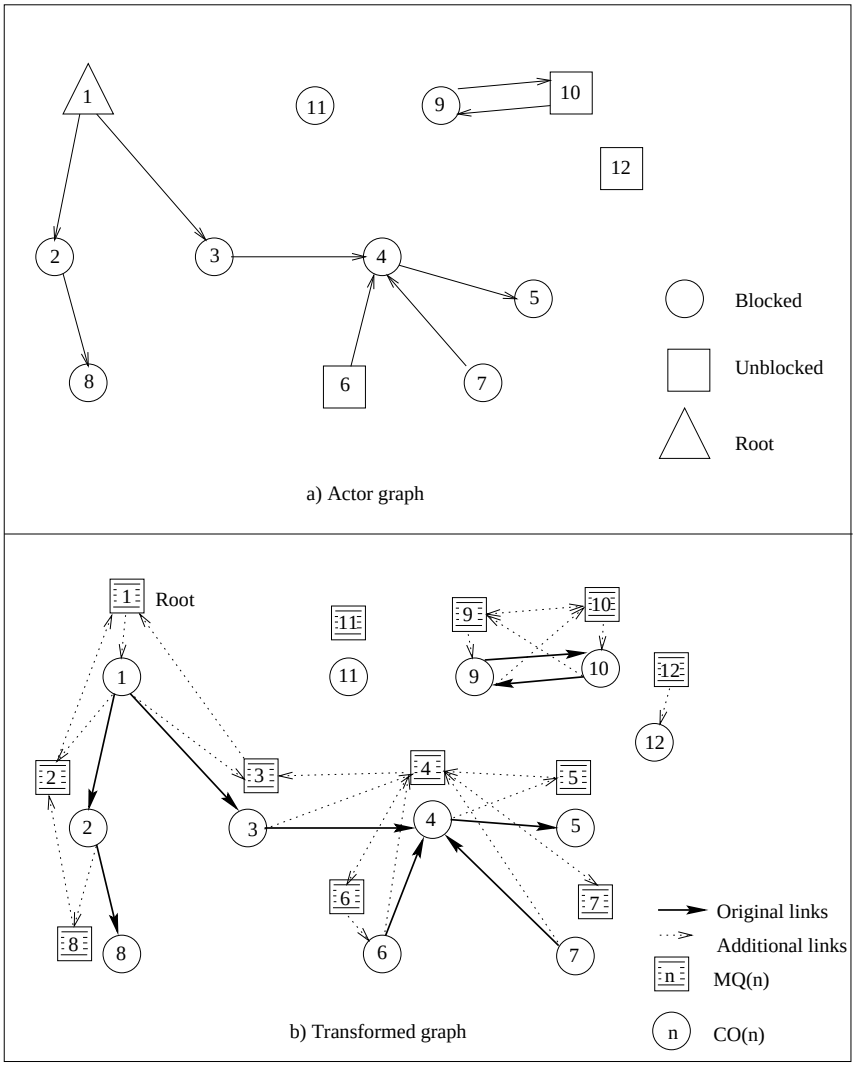

Figure 4.2: Transformation of the actor-reference graph

Lemma 4.2. *If two reachability sets R_a and R_b with heads* a *and* b *respectively, have an overlap, then the $MQ(a)$ and $MQ(b)$ must be reachable from each other in the object-reference graph.*

Proof. Let c be an actor common to both reachability sets.

If c is the same as either a or b, the lemma can be seen to hold trivially. If this is not the case, consider Figure 4.3. Since $CO(c)$ is reachable from $CO(a)$ it must be reachable from $MQ(a)$ also (since a is unblocked). If $CO(d)$ is the object through which $CO(c)$ is reachable from $MQ(a)$, then by construction $CO(d)$ must have a reference to $MQ(c)$ also. Hence $MQ(c)$ must be reachable from $MQ(a)$. Similarly, $CO(c)$ is reachable from $CO(b)$. Therefore, there must exist a path of references from $CO(b)$ to $CO(c)$. This implies that there must be a path from $MQ(c)$ to $MQ(b)$ (since if an actor x has a reference to another actor y , the $MQ(y)$ has a reference to the $MQ(x)$). Thus, $MQ(b)$ is reachable from $MQ(a)$. By symmetry, $MQ(a)$ is reachable from $MQ(b)$. □

4.3.1 Proof of correctness

Without loss of generality we can assume that there is a single root actor r in the actor-reference graph. If there is more than one root we can assume the existence of a hypothetical root actor which has references to the actual roots.

We also assume that the root actor r is always unblocked and hence $MQ(r)$ always has a reference to $CO(r)$. This assumption does not change the garbage status of any actor, as shown by the following lemma.

Lemma 4.3. *Let $G1$ be an actor-reference graph in which the root is blocked and $G2$ be a graph which differs from $G1$ only in that the root is unblocked. The garbage status of any actor in $G1$ is the same as its corresponding actor in $G2$.*

Proof. An actor is live or reachable from the root on the basis of a number of applications, say k, of the rules given in Section 2.4.1. First, by induction on k, we can show that any actor which is live in $G1$ is live in $G2$.

- Base case:

  For $k = 1$, we can have an actor as live only if it is the root actor. Clearly in both graphs the root actor is live.

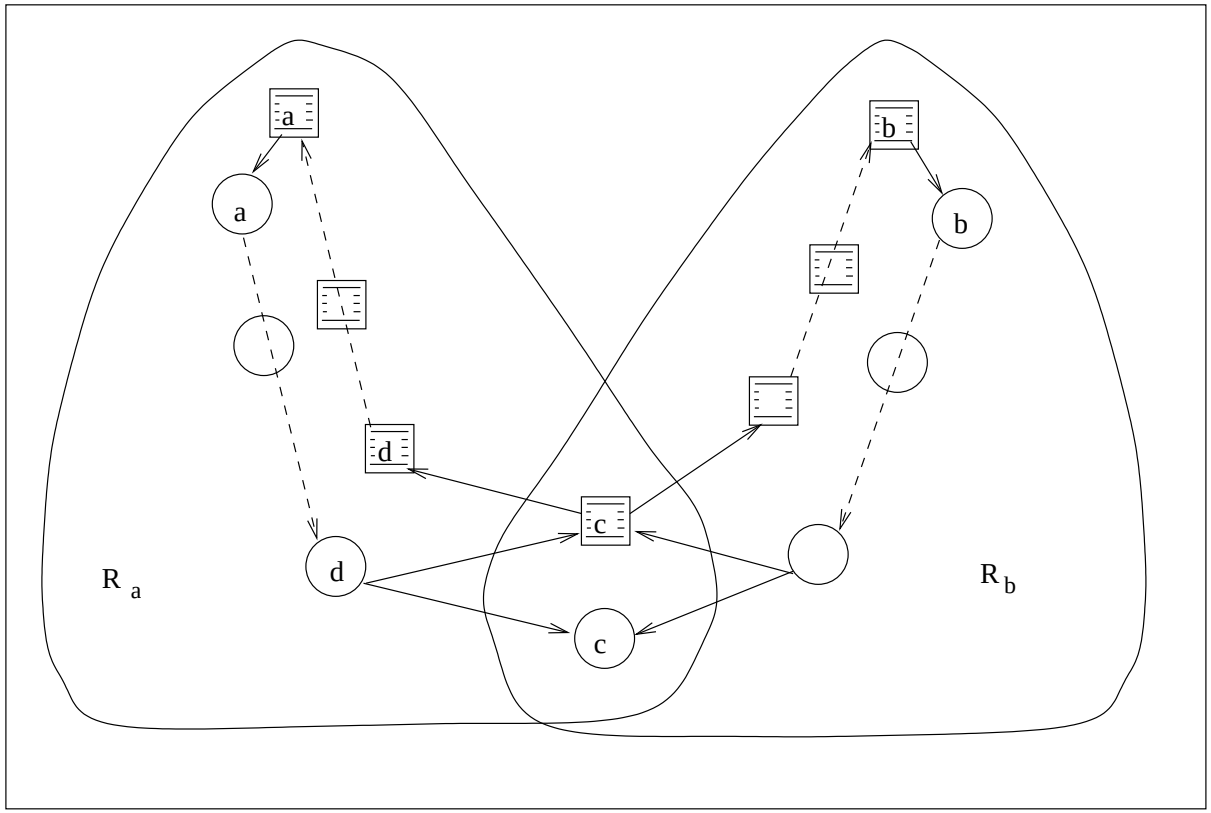

Figure 4.3: Two reachability sets with an overlap

- Inductive step:

  Assume that if an actor a is live in $G1$ in n applications of the rules, it is live in $G2$. If an actor b becomes live by another application of the rules 2 or 3 through a, it will be live in $G2$ by exactly the same rule through a. Rules 2 and 3 do not test the property of a being blocked or unblocked but do test b for that property. However, b will have the same status (blocked or unblocked) in $G2$ since b cannot be the root ($n > 1$) Thus an actor recognized as live by applied the $(n + 1)$ applications of the rules will be live in $G2$ also.

