Distributed Garbage Collection for Mobile Actor Systems: The Pseudo Root Approach

Wei-Jen Wang and Carlos A. Varela

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, USA
{wangw5, cvarela}@cs.rpi.edu
http://wcl.cs.rpi.edu/

Abstract. Automatic distributed garbage collection (GC) gives abstraction to grid application development, promoting code quality and improving resource management. Unreachability of *active objects* or *actors* from the root set is not a sufficient condition to collect actor garbage, making passive object GC algorithms unsafe when directly used on actor systems. In practical actor languages, all actors have references to the root set since they can interact with users, *e.g.*, through standard input or output streams. Based on this observation, we introduce *pseudo roots*: a dynamic set of actors that can be viewed as the root set. Pseudo roots use protected (undeletable) references to ensure that no actors are erroneously collected even with messages in transit. Following this idea, we introduce a new direction of actor GC, and demonstrate it by developing a distributed GC framework. The framework can thus be used for automatic life time management of mobile reactive processes with unordered asynchronous communication. This report is an extended version of [42]. It provides more information about how we built a distributed garbage collector with the help of the pseudo root approach. It also shows experimental results for local GC.

1 Introduction

Large applications running on the grid, or on the internet, require runtime reconfigurability for better performance, *e.g.*, relocating application sub-components to improve locality without affecting the semantics of the distributed system. A runtime reconfigurable distributed system can be easily defined by the actor model of computation [3, 13]. The actor model provides a unit of encapsulation for a thread of control along with internal state. An actor is either *unblocked* or *blocked*. It is unblocked if it is processing a message or has messages in its message box, and it is blocked otherwise. Communication between actors is purely asynchronous: non-blocking and non-First-In-First-Out (non-FIFO). However, communication is guaranteed: all messages are eventually and fairly delivered. In response to an incoming message, an actor can use its thread of control to modify its encapsulated internal state, send messages to other actors, create actors, or migrate to another host.

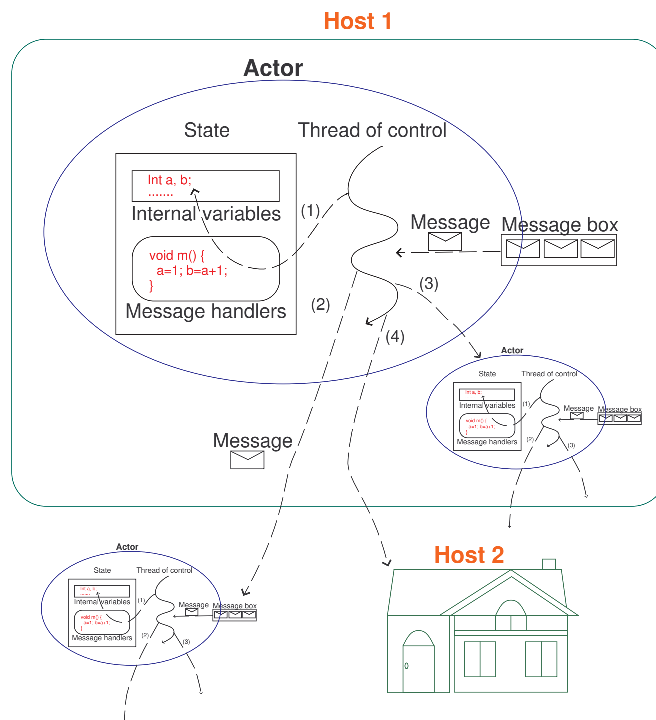

Fig. 1. An actor is a reactive entity which communicates with others by asynchronous non-FIFO messages. In response to an incoming message, it can use its thread of control to 1) modify its internal state, 2) send messages to other actors, 3) create actors, or 4) migrate to another host.

Many programming languages have partial or total support for actor semantics, such as SALSA [38], ABCL [46], THAL [18], Erlang [4], E [1], and Nomadic Pict [44]. Some libraries also support actor creation and use in standard object-oriented languages, such as the Actor Foundry [28] for Java, Broadway [35] for C++, and Actalk [8] for Smalltalk. In designing these languages or systems, memory reuse becomes an important issue to support dynamic data structures — such as linked lists. Automatic garbage collection is the key to enable memory reuse and to reduce programmers' efforts on their error-prone manual memory management.

The problem of distributed garbage collection (GC) is difficult because of: 1) information distribution, 2) lack of a global clock, 3) concurrent activities, and 4) possible failures of the network or computing nodes. These factors complicate detection of a consistent global state of a distributed system. Comparing to object-oriented systems, a pure actor system demands automatic GC as well, even more, because of its distributed, mobile, and resource-consuming nature. Actor GC is traditionally considered as a harder problem than passive object GC because of two additional difficulties to overcome:

1. Simply following the references from the root set of actors does not work in the actor GC model. Figure 2 explains the difference between the actor garbage collection model and the passive object GC model.
2. Unordered asynchronous message delivery complicates the actor garbage collection problem. Most existing algorithms cannot tolerate out-of-order messages.

Fig. 2. Actor 3, 4, and 8 are live because they can potentially send messages to the root. Object 3, 4, and 8 are garbage because they are not reachable from the root.

