Concurrent Garbage Collection in the Actor Model

Dan Plyukhin
University of Illinois at Urbana-Champaign
Urbana, IL, USA
daniilp2@illinois.edu

Gul Agha
University of Illinois at Urbana-Champaign
Urbana, IL, USA
agha@illinois.edu

Abstract

In programming languages where memory may be allocated dynamically, automatic garbage collection (GC) can improve the efficiency of program execution while preventing program errors caused by incorrectly removed memory locations. In actor systems, GC poses some challenges that make it much costlier than in the sequential setting: Besides references from reachable actors, we have to consider inverse references from potentially active actors to reachable actors. One proposal, adopted in the runtime for the actor programming language Pony, uses causal message delivery and a centralized detection algorithm. While this is efficient in a multicore setting, the solution is too expensive for a distributed actor runtime. In this work, we show how the causal order message delivery requirement may be removed. Specifically, we describe a tracing collector of distributed actor garbage with centralized and decentralized variants. Both are guaranteed not to collect any non-garbage actors (safety) and to eventually collect all garbage actors (liveness).

CCS Concepts • **Computing methodologies → Concurrent algorithms**;

Keywords garbage collection, distributed systems

ACM Reference Format:
Dan Plyukhin and Gul Agha. 2018. Concurrent Garbage Collection in the Actor Model. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE '18), November 5, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3281366.3281368

1 Introduction

In programs that use dynamic memory by creating objects at runtime, *Garbage Collection* (GC) of those objects that are no longer reachable can improve performance by freeing up memory. However, asking programmers to explicitly deallocate memory leads to two problems: One might introduce inefficiency in execution by failing to free up unused memory, and one might also introduce runtime errors by freeing memory of objects still in use by parts of the application. Automating GC makes it possible to avoid these problems.

In this paper, we consider the problem of automatic GC in distributed actor systems [2].

Standard tracing techniques like mark-and-sweep, while successful in sequential languages, are not immediately applicable to actors because tracing the objects reachable from a fixed "root set" is insufficient for finding all live (i.e. non-garbage) actors. For example, a worker continuously logging messages to the console is clearly not garbage, but may not be reachable from any other actor. It follows that a tracing approach requires determining both outgoing and *incoming* references to each actor, but it is not obvious how to do so efficiently; previous work like [15] used inverse acquaintance lists, which proved expensive to maintain.

A successful alternative approach targeting multicore systems is Pony's Message-based Actor Collection (MAC) [5], which can be implemented with actors and no synchronization requirements, while remaining competitive with implementations that offer no actor GC at all. MAC employs two forms of GC:

1. *Deferred reference counting* enables an actor A to detect when no other actor knows A's name. Once that occurs, A must be garbage because it cannot receive any further messages.
2. *Centralized cycle detection* detects all other forms of actor garbage, such as where two actors know one another's names but neither has any work to do.

This latter part depends on *causal message delivery*: If actor A sends a message m to actor B and subsequently sends m' to C, then m' must be delivered after m.

While causal message delivery can be implemented efficiently on multicore architectures, it decreases concurrency and incurs a significant additional cost to be added to a distributed system [4, 12]. In addition, all cycle detection in MAC is performed by a single actor, which is potentially a bottleneck.

The purpose of the present work is to show how both these problems can be solved by replacing causal cycle-confirmation protocol described above with an unordered protocol that does not require confirmation messages. Our

AGERE '18, November 5, 2018, Boston, MA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6066-1/18/11…$15.00
https://doi.org/10.1145/3281366.3281368

scheme requires very few additional messages to the application and makes constraints on message ordering or the activity of actors during garbage collection.

The layout of the paper is as follows: In Section 2 we define our formal model for actors and actor garbage. Section 3 introduces the notion of a *reference tracking scheme* (RTS) and its challenges in a distributed setting. Section 4 introduces a novel low-overhead distributed RTS, which is necessary for the presentation and proof of our complete GC system in Section 5. We discuss our approach in Section 6, present related work in Section 7, and briefly describe several future research directions in Section 8.

