Distributed Garbage Collection of Active Objects

By Douglas M. Washabaugh and Dennis Kafura

TR 90-53

Distributed Garbage Collection of Active Objects

by

Douglas M. Washabaugh
Digital Equipment Corporation
tay2-2/b4
153 Taylor Street
Littleton, MA 01460
washabaugh@quiver.enet.dec.com

Dennis Kafura
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106
kafura@vtopus.cs.vt.edu

ABSTRACT

This paper shows how to perform distributed automatic garbage collection of objects possessing their own thread of control. The relevance of garbage collection and concurrent objects to distributed applications is briefly discussed and the specific model of concurrent objects used in the paper is explained. The collector is comprised of a collection of independent local collectors, one per node, loosely coupled to a distributed global collector. The mutator (application), the local collectors and the global collector run concurrently. The synchronization necessary to achieve correct and efficient concurrent operation between the collectors and between the collectors and the mutator is presented in detail. An interesting aspect of the distributed collector is the termination algorithm: the collector algorithm running on one node, which considers itself to be "done", may become "undone" by the action of a collector algorithm on another node.

1. Introduction

The search for an effective paradigm for distributed computing has recently included object-oriented languages for distributed programming [Black 1987] [Yonezawa 1987] as well as distributed kernels supporting forms of object-oriented programming [Leddy 1989] [Dasgupta 1988]. For distributed systems to realize the well known advantages of object-oriented programming the development of a distributed run-time environment similar to that usually found in non-distributed object-oriented systems is required.

This paper focuses on an important element of the distributed object-oriented run-time environment: the garbage collector. While not all object-oriented languages use garbage collection, we argue below that garbage collection is preferable to programmer controlled memory management. The garbage collection problem considered in this paper is complicated by the fact that the objects being managed are active objects. An active object is one which encapsulates its own thread of control. We explain below why active objects are useful in general and why they are particularly useful in distributed programming. The actor model [Agha 1986] is used as the framework for presenting the algorithms in this paper. However, the basic ideas apply to a larger class of object models.

The use of automatic garbage collection and active objects is motivated by the following observations:

- programmer controlled memory management is notoriously error-prone [Bloom 1987]. Failing to return a resource that is no longer used or returning a resource that is being used are mistakes that are difficult to detect and repair. Furthermore, it is significantly more difficult for a programmer to correctly manage active resources than traditional data resources.

- the complexity of storage management in a distributed environment implies that it must be an integral part of the underlying automatic resource control system [Appel 1988]. It is unlikely, or at least severely expensive, for each designer to define and implement a collector for each new application.

- associating a thread of control with each object creates a uniform object model which, in a distributed environment, combines the distinct notions of object mobility [Jul 1988] and process migration [Cheriton 1988].

- autonomous, interacting real-world entities are more easily and more directly expressed in software as concurrent encapsulated objects with strictly limited interactions. This issue is explored further in [Kafura 1988].

This work is also motivated by a singular deficiency of virtually all existing collectors: they do not manage active objects.

The garbage collection system presented in this paper consists of two primary components: a local collector operating at each node and a distributed global collector. Both collectors utilize a similar mark-and-sweep algorithm. The local collector executes as required by the conditions of its local memory and is capable of reclaiming purely "local" garbage. Because it uses only local information, the local collector is not capable of reclaiming "global" garbage, that is, objects which depend on references to objects stored at other nodes. Periodically, the global collector, a distributed algorithm, is initiated. The global collector distributes information about objects dependent on references to non-local objects and marks those objects which are garbage. Object so marked on a given node are subsequently reclaimed by that node's local collector.

Although the local and global collectors execute concurrently, performance considerations dictate that they cooperate to avoid duplication of effort in marking nodes. Correctness requires that the local and global collectors properly synchronize updates to shared data structures. Furthermore, the collectors must coordinate properly with the mutator (the application) in order to guarantee the consistency of the actor system that is viewed by each.

The remainder of this paper is organized as follows. Section 2 describes the model of active objects, the actor model, that is used in this paper. Garbage actors are illustrated by example and an algorithm is given for reclaiming garbage actors in a non-distributed system. This section also contains a review of related collectors. Section 3 describes the local collector. The global collector, including the global collector's termination algorithm, is presented in Section 4. Section 5 defines the necessary synchronization among the local and global collectors and consistency between the collectors and the mutator. Conclusions are given in Section 6.

2. Actors and Garbage Collection

The features of the actor model relevant to garbage collection are illustrated by an example. The notation used in this example is summarized in Table 1 and the example is shown in Figure 1.

Table 1. Legend for Actor Figures

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

The state of the message driven computation in an actor system can be depicted by a graph whose nodes represent actors and whose directed arcs represent acquaintances. An acquaintance arc from actor S to actor T means that, whenever S is active, S can send a message to T. Actors are either active (drawn as circles) or blocked (drawn as squares). An active actor may send mail messages asynchronously to its acquaintances and may also create new actors.

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 relevant to the garbage collection problem are change in the state (active or blocked) of an individual actor and change in the topology of the system of actors. First, sending a message from an active actor to a blocked acquaintance allows the blocked actor to become active. For example, in Figure 1, if F send a message to G, G becomes active. Second, an active actor can send its own mail queue address or the mail queue address of one of its acquaintances to another of its acquaintances. This transformation changes the topology of the actor system by introducing a new acquaintance arc. For example, in Figure 1, if B were to become active it could send the mail address of C to G, causing a new acquaintance arc to be introduced from G to C.

