Garbage Collection of Actors

By Dennis Kafura, Doug Washbaugh
and Jeff Nelson

TR 90-9

Garbage Collection of Actors

Dennis Kafura
Department of Computer Science
Virginia Polytechnic Institute
and State University
Blacksburg, VA 24061
email: kafura@vtopus.cs.vt.edu
phone: 703-231-5568

Doug Washabaugh
Digital Equipment Corporation
tay2-2/b4
153 Taylor Street
Littleton, MA 01460
email:washabaugh@quiver.enet.dec.com
phone: 508-952-3535

Jeff Nelson
Digital Equipment Corporation
ZK02-3/N30
110 Spitbrook Road
Nashua, NH 03062
email: jnelson@tle.enet.dec.com
phone: 603-881-0867

Abstract

This paper considers the garbage collection of concurrent objects for which it is necessary to know not only "reachability", the usual criterion for reclaiming data, but also the "state" (active or blocked) of the object. For the actor model, a more comprehensive definition than previously available is given for reclaimable actors. Two garbage collection algorithms, implementing a set of "coloring" rules, are presented and their computational complexity is analyzed. Extensions are briefly described to allow incremental, concurrent, distributed and real-time collection. It is argued that the techniques used for the actor model applies to other object-based concurrent models.

Keywords: garbage collection, object-based concurrency, automatic memory management, actors, ACT++

Section I. Introduction

Great interest has recently been shown in object-based concurrent languages. These languages are considered useful for at least the following reasons:

- parallelism inherent in a system of concurrent objects can be exploited readily by parallel architectures [Athas 1987],
- asynchrony among entities in the real world can be represented directly by concurrent objects [Kafura 1988],
- distributed applications can be programmed "naturally" using concurrent objects interacting via messages [Black 1987] [Yonezawa 1987],
- conceptual economy results from a single object abstraction which unifies the notions of a processor (thread of control), memory (encapsulated variables) and communication (messages).

We are interested in using object-based concurrent languages in applications which are distributed, embedded and time sensitive [Kafura 1988].

A fundamental implementation concern for object-based languages is memory management. Automatic methods (i.e., garbage collection) for managing memory are preferable because:

- programmer controlled memory management is notoriously error-prone [Bloom 1987],
- a better division of responsibility results when the system does what it does best (manage resources) and the programmer does what programmers do best (design systems),
- system-wide optimization of memory usage and memory management mechanisms is best achieved in a regime of automatic resource control.

The above motivations apply to sequential as well as concurrent object-based languages. Additional reasons for using garbage collection in object-based concurrent languages are:

- it is significantly more difficult for a programmer to correctly manage concurrent objects than passive data because both reachability and state must be considered,

- garbage concurrent objects not only consume memory space but may also consume processing capacity making it even more imperative that garbage concurrent objects be identified quickly.

- in distributed applications it is unlikely that a programmer could devise a correct and efficient distributed algorithm for managing distributed resources.

As implied above, garbage collection of concurrent objects differs from garbage collection of sequential objects or data. Roughly speaking, an actor may be defined as garbage if it lacks either one (or both) of the following properties:

- <u>computable:</u> the actor is active or can become active hereafter

- <u>reachable:</u> the actor can send information to or receive information from a "root".

Virtually all previous garbage collectors have focused exclusively on determining an object's reachability. Notable exceptions are garbage collection schemes for functional, distributed and/or actor-based languages

Baker and Hewitt [Baker 1977] described a variation of a mark-and-sweep garbage collector for functional languages which collected garbage expressions. Garbage collection in functional languages was also explored by Hudak [Hudak 1982] [Hudak 1983] who presented algorithms for marking a directed graph representing a distributed functional program. These algorithms can be applied to the garbage collection of distributed functional objects. This work is relevant because of the close parallel between the concurrent evaluation of functional expressions and the concurrent execution of objects. However, there are basic differences between the functional model and the concurrent object model:

functional models do not have cyclic dependencies while concurrent objects may; concurrent objects do not evaluate to a single result as do functional expressions. Thus, the garbage collection techniques developed for functional languages are not directly applicable to object-based concurrent languages.