2 Model

All our terms – including, crucially, the definition of actor garbage – will be given in relation to an abstract actor system, similar to [1], described below. In particular, notions of migration and the "location" of an actor will not be necessary because garbage detection takes place at the actor level. Consequently, our work should be applicable to any system implementing the actor abstraction, independent of migration capabilities.

Garbage collection of shared passive data structures is outside the scope of our paper, since we assume actors share no state. However, we suggest how it could be integrated into our scheme in Section 6.3 by modeling objects as actors.

Actors An actor (denoted $A, B, C, \ldots$) is a stateful sequential processing entity with the capacity to spawn new actors, which can execute concurrently. Actors do not share state, and instead communicate by sending asynchronous messages to their target's globally unique actor *name* (denoted here by the lowercase $a : A$). These messages are buffered at their target's mailbox and processed one at a time. Actors are purely reactive, and only ever perform computations in response to a message. Thus, each message handler is guaranteed to access the actor's state in isolation with no low-level data races.

Most implementations ensure that messages from an actor A to an actor B are delivered in FIFO order. We will assume this holds in our model as well for simplicity, and then show how to easily lift the requirement in Section 6.

It is said that A is *acquainted with* B if A knows B's actor name (even if B does not know A's name). Actors initially only know their own names and those provided as arguments to their constructor. Subsequently, the only way for A to learn new names is by creating an actor C, at which point it learns $c : C$, or by receiving names in a message.

Loosely speaking, an actor *configuration* is the state of the system, from some point of view, at a particular point in time [1]. The *topology* induced by a configuration is a directed graph, where nodes correspond to actors and there is an edge from A to B if and only if A is acquainted with B.

Garbage Actors Our definition of garbage (see Fig. 1) is based on that of [15], with some careful modifications.

1. We say that A can *reach* B if A *is* B or A has an acquaintance A' that can reach B. Reachability is therefore the transitive closure of the acquaintance relation, with reflexivity because we assume that an actor can always send messages to itself.
2. An actor is a *root* if it can communicate directly with the outside world. Examples include actors representing users, the console, or a database connection.
3. An actor is *blocked* when it is not executing and has no unprocessed messages (including those in transit); it becomes *unblocked* as soon as a new message is sent to it.
4. An actor is *potentially unblocked* if it is reachable from an unblocked actor. Otherwise, it is *quiescent*.
5. An actor is *live* if it is in the root set, is reachable from a live actor, or is *potentially unblocked* and can reach a live actor.
6. An actor is *garbage* if it is not live; notice that once an actor becomes garbage, it remains so for the remainder of its lifetime.

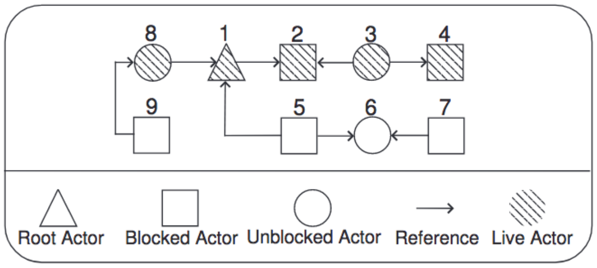


Figure 1. Diagram, taken from [15], indicating when an actor is garbage. Actor 2 is live because it is reachable from the root; Actors 3 and 8 are live because they can reach a live actor. Although Actor 6 is potentially unblocked, it cannot reach a live actor and is therefore garbage.

A garbage detection algorithm is said to be *sound* if it only ever returns true garbage actors, and *complete* if every garbage actor is eventually detected. A garbage collection (GC) scheme is sound and complete if and only if its underlying garbage detection algorithm is sound and complete.

The creators of the SALSA 1.0 garbage collector remark that, in practice, *every* actor has a reference to the root set because it can, for example, write to the console [15]. It follows that every potentially unblocked actor in such a system is in fact live, and so garbage collection is reduced to the detection of quiescent sets. This argument, called the *Live Unblocked Actor Principle* (LUAP), is also (tacitly) assumed by MAC [5].