Previous distributed GC algorithms (including actor GC algorithms) rely on First-In-First-Out (FIFO) communication which simplifies detection of a consistent global state. A distributed object GC algorithm either adopts: 1) a lightweight reference counting/listing approach which cannot collect distributed mutually referenced data structures (*cycles*), 2) a trace-based approach which requires a consistent state of a distributed system, or 3) a hybrid approach [2].

In actor-oriented programming languages, an actor must be able to access resources which are encapsulated in service actors. To access a resource, an actor requires a reference to it. This implies that actors keep persistent references to some special service actors — such as the file system service and the standard output service. Furthermore, an actor can explicitly create references to public services. For instance, an actor can dynamically convert a string into a reference to communicate with a service actor, analogous to accessing a web service by a web browser using a URL.

Actor mobility is another new challenge to overcome. The concept of *in-transit* actors complicates the design of actor communication — locality of actors can change, which means even simulated FIFO communication with message redelivery is impractical, or at least limits concurrency by unnecessarily waiting for message redelivery. FIFO communication is an assumption of existing distributed GC algorithms. For instance, distributed reference counting algorithms demand FIFO communication to ensure that a reference-deletion system message does not precede any application messages.

This research differs from previous actor GC models by introducing: 1) asynchronous, unordered message delivery of both application messages and system messages, 2) resource access rights, and 3) actor mobility.

Outline of This Paper

The remainder of the paper is organized as follows: In Section 2 we give the definition of garbage actors. In Section 3 we propose the pseudo root approach — a mobile actor garbage collection model for distributed actor-oriented programming languages. In Section 4 we present an implementation for the proposed actor garbage collection model. In Section 5 we provide a concurrent, snapshot-based global actor garbage collector to collect distributed cyclic garbage. In Section 6 we show the experimental results. In Section 7 we discuss related work. Section 8 contains concluding remarks and future work.

2 Garbage in Actor Systems

The definition of actor garbage comes from the idea of whether an actor is doing meaningful computation. Meaningful computation is defined as having the ability to communicate with any of the *root actors*, that is, to access any resource or public service. The widely used definition of live actors is described in [17]. Conceptually, an actor is live if it is a root or it can either potentially: 1) receive messages from the root actors or 2) send messages to the root actors. The set of actor garbage is then defined as the complement of the set of live actors. To formally describe our new actor GC model, we introduce the following definitions:

- **Blocked actor**: An actor is blocked if it has no pending messages in its message box, nor any message being processed. Otherwise it is unblocked.

- **Reference**: A reference indicates an address of an actor. Actor A can only send messages to Actor B if A has a reference pointing to B.
- **Inverse reference**: An inverse reference is a conceptual reference in the counter-direction of an existing reference.
- **Acquaintance**: Let Actor A have a reference pointing to Actor B. B is an acquaintance of A, and A is an inverse acquaintance of B.
- **Root actor**: An actor is a root actor if it encapsulates a resource, or if it is a public service — such as I/O devices, web services, and databases.

The original definition of live actors is denotational because it uses the concept of "potential" message delivery and reception. To make it more operational, we use the term "*potentially live*" [10] to define live actors.

- **Potentially live actors**:
  - Every unblocked actor and root actor is potentially live.
  - Every acquaintance of a potentially live actor is potentially live.
- **Live actors**:
  - A root actor is live.
  - Every acquaintance of a live actor is live.
  - Every potentially live, inverse acquaintance of a live actor is live.

3 The Pseudo Root Approach

The pseudo root approach is based on the *live unblocked actor principle* — a principle which says every unblocked actor should be treated as a live actor. Every practical actor programming language design abides by this principle. With the principle, we integrate message delivery and reference passing into reference graph representation — *sender pseudo roots* and *protected references*. The pseudo root approach together with *imprecise inverse reference listing* enables the use of unordered, asynchronous communication.

The Live Unblocked Actor Principle Without program analysis techniques, the ability of an actor to access resources provided by an actor-oriented programming language implies explicit reference creation to access service actors. The ability to access local service actors (e.g. the standard output) and explicit reference creation to public service actors make the following statement true: "*every actor has persistent references to root actors*". This statement is important because it changes the meaning of actor GC, making actor GC similar to passive object GC. It leads to the *live unblocked actor principle*, which says every unblocked actor is live. The live unblocked actor principle is easy to prove. Since each unblocked actor is: 1) an inverse acquaintance of the root actors and 2) defined as potentially live, it is live according to the definition of actor GC.

With the live unblocked actor principle, every unblocked actor can be viewed as a root. Liveness of blocked actors depends on the transitive reachability from unblocked actors and root actors. If a blocked actor is transitively reachable

from an unblocked actor or a root actor, it is defined as potentially live. With persistent root references, such potentially live, blocked actors are live because they are inverse acquaintances of some root actors. This idea leads to the core concept of *pseudo root* actor GC.

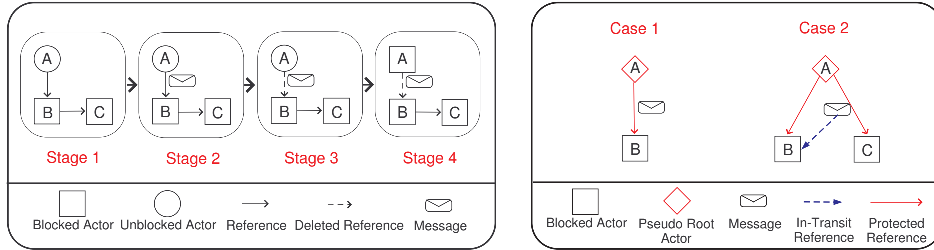

Fig. 3. The left side of the figure shows a possible race condition of mutation and message passing. The right side of the figure illustrates both kinds of sender pseudo root actors.

