Concurrent and Distributed Garbage Collection of Active Objects
Dennis Kafura, Member, IEEE, Manibrata Mukherji, and Douglas M. Washabaugh

Abstract—This paper shows how to perform concurrent and distributed automatic garbage collection of objects possessing their own thread of control. The relevance of garbage collection and active objects to distributed applications is briefly discussed and the specific model of active objects used in the paper is explained. The collector is comprised of independent local collectors, one per node, and a distributed global collector. The mutator (application), the local collectors and the global collector run concurrently. An important part of this paper is the detailed presentation of the algorithms necessary to achieve correct concurrent operation among the collectors and between the collectors and the mutator. The collector builds on previous algorithms for taking snapshots in distributed systems and for detecting termination.

I. INTRODUCTION

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: objects encapsulating their own threads of control. We explain below why active objects are useful in general and why they are particularly useful in distributed programming. While the actor model [1] is used as the framework for presenting the algorithms in this paper, the basic ideas apply to any model based on asynchronous message passing among active entities. The active entities may be heavyweight processes or lightweight threads encapsulated within an object. The basic problem of garbage collecting actors is developed in [17] and [14]. A real-time local collector is described in [23].

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

- the complexity of storage management in a distributed environment implies that it must be an integral part of the underlying automatic resource control system [2].
- encapsulating a thread of control within each object creates a uniform object model which combines the distinct notions of object mobility and process migration.
- autonomous, interacting real-world entities are more easily and more directly expressed in software as active objects [15].

This paper is also motivated by the relatively few garbage collectors that manage distributed and active objects.

The main contribution of this paper is the design of the global collector for distributed active objects based on the architecture shown in Fig. 1. The mutator at each node consists of active objects that are stored in that node’s local memory. A node’s local collector takes an instantaneous “snapshot” of the actors stored in that node’s memory and uses a coloring algorithm to determine which actors may be reclaimed. The local collector 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 on each node is comprised of three tasks whose names and roles are:

- **Initializer**: This task captures a globally coherent “snapshot” of the local actors. Capturing this snapshot involves cooperation among the initializer tasks on different nodes and it requires the initializer to monitor, for a limited period, the messages being received by its local mutator.
- **Global Marker**: This task colors the snapshot using an extended version of the local collector’s coloring algorithm. The extended coloring algorithm requires interactions with the global agents on other nodes.
- **Global Agent**: This task receives messages from global markers on other nodes, reactivates its local global marker task as required, and cooperates with other global agents to detect the termination of the global markers.

The use of snapshots is important in this organization as they allow the mutator, the local collector and the global collector to execute concurrently.

The remainder of this paper is organized as follows. Section II describes the model of active objects, the actor model, that is used in this paper. Garbage actors are defined and illustrated by example. This section also contains a review of related collectors. Section III describes the local collector. The
global collector, including the global collector's termination algorithm, is presented in Section IV. Conclusions are given in Section V.

II. GARBAGE COLLECTION IN A DISTRIBUTED ACTOR SYSTEM

This section presents those aspects of a distributed system and the actor model that are relevant to the problem of garbage collecting actors in a distributed actor system. In the first subsection, the key terms used in the paper are defined and a simple example is given to illustrate these definitions. In the second subsection a comparison with related garbage collectors is given.

A. Definitions

A distributed processing system consists of basic elements for computing (nodes) and communication (channels). These terms are defined as follows:

Definition: A node is a reliable, autonomous computing device containing its own local memory and providing a runtime environment for the execution of user programs.

The nodes in the distributed system are indexed 0...N-1. The node with index 0 initiates the global garbage collection and recognizes the termination of the global markers. The fixed role of node 0 is not an inherent limitation and is used only for simplicity. A more dynamic scheme could be used at the expense of additional complexity.

The communication among nodes is enabled by message-bearing channels defined as:

Definition: A channel is a directed communication link connecting two nodes over which messages may be sent with the guarantee of their delivery in sequence and free from errors, loss or duplication. The channel is directed in the sense that each channel appears to one of its nodes as an incoming channel and to the other node as an outgoing channel.

A pair of oppositely directed channels is assumed to exist between any two nodes; this allows bidirectional communication between any two nodes. The channels are logical communication links that can be implemented in a variety of ways.

The basic element of the actor model is, of course, an actor that is defined as follows:

Definition: An actor is a uniquely named, autonomous, message-driven agent that performs local computation, communicating asynchronously with other actors, and creates new actors.

An actor has a single mail queue to buffer messages that have been received but not yet processed. Message passing is reliable but unordered. To send a message, the sending actor must know the name of the receiving actor. The relationship between communicating actors is described in the following terms:

Definition: For two actors, A and B, A has B as an acquaintance if and only if A knows the name of B. If A has B as an acquaintance, A is the inverse acquaintance of B.

The acquaintances of an actor define the set of other actors to which messages can be sent. The inverse acquaintances of an actor define from which other actors messages may be received.

Since an actor is message-driven, it may be in one of two processing states depending on the availability of a message in the actor's message queue.

Definition: The processing state of an actor is either active or blocked. An active actor is an actor that is currently performing an internal computation in response to a message. A blocked actor is an actor that is currently awaiting the arrival of a message to its empty mailbox.

As part of its internal computation, an active actor may send mail messages asynchronously to its acquaintances and create new actors. A blocked actor is incapable of any action. The sole reason for an actor to become blocked is the absence of a mail message and only the arrival of a mail message will cause a blocked actor to become active.

A complete application consists of several, perhaps numerous, individual actors programmed to achieve some overall purpose. From the system's perspective, an actor system is defined as:

Definition: An actor system is a collection of actors that are managed by the system as the members of a single group. This definition implies that garbage collection is performed in the context of the entire application.

One way in which the inherent concurrency in an actor system can be realized is by means of the physical parallelism inherent in the nodes in a distributed processing system. This yields the notion of:

Definition: A distributed actor system is an actor system whose actors have been assigned to the nodes of a distributed processing system.