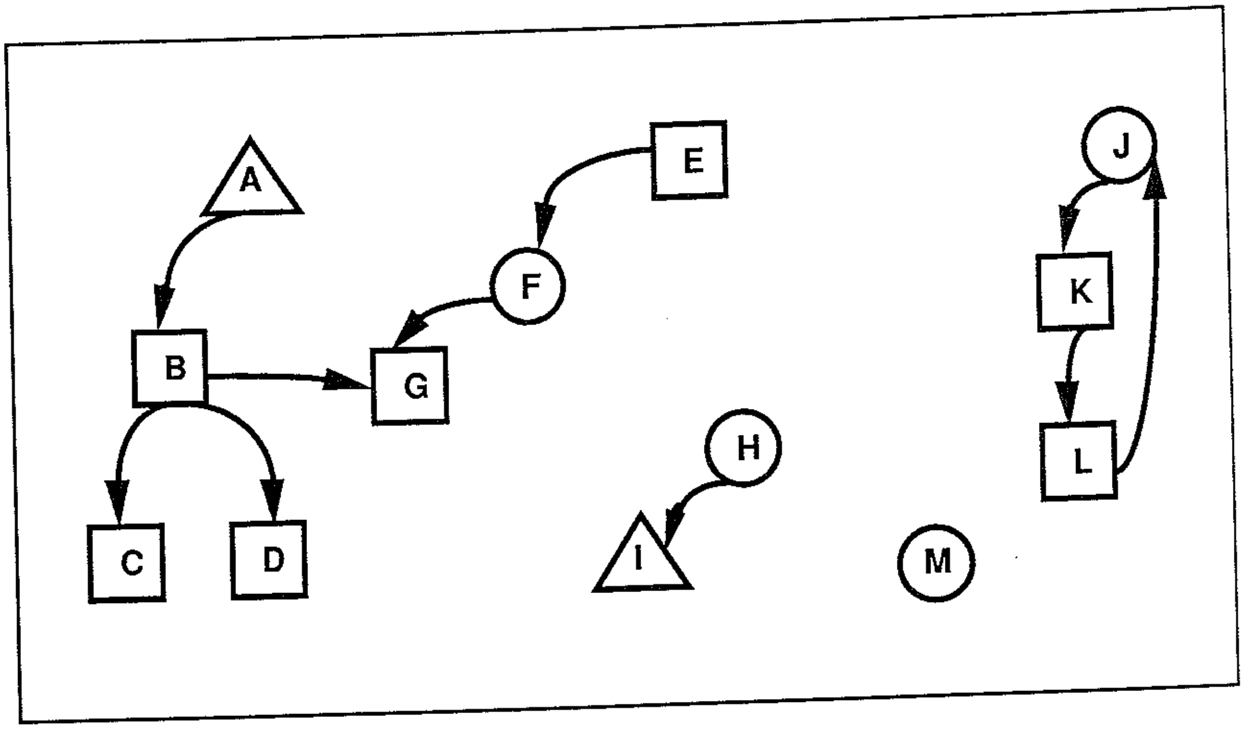

Figure 1. An Actor System

Informally, garbage actors are those whose presence or absence from the system cannot be detected by external observation excluding any visible effects due simply to the consumption of resources by garbage actors (e.g., increasing response time). To make this idea more concrete, a third type of actor, drawn as a triangle in Figure 1, is introduced. These actors are defined to be the "roots" of the actor system. Root actors are always consider to be active. Intuitively, root actors are the means by which the actor computation affects the "outside world." Root actors, for example, represent actuators or output ports.

Somewhat more precisely, 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" refers to the results of applying the two transformations described above. A more comprehensive definition of garbage actors is given in [Washabaugh 1990].

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

Let us now consider which actors in Figure 1 are garbage. 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 for identifying garbage actors. . Actor E is blocked and there is no way for it to become active because it is not the acquaintance of any other actor. However, actor H also is not the acquaintance of any other actor, 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 state of an actor and the current topology is complex.

Notice that if a simple marking algorithm is used for the system shown in Figure 1, actors E,F and H would be incorrectly marked as garbage because they are not reachable from a root. Also, simple 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 garbage they would not be considered as garbage by a reference counting scheme.

A Non-Distributed Collector

The Push-Pull algorithm shown in Figure 2 implements the rules defined in [Nelson 1989] to "color" actors in a system. A more detailed description of the algorithm is contained in [Kafura 1990]. The algorithm uses three sets. Each set is named by a color with the following meanings:

•White	Actors in this set are not necessarily reachable from a root actor.
•Gray	Actors in this set are reachable from a root actor, but may not necessarily become active.
•Black	Actors in this set are non-garbage.They are either root actors or are both reachable from a root actor and potentially active.

Every actor is always in exactly one of these three sets.The color of an actor is the color of the set of which it is currently an element.

The algorithm's name comes from its use of two coroutines: one coroutine "pushes" actors from the white and the gray sets into the black set, and the other coroutine "pulls" actors from the white and gray sets into the black set. It is assumed that the mutator is halted while the coloring algorithm executes.

```
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 2. 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.

Related Garbage Collectors

The vast majority of garbage collection algorithms apply only to non-distributed passive objects. Fewer collectors have been developed for distributed systems but they focus exclusively on determining an object's "reachability" which, as was seen in Figure 1, is too weak a criteria for collecting active objects such as actors. Other collectors have been designed for active objects but these are either non-distributed or use a more limited model of computation.

One of two recent distributed garbage collectors of interest is the garbage collector included in the Emerald System [JUL88]. Emerald is an object based language and system for constructing distributed programs. Emerald's garbage collector uses a local and distributed garbage collector. Local collectors are independent of other nodes, but the distributed collector requires that all nodes cooperate. Each collector is based on the mark and sweep method. Distributed garbage collection consists of nodes exchanging coloring information. When it is determined that all nodes have completed their coloring [by an undescribed consensus algorithm], garbage collection is complete. The garbage collector presented in this paper and the Emerald garbage collector have several similarities. Both use a local and distributed garbage collector, both require global synchronization for distributed garbage collection, and both exchange coloring information among nodes. The key difference between the collectors is their marking algorithms: Emerald uses a traditional mark and sweep algorithm based on reachability.

The second related distributed garbage collector, due to Schelvis [SHEL89], is an incremental and concurrent algorithm based on timestamps. No synchronization between hosts is necessary, and hosts may be temporarily inaccessible. It is claimed to be the first incremental algorithm that is capable of collecting cyclic distributed garbage. A local collector has two functions: collect local garbage, and to gather and send remote references to the appropriate remote host. Each host maintains a host entrance table which is a timestamped list of nodes that are remotely referenced. A host entrance table is considered a root object, so all nodes that it points to are non-garbage. This garbage collector is different from the one described in this paper because it does not require global synchronization. But, because of the large number of internode packets, it not necessarily more efficient than a synchronized approach.

The most relevant previous collectors for active objects are due to Hudak, working with a functional language, and Halstead, using actor-based language.

Hudak [Hudak 1982] [Hudak 1983] presented algorithms for garbage collection of a distributed functional program. 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.

3. Local Marking Algorithms

The local marking algorithm determines which actors on a single node of a distributed system are garbage and which are not. The algorithm is invoked whenever a node detects the need for additional free memory. The algorithm is extended in later sections to run concurrently with the mutator and the global garbage collector.

Figure 3 shows a node with four actors. The rectangle surrounding the actors represents the node on which they reside. Actors B and D have remote acquaintances while actors A and C are themselves acquaintances of actors on other nodes. Actors such as A and C are said to have remote inverse acquaintances.

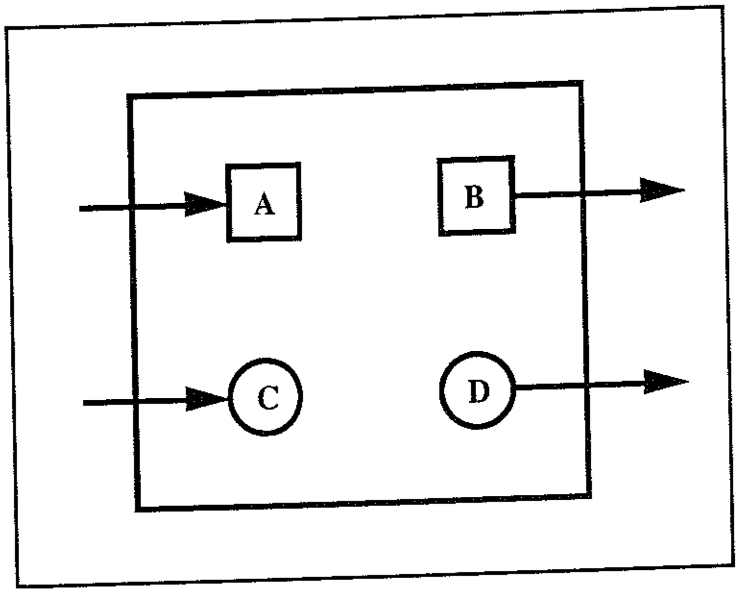

Figure 3. Examples of Remote Acquaintances

This example illustrates that the local collector usually lacks sufficient information to determine whether or not to garbage collect an actor with remote connections. For example, if a black actors on other nodes have actors A and C as acquaintances, then A and C are non-garbage. If actor D has a black acquaintance then it too is non-garbage. Actor B however, is garbage regardless of its remote acquaintance. Therefore, when performing local marking with remote acquaintances, it must be assumed that any remote acquaintances or inverse acquaintances are black.

The local collector cooperates with the global collector by partitioning the actors into two sets: those which are affected by remote acquaintances and those which are not. When the global collector performs its markings, it need only consider those actors which are affected by remote acquaintances. The advantages of this approach is that the global marker examines a smaller number of actors and does not duplicate colorings already made by the local marker.

The first step in the cooperative local marker colors the actors using the Push-Pull algorithm presented in the preceding section. The second step determines global dependencies using the algorithm shown in Figure 4. This algorithm is an adaptation of the "push-pull" algorithm which uses three sets "R" (remote), "P" (possibly remote) and "L" (local). "R" is analogous to "black", "P" is analogous to "gray" and "L" is analogous to white. All actors marked as "R" are needed by the global collector for it to correctly perform its marking. Black actors are handled differently than other actors because there is no purpose in propagating the remote dependency property to a black actor's acquaintances since they must already be colored black (i.e., they are known not to be garbage). At the termination of the algorithm all white or gray actors not members of the "R" set are garbage.

Figure 5 shows a configuration before and after this algorithm is applied. Actor J is in the "L" set because it is not affected by any remote acquaintances. Actor F is in the "P" set because, although it has a remote acquaintance, it cannot become active. The local collectors may reclaim actors J and F. Actors D, E, G, H, I are in the "R" set because they are dependent upon remote connections. Actors B and C are in the "R" set because their colors are needed by the global marker.

