A Non-blocking Snapshot Algorithm for Distributed Garbage Collection of Mobile Active Objects

Wei-Jen Wang and Carlos A. Varela

Abstract

Distributed actor garbage collection differs from distributed object garbage collection in that it needs to consider in-transit message detection, unordered message reception, and actor migration. In this paper, we propose a new snapshot-based distributed actor garbage collection algorithm. The algorithm does not require First-In-First-Out or blocking communication, nor message logging. Furthermore, actor migration is allowed while capturing global snapshots and partial snapshots can be safely used to collect garbage, therefore not requiring comprehensive cooperation among all computing nodes. These features make it unique in the area of distributed garbage collection. We formally prove the following safety and conditional liveness properties of the algorithm: 1) the garbage in a global snapshot, created by composing several local snapshots, remains the same from the beginning to the end of the global snapshot algorithm, and 2) garbage is eventually collected if the global garbage collection algorithm is periodically activated and every blocked actor is always captured before a global garbage collection phase is triggered.

Index Terms

D.4.7.b distributed systems, D.4.2.c garbage collection, D.3.4.f memory management, D.4.m miscellaneous, distributed snapshot algorithms, actors, active objects, mobile objects.

I. Introduction

Automatic memory management, usually called garbage collection (GC), is a technique to reclaim unreferenced memory space. It raises the level of programming by preventing programmers' error-prone manual memory space manipulations. The problem of garbage collection is to find the complement set of transitively reachable objects from the root set, where the root set refers to the set of objects directly available to access by threads of control. Garbage collection in a distributed environment is difficult — no global clock, no centralized memory, inherent concurrency, and possible failures of computing nodes and the network. These factors complicate detection of a consistent global state of a distributed system, required for finding all distributed garbage.

A. *Garbage Collection in Actor Systems*

Recently, mobile active objects are becoming more and more important because of the uprising development of grid and pervasive computing. A mobile active object can migrate to another computing host, which enables runtime application reconfiguration. Runtime reconfiguration is important to improve computing quality because the state of resources is always changing. For example, an active object can migrate from a busy computing host to another idle host to achieve dynamic load balancing. Another example is that an active object can migrate from a cell phone to another computing host such as a laptop in case of low battery.

The most widely adopted model for reasoning about active objects is *the actor model of computation* [1], [12]. An actor system is comprised of uniquely named, autonomous reactive active objects, namely the *actors*. Communication of actors is *purely asynchronous, guaranteed, and fair* — they always send messages in a *non-blocking* manner. Even though messages may arrive unordered, they are eventually delivered. Furthermore, an actor can either send messages to its *acquaintances*, to whom it has explicit references, or some predefined special actors such as the output service. An actor consists of an encapsulated thread of control, its internal state, and a message box to buffer incoming messages. An actor

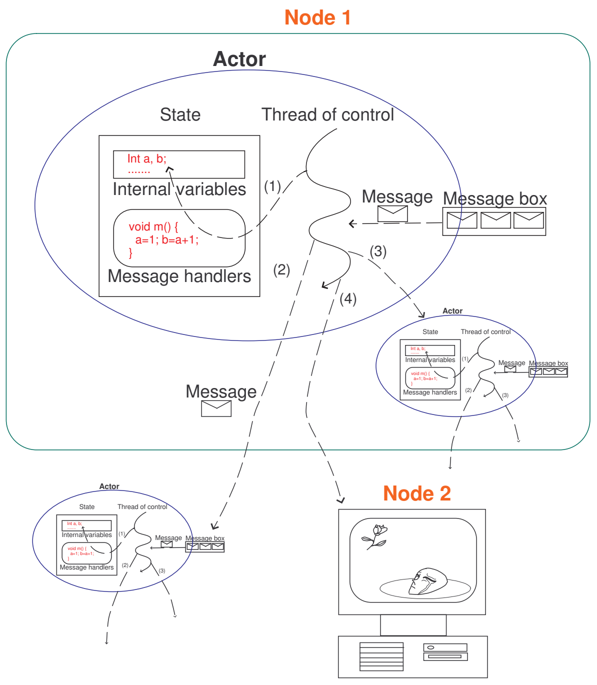

Fig. 1. An actor is a reactive entity which communicates with others by asynchronous messages in a non-blocking manner. In response to an incoming message, it can use its thread of control to 1) modify its internal state, 2) send messages to other actors, 3) create actors, or 4) migrate to another computing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

is *unblocked* if it is processing a message or has messages in its message box; otherwise it is *blocked* waiting for incoming messages. An actor buffers incoming messages in its message box, and its thread of control keeps retrieving and processing them. 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 computing node (see Figure 1).

The actor model of computation provides a natural semantics for distributed systems because it has several preferable features: purely asynchronous communication (non-blocking and unordered communication), and state encapsulation with an internal thread. With these features, a distributed system can be easily reconfigured at runtime — migration of an actor is as easy as migration of its encapsulated state and its message box, without worrying about state corruption. Comparing to the synchronous *object method invocation model* (or the *Remote Procedure Call model*), the actor model is more concurrent because actors do not block for any return value. Additionally, state encapsulation facilitates dynamic load balancing [10] by reallocating the actors during computation. Many programming languages have partial or total support for actor semantics, such as SALSA [36], ABCL [44], THAL [17], and Erlang [2]. Some libraries also support the actor model of computation, such as Actor Foundry [26] for Java, Broadway [33] for C++, IOS for MPI library [11], and Actalk [6] for Smalltalk.

Distributed garbage collection has been developed for decades. However, literature for *active object garbage collection* is scarce. The problem of actor garbage collection is different in nature from passive object garbage collection. Actor computations must be able to directly or indirectly provide results to a set of predefined output devices, such as consoles, printers, file systems, or databases. Actors may also obtain information from input devices, such as keyboards or sensors, to output results if they can communicate with them. As a result, the input or output devices are represented by a *root set of actors*. Actors that can directly or indirectly communicate with the root set of actors are live. Figure 2 illustrates a key difference between actor garbage collection and passive object garbage collection.

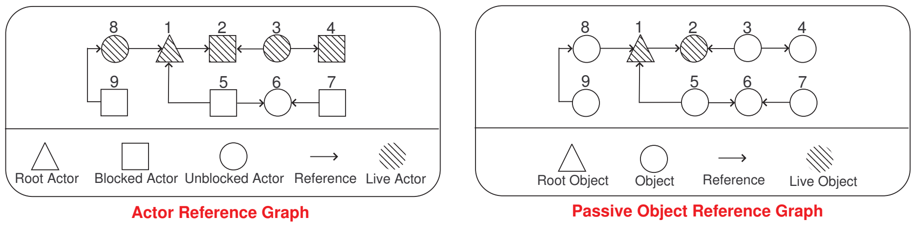

Fig. 2. Actors 3, 4, and 8 are live because they can potentially send messages to the root. Objects 3, 4, and 8 are garbage because they are not reachable from the root.

B. Snapshot Algorithm in Actor Garbage Collection

A snapshot algorithm executes in parallel with applications to obtain the configuration of a system, also referred to as the *snapshot*. In a distributed system, active entities, such as MPI processes and actors, can send messages to affect each other, which means a snapshot algorithm must take care of both in-transit messages and each local state of the system. A snapshot must be causally consistent, but different variations of snapshot algorithms have different requirements of causal consistency. Actor garbage collection requires that no live actors can be collected (*safety*), and garbage is eventually collected (*liveness*). Sometimes the ability to collect garbage one chunk at a time (incrementality) is important when a system is large or it needs interactivity. Snapshots are used in garbage collection [15], [28], [37], [38], as well as in many other areas, such as distributed termination detection [24] and distributed checkpointing for execution rollback [7], [30].

Snapshot-based algorithms greatly fit in the need of actor systems because they have less interference with applications than blocking-based algorithms do. A snapshot-based actor garbage collection algorithm first obtains a consistent global state, and then identifies garbage over the global state. Snapshot-based algorithms can detect a causally consistent snapshot, which means each snapshot is causally independent from each other. However, when a snapshot-based algorithm is applied to a mobile actor system, it encounters difficulties detecting in-transit actors because these actors are being transmitted over the network and are therefore hard to detect. The end result is that global snapshots may be missing actors or may contain duplicate actors; both views inconsistent with the global system state. In-transit messages cause three additional difficulties to detect actor garbage in a distributed environment. Firstly, in-transit messages may carry references that can affect the global state. Secondly, a reference to an actor can be deleted, while a message to it is in transit because actor communication is non-blocking. This implies that there may exist a blocked actor such that nobody knows about it but it is live. In other words, following references to detect garbage is not reliable in actor systems. Thirdly, actors communicate with each other by asynchronous and non-First-In-First-Out (non-FIFO) messages, which violates preconditions of most existing garbage collection algorithms and snapshot algorithms. Using existing snapshot algorithms for distributed garbage collection tends to be more restrictive — FIFO message delivery, blocking communication, huge space for message and state logging, or a distributed clock must be used. Furthermore, they rely on completely cooperative computing nodes to maintain safety of garbage collection.

Incrementality can be provided by using a non-blocking, non-FIFO actor reference listing algorithm to figure out whether an actor is remotely referenced, either from in-transit actors or actors at remote computing nodes. We have developed such an actor reference listing algorithm as part of the *pseudo-root approach* [40], in order to help building a consistent partial snapshot for actor garbage collection.

The key to make our snapshot algorithm flexible is to utilize some features of garbage collection. Actor garbage collection does not care about all the states that a system maintains. Its interest is the referential relationship of actors, which can be represented by a conceptual actor reference graph. The most important thing is that the snapshot does not need to be strictly causally consistent if an actor garbage collection algorithm can guarantee liveness and correctness. The trick to achieve this goal is to use an alternative approach to make an actor live, that is, the state of a root is equal to the state of an actor which is referenced by a root. Starting from this idea, we introduce the pseudo-root approach, and then propose a partial snapshot algorithm for actor garbage collection, a snapshot recording model to reason about the behavior of the snapshot algorithm, and safety and liveness proofs for correctness of the snapshot algorithm.

Our approach is unique and novel since it is purely asynchronous — non-blocking and non-First-In-First-Out, which means that it preserves the nature of the actor model. Furthermore, it can support hierarchical garbage collection [19] in a non-blocking manner even when some actors are migrating.

Outline of This Paper

The remainder of the paper is organized as follows: In Section II we give the definition of garbage actors and the problem of the distributed snapshot for actor garbage collection. In Section III we describe the general idea of the snapshot algorithm and the pseudo-root approach. In Section IV we present a computing model and proofs for the proposed snapshot algorithm. In Section V we discuss related work. Section VI contains concluding remarks and future work.

II. Problem Definition

The section introduces the definition of actor garbage collection, the computing environment for mobile actor systems, and the assumptions for the computing environments. We also address the causally inconsistent snapshot problem caused by asynchronous message delivery.

