Scalable Distributed Garbage Collection for Systems of Active Objects*

Nalini Venkatasubramanian** ***, Gul Agha***, and Carolyn Talcott†
email: nalini@cs.uiuc.edu, agha@cs.uiuc.edu, clt@sail.stanford.edu

[1] University of Illinois, Urbana-Champaign, IL 06120, USA
[2] Stanford University, Stanford, CA USA

Abstract. Automatic storage management is important in highly parallel programming environments where large numbers of objects and processes are being constantly created and discarded. Part of the difficulty with automatic garbage collection in systems of *active* objects, such as actors, is that an active object may not be garbage if it has references to other reachable objects, even when no other object has references to it. This is because an actor may at some point communicate its mail address to a reachable object thereby making itself reachable. Because messages may be pending in the network, the asynchrony of distributed networks makes it difficult to determine the current topology. Existing garbage collection schemes halt the computation process in order to determine if a currently inaccessible actor may be potentially active, thus precluding a real-time response by the system. We describe a generation based algorithm which does not require ongoing computation to be halted during garbage collection. We also outline an informal proof of the correctness of the algorithm.

Keywords: actors, asynchrony, distributed systems, generation scavenging, network clearance, broadcast and bulldoze communication, snapshot.

1 Introduction

We describe a garbage collection algorithm, HDGC (hierarchical distributed garbage collection), for systems of active objects distributed across a network of nodes. An important advantage of our algorithm is that it is non-disruptive: it does not halt or otherwise interfere with the ongoing computation process. A

* This research was partially supported by DARPA contract NAG2-703, by DARPA and NSF joint contract CCR 90-07195, by ONR contract N00014-90-J-1899, and by the Digital Equipment Corporation.

** Current address: Hewlett Packard Company, 19111 Pruneridge Avenue MS44UT, Cupertino, CA 95014, USA.
*** Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801
† Department of Computer Science, Stanford University, Stanford, CA 94305

novel feature is the recording of a GC-snapshot to obtain a consistent local and global view of the accessability relation. The algorithm is described in terms of the actor model. However, it is applicable to any language supporting dynamic creation and reconfiguration of objects (passive or active), executed on a network with a *global name space* distributed across the nodes [1]. The HDGC algorithm can be adapted to a wide range of parallel architectures including fine, medium or large grained MIMD machines, message passing, shared memory or distributed shared memory machines, or networks of workstations. This paper presents the conceptual aspects of the algorithm. An implementation effort is in progress. There are numerous possible optimizations. These are discussed briefly in the conclusion.

The Actor Model [Hew77,Agh86] provides a good abstraction for discussing concurrent computation in distributed systems. Here, the universe contains computational agents called *actors*. Each actor has a conceptual location (its *mail address*) and a *behavior*. The only way one actor can influence the actions of another actor is to send the latter a communication. Communication between actors is asynchronous, and every communication sent will be delivered after some finite but unbounded delay (fairness of mail delivery). If an actor α knows the mail address of an actor β, then β is called a *forward* acquaintance of α and α is called an *inverse* acquaintance of β. An actor can send communications only to its forward acquaintances. Mail addresses of actors may be communicated: thus the interconnection topology is dynamic. We call the actor addresses occurring in a communication the acquaintances of that communication. On receiving a communication, an actor processes the message and as a result may cause one or more of the following actions: (1) creation of a new actor, (2) alteration of its behavior and its acquaintances, (3) transmission of a message to an existing actor. Every actor is equipped with a mailbox that queues incoming communications.

In order to make sense of the notion of distributed memory management we need to refine the abstract actor model to account for local grouping of actors on nodes (processing units) and to account for the network interconnecting these nodes. We assume that the network consists of channels linking pairs of nodes. Each channel consists of a pair of directed links (one in each direction) with infinite message buffers. We require that message order is preserved across a single link and that the network routing satisfies certain progress-only constraints that will be made precise in the next section. In addition to normal messages between actors, there will also be special messages used for GC.

Traditionally, garbage is detected by starting with some pre-defined root set and forming the transitive closure of the acquaintance (referenced objects) relation. In actor-like systems there are two problems. First, the acquaintance relation is distributed and changes dynamically. Thus we must find a way of establishing a GC start time and determining the acquaintance relation as of this point in time, as a distributed snapshot. Second, simply following acquaintance links from the root set is not adequate. This is because, using that definition, a non-reachable actor can become reachable, at some later time, by communi-

[1] This memory architecture is often referred to as *distributed shared memory*

cating its address to a reachable actor. These problems are addressed in the HDGC algorithm by first obtaining a (locally and globally) consistent snapshot of the acquaintance relation, then computing reachability according to an algorithm that accounts for actors that are potentially reachable relative to the snapshot. The HDGC algorithm is *conservative*, i.e., it identifies only a subset of inaccessible objects during a GC. For example, a potentially reachable object may become inactive without communicating its mail address to any reachable object. However, all unreachable objects will be collected by some subsequent GC.