Pseudo Root Actor Garbage Collection The pseudo root actor GC starts actor garbage collection by identifying some live (not necessarily root) or even garbage actors as pseudo roots. There are three kinds of pseudo root actors: 1) root actors, 2) unblocked actors, and 3) *sender pseudo root actors*. The sender pseudo root actor refers to an actor which has sent a message and the message has not yet been received. The goal of sender pseudo roots is to prevent erroneous garbage collection of actors, either targets of in-transit messages or whose references are part of in-transit messages. A sender pseudo root always contains at least one protected reference — a reference that has been used to deliver messages which are currently in transit, or a reference to represent an actor referenced by an in-transit message — which we call an *in-transit reference*. A protected reference cannot be deleted until the message sender knows the in-transit messages have been received correctly.

Asynchronous communication introduces the following problem (see the left side of Figure 3): application messages from Actor A to Actor B can be in transit, but the reference held by Actor A can be removed. Stage 3 shows that Actor B and C are likely to be erroneously reclaimed, while Stage 4 shows that all of the actors are possibly erroneously reclaimed. Our solution is to temporarily keep the reference to Actor B undeleted and identify Actor A as live (Case 1 of the right side of Figure 3). This approach guarantees liveness of Actor B by tracing from Actor A. Actor A is named the *sender pseudo root* because it has an in-transit message to Actor B and it is not a real root. Furthermore, it can

be garbage but cannot be collected. The reference from A to B is protected and A is considered live until A knows that the in-transit message is delivered.

To prevent erroneous GC, actors pointed by in-transit references must unconditionally remain live until the receiver receives the message. A similar solution can be re-used to guarantee the liveness of the referenced actor: the sender becomes a sender pseudo root and keeps the reference to the referenced actor undeleted (Case 2).

Using pseudo roots, *the persistent references to roots can be ignored*. Figure 4 illustrates an example of the mapping of pseudo root actor GC. We can now safely ignore: 1) dynamic creation of references to public services and 2) persistent references to local services.

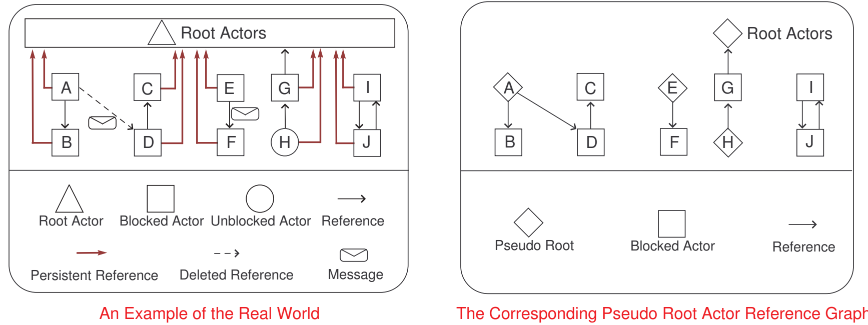

Fig. 4. An example of pseudo root actor garbage collection which maps the real state of the given system to a pseudo root actor reference graph.

Imprecise Inverse Reference Listing In a distributed environment, an inter-node referenced actor must be considered live from the perspective of local GC. To know whether an actor is inter-node referenced, each actor should maintain inverse references to indicate if it is inter-node referenced. This approach usually refers to *reference listing*. Maintaining precise inverse references in an asynchronous way is performance-expensive. Fortunately, imprecise inverse references are acceptable if all inter-node referenced actors can be identified as live — inter-node referenced actors can be pseudo root actors (*global pseudo roots*), or transtively reachable from some local pseudo root actors to guarantee their liveness.

4 Implementation of the Pseudo Root Approach

To implement the proposed pseudo root approach, we propose the *actor garbage detection protocol*. The actor garbage detection protocol, implemented as part of the SALSA programming language [38, 45], consists of four sub-protocols — the *asynchronous ACK protocol*, the *reference passing protocol*, the *migration protocol*, and the *reference deletion protocol*. Messages are divided into two categories — the *application messages* which require asynchronous acknowledgements, and the *system messages* that will not trigger any asynchronous acknowledgement.

