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://www.cs.rpi.edu/wwc/

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.

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 [2, 8]. 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.

Many programming languages have partial or total support for actor semantics, such as SALSA, ABCL, THAL, Erlang, E, and Nomadic Pict. Some libraries also support actor creation and use in object-oriented languages, such as the Actor Foundry for Java, Broadway for C++, and Actalk 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 1 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.

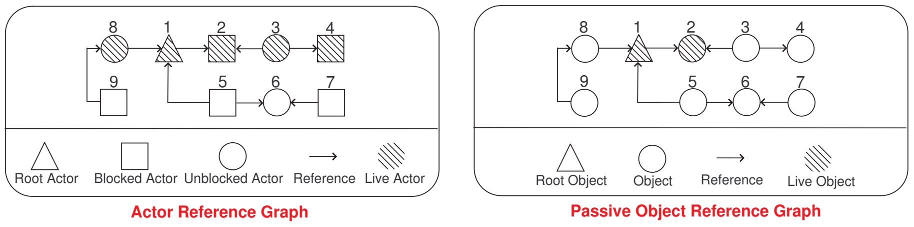

Fig. 1. Actor 3, 4, and 8 are live because they can potentially send messages to the root. Object 3, 4, and 8 are garbage because they are not reachable from the root.

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 [1].

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 re-delivery 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.

The remainder of the paper is organized as follows: In Section 2 we give the definition of garbage in actor systems. 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 of the proposed actor GC model. In Section 5 we briefly describe a concurrent, snapshot-based global actor garbage collector to collect distributed cyclic garbage. In Section 6 we show 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 [12]. 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*" [7] 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.

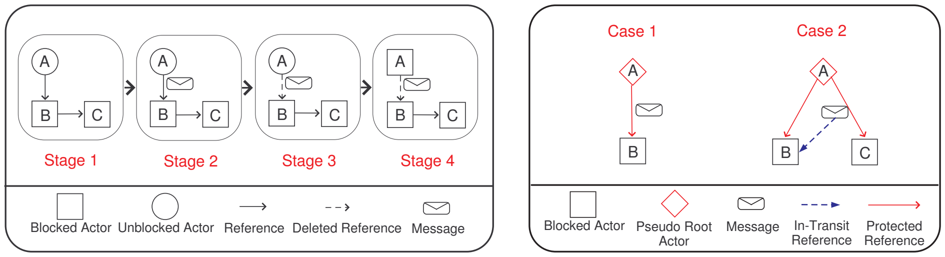

Fig. 2. The left side of the figure shows a possible race condition of mutation and message passing. The right side of the figure illustrates both kinds of sender pseudo root actors.

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 2): 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 2). 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 3 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.

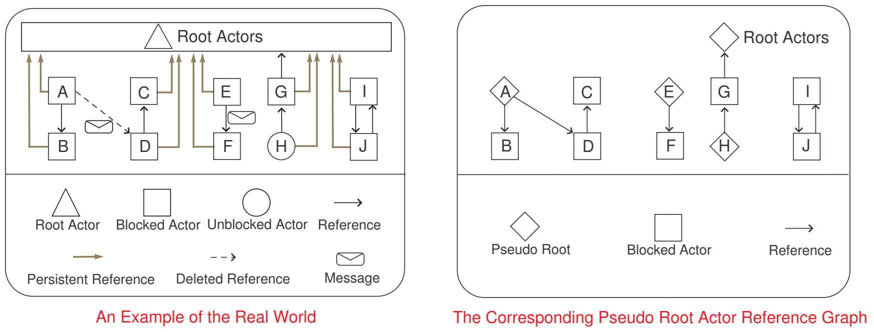

Fig. 3. An example of pseudo root actor garbage collection which maps the real state of the given system to a pseudo root actor reference graph.

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 [28, 34], 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.