In a distributed actor system, each node typically stores several, and possibly many, actors. The logical communication paths between actors, represented by the acquaintance relation, are mapped to channels. Several different acquaintance relations may be mapped onto the same channel. These mapping of actors to nodes and acquaintances to channels imply that the node's run-time system contains functions for actor scheduling and channel multiplexing.

To reclaim garbage actors in a distributed actor system requires information about the condition of each actor. The information about each actor collectively defines the condition of the distributed system itself. This information is captured in a snapshot of the system. These notions are defined as follows:

Definition: The state of a distributed actor system is a representation of the processing state and set of acquaintances for each actor in the distributed actor system.
**Definition:** A **snapshot** is an instantaneous recording of the state of the distributed actor system at a given point in time.

The snapshot of a two-node distributed actor system is shown in Fig. 2. The snapshot of an actor system can be represented by a graph whose nodes represent actors and whose directed arcs represent acquaintances. Active actors are drawn as circles; blocked actors are drawn as squares. We will see in Section IV how a snapshot is obtained.

The algorithms in this paper will use the following notation to represent information about each actor:

- \( \text{actor.state} = \text{active} | \text{blocked} \)
- \( \text{actor.node} = 0 \ldots N-1 \)
- \( \text{actor.acq} = \{ \text{set of actors} \} \)
- \( \text{actor.inverse} = \{ \text{set of actors} \} \)

where state is the actor’s processing state, \( N \) is the number of nodes in the distributed actor system, acq is the set of acquaintances, and inverse is the set of inverse acquaintances.

As an example of the notation, actor E in Fig. 2 would be described as:

- \( E.\text{state} = \text{active} \)
- \( E.\text{node} = 1 \)
- \( E.\text{acq} = \{ D, G \} \)
- \( E.\text{inverse} = \{ F \} \)

In Sections III and IV these additional fields will be added:

- \( \text{actor.color} = \text{black} | \text{gray} | \text{white} \)
- \( \text{actor.new} = \text{true} | \text{false} \)
- \( \text{actor.inquiry} = \text{true} | \text{false} \)

These latter fields will be described when they are needed and introduced here only for completeness. Descriptive phrases rather than more precise notation will be used whenever this promotes clarity. For example, the phrase “for all acquaintances of actor A” might be used instead of “for all \( x \), where \( x \) is an element of \( A.\text{acq} \).”

A distributed actor system changes from its current state to a new state as the result of one of a defined set of state transitions. The state transitions are:

**Definition:** A **state transition** occurs when exactly one of the following changes is made to the state of a distributed actor systems: the processing state of an actor is changed, an acquaintance is added or deleted, or a new actor is created.

These possibilities will be illustrated in reference to Fig. 2. First, sending a message from an active actor to an acquaintance that is blocked changes the processing state of that acquaintance to active. For example, in Fig. 2, if \( E \) sends a message to \( D \), \( D \) 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 Fig. 2, \( E \) could send the mail address of \( D \) to \( G \), causing a new acquaintance arc to be introduced from \( G \) to \( D \). Third, if actor \( I \) creates a new actor, say \( P \), on node 1, a new, blocked actor would be created on node 1 and a new acquaintance arc would be drawn from \( I \) to \( P \).

During the course of a computation, an actor system moves through a series of states, each state reached from the previous state by a single state transition. The garbage collector is not interested in a particular state but in all the states that might follow from the current state under any combination of state transitions. This motivates the notion of a reachable state.

**Definition:** A **reachable state** of a distributed actor system is any state that, from the current state, holds after a sequence of zero or more state transitions.

The set of all reachable states represents all possible future states of the computation. The garbage collector must explore these future states to determine the usefulness of each actor. Informally, useless (i.e., 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 Fig. 2, is introduced. These actors are considered to be the “roots” of the actor system and are defined as follows:

**Definition:** A **root actor** is an actor that is never garbage, it exists throughout the lifetime of the application, and it may spontaneously generate messages.

Intuitively, root actors are the means by which the actor computation affects the “outside world.” Root actors, for example, represent users, actuators or output ports. Root actors are always considered to be active.

Somewhat more precisely, then, a garbage actor can be defined as follows:

**Definition:** A **garbage actor** in a state of a distributed actor system is an actor that is not a root actor and, in all reachable states, cannot communicate with a root actor.

A more comprehensive definition of garbage actors is given in [17]. The definition of garbage actors shows that the property of being garbage is a “stable” property in the sense of [6]: if a stable property holds in a given state (i.e., the snapshot state) it holds in all states reachable from that state. Thus, any garbage actors found in the snapshot state can be safely reclaimed without regard for the current state of the actor system.

Let us now consider which actors in Fig. 2 are garbage. Clearly J and K are garbage because they form a cycle isolated from any root actor. Similarly, the individually isolated actors N and O are also garbage. Actors F, L, and M are also garbage because they are blocked and have no means of becoming
active in the future (each is not the acquaintance of any other actor). All other actors are not garbage. To see, for example, that actor G is not garbage, consider the following sequence of actions:

- A sends its own address to B;
- E sends the address of D to G;
- G sends a message, MSG, to D;
- D sends MSG to C;
- C sends MSG to B;
- B sends MSG to A.

Thus, information generated at G can reach the root A. Parts of this sequence also show why actors A–E are also not garbage.

The example in Fig. 2 motivates the need for both a local and a global collector as shown in Fig. 1. Notice in Fig. 2 that actors F, M, and O are known to be garbage and I is known not to be garbage based only on information available at node 1. Thus, a local collector can, in some cases, reclaim resources without interaction with other nodes. However, node 1 cannot correctly determine whether any of E, G, or J are garbage using only local information. Similarly, node 0 cannot determine whether D or K are garbage using only local information. A global collector is needed.

As illustrated above, when an acquaintance arc connects actors that are on different nodes the local collectors on each node may lack sufficient information to determine whether or not these actors are garbage. Such actors must be treated appropriately by the local collectors and are defined as follows:

**Definition:** An actor is remotely dependent if it has an acquaintance on another node or has an inverse acquaintance on another node.