A. Live Actor Definition

The definition of actor garbage comes from whether or not an actor can possibly perform meaningful computations, which is defined as having the ability to communicate with any of pre-defined *root actors*. For example, an actor is considered live if it can possibly send messages to any output resource (*i.e.* a printer) or public service (*i.e.* a web service). Kafura et. al. have provided a definition of live actors [16], which is widely adopted in the literature. 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 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 actor a has a reference pointing to Actor b. The reference from a to b is denoted as $\overline{ab}$.
- **Inverse reference**: An *inverse reference* is a conceptual reference in the reverse direction of an existing reference. It is used by a garbage collection mechanism to figure out whether an actor is remotely referenced.
- **Acquaintance**: Let Actor a have a reference pointing to Actor b. Actor 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. We adopted Dickman's idea of potential live actors [9] to re-define live actors, as follows:

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

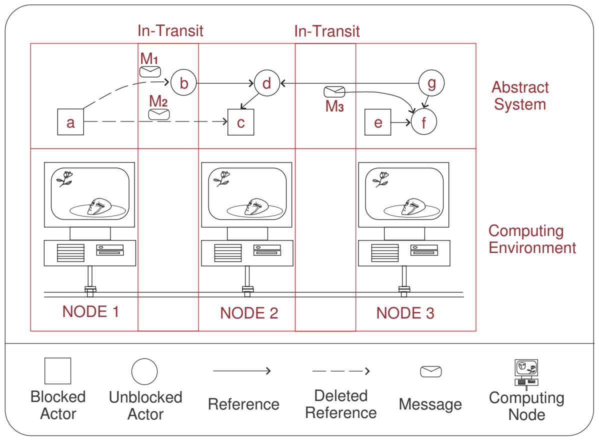

Fig. 3. This figure shows an example of a mobile actor system. Computing nodes 1, 2, and 3 are connected. Actor b is migrating; Actors a, c, and e are blocked; Actors d, f, g are unblocked. Messages M_1 and M_2 are in transit but their respective references $\overline{ab}$ and $\overline{ac}$ have been deleted; notice that references $\overline{ab}$ and $\overline{ac}$ had to exist before M_1 and M_2 were sent. Message M_3 with a reference to actor f is sent towards Actor d.

- A root actor is *live*.
- Every acquaintance of a live actor is *live*.
- Every potentially live, inverse acquaintance of a live actor is *live*.

B. The Distributed Mobile Actor Garbage Collection Problem

The problem of mobile actor garbage collection is related to how a distributed mobile actor system performs computing tasks. We assume that the grid or pervasive computing environment consists of computing nodes which provide CPU time and memory space for computations, while there is no specific topology assumption for how the nodes are connected. Liveness of the algorithm might be affected by permanent node failures, but safety is guaranteed even though some computing nodes may become uncooperative temporarily. Applications, composed of mobile actors, are running on the computing nodes. Mobile actor garbage collection in such computing environments can be viewed as follows: *Given a mobile actor system, identify actor garbage at some time point over a subset of the computing nodes*. We assume that local snapshots at single computing nodes are given and are consistent with the state at that node at the beginning of the global snapshot algorithm. We also make the assumption that every actor has a reference to itself.

Mutation operations are performed by actors to change the conceptual actor reference graph over the subset of nodes in consideration. They consist of 1) actor creation, 2) references passing/reception, 3) reference deletion, 4) migration, and 5) actor state transition from unblocked to blocked and vice versa. Because actor communication is asynchronous, mutation operations can occur out of order. For instance, an actor can send out a message and then remove the reference to the message target before the message is received. Furthermore, a migrating actor is temporarily difficult to detect because it is in transit. Figure 3 illustrates a mobile actor computing environment.

C. Live Unblocked Actor Principle

A practical actor-oriented programming language should provide the ability to access resources, or to communicate with predefined public services. An important assumption we make is that every actor has access rights to the root set. Without program analysis techniques, we must assume that every actor has persistent references to the root set. This precondition leads to the *live unblocked actor principle*, which says that every unblocked actor is live. The live unblocked actor principle is easy to prove. Since each

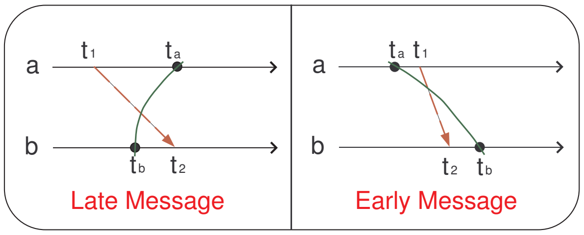

Fig. 4. Time lines to illustrate late and early messages. At the left side, Actor a sends a message to Actor b at t_1 and then its state is recorded at t_a; the state of Actor b is recorded at t_b and then it receives the message at t_2. At the right side, the state of Actor a is recorded at t_a and then it sends a message to Actor b at t_1; Actor b receives it at t_2 and then its state is recorded at t_b.

unblocked actor is: 1) an inverse acquaintance of the root actors and 2) defined as potentially live, it is live according to the live actor definition. The live unblocked actor principle makes actor garbage collection very similar to passive object garbage collection because every unblocked actor can be treated as a root actor directly without changing the meaning of actor garbage. With the live unblocked actor principle, every unblocked actor can be viewed as a root. Therefore, every potentially live actor is live because they can possibly receive messages from unblocked actors and then send messages to the root actors. This idea leads to the core concept of *pseudo-root actor garbage collection* [40], which is introduced in Sub-Section III-C. The pseudo-root approach can support incremental actor garbage collection (a chunk of garbage at a time) and handle temporarily uncooperative actors and computing nodes. We assume that actors migrate in response to a message requesting migration. Therefore, migrating actors are unblocked, and by the live unblocked actor principle are also live.

D. Causally Consistent Snapshot

A snapshot algorithm must guarantee the causal consistency of the obtained snapshot. The problem of causal consistency can be expressed by the order of message sending, message reception, and local state logging. Let Actor a send an application message at time t_1 to a remote actor b, and Actor b receive the message at time t_2. Let t_a and t_b be the time points when a snapshot is taken for a and b respectively. Note that $t_2 > t_1$ is always true because of the causal relationship of message sending. There are two kinds of inconsistent snapshots caused by different orders of message delivery (refer to Figure 4):

- **Late message (in-flight message)**: If $(t_1 < t_a) \wedge (t_b < t_2)$, the message is said to be *late*. A late message does not affect causal consistency because it is irrelevant to the recorded state of Actor b. It is only important to a system which needs to replay messages right after a global snapshot (*i.e.* system rollback upon failures).
- **Early message (inconsistent message)**: If $(t_a < t_1) \wedge (t_2 < t_b)$, the message is said to be *early*. It is causally inconsistent because a message produced by a future unrecorded state of Actor a, affects the recorded state of Actor b.

Mobile actor garbage collection must solve the early message problem. A snapshot-based algorithm is both safe and live if either the snapshot does not contain early messages, or early messages do not affect safety and liveness.

III. Non-blocking Snapshot Algorithm and Pseudo-Root Approach

Given a non-blocking, non-FIFO reference listing algorithm such as the *pseudo-root approach* [40], many actor garbage collection problems can be simplified:

1) In-transit messages and in-transit references are represented as part of the actor reference graph to guarantee safety and liveness. For instance, a message from Actor a to Actor b is represented as a reference from Live actor a to Actor b, and the relationship is detectable in Actor a.

2) Remotely referenced actors can be identified by using inverse references.
3) Actor garbage collection does not stop applications.

Unfortunately, such a reference listing algorithm cannot identify distributed mutually referenced actor garbage (distributed cycles). We propose a snapshot algorithm for distributed actor garbage collection to solve this problem. The fundamental idea is to put a partial collection of actors into a snapshot (the local actor reference graph) at each computing node, and then to keep watching the collection until the snapshot algorithm terminates. The snapshot may mutate whenever any actor belonging to the snapshot mutates. No new garbage is created in the snapshot by mutation operations, but applications do create new garbage which will only be detected at the next actor garbage collection phase. To generalize in one sentence, the goal of the snapshot algorithm is to maintain a superset actor reference graph `G1` of the real actor reference graph `G2` at the time that the snapshot algorithm begins, where *the set of pseudo-roots of* `G1` *is a superset of that of* `G2` *and the set of references of* `G1` *is also a superset of that of* `G2`. The proposed snapshot algorithm is obviously safe because the set of garbage of `G1` is a subset of that of `G2`, but it produces *floating garbage*. Floating garbage refers to actors which become garbage during a garbage collection phase, but cannot be detected in that phase. Any garbage collector that uses this approach cannot detect floating garbage of `G1`, but it can detect the floating garbage in the next garbage collection cycle because garbage cannot become live any longer.

The non-blocking snapshot algorithm consists of two parts: local state logging and global synchronization. Local state logging is performed by local garbage collectors. A global agent is assigned to initialize and to terminate the snapshot, where two global synchronizations are enough for a causally consistent global snapshot — one to trigger local state logging and the other one to terminate local state logging. Unlike other distributed snapshot-based algorithms, our algorithm does not require message logging. Instead, monitoring mutation operations is enough.

A. Local State Logging

Local state logging is triggered by a global synchronization agent, which requests the local garbage collector to form a closed group of actors and then starts to monitor mutation operations on that closed group. Newly created actors are automatically excluded from the closed group; migrating or migrated actors are segregated by the local snapshot procedure. Reference deletion is not logged because we want to ensure a live actor remains live by following the original path from the beginning to the end of the local state logging. The state logging procedure for local snapshot has to ensure that: 1) deleted references, including inverse references, are logged in the local snapshot, 2) migrating or migrated actors are segregated dynamically from the closed group and their acquaintances become remotely referenced. Figure 5 shows an example of how the local state logging works. The local state logging algorithm is modeled as a special actor which responds to a global garbage collection request from the global synchronization agent. Note that it does not stop any mutation operations, including migration.

B. Global Snapshot Synchronization

The global synchronization agent is devised to coordinate a meaningful global snapshot among several computing nodes. Since each computing node logs local state independently, some kind of global synchronization must be used to ensure that no early messages or migrating actors can be received before the state logging starts. This goal can be achieved by enforcing the participating local snapshots to have a common overlapping time range during local state logging. An overlapping time range also ensures that no actor can appear more than once at the participating local snapshots with the help of local state logging. Let the common overlapping time range start at time t_1 and finish later to become available for global snapshot merging. It is obvious that the set of garbage at each local snapshot is fixed after t_1. Our algorithm also guarantees that the set of global garbage in the global snapshot, combined by the participating local snapshots, is fixed after t_1. To prevent some kind of temporary failures from stopping global garbage collection, the synchronization agent can use a time-out to keep the global snapshot going. The pseudo-code is shown in Figure III-B and 6.