By a similar argument we can prove that, if an actor is live in $G2$ it is live in $G1$. Therefore, garbage status of all actors in $G1$ remains unchanged in $G2$. □

We can form the reachability sets $R_{a1}, R_{a2}, R_{a3} \ldots$ of all the unblocked actors $a1, a2, a3 \ldots$ and the reachability set R_r from the root actor r (by the above lemma, r can be treated as being unblocked). Any such set might be disjoint to all other sets or might overlap with one or more sets. Also, by Lemma 4.1, all the elements of any set have the same garbage status, so it is meaningful to say that an entire set is live or garbage. Figure 4.4 illustrates this property.

It is obvious that all the elements of R_r are live. If any set has an overlap with a live set it is live (because it has at least one live actor which it must have in common to the live set in order to have an overlap).

Let us define the O relation between two reachability sets R_1 and R_2 to mean that there *is* at least one actor common to both R_1 and R_2. Let Θ represent the set of all the reachability sets in the transitive and reflexive closure of the O relation with R_r. Let $\mathcal{R}$ stand for the union of all the reachability sets in Θ. Then we have the following lemma:

Lemma 4.4. *An actor* a $\in \mathcal{R} \Leftrightarrow$ a *is live.*

Proof. For the forward direction of the double implication, consider an arbitrary set $R_i \in \Theta$. If R_i is R_r itself, then obviously all actors in R_i are live. If R_i is O-related to R_{i-1} which in turn is O-related to R_{i-2} and so on till R_1 which is O-related to R_r, then by Lemma 4.1, R_1 must be live (since at least one actor in it which is common to R_r is live) and for the same reason $R_2, R_3 \ldots R_i$ all must be live.

For the reverse direction of the double implication, we give an inductive proof. An actor which is live must be proved to be live by one or more applications of the rules in Section 2.4.1.

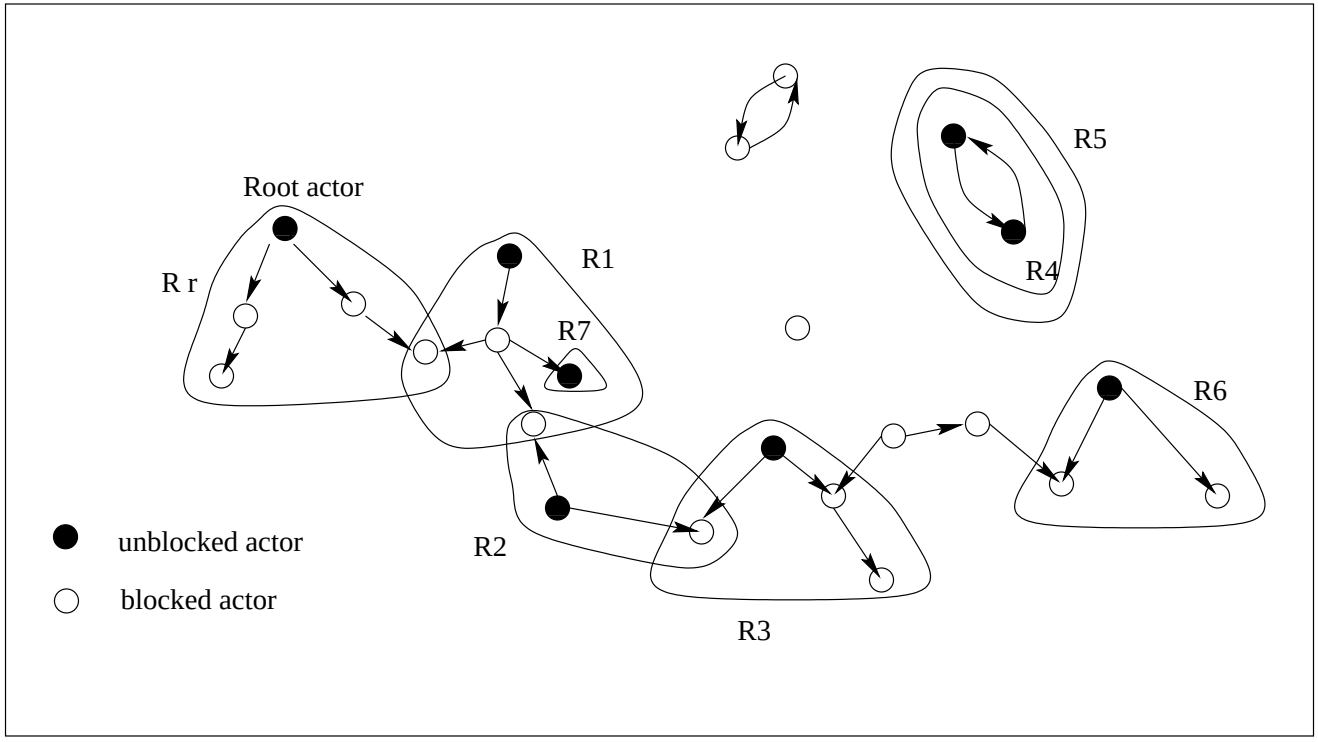

Figure 4.4: Reachability sets of actors

Our inductive hypothesis states that for k applications of this rule, where k is any natural number, all actors proved to be live belong to $\mathcal{R}$. Therefore, a is live $\Rightarrow a \in \mathcal{R}$

- Base case:

  The first application of the rules is for the root actor itself which trivially belongs to $\mathcal{R}$.

- Inductive step:

  Assuming the inductive hypothesis for all $k \leq n$, let us consider an actor a which is proved to be live in the $(n+1)th$ applications of the rules from an actor b. If the last application is Rule 2 (Every forward acquaintance [a in this case] of a live actor [b] is live), then a would belong to the same reachability set as b which is already in $\mathcal{R}$. If the last application is Rule 3 (Every inverse acquaintance [a] of a live actor [b] which is not permanently blocked is live), then a must belong to some reachability set (since it is not permanently blocked). This reachability set shares b with b's reachability set and hence it is also in Θ. Hence $b \in \mathcal{R}$.

□

We prove that the transformation preserves the following properties:

Theorem 4.1.

1. *If an actor* a *is live in the actor-reference graph, then* $CO(a)$ *will be considered live by a passive object garbage collector in the transformed graph.*

2. *If an actor* a *is garbage in the actor-reference graph, then* $CO(a)$ *will be recognized as garbage by a passive object garbage collector in the transformed graph.*

Proof.

1. Using Lemma 4.4, if an actor is live in the actor-reference graph, it must be a member of some reachability set in the transitive and reflexive closure of the O relation with R_r.

   (a) If the actor a belongs to R_r, then $CO(a)$ in the transformed graph is clearly live because it would be reachable from $CO(r)$ which in turn is reachable from $MQ(r)$ which is a root in the transformed graph.

(b) A live actor a may also belong to $R_i \neq R_r$ with R_i being in Θ. Let h_i be the head of R_i. In the transformed graph, $MQ(h_i)$ must be reachable from the $MQ(r)$. To see this, consider the sequence of reachability sets $R_r, R_1, R_2, \ldots, R_i$ where each consecutive reachability set in the sequence has an overlap. By Lemma 4.2, the Mailqueue objects corresponding to the heads of overlapping reachability sets are connected to each other. $MQ(h_i)$ is reachable from $MQ(h_{i-1})$ which in turn is reachable from $MQ(h_{i-2})$ and so on with the result that $MQ(h_i)$ is actually reachable from $MQ(r)$. Therefore, $MQ(h_i)$ will be considered live and hence the $CO(h_i)$ will be live ($(MQ(h_i)$ has a reference to $CO(h_i)$ as h_i is unblocked). This implies that $CO(b)$ will be considered live as it is reachable from $CO(h_i)$ (by virtue of b being in the reachability set of h_i) which is considered live.

2. According to Lemma 4.4, an actor is garbage if it falls in one of the following cases:

   (a) a blocked actor a which is not a member of any reachability set. $CO(a)$ in the transformed graph will be immediately recognized as garbage as it will be unconnected from $MQ(a)$ and is unreachable from any live object. Even if such an actor has a reference to a live actor it will not be live in the transformed graph as shown for actor a_1 in Figure 4.5.