Halstead's garbage collector for distributed actors ([Halstead 1978]) uses the concept of an *actor reference tree*, which is a set of processors and connections between processors such that each processor has a reference to the actor. Garbage collection is performed by the reducing the actor reference tree until it contains a single processor. A local garbage collector is then used on each processor to collect garbage actors. A drawback of this method is that it cannot detect cyclic garbage. The algorithms presented in this paper collect cyclic garbage.

Emerald uses an object-based language for programming distributed applications [Black 1987]. Garbage collection in Emerald is discussed briefly in [Jul 1988]. It is interesting to note that the overall structure of Emerald's distributed collection system is very similar to our own (see Section 5). From the standpoint of the actor model, the principle limitation of the Emerald approach is that it considers only reachability of objects. As indicated above this is too weak a criterion for collecting garbage in the actor model.

The remainder of this paper is organized as follows. Section 2 presents a brief overview of the actor model, define reclaimable actors and show more explicitly why previous garbage collection techniques are not directly applicable for collecting reclaimable actors. Section 3 describes the rules for identifying reclaimable actors and shows two algorithms for implementing these rules. The computational complexity of the two algorithms is analyzed in Section 4. Finally, Section 5 briefly describes how these algorithms can be extended to allow for concurrent, incremental, real-time and distributed collection.

Section II. The Actor Model and Reclaimable Actors

The Actor Model

A detailed description of the actor model can be found in [Agha 1986]. The principle features and terminology of the actor model which relate to the garbage collection problem are these:

- **actor**: a concurrently active object. There are no passive entities. Each actor is uniquely identified by the address of its single mail queue.

- **acquaintance**: actor B is an acquaintance of actor A if B's mail queue address is known to actor A.

- **inverse acquaintance**: if actor A is an acquaintance of actor B, then actor B is an inverse acquaintance of A.

- **acquaintance list**: a set of mail queue addresses including any mail queue address contained in a message on the actors mail queue or in transit to the mail queue. This accounts for delays in message processing.

- **topology**: actors may be dynamically created and actors may be dynamically bound (i.e., acquaintances can be passed at run-time through mail messages).

- **behavior**: a thread of execution within an actor. There may be many active threads within an actor, each thread processing a different mail message.

- **blocked actor**: an actor all of whose behaviors are blocked.

- **active actor**: an actor with at least one active behavior.

- **root actors**: an actor designated as being "always useful". Examples of root actors are those which have the ability to directly affect real-world through sensors, actuators , I/O devices, users, etc.

It is important to make the following two observations. First, the garbage collection problem and its solution as presented in this paper are not limited to the actor model. Similar techniques should be applicable to any object-based concurrent model which has the following general properties: encapsulated objects interact exclusively via messages; the communication structure is not static; the state and acquaintances of each object can be determined. Second, the extended message passing primitives used in the actor-based language which we are defining, ACT++ [Kafura 1990], do not interfere with the properties just noted.

Table 1 shows the symbols used to depict an actor system. An example actor system is shown in Figure 1.

Table 1. Legend for Actor Figures

Symbol	Interpretation of Symbol
□	Blocked Actor
○	Active Actor
△	Root Actor
→	Acquaintance Arc

To illustrate the features of the actor model and to form an appreciation of the complexity of determining which actors are garbage, consider the actor system shown in Figure 1. In this figure, actors A and I are root actors and by definition are not garbage. It can be easily seen that actors J,K,L and M are garbage - they cannot communicate with a root actor. Whatever actions they take cannot be made visible to the outside world. Notice that J and M are active while K and L are blocked. This shows that state alone is not a sufficient criterion. Actor E is blocked and there is no way for it to become active because it has no inverse acquaintances. However, actor H also has no inverse acquaintances but it is not garbage because it is active and can communicate directly with the root actor I. Message from the root actor A can reach actors B,C,D and G. If these messages contain A's mail queue address, these four actor can become active and communicate directly with a root actor. Hence, they are not garbage. Finally, actor F could send a message to G containing F's own mail queue address. G in turn could send A's mail queue address to F allowing F to communicate with the root actor A. So F is not garbage. This example illustrates that the relationships between the computable and reachable criteria is not simple.