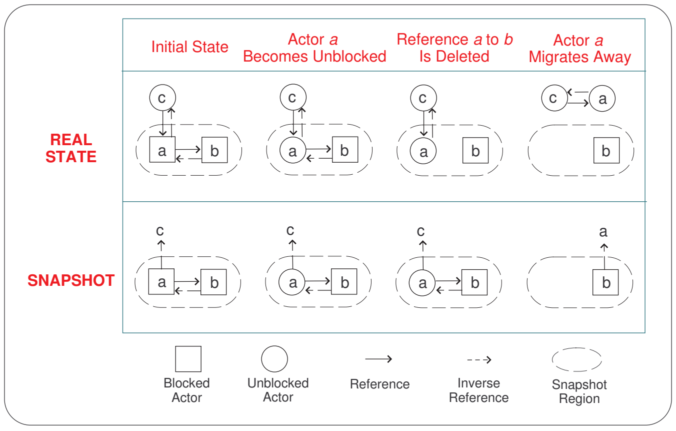

Fig. 5. This figure shows an example of local state logging. The upper part demonstrates the actor reference graph in the real world, while the lower part illustrates how local state logging works. At the beginning of local state logging, Actor a is referenced by Actor c; Actor b is referenced by Actor a. Actor a and Actor b are put in a closed group for state logging. At the second stage, Actor a becomes unblocked to execute something, and the snapshot should detect the event. At the third stage, Actor a deletes Reference $\overline{ab}$. Although Actor b becomes garbage at this stage, it is live in the local snapshot because it is reachable from a pseudo-root (unblocked) actor, Actor a. At the last stage, Actor a migrates away, and the local snapshot should reflect the fact that Actor a is missing. Meanwhile, all its acquaintances should become remotely referenced because the local snapshot must not produce new garbage. At this stage, no actor in the local snapshot is garbage. Actor a is not garbage either because it does not belong to the closed group.

```
Algorithm distributed_snapshot
 1. create a unique task number T
 2. for each computing node X do
 3.   asynchronously execute local_monitor(T)
 4.   //each may reply YES or NO
 5. wait until
 6.   1) every computing node has replied, or 2) timeout
 7. for each computing node X which replies YES do
 8.   asynchronously execute local_snapshot(T)
 9.   //each may reply OK or FAILED
10. wait until
11.   1) all computing nodes have replied OK, or 2) timeout
```

Fig. 6. The distributed snapshot algorithm. A meaningful global snapshot consists of the local snapshots of the computing nodes that reply 'OK'.

C. The Pseudo-Root Approach

Actor garbage collection is difficult even in a single computing node. For instance, incremental garbage collection under non-blocking and non-FIFO communication had not been solved until recently. Our previous research, the pseudo-root approach [40], provides a feasible solution for this problem. With the pseudo-root approach, the proposed snapshot-based actor garbage collection algorithm can be performed in a non-blocking manner — that is, garbage collection does not stop the mutation operations of the application.

The pseudo-root approach, which is based on the live unblocked actor principle, makes actor garbage collection similar to passive object garbage collection. Actor garbage collection starts by identifying some live (not necessarily root) or even garbage actors as pseudo-roots. There are four types of pseudo-root actors: 1) root actors (persistent services), 2) unblocked actors, 3) *sender pseudo-root actors*, and 4) *global pseudo-root actors*. The second kind of pseudo-root actor is live according to the live unblocked actor principle (see Subsection II-C). The last two types of pseudo-root actors will be explained in the following paragraphs.

```
// Local Snapshot Actor:
1. Working_List L ← EMPTY
2. Group_of_Actors P ← EMPTY
3. Snapshot_Table ST ← EMPTY

Procedure local_monitor(Task T)
 1. if local_host_status = Cannot_take_a_snapshot then
 2.   reply NO
 3. else
 4.   L.pushTask(T)
 5.   if size(L) = 1 then
 6.     obtain a closed group of actors P
 7.     for each actor A in P
 8.       if A = NULL then
 9.         remove A from P // A has migrated away
10.       else
11.         enable state_logging of A
12. reply YES

Procedure local_snapshot(Task T)
 1. if L.find(T) = FALSE then
 2.   reply FAILED
 3. else
 4.   Snapshot S ← empty
 5.   for each actor A in P do
 6.     S.recordActor(A)
 7.   for each actor A in P do
 8.     stop state_logging of A
 9.   // save S into the snapshot table ST by
10.   // denoting the working list L on it (Line 11)
11.   ST.add(L,S)
12.   L.clear()
13.   reply OK
```

Fig. 7. The local snapshot actor.

Pseudo-root actors can be treated as roots directly, serving as the starting points of Depth-First-Search or Breadth-First-Search (a trace-based algorithm) to transitively mark live actors. A trace-based garbage collection algorithm works perfectly in a synchronous system with method invocations (or procedure calls). However, it causes problems on a system with non-blocking message passing. In a non-blocking communication environment, an actor can send a message by using a reference, and immediately delete the reference before the message has been received. A trace-based algorithm may mistake a live actor for garbage in such scenario. A similar problem can happen while an actor is passing a message to another actor and the message contains a reference. The actor referenced by an in-transit message may not be marked live by a trace-based algorithm in such case.

We introduce the idea of sender pseudo-roots to prevent erroneous garbage collection of actors in an asynchronous communication environment: either targets of in-transit messages or actors 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*. Both cases are illustrated in Figure 8. Deletion of a protected reference makes the reference unavailable for the application, but still visible to the garbage collection mechanism. A protected reference can only be physically deleted when the message sender knows the in-transit message has been received correctly.

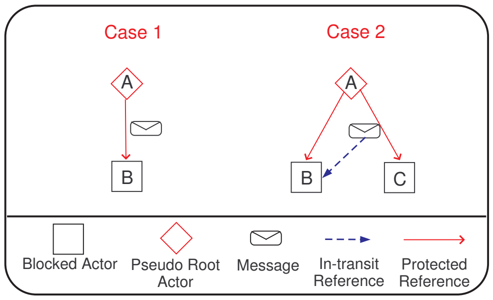

Fig. 8. Sender pseudo-root actors for in-transit messages and in-transit references.

Global pseudo-root actors consist of remotely referenced actors and potentially live actors with outgoing references to remote actors. They are live because they can possibly send a message to a root actor. With the live unblocked actor principle, the second case can be ignored since every potentially live actor is live. To identify remotely referenced actors, each actor can maintain inverse references to figure out its inverse acquaintances. Notice that the inverse references are not visible to applications. Unfortunately, precisely identifying remote inverse acquaintances in a non-blocking way is impossible. An alternative approach must be used — we must guarantee that remotely referenced actors either have an inverse reference to any remote actor, or are transitively reachable from some local pseudo-root actors (including itself).

To guarantee the existence of an inverse reference to a remote actor or reachability from a local pseudo-root, we only need to consider and constrain three kinds of mutation operations: actor creation, reference deletion, and reference passing: [1]

- **Actor creation**: We restrict actors to always be created locally. When an actor is created, the created actor atomically and automatically gets an inverse reference to the creator. In other words, precise inverse references are preserved after actor creation. Remote actor creation can be modeled by local creation followed by migration.
- **Reference deletion**: To ensure liveness of the pseudo-root approach, reference deletion must be handled carefully. An inverse reference should be deleted if its corresponding reference has been deleted. Furthermore, if a protected reference is deleted by the application, the reference should be preserved by the garbage collection system but should no longer be available to the application.
- **Reference passing**: We have designed a protocol to support reference passing under non-blocking, non-FIFO communication (see Figure 9). Unlike actor creation, reference passing may create asymmetric pairs of references and inverse references. The protocol combines the idea of sender pseudo-roots and inverse reference registration to ensure that the actor whose reference is being passed is referred to by a pseudo-root which appears in the actor's inverse reference list. This guarantees the actor will not be erroneously collected during reference passing.

IV. Computing Model and Snapshot Algorithm Properties

In this section, we define the model of local state logging, and then use the reachability relationship to prove safety and liveness of local garbage collection based on the local snapshot. We then formalize snapshot composition, and provide safety and liveness proofs for snapshot composition. Proofs of lemmas can be found in the Appendix.

[1]Migration is not considered because we treat migrating actors as live actors and migration does not affect the existence of inverse references.

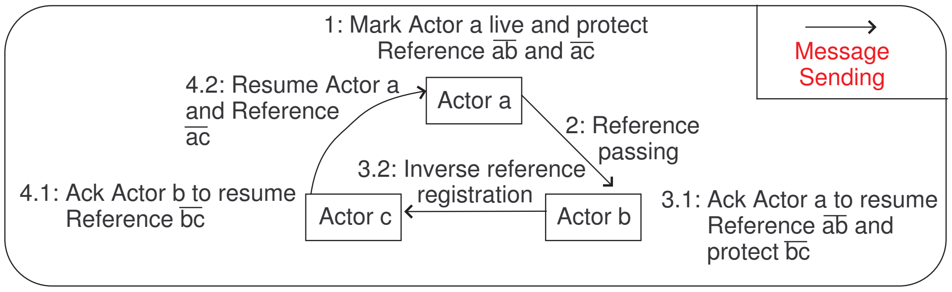
1: Mark Actor a live and protect Reference ab and ac
Message Sending
4.2: Resume Actor a and Reference ac
Actor a
2: Reference passing
3.2: Inverse reference registration
4.1: Ack Actor b to resume Reference bc
Actor c
Actor b
3.1: Ack Actor a to resume Reference ab and protect bc

Fig. 9. This figure illustrates the protocol of reference passing, in which each state is triggered by asynchronous messages. Let Actor a send a message to Actor b and the message contain a reference pointing to Actor c. At the beginning, Actor a becomes a sender pseudo-root and protects Reference $\overline{ab}$ and $\overline{ac}$ from deletion, and then sends the message to Actor b. When Actor b receives the message, it 1) gets the reference pointing to Actor c, 2) sends an asynchronous acknowledgement to Actor a to change the protected reference $\overline{ab}$ from protected to unprotected, 3) protects Reference $\overline{bc}$ from deletion to prevent the race between inverse reference registration and deletion messages, and then 4) sends an inverse reference registration message to Actor c. Upon receiving the inverse reference registration message, Actor c knows that it is referenced by Actor b, and then it sends two asynchronous acknowledgements — one to Actor a to change Reference $\overline{ac}$ from protected to unprotected, and the other one to Actor b to change the protected reference $\overline{bc}$ as well. Note that whenever Actor b has the reference pointing to Actor c, it can use it immediately. This protocol is non-blocking because no actor is waiting for any other actors; it is non-FIFO because we do not assume any message reception order.

A. Computing Model

Reachability from an actor to another actor is important for garbage collection, defined as follows:

Definition 4.1: Transitive reachability
Actor b is transitively reachable from Actor a, denoted by

$$a \rightsquigarrow b,$$

if and only if $a = b \vee (\exists c : \overline{ac} \wedge c \rightsquigarrow b)$. Otherwise, we say $a \not\rightsquigarrow b$.

The transitive reachability relationship is *asymmetric* ($a \rightsquigarrow b \not\Leftrightarrow b \rightsquigarrow a$), *reflective* ($a \rightsquigarrow a$), and *transitive* ($(a \rightsquigarrow b) \wedge (b \rightsquigarrow c) \Rightarrow (a \rightsquigarrow c)$).

Definition 4.2: Actor configuration (snapshot).
An actor configuration (snapshot),

$$S = \langle V, E, PS, IR \rangle,$$

is a 4-tuple where

- V is a set of actor names.
- E is a set of references. $E = \{\overline{xy} \,|\, x \in V \wedge \overline{xy}$ is a reference.$\}$
- PS is a set of pseudo-roots, excluding global pseudo-roots. $PS \subseteq V$.
- IR is a set of inverse references pointing to external actors. $IR = \{\overline{xy} \,|\, y \in V \wedge x \notin V\}$.

Definition 4.3: Receptionists, actor references, local actor references, and external inverse references.
Let S be an actor configuration and Actor $a \in S.V$. Then we define the set of receptionists (remotely referenced actors) $S.RE$, actor references $a.ref$, local actor references $a.lref$, and external inverse references $a.xir$.

- $S.RE = \{y \,|\, \overline{xy} \in S.IR\}$.
- $a.ref = \{\overline{ay} \,|\, \overline{ay} \in S.E\}$.
- $a.lref = \{\overline{ay} \,|\, \overline{ay} \in S.E \wedge y \in S.V\}$.
- $a.xir = \{\overline{xa} \,|\, \overline{xa} \in S.IR\}$.