We contend that LUAP is too pessimistic, as most actors never need to *directly* affect the world because they are merely helpers to the live actors. For example, consider how a cyclic distributed hash table made up of actors might always be potentially unblocked because it is rebalancing or replicating data for fault-tolerance purposes. Such a data structure would never be reclaimed, even if unreachable from the live set, because it's never quiescent. We therefore will not assume LUAP, and propose that users distinguish which actors have have a reference to the root set with, say, an `IOActor` trait.

3 References

The foundation of our garbage detection algorithm is *proactive reference tracking*, which is an instance of a distributed *reference tracking scheme* (RTS) for actors.

Reference Tracking A *reference tracking scheme* (RTS) is a sound but incomplete GC that enables every actor B to determine:

1. When no other actor knows B's name, and
2. When there are no undelivered messages to B.

Actors meeting conditions (1) and (2) are quiescent garbage, and may therefore safely destroy themselves.

An RTS detects these conditions by assuming that actors only communicate by means of *references*. A reference $x : A \to B$ is an abstract data type that can only be used by its designated *owner* A to send messages to a *target* B. After A no longer needs to send messages to B, the reference x must be *released*, as defined below. We say that A is the (unique) owner of x and that A is a (not necessarily unique) owner of B.

Thus we have two basic requirements:

Requirement 0: *Actor A can only send a message to actor B if A possesses an unreleased reference to B.*

Requirement 1: *The owner A of a reference to B must eventually release that reference after it no longer needs to send messages to B.*

An RTS implementation must therefore define:

1. *How to share references:* Given that A possesses $x : A \to B$ and $y : A \to C$, how can it create $z : C \to B$ so that C can access B?
2. *How to release references:* The RTS must specify a *release protocol* for informing the target when references are destroyed.

Note that an actor can simultaneously possess multiple *distinct* references to the same target, for example if A sends $y : C \to B$ to C and some D concurrently sends $z : C \to B$. Thus y, z are not "merged" at C, and each one must be independently released.

Finally, actor creation must also be modified by the RTS, so as to return a reference instead of a mere actor name.

Naïve Reference Counting Before delving into proactive reference tracking, let us briefly digress to show what kinds of race conditions we mean to avoid. Below we define the simple *asynchronous reference counting* RTS, which is sound in causal multicore systems but not in our model [11].

In reference counting approaches, a reference is simply an actor name with a designated owner. Actors initially have an integer *reference count* initialized to 1, and must handle the following messages for the release protocol:

- INC(): Increment local reference count.
- DEC(): Decrement local reference count; if it fell to 0, then release all references and destroy this actor.

Finally, the following methods show how to create and release references:

- CREATE_REF($x : A \to B$, $y : A \to C$): Send an INC message to B and return $b : B$ to be owned by C.
- RELEASE($x : A \to B$): Send B a DEC message.

The problem with this approach is a race condition between INC and DEC messages. In Fig. 2, actor C essentially releases A's reference instead of its own, because B didn't learn about C's reference in time.

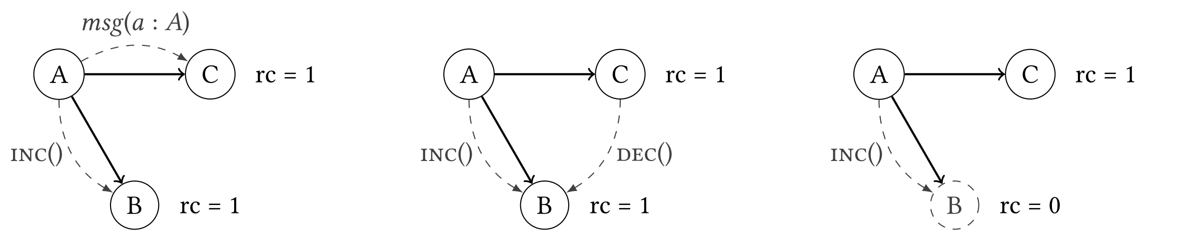
Several distributed reference counting approaches have been developed to deal with this type of race condition [3, 10, 17], but they are inapplicable to our garbage detection algorithm in Section 5.