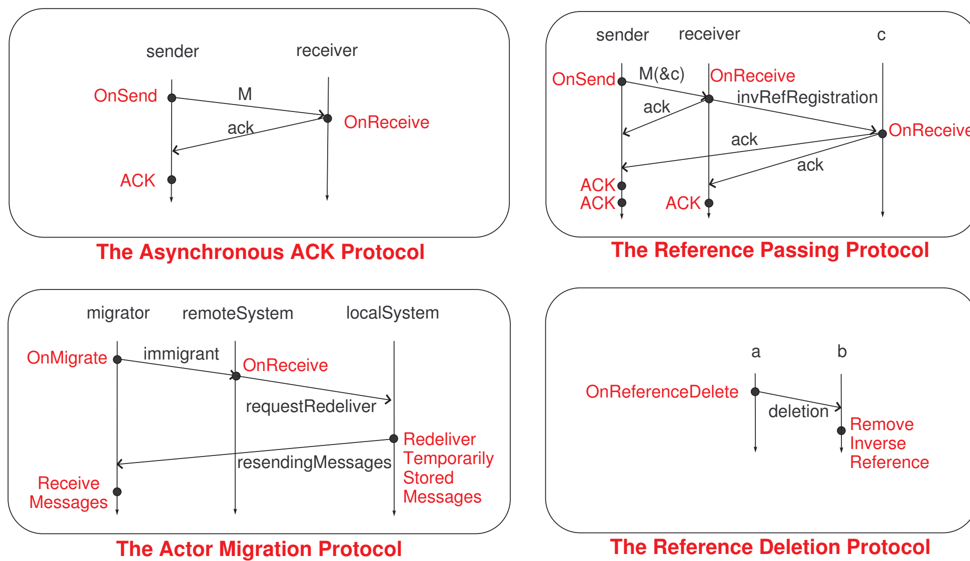

Fig. 5. The actor garbage detection sub-protocols.

The Asynchronous ACK Protocol The asynchronous ACK protocol is designed to help identifying sender pseudo roots. Each reference maintains a counter, `count`, for expected acknowledgements. A reference can be deleted only if its expected acknowledgement count is zero. An actor is a sender pseudo root if the total expected acknowledgements of its references are greater than zero. The protocol is shown in the left upper part of Figure 5, in which actor `sender` sends a message to actor `receiver`. The event handler `OnSend` is triggered when an application message is sent; the event handler `OnReceive` is invoked when a message is received. If a message to receive requires an acknowledgement, the event handler `OnReceive` will generate an acknowledgement to the message sender.

The message handler `ACK` is asynchronously executed by an actor to decrease the expected acknowledgement count of the reference to actor `receiver` held by actor `sender`. With the asynchronous ACK protocol, the garbage collector can identify sender pseudo roots and protected references from the perspective of implementation:

- A *sender pseudo root* is one whose total expected acknowledgement count of its references is greater than zero.
- A *protected reference* is one whose expected acknowledgement count is greater than zero. A protected reference cannot be deleted.

The Reference Passing Protocol The reference passing protocol specifies how to build inverse references in an asynchronous manner. A typical scenario of reference passing is to send a message `M` containing a reference to `c`, from `sender` to `receiver`. The reference `(sender,receiver)` and the reference `(sender,c)` are protected at the beginning by increasing their expected acknowledgement counts. Then `sender` sends the application message `M` to `receiver`. Right after `receiver` has received the message, it generates an application message `invRefRegistration` to `c` to register the inverse reference of `(receiver,c)` in `c`. A special acknowledgement from `c` to `sender` is then sent to decrease the count of the protected reference `(sender,c)`. Making `invRefRegistration` an application message is to ensure that reference deletion of reference `(receiver,c)` always happens after `c` has built the corresponding inverse reference. The protocol is shown in the right upper side of Figure 5.

The Migration Protocol Implementation of the migration protocol requires assistance from two special actors, `remoteSystem` at a remote computing node, and `localSystem` at the local computing node. An actor migrates by encoding itself into a message, and then delivers the message to `remoteSystem`. During this period, messages to the migrating actor are stored at `localSystem`. After migration, `localSystem` delivers the temporarily stored messages to the migrated actor asynchronously. Every migrating actor becomes a pseudo root by increasing the expected acknowledgement count of its self reference. The migrating actor decreases the expected acknowledgement count of its self reference when it receives the temporarily stored messages. The protocol is shown in the left lower side of Figure 5.

The Reference Deletion Protocol A reference can be deleted if it is not protected — its expected acknowledgement count must be zero. The deletion automatically creates a system message to the acquaintance of the actor deleting the reference to remove the inverse reference held by the acquaintance. The protocol is shown in the right lower side of Figure 5.

Safety of Actor Garbage Detection Protocol The safety of local actor GC in a distributed environment is guaranteed by the following invariants:

1. Let $x \neq y$. If Actor y is referenced by a non-pseudo-root actor x, actor y must have an inverse reference to Actor x.
2. If an actor is referenced by several pseudo roots, either it has at least one inverse reference to one of the pseudo roots, or it is a pseudo root.

The above two invariants together guarantee *the property of one-step back tracing safety*. The property says that if an actor is inter-node referenced, the actor either can be identified as a remotely dependent pseudo root by one-step back tracing through its registered inverse references, or is reachable from some local pseudo roots.

5 Collecting Distributed Cyclic Garbage Actors

Although local garbage collectors and the actor garbage detection protocol together can identify all local garbage and distributed acyclic garbage, they cannot identify distributed cyclic garbage — the kind of garbage that requires the global state of a system to determine whether or not an actor is garbage. The difficulty to obtain a coherent global state of a distributed system complicates distributed garbage collection. The cost of obtaining a precise global state of a mobile actor system is unacceptable and impractical. Fortunately, a precise coherent global state of a distributed mobile actor system is not necessary for distributed actor garbage collection. Instead, an approximate one is enough for actor garbage collection which only requires a stable set of garbage actors. with the help of the pseudo root approach, we propose the partial approximate snapshot algorithm which: 1) provides such an approximate snapshot for distributed actor garbage collection and 2) is fault tolerant.