Definition 4.4: Transitive relationship (mutation operation) on actor configurations.
Let S be an actor configuration (snapshot) and Actor $a \in S.V$. Then, $\rightarrow$ is defined as follows:

- $a.MI$: *Actor migration.*
  $\langle V, E, PS, IR \rangle \xrightarrow{a.MI} \langle V - \{a\}, E - a.ref, PS - \{a\}, IR \cup a.lref - a.xir \rangle$.

- $a.CR(b)$: *Reference creation.*
  $\langle V, E, PS, IR\rangle \xrightarrow{a.CR(b)} \langle V, E \cup \{\overline{ab}\}, PS, IR\rangle$.
- $a.CA(b)$: *Actor creation.*
  $\langle V, E, PS, IR\rangle \xrightarrow{a.CA(b)} \langle V, E \cup \{\overline{ab}\}, PS, IR\rangle$.
- $a.MR$: *Message reception.*
  $\langle V, E, PS, IR\rangle \xrightarrow{a.MR} \langle V, E, PS \cup \{a\}, IR\rangle$.
- $a.IRR(b)$: *Inverse reference registration.*
  $\langle V, E, PS, IR\rangle \xrightarrow{a.IRR(b)} \langle V, E, PS, IR \cup \{\overline{ba}\}\rangle$.

To concisely describe relationships of actors under mutation operations in snapshots, we introduce the following definitions:

Definition 4.5: Transitive state transition.
Let S_1 and S_2 be actor configurations.

$$S_1 \rightarrow^* S_2 \iff (S_1 = S_2) \vee (\exists S_x : (S_1 \rightarrow S_x) \wedge (S_x \rightarrow^* S_2)).$$

Definition 4.6: Constrained reachability at actor configurations
Let a and b be actor names, and S be an actor configuration.

$$\begin{aligned} a \rightsquigarrow b \textit{ at } S \iff & ((a = b) \wedge (a \in S.V)) \vee \\ & (\exists x : (\overline{ax} \in a.lref) \wedge (x \rightsquigarrow b \textit{ at } S)). \end{aligned}$$

Otherwise, we say $a \not\rightsquigarrow b$ *at* S.

Definition 4.7: Constrained live actors at actor configurations.
Let a be an actor name, and S be an actor configuration.

$$Live(a) \textit{ at } S \iff (\exists x : (x \in S.PS \cup S.RE) \wedge (x \rightsquigarrow a \textit{ at } S)).$$

Otherwise, we say $\neg Live(a)$ *at* S.

Definition 4.8: Migration during snapshot state transition.
Let a be an actor name. Let $S_s \rightarrow^* S_e$.

$$Migrated(a, S_s, S_e) \iff \exists S_i, S_j : S_s \rightarrow^* S_i \xrightarrow{a.MI} S_j \rightarrow^* S_e.$$

Otherwise, we say $\neg Migrated(a, S_s, S_e)$.

The snapshot mutation operations correspond to real-world computations but have a different effect. Therefore, they are restricted by the actor model — only live actors can become unblocked; only unblocked and root actors can compute; only live actors can become referenced. We formalize these restrictions using the following propositions, where S_s and S_e are actor configurations:

Proposition 4.9: Initial state of MI operation.

$$S_s \xrightarrow{a.MI} S_e \implies (a \in S_s.PS).$$

Proposition 4.10: Initial state of CR operation.

$$S_s \xrightarrow{a.CR(b)} S_e \implies ((a \in S_s.PS) \wedge (Live(b) \textit{ at } S_s)).$$

Proposition 4.11: Initial state of CA operation.

$$S_s \xrightarrow{a.CA(b)} S_e \implies ((a \in S_s.PS) \wedge (b \notin S_s.V)).$$

Proposition 4.12: Initial state of MR operation.

$$S_s \xrightarrow{a.MR} S_e \implies (Live(a) \textit{ at } S_s).$$

Proposition 4.13: Initial state of IRR operation.

$$S_s \xrightarrow{a.IRR(b)} S_e \implies ((Live(a) \textit{ at } S_s) \wedge (b \notin S_s.V)).$$

B. Local State Logging Properties

A local actor configuration never produces new garbage as the state mutates. The reason is the model does not delete references nor makes any actor blocked, except for actor migration. A migration operation breaks references and actors from the actor configuration. Meanwhile, it makes some actors referenced by an external actor, the migrating actor. We will prove that the local state logging model is correct. That is, we show that a migration operation does not affect reachability of actors from pseudo-roots, as formalized in Lemma 4.14. Therefore, migration does not add new garbage in the local snapshot.

Two actor configurations are used in the following lemmas and theorems, where S_s is the initial configuration, S_e is the final configuration of the local snapshot, and $S_s \rightarrow^* S_e$. We will use S_s and S_e directly without re-defining them again.

Lemma 4.14: Alternative guaranteed reachability for state transition.
$(a \rightsquigarrow b \textit{ at } S_s) \wedge (a \not\rightsquigarrow b \textit{ at } S_e) \Longrightarrow$
$(Migrated(b, S_s, S_e)) \vee (\exists \overline{yx} : ((x \rightsquigarrow b \textit{ at } S_e) \wedge (\overline{yx} \in S_e.IR) \wedge (Migrated(y, S_s, S_e))))$.

With Lemma 4.14, we now prove that the set of garbage is stable in the actor configuration during local state logging, as shown in Theorem 4.18. Theorem 4.18 directly turns into Corollary 4.19, which guarantees a stable set of local garbage during local state logging.

Lemma 4.15: $Live(a) \textit{ at } S_e \Longrightarrow Live(a) \textit{ at } S_s$.

Lemma 4.16: $Migrated(a, S_s, S_e) \Longrightarrow Live(a) \textit{ at } S_s$.

Lemma 4.17: $Live(a) \textit{ at } S_s \Longrightarrow ((Live(a) \textit{ at } S_e) \vee Migrated(a, S_s, S_e))$.

Theorem 4.18: Coherent live actors in a local snapshot.
$Live(a) \textit{ at } S_s \Longleftrightarrow (Live(a) \textit{ at } S_e) \vee (Migrated(a, S_s, S_e))$.

Proof: The proof is trivial by Lemma 4.15, 4.16, and 4.17. ■

Now, we can prove safety of local snapshot-based actor garbage collection. An actor is live at the beginning of local state logging if and only if it is live at the end or it has migrated.

Corollary 4.19: The stable property of the set of garbage actors of a local snapshot.
$\neg Live(a) \textit{ at } S_s \Longleftrightarrow (\neg Live(a) \textit{ at } S_e) \wedge (\neg Migrated(a, S_s, S_e))$.

Proof: The proof is trivial by Theorem 4.18. ■

C. Global Snapshot Algorithm Properties

Independent local state logging cannot reclaim global cyclic garbage. A coordinated action of local state logging is required to guarantee a causally consistent global snapshot. Let us assume that there are lots of computing nodes participating in a global snapshot activity. Figure 10 explains how global synchronization works. Now consider the synchronization pseudo-code in Figure 6. Let t_s be the time the last computing node replies `YES` (line 6), and t_e the time the last computing node finishes `local_snapshot` (line 11). When a computing node finishes a `local_snapshot`, the local actor configuration should remain the same. Let $S_{s,i}$ be the actor configuration of the local group of the computing node i at time t_x, where $t_s \leq t_x \leq t_e$. Let $S_{e,i}$ be the local actor configuration at time t_e. Local actor configurations at t_e can be obtained easily for garbage collectors because they never change again, while configurations at t_x are only used for proofs because they are volatile. Note that $S_{s,i} \rightarrow^* S_{e,i}$.

With the restriction of global synchronization, the algorithm guarantees that no actor can appear more than once among the participating local actor configurations.

Lemma 4.20: No actor appears more than once among coordinated local actor configurations.
Let $S_1, S_2, ..., S_m$ be coordinated local actor configurations.
$\forall i, j : (S_i.V \cap S_j.V = \emptyset)$ where $(i \neq j) \wedge (m \geq i, j \geq 1)$.

A global snapshot is composed of several different local snapshots. We introduce the *real-world actor configuration* to represent the computing state, and the *snapshot-composition operation* to compose local snapshots by identifying some local outgoing inverse references as global internal inverse references.

Definition 4.21: Real-world actor configuration.
A real-world actor configuration,

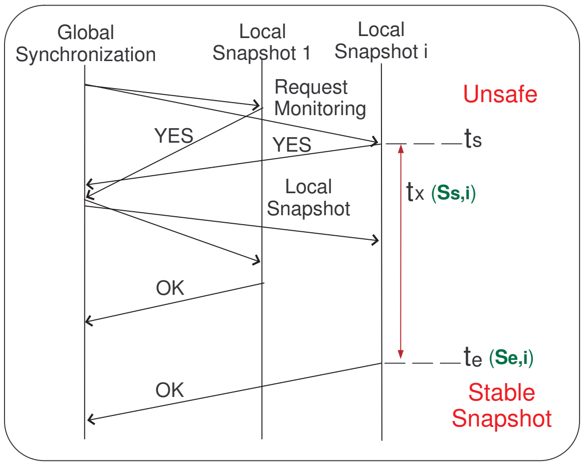

Fig. 10. Different phases of global synchronization.

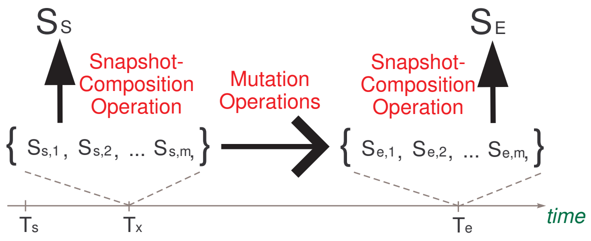
Ss
SE
Snapshot-Composition Operation
Mutation Operations
Snapshot-Composition Operation
{ Ss,1 , Ss,2 , ... Ss,m }
{ Se,1 , Se,2 , ... Se,m }
Ts
Tx
Te
time

Fig. 11. This figure shows the relationship of mutation operations, snapshots, and the snapshot-composition operation. There are two actor configuration sets in the figure — one is $\{S_{s,i} \,|\, m \geq i \geq 1\}$ at time t_x, and the other is $\{S_{e,i} \,|\, m \geq i \geq 1\}$ at time t_e, where $S_{s,i} \rightarrow^* S_{e,i}$ and t_x and t_e are defined in Figure 10 as time points. $S_S = (S_{s,1} \parallel S_{s,2} \parallel ... \parallel S_{s,m})$, and $S_E = (S_{e,1} \parallel S_{e,2} \parallel ... \parallel S_{e,m})$.

$$R = \langle V, E, PS, \emptyset \rangle,$$

is a special 4-tuple actor configuration which always represents the current state of the real world computations.

Definition 4.22: Binary snapshot-composition operation.
Let $S_x = \langle V_x, E_x, PS_x, IR_x \rangle$ and $S_y = \langle V_y, E_y, PS_y, IR_y \rangle$. Then

$$S_x \parallel S_y = \langle V_x \cup V_y, E_x \cup E_y, PS_x \cup PS_y, (IR_x - E_y) \cup (IR_y - E_x) \rangle,$$

where $V_x \cap V_y = \emptyset$.