4 Proactive Reference Tracking

Proactive reference tracking is an optimized form of *reference listing* – as used in [13, 15] – in which each reference has a distinct identifier to prevent race conditions. This is inspired by the representation of capabilities as an "unforgeable authentication token" [7] together with a designated owner and target address. We also assume these tokens are globally unique and efficiently comparable. Thus, we let each reference $x : A \to B$ represent a triple $(x, a : A, b : B)$, where x is the token and $a : A, b : B$ are the names of the owner and target actors, respectively.

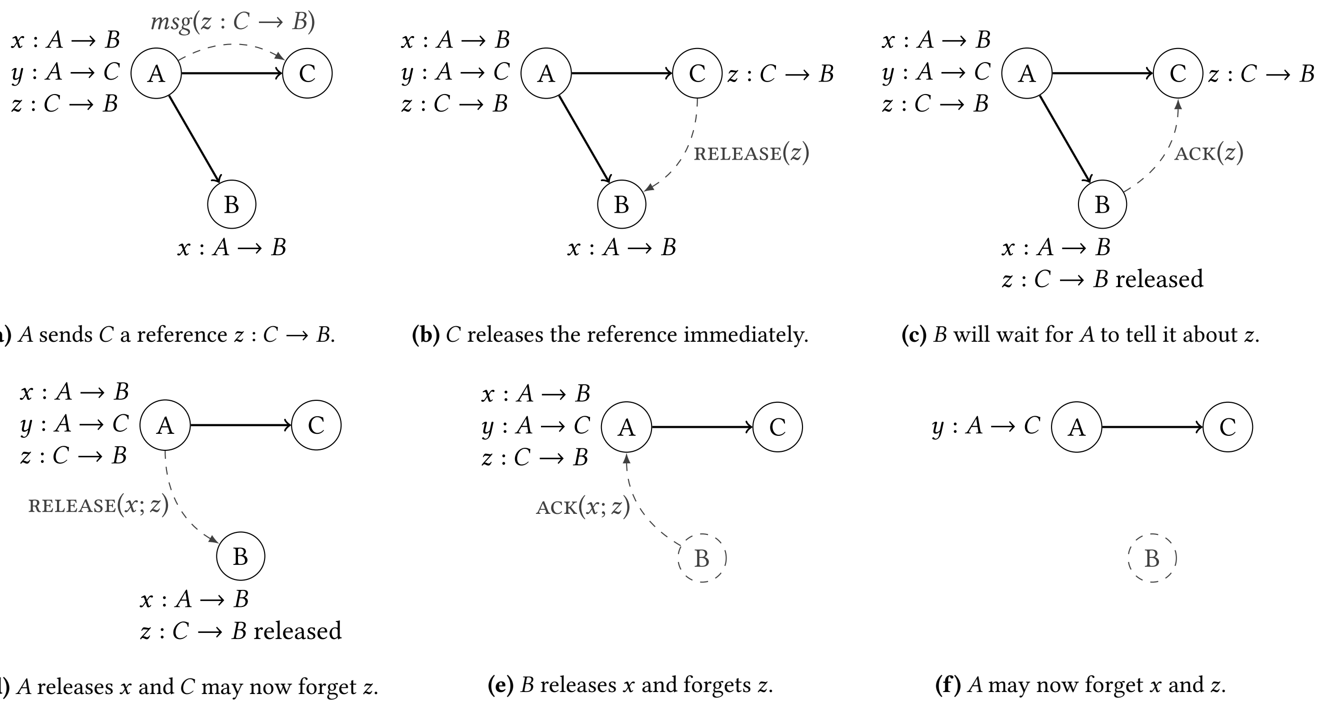
The authentication token serves to prevent the race condition in Fig. 2, where actors may accidentally release one another's references. The basic idea, represented in Fig. 3, is:

1. If C is releasing $z : C \to B$ but B hasn't learned about the creation of z yet, then B records the fact that z has been released locally. (Notice that a reference counting approach could not, in general, determine whether z has arrived at B yet.)
2. Although A could try to inform B about z immediately, we reason that C will not be garbage collected until $x : A \to B$ has been released. Consequently, it is sufficient to defer sending z up until the point when x is released. In general, the purpose of a RELEASE message will be both to release references and also to inform the target about new ones.
3. Even after sending a RELEASE message, the release protocol requires that actors do not forget about the



(a) *A* sends *C* a reference to *B*. **(b)** *C* releases the reference immediately. **(c)** *B* receives DEC() before INC() arrives.