5.1 The Concept of the Partial Approximate Snapshot

Distributed actor garbage collection requires a coherent actor reference graph to identify live actors. To obtain such an actor reference graph, existing actor garbage collection algorithms either use stop-the-world synchronization or FIFO (or simulated FIFO with timestamps) communication. The stop-the-world synchronization approach is expensive for concurrent or distributed systems. FIFO communication is a little better, but has to maintain the order of message reception which is expensive with actor migration. To avoid the use of stop-the-world synchronization or FIFO communication, the partial approximate snapshot algorithm is proposed to obtain a coherent actor reference graph.

The partial approximate snapshot algorithm comes from two different ideas: partial garbage collection and the approximate snapshot algorithm. Partial garbage collection divides a system into several regions and performs garbage collection in each region independently. Any failure at one region cannot affect results of other regions. Let t_1 be the time point to start the snapshot algorithm, t_2 be the time point to end the snapshot algorithm, and $t_1 < t_x < t_2$. The approximate snapshot is to maintain the minimal invariant of garbage collection: *the garbage*

set at t_x contains the same elements as the garbage set at t_1. The concept of the partial approximate snapshot algorithm is to take an approximate snapshot on a closed group of actors such that the snapshot has the minimal invariant of garbage collection.

Operation Restrictions To maintain reachability from roots during snapshot, reference deletion is not allowed or has to be recorded during snapshot. Incoming and outgoing inter-node references have to be preserved because they affect the definition of liveness from the perspective of local garbage collection. The state of an actor must be monitored during snapshot. If an actor has ever become a pseudo root, the actor is identified as a pseudo root in the snapshot.

Migration during garbage collection is supported by the proposed algorithm. If an actor is migrating, it is live because it is unblocked. Since it is live, it can be removed from the snapshot safely by adding the references it held to the set of incoming inter-node references. All newly created actors are excluded from the snapshot automatically. The partial approximate snapshot algorithm goes as follows:

1. Form a closed group of actors.
2. Monitor the state of each actor at the group. Preserve deleted references and incoming inter-group references of each actor at the group. If an actor is migrating, remove it from the group by adding the references it held to the incoming inter-group references.
3. Start to record the actor reference graph.
4. Maintain the partial approximate snapshot for actor garbage collection.

Merging Snapshots To maximize possible concurrency, each computing node can take a partial approximate snapshot independently and then a global mechanism is used to merge these local snapshots into a large one. The merging operation is meaningful if the merged snapshot maintains the minimal invariant of garbage collection, which requires two simple global synchronization steps:

1. Ask each computing node to form a closed group of actors, to monitor the state of actors, and to preserve deleted references and incoming inter-group references.
2. Wait until each computing node responds or timeout happens.
3. Ask each computing node to record its local snapshot.
4. Wait until each computing node responds or timeout happens.
5. Re-identify inter-group incoming references and inter-group outgoing references.

The first four steps are together called the *global partial approximate snapshot procedure*, while the last step is called the *partial approximate snapshot merging operation*.

5.2 The Distributed Actor Garbage Collector

We have implemented the partial approximate snapshot algorithm as an optional logically centralized service. Garbage except distributed cyclic garbage still can be reclaimed without it. The global collector gathers synchronized local partial approximate snapshots to form a global snapshot. Consequently, the global collector identifies garbage of the global partial approximate snapshot, and then requests the local collectors to reclaim the global garbage. No actor can be erroneously reclaimed even if a local garbage collector fails to deliver its local snapshot to the service, but actors referenced by those at the failed node cannot be reclaimed even if they are garbage. Actors work concurrently with the local garbage collectors and the centralized global actor garbage collector, but they have to pay performance penalty during garbage collection — the mutation operations such as migration and reference deletion have to be monitored. The algorithm has five steps:

1. Coordinate a global partial approximate snapshot,
2. Obtain all synchronized local partial approximate snapshots,
3. Merge the snapshots into a global snapshot,
4. Perform pseudo root garbage collection, and
5. Notify local collectors for the garbage list.

Figure 6 illustrates the concept of the algorithm.

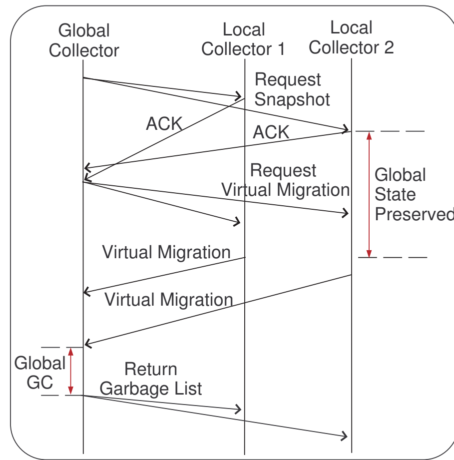

Fig. 6. A cycle of distributed actor garbage collection by the proposed centralized global actor garbage collector that manages two local actor garbage collectors. Virtual migration refers to transmitting a local snapshot to the server.

6 Experimental Results

Major concerns on the performance of distributed applications are mostly the degree of parallelism and the application execution time. In this section, we use several types of applications to measure the impact of the proposed actor garbage collection mechanism in terms of real execution time, CPU time, and overhead percentage.