The composition operation on actor configurations is *closed* ($\forall S_i, S_j$: $S_i \parallel S_j$ is also an actor configuration), *commutative* ($S_i \parallel S_j = S_j \parallel S_i$), and *associative* ($(S_i \parallel S_j) \parallel S_k = S_i \parallel (S_j \parallel S_k)$). The relationship of mutation operations, local snapshots, and the snapshot-composition operation are shown in Figure 11. Notice that S_S does not directly transit to S_E, and only the set $\{S_{e,i} \,|\, n \geq i \geq 1\}$ is observable.

The snapshot-composition operation can possibly identify new garbage which cannot be detected in each member of a snapshot set independently. However, the non-blocking, non-FIFO reference listing algorithm, such as the pseudo-root approach, has to guarantee Proposition 4.23 to solve inconsistency of two actor configurations — one has a reference to the other one and the other one does not have a corresponding inverse reference. Then we prove that garbage in the global snapshot remains stable (safety).

Proposition 4.23: Consistency guarantee of remote references and inverse references.
Let a and b be actor names, S and S_1 ... S_m be coordinated actor configurations, and R be the real-world actor configuration, then

$$(a \in R.V) \wedge (b \in S.V) \wedge (\overline{ab} \in R.E) \wedge (\overline{ab} \notin S.IR) \implies$$

$$(\exists r : (r \in S.PS) \wedge (\overline{rb} \in S.E)) \vee$$
$$(\exists r, S_i : (r \in S_i.PS) \wedge (\overline{rb} \in S_i.E) \wedge (\overline{rb} \in S.IR) \text{ where } m \geq i \geq 1) \vee$$
$$(\exists r : ((\overline{rb} \in S.IR) \wedge (\forall S_i : \overline{rb} \notin S_i.E) \text{ where } m \geq i \geq 1)).$$

The following lemmas and theorems require the concept of *constrained paths* to describe the relationship of actor reachability in a snapshot. We define them as follows:

Definition 4.24: Constrained path and its reference set.
Given $S = \langle V, E, PS, IR \rangle$, we define a constrained path P of $a_1 \rightsquigarrow a_n$ ***at*** S,
$$P = \overline{a_1 a_2 ... a_n} \text{ ***at*** } S$$
if and only if $\forall i : \overline{a_i a_{i+1}} \in S.E$ where $n > i \geq 1$, and we define the reference set of P as
$$PathSet(P) = \bigcup_{i=1}^{n-1} \{\overline{a_i a_{i+1}}\}.$$

To avoid redundant description in the following proofs, we define some variables as follows and then use them directly: Let R be the real-world actor configuration. Let $S_{s,1}$, $S_{s,2}$, ..., and $S_{s,m}$ be coordinated local snapshots. Let $S_S = (S_{s,1} \parallel S_{s,2} \parallel ... \parallel S_{s,m})$ and $S_E = (S_{e,1} \parallel S_{e,2} \parallel ... \parallel S_{e,m})$, where $S_{s,i} \rightarrow^* S_{e,i}$, $m \geq i \geq 1$. Let P be a path of $x \rightsquigarrow a$ ***at*** S_S s.t. $PathSet(P)$ contains the maximal inter-snapshot references of all paths of $x \rightsquigarrow a$ ***at*** S_S. Let the reference set be $PathSet(P).IR$. That is, the size of $PathSet(P).IR$, $\{\overline{cd} \,|\, \overline{cd} \in PathSet(P) \wedge (\exists i : \overline{cd} \in S_{s,i}.IR \text{ where } m \geq i \geq 1)\}$, is maximal.

Lemma 4.25: Actors which were reachable from a migrated actor at a local snapshot are reachable from some global pseudo-root at the merged global snapshot.
$(a \rightsquigarrow b$ ***at*** $S_{s,1}) \wedge (a \not\rightsquigarrow b$ ***at*** $S_{e,1}) \implies$
$Migrated(b, S_{s,1}, S_{e,1}) \vee (\exists z : (z \in S_E.RE) \wedge (z \rightsquigarrow b$ ***at*** $S_E))$.

Lemma 4.26: $((\neg Live(a)$ ***at*** $S_S) \wedge (a \in S_S.V)) \implies (\nexists y : (y \in R.PS) \wedge (y \rightsquigarrow a$ ***at*** $R))$.

Lemma 4.27: $(Live(a)$ ***at*** $S_S \wedge |PathSet(P).IR| = 0) \implies (Live(a)$ ***at*** $S_E)$.

Lemma 4.28: $Live(a)$ ***at*** $S_S \implies (Live(a)$ ***at*** $S_E) \vee (\exists i : Migrated(a, S_{s,i}, S_{e,i}), m \geq i \geq 1)$.

Lemma 4.29: $(Live(a)$ ***at*** $S_E) \vee (\exists i : Migrated(a, S_{s,i}, S_{e,i}), m \geq i \geq 1) \implies Live(a)$ ***at*** S_S.

Theorem 4.30: Coherent live actors in a global merged actor configuration.
$Live(a)$ ***at*** $S_S \iff (Live(a)$ ***at*** $S_E) \vee (\exists i : Migrated(a, S_{s,i}, S_{e,i}), m \geq i \geq 1)$.

Proof: The proof is trivial by Lemma 4.28 and 4.29. ■

Corollary 4.31: The stable property of the set of garbage actors in a global partial snapshot.
$\neg Live(a)$ ***at*** $S_S \iff ((\neg Live(a)$ ***at*** $S_E) \wedge (\nexists i : Migrated(a, S_{s,i}, S_{e,i}), m \geq i \geq 1)$.

Proof: This follows trivially from Theorem 4.30. ■

D. Liveness

The liveness proof of our snapshot model utilizes the property of stable garbage during state logging. If a garbage collector is periodically activated and all non-pseudo-root actors are selected, garbage is eventually collected.

Theorem 4.32: Conditional liveness of local garbage collection.
Let each local garbage collector be periodically activated, and use the local snapshot for actor garbage collection. If every non-pseudo-root actor is always selected, all local garbage is eventually identified by trace-based algorithms (DFS or BFS).

Proof: Since the set of local garbage actors is a subset of the local non-pseudo-root actors, all garbage actors are selected into the snapshot for garbage collection. According to Corollary 4.19, all garbage in this cycle can be identified and reclaimed in this cycle of garbage collection. The floating garbage produced during the current cycle can be identified at the next cycle because garbage actors cannot execute mutation operations, which means garbage is eventually identified. ■

A global garbage collector can use a distributed tracing algorithm to identify global garbage, including distributed cycles. Liveness of global garbage collection based on the snapshot algorithm can be proven in a similar manner of Theorem 4.32, and thus we skip the proof.

Theorem 4.33: Conditional liveness of distributed garbage collection that uses the snapshot algorithm.
Let the global garbage collection mechanism be periodically activated, and use the proposed snapshot

algorithm for distributed actor garbage collection. If every non-pseudo-root actor is always selected for snapshot, all garbage is eventually identified by distributed trace-based algorithms (DFS or BFS).

V. Related Work

The most well known snapshot-based garbage collection algorithm is proposed by Yuasa [45] as part of the Kyoto Lisp concurrent programming language, designed for passive object garbage collection in shared memory systems. The algorithm uses the snapshot-at-the-beginning strategy to preserve every reference of the beginning of garbage collection, and new objects allocated during garbage collection are preserved as well. The algorithm is as conservative as our local state logging strategy, except that our local state logging strategy considers migration and state of actors (unblocked or blocked). Because Yuasa's snapshot-based algorithm is 1) less interfering in applications than other kinds of garbage collection and 2) easy to implement and understand, it is used for real time Java garbage collection [3].

As mentioned in Section I, two kinds of snapshot algorithms are used in distributed garbage collection — the Chandy-Lamport algorithm [8] and the uncoordinated snapshot algorithm. Venkatasubramanian et al. [38] proposed a Chandy-Lamport based actor garbage collection algorithm, with the assumption of a fixed network topology. Because it reuses the Chandy-Lamport algorithm, FIFO communication is necessary to flush communication channels to capture the status of channels. Besides, any uncooperative computing node can fail a global garbage collection activity. Another Chandy-Lamport based, distributed actor garbage collection algorithm is proposed by Kafura et al. [15], which uses a global collector to coordinate activities of local state logging and global garbage collection. The algorithm does not make any assumption about network topology, but still shares the problem of the Chandy-Lamport algorithm. Neither of [38] and [15] consider the problems of actor migration.

There are several variations of uncoordinated snapshot algorithms. The idea is to simulate a global clock, and to find a causally consistent set of local snapshots among a huge set of uncoordinated snapshots where each computing node maintains more than one copy of local snapshots. Puaut's algorithm [28] is based on this approach. It is client-server based, and requires each computing node to maintain a time-stamp vector to simulate a global clock. Because finding a causally consistent global snapshot is not easy and time-consuming, the server only checks once on the local snapshots from clients. Therefore, the algorithm does not guarantee every global garbage collection activity can succeed even if every computing host is cooperative for garbage collection. Puaut's algorithm is not practical in grid computing environments because the overhead of messages increases as the number of computing nodes goes up. Besides, it is not easy to maintain a dynamic vector for each message in open computing environments where computing nodes can join and leave dynamically. Veiga et al. [37] proposed the distributed cycles detection algorithm (DCDA) [37], where asynchronous local snapshots have to be updated by local mutators to inform changes in the snapshots. The algorithm starts from suspecting an object as garbage, and a heavy cycle detection message *CDM* is then traversed among the snapshots to see if a cycle exists. If any traversed object is modified by the mutators, the current activity for global garbage collection must abort. A critical problem of this algorithm is that either: 1) stop-the-world synchronization for applications is required, or 2) a global garbage collection activity can be failed by any mutation operations.

There are other kinds of distributed actor garbage collection algorithms available in literature. For example, Vardhan and Agha have proposed a distributed actor garbage collection algorithm [34] which transforms each local actor reference graph into a passive object reference graph, and uses Schelvis' algorithm [31] for global garbage collection. An implementation of this algorithm [35] assumes: FIFO communication, and periodically performs stop-the-world garbage collection. All existing actor garbage collection algorithms in previous work violate the asynchronous, unordered assumption of actor communication, and none of them supports the concept of actor migration.

Distributed garbage collection for passive object systems are more common. Some are based on distributed reference counting [4], [5], [21], [25], [27], [32], [42], which cannot detect cyclic garbage but can serve as a fast mechanism to detect distributed acyclic garbage. These algorithms cannot be directly reused in actor systems because they assume FIFO communication, or blocking communication (*e.g.* remote procedure calls), or even both.

There are various distributed garbage collection algorithms that can detect cyclic garbage for passive object systems, and lots of them are hybrid with different approaches. We classify these algorithms according to their most noticeable feature as follows: 1) Global-time-based algorithms such as Hughes' algorithm [14], which uses global time-stamp propagation from roots to guarantee event orders. 2) Remote reference server-based algorithms such as [18] in which local collectors have to report remote references to a server. 3) Trial-deletion-based algorithms such as Vestal's algorithm [39], which tries to virtually delete a reference to see if a garbage cycle can be broken. 4) Heuristics-based algorithms such as [20], [22], [23]. The idea is to efficiently suspect some objects as garbage and then to verify them. 5) Group-collection-based algorithms such as [13], [19], [29]. The idea is to collect garbage in static or dynamic groups to achieve more incrementality. Dynamic groups are established by heuristics such as the age of objects (generations) or reachability from roots.