The remainder of the paper is organized as follows. In §2 we outline the full HDGC algorithm. In §3 we present the algorithm for establishing a consistent snapshot of the acquaintance relation at the start of GC. In §4 we define reachability and present an algorithm for marking reachable objects. §5 contains an informal outline of a proof of correctness. §6 contains concluding remarks.

2 Hierarchical Distributed Garbage Collection

A hierarchical organization partitions a distributed system into smaller subsystems. These subsystems may in turn be further partitioned. The topmost level of the hierarchy is the entire system. The lowermost level of the hierarchy has a single node per subsystem. There may be zero or more intermediate levels. The organization of the distributed system into subsystems may be static or dynamic (cf. [LQP92]). The motivation for dividing a large, distributed system into smaller subsystems is to avoid the bottleneck inherent in global resource management.

To accurately determine garbage in a subsystem at any level other than the top level, it is necessary to know which internal actor addresses have been communicated to some external actor. Such actors are called the *receptionists* of the subsystem. They must be considered reachable (part of the root set) for a GC local to the subsystem. A receptionist table is constructed by adding an actor whenever a reference to that actor is passed out of the subsystem. This provides a conservative approximation to reachability. It can be improved by determining when entries in the receptionist tables are no longer accessible, but this requires global cooperation. The approximation can also be improved by maintaining a reference count of the number of outstanding references to each receptionist (cf. [SGP90]). This also entails some overhead.

With purely local GC, the additional synchronization mechanisms presented here are not necessary. Each node independently maintains a receptionist table that becomes a part of the root set for local GC. We believe that accurate receptionist table management entails a high overhead especially if the objects are not large grained. Furthermore, we would like to provide a mechanism to preserve the globality of information required for massively parallel MIMD computation. Note that the algorithm presented here can be suitably modified to avoid synchronization if GC is purely local.

We present the HDGC Algorithm in the context of a two level hierarchy, i.e. global and node level collections. The generalization to hierarchies with intermediate levels is relatively straightforward. We can use any of the traditional algorithms for local GC. The best algorithm to use will depend on the granularity of the nodes as well as on particular application domains. It is not necessary for all nodes to use the same algorithm.

The HDGC algorithm consists of five steps: Pre-GC, DistributedScavenge, Local-Clear Initiation, Local-Clear, and Post-GC. There is a unique (per subsystem) special actor designated as the GC-root actor. Requests for GC go to the GC-root actor and sequencing of the GC steps are synchronized through the GC-root actor. Thus the algorithm does not require a global clock in the system. We describe the purpose of each step below. The steps are initiated and carried out by communication of GC related messages. Details are given in the following sections. The behavior of the GC-root actor will be described after these details have been filled in.

Step 1: Pre-GC. In a system with distributed state there is no uniquely determined global state. Thus to compute some property of the state it is generally necessary to determine a global snapshot that determines a consistent view of the state. In the case of the acquaintance relation for an actor system, the problem of obtaining a consistent global snapshot involves an additional subtilty. The asynchrony of communication together with the ability to communicate acquaintances means that at any given time, there can be communications in the network whose acquaintances are no longer acquaintances of the sender, and not yet acquaintances of the receiver. This means that before a snapshot of the acquaintance relation can be taken, the network must be cleared of such communications. During the pre-GC step each node is notified that a GC has been initiated, and the network is cleared of messages in transit at the time GC was initiate. This defines a local start-of-GC time on each node that is globally consistent. Each node records GC information relative to its start-of-GC time that will persist throughout the duration of the GC. The combined local information forms a consistent global snapshot of the system state that is adequate to determine the reachability of each actor in the system. We call this the *GC snapshot*. A detailed description of information and of the process of recording the GC snapshot is presented in section 3.

Step 2: The Distributed Scavenge Phase. During this step, actors that are non-garbage relative to the GC snapshot are marked *touched*. The definition of non-garbage and the *distributed scavenge algorithm* for marking non-garbage actors is described in section 4.

Step 3: Local-Clear Initiation. Each node in the system is informed that the distributed scavenge phase has completed and local clearance begins. On each node, objects not marked touched are cleared from local memory, according the nodes method of memory management, and any other actions (updating receptionist tables, etc.) entailed by this reclamation are carried out.

Step 4: Local-Clear Phase. This step detects when all nodes have completed the local clearance initiated in the previous step.

Step 5: Post GC Broadcasts. This step informs each node that the current GC is complete: each node can now note that GC is no longer in progress and update necessary information to reflect this state. At the end of this step a new GC can be initiated at anytime.