   (b) an actor which is the member of a reachability set R_i which is not in the transitive closure of the O relation with R_r. R_i has to be either totally isolated from all other sets or can be connected to other sets through a sequence of two or more blocked actors as shown in Figure 4.4 for reachability set R_6. If R_i is totally isolated, then all the objects corresponding to actors in R_i will be clearly garbage as the subgraph formed by them is unconnected to the root object. For the other case, consider the situation in Figure 4.6 where R_i is "bridged" to R_{i-1} with a sequence of actors $a_0, a_1, a_2, \ldots, a_n, a_{n+1}$. a_0 belongs to R_{i-1}, a_{n+1} belongs to R_i and all the other actors in the sequence are blocked. n must at least be one, otherwise R_i and R_{i-1} would overlap. The direction of references between a_1 and a_0; and a_n and a_{n+1} must be as shown in the figure, otherwise a_1 and a_n would have to be included in the corresponding reachability sets. It is clear from the figure that regardless of the direction of references in the actors $a_2, a_3, \ldots, a_{n-1}$; a_0 and a_{n+1} are not reachable

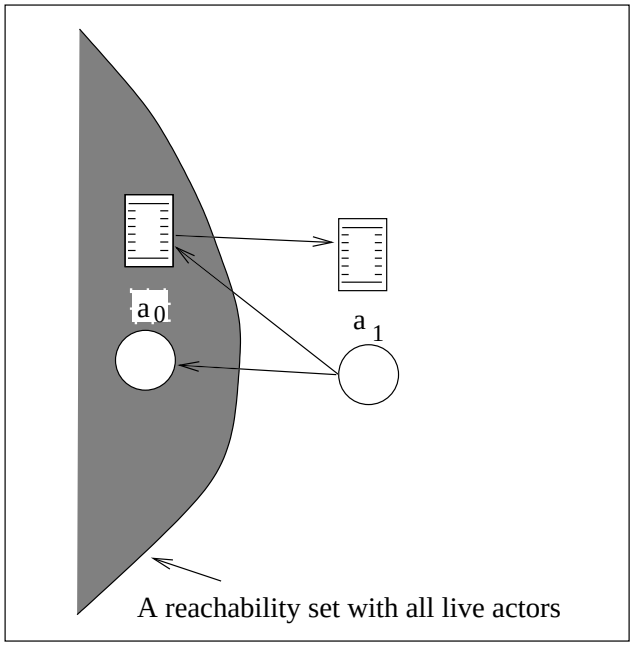

Figure 4.5: A blocked actor having a reference to a live actor

from each other. Thus the existence of a "bridge" of blocked actors does not alter the connectivity of R_i from the root. So all the objects in R_i will be recognized as garbage which is correct behavior.

□

4.4 Cost of transformation and optimizations

The extra information added in the transformed graph consists of the Mailqueue objects, the references between Mailqueue objects and the reference between the object and its Mailqueue for an unblocked actor. At first sight this might appear to be excessive overhead. In practice, we do not need to maintain a separate Mailqueue object for every actor. The link between the object and its Mailqueue can be represented by a single bit in the state of the actor. In a mark-sweep like scheme, separate mark bits can be kept for both the object and its Mailqueue in the state of the actor itself. A mark message from Mailqueue of one object to the Mailqueue of another actor can be emulated by sending a message to the other actor asking it to turn the mark bit of its Mailqueue object to the desired color. We have to maintain inverse acquaintances

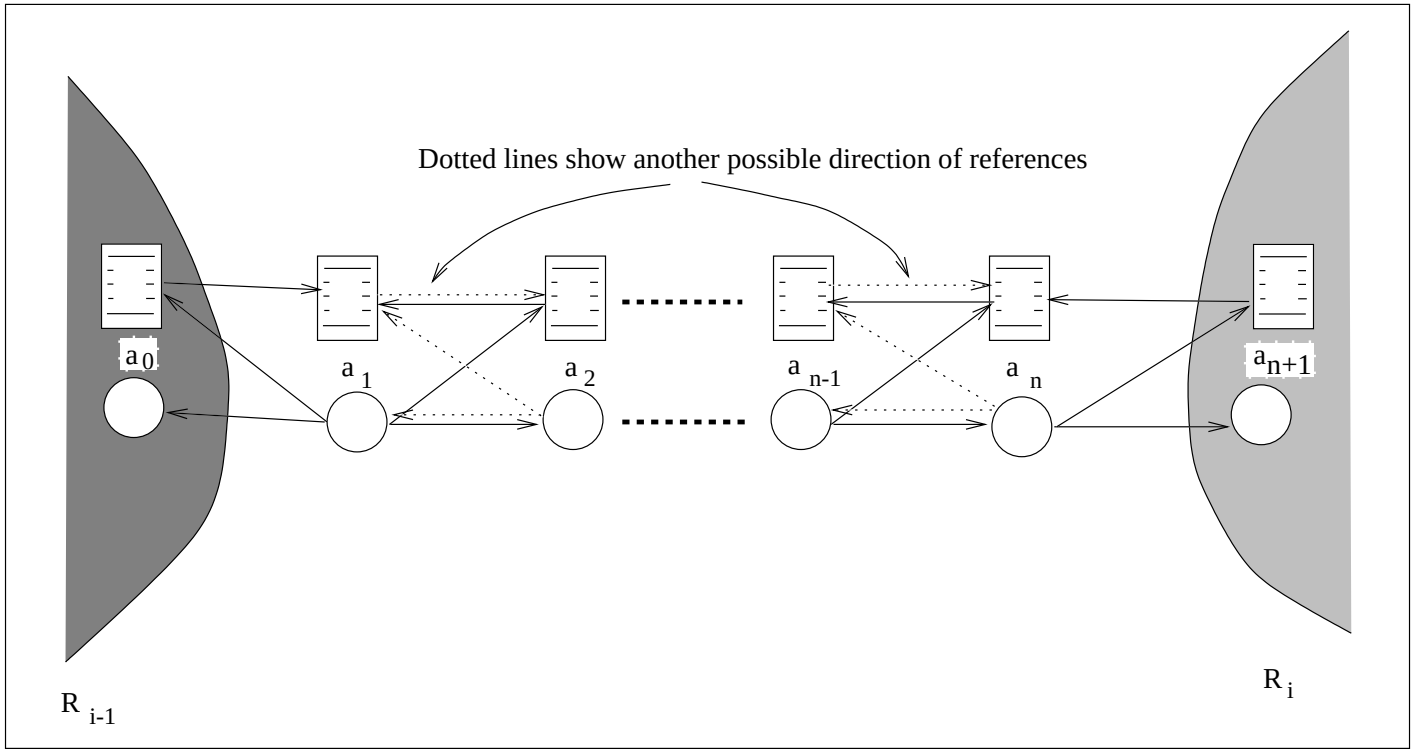

Figure 4.6: Two reachability sets bridged by a sequence of blocked actors

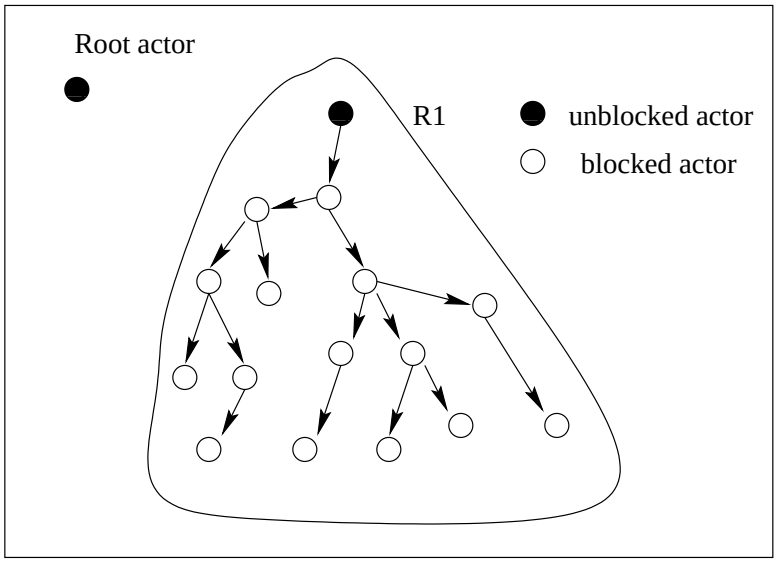

Figure 4.7: A reference graph in which maintaining inverse acquaintances is advantageous.

in order for the Mailqueue of an actor to be able to refer to the Mailqueues of the actor's inverse acquaintances.

The main additional cost in transforming the graph is building of inverse acquaintances for all actors. The inverse acquaintances can be established at the time of garbage collection or can be maintained incrementally depending on the choice of the garbage collection algorithm. Maintaining inverse acquaintances presents a tradeoff. There are some reference graphs in which maintaining inverse acquaintances would be an unnecessary overhead. However, in certain reference graphs it can be advantageous. Consider Figure 4.7, which shows a reference graph with an unblocked actor having a large reachability set and a single root actor which does not have any forward or inverse acquaintances. Any algorithm which does not keep inverse acquaintances would have to trace the entire reachability set of the unblocked actor in order to make sure there is no actor through which a message can be passed to the root actor. On the other hand, if inverse acquaintances are maintained it can be easily seen without tracing the entire reachability set that the set is garbage.

4.5 Correspondence with a known garbage collection algorithm

We can formulate an algorithm for garbage collection of actors based on a simple mark-sweep on the transformed object-reference graph. The roots of the transformed graph are the Mailqueues corresponding to the root actors in the actor-reference graph. This gives us a simple algorithm for garbage collection of actors based on mark-sweep. A close look at this algorithm reveals that it is very similar to a known algorithm for garbage collection of actors as given by Venkatasubramanian [36]. This algorithm is described in Figure 3.2 on page 19.