```
Initial Conditions:

        The "P" set contains all non-black blocked actors with a remote acquaintance.
        The "R" set contains all actors with remote connections not already in P.
        The "L" set contains all remaining actors


Puller:
        BEGIN
                FOR [each non-black actor in the "R" set]
                        place "L" and "P" acquaintances of the actors in the "R" set
                END FOR
                -> Pusher
        END

Pusher:
        BEGIN
                FOR [all actors member of the "L" set]
                        CASE: actor is active and acquaintance is "R" or "P"
                                        -> place actor in "R" set

                        CASE: actor is blocked and acquaintance is "R" or "P"
                                        -> place actor in "P" set
                END FOR
                IF [any actors were placed in the "R" or "P" set] THEN -> Puller
        END

Termination:

        All white or gray actors are garbage unless in the "R" set

Note:  All references to acquaintances in the pusher and puller refer to local acquaintances
only.
```

Figure 4. Propagation of Remote Dependencies

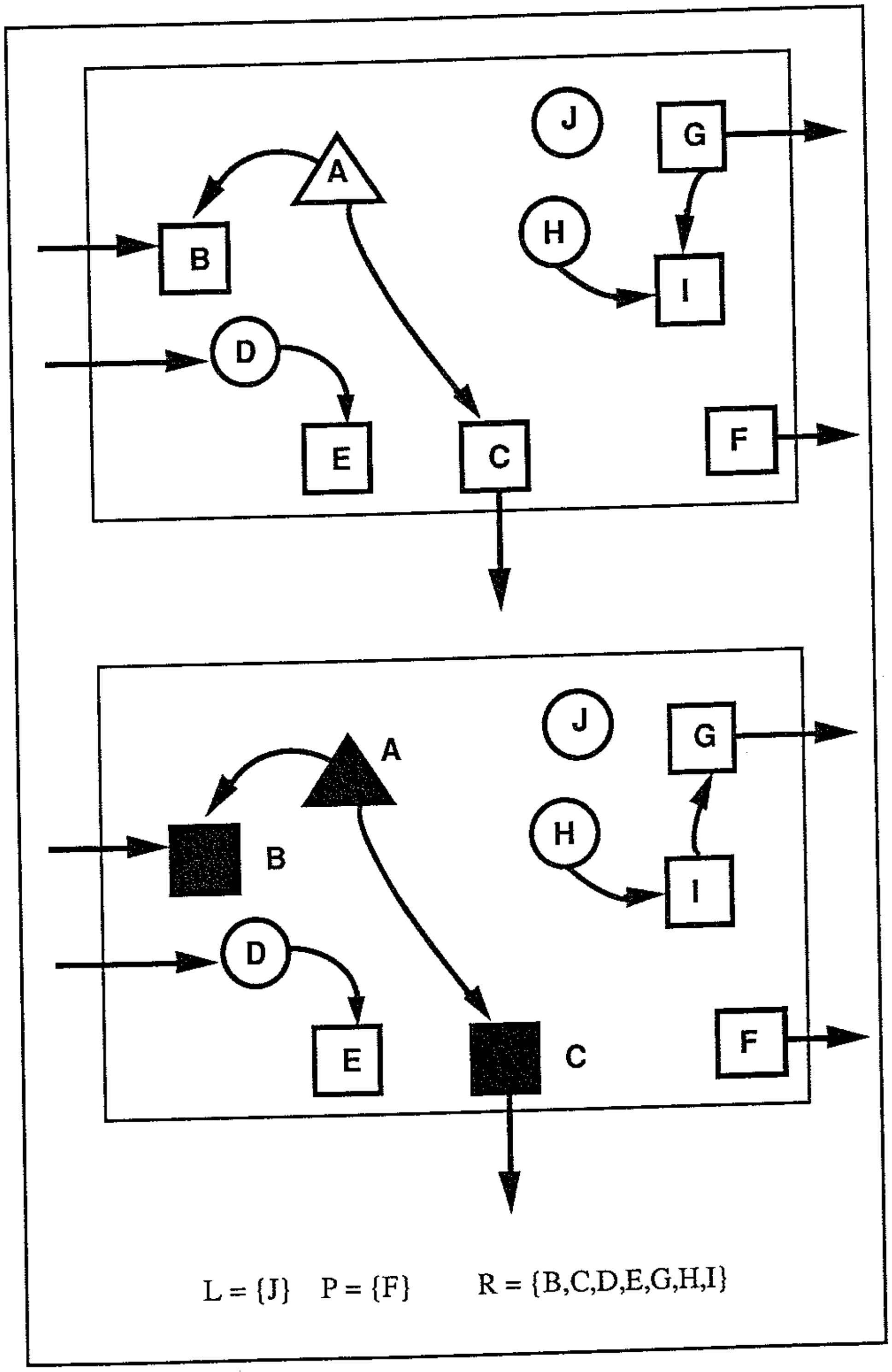

Figure 5. Example of Cooperative Local Marker Algorithm

4. Global Garbage Collection

Global garbage collection is performed by three entities: director, agent and global marker. Figure 6 shows the communication paths among the various tasks. Tasks in a solid block execute on the same node. Tasks separated by arrows communicate via asynchronous message passing.

The first task, the director, runs on only one node at a time and coordinates the efforts of the many global markers. The second task, the global agent, also runs on each node in the distributed system and handles the remote communication for the global markers. The third task, the global marker, runs on each node in the distributed system, and performs the marking of the actors local to the node on which it runs.

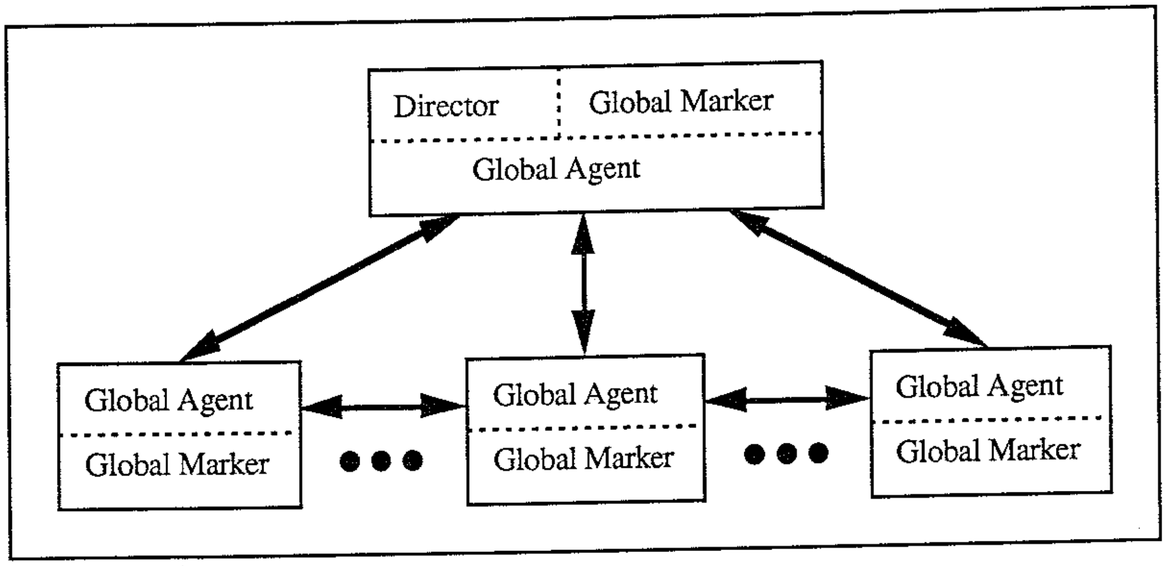

Figure 6. Relationship of Garbage Collection Components

Global Director

The global director initiates and terminates the following three phase of the garbage collection:

- Initialization: initialize the data structures used to perform garbage collection. At the end of this phase, each node is prepared to cooperate with other nodes in the task of marking actors. All work done by nodes during this phase is intra-node.
- Marking: each node marks actors using both local information and information garnered by exchanging messages with other nodes. Global marking uses the local collectors last set of complete markings. When the marking phase is complete, all globally dependent actors are marked as garbage or non-garbage.

- Reclamation: nodes collect garbage actors and return the freed memory to the free memory pool. When the reclamation phase completes, garbage collection is done.

Phase transitions are controlled by passing messages between the director and the global agents.

The goals of the global director are:

- Maximize concurrency within each phase
- Evenly distribute work when possible
- Minimize the number of synchronization messages

Concurrency within a phase is maximized by enabling as many nodes as possible to work simultaneously. Work can be distributed evenly by using decentralized algorithms. Minimizing the number of synchronization message is important because it reduces network traffic.

Recognizing the termination of the initialization phase and the reclamation phase is straightforward. Since these two phases work entirely with local information, only one message is required per node to indicate that it has finished one of these phases.

The termination of the marking phase is more difficult to recognize. Because there is cooperation between nodes, it is possible that a "finished" node may become "unfinished". Figure 7 shows a configuration of two nodes and three actors. If node 1 performs its coloring before node 2, then actor A is marked as garbage and node 1 considers itself finished. In this example node 1 must redo its marking after node 2 completes the coloring of actor C. The marking phase terminates when all global markers are finished "at the same time." When this is true, all markers are guaranteed to remain finished.