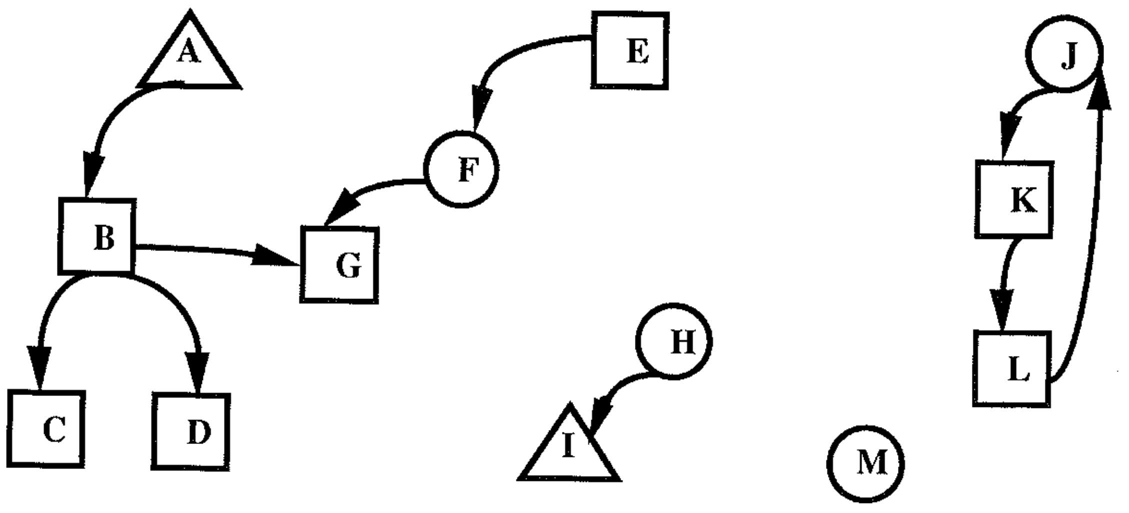

Figure 1. An Actor System

Notice that if a traditional marking algorithm is used for the system shown in Figure 1, actors E,F and H are not reachable from a root. These three actors would be incorrectly marked as garbage. Also, reference counting can miss actors which are garbage. In Figure 1, actors J, K and L all have non-zero reference counts. Even though all of them are garage they would not be considered as garbage by the reference counting scheme.

Definition of Garbage in the Actor Model

Informally, garbage actors are those whose presence or absence from the system cannot be detected by external observation. Excluded from interest are any visible effects due simply to the consumption of resources by garbage actors (e.g., increasing response time).

Agha [Agha 1986] defined an actor to be garbage if it is not processing any messages, if it is not the target of some undelivered message, and if it is not an acquaintance of any other actor. This definition is not comprehensive - there are actors which, according to the informal notions given above, are garbage but do not conform to Agha's definition. For example, in Figure 1 actor J cannot communicate with a "root" actor. By our understanding it is, therefore, garbage. However, if fulfills none of Agha's requirements. A more comprehensive definition of garbage actors, given in [Nelson 1989] or [Washabaugh 1990], is now briefly reviewed.

A garbage actor is one which is:

1. not a root actor, and
2. cannot *potentially* receive a message from a root actor, and
3. cannot *potentially* send a message to a root actor.

In this definition, the term " potentially" requires further clarification. An actor *can* send a message to a root actor only if the actor is active and has the root actor as a direct acquaintance. There is a set of transformations that can change an actor graph from a representation of what can *currently* happen to what can *potentially* happen. The two transformations concern change in the state (ready, blocked) of an individual actor and change in the topology of the system of actors. First, sending a message from an active or root actor to a blocked acquaintance allows the blocked actor to become active. This transformation reflects the ability of an actor to alter the state (ready or blocked) of another actor.The second transformation occurs when a root or active actor sends its own mail queue address or the mail queue address of one of its acquaintances to another of its acquaintances. This transformation reflects the ability of an actor to send a mail queue address, thereby changing the topology of the actor system.

Suppose an actor system has the above transformations repeatedly applied until no more transformations can be applied. Then, all actors which are not direct acquaintances of root actors in the resulting topology are garbage.

One key property of garbage actors is that they cannot become non-garbage. This is because actors are only determined to be garbage when there is no possibility of communication between it and a root actor. Therefore, once an actor is marked as garbage, there is no possible sequence of transformations which would cause the garbage actor to become non-garbage.