We can see that there is a correspondence between Venkatasubramanian's algorithm and our mark-sweep on the object-reference graph. When Venkatasubramanian's algorithm proceeds to mark an actor a black, our algorithm proceeds to mark both $CO(a)$ and $MQ(a)$. When the actor is marked gray in Venkatasubramanian's algorithm, $MQ(a)$ but not $CO(a)$ is marked in our algorithm. In Venkatasubramanian's algorithm, if *scavengeInverse* is invoked on an unblocked actor, the actor is marked black. In our algorithm, the $MQ(a)$ is marked and since it has a reference to its object (as the actor is unblocked), $CO(a)$ is also marked. Thus marking an actor a black corresponds to marking both $MQ(a)$ and $CO(a)$, marking gray corresponds to just marking $MQ(a)$ and a state of white corresponds to neither $MQ(a)$ nor $CO(a)$ being marked.

Chapter 5

Distributed Garbage Collection of Actors

In Chapter 4, we saw how an actor-reference graph can be transformed into a passive object graph which can then be used by a garbage collector that works for passive objects to collect garbage actors. This chapter describes garbage collection of actors in a distributed environment using this transformation.

5.1 Distributed garbage collection

Garbage collection essentially finds objects or actors that are not connected to the root set in the reference graph. In a distributed system, the reference graph may be spread across different hosts and may be changing dynamically. To find garbage, ideally one would like to construct a global snapshot of the reference graph and identify objects or actors which are garbage based on that graph. Since there is no omniscient observer in a distributed system, a global snapshot is usually constructed by taking a consistent cut of the global state and accounting for any messages in transit in the network. The reader is referred to [5, 34] for details. However, such a "global-snapshot" approach requires synchronization among all the hosts in the system which raises questions about scalability. For this reason, most algorithms try to avoid taking a global-snapshot.

There are several other algorithms available in the literature as described in Chapter 3. We have chosen to adapt Schelvis' [33] work for garbage collection of actors. The reasons for this choice are:

- Schelvis' algorithm does not require synchronization among hosts and hence is more scalable.
- It allows autonomous local garbage collection.
- It allows garbage collection to proceed concurrently with the mutator.
- It can collect cycles of garbage objects that span different hosts.

It appears that Schelvis' algorithm has some unique features and advantages but it has not been used much in the garbage collection community. Recently, Louboutin [23] has proposed a reactive algorithm with similar features, but a few implementation issues are not resolved, as mentioned in Section 3.2.4.

The problem of garbage collection is split into: a) local garbage collection at each host and b) global garbage detection service that identifies garbage which cannot be recognized on the basis of local information alone. Both the local garbage collection and the global garbage detection operate on the passive object graph obtained from the actor-reference graph by the transformation described earlier. The objects identified as garbage in the transformed graph directly correspond to garbage actors in the original graph. As long as we maintain the invariants required by the transformation, we shall be able to identify garbage actors correctly using the transformation technique. Therefore, in the rest of the thesis, we focus our attention on garbage collection of *objects* in the transformed graph. Local garbage collection on each host is allowed to proceed independently of other hosts and maintains enough information to support global garbage detection. Hosts cooperate only very loosely for global garbage detection and the garbage collector executes concurrently with the mutator. In order for the algorithm to be correct, we must ensure that it collects the same objects or actors as garbage as the "global-snapshot" approach described in the previous paragraph would. In particular we must address these issues:

- maintaining consistency of the global state on which the garbage collector acts upon

- taking into account any messages in transit
- maintaining invariants required by the transformation from the actor-reference graph to the passive object graph
- handling concurrency between the mutator and the garbage collector

We first describe local garbage collection and then explain how these issues are handled.

5.2 Local garbage collection

Each actor in the host maintains at all times a list of forward and inverse acquaintances that it holds. It also maintains information needed to store the states of the corresponding object and the Mailqueue in the transformed graph (as described in Chapter 4). Following the terminology of Chapter 4, we use the shorthand $CO(a)$ and $MQ(a)$ respectively to stand for the corresponding object of an actor a and the Mailqueue object of a in the transformed graph. The parameter a is dropped when it is clear from the context.

Objects referenced from outside the host are called *global roots* following the terminology of Louboutin[23]. The set of global roots consists of the COs and the MQs of any actors referenced from outside and the MQs of actors which reference any actor outside (because the MQ of the remote actor would reference the MQ of the local actor). Global roots are conservatively included in the root set for local garbage collection because it is not possible to locally determine whether the remote object that references a global root is reachable from an actual root at some other site or not. It is the task of the global garbage detection service to identify the global roots that are not reachable from any actual root and remove them from the root set of local garbage collection. Local garbage collection continues to work from a conservative root set which is progressively updated by global garbage detection so that all garbage is ultimately collected. A mark-sweep algorithm is used for local garbage collection.

5.3 Global root graph

The concept of a global root graph has been borrowed from Louboutin [23]. The vertices of this graph consist of the global roots of all hosts and local roots of some hosts. The local roots that

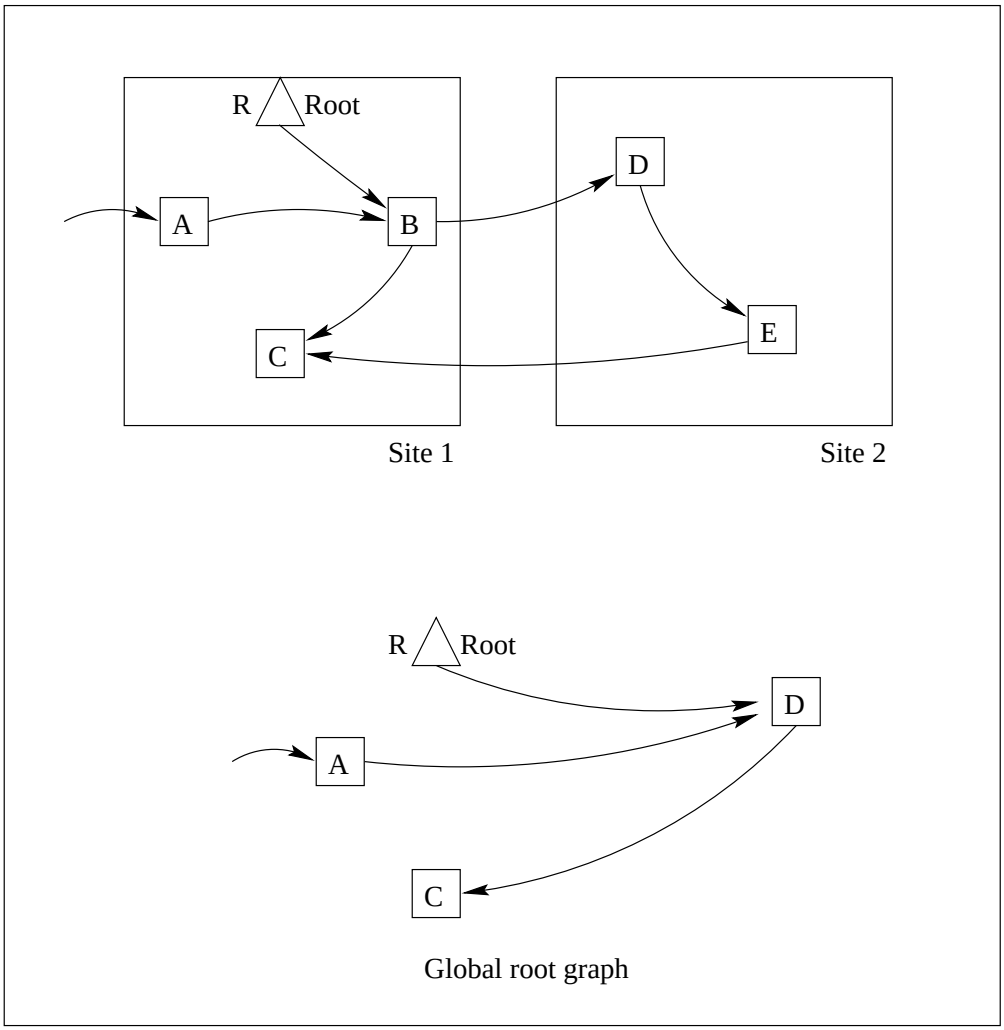

Figure 5.1: An object graph and its global root graph

are included in the set of vertices are only those from which a path exists to a remote object. Each outgoing path from a vertex, which crosses the site boundary (possibly through several objects located at the same site as the vertex) and refers to a remote global root becomes an edge between the vertex and the remote global root. Figure 5.1 shows an object graph in the upper portion and the corresponding global root graph in the lower portion (We have shown a simple object graph for the sake of illustration, but the concept of the global root graph is equally applicable to an object graph obtained from the transformation of an actor-reference graph).