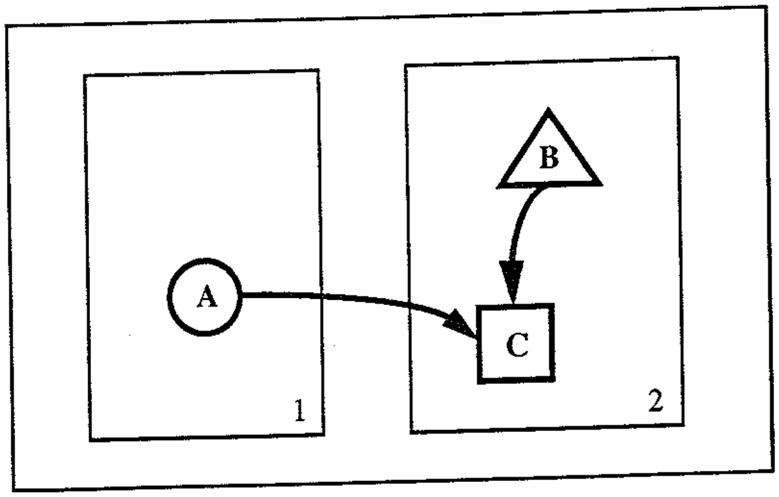

Figure 7. Example of a Node Becoming "Unfinished"

Phase transitions are controlled by passing packets between nodes. The packets, shown in Table 2, are divided into two categories: tokens and messages.A token is a packet which indicates ownership. For this reason, only one copy of the token is in existence at any instant. Therefore, tokens are transmitted from one node to another node; broadcast transmission of tokens is not permitted. An ordering is used to transmit tokens by assigning each node a unique number. A logical ring is defined by stipulating that nodes may only transmit to the next highest numbered node. The highest numbered node may only send to the lowest numbered node. A message, on the other hand, is a packet that may be broadcast to all other nodes or to any other single node.

Table 2. Synchronization Packets

Packet	**Sender/Receiver**	**Description**
	Initialization	
Start_initialization	Director-> Agent	Agent and marker being initialization
Done_initialization	Agent -> Director	Agent and marker have finished initialization
	Marking	
Being_Marking	Director -> Agent	Agents tell their markers to begin marking.
Done_Marking	Agent -> Agent	Marking is complete for all agents between the director and this agent (a field in the token contains the identity of the current director). If received by the agent of the current director, the director is notified. Note: This is the only token, all other packets are messages.
	Reclamation	
Begin_Reclamation	Director -> Agent	Agents will tell the markers to begin reclaiming garbage actors.

The first time the director is started, it executes on an arbitrary node. On subsequent invocations, the last node to be the director begins the next cycle as director. There are several methods to determine when the director should start. One method is for the director to maintain a timer controlling the interval between global collections. A second method is for agents to forward requests to the director when they want to start a collection. In this case, the director could start the collection when:

- A single request is received or,
- A majority of requests is received or,
- Requests from all nodes are received.

For the purpose of discussion, any of these methods is sufficient. The method used in an implementation depends upon the particular environment in which the garbage collection takes place.

The director starts the garbage collection process by sending a begin_initialization message to all global agents. This message causes the receiving global agent to initialize itself and its companion global marker. For a time, all global agents work on their initialization in parallel. When a global agent completes its initialization, it sends a done_initialization message to the director. When all global agents have responded with this message, the initialization phase is complete.

The director begins the marking phase by sending a begin_marking message to all global agents. When the director's node completes its marking, it forwards a done _marking token to the next agent. When the agent receives the token, it holds it until its marker finishes. At this point, the agent checks to see if it has given work to any agent between itself and the director inclusive. If it has, it assumes that these agents may have become "undone" after having been finished. The agent then claims the title of director (changes a field in the token), clears its "work given" list, and forwards the done_marking token. It does not need to notify the former director that is no longer the director because the done_marking token always contains the identity of the current director.

When the current director's agent receives the done_marking token, the director is notified that all nodes have completed their marking phase. The director now sends a begin_reclamation message to all agents. When an agent receives this message, it begins reclamation. All nodes may perform their reclamation in parallel. A done_reclamation message is not necessary because the end of the reclamation phase is detectable implicitly. If the director starts another global garbage collection session before the reclamation is complete on a node, the node's agent refrains from sending its done_initialization message until reclamation is complete.