3 Asynchrony in Distributed GC

In this section we describe how the start-of-GC time is established and how the recording of the GC snapshot is accomplished. The key idea is that in addition to ordinary (actor-to-actor) communications, new types of messages are introduced that propagate through the network in pre-established patterns, and can thus be used for various forms of synchronization. To describe these messages, we make additional assumptions about the network topology.

3.1 Message Routing in the Network

For simplicity we restrict our attention to networks of nodes that form two dimensional grids. Such a grid contains an $m \times n$ array of nodes. Each node is designated by a pair of integers (a_1, a_2), where $1 \leq a_1 \leq m$ and $1 \leq a_2 \leq n$. A node (a_1, a_2), is an *Fneighbor* of a node (b_1, b_2) if either $a_1 = b_1 + 1$ and $a_2 = b_2$, or $a_2 = b_2 + 1$ and $a_1 = b_1$. Similarly, a node (a_1, a_2), is a *Bneighbor* of a node (b_1, b_2) if $a_1 = b_1 - 1$ and $a_2 = b_2$, or $a_2 = b_2 - 1$ and $a_1 = b_1$. Connecting each Fneighbor/Bneighbor pair of nodes X/Y is a channel comprised of a pair of unidirectional FIFO links, one from X to Y and one from Y to X. An *Fpath* is a path in the network that progresses only along Fneighbor links. A path in the network that progresses only along Bneighbor links is a *Bpath*. We call $(1, 1)$ the *start node* of the system. It is the unique node from which there exists an Fpath to every other node in the system. Dually, we call (m, n) the *finish node*. It is the unique node from which there exists a Bpath to every other node in the system.

Ordinary messages are assumed to be routed from the node where the sender resides to the node where the receiver resides via paths that are *progress-only* in the sense that the paths contain at most one Fpath segment and at most one Bpath segment. Thus the route of an ordinary message is either an Fpath, a Bpath, an Fpath followed by a Bpath, or a Bpath followed by an Fpath.

In addition to ordinary messages, we introduce two kinds of node-to-node messages: *broadcast messages* and *bulldoze messages*. These messages propagate to every node in the system, and are used for synchronization and network clearance. The node-to-node messages may also contain information indicating actions to be carried out. *Broadcast messages* are propagated from the start node to all the nodes in the network, along some subset of links. The protocol for propagating a broadcast message is illustrated in Figure 4. Each node has a designated set of *broadcast predecessors* and *broadcast successors*. A node can issue a broadcast message to its *broadcast successors* only after it has received the message from all of its broadcast predecessors. The broadcast is considered

complete when the finish node has received messages from all of its broadcast predecessors.

-bb-error =

Fig. 1. The Broadcast Wavefront: The figure shows the broadcast messages traversing through the network as a wavefront. The broadcast messages are initiated at the *start* node and travel along indicated route to every node in the network.

There are two types of bulldoze messages, *Fbulldoze messages* and *Bbulldoze messages*. *Fbulldoze messages* are initiated at the start node and propagate along all Fneighbors links. Non-start nodes in the network issue an Fbulldoze message to their Fneighbors only after they receive Fbulldoze message from all of their Bneighbors. Dually, *Bbulldoze messages* are initiated at the finish node and propagate along all Bneighbor links, and non-finish nodes in the network can issue a Bbulldoze message to its Bneighbors only after it receives the Bbulldoze message from both its Fneighbors. The propagation of a bulldoze message forms a wave as illustrated in Figure 5. Bulldoze messages traverse every channel in the network and, by the FIFO assumption on links, force messages already in the network to be cleared along the direction of the bulldoze. A broadcast message does not in general traverse all forward links in the network. Thus the number of messages needed to accomplish a broadcast is less than the number of messages needed to accomplish a bulldoze.

-bb-error =

Fig. 2. The Forward and Backward Bulldoze Wavefronts: The figure shows the forward (FB) and backward bulldoze (BB) messages traversing through the network as a wavefront. The FB messages are initiated at the *start* node and travel along Fpaths until they reach the *finish* node. The BB messages are initiated at the *finish* node and travel along Bpaths until they reach the *start* node.

3.2 Obtaining a Consistent GC Snapshot

A GC snapshot consists of acquaintance and active status information that determines a consistent global view of the state of the system at start-of-GC time. Each node records, for each of its actors, its GC-acquaintances, its GC-inverse-acquaintances, and whether or not it was active at start-of-GC time. The GC-acquaintances of an actor are the current acquaintances, plus any acquaintances in messages in the network prior to the start of actual garbage collection. This is a safe approximation of the actors acquaintances, and insures that actors actually forgotten by one actor but sent in messages during GC will not be lost. The GC-inverse-acquaintances of an actor the set of actors having that actor as

a GC-acquaintance. This information is used to account for apparently unreachable actors that might communicate their mail addresses to a reachable actor. The GC acquaintance information is used only for GC and can be discarded when the GC for which it was created is complete.