The edges of the global root graph are constructed at the time of local garbage collection. While marking is being performed for local garbage collection from a particular root a (a local root or a global root), if the marking leads to a reference to b that goes out of the site, an edge is included in the global root graph from a to b. Different mark bits are used for different roots so that all such paths can be discovered. This involves some extra work as objects reachable from different roots will have to be marked separately for each root.

The global garbage detection service operates on the global root graph. The root set for this graph consists of those vertices that are actually local roots. If a global root becomes unreachable from all of the roots of this graph, the global root is not reachable from any local root residing in a remote host. This is so, because if the global root is reachable from any such local root, there would be a path from that local root to the global root (by construction of the global root graph). Note that the global root might still be reachable from the local root located on the same site as the global root as intra-site paths are not considered in the global root graph. The global garbage detection service only verifies whether global roots are reachable from remote local roots or not. A global root discarded by the global garbage detection service is no longer considered to be in the root set of local garbage collection. The local garbage collection is free to reclaim it if it is not referenced by any of its own roots.

5.4 Maintaining a consistent state for garbage collection

Due to the asynchrony inherent in distributed systems, we have to make sure that the algorithm that we apply operates on a consistent state of the system. With respect to garbage collection, it becomes especially important to account for any state information present in messages in transit through the network. This problem is not present in "traditional" distributed object-oriented systems because they usually use a model of synchronous communication. The entire period of time that a message is in the network, the client object is blocked and hence its state remains unchanged. Therefore the state captured in the message is still represented in the client object. In contrast, communication is asynchronous in actor systems. Figure 5.2 depicts a situation where a message in transit carries state information which is not present in either host. The initial reference graph is shown in (I). Object A passes a reference that it has for C to the remote object B and immediately deletes its reference to C as shown in (II). While the

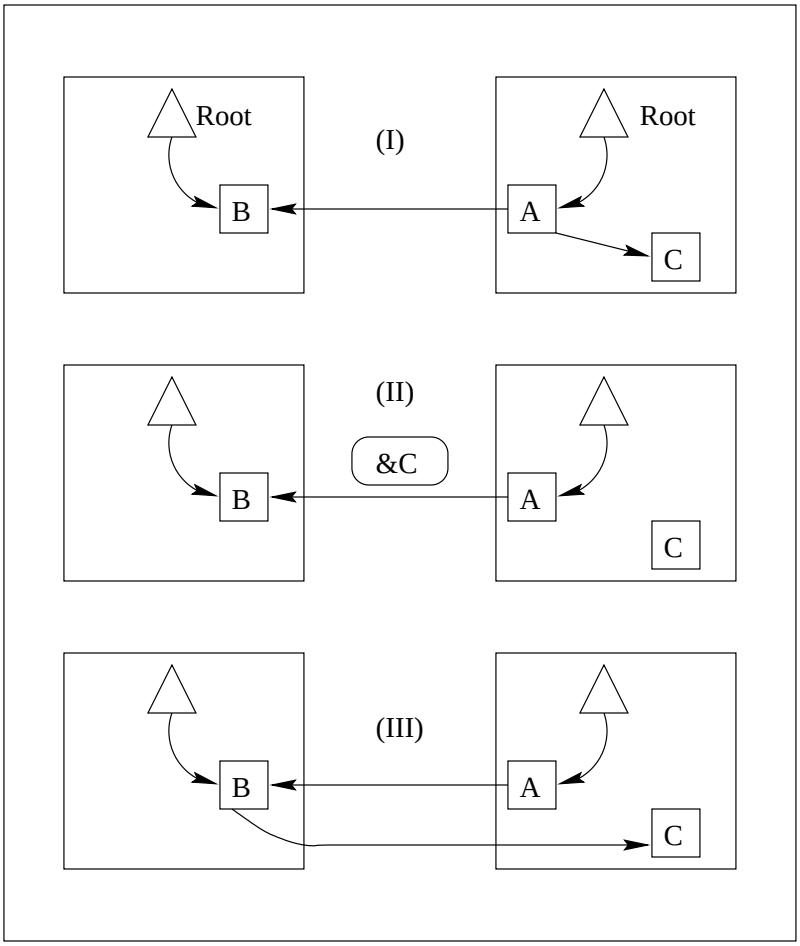

Figure 5.2: Reference carried by a message in transit

message is still in transit from A to B, local garbage collection is performed on both sites. In this case it can lead to C being incorrectly treated as garbage.

The problem is solved by suspending an object which sends a message across the network until the reception of that message is acknowledged. Note that this is not in contradiction of the semantics of asynchronous communication, as the sender waits only till the acknowledgment of the message and *not* until the remote object executes the method requested and returns a reply. The period of waiting is restricted to the round trip network delay and acknowledgment generation and processing time. The state information of the sender (at least with respect to garbage collection) remains unchanged for the period of waiting. In our earlier example, A

would not be able to delete the reference to C until B has already gained the reference to C, thereby avoiding the incorrect reclamation of C.

The global state of the distributed system is partitioned into sets of local states. Since the sender is suspended for the period a message is in the network, there is no extra state information in the network to be accounted for. This also prevents sites from receiving messages in incorrect causal order. Thus, at any time the global state, as constructed from the union of the local states, is consistent. Garbage collection would therefore operate correctly. Although suspending sending actors is undesirable, any algorithm for garbage collection in a system with asynchronous communication, which tries to avoid taking a global snapshot of the system has to use some mechanism to ensure that information stored in messages in transit is captured. An alternative to suspending the sending actor is to keep a record of the information transmitted with the message and disallow any action which would make destructive changes to this information, till an acknowledgment of the message is received.

5.5 Invariants required by the transformation

In section 4.2, we mentioned the various invariants needed by the transformation. As the actor-reference graph evolves, so does the transformed graph and we have to ensure that, at all times, the invariants are preserved.

One of the invariants is that if actor a has a reference to actor b then $CO(a)$ has references to $CO(b)$ and $MQ(b)$; and $MQ(b)$ has a reference to $MQ(a)$. If both a and b reside on the same site, this is easily accomplished by appropriate locking mechanisms. If b is on a different site, a message is sent to b's site to indicate that $MQ(b)$ should now contain a reference to $MQ(a)$. a has to be suspended for this period to ensure that the information contained in the message is not lost. This is similar to the case of suspending an actor which sends a message across the network, as discussed in the previous paragraph. Similarly, when a loses the reference to b, a message is sent to b's host asking it to remove the reference of $MQ(a)$ from $MQ(b)$, with a being suspended all this time. This is needed to ensure the correct operation of the garbage collector[1]. Handling of the references from $CO(a)$ to $CO(b)$ and $MQ(b)$ is easier. Since $MQ(b)$

[1] An example would serve to illustrate the error: a sends a request r_1 to b asking it to remove the reference of $MQ(a)$ from $MQ(b)$. Meanwhile, through some other actor, a again gains a reference to b and sends a request r_2

and $CO(b)$ physically reside in the same actor, their references can be encoded in a single reference in $CO(a)$.

Another invariant is that when an actor a is unblocked, *i.e.* it has pending messages in its mail-queue or is currently processing a message, there should be a reference from $MQ(a)$ to $CO(a)$. This is easy to maintain as it can be represented by a single bit in the actor's state. An actor whose mail-queue is empty might still have a message destined for it in transit in the network. Such an actor is still considered blocked. This does not result in any problems as the remote actor which is sending the message is suspended until the acknowledgment of the receipt of the message. We treat the state of the receiver actor as being the same as what it would be if the sender actor had still not put the message on the network.

5.6 Concurrency between the mutator and the collector

As mentioned earlier, the mutator is allowed to execute concurrently with the garbage collector. The mutator may write into unmarked objects and destroy the reference relationship at the initiation of garbage collection. We take an approach similar to the standard method of *write barrier* with *snapshot-at-beginning*. The references that each object has at the time of initiation of garbage collection are saved for use in the marking phase. If the mutator tries to delete a reference which was present before the initiation of garbage collection, the reference is not deleted immediately but only after garbage collection is finished.

5.7 Global garbage detection service

The global garbage detection service operates on the global root graph and identifies global roots which are no longer reachable from the roots in the graph. This section gives the details of this algorithm.

The algorithm is based on the idea that information about the liveness of an object is distributed across several sites and it is possible that a merge of pieces of information about the global root graph may provide enough information to decide whether an object is live or garbage. Different sites periodically exchange information about the object graph and combine this

to b so that $MQ(b)$ can have a reference to $MQ(a)$. If r_1 is delivered later than r_2, it would erroneously destroy the reference of $MQ(a)$ from $MQ(b)$.

information with the local state to decide about reachability of objects. The unit of information which is exchanged is called a *timestamp* packet. These packets are sent asynchronously along the edges of the global root graph from one global root to another and the frequency at which they are sent can be varied depending on whether quick reclamation of global garbage is desired or not.