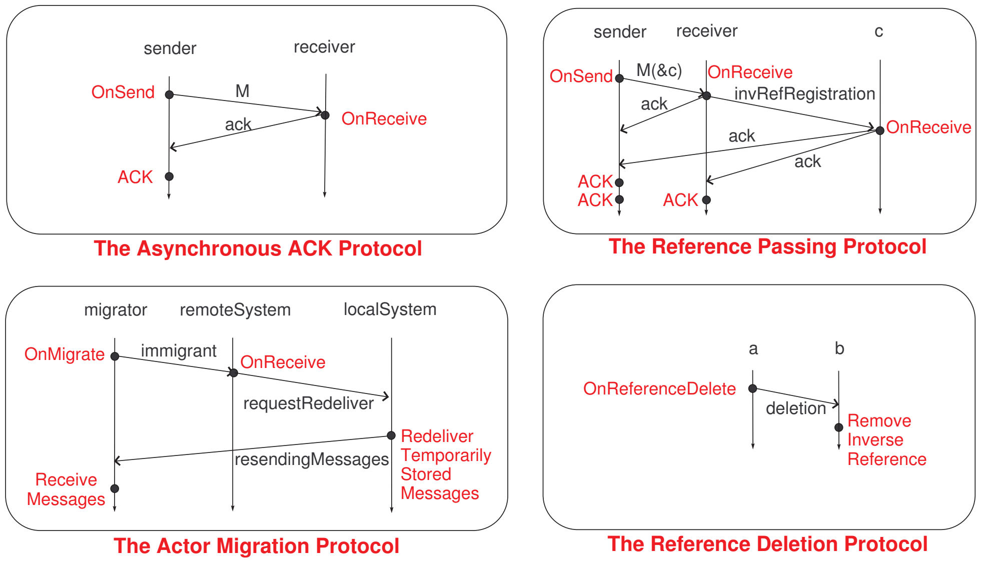

Fig. 4. The actor garbage detection sub-protocols.

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 4, 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 4.

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 4.

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 4.

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 Actor Garbage

In order to collect distributed cyclic garbage, we need to obtain a consistent global view of the system. With the help of the pseudo root approach, we have devised a logically centralized global garbage collector, which is concurrent (does not stop applications), asynchronous, and non-FIFO. The global collector triggers distributed GC periodically, decides which computing nodes to include, asks each computing node to return a local snapshot, merges the snapshots, identifies garbage, and then notifies each computing node of the garbage list. Before snapshots are returned, deleted references or inverse references are preserved for consistency. Migrating or migrated actors are removed from the group of GC, and actors referenced by them are identified as pseudo roots directly. Actors that have been unblocked or are currently unblocked during garbage collection are also pseudo roots. Details of the global collector can be found in [32].

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 GC mechanism in terms of real execution time and overhead percentage. The results are shown in Table 1, and each result of a benchmark application is the average of ten execution times. To show the impact of GC, the measurement for actor GC uses two different mechanisms: *No GC* and *DGC*. "No GC" means nothing is used; "DGC" means local garbage collectors are activated every two seconds or in case of insufficient memory, and distributed GC starts every 20 seconds. The results of local GC experiments can be found in [32].

Distributed Benchmark Application

Each distributed benchmark application is executed at four dual-processor Solaris machines. These machines are connected by Ethernet. 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 at 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.

Table 1. The distributed garbage collection experimental results (measured in seconds).

	Application(Argument)/Number of Actors							
Mechanism	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 (Real)	1.722	3.974	3.216	8.527	13.120	426.151	6.165	39.011
DGC (Real)	2.091	4.957	3.761	9.940	17.531	461.757	6.715	38.955
DGC Overhead (Real)	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 [16, 4, 20, 21, 3, 33, 25]. 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 [4]. Since actor communication is asynchronous and unordered, 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 [10] uses global timestamp propagation from roots which is very sensitive to failures. Liskov et al. [13] present a client-server based algorithm which requires every local collector to report inter-node references to a server. Vestal's algorithm [31] tries to virtually delete a reference to see whether or not an object is garbage. Maheshwari et al. [17, 18] and Le Fessant [15] propose heuristics based algorithms to suspect some objects as garbage and then to verify the suspects. Lang et al. [14] propose a group-based tracing algorithm to collect garbage hierarchically. Rodrigues et al. [23] present a dynamically partitioning approach to form a group of objects for global garbage collection. Veiga et al. [29] propose a heuristics and snapshot based algorithm, in which any change to the snapshots may force current global garbage collection to quit. Hudson et al. [9] 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. A car/train can be disposed of if there are no incoming inter-car/inter-train references to it. Blackburn et al. [5] suggest a methodology to derive a distributed garbage collection algorithm from an existing distributed termination detection algorithm [19], 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.