Figure 2. Example of a race condition in naïve reference counting. Circles represent actors and "rc" is their corresponding reference count. Solid arrows represent existing references, while dashed grey arrows represent sent messages.



(a) *A* sends *C* a reference $z : C \longrightarrow B$. **(b)** *C* releases the reference immediately. **(c)** *B* will wait for *A* to tell it about *z*.

(d) *A* releases *x* and *C* may now forget *z*. **(e)** *B* releases *x* and forgets *z*. **(f)** *A* may now forget *x* and *z*.

Figure 3. Example of how proactive reference counting avoids race conditions. The annotations next to each actor indicate (a subset of) the references they know about. Initially, *A* knows it has references x, y to B, C respectively and also knows about $z : C \longrightarrow B$ after creating it. Next, *C* receives *z* and tries to release it, but will not forget *z* until it has received an acknowledgment. When *B* learns that *z* has been released, it saves this fact until it learns about *z* from *A*. Finally, *B* learns about *z* at the same time as it learns *x* has been released – at this point it has no more references and deletes itself.

references they create until they have received an *acknowledgment* message ACK_RELEASE. This makes the Chain Lemma possible, which will be essential to the garbage detection algorithm of Section 5.

We will only say that a reference $x : A \longrightarrow B$ is *released* once *B* has received the RELEASE message for *x*. Thus *A* does not know whether *x* is released until it receives the acknowledgment.

Points 2 and 3 lead to the following new requirements for our RTS:

Requirement 2: *The owner A of a reference $x : A \longrightarrow B$ is responsible for sending B every reference $y : C \longrightarrow B$ it creates while possessing x, and it must do so before all its own references to B are released.*

Requirement 3: *Actor A must keep track of every reference $x : C \longrightarrow B$ it creates until it sends x to B and receives an acknowledgment to that message.*

Listings 1 and 2 give expository pseudocode describing our RTS from the perspective of actor *A*.

Each actor *A* has the following fields:

Listing 1 User Interface

```
procedure ACTOR($b : B$)                                  ▹ Constructor
    ($a : A$) = new ActorName()
    ($x : B \longrightarrow A$) = new Reference()
    add $x$ to this.owners
    return $x$
                                                          ▹ Methods
procedure SPAWN( )
    ($a : A$) = this.name
    ($x : A \longrightarrow B$) = new Actor($a : A$)
    add $x$ to this.refs
    return $x$

procedure CREATE_REF($x : A \longrightarrow B, y : A \longrightarrow C$)
    ($z : C \longrightarrow B$) = new Reference()
    add $z$ to this.memory
    return $z$

procedure RECEIVE($x : A \longrightarrow B$)
    add $x$ to this.refs

procedure RELEASE($x : A \longrightarrow B$)
    remove $x$ from this.refs
    add $x$ to this.deactivated_refs
    if $\{y : A \longrightarrow B \mid y \in$ this.refs$\} = \emptyset$ then
        UPDATE_INFO($x$.target)
```

- *this.refs* tracks the references possessed by A.
- *this.deactivated_refs* tracks references that A is planning to release.
- *this.owners* tracks unreleased references known to A.
- *this.released_owners* contains *released* references $x : B \longrightarrow A$ where the creator of x has not told A about x.
- *this.memory* tracks all the references $x : C \longrightarrow B$ created by A that have yet to be sent to B.

Listing 1 defines the RTS interface. Actor constructors must be modified to return a reference instead of an actor name and to store this reference locally. Reciprocally, SPAWN shows how the creator must provide its actor name for the creation of that reference and store it locally once received.