Each global root n maintains a set P_n of the most recent packets it has received from different global roots that have a reference to n. Based on P_n, n calculates the packet that it will send out to the global roots that it has references to. A global root recognizes that it has become unreachable in the global root graph when the set P_n achieves a certain property. When a global root loses an edge to another global root, it sends out a special packet, called the *empty* packet ϵ, to that global root. Also, the vertices in the global root graph which are actually local roots of some sites, always send out a *root-packet*.

Each global root has a unique timestamp which consists of a tuple (*localtime.hostid*) corresponding to the the time it was first made the global root. We use the timestamp of a global root to refer to it. Thus global root n is the global root with timestamp n. Each timestamp packet is a list of one or more timestamps. The last timestamp in a packet may carry an accent. For example, the packet fe' means that there is no path from a root via e to f, while a packet fe means that there may be a path from a root via e to f. The former is called an *answer* packet while the latter is called a *question* packet.

The notations used in the presentation of the algorithm are given below:

- timestamp ordering: $(12.3) > (6.10) > (6.9)' > (6.9)$
- $first(feb) = f$, $last(feb) = b$, $length(feb) = 3$
- lexical packet ordering:
  $fed > fecb > fec, fec' > fec$
- prefixing: $\epsilon, f, fe, fec \subseteq fec$
- $strip(fecb, c) = fec$
- adding:
  $fec + b = fecb$, $fec + c = fec$, $fe + e' = fe'$

- if $fec' \in P_n$ then fec, $fecb$, $fecba'$ are called obsolete.

Mutator:
When a remote reference from m to n is created, a packet m is added to P_n;
When a global root is created it is assigned a unique timestamp.

Algorithm 5.1: Mutator part of Schelvis' algorithm

Algorithms 5.1, 5.2 describe the method. The procedure *sendfrom(n)* refers to the procedure used to compute a timestamp packet at global root n. Similarly *receive(m,p,n)* refers to the computation done in response to receiving a timestamp packet p from global root m by global root n.

For more details regarding Schelvis' algorithm and for a proof of its correctness, the reader is referred to Schelvis [33].

All nodes n have the empty packet ϵ in P_n. The packets are computed in a manner such that the following invariants are preserved. For a node n with packets received P_n and $p_n = MTP(P_n, n)$:

- $p_n = t_1 t_2 \ldots t_k \Rightarrow t_1 > t_2 > \ldots t_k$
- $p_n = \max P_n$
- packets sent by a node are equal to or larger than the packets it sent in the past.

Since there is no time order between the activities of garbage collection at different hosts, no synchronization is necessary. The algorithm is able to collect cyclic garbage and can proceed concurrently with the mutator.

```
procedure sendfrom(n)
begin
    p := MTP(P_n,n)
    send p
    "if no roots, remove node"
    if answer(p) and length(p) = 1 then
        remove n
    endif
end

MTP(P_n,n)
begin
    p = strip(max(P_n),n)
    if ∀ non-obsolete q ∈ P_n: q = p + t' or q ⊆ p
    "where t is some timestamp"
    then
        if answer(p) then
            return (p)
        else
            return (p+n')
        endif
    else
        if answer(p) then
            n := localtime := max(localtime,last(p)+Δt)
            return(strip(p,n)+n)
        else
            return(p+n)
        endif
    endif
end

procedure receive(m,p,n)
begin
    "p_n ∈ P_m is the packet that was previously
    received from n; p_m is the last packet created by m"
    if p = ε then
        remove p_n from P_m
        if p_m > MTP(P_m, m) then
            m := localtime := max(localtime,first(p_n)+Δt)
        endif
    else
        replace p_n in P_m by p
    endif
end
```

Algorithm 5.2: Collector part of Schelvis' algorithm

Chapter 6

Implementation

This chapter describes the implementation of garbage collection of actors in a distributed environment.

6.1 Underlying architecture

We have implemented a garbage collector for actors in the Actor Foundry (release 0.1.8) [1], which is an Actor system written in Java™. The run-time environment of the Actor Foundry consists of one or more Java Virtual Machines (JVMs) running on possibly different hosts. Actors can be created on local or remote hosts and messages can be sent between actors. Each Actor encapsulates a thread of control and has a mail address and a mail queue. Actors process their messages in the order they appear in the mail queue. Migration of actors is supported in the Foundry but the current garbage collector does not support it.

The Actor Foundry is organized so that its different parts are required to implement certain interfaces. This enables components to be replaced with behaviorally equivalent components. Figure 6.1 shows the software architecture of the Actor Foundry. The main components of interest to us are presented below. The reader is referred to [1] for details about other features of the Foundry.

6.1.1 The Actor Manager

This part of the Foundry is responsible for actual management of actors. It keeps a list of actors managed by itself and allows creation of new actors, reception of messages for the managed

actors, migration of actors and invocation of different services. The Java interface *ActorManager* has to be implemented by any class which purports to be an Actor Manager. In the current release, there is a *BasicActorManager* which provides this implementation.

6.1.2 The Actor interface

A Java interface called *Actor* provides the user of the Actor Foundry the basic actor primitives:

- *create* (for creating new actors),
- *send* (for sending messages),
- *call* (similar to send except that the sending actor is blocked till a result is received from the actor to whom the message was sent) and
- *invoke service* (to invoke some service like name lookup).

The *ready* primitive normally associated with actors is not available. However, the same effect can be achieved by storing state in the Java program. Users are not allowed direct access to the internal data structures of the Foundry and must program actor operations via these primitives.

6.1.3 The Actor implementation

The actual implementation of the primitives given by the *Actor* interface must be provided by a Java class which implements the Java interface *ActorImpl*. The current release provides a class called *BasicActorImpl* for this purpose.

6.1.4 The Scheduler

The purpose of the scheduler is to provide fair scheduling to all threads including those belonging to actors. A Java interface called *Scheduler* specifies the methods a scheduler for the Foundry has to implement. The current release provides two different schedulers: a *BasicScheduler* which implements preemptive scheduling and a *NoScheduler* which lets the default scheduling mechanism of the Java Virtual Machine be used.

6.1.5 Actor Names

In the Foundry, the mail addresses of actors are represented by a Java class called *ActorName*.

6.1.6 The Name Service

The name server provides a binding from ActorNames to the actual physical location of the actors. The Name Service interface specifies the methods to be implemented by the name server.

6.1.7 The Transport Layer

This interface specifies the functions of a transport layer which can be used to transmit messages between Actor Managers at different hosts. Currently, a transport layer based upon TCP and a transport layer based upon UDP (but with an additional mechanism to ensure reliability) are available in the Foundry. For our implementation of the garbage collector, the transportation layer based on UDP is used. Communication is thus asynchronous with unbounded delay but guaranteed delivery.

6.1.8 The Request Handler

The Request Handler provides the interface to the Actor Manager for getting services from the Name Service and the Transport Layer.

6.2 Garbage collector implementation

For the implementation of the garbage collector, we have provided a Java class *GCActorManager* to implement the *ActorManager* interface, a Java class *GCActorImpl* to implement the *ActorImpl* interface and a Java class *GCActorName* in place of *ActorName*. These classes provide the extra functionality needed to support garbage collection. Existing code in the *BasicActorManager*, *BasicActorImpl* and the *ActorName* is reused by Java's inheritance mechanism. Each node also has a privileged actor called the *GCActor* to coordinate the activities relating to garbage collection.

By default, computation in the Actor Foundry is started by opening up a shell which interacts with the foundry using a *Shell Service*. However, for our implementation, the actor computation is started by activating the Actor Managers of all nodes and then "booting" the computation by starting up a "boot" actor at a particular node. This actor might create other