For a global snapshot of the state of the system, we need to guarantee that both *local consistency* and *global consistency* have been achieved. Every node in the system needs a point of reference in time with respect to which it determines the accessibility or inaccessibility of actors in its memory. Once a node has established this point and recorded the necessary information, we have attained *local consistency*. *Global consistency* is a point in time when all participating nodes have agreed on a particular state of the distributed system.

In order to determine which messages were in the network prior to the start of GC and which entered after, ordinary messages are given *tags* to classify them as *old* or *new* messages. *Old* (resp. *new*) messages are messages which were created prior to (resp. after) the time of the GC snapshot. When GC is initiated, all messages in the network are tagged *old*. During the process of recording the GC snapshot, the network will be cleared of *old* messages by means of the forward and backward bulldoze messages explained above.

To obtain the GC snapshot, first a pre-GC message is broadcast to every node in the system. When a node receives the pre-GC broadcast message, it initializes the GC-acquaintances of each actor residing on that node with (1) its current acquaintances and (2) all acquaintances contained in messages currently residing in its mail queue. Any acquaintances contained in *old* messages subsequently obtained from the network are added to the GC-acquaintances. It also initializes GC-inverse-acquaintances to be empty. When the pre-GC broadcast is complete, a pre-GC Fbulldoze message is initiated (by the finish-node). When the pre-GC Fbulldoze message passes a node, it marks as active any objects with non-empty mailqueue. The active status of this node is retained for the current GC even though the node may become inactive during GC. Any messages subsequently communicated from that node are be tagged *new*. The *new* tag on a message guarantees the recipient of the message that any acquaintances communicated in the message have already been accounted for. When the Fbulldoze message reaches the finish node a Bbulldoze message is initiated. When the Bbulldoze message passes a node, this signals that the recording of GC-acquaintances is complete. The node sends *I-know-you* messages from each of its actors to each GC-acquaintance of that actor. When an I-know-you message from actor A to actor B is received then actor A is added to the GC-inverse-acquaintances of actor B. A second forward and backward bulldoze phase is required to clear the network of I-know-you messages. This is initiated by the start node upon completion of the first backward bulldoze wave. When the second forward/backward bulldoze wave is complete, the start node sends a pre-GC-complete message to the root node. At this point, all *old* and *I-know-you* messages in the system have been cleared from the network and the snapshot information is recorded.

The backwards bulldoze messages are needed for both the recording of GC-acquaintances and GC-inverse-acquaintances, since the forward bulldoze only

clears forwards links and there may be messages traversing backwards links that need to be recorded. To see this, note that after an object, say A, has received the pre-GC Fbulldoze message it can send only *new* messages. However, it may receive *old* messages from an actor H which has not yet received the pre-GC Fbulldoze message (see Figure 6).

-bb-error =

Fig. 3. The Bulldoze Wavefront in Progress: The bulldoze wavefront is halfway through the system. Although object A has started recording acquaintances and issues only *new* messages, it can receive and *old* messages from object H which has not yet received the bulldoze wave.

4 Asynchrony in Distributed GC

In this section we describe how the start-of-GC time is established and how the recording of the GC snapshot is accomplished. The key idea is that in addition to ordinary (actor-to-actor) communications, new types of messages are introduced that propagate through the network in pre-established patterns, and can thus be used for various forms of synchronization. To describe these messages, we make additional assumptions about the network topology.

4.1 Message Routing in the Network

For simplicity we restrict our attention to networks of nodes that form two dimensional grids. Such a grid contains an $m \times n$ array of nodes. Each node is designated by a pair of integers (a_1, a_2), where $1 \leq a_1 \leq m$ and $1 \leq a_2 \leq n$. A node (a_1, a_2), is an *Fneighbor* of a node (b_1, b_2) if either $a_1 = b_1 + 1$ and $a_2 = b_2$, or $a_2 = b_2 + 1$ and $a_1 = b_1$. Similarly, a node (a_1, a_2), is a *Bneighbor* of a node (b_1, b_2) if $a_1 = b_1 - 1$ and $a_2 = b_2$, or $a_2 = b_2 - 1$ and $a_1 = b_1$. Connecting each Fneighbor/Bneighbor pair of nodes X/Y is a channel comprised of a pair of unidirectional FIFO links, one from X to Y and one from Y to X. An *Fpath* is a path in the network that progresses only along Fneighbor links. A path in the network that progresses only along Bneighbor links is a *Bpath*. We call $(1, 1)$ the *start node* of the system. It is the unique node from which there exists an Fpath to every other node in the system. Dually, we call (m, n) the *finish node*. It is the unique node from which there exists a Bpath to every other node in the system.