Section III. Marking Algorithms

The marking algorithms presented in this section assumed that the mutator is halted and that all actors in the system reside on the same node. In the final section of this paper we will describe extensions which remove these restrictions.

The marking algorithms use three colors (white, gray and black) which, at then end of marking, have the following meanings:

White: Actors colored white are not reachable from a root actor.

Gray: Actors colored gray are reachable from a root actor, but can not become active.

Black: Actors colored black are non-garbage. They are either root actors or are both reachable from a root actor and potentially active.

The colors of actors can only be darkened. Black is darker than gray which is darker than white.

Coloring Rules

Underlying the marking algorithms is a set of coloring rules defined in [Nelson 1989] and described in Figure 2. Rule 1 colors black actors that can receive message directly from a non-garbage actor. Rule 2 colors black actors that can send a message directly to a non-garbage actor. Rule 3 colors gray actors that could be non-garbage if they could become active. Rule 4 colors black actors which are currently gray (reachable) and can have a message sent to them from some active actor. The actor is colored black because it has both the reachable and computable properties. Rule 5 colors gray all blocked actors which, if they became computable, could send a message directly to a gray actor.

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:

Rule 1.	Color black all acquaintances of black actors
Rule 2.	Color black all inverse acquaintances of black actors if the inverse acquaintance is not blocked.
Rule 3.	Color gray all inverse acquaintances of black actors if the inverse acquaintance is blocked.
Rule 4.	Color black all inverse acquaintances of gray actors if the inverse acquaintance is not blocked.
Rule 5.	Color gray all inverse acquaintances of gray actors if the inverse acquaintance is blocked.

3. Actors that are colored black are not garbage. Gray and white actors are garbage and can be reclaimed.

Figure 2 Nelson's Coloring Rules

Push-Pull Marking Algorithm

One implementation of the coloring rules uses two coroutines, a Pusher and a Puller, to move actors between black, gray, and white sets. The Pusher operates on actors in the white set. It pushes actors from the white set into the gray and black sets. The Puller operates on actors in the black set. It pulls actors from the white and gray sets into the black set. The algorithm for the Push-Pull marker is shown in Figure 3.

The pusher coroutine implements rule 1 of Nelson's algorithm. The first case of the puller implements rules 2 and 4, and the second case implements rules 3 and 5.

```
BEGIN Initialization
    All root actors are placed in the black set.
    All other actors are placed in the white set.
    Resume Puller
END Initialization

BEGIN Puller
    FOR [each actor in the black set not yet examined]
        place non-black acquaintances of the actor in the black set
    END FOR
    resume Pusher
END Puller

BEGIN Pusher
    FOR [each actor in the white set]
        CASE:  actor is active and an acquaintance is black or gray
                   -> place actor in black set
        CASE:  actor is blocked and an acquaintance is black or gray
                   -> place actor in gray set
    END FOR
    IF [any actors were placed in the black or gray set]
        THEN resume Puller
        ELSE Termination
END Pusher

Termination:
    All actors which are not black are garbage
```

Figure 3. Push-Pull Algorithm

The actions of the Push-Pull algorithm are illustrated using the actor system shown in Figure 1. The initialization step puts actors A and I (root actors) in the black set and all other actors in the white set. For simplicity assume that actors are examined in alphabetical order. In the first pass, the Puller moves B into the black set since it is an acquaintance of A. A, B and I are now in the black set and A has been examined. The Puller next examines B and pulls its acquaintances (C,D and G) into the black set. The remaining elements in the black set do not have acquaintances so the Puller will finish without adding any other actors to the black set. The actors in the white set are now E,F,H,J,K,L and M. The Pusher will move F and H to the black set and leave all others unchanged. On pass 2 the Puller makes no changes while the Pusher moves E to the gray set (it is reachable but cannot become active). On pass three the Puller again takes no action. When the Puller also takes no action, it terminates. At the termination, the black set contains A, B, C, D, F, G, H, I. All other actors are garbage.