Remotely dependent actors must be handled by the global collector.

Fig. 2 also shows that two criteria often used in garbage collection do not properly detect garbage actors. The first criterion is that all nongarbage objects can be discovered by a directed traversal starting at a root object. Consider actor I in Fig. 2; no traversal starting at a root will lead to actor I. However, actor I is not garbage as it is active and can send a message to a root actor. If a broader definition of traversal is used—one that ignored the direction on the arcs, for instance—then some garbage actors would not be recognized. This occurs with actor M in Fig. 2 that is garbage because it is blocked and cannot be activated by the arrival of a message as it is not known to any other actor. In general, traversal criterion are insufficient because the topology of the system alone is not sufficient to discriminate between garbage and nongarbage actors. The processing state (active, blocked) of an actor is also critical. The second criterion, reference counting, can also miss actors which are garbage. In Fig. 2, actors J and K, forming an isolated cycle of blocked actors, have nonzero reference counts even though both of them are garbage.

### B. Related Garbage Collectors

The related garbage collectors are reviewed in two categories: first, those that manage active objects; second, those that are distributed.

The most closely related work is by Puu [18] who builds on our own earlier results. The major advantages of Puu’s approach are the use of independent local collectors and, at the global level, the use of timestamps to avoid global synchronization. The major disadvantage of this work is that the global collector is centralized and, therefore, susceptible to congestion at the global collector’s node. In contrast, the collector in this paper distributes the work of the global collector among the nodes themselves.

Another closely related work, by Venkatasubramanian et al. [21], uses the actor model. Their collector takes a snapshot of the actor system by means of a protocol that requires the individual actors to send and receive messages. The advantage of this approach is that is more efficient if a highly parallel machine is used or if there are very few actors in a distributed environment. The disadvantage of their approach is when there are many actors in a distributed environment. In this case the extensive message passing required by their snapshot algorithm will block the mutator for a significant period of time. The approach used in this paper, where the snapshot is taken by a collector external to the actor system, will fare better when there are many actors per node. Another difference between their work and that presented in this paper is that we use a logical ring structure for detecting termination while they assume a grid structure for this purpose.

The actor model was also investigated by Dickman [7]. The major advantage of Dickman’s work is the improvements made in the space and/or time requirements of the collection algorithms. By using back-pointers, Dickman developed methods that have lower time complexity, but require more space, than the coloring algorithms used in previous papers. The disadvantage of Dickman’s approach is that it assumes an initial snapshot of the actor system is available and that termination is detected. This paper is more explicit about how the snapshot is taken and how the termination is detected.

Halstead’s garbage collector for distributed actors [10] 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. The major advantage of this work is that is was the earliest attempt to cope with the issue of reclaiming actors. The major disadvantages of this method is that it cannot detect cyclic garbage and it assumes a very limited (tree) topology. The algorithms presented in this paper collect cyclic garbage and allow an arbitrary topology.

Also related to the work presented in this paper are collectors that operate in a distributed environment. The most notable difference between the collector in this paper and other distributed collectors is the explicit concern for active objects. Most distributed collectors are based on reference counting or simple traversal, properties that do not correctly identify garbage active objects as was illustrated by the example in Fig. 2. In favor of these other distributed collectors is their greater efficiency, derived from the simpler rules (e.g., reference counting) on which they are based, or their ability to tolerate failures in the nodes or communication links. The collector in this paper is not fault tolerant.
The advantage of Schelvis collector is that it does not require global synchronization. However, because of the large number of internode packets, it is not necessarily more efficient than a synchronized approach.

III. LOCAL COLLECTION

The local collector may be initiated on a node based on any one of several criteria related to either the node’s memory, or processing resources, or both. With respect to memory, a node may initiate the local collector if the ratio of free to total memory falls below some limit or if the ratio of blocked actors to active actors exceeds some threshold. Such indicators might suggest that space is being consumed by garbage actors. With respect to its processing capacity, a node might initiate a local collection if its processor utilization exceeds some limit, suggesting that unnecessary computation is being performed by garbage actors. The criteria to initiate the local collector can be more elaborate than these simple rules and may involve a combination of memory and processor indicators. These examples are given only to suggest the breadth of possibilities.

The local collector uses three sets each named by a different color. 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 phrases “the color of an actor” and “the color of the set containing the actor” will be used interchangeably. The key step in the local collector is a marking, or coloring, of actors using the rules defined in [17] and elaborated in [14]. During the application of the rules each color has the following meaning:

- White: It has not been shown that actors in this set can communicate with a root actor
- Gray: Actors in this set are blocked but could communicate with a root actor if they can become active which has not yet been shown.
- Black: Actors in this set are nongarbage. They are either root actors, can communicate with a root actor, or are “remotely dependent.”

Remotely dependent actors are colored black to prevent their reclaimation by the local collector. The true coloring of remotely dependent actors is done by the global algorithm given in Section IV.

The algorithm, shown in Fig. 3, is named the push-pull algorithm as it uses 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. Before the puller is started for the first time, a snapshot of all local actors is obtained. While more sophisticated schemes are possible, the simple mechanism used here simply halts the mutator while the snapshot is taken. The extent to which the mutator must be halted during the distributed collection phase will be considered in Section IV.

Table I shows the result of applying the push-pull algorithm to the example system shown in Fig. 2. Notice that actors F, M, and O have been correctly identified as garbage actors on node 1 and that this determination is made by node 1 using only local information. Similarly, node 0 is able to determine that actors L and N can be reclaimed using only local information.
Table I

<table>
<thead>
<tr>
<th>PUSH-PULL ALGORITHM EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 0</td>
</tr>
<tr>
<td>Initialization</td>
</tr>
<tr>
<td>Gray: [1]</td>
</tr>
<tr>
<td>White: [B, C, L, N]</td>
</tr>
<tr>
<td>After Push-Pull algorithm</td>
</tr>
<tr>
<td>Gray: [L]</td>
</tr>
<tr>
<td>White: [N]</td>
</tr>
<tr>
<td>Local Garbage</td>
</tr>
</tbody>
</table>

Also notice that the local collector on node 1 has determined that actor I is nongarbage.