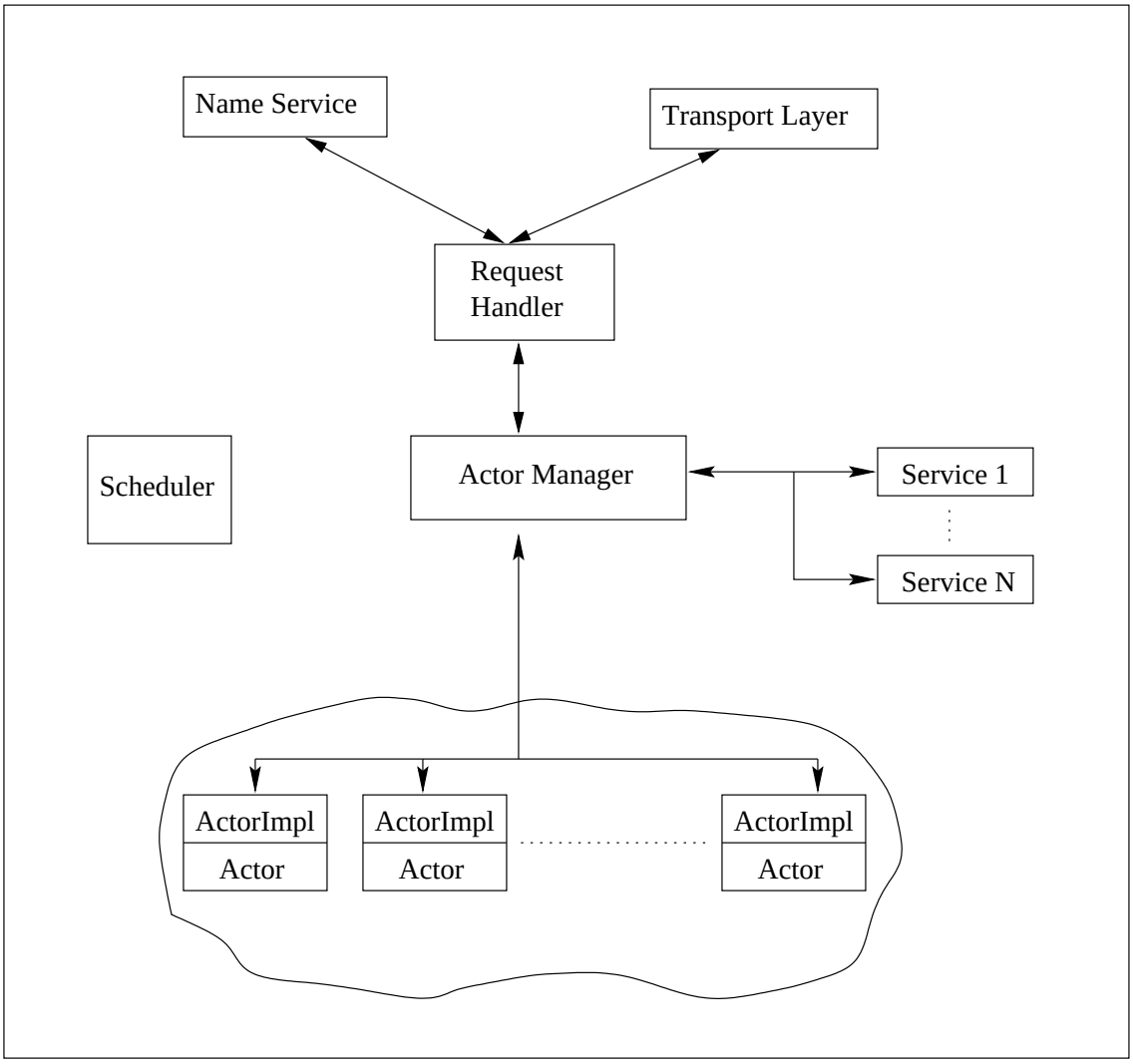

Figure 6.1: Architecture of the Actor Foundry

actors (which may be at the same or a different node) and continues the computation by sending messages to any created actors or to itself.

At every host, local garbage collection is initiated periodically by sending a message to the *GCActor*. On receipt of this message, the GCActor saves the list of actors managed at the local host. All actor activity is frozen while the list is being created. To allow concurrency between the mutator and the local garbage collector, all the states in the system are colored by a *GCcolor* with the GCcolor being changed at every initiation of local garbage collection. Thus, all events and actor states in the local host are partitioned into non-intersecting sets based upon whether they were initiated before or after the start of local garbage collection. In reality only two different colors are needed as runs of local garbage collection do not overlap. A run of the local garbage collection uses information exclusively as encoded in the set of events and states of the colors preceding that run. By this means, the local garbage collector mimics a "stop-the-world" approach in which the mutator is halted while garbage collection is in progress.

During local garbage collection, for each object O belonging to the set of local roots and global roots in the local host, a marking trace is performed starting from that object. If during such a trace, a reference is encountered leading to a remote object, a global-root graph edge is added between O and the remotely referenced object. Once tracing has been completed, any object left untouched by tracing is reclaimed. This completes the local part of garbage collection.

After execution of the local garbage collector, each host sends out *timestamp* packets in accordance with Schelvis' algorithm as described in Section 5.7. The receipt of such packets might lead to discarding some global roots causing a reduction in the root set for local garbage collection. A Java class *Packet* encapsulates the functionality of the timestamp packets and a Java class *Globalroot* contains the functions for adding and deleting edges in the global root graph and for sending and receiving timestamp packets according to Schelvis' algorithm.

6.2.1 Support from Java's garbage collector

The Java programming language [3] has support for garbage collection. We cannot use this support to automatically collect garbage actors because actors have an encapsulated thread of control which always makes them non-garbage according to the semantics of Java. However, garbage collection available in Java is used to detect when an actor loses a reference to another

actor. As we mentioned earlier, references to other actors are represented by Actor Names in the Foundry. Actor Names themselves are regular Java objects. An actor gets an Actor Name either by virtue of creating a child actor or through a message from another actor. When the Actor Name gained by an actor goes out of scope or is lost to the actor, Java's internal garbage collection tries to reclaim it. We use the *finalization* interface available in Java to detect this event and record the fact that the given actor has lost a reference to another actor. In our implementation, all Actor Names are instances of the Java class *GCActorName*. Each actor maintains for every acquaintance it has, a reference count of number of GCActorNames by virtue of which it has that acquaintance. As GCActorNames are lost and get reclaimed by Java's internal garbage collector, the reference count is decremented. When the reference count associated with an acquaintance drops to zero, the actor is recognized as no longer having that acquaintance.

6.2.2 Detection of reference passing among actors

We detect the passing of an Actor Name in a message from one actor to another by scanning the parameters contained in the message. It is assumed that Actor Names are not passed as components of an array or embedded inside another object.

6.2.3 Issues in the construction of the global root graph

Schelvis' algorithm requires that when an edge from a global root m is added to another global root n, a packet m is added in the set of packets that n has. However, it might happen that n is elevated to the status of a global root[1] but the global roots that would have an edge to this global root would be known only after the next local garbage collection at the remote host. This creates a situation where we have a global root but we do not know the edges incident upon it. We temporarily treat such a root n as a local root. Once local garbage collection at the remote host is executed and the edges of the graph leading to n are found, it is treated as dictated by the algorithm.

[1]The object might be created from a remote host or its reference passed to a remote object

No.	Total time without GC	Total time with GC	Ratio
1	1.40 seconds	2.235 seconds	1.596

Table 6.1: Timings for 5-queens problem on a single host

6.3 Results

To get a rough idea of the cost of GC (garbage collection) in the current implementation, an actor program was run on the Actor Foundry. The program implements an exhaustive search solution for the *N-queens problem*. The problem is to put N queens on an N by N chess-board such that no queen is under attack from another one according to the rules of chess. In the implementation a single actor, C, starts the computation with an empty chess board. It places a single queen in one of the squares on the first row and creates an actor to solve the remainder of the problem. One actor is created for every square on the first row. When a newly created actor receives a partially filled chess-board it places queens on the row following the rows that have already been filled and spawns additional actors to do the remainder of the computation. If an actor manages to fill all rows, it sends a message to C notifying it of the solution. The program generates a large amount of garbage.

6.3.1 Single host results

Table 6.1 presents the results for the solution of the N-queens problem for a board size of 5, running on the Actor foundry on a Sun Ultra-2 Workstation under normal load. The garbage collector was run once every second. Each timing is the median time of 10 executions. The ratio of time taken with GC running and without GC is seen to be 1.596 which shows a considerable overhead associated with GC. We feel that some of this overhead can be reduced by optimizing the implementation which has not be done due to scarcity of time.

To analyze the cost of garbage collection, timings were obtained with different parts of the garbage collector running. Table 6.2 shows the results.

The "Overhead" column in Table 6.2 gives the ratio to the time taken without any garbage collection. Item 1 in the table is the case of the entire part of the garbage collector running. In item 2, we have all the overhead of saving acquaintances and detecting loss of Actor references but the actual garbage collection marking is not executed. In the next item, apart from not

No.	Explanation	Total time	Overhead ratio
1	All parts of GC running	2.235 seconds	1.59
2	No marking phase during execution	2.160 seconds	1.54
3	No acquaintances are saved	1.965 seconds	1.40
4	Loss of Actor references not detected	1.395 seconds	0.99

Table 6.2: Breakup of the cost of garbage collection for the 5-queens problem on a single host

No.	Total time without GC	Total time with GC	Overhead ratio
1	2.834 seconds	3.741 seconds	1.32

Table 6.3: Timings for the 4-queens problem on two hosts

executing the marking phase, acquaintances (including inverse acquaintances) are not saved. Finally, in the last item there is practically no overhead. The timing for item 4 is reported as being less than the case of no garbage collector. This is attributed to statistical variance. We see that even without running the marking phases, there is considerable overhead in maintaining the list of acquaintances as shown by item 3 in the table.

6.3.2 Results for two hosts

To investigate the costs associated with running the garbage collector in a distributed environment, the N-queens problem with board size of 4 was run on two Sun Ultra-2 workstations connected by an Ethernet network. A single actor on one of the hosts initiates the computation and every alternate actor is spawned on the remote machine. As before all reported timings are the median values for 10 runs. Table 6.3 presents the results obtained. The ratio of time taken with GC running and without GC is seen to be 1.32. The reason for lower overhead in this case (as compared to the single host case) could be that network communication delays mask the local overhead of garbage collection.