VI. Conclusion and Future Work

The actor model of computation is an excellent reasoning and development paradigm for grid and pervasive computing applications because of its ability to model parallel computing [17] and dynamic application reconfiguration [10], [36]. Since actor garbage collection is required in actor systems, research in actor garbage collection is imperative as the fields of grid and pervasive computing mature. In this paper, we have proposed a snapshot-based distributed actor garbage collection algorithm, along with the pseudo-code, the computing model, and proofs of correctness. Using the pseudo-root approach, the snapshot-based algorithm does not require FIFO or blocking communication, nor comprehensive cooperation of each computing node during global snapshot. These features make our algorithm unique in the area of distributed garbage collection. Furthermore, the snapshot algorithm supports actor migration and works concurrently with mutation operations, which reduces interruption of applications. We have implemented the algorithm as a logically centralized global garbage collector. No message logging is required in our algorithm which demands less space than traditional snapshot-based algorithms.

Global synchronization usually implies long waiting time. Instead of waiting at each computing node, the global synchronization service controls the distributed garbage collection flow to enable non-blocking distributed snapshot. The non-blocking snapshot algorithm for distributed garbage collection has been implemented as part of the SALSA programming language, whose open source can be downloaded from [43]. Our previous research [40], [41] has shown the overhead of distributed garbage collection to be acceptable (on average 16.4%).

When resource (root) access restrictions are considered, actor garbage collection becomes different from what we have described: in such case, the live unblocked actor principle is no longer true. Without the live unblocked actor principle, active garbage actors are possible — they can perform mutation operations but cannot possibly do any meaningful computation. Actor garbage collection in this context has not been explored and further research is needed to study the interaction of garbage collection with practical security models.

References

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

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.

[3] D. F. Bacon, P. Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 81–92, San Diego, California, June 2003.

[4] 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.

[5] 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.

[6] 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.

[7] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of MPI programs. In *Proceedings of the 2003 ACM SIGPLAN Symposium on Principles of Parallel Programming (PPoPP-03)*, pages 84–94. ACM Press, 2003.

[8] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[9] 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.

[10] K. El Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela. The Internet Operating System: Middleware for adaptive distributed computing, 2006. To appear in International Journal of High Performance Computing Applications (IJHPCA).

[11] K. El Maghraoui, B. Szymanski, and C. Varela. An architecture for reconfigurable iterative MPI applications in dynamic environments. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Proc. of the Sixth International Conference on Parallel Processing and Applied Mathematics (PPAM'2005)*, number 3911 in LNCS, pages 258–271, Poznan, Poland, September 2005.

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

[13] 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.

[14] 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.

[15] D. Kafura, M. Mukherji, and D. Washabaugh. Concurrent and distributed garbage collection of active objects. *IEEE TPDS*, 6(4), April 1995.

[16] 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.

[17] W. Kim. *THAL: An Actor System for Efficient and Scalable Concurrent Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1997.

[18] R. Ladin and B. Liskov. Garbage collection of a distributed heap. In *International Conference on Distributed Computing Systems*, Yokohama, June 1992.

[19] B. Lang, C. Queinnec, and J. Piquer. Garbage collecting the world. In *POPL'92 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 39–50. ACM Press, 1992.

[20] F. Le Fessant. Detecting distributed cycles of garbage in large-scale systems. In *Principles of Distributed Computing (PODC)*, Rhodes Island, Aug. 2001.

[21] 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.

[22] U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by controlled migration. In *PODC'95 Principles of Distributed Computing*, 1995.

[23] U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by back tracing. In *PODC'97 Principles of Distributed Computing*, pages 239–248, Santa Barbara, CA, 1997. ACM Press.

[24] J. Misra. Detecting termination of distributed computations using markers. In *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 290–294, 1983.

[25] L. Moreau. Tree rerooting in distributed garbage collection: Implementation and performance evaluation. *Higher-Order and Symbolic Computation*, 14(4):357–386, 2001.

[26] Open Systems Lab. The Actor Foundry: A Java-based Actor Programming Environment, 1998. http://osl.cs.uiuc.edu/foundry/.

[27] 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.

[28] I. Puaut. A distributed garbage collector for active objects. In *OOPSLA'94 ACM Conference on Object-Oriented Systems, Languages and Applications*, pages 113–128. ACM Press, 1994.

[29] 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.

[30] Y. Sato, M. Inoue, T. Masuzawa, and H. Fujiwara. A snapshot algorithm for distributed mobile systems. In *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, pages 734–743. IEEE Computer Society, 1996.

[31] M. Schelvis. Incremental distribution of timestamp packets — a new approach to distributed garbage collection. *ACM SIGPLAN Notices*, 24(10):37–48, 1989.

[32] M. Shapiro, P. Dickman, and D. Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapports de Recherche 1799, INRIA, Nov. 1992.

[33] D. C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996. TR UIUCDCS-R-96-1950.

[34] A. Vardhan. Distributed garbage collection of active objects: A transformation and its applications to java programming. Master's thesis, UIUC, Urbana Champaign, Illinois, 1998.

[35] 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.

[36] C. A. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 ACM Conference on Object-Oriented Systems, Languages and Applications*, 36(12):20–34, Dec. 2001.

[37] L. Veiga and P. Ferreira. Asynchronous complete distributed garbage collection. In O. Babaoglu and K. Marzullo, editors, *IPDPS 2005*, Denver, Colorado, USA, Apr. 2005.

[38] 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.

[39] S. C. Vestal. *Garbage collection: An exercise in distributed, fault-tolerant programming*. PhD thesis, University of Washington, Seattle, WA, 1987.

[40] W. Wang and C. A. Varela. Distributed garbage collection for mobile actor systems: The pseudo root approach. In *Advances in Grid and Pervasive Computing, First International Conference, GPC 2006*, volume 3947 of *Lecture Notes in Computer Science*, pages 360–372. Springer, May 2006.

[41] 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.

[42] 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.

[43] Worldwide Computing Laboratory. The SALSA Programming Language, 2005. http://wcl.cs.rpi.edu/salsa/.

[44] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*. MIT Press, Cambridge, Mass., 1990.

[45] T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.

APPENDIX

In this appendix, we provide proofs for the proposed snapshot computing model. The proofs for local state logging properties can be found in Sub-Section A; the proofs for global snapshot algorithm properties are described in Sub-Section B.

A. Local State Logging Properties

Two actor configurations are used in the following lemmas, where S_s is the initial configuration, S_e is the final configuration of the local snapshot, and $S_s \longrightarrow^* S_e$. We will use S_s and S_e directly without re-defining them again in this Sub-Section.

To concisely describe a sequence of state transitions excliuding migration, we provide the definition of no-migration transitive state transition as follows:

Definition 1.1: No-migration transitive state transition.
Let S_1 and S_2 be actor configurations.

$$S_1 \xrightarrow{(\neq MI)}{}^{*} S_2 \iff$$
$$(S_1 = S_2) \vee ((\forall a : S_1 \xrightarrow{mo} S_x \textit{ where } mo \neq a.MI) \wedge (S_x \xrightarrow{(\neq MI)}{}^{*} S_2)).$$

Similarly,

$$S_1 \xrightarrow{(\neq a.MI)}{}^{*} S_2 \iff$$
$$(S_1 = S_2) \vee ((S_1 \xrightarrow{mo} S_x \textit{ where } mo \neq a.MI) \wedge (S_x \xrightarrow{(\neq a.MI)}{}^{*} S_2)).$$

Lemma 4.14 Alternative guaranteed reachability for state transition.
$(a \rightsquigarrow b \textit{ at } S_s) \wedge (a \not\rightsquigarrow b \textit{ at } S_e) \Longrightarrow$
$(Migrated(b, S_s, S_e)) \vee (\exists \overline{yx} : (x \rightsquigarrow b \textit{ at } S_e) \wedge (\overline{yx} \in S_e.IR) \wedge (Migrated(y, S_s, S_e)))$.

Proof: The only operation to remove a reference or an actor from a snapshot is MI. There are two cases to be considered:

- Case 1: Let $(b \notin S_e.V)$. Since the only operation to remove b is $b.MI$, then there must exist S_i, S_j $s.t.\ S_s \longrightarrow^* S_i \xrightarrow{b.MI} S_j \longrightarrow^* S_e$, where $b \in S_i.V \wedge b \notin S_j.V$. Then $Migrated(b, S_s, S_e)$.
- Case 2: Let $(b \in S_e.V)$. Because $(a \not\rightsquigarrow b \textit{ at } S_e)$, there exists a reference at S_s and it is removed by a MI operation during state transitions, making $a \not\rightsquigarrow b$. This case can be proven by induction. Let the number of performed MI operations be n during state transitions where $n \geq 1$.

  Basis: Prove that the statement is true for $n = 1$.
  Let $S_s \longrightarrow^* S_i \xrightarrow{y.MI} S_j \longrightarrow^* S_e$ where $((a \rightsquigarrow y) \wedge (x \rightsquigarrow b) \wedge (\overline{yx} \in S_i.E))$ and $(a \not\rightsquigarrow b \textit{ at } S_j)$. Therefore, 1) $x \in S_j.RE$, 2) $x \rightsquigarrow b \textit{ at } S_j$, and 3) $\overline{yx} \in S_j.IR$. Because no more MI is performed, $((x \rightsquigarrow b \textit{ at } S_e) \wedge (\overline{yx} \in S_e.IR) \wedge (S_s \longrightarrow^* S_i \xrightarrow{y.MI} S_j \longrightarrow^* S_e))$ which proves the basis.
  Induction step: Assume the statement is true for n where $k \geq n \geq 1$, and show that it is true for $n = k + 1$.
  Let S_i be the state right before the last MI is performed. That is, $\exists y : S_i \xrightarrow{y.MI} S_j$. Because $(b \in S_e.V)$, $b.MI$ cannot be the last MI operation.
  Let $\overline{yx}$ be the reference removed by the last MI, $y.MI$. According to the induction hypothesis, $\exists \overline{zw}, S_k, S_m : ((w \rightsquigarrow y \rightsquigarrow x \rightsquigarrow b \textit{ at } S_i) \wedge (\overline{zw} \in S_i.IR) \wedge (S_s \longrightarrow^* S_k \xrightarrow{z.MI} S_m \longrightarrow^* S_i))$.
  Now consider the following sub-cases:

  Sub-case 1: Let $(w \rightsquigarrow b \textit{ at } S_j)$, which means $((w \rightsquigarrow b \textit{ at } S_j) \wedge (\overline{zw} \in S_j.IR) \wedge (S_s \longrightarrow^* S_k \xrightarrow{z.MI} S_m \longrightarrow^* S_j)))$. Since $(S_j \xrightarrow{(\neq MI)}{}^{*} S_e)$ do not remove any references or actor names, $((w \rightsquigarrow b \textit{ at } S_e) \wedge (\overline{zw} \in S_e.IR) \wedge (S_s \longrightarrow^* S_k \xrightarrow{z.MI} S_m \longrightarrow^* S_e)))$. Thus the lemma is true for Sub-case 1.
  Sub-case 2: The concept of this sub-case is shown in Figure 12. Let $(w \not\rightsquigarrow b \textit{ at } S_j)$. Since $((w \rightsquigarrow b \textit{ at } S_i) \wedge (w \not\rightsquigarrow b \textit{ at } S_j))$ and only one MI is performed during $(S_i \xrightarrow{y.MI} S_j \xrightarrow{(\neq MI)}{}^{*} S_e)$, the statement can be proven by re-using the basis.