This section concludes by arguing that the push-pull algorithm is guaranteed to terminate. Notice that the algorithm terminates when the pusher terminates. The pusher will terminate if no actors were darkened in color on the most recent execution of the pusher. There can only be a finite number of such darkenings as there are a finite number of actors, a finite number of colors and an actor’s color can only be darkened. Thus, the pusher must eventually terminate.

IV. GLOBAL COLLECTION

As shown in Fig. 1, the global collector is comprised of three tasks: the initializer, the global agent and the global marker. In this section the roles played by each of these tasks is described in detail. This description is an extension of that given in [22].

Global garbage collection proceeds in three stages:

- **Initialization**: each node constructs the local part of a “globally consistent” snapshot of the mutator (actor system) and initializes the local data structures used to perform garbage collection. The meaning of “globally consistent” is given below.
- **Marking**: each node marks its local actors using both the local snapshot information and information obtained by exchanging messages with other nodes. Detecting the termination of this phase is an important part of the algorithm.
- **Reclamation**: each node reclaims the resource assigned to garbage actors. When the reclamation phase completes, the global garbage collection is done.

The initialization phase is begun by the initializer task at node 0. At the end of the initialization phase, control is passed at each node to the global agent at that node. The global agent and the global marker collaborate to perform the global coloring algorithm, to recognize the termination of the coloring algorithm and to initiate the reclamation phase. The reclamation phase is straightforward and, it will not be considered further except to note that all nodes perform their reclamation in parallel.

The three main problems in performing global garbage collection are how to:

- construct snapshots at each node that are globally consistent;
- color the snapshots in the presence of remote acquaintances;
- detect the termination of the global coloring algorithm.

The first of these problems, the snapshot problem, is handled by the initializer tasks (Section IV-A). The second and third problems, coloring and termination, are handled by the global agent (Section IV-B) and global marker (Section IV-C), respectively.

A. Initializer Task

The initializer task on node i constructs a snapshot of the portion of the mutator (actor system) on node i so that the local snapshots taken at each node are globally consistent. An example of obtaining a snapshot is given to illustrate the problem of global consistency and also to motivate a portion of the overall architecture that was shown in Fig. 1.

A simple system is shown in Fig. 4. Fig. 4(a) shows an initial state in which the actors B and D on node 0 are both blocked and the two actors on node 1 are both active. Consider the following sequence of events:

- Node 0 takes a local snapshot;
- actor C constructs a message, MSG, containing the identify of actor E;
- actor C sends message MSG to B; removes its arc to E and blocks;
- Node 1 takes a local snapshot.

Naively combining the two snapshots results in Fig. 4(b). In Fig. 4(b) the actors B, C, D, and E would be considered garbage because B, C, and D are permanently blocked while E is incapable of any communication. However, the true state of the mutator is shown in Fig. 4(c) where none of these actors...
are garbage. To avoid this problem, the snapshots of the nodes must be reconciled so that the effects of messages in transit, like MSG, are not lost. The coloring algorithm may be applied to the system as pictured in Fig. 4(a) (before MSG is sent) or in Fig. 4(c) (after MSG is received).

The algorithm used by the initializer tasks to insure globally consistent snapshots is due to Chandy et al. [6]. The Chandy-Lamport algorithm is adapted for use in the distributed garbage collector. This adaptation is described and a convincing, albeit informal, argument for the correctness of the adaptation is given.

The Chandy-Lamport algorithm constructs a globally consistent set of local snapshots sufficient to identify "stable properties." A stable property, roughly, is one which, if true in some state, is true in all reachable states. The property "is a garbage actor" is a stable property because the coloring algorithm identifies an actor as garbage if and only if there is no reachable state in which it is nongarbage.

The key rules in the Chandy-Lamport algorithm are:
- when a node takes a snapshot it places "markers" in each of its outgoing channels;
- when a node receives a "marker", it takes a snapshot if one does not already exist;
- after a node takes a snapshot, all messages on an incoming channel are recorded until a "marker" is received on that channel; and
- when markers have been received on all incoming channels the original snapshot is updated to reflect the effect of the recorded messages.

The recorded messages can have two effects on the snapshot: (1) changing the receiving actor from blocked to active, and/or (2) adding new acquaintances to the receiving actor. These effects are illustrated in the following discussion.

Using the example in Fig. 4, the sequence of actions in the Chandy-Lamport algorithm are as follows. Node 0 takes a snapshot and sends a marker on its single outgoing channel. On node 1, actor C sends message M to actor B, deletes the arc to actor E and blocks. When node 1 receives the marker, it takes a local snapshot and leaves a marker in its outgoing channel. Since it has received markers on all of its input channels and no messages need to be accounted for, the snapshot of node 1 is complete. Node 0, however, must observe the arrival of message M and apply its effects to the snapshot taken earlier. In this case, B is changed to active and is given an additional acquaintance (for E). The arrival of the marker on the input channel of node 0 indicates that node 0 has its snapshot is complete. The final snapshots of the two nodes are those shown in Fig. 4(c). Working with this snapshot, the global coloring algorithm would recognize that none of actors B, C, D, or E are garbage.

Notice that the initializer task must update its local snapshot to reflect the effects of messages received by the mutator on an incoming channel until a marker is received on that channel. Thus, during the (hopefully brief) interval during which the channel markers are being propagated, the initializer task must monitor the messages being delivered to the mutator.