Ordinary messages are assumed to be routed from the node where the sender resides to the node where the receiver resides via paths that are *progress-only* in the sense that the paths contain at most one Fpath segment and at most one Bpath segment. Thus the route of an ordinary message is either an Fpath, a Bpath, an Fpath followed by a Bpath, or a Bpath followed by an Fpath.

In addition to ordinary messages, we introduce two kinds of node-to-node messages: *broadcast messages* and *bulldoze messages*. These messages propagate to every node in the system, and are used for synchronization and network clearance. The node-to-node messages may also contain information indicating actions to be carried out. *Broadcast messages* are propagated from the start node to all the nodes in the network, along some subset of links. The protocol for propagating a broadcast message is illustrated in Figure 4. Each node has a designated set of *broadcast predecessors* and *broadcast successors*. A node can issue a broadcast message to its *broadcast successors* only after it has received the message from all of its broadcast predecessors. The broadcast is considered complete when the finish node has received messages from all of its broadcast predecessors.

-bb-error =

Fig. 4. The Broadcast Wavefront: The figure shows the broadcast messages traversing through the network as a wavefront. The broadcast messages are initiated at the *start* node and travel along indicated route to every node in the network.

There are two types of bulldoze messages, *Fbulldoze messages* and *Bbulldoze messages*. *Fbulldoze messages* are initiated at the start node and propagate along all Fneighbors links. Non-start nodes in the network issue an Fbulldoze message to their Fneighbors only after they receive Fbulldoze message from all of their Bneighbors. Dually, *Bbulldoze messages* are initiated at the finish node and propagate along all Bneighbor links, and non-finish nodes in the network can issue a Bbulldoze message to its Bneighbors only after it receives the Bbulldoze message from both its Fneighbors. The propagation of a bulldoze message forms a wave as illustrated in Figure 5. Bulldoze messages traverse every channel in the network and, by the FIFO assumption on links, force messages already in the network to be cleared along the direction of the bulldoze. A broadcast message does not in general traverse all forward links in the network. Thus the number of messages needed to accomplish a broadcast is less than the number of messages needed to accomplish a bulldoze.

-bb-error =

Fig. 5. The Forward and Backward Bulldoze Wavefronts: The figure shows the forward (FB) and backward bulldoze (BB) messages traversing through the network as a wavefront. The FB messages are initiated at the *start* node and travel along Fpaths until they reach the *finish* node. The BB messages are initiated at the *finish* node and travel along Bpaths until they reach the *start* node.

4.2 Obtaining a Consistent GC Snapshot

A GC snapshot consists of acquaintance and active status information that determines a consistent global view of the state of the system at start-of-GC time. Each node records, for each of its actors, its GC-acquaintances, its GC-inverse-acquaintances, and whether or not it was active at start-of-GC time. The GC-acquaintances of an actor are the current acquaintances, plus any acquaintances in messages in the network prior to the start of actual garbage collection. This is a safe approximation of the actors acquaintances, and insures that actors actually forgotten by one actor but sent in messages during GC will not be lost. The GC-inverse-acquaintances of an actor the set of actors having that actor as a GC-acquaintance. This information is used to account for apparently unreachable actors that might communicate their mail addresses to a reachable actor. The GC acquaintance information is used only for GC and can be discarded when the GC for which it was created is complete.

For a global snapshot of the state of the system, we need to guarantee that both *local consistency* and *global consistency* have been achieved. Every node in the system needs a point of reference in time with respect to which it determines the accessibility or inaccessibility of actors in its memory. Once a node has established this point and recorded the necessary information, we have attained *local consistency*. *Global consistency* is a point in time when all participating nodes have agreed on a particular state of the distributed system.

In order to determine which messages were in the network prior to the start of GC and which entered after, ordinary messages are given *tags* to classify them as *old* or *new* messages. *Old* (resp. *new*) messages are messages which were created prior to (resp. after) the time of the GC snapshot. When GC is initiated, all messages in the network are tagged *old*. During the process of recording the GC snapshot, the network will be cleared of *old* messages by means of the forward and backward bulldoze messages explained above.