Local Benchmark Application

Three different benchmark applications are designed to measure the impact of the local garbage collection mechanism. These applications are *Fibonacci number (Fib)*, *N queens number (NQ)*, and *Matrix multiplication (MX)*. Each application is executed at a dual-processor Solaris machine. The applications are described as follows:

- Fibonacci number (Fib): Fibonacci number, abbreviated as *Fib*, takes one argument and then computes the Fibonacci number. It is a coordinated tree-structure computation, with sequential execution of number 29 or 30 at each leaf.
- N queens number (NQ): N queens number, abbreviated as *NQ*, takes one argument to calculate the total solutions of the N queens problem by creating $(N-1)\times(N-2)$ actors for parallel execution and one actor for coordination.
- Matrix multiplication (MX): Matrix multiplication, abbreviated as *MX*, requires two files for application arguments, each of which contains a matrix. The application calculates one matrix multiplication of the given two matrices.

Distributed Benchmark Application

Each distributed benchmark application is executed at four dual-processor Solaris machines, and initialized at another dual-processor Solaris machine. These machines are connected by Ethernet. The distributed garbage collector is activated every 20 seconds. The benchmark applications are described as follows:

- Distributed Fibonacci number with locality (Dfibl): *Dfibl* optimizes the number of inter-node messages by locating four sub-computing-trees at each computing node.
- Distributed Fibonacci number without locality (Dfibn): *Dfibn* distributes the actors in a breadth-first-search manner.
- Distributed N queens number (DNQ): *DNQ* equally distributes the actors to four computing nodes.
- Distributed Matrix multiplication (DMX): *DMX* divides the first input matrix into four sub-matrices, sends the sub-matrices and the second matrix to four computing nodes, performs one matrix multiplication operation, and then merges the data at the computing node that initializes the computation.

Garbage Collection Mechanism to Measure

To show the impact of garbage collection, the measurement for actor garbage collection uses three different mechanisms: *No GC*, *GDP Only*, and *Local GC*. The mechanisms are described as follows:

- No GC: Data structures and algorithms for garbage collection are not used.
- GDP Only: The local garbage collector is not activated. Only the garbage detection protocol is used.
- Local GC: The local garbage collector is activated every two seconds or in case of insufficient memory.

The local experimental results are shown in Table 1, and the distributed results are in Table 2. Each result of a benchmark application is the average of ten execution times.

Table 1. Measurement on a Dual-processor Solaris Machine (measured in seconds).

Mechanism	Application(Argument)/Number of Actors					
	Fib(38) /109	Fib(41) /465	NQ(13) /133	NQ(15) /183	MX(100^2) /3	MX(150^2) /3
No GC (Real/CPU)	1.70/2.57	5.09/8.66	1.84/2.16	6.72/11.58	1.84/1.93	2.63/2.84
GDP ONLY (Real/CPU)	2.14/3.07	6.20/10.21	2.42/2.55	7.63/12.84	2.16/2.24	2.97/3.17
Local GC (Real/CPU)	2.13/3.08	6.63/10.54	2.55/2.79	7.97/13.30	2.16/2.24	3.03/3.20
GDP Overhead (Real/CPU)	25%/20%	22%/18%	32%/18%	14%/11%	17%/16%	13%/11%
LGC Overhead (Real/CPU)	25%/20%	30%/22%	39%/29%	19%/15%	17%/16%	15%/13%

Table 2. Measurement in a distributed environment. Real time is measured in seconds.

Mechanism	Application(Argument)/Number of Actors							
	Dfibl(39) /177	Dfibl(42) /753	Dfibn(39) /177	Dfibn(42) /753	DNQ(16) /211	DNQ(18) /273	DMX(100^2) /5	DMX(150^2) /5
No GC	1.722	3.974	3.216	8.527	13.120	426.151	6.165	39.011
DGC	2.091	4.957	3.761	9.940	17.531	461.757	6.715	38.955
DGC Overhead	21%	25%	17%	17%	34%	8%	9%	0%

7 Related Work

Distributed garbage collection has been studied for decades. The area of distributed passive object collection algorithms can be roughly divided into two categories — the reference counting (or listing) based algorithms and the indirect distributed garbage collection algorithms. The reference counting (or listing) based algorithms cannot collect distributed cyclic garbage — such as [22,

6, 27, 29, 5, 43, 33, 34]. They are similar to the proposed actor garbage detection protocol but they tend to be more synchronous — all of them rely on First-In-First-Out (FIFO) communication or timestamp based FIFO (simulated FIFO) communication, and some of them are even totally synchronous by using remote-procedure-call [6]. Since actor communication is defined as asynchronous, unordered, and message-driven, these algorithms cannot be reused directly by actor systems.