Each time A wishes to share a reference to B with some C, it must create a new reference with MAKE_REF($x : A \longrightarrow B,\ y : A \longrightarrow C$). Note that y is necessary both for specifying the new owner of z and ensuring that the new reference $z : C \longrightarrow B$ can actually be sent to C.

Incoming messages containing a reference x must be handled by first invoking RECEIVE(x), and then invoking RELEASE(x) when no longer needed. The latter will in turn call UPDATE_INFO, triggering a release message to be sent to the target when there are no references to the target left in the local heap. This is only a suggestion; UPDATE_INFO can be called as (in)frequently as desired, leading to a user-controllable trade-off between reducing the size of *this.memory* and reducing message congestion.

Listing 2 Release Protocol *(without cycle detection)*

```
procedure UPDATE_INFO($b : B$)
    ($a : A$) = this.name
    $releasing$ = $\{x : A \longrightarrow B \mid x \in$ this.deactivated_refs$\}$
    remove $releasing$ from this.deactivated_refs

    $created$ = $\{x : C \longrightarrow B \mid x \in$ this.memory$\}$
    remove $created$ from this.memory

    $n$ = this.seqnum++
    this.on_ack[$n$] = $releasing \cup created$
    $b$ ! RELEASE_RECV($a : A, n, releasing, created$)

procedure RELEASE_RECV($a : A, n, releasing, created$)
    ($b : B$) = this.name
    for $x \in releasing$ do
        if $x \in$ this.owners then
            remove $x$ from this.owners
        else
            add $x$ to this.released_owners
    for $x \in created$ do
        if $x \in$ this.released_owners then
            remove $x$ from this.released_owners
        else
            add $x$ to this.owners
    $a$ ! ACK_RELEASE($b : B, n$)

procedure ACK_RELEASE($b : B, n$)
    delete this.on_ack[$n$]
```

Listing 2 implements the release protocol. It begins with a local method call to UPDATE_INFO with a target actor B. This gathers deactivated and recently created references and sends them asynchronously to B. To satisfy Requirement 3, the released references are stashed away locally until the acknowledgment is received by moving them from *memory* into *on_ack[n]*; the sequence number n is used to disambiguate acknowledgments, as there may be multiple concurrent release messages sent to the same target.

Upon receiving the release message, the target updates its state to account for the new information and sends an acknowledgment message. Once that has been received, the releasing actor may finally forget the references it held and created.

Let us say that A *knows* $x : C \longrightarrow B$ (where C or B may be equal to A) if x is in *A.refs, A.owners, A.memory*, or *A.on_ack[n]* for some n. The set of all such references is called A's *knowledge set.* If x is in *A.refs* or *A.owners*, then it is necessarily unreleased; otherwise, it may or may not be.

Not every unreleased reference targeting B is in B's knowledge set. However, since actors keep track of the references they create, the following lemma tells us that there is always a "path" from B to all of its owners by following a sequence of references:

Lemma 4.1. (Chain Lemma) *If $x : A \to B$ is an unreleased reference then there exists a finite, not necessarily unique, sequence of distinct actors $A_1, \dots, A_n$ and references $x_1, \dots, x_n$ such that $A = A_n$, $x = x_n$, and:*

$$\begin{aligned} B &\textit{ knows } (x_1 : A_1 \to B) \\ A_1 &\textit{ knows } (x_1 : A_1 \to B) \textit{ and } (x_2 : A_2 \to B) \\ &\dots \\ A_{n-1} &\textit{ knows } (x_{n-1} : A_{n-1} \to B) \textit{ and } (x_n : A_n \to B) \\ A_n &\textit{ knows } (x_n : A_n \to B) \end{aligned}$$

Proof. Routine induction on the events of the system, showing that each of the operations in Listings 1 and 2 preserves this property. Notice that without Requirement 3, some A_i triggering a release message to B would temporarily break the chain. □

It follows that *A.owners* is empty if and only if A has no incoming references and no undelivered messages from other actors. After A learns that it is quiescent, it releases all its references and may be destroyed once all the acknowledgments return. Thus, the RTS is sound and leaves only temporary "floating garbage" waiting to be reclaimed.