The initializer tasks are shown in Fig. 5. The initializer task for node 0 is slightly different because it is the task that initiates the entire global garbage collection by sending begin_initialization messages to all other nodes. Once started, the sequence of actions taken by an initializer task at a node are as follows. First, it initializes all variables used by the global agent and global marker tasks. Second, it takes a local snapshot while the mutator is halted. In practical applications this step must be as short as possible. Third, it cooperates with the other initializer tasks to form a local snapshot that is globally coherent using the Chandy-Lamport algorithm. The end_initialization messages are used to signal the completion of this initialization phase. The initializer task at node 0 recognizes the termination of the snapshot phase and sends begin_marking messages to all nodes. The node 0 initializer task also starts the termination detection algorithm (to be discussed later). Finally, each initializer task invokes (and waits for the completion of) the global agent task at its node. When the global agent returns to the initializer, the reclamation of local garbage actors has been completed and the snapshot is discarded.

There are several methods to determine when the global collection should start. One method is for the initializer at node 0 to maintain a timer controlling the interval between global collections. A second method is for the local collectors to forward requests to the initializer at node 0 when they want to start a collection based on the utilization of local memory or the frequency of local collections. The global collection could start when a single request is received, a majority of requests are received, or requests from all nodes are received. For the purpose of this paper, any of these methods is sufficient. The method used in an implementation depends upon the particular environment in which the garbage collection takes place.

B. Global Marking

This section explains how the global marking is performed without regard for how the termination of this marking process is detected. The next subsection will illustrate why the termination detection is nontrivial and will show how the marking algorithms presented below are extended to support the termination detection.
1) Requirements for Global Marking: The simple example in Fig. 6, depicting three nodes each with a local marking task, will be used to illustrate the communication that is required among the marking tasks. Initially, all of the actors on all nodes are colored white, except for the root node, A, that is colored black. Consider a scenario in which the marking task at node 0 executes first. This node will change B’s color to gray and then stop as there are no remaining actors to be colored. Assume that the marker on node 2 is the next to execute. The marker on node 2 will not know how to color D because it does not know the color of D’s remote acquaintance (i.e., C). This illustrates that the marker tasks must be able to inquire about the color of actors on other nodes. In our scenario, node 2 inquires about the color of C from node 1. In this case node 2 will be told that the C’s color is white and, hence, the marker on node 2 will leave D colored white and will have no more work to do. Now assume that the marker on node 1 executes. It will inquire about the color of B, be told that B’s color is gray and, therefore, node 1 will color C black. Having colored C black, the marker on node 1 will also want to color all of C’s acquaintances (i.e., B) black as well. To do so requires that the marker on node 1 be able to inform the marker on node 0 about a change that must be made to the color of one of node 0’s actors. This information will cause the marker on node 0 to color B black. There is, however, one final requirement. Because of the acquaintance between C and D and the change in color of C, node 2 must be told to restart its marking. The marker on node 2 would then re-inquire about the color of C and, when told that C’s color is black, color D black as well.

The example in Fig. 6 illustrates that the local marker (shown in Fig. 3) must be extended to:

- inquire about the color of actors on other nodes,
- change the color of actors on other nodes, and
- restart another node’s marker.

These three extensions are referred to as the remote coloring operations.

2) Remote Coloring Operations: In implementing the remote coloring operations three steps are taken to minimize overhead. First, each actor is given an additional field (i.e., the actor.new field introduced in Section II) recording whether the actor has been newly placed in the black set. Because the global marker may be restarted several times, this field allows the marker not to reprocess actors in the black set for whom all actions have already been taken. Second, an “inquiry” field (i.e., the actor.inquiry field introduced in Section II) is added to each actor. This “inquiry” field is set when a message is sent to determine the color of one of the actor’s remote acquaintances. As an actor may be encountered several times during a marking phase, the inquiry field information prevents duplicate inquiry messages from being sent. Third, a boolean array node [0..N-1] is used to record what other nodes should have their global markers restarted. This information is collected and the “restart” messages are sent at convenient places in the algorithm. This minimizes sending redundant “restart” messages to a node.

The three extensions noted above and the manipulations of the “new addition” and “inquiry” fields are implemented by a set of utility routines that are shown in Fig. 7. These utility routines also manipulate several variables that are related to the detection of termination. These variables are:

WHO[0..N-1] WHO[i] is a count of the number of messages sent to node i for which matching reply messages have not been received. Note: this count does not apply to messages sent by the mutator. The name is an acronym for work handed out.

Thissnode_Color a color (black/white) given to the node as part of the termination detection.

Roughly, a node’s color is black if it does not consider itself ready to terminate. Thus, for example, if a node knows that its marking algorithm is to be restarted in the near future, it will not consider itself ready to terminate. Also, if there are unacknowledged messages, the node will not consider itself ready to terminate. These, and other variables related to termination, will be discussed in detail later.

The major steps in the utility routines are these. The Color_Local_Actor routine changes the color of the specified actor if such a coloring would darken the color of the actor (e.g., from white to gray or black, or from gray to black). The “changed” parameter, passed by reference, indicates if the color of the actor was changed. If the actor has just been colored black, the actor’s “new addition” field is set and its inquiry field is reset. The inquiry field is reset because the color of the actor is now certain and any previous inquiries are no longer needed. Finally, the node array is updated to record what other nodes should have their marking algorithms restarted. A node’s marking algorithm should be restarted if
that node has an actor with a reference to the actor that has just been darkened. The need for this restarting was illustrated in Fig. 6 above. Coloring a remote actor (Color_Remote_Actor)
amounts to sending a message to the node containing that actor and setting some information relevant to the termination detection. Inquiring about the color of a remote actor (Inquire_Remote_Color) also involves sending a message to the node containing the actor in question. The inquiry field in the local actor is set to avoid sending another inquiry message if this actor is encountered again by the marking algorithm. Finally restarting other nodes (Restart_Nodes) is achieved by sending “restart” messages to each node for which node[i]=1.

3) Global Push-Pull Algorithm: The outer level of the global marker is shown in Fig. 8. As with the local marker, the global marker has two components: a global pusher and a global puller. The global pusher (puller) performs the same logical action as the local pusher (puller), differing in the three extension noted above. The global marker continues to iterate as long as changes are made in the color of at least one actor. If a marking iteration does not change the color of any actor, then the marker will halt. The marker may be restarted due to an event on another node. The restarting was illustrated above in Fig. 6.