There are various indirect distributed garbage collection algorithms for passive object systems. The most important feature of these algorithms is that they collect at least some distributed cyclic garbage. Hughes' algorithm [15] uses global timestamp propagation from roots which requires long pause time for garbage collection and is very sensitive to failures. Liskov et al. [23, 19] present a client-server based algorithm which requires every client (local collector) to report inter-node references to a server. Vestal [41] assumes assistance from acyclic reference counting and tries to virtually delete a reference to see whether or not an object is garbage, which cannot detect all cycles. Maheshwari et al. [24, 25] and Le Fessant [21] propose heuristics based algorithms which use the minimal number of inter-node references from a root to suspect some objects as garbage and then verifies the suspects. Lang et al. [20] propose a group-based tracing algorithm with the help of a reference listing protocol and local garbage collection to collect garbage hierarchically, which does not support migration and must stop the mutators while garbage collection is performing. Rodrigues et al. [31] present a dynamically partitioning approach which starts from suspecting an object as garbage, traces from it, and then forms a group for global garbage collection. During global garbage collection, mutators must be suspended for local live object marking. Overlapped partition can occur which either causes deadlocks, or no work can be done. The algorithm proposed by Veiga et al. [39] is also heuristics based, and it uses asynchronous local snapshots to identify global garbage. Any change to the snapshots has to be updated by local mutators, forcing current global garbage collection to quit. Hudson et al. [14] propose a generational collector where the address space of each computing node is divided into several disjoint blocks (cars), and cars are grouped together into several distributed trains. Each train represents a generation of objects, and forms a ring structure for distributed management. Objects can only move from an older generation car to a younger generation car, and the oldest car is eventually inspected. A car/train can be disposed of if there are no incoming inter-car/inter-train references to it. Blackburn et al. [7] suggest a methodology to derive a distributed garbage collection algorithm from an existing distributed termination detection algorithm [12, 11, 26], in which the distributed garbage collection algorithm developers must design another algorithm to guarantee a consistent global state. All of the above algorithms cannot be reused directly in actor systems because actors and passive objects are different in nature.

The definition of garbage actors is different from the perspective of passive object garbage collection. For instance, the marking phase of passive object garbage collection has only two choices, the depth-first-search and the breadth-

first-search. Marking algorithms for actor garbage collection are relatively various, including Push-Pull, Is-Black by Kafura et al. [17], Dickman's graph partition merging algorithm by Dickman [10], and the actor transformation algorithm by Vardhan and Agha [36, 37]. Most distributed actor garbage collection algorithms are snapshot based due to the autonomous nature of actors. The algorithm proposed by Kafura et al. [16] uses the Chandy-Lamport snapshot algorithm [9] to determine a precise global state, which is expensive and requires FIFO communication to flush communication channels. Venkatasubramanian et al. [40] assume a two-dimensional grid network topology, and the algorithm requires FIFO communication to flush communication channels. Puaut's algorithm [30] is client-server based, and requires each computing node to maintain a timestamp vector to simulate a global clock. It makes a message become larger while the number of computing nodes increases. Vardhan's algorithm [36] transforms each local actor reference graph into a passive object reference graph, and uses Schelvis' algorithm [32] for global garbage collection. It assumes: 1) First-In-First-Out (FIFO) communication, 2) temporarily suspending the message sender which is waiting for an acknowledgement from the message receiver, and 3) periodically performs stop-the-world garbage collection. All existing actor garbage collection algorithms violate the asynchronous, unordered assumption of actor communication, and all of them do not support the concept of actor migration.

8 Conclusion and Future Work

In this paper, we have redefined garbage actors to make the definition more operational. We also introduced the concept of pseudo roots, making actor GC easier to understand and to implement. The most important contribution of this paper is *the actor garbage collection framework for actor-oriented programming languages*. Implementation of actor GC is available since version 1.0 of the SALSA programming language [45, 38]. Unlike existing actor GC algorithms, the proposed framework does not require FIFO communication or stop-the-world synchronization. Furthermore, it supports actor migration and it works concurrently with mutation operations. This feature reduces interruption of users' applications. The proposed logically centralized global garbage collector is safe in the case of failures since it does not collect actors which are referenced by unknown actors.

Future research focuses on the idea of resource access restrictions, which is part of distributed resource management. By applying the resource access restrictions to actors, the live unblocked actor principle is no longer true — not every actor has references to the root actors. Another direction of this research is to modify the partitioning based passive object GC algorithms to increase scalability. Last but not least, testing the GC algorithms on real-world applications running on large-scale distributed environments is necessary to further evaluate their scalability and performance.

Acknowledgements

We would like to acknowledge the National Science Foundation (NSF CAREER Award No. CNS-0448407) for partial support for this research.

References

1. E. http://www.erights.org/.
2. S. E. Abdullahi and A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330–373, 1998.
3. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
4. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.
5. D. I. Bevan. Distributed garbage collection using reference counting. In J. W. de Bakker, L. Nijman, and P. C. Treleaven, editors, *PARLE'87 Parallel Architectures and Languages Europe*, volume 258/259 of *Lecture Notes in Computer Science*, pages 176–187, Eindhoven, The Netherlands, June 1987. Springer-Verlag.
6. A. Birrell, D. Evers, G. Nelson, S. Owicki, and E. Wobber. Distributed garbage collection for network objects. Technical Report 116, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, Dec. 1993.
7. S. M. Blackburn, R. L. Hudson, R. Morrison, J. E. B. Moss, D. S. Munro, and J. Zigman. Starting with termination: a methodology for building distributed garbage collection algorithms. *Aust. Comput. Sci. Commun.*, 23(1):20–28, 2001.
8. J.-P. Briot. Actalk: a testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP'89)*, pages 109–129. Cambridge University Press, 1989.
9. K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.
10. P. Dickman. Incremental, distributed orphan detection and actor garbage collection using graph partitioning and Euler cycles. In O. Babaoglu and K. Marzullo, editors, *Tenth International Workshop on Distributed Algorithms WDAG'96*, volume 1151 of *Lecture Notes in Computer Science*, Bologna, Oct. 1996. Springer-Verlag.
11. E. W. Dijkstra and C. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, Aug. 1980.
12. N. Francez. Distributed termination. *ACM Trans. Program. Lang. Syst.*, 2(1):42–55, 1980.
13. Hewitt, C. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8-3:323–364, June 1977.
14. R. L. Hudson, R. Morrison, J. E. B. Moss, and D. S. Munro. Garbage Collecting the World: One Car at a Time. In *OOPSLA'97*, 1997.
15. R. J. M. Hughes. A distributed garbage collection algorithm. In J.-P. Jouannaud, editor, *Record of the 1985 Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 256–272, Nancy, France, Sept. 1985. Springer-Verlag.
16. D. Kafura, M. Mukherji, and D. M. Washabaugh. Concurrent and distributed garbage collection of active objects. *IEEE Trans. On Parallrl and Distributed Systems*, 6(4), April 1995.