To obtain the GC snapshot, first a pre-GC message is broadcast to every node in the system. When a node receives the pre-GC broadcast message, it initializes the GC-acquaintances of each actor residing on that node with (1) its current acquaintances and (2) all acquaintances contained in messages currently residing in its mail queue. Any acquaintances contained in *old* messages subsequently obtained from the network are added to the GC-acquaintances. It also initializes GC-inverse-acquaintances to be empty. When the pre-GC broadcast is complete, a pre-GC Fbulldoze message is initiated (by the finish-node). When the pre-GC Fbulldoze message passes a node, it marks as active any objects with non-empty mailqueue. The active status of this node is retained for the current GC even though the node may become inactive during GC. Any messages subsequently communicated from that node are be tagged *new*. The *new* tag on a message guarantees the recipient of the message that any acquaintances communicated in the message have already been accounted for. When the Fbulldoze message reaches the finish node a Bbulldoze message is initiated. When the Bbulldoze message passes a node, this signals that the recording of GC-acquaintances is complete. The node sends *I-know-you* messages from each of its actors to each

GC-acquaintance of that actor. When an I-know-you message from actor A to actor B is received then actor A is added to the GC-inverse-acquaintances of actor B. A second forward and backward bulldoze phase is required to clear the network of I-know-you messages. This is initiated by the start node upon completion of the first backward bulldoze wave. When the second forward/backward bulldoze wave is complete, the start node sends a pre-GC-complete message to the root node. At this point, all *old* and *I-know-you* messages in the system have been cleared from the network and the snapshot information is recorded.

The backwards bulldoze messages are needed for both the recording of GC-acquaintances and GC-inverse-acquaintances, since the forward bulldoze only clears forwards links and there may be messages traversing backwards links that need to be recorded. To see this, note that after an object, say A, has received the pre-GC Fbulldoze message it can send only *new* messages. However, it may receive *old* messages from an actor H which has not yet received the pre-GC Fbulldoze message (see Figure 6).

-bb-error =

Fig. 6. The Bulldoze Wavefront in Progress: The bulldoze wavefront is halfway through the system. Although object A has started recording acquaintances and issues only *new* messages, it can receive and *old* messages from object H which has not yet received the bulldoze wave.

5 Detection of garbage

In this section we give a definition of reachability that takes into account the ability of an active object to become known by communicating its mail address. We then present an algorithm for marking objects that are reachable according to this definition. We conclude with a description of the behavior of the GC-root actor, which provides an overview of the complete HDGC algorithm.

5.1 Definition of reachability

The definition of reachable objects in an actor-based system is derived from the work of Kafura et al [KWN90]. The *root set* is a pre-defined set of actors from which reachability is traced. It includes actors referenced in the current computation state of the system (environment variables, control structures like stacks etc.). A GC snapshot of the system state determines a conservative approximation of the acquaintance relation. As mentioned in the introduction, in an actor computation, the transitive closure of this relation starting from the root set is not adequate to determine reachability, since an inverse acquaintance of a reachable actor may communicate its mail address at any point of time to its reachable acquaintance, thereby making itself reachable. Thus we cannot ignore the inverse acquaintances in determining reachability.

An actor which is currently processing messages or has messages pending in the network or in its mail queue is an *active actor*, otherwise, it is an *inactive actor*. An inactive actor which is not connected by the transitive closure of the inverse acquaintance relation to an active actor is a *permanently inactive actor*. An actor that is permanently inactive can never communicate its mail address and can be safely regarded as unreachable. The set of *reachable actors* is defined inductively as the least set such that:

- A root actor is a reachable actor.
- Every forward-acquaintance of a reachable actor is reachable.
- If an actor is reachable, then every inverse acquaintance of that actor which is not permanently inactive is reachable.

A *garbage actor* is an actor which is not reachable according to the above definition.

5.2 Distributed Scavenging

The algorithm for marking the reachable objects in the system, distributed scavenging, follows the inductive definition of reachability. To record the reachable objects, each object of the system has associated with it an *object-status* which may be *touched*, *untouched* or *suspended*. Touched objects are objects which are known to be reachable. Untouched objects have not yet been visited during GC. Objects that remain untouched at the completion of GC are unreachable. Suspended objects are inactive objects that are inverse acquaintances of reachable (touched) objects. If an active inverse acquaintance of such an object is found then the object will become touched. An object that remains suspended at the completion of GC is also unreachable. When GC is initiated all actors in the system have status untouched. Any actors created after the start of GC on a node are marked as touched.

The marking of objects is accomplished by propagation of GC and GC^{-1} messages from the roots and by backpropagation of GC-ack and GC^{-1}ack messages. It is initiated at the GC-root by sending GC messages to all the root actors. It is complete when GC-acks have been received by the GC-root from all root actors. The process of *touching* the accessible nodes is carried out in accordance with the *Principle of Monotonicity* which states that once an actor has been marked as *touched* during a GC, it cannot subsequently be untouched or suspended during the same GC. Below we summarize the actions caused by receipt of one of the GC marking messages.
A GC message from actor B to actor A is processed as follows:

- if A is touched then a GC-ack message is sent to B from A
- if A is untouched then A becomes touched, and
  - a GC message is sent to each GC-acquaintance of A,
  - a GC^{-1} message is sent to each GC-inverse-acquaintance of A,
  - When GC-ack/GC^{-1}ack messages have been received from all GC-acquaintances and GC-inverse-acquaintances, a GC-ack is sent to B from A.