```
Begin Initialization_Phase
        send begin_initialization message to all agents
        wait for response from all agents
        send begin_marking message to all agents
End Initialization_Phase

Begin Marking_Phase
        wait for director's agent to announce completion of marking
        tell agent to forward done_token
End Marking_Phase

Begin Reclamation_Phase
        send begin_reclamation message to all agents
        wait for response from all agents
End  Reclamation_Phase

Begin Director
        Initialization_Phase
        Marking_Phase
        Start reclamation  phase if notified by agent
End Director
```

Notes:
1.The reclamation phase may or may not be started by this node. It depends upon the order in which global markers complete.

2.The phrase "wait for response from all agents" includes this node's agent.

Figure 8. Algorithm for Director

Global Agent

A global agent is the intermediary between global markers and the other garbage collection entities. It handles communications that synchronize phases and colors actors. It provides services to both the global marker and other global agents. Figure 9 shows the algorithm for the global agent. Figure 10 shows the algorithm for servicing the work queue.

```
BEGIN agent
    wait for begin_initialization message
    work_handed_off = { }
    execute global_marker_initialization
    send initialization_done message

    execute global_marker
    DO
        update = service_work_queue
        IF [update] THEN global_marker
        IF [done_marking message received] THEN
            IF [done_marking message == "this node director"]  THEN
                tell director to announce reclamation phase
            ELSE
                IF [work given to agent between node and director inclusive]
                    message done_marking = "this node is director"
                    END IF
            END IF
            work_handed_off = { }
            forward done_marking message to next agent
        END IF
    UNTIL begin_reclamation message received

    do global_marker_reclamation

END agent
```

Figure 9. Algorithm for Global Agent

```
BEGIN color_actor (actor, color)
    IF [actor is local] THEN
        place actor in color set
        IF [actor darkened] THEN update = true
    ELSE
        send msg remote_color (actor, color) to remote node
    END IF
    FOR [all nodes of the actor's remote inverse acquaintances]
        send msg color_change (actor, color, inverse_acquaintance) to nodes
    END FOR
    RETURN update
END color_actor

BEGIN service_work_queue
    update = false
    WHILE [queue  not empty]
        CASE: remote_color (actor, color)
                        update = color_actor (actor, color) OR update

        CASE: color_change
                        update = true

        CASE: inquire (remote_actor, remote_state, local_actor)
            IF (remote_state == active) AND (local_actor black or gray)
                remote_color (remote_actor, black)
            IF (remote_state == blocked) AND (local_actor black or gray)
                remote_color (remote_actor, black)
    END WHILE
    RETURN update
END service_work_queue
```

Figure 10. Algorithms for Support Routines for Global Agent

The global agent first waits for a begin_initialization message from the director. When this message is received, the global marker is initialized and the set of nodes to which work has been given is set to null.Next, the agent calls the global marker to perform the first pass of the marking of the actors. When the coloring is complete, the agent services its work queue. The work queue contains coloring commands that were received from other nodes. If servicing the work queue caused any actors to be darkened, then the global marker is again repeated. This cycle continues until a marking_done message is received. If this node is the director, then the director is informed that all marking is complete. The director then notifies the agent to begin the reclamation phase. If this node is not the director, then the agent must check whether it has given any work to nodes between it and the director. If so, it claims the role of director. In either case, the list of nodes to which work was given is purged. The marking_done message, which contains the identity of the current director is then forwarded.The agent eventually receives a begin_reclamation message, at which time all marking is complete and reclamation of garbage actors may begin.Figure 11 shows the services that the global agent provides to the global marker to assist it in assessing and coloring remote nodes.