5 Garbage Collection

5.1 Reduction to Closed Set Detection

In this section we define *closed sets of actors* and show how the problem of actor garbage collection can be reduced to the detection of these closed sets. Crucial to our definitions is the notion of *connectedness*: the reflexive, transitive, and symmetric closure of the acquaintance relation. That is, A is *connected to* B if and only if there exists an actor A' such that A' is connected to B and either A can reach A' or A' can reach A.

A set G of actors is *strictly closed* at time t if, for each actor $A \in G$, every actor B connected to A is also in G. Equivalently, $In_t(G) \subseteq G$ and $Out_t(G) \subseteq G$, where:

$$In_t(G) = \{A \mid \exists B \in G,\ \exists x : A \to B \text{ unreleased at time } t\}$$

$$Out_t(G) = \{A \mid \exists B \in G,\ \exists x : B \to A \text{ unreleased at time } t\}$$

The *closure* of a collection of actors S is the smallest strictly closed set containing S.

Because references to existing actors are unforgeable, it follows that a strictly closed set G at time t can only evolve so that at time $t' > t$, we have $In_{t'}(G) \subseteq G^*$ and $Out_{t'}(G) \subseteq G^*$ where G^* is the closure of G under actor creation. At such a future time t', the set G is said to be *closed.*

Let us now classify garbage in terms of connectedness. From the definitions in Section 2, it follows that all garbage falls in one of the three categories:

1. Actors disconnected from the live set;
2. Quiescent actors connected to the live set;
3. Potentially unblocked actors that become disconnected from the live set after all quiescent actors are destroyed.

For an example of the latter, notice in Fig. 1 how actor 6 is live, but can neither reach nor be reached by the live set because it is only connected via the quiescent actor 5.

Now the problem of sound and complete garbage collection can be reduced to detecting closed and quiescent sets: Take G to be the smallest closed set containing the root set at time t. Then all actors outside of G and all quiescent actors inside of G are garbage, and may be deleted. This causes all garbage in the third category above to fall into the first category, which will be caught in the next GC iteration.

5.2 Closed Set Detection

An actor's *snapshot* at time t is the state of its knowledge set, as defined in Section 4, at time t. We would like to say that if a certain set of actor snapshots (all from distinct actors at distinct times) "appears closed", then the actors that took those snapshots indeed form a closed set. We shall find, after several modifications to the implementation in Listing 2, that this is indeed possible.

Below, we define the notion of a *closed set of snapshots* at time t, which we show in the next section corresponds to a closed set of actors at t.

Closure rule: A set G of snapshots is closed if and only if it is nonempty and the following holds: For each $x : B \to C$ in A's snapshot (where B or C may be the same as A), $C \in G$ and, unless C's snapshot indicates that x was released, $B \in G$ as well.

The intuition for this is straightforward: Any truly closed set containing an actor B should include all its owners and references. Since B does not know all its owners, we use the memory of owners already in G and the chain rule to determine the next owner to look for. If A_i knows about $x_{i+1} : A_{i+1} \to B$ but B knows, at the time of its snapshot, that x_{i+1} is released, then B also knows about all the references to it that were created using x_{i+1}. This makes it unnecessary to collect a snapshot from A_{i+1} unless another actor in G implicates it.

Notice that, from the definition, one can always determine a nonempty collection of actors that need to be added to a non-closed set in order to make it closed – this is in contrast to a system where one can tell that a set is not closed, but not whom to ask in order to make progress. Consequently, the following two algorithms for closed set detection can always make progress, and terminate so long as they proceed more quickly than actor creation.

Centralized approach: A designated actor requests a snapshot from each actor in the root set (in parallel). Based on each response, the Closure rule allows it to determine which actors (if any) should be queried next. This approach requires approximately $2N$ messages in the absence of topology changes, where N is the number of actors in the closure of the root set.

Decentralized approach: The initial actor A takes a snapshot and asks all its references and known owners to send their snapshots to A. Each actor, upon receiving the request, in turn asks all its neighbors to send snapshots to A. This approach requires more messages for highly connected graphs, but terminates more quickly.