Marking algorithms for actor garbage collection are relatively various, including Push-Pull, Is-Black by Kafura et al. [12], Dickman's algorithm [7], and the actor transformation algorithm by Vardhan and Agha [26, 27]. Most distributed actor garbage collection algorithms are snapshot based. The algorithm proposed by Kafura et al. [11] uses the Chandy-Lamport snapshot algorithm [6] to determine a precise global state, which is expensive and requires FIFO communication to flush communication channels. Venkatasubramanian et al. [30] assume a two-dimensional grid network topology, and the algorithm also requires FIFO communication to flush communication channels. Puaut's algorithm [22] is client-server based, and requires each computing node to maintain a timestamp vector to simulate a global clock. Vardhan's algorithm [26] transforms each local actor reference graph into a passive object reference graph, and uses Schelvis' algorithm [24] for global garbage collection. It assumes: FIFO communication, and 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 [34, 28]. 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. S. E. Abdullahi and A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330–373, 1998.
2. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
3. D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE'87*, volume 258/259 of *Lecture Notes in Computer Science*, pages 176–187, Eindhoven, The Netherlands, June 1987. Springer-Verlag.
4. 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.
5. 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.
6. K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.
7. P. Dickman. Incremental, distributed orphan detection and actor garbage collection using graph partitioning and Euler cycles. In *WDAG'96*, volume 1151 of *Lecture Notes in Computer Science*, Bologna, Oct. 1996. Springer-Verlag.
8. Hewitt, C. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, June 1977.
9. R. L. Hudson, R. Morrison, J. E. B. Moss, and D. S. Munro. Garbage collecting the world: One car at a time. *SIGPLAN Not.*, 32(10):162–175, 1997.
10. J. Hughes. A distributed garbage collection algorithm. In *Record of the 1985 Conference on Functional Programming and Computer Architecture*, volume 201 of *LNCS*, pages 256–272, Nancy, France, Sept. 1985. Springer-Verlag.
11. D. Kafura, M. Mukherji, and D. Washabaugh. Concurrent and distributed garbage collection of active objects. *IEEE TPDS*, 6(4), April 1995.
12. D. Kafura, D. Washabaugh, and J. Nelson. Garbage collection of actors. In *OOP-SLA'90*, pages 126–134. ACM Press, October 1990.
13. R. Ladin and B. Liskov. Garbage collection of a distributed heap. In *International Conference on Distributed Computing Systems*, Yokohama, June 1992.
14. B. Lang, C. Queinnec, and J. Piquer. Garbage collecting the world. In *POPL'92*, pages 39–50. ACM Press, 1992.
15. F. Le Fessant. Detecting distributed cycles of garbage in large-scale systems. In *Principles of Distributed Computing (PODC)*, Rhodes Island, Aug. 2001.
16. C. Lermen and D. Maurer. A protocol for distributed reference counting. In *ACM Symposium on Lisp and Functional Programming*, ACM SIGPLAN Notices, pages 343–350, Cambridge, MA, Aug. 1986. ACM Press.
17. U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by controlled migration. In *PODC'95*, 1995.
18. U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by back tracing. In *PODC'97*, pages 239–248, Santa Barbara, CA, 1997. ACM Press.
19. J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *J. Syst. Softw.*, 43(3):207–221, 1998.
20. L. Moreau. Tree rerooting in distributed garbage collection: Implementation and performance evaluation. *Higher-Order and Symbolic Computation*, 14(4):357–386, 2001.

21. J. M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *PARLE'91*, volume 505 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 1991. Springer-Verlag.
22. I. Puaut. A distributed garbage collector for active objects. In *OOPSLA'94*, pages 113–128. ACM Press, 1994.
23. H. Rodrigues and R. Jones. A cyclic distributed garbage collector for Network Objects. In *WDAG'96*, volume 1151 of *Lecture Notes in Computer Science*, pages 123–140, Bologna, Oct. 1996. Springer-Verlag.
24. M. Schelvis. Incremental distribution of timestamp packets — a new approach to distributed garbage collection. *ACM SIGPLAN Notices*, 24(10):37–48, 1989.
25. M. Shapiro, P. Dickman, and D. Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapports de Recherche 1799, INRIA, Nov. 1992.
26. A. Vardhan. Distributed garbage collection of active objects: A transformation and its applications to java programming. Master's thesis, UIUC, Urbana Champaig, Illinois, 1998.
27. A. Vardhan and G. Agha. Using passive object garbage collection algorithms. In *ISMM'02*, ACM SIGPLAN Notices, pages 106–113, Berlin, June 2002. ACM Press.
28. C. A. 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.
29. L. Veiga and P. Ferreira. Asynchronous complete distributed garbage collection. In O. Babaoglu and K. Marzullo, editors, *IPDPS 2005*, Denver, Colorado, USA, Apr. 2005.
30. N. Venkatasubramanian, G. Agha, and C. Talcott. Scalable distributed garbage collection for systems of active objects. In *IWMM'92*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
31. S. C. Vestal. *Garbage collection: An exercise in distributed, fault-tolerant programming*. PhD thesis, University of Washington, Seattle, WA, 1987.
32. W. Wang and C. A. Varela. Distributed garbage collection for mobile actor systems: The pseudo root approach. Technical Report 06-04, Dept. of Computer Science, R.P.I., Feb. 2006. Extended Version of the GPC'06 Paper.
33. P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87*, volume 258/259 of *Lecture Notes in Computer Science*, pages 432–443, Eindhoven, The Netherlands, June 1987. Springer-Verlag.
34. Worldwide Computing Laboratory. The SALSA Programming Language, 2002. Work in Progress. http://www.cs.rpi.edu/wwc/salsa/.