```
Event Receive Remote_Color  (actor, color) message
        BEGIN
                place request in work queue of the global marker
        END

Event Receive Inquire (local_actor, actor_state, remote_actor) message
        BEGIN
                place request in work queue of the global marker
        END

BEGIN transmit_ remote_color (actor, color)
        work_handed_off = add remote node to list
        transmit message to the specified agent
END

BEGIN transmit_inquire (actor_id,  actor_state, remote_actor)
        work_handed_off = add remote node to list
        transmit  message to the specified agent
END
```

Figure 11. Services Provided by the Global Agent

Global Marker

The global agent calls the global marker to color the actors which are dependent on global information. The global marker may be implemented by making three modifications to the push-pull algorithm. The first modification is to the pusher part of the algorithm. If a remote actor is colored black, then the agent service color_actor is used. Thus, the actual coloring is performed by the remote node. A second modification is required to access the color of a remote actor. Rather than sending a request and blocking until a response is received, the pusher instead uses the agent's inquire to send the information about the current actor to the remote node. While the remote node determines whether the actor should be colored, the puller continues. If the remote node determines the actor should be colored, it uses the color_actor service to tell the puller's agent to darken it. For example, in Figure 12 node 1 sends a message to node 2 to determine if actor A should be darkened. Node 1 continues its coloring, and eventually receives a message from node 2 telling it to darken actor A.The third modification is necessary to handle the darkening of actors which have remote inverse acquaintances. When the actor is darkened, it must notify the remote node to re-run its coloring algorithm. Figure 12 shows a situation in which the darkening of an actor causes other actors on a remote node to be non-garbage. If node 1 completes its coloring before node 2 darkens actor C, then node 2 must notify node 1 to re-run its coloring algorithm.

Figure 13 shows the algorithm for the global marker. The routine color_actor is described in Figure 10.

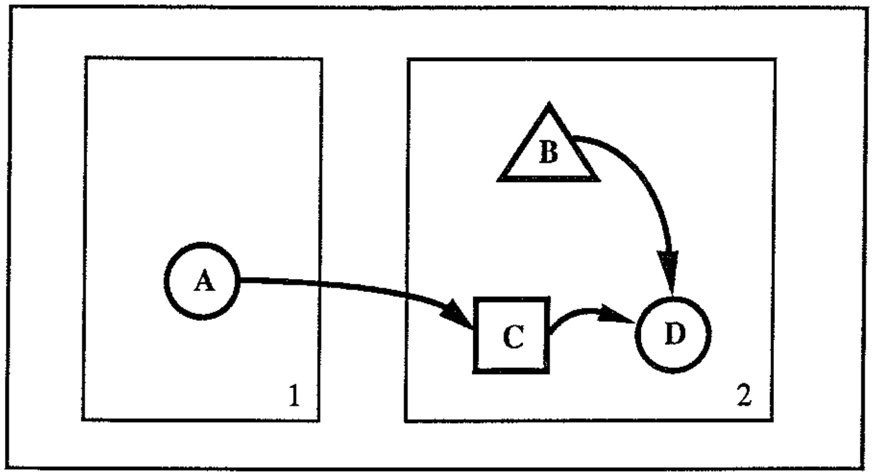

Figure 12. Global Marking and Inverse Remote Acquaintances

```
BEGIN Initialization
        All black globally dependent actors are placed in the black set
        All other globally dependent actors are placed in the white set
END Initialization

BEGIN Puller
        FOR [each actor in the black set]
                color_actor (non-black_acquaintance, black)
        END FOR
        ->Pusher
END Puller

BEGIN Pusher
        FOR [each actor in the white set]
                CASE: actor is active and acquaintances is remote
                                        -> inquire (actor, "active", acquaintance)
                CASE: actor is active and an acquaintance is black or gray
                                        -> color_actor (actor, "black")
                CASE: actor is active and acquaintances is remote
                                        -> inquire (actor, "blocked", acquaintance)
                CASE: actor is blocked and an acquaintance is black or gray
                                        -> color_actor (actor, "gray")
        END FOR
        IF [any actors were placed in the black or gray set] THEN -> Puller
END Pusher

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

Figure 13. Algorithm for Global Push-Puller

5. Mutator and Collector Concurrency

This section describes the synchronization that is needed to allow the mutator, the local collector and the global collector to run concurrently without interference. Since the global collector and the mutator do not directly interact, synchronization is required between the local and global collectors, described first below, and also between the mutator and the local collector, described next.

Local and Global Garbage Collector Synchronization

We have already seen in Section 3 that the cooperative local collector provides the local information needed by the global collector. Beyond this cooperation, synchronization is needed to prevent interference between the two collectors in two cases:

- concurrent deletion of actors: neither collector should delete actors that are currently being used by the other collector.
- concurrent coloring of actors: the colorings by one collector must not interfere with the colorings done by the other collector.

Concurrent deletion is synchronized through the data structure gc_info, shown in Figure 14. A copy of this data structure is associated with each actor. When the global collector deletes an actor, it sets the delete_request bit in the actor's gc_info field. The actual deletion of the actor is done by the local collector during its next reclamation cycle. By design, the cooperative local collector cannot delete actors that are relevant to the global collector.

Concurrent coloring uses the other fields in the gc_info data structure. When either the local or global collector starts its marking phase, it uses gc_side to choose the side of gc_info on which to operate.

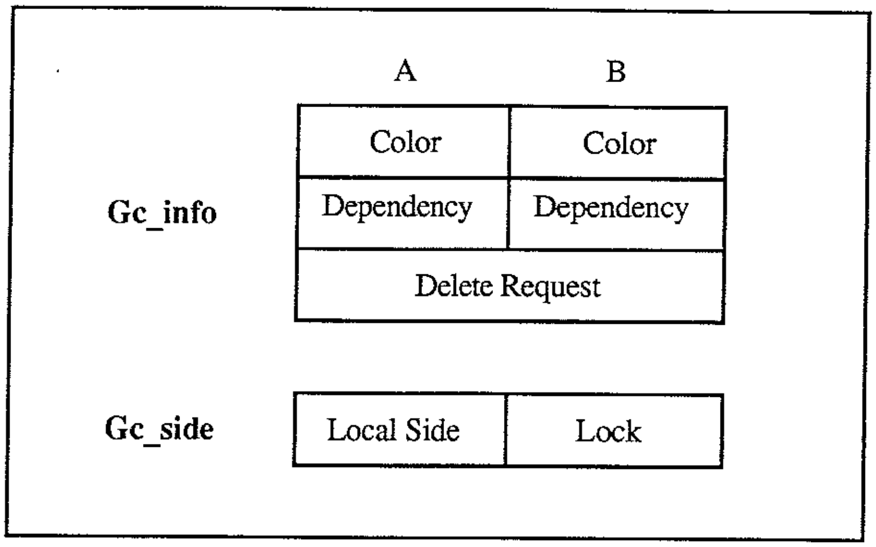

Figure 14. Data Structures for Local and Global Collector Synchronization

When the global collector starts, it locks local_side. The side indicated by the complement of the local_side is used by the global collector. When the global collector is finished, it unlocks local_side. When the local collector runs, its uses the fields indicated by local_side. When it finishes, it does an atomic "test and complement" of local_side. If the lock is not set (i.e. the global collector is not running), local_side is complemented. Thus, the next time the global collector runs, it uses the latest information from the local collector. But, if the lock was set, the value of local_side remains unchanged. The next time the local collector runs, it operates on the same side, thus overwriting its old data and leaving the global collector's markings undisturbed. This algorithm is summarized in Figure 15.

```
BEGIN Local
    use side indicated by local_side
    mark actors
    DO_ATOMICALLY
        IF [gc_side unlocked]
        THEN complement local_side
    END_DO_ATOMICALLY
END Local

BEGIN Global
    lock gc_side
    use side indicated by the complement of local_side
    mark actors
    unlock gc_side
END Global
```

Figure 15. Locking Mechanisms

The above algorithm makes two assumptions:

- a local collection is performed before the first global collection, and
- at least one update is done by the local collector between passes of the global garbage collection.