– if A is suspended then A becomes touched, and
  • a GC message is sent to the GC-acquaintances of A,
  • When GC-ack messages have been received from all GC-acquaintances and outstanding GC^{-1}ack messages have been received from GC-inverse-acquaintances (to GC^{-1} messages sent at suspension time) then a GC-ack is sent to B from A.

A GC^{-1} message from actor B to actor A is processed as follows:

– if A is touched then a GC^{-1}ack is sent to B from A
– if A is untouched then
  • if A is active then A becomes touched, and proceeds as in the GC message case,
  • if A is inactive, then A becomes suspended and sends GC^{-1} messages to its GC-inverse-acquaintances. When GC^{-1}ack messages have been received from all GC-inverse-acquaintances, a GC^{-1}ack is sent to B from A.
– if A is suspended then it remains suspended and sends a GC^{-1}ack to B

This basic distributed scavenging algorithm can be adapted to provide a generational version by extending Ungar's Generation Scavenging scheme [Ung84]. A tag field associated with every actor which encodes the generation to which the actor belongs. When a GC is called, the generation bits in the tag field of accessible objects are altered. This is logically equivalent to moving the object from one generation to another. The *copy-count* bits, also a part of the tag field, are used to implement a tenuring policy and are incremented whenever the object survives a GC. When this count reaches a threshold value, the object is tenured from ScavengeSpace to Oldspace.

5.3 Behavior of a GC-root actor

An overall view of HDGC is given by describing the behavior of the GC-root actor. The GC-root actor remembers whether or not a GC is currently in progress. We summarize below the actions of the GC-root actor for each message it can receive.

– GC-initiate: This can come from any node wishing to initiate a GC. If a GC is not in progress, then a pre-GC broadcast is initiated at the start node and the GC-root remembers that a GC is in progress, otherwise the sender is informed that a GC is in progress.
– pre-GC-complete: This is sent by the start node when the second forward/backward bulldoze wave is complete. The distributed scavenge phase is initiated by sending GC messages to each root actor. When GC-acks have been received from all the root actors, a Local-Clear-Init broadcast is initiated at the start node. Local clearance is begun at each node when this broadcast is received.
– Local-Clear-Complete: This is sent by the finish node when the local clearance is complete. A post-GC broadcast is initiated at the start node.

– GC-complete: This is sent by the finish node when the post-GC broadcast is complete. Now each node marks all messages as old and all remaining actors as untouched [by flipping the interpretation of the tags]. The GC-root now remembers that GC is not in progress and is ready to initialize another GC.

6 Informal sketch of Correctness for HDGC

The correctness of the Hierarchical Distributed Garbage Collection Scheme is expressed by the following four theorems. The first two represent safety properties and the last two represent liveness properties.

Theorem 1. *A non-garbage actor will not be collected by the distributed garbage collection algorithm.*

Theorem 2. *The user program progresses as normal without any semantic interference with the distributed garbage collection algorithm.*

Theorem 3. *The HDGC scheme terminates for every execution.*

Theorem 4. *Every garbage object will eventually be collected.*

To establish these theorems we assume that a GC is initiated only under the following conditions.
Initial Conditions:

– *All actors in the system are untouched*
– *Messages in the system are of one kind – Old messages*

We recall the properties of actors and the underlying network that we have assumed.

1. There are a finite number of actors in the system.
2. Along a single link in the network messages are communicated in a FIFO fashion.
3. Message routing is progress-only in the sense described in section 2.
4. The mutator cooperates with the collector. Any new actors created during GC are created as touched actors, and any new messages created during GC are tagged as new.
5. The mutator does not interfere with the collector. The mutator does not modify data used during GC — the GC-acquaintances and GC-inverse-acquaintances of an actor, an actors active status and other GC status information, or a messages old/new tag.
6. A garbage actor can never become non-garbage.

We have not specified the details of how a node carries out it local clearance but we make certain requirements. Namely, that only untouched or suspended objects on a node are collected, and that local clearance at a node terminates. The correctness theorems follow from the HDGC step lemmas and GC invariant lemmas stated below. A rigorous proof of these lemmas is beyond the scope of this paper and will appear in a forthcoming publication.

6.1 HDGC Step Lemmas

The following lemmas express the crucial properties of each of the steps of the HDGC algorithm. For informal proofs of these lemmas, see [Ven91]. Recall that the *GC snapshot* consists of the GC-acquaintances, GC-inverse-acquaintances, and active status for each actor in the system. This information together with the root set determines a consistent global view of the reachability relation for the purposes of the GC.