Coloring Algorithm Is Black

This coloring algorithm, shown in Figure 4, uses two colors and a *visit* field. The first rule in the repeat loop colors black acquaintances of black actors. The second rule does a depth first search from active actors for a black actor. If a black actor is found, then the originating active actor is colored black. The visit field is used to detect cycles during the depth first search.

```
All anchors are colored black, all other actors are colored white
pass = 0

BEGIN Is_Black
    REPEAT
        increment pass
        color black all acquaintances of black actors
        FOR all white active actors DO Depth_First(actor, pass)
    UNTIL [no new markings are made]
END Is_Black

BEGIN Depth_First(actor, pass)
    IF [actor == black] RETURN true                       ;found black
    ELSE IF [actor.visit == pass] RETURN false            ;cycle detected
    ELSE
        actor.visit = pass
        FOR [each acquaintance of the actor]
            IF [depth_first(acquaintance, pass)==true]    ;color return path
                color actor black
                RETURN true
            END IF
        END FOR
    END IF
    RETURN false
END Depth_First

Termination:
    All non-black actors are garbage
```

Figure 4. Is_Black Algorithm

Figure 5 shows an actor configuration which demonstrates some features of the "Is_Black" algorithm. At the start of the algorithm, actor F is colored black because it is a root actor, and all other actors are colored white. The algorithm begins by coloring actor G black because it is an acquaintance of a black actor.

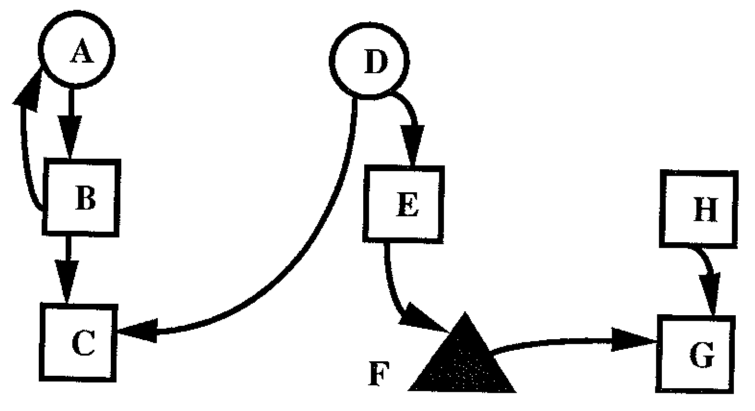

Figure 5. Is_Black Example

The algorithm next does a depth first search from active actor A. Note the cycle between actors A and B. The algorithm applies the function Is_Black to actor B, which in turn, queries actor A. The query from actor A returns false , because it has already been visited. When actor B queries actor C, it returns false because it is not black, nor does it have any black acquaintances. Therefore, the depth first search from actor A failed, so it remains white.

The algorithm next does a depth first search from active actor D. It eventually encounters actor F, which is colored black, so actors D and E are colored black. Actor E was colored black because it was on the search path.

Because an actor was colored, the first step of the algorithm is repeated. This causes actor C to be colored black because it is an acquaintance of black actor D. Next, a depth first search is done from active actor A. This time the depth first search finds black actor C, so actor A is colored black.

Since an actor was again darkened, the algorithm is repeated. The first step colors actor B black, because it is an acquaintance of black actor A. The algorithm repeats itself once more, but no more colorings are done. At the termination, all actors except for actor H, which is the only garbage actor, are colored black.

Section IV. Computational Complexity

This section compares the worst-case space and time complexity of the two algorithms presented in the previous section.

Space Complexity

The Push-Pull algorithm requires that each actor contain a color field or a link to other actors that are in the same color set. In either case, the size of the field is constant. Therefore, its space complexity is:

$$\text{Space Complexity} = k_1 N = \text{Order}(N)$$

where N is the number of actors in the system and k_1 is the size of the color or link field.

The Is_Black algorithm recursively does depth-first searches, which require stack space. Each recursion requires a constant amount of stack space, and the worst-case number of recursions is the maximum of:

1. Number of actors traversed until a cycle is reached.
2. Number of actors traversed until an actor with no acquaintances is reached.

Thus, the worst-case space complexity is:

$$\text{Space Complexity} = k_1 N = \text{Order}(N)$$

where N is the number of actors in the system and k_1 is the size of a stack frame on recursion.