The first assumption insures that the global garbage collector has a set of markings from which to work. The second assumption precludes the case of the global collector working on very old data. In the worst case, the global collector is running every time the local collector attempts to complement local_side. An alternative to the second assumption is to add a third side to the garbage collection structure. This enables the local collector to write to a new copy rather than over-writing an old copy. This guarantees the local collector gives the global collector a relatively fresh copy of actor markings. One method of guaranteeing the two assumptions is to maintain a count of the number of local collections and global collections completed (local_pass_count for the local collector and global_pass_count for the global collector). The first assumption is met by starting a global collection only when local_pass_count is greater than zero. The second assumption is met by the global collector saving local_pass_count before it starts a collection. A subsequent global collection can start when the current local_pass_count exceeds the saved local_pass_count.

Mutator and Collector Concurrency

Up to this point, it has been assumed that the mutator and garbage collectors do not run concurrently, i.e. that the mutator is halted during garbage collections. This assumption is not acceptable and will now be removed.

To guarantee that the garbage collector finishes in a finite amount of time, the collector must ignore actors or acquaintances created after the beginning of the collection. As shown in Figure 16, consistency requires that the collector also ignore deletion of acquaintances that occur after the beginning of the collection. The top row of Figure 16 shows the addition the an acquaintance (B to C) and the deletion of an acquaintance (A to C) over a period of time. At no instance in the transformation is actor C garbage. The second row of the figure shows the actor graph as it appears to a garbage collector that processes new deletions, but not new additions. This causes actor C to be incorrectly identified as garbage. Therefore, since changes by the mutator cannot be observed during a garbage collection, the garbage collector must work on a system that appears to be "frozen", i.e. in a consistent state.

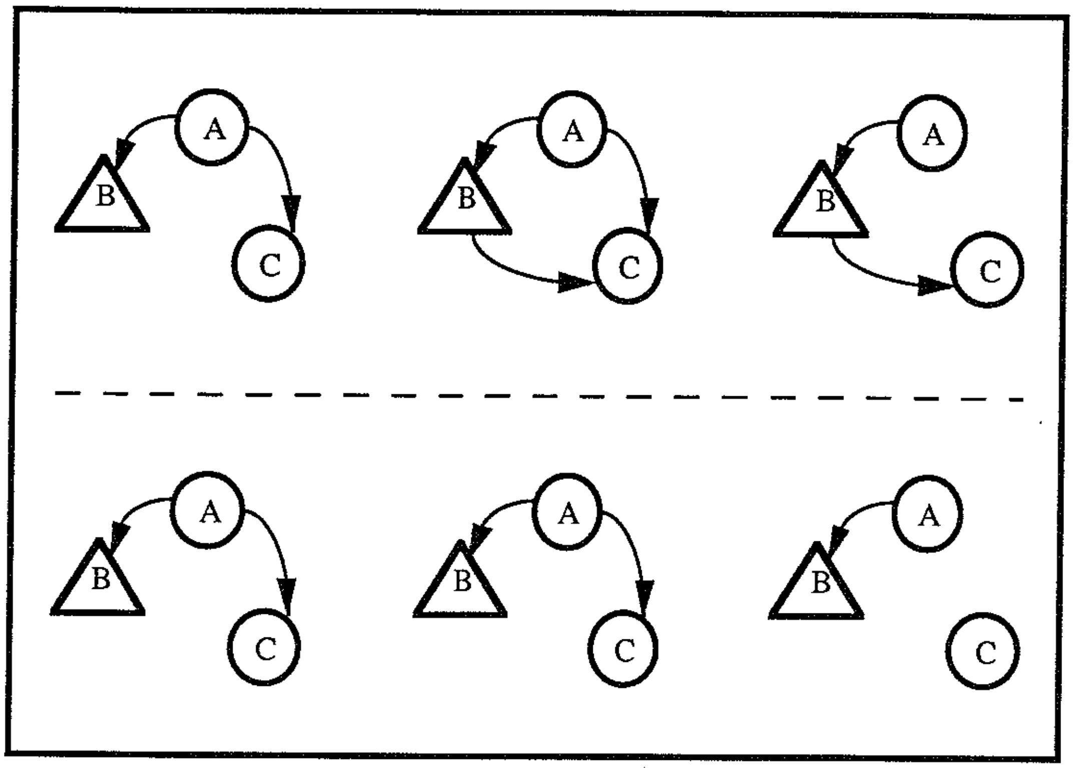

Figure 16. Additions and Deletions

A consistent state can be achieved without halting the mutator by using time stamps, called epoch numbers. In this approach the system maintains a current epoch number ("time") which is incremented at each invocation of the garbage collector. Each actor and each acquaintance carries an epoch number corresponding to the "time" of its creation. During the marking, the collector processes only those actors and acquaintances where "object's epoch < current epoch". This rule insure that the collector operates on a consistent system state.

To guarantee a consistent state, the local collector uses a local epoch number and the global collector uses a global epoch number. Each local collector is responsible for updating its own epoch number, which is done by incrementing it at the start of initialization. The director, however, is responsible for updating each of the global epoch numbers. This is done by including the new global epoch number in the initialization message sent to each global agent.

Because the new global epoch number is distributed through messages to the individual nodes, there can be no guarantee that all nodes will always have the same global epoch number. Therefore, it is possible for an acquaintance to be created or deleted between two nodes with different epoch numbers. It is now shown how such an acquaintance is recognized by both or neither of the nodes.

Figure 17 shows two nodes, A and B, in which the global epoch numbers have not yet been updated on both nodes. Suppose that an acquaintance from an actor on node A is created to an actor on node B at the indicated time. When node A sends the create_remote_acquaintance message to node B, it includes its current global epoch number. When node B processes the message, it tags the acquaintance with the global epoch number which A sent, 11.

When node A's global garbage collector runs, it processes all acquaintances created before epoch 11. Thus node A does not process the new acquaintance since it is tagged with epoch 11. Node B's global garbage collector also does not process the acquaintance because it is tagged with an epoch number equal to the current epoch number.