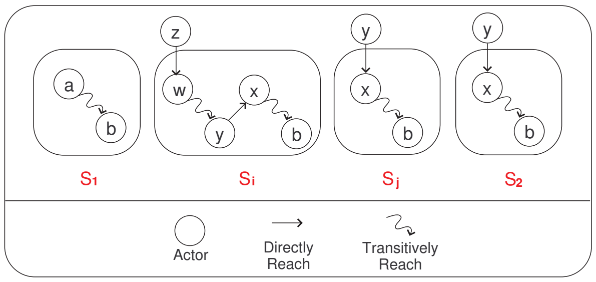

Fig. 12. Sub-case 2 of the induction step of the proof for Lemma 4.14.

Therefore, one can conclude that the statement is true by induction.

■

Lemma 4.15 $Live(a)$ *at* $S_e \implies Live(a)$ *at* S_s.

Proof: The statement can be proven by contradiction. Assume the statement is wrong, which means $(\neg Live(a)$ *at* $S_s) \implies \forall y : (y \in (S_s.PS \cup S_s.RE)) \wedge (y \not\rightsquigarrow a$ *at* $S_s)$. According to Proposition 4.9, 4.10, 4.11, 4.12, and 4.13, a cannot execute any mutation operation, and must remain the same state at S_e. By proposition 4.10, y cannot create a reference to a because $(Live(a)$ *at* $S_s)$ is required. Therefore, $(\forall y : (y \in (S_e.PS \cup S_e.RE)) \wedge (y \not\rightsquigarrow a$ at $S_e)) \implies (\neg Live(a)$ *at* $S_e)$, which contradicts the premise. ■

Lemma 4.16 $Migrated(a, S_s, S_e) \implies Live(a)$ *at* S_s.

Proof: $(Migrated(a, S_s, S_e)$ implies $(\exists S_i, S_j : S_s \rightarrow^* S_i \xrightarrow{a.MI} S_j \rightarrow^* S_e))$. By Proposition 4.9, $a \in S_i.PS$. By Definition 4.6 and 4.7, $Live(a)$ *at* S_i. By Lemma 4.15, $Live(a)$ *at* S_s. ■

Lemma 4.17 $Live(a)$ *at* $S_s \implies ((Live(a)$ *at* $S_e) \vee Migrated(a, S_s, S_e))$.

Proof: By Definition 4.6, $Live(a)$ *at* $S_e \iff \exists x : (x \in (S_e.PS \cup S_e.RE)) \wedge (x \rightsquigarrow a$ *at* $S_e)$. By Definition 4.8, $Migrated(a, S_s, S_e) \iff (\exists S_i, S_j : S_s \rightarrow^* S_i \xrightarrow{a.MI} S_j \rightarrow^* S_e))$. Assume the statement is wrong, which means:

Assumption 1: $\forall x : (x \in (S_e.PS \cup S_e.RE)) \wedge (x \not\rightsquigarrow a$ *at* $S_e)$ and

Assumption 2: $\forall S_i, S_j : S_s \rightarrow^* S_i \xrightarrow{a.MI} S_j \rightarrow^* S_e$ is false, which means $a \in S_e.V$

Let $z \in S_s.V$ and $(z \in (S_s.PS \cup S_s.RE)) \wedge (z \rightsquigarrow a$ *at* $S_s)$ from the premise. One can conclude from Assumption 1 and Assumption 2 that $(z \in (S_e.PS \cup S_e.RE)) \wedge (z \not\rightsquigarrow a$ *at* $S_e)$. By using Lemma 4.14, we know that $(z \rightsquigarrow a$ *at* $S_s) \wedge (z \not\rightsquigarrow a$ *at* $S_e)$ implies either:

Conclusion 1: $\exists S_i, S_j : S_s \rightarrow^* S_i \xrightarrow{a.MI} S_j \rightarrow^* S_e$, or

Conclusion 2: $\exists \overline{dc} : ((c \rightsquigarrow a$ *at* $S_e) \wedge (\overline{dc} \in S_e.IR) \wedge (\exists S_i, S_j : S_s \rightarrow^* S_i \xrightarrow{d.MI} S_j \rightarrow^* S_e))$.

Conclusion 1 contradicts Assumption 2. Conclusion 2 contradicts Assumption 1 because $\exists c : c \in S_e.RE \wedge (c \rightsquigarrow a$ *at* $S_e)$. Therefore, the lemma is true. ■

B. Global Snapshot Algorithm Properties

We define some variables as follows and then use them directly in this sub-section: Let R be the real-world actor configuration. Let $S_{s,1}$, $S_{s,2}$, ..., and $S_{s,m}$ be coordinated local snapshots. Let $S_S = (S_{s,1} \parallel S_{s,2} \parallel ... \parallel S_{s,m})$ and $S_E = (S_{e,1} \parallel S_{e,2} \parallel ... \parallel S_{e,m})$, where $S_{s,i} \rightarrow^* S_{e,i}$, $m \geq i \geq 1$. Let P be a path of $x \rightsquigarrow a$ *at* S_S s.t. $PathSet(P)$ contains the maximal inter-snapshot references of all paths of $x \rightsquigarrow a$ *at* S_S. Let the reference set be $PathSet(P).IR$. That is, the size of $PathSet(P).IR$, $\{\overline{cd} \mid \overline{cd} \in PathSet(P) \wedge (\exists i : \overline{cd} \in S_{s,i}.IR$ where $m \geq i \geq 1)\}$, is maximal.

Lemma 4.20 No actor appears more than once among coordinated local actor configurations.
Let $S_1, S_2, ..., S_m$ be coordinated local actor configurations.
$\forall i, j : (S_i.V \cap S_j.V = \emptyset)$ where $(i \neq j) \wedge (m \geq i, j \geq 1)$.

Proof: Only migration can cause an actor to appear twice in different locations. Let computing node i and j be two arbitrary computing nodes whose final local actor configurations are S_i and S_j respectively. Assume $\{a\} \subseteq (S_i.V \cap S_j.V)$ and the last two logged appearances of a be in computing node i, and then migrate to j. Also let t_{si} be the time computing node i starts state logging, t_{ei} computing node i finishes state logging, t_{sj} the time computing node j starts state logging, and t_{ej} the time computing node j finishes state logging. Let the time to migrate from node i be t_{smig}, and to arrive at node j be t_{emig}.

Case 1: $t_{smig} > t_{ei}$.
Now consider three sub-cases:

Sub-case 1.1: $t_{emig} > t_{ej}$.
a cannot be logged in S_j.
Sub-case 1.2: $t_{ej} \geq t_{emig} \geq t_{sj}$.
S_j cannot include any new actor during this period of time.
Sub-case 1.3: $t_{sj} > t_{emig}$ (an early message of migration).
Considering $t_{sj} > t_{emig}$, $t_{emig} > t_{smig}$, and $t_{smig} > t_{ei}$, we get $t_{sj} > t_{ei}$. However, the global synchronization mechanism guarantees that $t_{ei} \geq t_{sj}$ because snapshot termination must wait for a consensus of all participating computing nodes. Therefore, this sub-case is impossible due to a contradiction.

Case 2: $t_{ei} \geq t_{smig} \geq t_{si}$.
The MI operation guarantees that $\{a\} \notin S_i.V$.
Case 3: $t_{si} > t_{smig}$.
a cannot be logged in S_i.

■

Lemma 4.25 Actors which were reachable from a migrated actor at a local snapshot are reachable from some global pseudo-root at the merged global snapshot.
$(a \rightsquigarrow b \textbf{ at } S_{s,1}) \wedge (a \not\rightsquigarrow b \textbf{ at } S_{e,1}) \implies$
$Migrated(b, S_{s,1}, S_{e,1}) \vee (\exists z : (z \in S_E.RE) \wedge (z \rightsquigarrow b \textbf{ at } S_E))$.

Proof: According to Lemma 4.14, one of the following statements is true.

Case 1: $\exists S_j, S_k : S_{s,1} \rightarrow^* S_j \xrightarrow{b.MI} S_k \rightarrow^* S_{e,1}$ by Definition 4.8.

Case 2: $\exists S_j, S_k, \overline{wx} : ((x \rightsquigarrow b \textbf{ at } S_{e,1}) \wedge (\overline{wx} \in S_{e,1}.IR) \wedge (S_{s,1} \rightarrow^* S_j \xrightarrow{w.MI} S_k \rightarrow^* S_{e,1}))$.

Figure 13 helps understand the proof. If Case 1 is true, the statement to prove is also true. Now consider the Case 2. Let $S'_E = (S_{e,2} \parallel S_{e,3} \parallel ... \parallel S_{e,m})$. Let S_j, S_k be the actor configurations and $\overline{wx}$ be the reference to make Case 2 true. Because $w \in S_{s,1}.V$ and no duplicate actor name is allowed (Lemma 4.20), we know $(w \notin S_{s,i}.V$ where $m \geq i > 1)$. Consequently, $w \notin S_{e,i}.V$ where $m \geq i > 1$ because no mutation operation can add any new actor name. Since $(w \notin S_{e,i}.V$ where $m \geq i > 1)$, we get $(\overline{wx} \notin S_{e,i}.E$ where $\forall i : m \geq i > 1)$, which also implies that $\overline{wx} \notin S'_E.E$. From Case 2 we know $(\overline{wx} \in S_{e,1}.IR)$, and thus we get $\overline{wx} \in (S_{e,1}.IR - S'_E.E)$.

Now let us compose $S_{e,1}$ and S'_E. We find that $(\overline{wx} \in (S_{e,1} \parallel S'_E).IR)$, which is equal to $(\overline{wx} \in S_E.IR)$. Therefore, $(x \in S_E.RE)$. Case 2 also says that $((x \rightsquigarrow b \textit{ at } S_{e,1})$, indicating $((x \rightsquigarrow b \textit{ at } S_E.E)$. By replacing x with z, we finish the proof. ■

Lemma 4.26 $((\neg Live(a) \textbf{ at } S_S) \wedge (a \in S_S.V)) \implies (\nexists y : (y \in R.PS) \wedge (y \rightsquigarrow a \textbf{ at } R))$.

Proof: Assume the lemma is wrong, which means $(\exists y : (y \in R.PS) \wedge (y \rightsquigarrow a \textbf{ at } R)) \implies (\exists w, x, y : (y \rightsquigarrow w \textbf{ at } R) \wedge (\overline{wx} \notin S_S.E) \wedge (x \rightsquigarrow a \textbf{ at } S_S))$. Since $\overline{wx} \in S_S.IR \implies ((x \in S_S.RE) \wedge (x \rightsquigarrow a \textbf{ at } S_S)) \implies Live(a) \textbf{ at } S_S$ which contradicts the premise, we know $\overline{wx} \notin S_S.IR$. According to Proposition 4.23, $(\exists z : (z \in S_S.IR \cup S_S.PS) \wedge (z \rightsquigarrow a \textbf{ at } S_S)) \implies (Live(a) \textbf{ at } S_S)$ which also contradicts the premise.

■

Lemma 4.27 $(Live(a) \textbf{ at } S_S \wedge |PathSet(P).IR| = 0) \implies (Live(a) \textbf{ at } S_E)$.

Proof: Let $a \in S_{s,1}.V$ without losing generality. $|PathSet(P).IR| = 0$ implies that $(PathSet(P) \cap S_{s,1}.IR) \cup (PathSet(P) \cap S_{s,2}.IR) \cup ... \cup (PathSet(P) \cap S_{s,m}.IR) = \emptyset$. Consequently, $\exists x : (x \in (S_S.PS \cup S_S.RE)) \wedge (x \rightsquigarrow a \textbf{ at } S_{s,1})$.