The global puller algorithm, shown in Fig. 9, operates as follows. Recall that the global puller colors black all actors that are acquaintances of actors in the black set. As explained above, the global puller only considers “new additions” to the black set. The local acquaintances of each such new addition are colored black using the Color_Local_Actor utility. For each remote acquaintance of an actor in the black set, a remote_color message is sent using the Color_Remote_Actor utility. This message requests that the color of the remote acquaintance be made black. Once each new actor in the black set has been examined, other nodes are restarted as necessary.

The global pusher algorithm, shown in Fig. 10, moves actors from the white set to the gray or black sets. In cases where the white actor’s acquaintance is local, the white actor is colored black if it is active and its acquaintance is black or gray. If the white actor is blocked and its local acquaintance is black or gray, the white actor is moved to the gray set. Other actions done by the Color_Local_Actor were explained earlier. In cases where the white actor’s acquaintance is remote, the Inquire_Remote_Color utility is used. The state (active/blocked) information of the local white actor is sent to the remote node so that the coloring rule can be evaluated at the remote node. The remote node then has the responsibility of sending an appropriate color message in response. The code for handling the inquire message is, as will be seen, in the global agent. Finally, as with the global puller, the global pusher restarts other nodes after it completes its examination of the set of white actors.

C. Global Agent

The global agent is responsible for coordinating the activities that occur during the marking phase. The principle activities, as shown in Fig. 11, are:

- (re)starting the global marker,
- reacting to messages sent from global markers on other nodes,
- starting other nodes, and
- detecting the termination of the marking phase.

As shown in Fig. 11, the control flow within the global agent is straightforward. After receiving a begin_marking message, the global marker is executed for the first time. Thereafter, until termination is detected, the global agent processes messages that it receives from other nodes, restarts other nodes, restarts the local global marker if it itself has received a restart message, and finally, checks for the termination of the marking phase. The global agent is message driven; each of its iterations is triggered by the arrival of some message.

The global marker algorithm has already been explained and the procedure to restart other nodes, was shown in Fig. 7. Process_Messages is described below (see Fig. 12) and Check_Termination is described in the Section IV-D (see Fig. 13).

The global agent has several variables that guide its actions. These variables are shared among the global agent and its components. These variables and their meaning are:
Global Agent at Node 1

\begin{verbatim}
begin
  Await begin, marking message; start Global Marker;
  do
    received = false;
    Await arrival of a message;
    Process Message();
    Report Node(Node);
    if (received)
      begin
        received = false;
        return Global Marker;
        end
    until term_mark;
  end
end
\end{verbatim}

Fig. 11. The global agent.

variable meaning

restarted a boolean variable indicating if the Global Marker should be restarted

done_token a colored (black/white) token passed among the Global Agents as part of the termination detection

got_token a boolean variable indicating if this node has received the done_token

term_mark a boolean variable set to true when termination is detected

Only the first of these variables pertains to the marking; the others will be considered in the discussion of the termination algorithm in Section IV-B.

The message processing done by the global agent is shown in Fig. 12. The simple iteration extracts an enqueued message and handles the message according to its type.

For a remote_color message, the attempt is made to change the local actor to the color contained in the message. If this attempt has the effect of darkening the color of the actor then the restarted flag is set. This will cause the global marker on this node to be executed again in the near future. In any event, a reply message is sent to acknowledge receipt of the remote_color message.

There are two forms of reply messages. In both forms of reply, the count of unacknowledged message for the replying node in the WHO array is reduced by one. The reply_with_color message carries a "piggybacked" request to change the color of a local_actor to new_color. This second form of reply is generated by the processing of an inquire message considered next.

An inquire message contains the identity and state of a remote actor that has local_actor as an acquaintance. Given the state of the remote actor and the color of the local actor, the color rules are evaluated to determine if the remote actor should be colored white, gray, or black. The resulting color is returned through a reply_with_color message.

The last three messages (restart, token and begin_reclamation) are simple. The restart message simply acknowledges receipt of the message and sets its restarted flag to true. The flag will cause the global marker to be restarted in the near future. The token message simply notes that the termination token (described later) has arrived and a flag is set to check for termination in the near future. The begin_reclamation message is sent by node 0 when termination of the global marking phase has been detected. A flag is set that will cause the global agent to terminate after reclaiming all garbage actors on this node.

D. Termination Detection

1) Dijkstra's Algorithm: The termination of the global garbage collector is determined using the termination detection algorithms developed by Dijkstra and others in [8] and [9]. In applying these algorithms we consider the global garbage collector to be the distributed computation and superimpose the termination detection algorithm over it. The termination detection process does not interfere with the global garbage collector just as the global collector does not interfere with the mutator.

In Dijkstra's algorithm the N nodes engaged in a distributed computation communicate by instantaneous and reliable message passing. A node engaged in the distributed computation is called active while a node not engaged in the distributed computation is called passive. Only active nodes can send messages; passive nodes become active on receipt of a message. The distributed computation terminates only when all nodes are passive.

For the purpose of detecting termination, the N nodes, numbered 0..N-1, are assumed to be arranged in a ring and a token is passed around this ring. Node 0 initiates a "probe" for termination by sending the token to node N-1. The token is subsequently sent to lower numbered nodes (i.e., from node i to node (i-1)), and eventually returns to node 0. When the token returns to node 0 either termination is detected or another probe is begun. The communication of the token is assumed to be independent of the message passing pertaining to the distributed computation.
Each node and the token has a color, either black or white. The color of a node, the token color, and the rules for propagating the token are as follows:

- **Rule 0**: Node 0 initiates a probe by making itself white and sending a white token to node $N - 1$.
- **Rule 1**: When active, node $i + 1$ keeps the token; when passive, it hands over the token to node $i$.
- **Rule 2**: When node $i + 1$ propagates the token, it hands over a black token to node $i$ if it is black itself, whereas while being white it leaves the color of the token unchanged.
- **Rule 3**: Upon transmission of the token to node $i$, node $i + 1$ becomes white.
- **Rule 4**: A node sending a message that is part of the distributed computation (the global garbage collector in our case) makes it itself black.
- **Rule 5**: After the completion of an unsuccessful probe, node 0 initiates a next probe.