17. D. Kafura, D. Washabaugh, and J. Nelson. Garbage collection of actors. In *OOPSLA'90 ACM Conference on Object-Oriented Systems, Languages and Applications*, pages 126–134. ACM Press, October 1990.
18. W. Kim. *THAL: An Actor System for Efficient and Scalable Concurrent Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1997.
19. R. Ladin and B. Liskov. Garbage collection of a distributed heap. In *International Conference on Distributed Computing Systems*, Yokohama, June 1992.
20. B. Lang, C. Queinnec, and J. Piquer. Garbage collecting the world. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 39–50. ACM Press, 1992.
21. F. Le Fessant. Detecting distributed cycles of garbage in large-scale systems. In *Principles of Distributed Computing (PODC)*, Rhodes Island, Aug. 2001.
22. C.-W. Lermen and D. Maurer. A protocol for distributed reference counting. In *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, ACM SIGPLAN Notices, pages 343–350, Cambridge, MA, Aug. 1986. ACM Press.
23. B. Liskov and R. Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In J. Halpern, editor, *Proceedings of the Fifth Annual ACM Symposium on the Principles on Distributed Computing*, pages 29–39, Calgary, Aug. 1986. ACM Press.
24. U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by controlled migration. In *Proceedings of PODC'95 Principles of Distributed Computing*, 1995.
25. U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by back tracing. In *Proceedings of PODC'97 Principles of Distributed Computing*, pages 239–248, Santa Barbara, CA, 1997. ACM Press.
26. J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *J. Syst. Softw.*, 43(3):207–221, 1998.
27. L. Moreau. Tree rerooting in distributed garbage collection: Implementation and performance evaluation. *Higher-Order and Symbolic Computation*, 14(4):357–386, 2001.
28. Open Systems Lab. The Actor Foundry: A Java-based Actor Programming Environment, 1998. Work in Progress. http://osl.cs.uiuc.edu/foundry/.
29. J. M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In Aarts et al., editors, *PARLE'91 Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 1991. Springer-Verlag.
30. I. Puaut. A distributed garbage collector for active objects. In *Proceedings of the Ninth Annual Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 113–128. ACM Press, 1994.
31. H. C. C. D. Rodrigues and R. E. Jones. A cyclic distributed garbage collector for Network Objects. In O. Babaoglu and K. Marzullo, editors, *Tenth International Workshop on Distributed Algorithms WDAG'96*, volume 1151 of *Lecture Notes in Computer Science*, pages 123–140, Bologna, Oct. 1996. Springer-Verlag.
32. M. Schelvis. Incremental distribution of timestamp packets — a new approach to distributed garbage collection. *ACM SIGPLAN Notices*, 24(10):37–48, 1989.
33. M. Shapiro. A fault-tolerant, scalable, low-overhead distributed garbage collection protocol. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, Pisa, Sept. 1991.
34. M. Shapiro, P. Dickman, and D. Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapports de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, Nov. 1992.

35. D. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996. TR UIUCDCS-R-96-1950.
36. A. Vardhan. Distributed garbage collection of active objects: A transformation and its applications to java programming. master's thesis. Master's thesis, University of Illinois at Urbana Champaign, Urbana Champaig, Illinois, 1998.
37. A. Vardhan and G. Agha. Using passive object garbage collection algorithms. In D. Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 106–113, Berlin, June 2002. ACM Press.
38. C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, Dec. 2001.
39. L. Veiga and P. Ferreira. Asynchronous complete distributed garbage collection. In O. Babaoglu and K. Marzullo, editors, *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, Colorado, USA, Apr. 2005.
40. N. Venkatasubramanian, G. Agha, and C. Talcott. Scalable distributed garbage collection for systems of active objects. In *Proceedings of International Workshop on Memory Management, volume 637 of Lecture Notes in Computer Science*. Springer-Verlag, 1992.
41. S. C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming*. PhD thesis, University of Washington, Seattle, WA, 1987.
42. W. Wang and C. A. Varela. Distributed garbage collection for mobile actor systems: The pseudo root approach. In *GPC'06*. Springer-Verlag, 2006.
43. P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In J. W. de Bakker, L. Nijman, and P. C. Treleaven, editors, *PARLE'87 Parallel Architectures and Languages Europe*, volume 258/259 of *Lecture Notes in Computer Science*, pages 432–443, Eindhoven, The Netherlands, June 1987. Springer-Verlag.
44. P. Wojciechowski and P. Sewell. Nomadic pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2), 2000.
45. Worldwide Computing Laboratory. The SALSA Programming Language, 2002. Work in Progress. http://www.cs.rpi.edu/wwc/salsa/.
46. A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*. MIT Press, Cambridge, Mass., 1990.