Time Complexity

Figure 6 shows an example of the worst case actor graph for the Push-Pull algorithm. The initialization phase colors actor E black and actors A, B, C, D white. Since the order in which the algorithms examines the actors is arbitrary, let us suppose that the Pusher always examines the actors in alphabetical order while the Puller always examines them in reverse alphabetical order. In pass 1, after having examined and taken no action for actors A,B and C, actor D is moved to the gray set . Similarly in pass 2, actor C is moved to the gray set and in pass 3, actor B is moved to the gray set. In each of these passes all actors between actor A and the actor eventually moved to the gray set are examined. In pass 4, actor A is colored black. During these first four passes the puller has taken no actions (since E has no acquaintances). In the next four passes the pusher takes no action, because the white set is now empty, while the puller examines all actor in the gray set and moves one of them on each pass into the black set.

In general, the worst case time complexity for the Push-Pull algorithm is:

$$\begin{aligned}
\text{Complexity} &= \text{Number of passes} * \text{Number of actors considered on each pass}\\
\text{Complexity} &= \text{Order(Length)} * \text{Order(Length)}\\
\text{Complexity} &= \text{Order}(\text{Length}^2)\\
&= \text{Order}(N^2)
\end{aligned}$$

The worst-case situation occurs when all N actor form the chain shown in Figure 6.

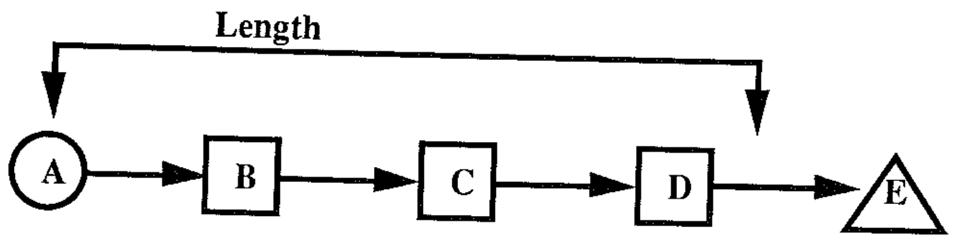

Figure 6. Worst Case for Push-Pull Algorithm

Figure 7 shows an example of the worst case for the Is_Black algorithm. Suppose that the algorithm considers the actors in this example in alphabetical order. The initialization phase colors actors A, B, C, D, E white and actor F black. First, a depth-first search is done starting at actor A. It fails, so another depth-first search is started at actor C. It too fails. The next search is started at actor E, which succeeds. Actors E and D are now colored black. The next depth-first search is from actor A, which fails, but the next one, from actor C, succeeds. Actors C and B are now colored black. The next search from actor A succeeds, so it too is colored black.

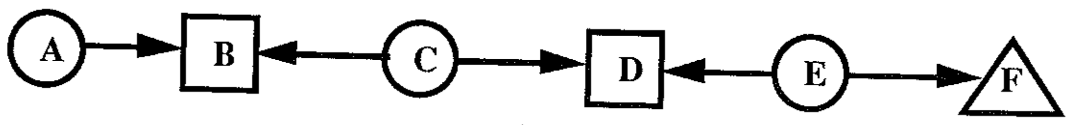

Figure 7. Worst Case for Is_Black Algorithm

The worst case complexity of the Is_Black algorithm, in terms of number of actors examined, is:

Complexity = Passes * Number of searches * Number of actors searched
Complexity = (N/2) * (N) * 2
Complexity = Order(N^2)

Of the two algorithms, the Push-Pull algorithm has the lower space complexity. However, both algorithms have equal worst-case time complexities, although it is not possible to determine which is the most efficient in typical actor graphs.

Section IV. Conclusions and Extensions

In this paper we have presented a comprehensive definition of garbage in actor systems. Based on this definition a set of coloring rules were given for marking garbage actors. Two algorithms implementing the coloring rules were then presented. The space and time complexity of the algorithms were determined.

The garbage collection algorithms presented in this paper are the first steps in achieving a practical garbage collector for actor systems. The weakness of these algorithms is that they assume that the actor system (the mutator) is halted while the marking and reclamation are performed. Forcing the mutator to halt during garbage collection is unacceptably restrictive for most applications. A series of extensions, fully presented in [Washabaugh 1990], allow the garbage collection to achieve the following goals:

<u>concurrent mutator/collector:</u>

To allow for concurrent execution, the mutator and collector must cooperate in two ways. First, they must synchronize their access to shared implementation structures. This is easy. What is harder is the second form of cooperation: the collector must take a snapshot of the actor system on which it will work while allowing the mutator to migrate away from this snapshot state.