The algorithm maintains an invariant that is the logical OR of the three following requirements:

- $P_0$: (for all $i : t < i < N$ : Node $i$ is passive);
- $P_1$: (there exists $j : 0 <= j <= t : Node j$ is black);
- $P_2$: The token is black;

where $t$ is the node at which the token currently resides. Termination is detected when

- the token comes back to node 0;
- only $P_0$ is true; and
- Node 0 is passive.

The probe for termination detection is unsuccessful if either of $P_1$ or $P_2$ is true when the token reaches node 0.

2) **Adapting Dijkstra’s Algorithm**: Our model of distributed garbage collection satisfies all the requirements of Dijkstra’s termination detection algorithm except that in our model message passing is not considered to be instantaneous. As a result, messages might take an arbitrary but a finite amount of time to reach their destinations. This departure makes it necessary to extend Dijkstra’s node and token coloring scheme.

Following the idea in [8], we require that each message sent out from a node must be acknowledged by the receiver. The acknowledgment ensures that a message sent out by a node has actually reached the destination node and is not being delayed by the network. To keep track of the message passing among nodes, whenever a message is sent out, the appropriate element of the WHO array is incremented and whenever an acknowledgment is received the same element is decremented. Note that it is not important to pair up each message sent out with its matching acknowledgment. Instead, the algorithm relies on whether the total number of messages sent out to a particular node has been acknowledged by that node.

The Check_Termination algorithm, shown in Fig. 13, implements modified forms of the five rules given above for Dijkstra’s algorithm. The only modifications to the original rules are those necessary to account for the possible delays in message delivery.

The following discussion of the modified rules provides an informal correctness argument for our termination detection algorithm.

**Rule 0**: We restate this rule as follows: Node 0 initiates a probe by making itself white only if WHO[i] = 0, for all $i$, and sending a white token to node $N - 1$. If a white token is received, node 0 forwards a white token to node $N - 1$ without changing its own current color.

This rule and rule 5 below is realized by lines 2–19 in the Check_Termination algorithm.

**Rule 1**: We interpret “active” as being the state of the global collector in which either the global marker is executing OR the global agent is processing messages from its work_queue. The token is forwarded only when a node is not “active”.

From the structure of the global agent in Fig. 11, it is evident that the termination detection algorithm, Check_Termination, which forwards the token, executes only when the node is not “active”.

**Rule 2**: Note that this rule applies to nodes 1 through $N - 1$ and NOT to node 0 since node 0 does not propagate the token, it initiates and re-initiates the probe.

This rule is realized by lines 21 through 27 in the Check_Termination algorithm which are self-explanatory. Lines 23 and 24 will be discussed along with Rule 3.

**Rule 3**: This rule is restated as follows: Upon transmission of the token to node $i$, node $i + 1$ becomes white only if WHO[i] = 0, for all $i$. (This rule applies to nodes 1 through $N - 1$).

This rule is realized by lines 23 and 24 in the Check_Termination algorithm. The reason for this change will be discussed in Rule 4 next.

**Rule 4**: We restate this rule as follows: A node sending a message to node $i$ (a) colors itself black, and (b) increments WHO[i].

Incrementing WHO[i] ensures that the handshaking will take place properly and coloring the node black signifies that the token must make another pass around the network to account for the remote activities that this message may
initiate. This rule is realized by the Color_Remote, Actor_Inquire, Remote_Color, and Restart_Nodes utilities which are the only sections in the algorithm from which messages are sent to other nodes.

Rule 5: A probe is unsuccessful if node 0 receives a black token from node 1.

A black token signifies that there is at least one node in the network that sent out a message which may cause a “passive” node to become “active”. To take into account this possible revival of activity, node 0 has to reinitiate the probe. This rule is realized by the Check_Termination algorithm and was discussed along with rule 0 above.

One major change, that applies to all the nodes and is necessary to take into account the delays in message transmission, is that of checking the WHO array before changing the color of a node to white. If not all the elements in WHO are not zero (i.e., WHO is not empty), a node, node I, say, is awaiting acknowledgments from at least one other node. Now, both the messages sent out by node I and the acknowledgments it is expecting, may be delayed by an arbitrary amount of time by the network. If node I colors itself white (with WHO nonempty), transmits a black token, node 0 initiates another probe on seeing a black token, a white token arrives at node I in the next pass, and still a message sent out by node I or an acknowledgment expected by node I is held by the network, then node I will forward a white token that may trigger a false termination. To counter this situation, the coloring of a node from black to white has been made contingent on the existence of an empty WHO array.

When node I, say, is black and WHO is empty, the color of node I is changed to white but it still needs to forward a black token so that another probe is initiated. This is because, on its last pass around the network from node I back to it, a message from node I may have arrived at another node, node j, say, only after the token was forwarded by node I. This, once again, may trigger a false termination.

The termination of the global collector is determined by the Check_Termination task executing on node 0 only when the two following conditions are true:

1) The done_token is white (line 4). This signifies that the global marker is idle on every node and the WHO array is empty on every node, except, possibly, on node 0 itself.

2) The color of node 0 is not black (line 6) which means the color must be white. If the color of node 0 is white then WHO must be empty.

The above conditions make Dijkstra’s invariant true by making P0 true and both P1 and P2 false. They also ensure that the token is at node 0 and node 0 is not “active”. Thus, it can be concluded that the global collector distributed algorithm is not executing on any node. Therefore, the Check_Termination task on node 0 sends a begin_reclamation message to all nodes triggering local garbage collections (line 13).

The last question to be answered is whether termination will ever be detected. It will now be argued that the distributed collector does terminate. Recall that it was argued in Section III that the push-pull algorithm on a single node will terminate.