Lemma 1. [*i*]. *The pre-GC step terminates*
[*ii*]. *At the end of pre-GC, all messages in the network are new and all objects existing prior to initiation of GC are marked untouched.*
[*iii*]. *At the end of pre-GC, the GC snapshot is a consistent distributed snapshot of the acquaintance relation relative to the start-of-GC time.*

Lemma 2. [*i*]. *The Distributed Scavenge phase marks all objects that are reachable according to the GC snapshot as* **touched**.
[*ii*]. *The DistributedScavenge phase marks all objects that are unreachable according to the GC snapshot as* **untouched** *or* **suspended**.
[*iii*]. *The Distributed Scavenge phase terminates.*

Lemma 3. *The Local-Clear-Initiation terminates and local clearance is initiated on every node in the system.*

Lemma 4. *The Local-Clear step terminates.*

Lemma 5. *The termination of GC is correctly detected and all the nodes in the system are informed of the same.*

Lemma 6. [*i*]. *The GC snapshot persists through out the duration of a given GC.*
[*ii*]. *The touching process is monotonic, i.e., once an actor has been marked as* touched *during a GC, it cannot subsequently be untouched or suspended during the same GC.*
[*iii*]. *Only one GC can be active in the system at a point in time.*

7 Conclusions and future work

In this paper, we have proposed a novel algorithm for garbage collection in scalable distributed systems of active objects called *hierarchical distributed garbage collection*. An informal sketch of the proof of correctness of HDGC has been outlined. A formal proof of correctness will appear in a forthcoming paper. To formalize the proof of a distributed garbage collection algorithm, we formally express the GC process as a transition relation and show that the possible computations of the system satisfy the step lemmas and that these in turn imply the desired correctness properties. The key concept for our formalization is to classify actors into object-level (application) actors and meta-level (system) actors. Meta-level actors can access information about object-level actors that other object-level actors cannot access. In particular, they can modify fields in the data structures representing object-level actors such as status, tags, mailqueue,

acquaintances, and behavior. Some meta-level actors simply serve as resource managers for a node. This provides encapsulation of the resource management facilities, and allows us to deal with system management and application management within a single unified framework—the actor model.

Any mechanism for efficient GC in a large system must be conservative. Generational storage management techniques are conservative and they exploit characteristic reference patterns observed in many applications [Ung84]; we therefore believe that they are well-suited to machines with large numbers of processing elements. As we avoid physically moving objects across generations, this scheme also turns out to be less error prone because interprocessor management of forwarding pointers can get very complex and frustrating. In actor based systems, GC involves more than data deallocation. An actor is a basic entity within which behavior (code), communication information and task processing information is embedded. When an actor is deleted, all resource management responsibilities associated with an actor disappear. Memory management in Actors is more than a data management facility, it is a process management facility as well.

What we have avoided in this paper is a detailed discussion of optimizations to the HDGC scheme. A consideration of various deficiencies of the this scheme has revealed some optimizations which can reduce the time and space overheads encountered in synchronization, name translation and bookkeeping. In addition to possible optimizations, this research has also brought to the surface many interesting issues. Compaction of memory to obtain locality, static analysis for optimal actor allocation and placement, lifetime analysis, and extensions of the HDGC algorithm to exhibit fault tolerance and real-time behavior are a few. We believe that the ability to design efficient, scalable, concurrent systems does not lie in esoteric programming paradigms and architectures that are difficult to comprehend. It lies in representing applications as well classified, intuitive specifications and organizing hardware resources to render flexible and manageable concurrency using natural strategies such as *hierarchical resource management.*

References

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8-3:323–364, June 1977.

[KWN90] Dennis Kafura, Doug Washabaugh, and Jeff Nelson. Garbage collection of actorrs. In Norman Meyrowitz, editor, *1990 ECOOP/OOPSLA Proceedings*, pages 126–134, Ottawa, Canada, October 1990. ACM Press.

[LQP92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage Collecting the World. In *Nineteenth Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 39–50, 1992.

[SGP90] Marc Shapiro, Olivier Gruber, and David Plainfosse. A garbage detection protocol for a realistic distributed object-support system. Technical Report 1320, INRIA, November 1990.

[Ung84] David M. Ungar. Generation scavenging - a non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments*, pages 157–167. Pittsburgh, PA, April 1984.

[Ven91] Nalini Venkatasubramanian. Hierarchical garbage collection in scalable distributed systems. Master's thesis, University of Illinois, Urbana-Champaign, Dept. of Computer Science, Urbana, IL, forthcoming 1991.