Now suppose that the acquaintance is created in the opposite direction, from the lower epoch to the higher epoch, i.e. from node B to node A. Node B's global garbage collector processes the acquaintance because it runs at epoch 11 and the acquaintance is tagged with epoch 10. Node A's global garbage collector also processes the acquaintance because it runs at epoch 11 and the acquaintance is tagged with epoch 10. Even though node A has already updated its epoch number in the second case, it could not have started its global collection. This is because node B has not received nor acknowledged the new global epoch number, and the director does not notify the nodes to begin marking until all nodes have acknowledged the new epoch number. A similar argument shows that the deletion of inter-node acquaintances are also correctly processed.

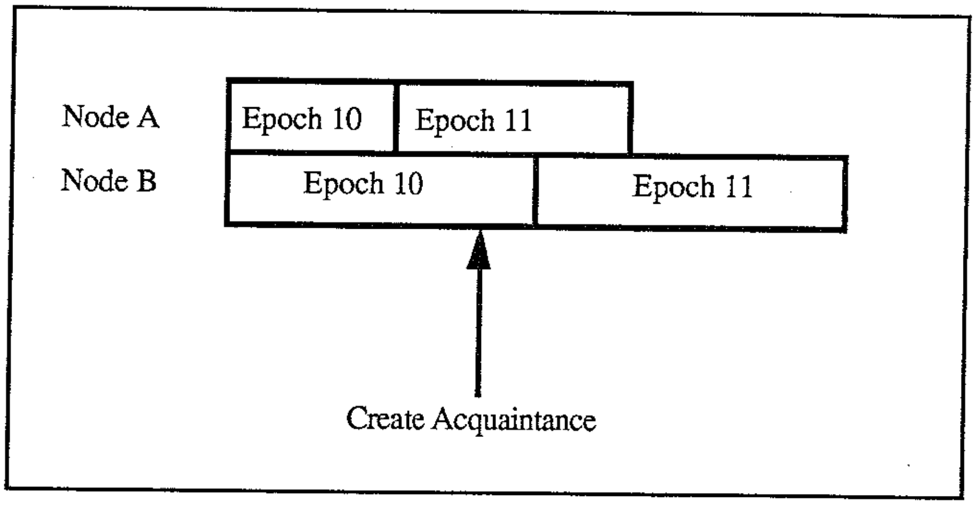

Figure 17. Global Epoch Numbers and Creation of Remote Acquaintances

5. Conclusions

In this paper we have shown how to automatically reclaim active objects in a distributed computing system. The proposed architecture permits:

- autonomous local garbage collection without global synchronization
- local and global collectors to operate concurrently
- local and global collectors to operate concurrently with the mutator

The algorithms used by the local and global collectors are designed to:

- collect actors that are part of a local or distributed cyclic structure
- synchronize the local and global collectors
- control the termination of the global collector
- achieve consistency among the mutator, local collectors and global collector
- eliminate redundant work (coloring) between the local and global collectors.

The collector presented in this paper is one of very few to address the problem of reclaiming active objects and also one of the few to consider reclamation in a distributed computing system.

There are several limitations of the collector given in this paper. These limitations - and the future work needed to remove them - are discussed below.

The performance of algorithms for collecting garbage data have improved because the characterization of the garbage in these systems has improved. An example of such an improvement are generational garbage collectors. The same improved characterization is needed for the actor model. Currently there is almost no knowledge of the characteristics of actor garbage in either the local or global environment. Some questions to investigate are:

- What percentage of local garbage is cyclic?
- What percentage of actors in a system are globally dependent?
- How do the lifetimes of actors vary?
- Do the lifetimes of actors suggest a generational method?

The performance of the garbage collector described in this paper is not well understood. Two methods of evaluating the performance of the garbage collector are simulation or empirical study. Some measures of the garbage collector's performance to investigate are:

- the amount of time needed to do a local garbage collection
- the amount of time needed to do a global garbage collection
- the frequency of local and global garbage collections
- the affect of garbage collection on the performance of the mutator

In distributed systems, it is likely that some nodes may be unavailable for large periods of time. When this occurs, it is important that garbage collection continue, albeit at a reduced level. In the garbage collection scheme presented, this unavailability does not affect local collection, but it does prevent global collection from occurring. A direction of future work is to investigate how to achieve fault tolerant garbage collection.

The garbage collector presented in this paper supports the basic actor model. An extension to the actor model, ACT++[Kafura and Lee 1990], includes a special mail queue known as a Cbox. A Cbox is a programming convenience that simplifies actor programming. It is used to receive the result of work given to another actor. Attempts to read from a Cbox before the delivery of the result causes the reader to block. In order to support the ACT++ language, the presented garbage collectors must be modified to incorporate Cboxes.

Finally, none of the algorithms presented in this paper have been proven to be correct. This is an area which needs further investigation.

References

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

[Appel 1988] Appel, Andrew W. and Hanson, David R., "Copying Garbage Collection in the Presence of Ambigous References," Princeton University, Department of Computer Science, Technical Report CS-TR-162-88, June 1988.

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

[Cheriton 1988] Cheriton, D., "The V Distributed System," Communications of the ACM, March 1988.

[Dasgupta 1988] Dasgupta, P. LeBlanc, R.J. and Appelbe, W.F., "The Clouds Distributed Operating System: A functional description, related work and implementation details," Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, June 1988.

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

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

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

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

[Kafura 1988] Kafura, Dennis G., "Concurrent Object Oriented Real-Time Systems Research", Virginia Polytechnic Institute and State University, Department of Computer Science, Technical Report 88-47, September, 1988.

[Kafura 1990] Kafura, Dennis and Washabaugh, Doug, "Garbage Collection of Actors," Proceedings: Joint OOPSLA/ECOOP'90 Conference, Ottawa, Canada, October, 1990.

[Leddy 1989] Leddy, W.J. and Smith, K.S., "The Design of the Experimental Systems Kernel," Proceedings: Hypercube and Concurrent Computer Applications, 1989, Monterey, CA.

[Nelson 1989] Nelson, Jeff Automatic, Incremental, On-the-fly Garbage Collection of Actors, M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, 24061-0106, February 1989.

[Schelvis 1989] Schelvis, Marcel, "Incremental Distribution of Timestamp Packests: A New Approach to Distributed Garbage Collection," ACM SIGPLAN Notices, Volume 24, #10, Oct., 1989.

[Washabaugh 1990a] Doug Washabaugh, Real-Time Garbage Colection of Actors in a Distributed Systems, M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, 24061-0106, February 1990.

[Washabaugh 1990b] Doug Washabaugh and Dennis Kafura, "Real-Time Garbage Collection of Actors", Virginia Polytechnic Institute and State University, Department of Computer Science, Technical Report 90-24, April, 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.