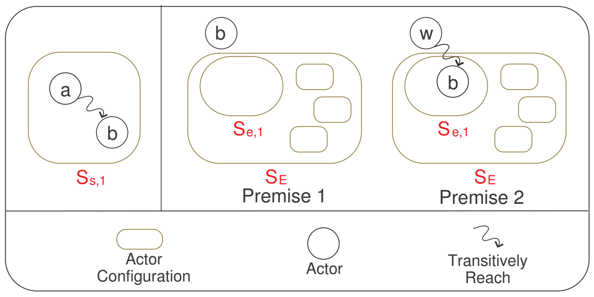

Fig. 13. This figure shows two different possible global actor configurations where $\exists S_k, S_m : S_{s,1} \rightarrow^* S_k \xrightarrow{a.MI} S_m \rightarrow^* S_{e,1}$.

Case 1: Now consider the case that $x \rightsquigarrow a$ ***at*** $S_{e,1}$.

Sub-case 1.1: Let $(x \in S_S.PS)$, which implies $(x \in S_{s,1}.PS)$ since $(x \rightsquigarrow a$ ***at*** $S_{s,1})$. Because $x \in S_{e,1}.V$, no migration operation is executed. Since $(x \in S_{s,1}.PS)$, we know $(x \in S_{e,1}.PS)$ such that $((x \in (S_E.PS \cup S_E.RE)) \wedge (x \rightsquigarrow a$ ***at*** $S_E))$, implying $Live(a)$ ***at*** S_E.

Sub-case 1.2: Let $x \in S_S.RE$. This implies that $\exists \overline{zx} : (\overline{zx} \in S_{s,1}.IR) \wedge (\overline{zx} \notin S_{s,i}.E$ where $m \geq i \geq 1)$. $x \in S_{e,1}.V$ implies that $S_{s,1} \xrightarrow{\neq x.MI}{}^{*} S_{e,1}$. Therefore, $(x \in S_{e,1}.V)$ which also implies $\exists \overline{zx} : (\overline{zx} \in S_{e,1}.IR) \wedge (\overline{zx} \notin S_{e,i}.E$ where $m \geq i \geq 1)$. Then $\exists \overline{zx} : \overline{zx} \in S_E.IR$. To conclude, $\exists x : (x \in S_E.IR) \wedge (x \rightsquigarrow a$ ***at*** $S_E)$, implying $Live(a)$ ***at*** S_E.

Case 2: Now consider the case that $x \not\rightsquigarrow a$ ***at*** $S_{e,1}$. According to Lemma 4.14, either $Migrated(a, S_{s,1}, S_{e,1})$ which means $\exists S_{ma}, S_{mb} : S_{s,1} \rightarrow^* S_{ma} \xrightarrow{a.MI} S_{mb} \rightarrow^* S_{e,1}$, or $\exists S_{ma}, S_{mb}, \overline{cw} : ((w \rightsquigarrow a$ ***at*** $S_{e,1}) \wedge (\overline{cw} \in S_{e,1}.IR) \wedge (S_{s,1} \rightarrow^* S_{ma} \xrightarrow{c.MI} S_{mb} \rightarrow^* S_{e,1}))$. Because $(c \in S_{s,1}.V)$, we know $(c \notin S_{s,i}.V$ where $m \geq i \geq 2)$, which also means that $c \notin S_{e,i}.V$ where $m \geq i \geq 2$. $(c \notin S_{e,i}.V$ where $m \geq i \geq 1)$ implies $(\forall d : \overline{cd} \notin S_{e,i}.E$ where $m \geq i \geq 1)$. Since $\overline{cw} \in S_{s,1}.IR$ and $(\forall d : \overline{cd} \notin S_{e,i}.E$ where $m \geq i \geq 1)$, we get $\overline{cw} \in S_E.IR$ which also means $w \in S_E.RE$. Therefore, $(w \in (S_E.PS \cup S_E.RE)) \wedge (w \rightsquigarrow a$ ***at*** $S_E))$, implying $Live(a)$ ***at*** S_E. ■

Lemma 4.28 $Live(a)$ ***at*** $S_S \implies (Live(a)$ ***at*** $S_E) \vee (\exists i : Migrated(a, S_{s,i}, S_{e,i}), m \geq i \geq 1)$.

Proof: Let $a \in S_{s,1}.V$ without losing generality. Let $|PathSet(P).IR| = n$.

Basis: The case of $n = 0$ is proven by Lemma 4.27.

Induction step: For each $k \geq 0$ assume the statement is true, and prove the case $n = k + 1$.

Let $\overline{pq}$ be any inter-snapshot reference of $x \rightsquigarrow a$ ***at*** S_S s.t. $((x \rightsquigarrow p$ ***at*** $S_S) \wedge (q \in S_{s,1}.V) \wedge (\exists j : p \in S_{s,j}.V$ where $m \geq j \geq 2) \wedge (q \rightsquigarrow a$ ***at*** $S_{s,1}))$. Consequently, the induction assumption indicates either 1) $Live(p)$ ***at*** S_E, or 2) $\exists i, S_{ms,i}, S_{me,i} : (S_{s,i} \rightarrow^* S_{ms,i} \xrightarrow{p.MI} S_{me,i} \rightarrow^* S_{e,i})$, $m \geq i \geq 1$ by Definition 4.8.

These cases are discussed as follows:

Case 1: $(\exists f : (f \in (S_E.PS \cup S_E.RE)) \wedge (f \rightsquigarrow p$ ***at*** $S_E))$. There are two sub-cases to deal with:

Sub-case 1.1: Let $q \rightsquigarrow a$ ***at*** $S_{e,1}$. This means $Live(a)$ ***at*** S_E.

Sub-case 1.2: Let $q \not\rightsquigarrow a$ ***at*** $S_{e,1}$. By Lemma 4.25, we know

- $\exists S_{ms,1}, S_{me,1} : S_{s,1} \rightarrow^* S_{ms,1} \xrightarrow{a.MI} S_{me,1} \rightarrow^* S_{e,1}$, or
- $Live(a)$ ***at*** S_E.

Case 2: Now consider that $\exists i, S_{ms,i}, S_{me,i} : (S_{s,i} \rightarrow^* S_{ms,i} \xrightarrow{p.MI} S_{me,i} \rightarrow^* S_{e,i})$, $m \geq i \geq 1$. There are two sub-cases:

Sub-case 2.1: Assume $q \rightsquigarrow a$ ***at*** $S_{e,1}$. Let $S_{s,i}$ have transitted to $S_{ms,i}$ and $S_{s,1}$ have transitted to $S_{ms,1}$ at time t_x. Notice that $S_{s,i} \rightarrow^* S_{ms,i} \rightarrow^* S_{e,i}$ and $S_{s,1} \rightarrow^* S_{ms,1} \rightarrow^* S_{e,1}$.

Sub-case 2.1.1: Assume $\overline{pq} \in S_{ms,1}.IR$. Since $q \in S_{e,1}$, we get $\overline{pq} \in S_{e,1}.IR$. Because $p \in S_{s,i}.V$ and no duplicate actor name is allowed (Lemma 4.20), we know $(p \notin S_{s,j}.V$ where $m \geq j \geq 1)$. Consequently, $p \notin S_{e,i}.V$ where $m \geq i \geq 1$ because no mutation operation can add any new actor name. Since $(p \notin S_{e,i}.V$ where $m \geq i \geq 1)$, we get $(\overline{pq} \notin S_{e,i}.E$ where $\forall i : m \geq i \geq 1)$. Therefore, $(q \in S_E.RE \wedge q \rightsquigarrow a$ ***at*** $S_E) \implies Live(a)$ ***at*** S_E.
Sub-case 2.1.2: Assume $\overline{pq} \notin S_{ms,1}.IR$. According to Proposition 4.23, there are three cases to consider as follows:

- Let $(\exists r : (r \in S_{ms,1}.PS) \wedge (\overline{rq} \in S_{ms,1}.E))$ be true. If $(\exists S_{rms,1}, S_{rme,1} : S_{ms,1} \rightarrow^* S_{rms,1} \xrightarrow{r.MI} S_{rms,1} \rightarrow^* S_{e,1})$ is true, we can prove the statement by Lemma 4.25. Otherwise, $r.MI$ has never been executed. Therefore, $((r \in S_{e,1}.PS) \wedge (r \rightsquigarrow q \rightsquigarrow a$ ***at*** $S_{e,1}))$, which implies $Live(a)$ ***at*** S_E.
- Let $(\exists r, S_{ms,j} : (r \in S_{ms,j}.PS) \wedge (\overline{rq} \in S_{ms,j}.E) \wedge (\overline{rq} \in S_{ms,1}.IR)$ where $S_{ms,j} \neq S_{ms,1})$ be true. First assume $\exists S_{rms,j}, S_{rme,j} : S_{ms,j} \rightarrow^* S_{rms,j} \xrightarrow{r.MI} S_{rms,j} \rightarrow^* S_{e,j}$. Since $\overline{rq} \in S_{ms,1}.IR$ and $q \in S_{ms,1}.V$, we know $\overline{rq} \in S_{e,1}.IR$. Because $\forall S_{e,newj} : r \notin S_{e,newj}.V$ where $m \geq newj \geq 1$ and $\overline{rq} \in S_{ms,1}.IR$, we get $\overline{rq} \in S_E.IR$. Therefore, $(q \in S_E.RE) \wedge (q \rightsquigarrow a$ ***at*** $S_E)$, indicating $Live(a)$ ***at*** S_E. Now consider the other case that $r.MI$ has never been executed. We find that $(r \in S_E.PS) \wedge (r \rightsquigarrow q \rightsquigarrow a$ ***at*** $S_E)$, implying $Live(a)$ ***at*** S_E.
- Let $\exists r : \forall S_{msi} : (\overline{rq} \in S_{ms,1}.IR) \wedge (\overline{rq} \notin S_{msi}.E)$ where $S_{msi} \neq S_{ms,1}$ be true, which implies that $(\overline{rq} \in S_{e,1}.IR) \wedge (\forall S_{e,i} : \overline{rq} \notin S_{e,i}.E)$ where $S_{e,i} \neq S_{e,1}$. Therefore, $(q \in S_E.RE) \wedge (q \rightsquigarrow a$ ***at*** $S_E)$, implying $Live(a)$ ***at*** S_E.

Sub-case 2.2: Let $q \not\rightsquigarrow a$ ***at*** $S_{e,1}$. Sub-case 2.2 can be proven by reusing the proof of Lemma 4.27.
We conclude the statement is true by induction.

■

Lemma 4.29 $(Live(a)$ ***at*** $S_E) \vee (\exists i : Migrated(a, S_{s,i}, S_{e,i}), m \geq i \geq 1) \implies Live(a)$ ***at*** S_S.

Proof: The lemma can be proven by contradiction. Assume $\neg Live(a)$ ***at*** S_S where $a \in S_S.V$. By Lemma 4.26, we get $(\forall y : (y \in R.PS) \wedge (y \not\rightsquigarrow a$ ***at*** $R))$. Since local state logging corresponds to the real world computing, Actor a cannot execute any mutation operations, which means $\neg Live(a)$ ***at*** S_E and thus contradicts the premise. ■