The only additional complication for the distributed algorithm is the possibility that the global marker on a node might be restarted indefinitely. This cannot happen because a node is restarted only when at least one actor on some other node has been darkened in color. Following the argument in Section III, this can only occur a finite number of times. Thus, it is guaranteed that the global collector will eventually terminate.

E. Time-Space Complexity

The execution time of the global garbage collector will be dominated by the time required to send messages among nodes. In the worst case, on the order of \(k^3\) total messages must be sent to garbage collect a distributed actor system of \(k\) actors.

An informal argument that this is the worst case is as follows. The activity of the garbage collector can be divided logically into “steps” each of which ends when a single actor has been colored. Since each actor can only be colored twice, there can be at most \(O(k)\) steps. During each step, the worst case is that each actor on each node would be examined and inquire messages would be sent for that actor’s acquaintances. For \(k\) actors, then can be at most \(O(k^2)\) acquaintances assuming that actors do not maintain redundant acquaintances (e.g., actor A would not have two acquaintance arcs to actor B). Thus, \(O(k^2)\) messages can be generated for each step. The worst case is, therefore, \(O(k^3)\).

The space required for the garbage collector is directly related to the number of acquaintances. Space is required for two types of information. First, space is required to store the acquaintance information and the inverse acquaintance information, both of which are clearly proportional to the number of acquaintances. Second, buffer space is required for the various types of messages sent between nodes. The number of message is also directly related to the number of acquaintances. An inquire message, for example, is only sent when an acquaintance exists. For \(k\) actors, there can be at most \(O(k^2)\) acquaintances. Thus, the space cost is \(O(k^3)\) in the worst case. The space required on each node is \(O(k^2/N)\) assuming the actors are evenly divided among the \(N\) nodes.

F. Example

This section concludes by reconsidering the simple example shown in Fig. 2. Table II below shows the result of executing the global collector with the snapshot shown in Fig. 2. This table shows only the end result of the coloring, it does not show the many messages that are exchanged or the starting and restarting of the two global markers that would be involved. It is beyond the scope of this paper to show the full details of the example.

The ability of the collector to reclaim cyclic garbage is illustrated in Table II. The two actors K and J form an isolated cycle. Both of these actors have been properly identified as garbage in Table II. In general the rules underlying the push-pull algorithm will never color an actor is such an isolated cycle. Thus, actors in an isolated cycle remain white and, therefore, are recognized as garbage.
TABLE II
GLOBAL COLLECTOR EXAMPLE

<table>
<thead>
<tr>
<th></th>
<th>Node 0</th>
<th>Node 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>[A], [I]</td>
<td>[H], [I]</td>
</tr>
<tr>
<td></td>
<td>[B, C, D, K, L, N]</td>
<td>[E, F, G, I, J, M, O]</td>
</tr>
<tr>
<td>After Termination</td>
<td>[A, B, C, D]</td>
<td>[E, G, H, I]</td>
</tr>
<tr>
<td></td>
<td>[L], [K, N]</td>
<td>[F, M]</td>
</tr>
<tr>
<td>Global Garbage</td>
<td>[K, L, N]</td>
<td>[F, J, M, O]</td>
</tr>
</tbody>
</table>

V. CONCLUSIONS

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

- automatic reclamation of active objects forming an arbitrary communication topology.
- autonomous local garbage collection without global synchronization.
- concurrent local and global collection.
- concurrent execution of the mutator with the local and global collectors.
- collection of actors that are part of a local or distributed isolated cyclic structure.

Critical operations of the collector involve creating a globally consistent snapshot of the actor system, marking the snapshot to detect reclaimable actors, and detecting the termination of the marking phase. Arguments were given in favor of the correctness of the algorithms used in these operations. One issue that deserves some discussion is the role of the inverse acquaintances. In practice, it is not efficient to maintain the inverse acquaintance information: extra space is required to store this information and additional messages are necessary to update an actor’s inverse acquaintance list whenever its name is passed between actors. The price to be paid for not maintaining the inverse acquaintances is that, during the global marking, a node will restart all other nodes whenever it darkens the color of an actor. This causes unnecessary work. A compromise strategy is for the nodes to exchange inverse acquaintance information only when the snapshot is being formed, perhaps piggy-backed with the channel markers. This issue needs additional study to determine the relative costs and benefits of these three alternatives.

Further optimization of the garbage collection algorithm is possible. For example, the global pusher algorithm can be revised to reduce the number of inquiry messages that it sends by: (1) checking all local acquaintances first so that no inquiry messages are sent if the actor can be darkened because of local information, and (2) changing the boolean inquiry flag to a counter so that redundant inquiry messages are avoided if the node is restarted before all replies have been received. These optimizations have not been included to simplify the presentation but should be included in an implementation.

There are several limitations of the collector given in this paper. These limitation—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 effect 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++ [16], 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.

REFERENCES


Dennis Kafura (M’83) received the Ph.D. degree from Purdue University in 1974. He joined the faculty at Virginia Tech in 1981. His current research interests include the measurement in software engineering and the application of object-oriented techniques to systems that are concurrent, distributed, embedded, and/or real-time in nature. Further information on the work with object-oriented programming is available from http://actor.cs.vt.edu.

Manibharta Mukherji received the B.C.S.E. degree in computer science and engineering from Jadavpur University, Calcutta, India, in 1986 and the M.S. degree in computer science and applications from Virginia Polytechnic Institute and State University in 1992. Currently, he is a Ph.D. candidate in computer science in the Department of Computer Science, VPI&SU. His research interests include the design and implementation of concurrent object-oriented programming languages and the application of process algebraic frameworks in providing semantics to such languages.

Douglas M. Washabaugh received the B.S.E.E. degree from Pennsylvania State University in 1984, and the M.S. degree in computer science from Virginia Tech in 1989. He has been developing computer network products at Digital since 1984. He is now a Principle Software engineer and is currently defining and developing software for FDDI and ATM network interface cards.