Chapter 7

Conclusions

In this thesis, we have presented a transformation of the actor-reference graph which has the property that if a passive object garbage collector is applied on the transformed graph precisely those objects will be collected which correspond to garbage actors in the original graph. This transformation enables us to reuse for actors, the algorithms that are developed for passive object-oriented systems. It also provides an elegant method to integrate garbage collection of active and passive objects in systems that support both kinds of objects. One requirement for the transformation is the knowledge of inverse acquaintances of all actors. Such a requirement may be considered to be an overhead. There exist actor garbage collectors which do not maintain inverse acquaintances. However, as discussed in section 4.4, in certain cases maintaining inverse acquaintances can reduce the effort required for garbage collection drastically.

Using the transformation technique described in Chapter 4, a previously described algorithm (Schelvis[33] for garbage collection in object oriented systems has been adapted for collecting garbage actors in a distributed environment. The algorithm has been implemented on an actor system based on Java. Experimental results indicate that the ratio of time taken with GC running and without GC is seen to be 1.6 for a single host case and 1.3 for the case of a network with two hosts.

7.1 Future Work

The following is a list of future work that is suggested:

- The implementation could be improved and optimized. The particular areas which can be improved are discussed below:

    1. Considerable overhead results from the need to scan messages for *ActorNames* and in maintaining a list of acquaintances for each actor at all times. If a large number of acquaintances are lost between local garbage collection runs, this leads to unnecessary work. It would be more efficient to generate the list of acquaintances held by each actor at the initiation of every local garbage collection run. This might need a modification in the Java Virtual Machine to allow access to the memory area of objects representing the actors.
    2. In the current implementation, *timestamp* packets are sent individually from each global root to its acquaintances in the global root graph. It might be more efficient to group all packets destined for the same node and transmit them in one message.
    3. Physical addresses of global roots could be cached to save on the overhead of name resolution each time a *timestamp* packet is to be sent.
    4. Currently, the loss of an acquaintance by an actor is detected when Java's internal garbage collector executes the *finalization* method of the *ActorName* by virtue of which the acquaintance was held. This is inefficient because we do not have fine control over Java's internal garbage collector. The Java Virtual Machine could be modified to allow greater cooperation between Java's internal garbage collector and the garbage collector for actors.

- Garbage collection could be enhanced to support migration of actors. Special care will have to be taken when the actor to be migrated is a root actor.

- Other algorithms could be adapted for actors using the transformation technique. A comparison could be made between the different algorithms used.

- An investigation could be made into the cost of maintaining inverse acquaintances. Empirical results could be obtained for comparing the cost of garbage collection, with and without inverse acquaintances, for different reference graphs.

- It would be interesting to see how garbage collection could be applied to other systems like Agent Systems. Agent systems are usually implemented in a distributed environment

and like actors they have autonomous processing power. Therefore the ideas developed in this thesis could be relevant to agent systems. The notion of garbage in agents could be explored.

Bibliography

[1] The Actor Foundry. http://www-osl.cs.uiuc.edu/foundry.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed System s.* MIT Press, Cambridge, Mass., 1986.

[3] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, Reading, MA, USA, second edition, 1998.

[4] Lex Augusteijn. Garbage collection in a distributed environment. In de Bakker et al. [10], pages 75–93.

[5] Ozalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

[6] Yves Bekkers and Jacques Cohen, editors. *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16–18 September 1992. Springer-Verlag.

[7] David I. Bevan. Distributed garbage collection using reference counting. In *PARLE Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187. Springer-Verlag, June 1987.

[8] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed garbage collection for network objects. Technical Report 116, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1993.

[9] Peter B. Bishop. Computer systems with a very large address space and garbage collection. MIT Report LCS/TR–178, Laboratory for Computer Science, MIT, Cambridge, MA., May 1977.

[10] Jacobus W. de Bakker, L. Nijman, and Philip C. Treleaven, editors. *PARLE'87 Parallel Architectures and Languages Europe*, volume 258/259 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 1987. Springer-Verlag.

[11] Peter Dickman. Optimising weighted reference counts for scalable fault-tolerant distributed object-support systems, 1992.

[12] Peter Dickman. Incremental, distributed orphan detection and actor garbage collection using graph partitioning and euler cycles. In Ozalp Babaoglu and Keith Marzullo, editors, *Tenth International Workshop on Distributed Algorithms WDAG'96*, volume 1151 of *Lecture Notes in Computer Science*, Bologna, October 1996. Springer-Verlag.

[13] R. John M. Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud, editor, *Record of the 1985 Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 256–272, Nancy, France, September 1985. Springer-Verlag.

[14] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[15] Neils-Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In Bekkers and Cohen [6].

[16] Dennis Kafura, Manibrata Mukherji, and Doug Washabaugh. Concurrent and distributed garbage collection of active objects. *IEEE Transactions on Parallel and Distributed Systems*, 6(4), April 1995.

[17] Dennis Kafura, Doug Washabaugh, and Jeff Nelson. Garbage collection of actors. In Norman Meyrowitz, editor, *OOPSLA'90 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 25(10) of *ACM SIGPLAN Notices*, pages 126–134, Ottawa, Ontario, October 1990. ACM Press.

[18] Tomio Kamada, Satoshi Matsuoka, and Akinori Yonezawa. Efficient parallel global garbage collection on massively parallel computers. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[19] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *International Conference on Distributed Computing Systems*, Yokohama, June 1992.

[20] Bernard Lang, Christian Quenniac, and José Piquer. Garbage collecting the world. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 39–50. ACM Press, January 1992.

[21] Fabrice Le Fessant, Ian Piumarta, and Marc Shapiro. An implementation for complete asynchronous distributed garbage collection. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[22] Sylvain Louboutin and Vinny Cahill. A lazy log-keeping mechanism for comprehensive global garbage detection on Amadeus. In *OOIS (Object-Oriented Information Systems) '95*, pages 118–132, London, December 1995. Springer-Verlag. Technical report TCD–CS–95–11.

[23] Sylvain R.Y. Louboutin. *A Reactive Approach to Comprehensive Global Garbage Detection.* PhD thesis, Trinity College, Dublin, 1998. In preparation.

[24] Sylvain R.Y. Louboutin and Vinny Cahill. Comprehensive distributed garbage collection by tracking causal dependencies of relevant mutator events. In *Proceedings of ICDCS'97 International Conference on Distributed Computing Systems.* IEEE Press, 1997.

[25] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by back tracing. In *Proceedings of PODC'97 Principles of Distributed Computing*, 1997.

[26] Khayri A. Mohamed-Ali. *Object Oriented Storage Management and Garbage Collection in Distributed Processing Systems.* PhD thesis, Royal Institute of Technology, Stockholm, December 1984.

[27] Jeffrey E. Nelson. Automatic, incremental, on-the-fly garbage collection of actors. Master's thesis, Virginia Polytechnic Institute and State University, 1989.

[28] José M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In Aarts et al., editors, *PARLE'91 Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1991.

[29] Isabelle Puaut. Distributed garbage collection of active objects with no global synchronisation. In Bekkers and Cohen [6].

[30] Isabelle Puaut. A distributed garbage collector for active objects. In *PARLE'94 Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science. Springer-Verlag, 1994. Also INRIA UCIS-DIFUSION RR 2134.

[31] Helena C. C. D. Rodrigues and Richard E. Jones. Cyclic distributed garbage collection with group merger. In Erik Jul, editor, *Proceedings of 12th European Conference on Object-Oriented Programming, ECOOP98*, Lecture Notes in Computer Science, pages 249–273, Brussels, July 1998. Springer-Verlag. Also UKC Technical report 17–97, December 1997.

[32] Gustavo Rodriguez-Riviera and Vince Russo. Cyclic distributed garbage collection without global synchronization in CORBA. In Peter Dickman and Paul R. Wilson, editors, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997.

[33] M. Schelvis. Incremental distribution of timestamp packets — a new approach to distributed garbage collection. *ACM SIGPLAN Notices*, 24(10):37–48, 1989.

[34] Reinhard Schwarz and Friedemann Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, 1994.

[35] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapports de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, November 1992. Also available as Broadcast Technical Report 1.

[36] Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. Hierarchical garbage collection in scalable distributed systems. Technical Report UIUCDCS-R-92-1740, Dept. of Computer Science, University of Illinois at Urbana-Champaign, April 1992.

[37] Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. Scalable distributed garbage collection for systems of active objects. In Bekkers and Cohen [6], pages 134–147.

[38] Douglas Markham Washabaugh. Real-time garbage collection of actors in a distributed system. Master's thesis, Virginia Polytechnic Institute and State University, 1989.

[39] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In de Bakker et al. [10], pages 432–443.

[40] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.