<u>incremental collection:</u>

When the mutator and collector are executed on a single processor system, concurrent mutator/collector operation may imply that the mutator is interrupted for long periods of time. It is useful to minimize the length of this interruption by interleaving incremental actions of the collector into allocation operations of the mutator. This is not too difficult with the Push-Pull algorithm because a "few" steps in the algorithm can be performed at each allocation. This is not possible with the Is_Black algorithm.

real-time collection:

Not only should the collector work incrementally, but the period of time during which the mutator is interrupted must be strictly bounded. Furthermore, it must still be guaranteed that the entire reclamation process is completed before the mutator consumes all available memory. The incremental extension of the Push-Pull algorithm can be strictly bounded.

distributed collection:

Collecting distributed garbage presents two major problems. First, the global collector must operate concurrently with the local collectors/mutators and must synchronize properly with the local collector. This synchronization can be achieved again by using a snapshot approach and by "time-stamping" inter-node acquaintances. Second, the distributed pieces of the global collector must be able to determine when to terminate. The termination is complicated because a global collector at one node may finish all of its work only to be reawakened later by the action taken at another node. Agreement can be achieved by using a rotating token which, if it ever returns to its last "owner", signals termination.

References

[Agha 1986] Gul Agha, Actors: A Model of Concurrent Computation in Distributed Systems, M.I.T. Press, Cambridge, Massachusetts, 1986.

[Athas 1987] W. Athas, "Fine Grain Concurrent Computations," Technical Report 5242:TR:87, Computer Science Department, California Institute of Technology, 1987.

[Baker 1977] Henry Baker and Carl Hewitt, "The Incremental Garbage Collection of Processes," M.I.T. Artificial Intelligence Laboratory, Memo 454, December 1977.

[Black 1987] Andrew Black, Norman Hutinson, Eric Jul, Henry Levy and Larry Carter, "Distribution and Abstract Types in Emerald," IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, January 1987, p.65-76.

[Bloom 1987] Tony Bloom and Stanley Zdonick, "Issues in the Design of an Object-Oriented Database Programming Language," OOPSLA'87, October 1987, p.441-451.

[Halstead 1978] Robert Halstead, "Multiple-Processor Implementations of Message Passing Systems," M.I.T. Laboratory for Computer Science, Technical Report 198, April 1978.

[Hudak 82] Paul Hudak and Robert Keller, "Garbage Collection and Task Deletion in Distributed Applicative Processing Systems," Symposium on Lisp and Funtional Programming, 1982, p.168-178.

[Hudak 1983] Paul Hudak, "Distributed Task and Memory Management," 2nd Annual ACM Symposium on Principles of Distributed Computing, 1983, p.277-289.

[Jul 1988] Eric Jul, Henry Levy, Norman Hutchinson and Andrew Black, "Fine-Grain Mobility in the Emerald System," ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988, p.109-133.

[Kafura 1988] Dennis Kafura, "Concurrent Object Oriented Real-Time Systems Research," Technical Report 88-47, Department of Computer Science, Virginia Tech, Blacksburg, VA, 1988.

[Kafura 1990] Dennis Kafura and Keung Lee, "ACT++: Building A Concurrent C++ With Actors," Journal of Object-Oriented Programming, to appear, 1990, also Technical Report 89-18, Department of Computer Science, Virginia Tech, Blacksburg, VA.

[Nelson 1989] Jeff Nelson, Automatic, Incremental, On-the-fly Garbage Collection of Actors, M.S. Thesis,Department of Computer Science, Virginia Tech, Blacksburg, VA, February 1989.

[Washabaugh 1990] Doug Washabaugh, Real-Time Garbage Collection of Actors in a Distributed System, M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, February 1990.

[Yonezawa 1987] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda, "Modelling and Programming in an Object-Oriented Concurrent Language, ABCL/1," in Object-Oriented Concurrent Programming (A. Yonezawa and M. Tokoro, eds), p.55-89, MIT Press, Cambridge, Massachusetts, 1987.