5.3 Soundness

To ensure that a closed set of snapshots corresponds to a closed set of actors, we must add two new rules with corresponding changes to the release protocol shown in Listing 3.

The first rule prevents an actor's owners from forgetting the references they create before taking a snapshot. This prevents chains from being "broken" by references getting released during closed set detection.

Invalidation rule: If A is releasing its references to B and learns that B is taking a snapshot, then A must take a snapshot before forgetting its references to B.

To satisfy this rule, we add a new field *this.seqnum* in Listing 3. This field, initially 0, is incremented each time an actor responds to a snapshot request (of which there is only one for each GC pass). Thus, knowledge about whether B has taken a snapshot is obtained by piggybacking *B.seqnum* in the ACK_RELEASE message: If A has not yet received the latest snapshot request, then *A.seqnum* will be less than *B.seqnum.* When A detects this in ACK_RELEASE, it will cache its current snapshot to be subsequently relayed to the garbage collecting actor.

The next rule, essentially dual to Requirement 3, prevents an actor from sending new references to those that have already taken a snapshot and subsequently forgetting those same references before itself taking a snapshot. Thus references can no longer "slip through the cracks" of the closed set detection phase. Knowledge about whether B took a snapshot is obtained the same way as in the Invalidation rule.

Requirement 4: Actor A must keep track of every reference $x : B \to C$ it creates until it knows that B is not taking a snapshot.

Requirements 3 and 4 combine to mean that a reference $x : B \to C$ created by A cannot be forgotten until it has received acknowledgments from both B and C. This need not mean that either of the targets were in fact released, since an acknowledgment could be triggered by invoking the UPDATE_INFO method with an empty *releasing* set.

Listing 3 Release Protocol *(with cycle detection)*

```
procedure UPDATE_INFO($b : B$)
    ($a : A$) = this.name
    $releasing$ = $\{x : A \to B \mid x \in$ this.deactivated_refs$\}$
    remove $releasing$ from this.deactivated_refs

    $created$ = $\{x : C \to B \mid$
            $x \in$ this.memory, $\neg x$.target_releasing$\}$
    $sent$ = $\{x : B \to C \mid$
            $x \in$ this.memory, $\neg x$.owner_releasing$\}$
    for all $x \in created$ do $x$.target_releasing = True
    for all $x \in sent$ do $x$.owner_releasing = True

    $n$ = this.seqnum++
    this.on_ack[$n$] = $releasing \cup created \cup sent$
    $b$ ! RELEASE_RECV($a : A$, $n$, $releasing$, $created$)

procedure RELEASE_RECV($a : A$, $n$, $theirRefs$, $createdRefs$)
    . . .
    $a$ ! ACK_RELEASE($b : B$, $n$, this.latest_snapshot)

procedure ACK_RELEASE($b : B$, $n$, $snap\_id$)
    for all ($x : C \to B$) $\in$ this.on_ack[$n$], where $C \neq A$ do
        if $x$.owner_ack then
            remove $x$ from this.memory
        else
            $x$.target_ack = True
    for all ($x : B \to C$) $\in$ this.on_ack[n] do
        if $x$.target_ack then
            remove $x$ from this.memory
        else
            $x$.owner_ack = True
    if this.latest_snapshot < $snap\_id$ then
        this.snapshots[$snap\_id$] = SNAPSHOT( )
        this.latest_snapshot = $snap\_id$
    delete this.on_ack[$n$]
```

The principal challenge of closed set detection is to ensure that after an actor takes a snapshot, all subsequent references it receives are also to be in the closure. That is, it must be the case that $\forall t > t_B,\ Out_t(B) \subseteq G^*$, where t_B is the time of B's snapshot. The main idea of our soundness proof is the observation that the outgoing references of actors can be controlled by knowing about their incoming references, as formalized in the following key lemma.

(The following arguments make reference to a *global time*, by which we mean any total ordering of events compatible with the usual causation relation.)

Lemma 5.1. *Let $G_t \subseteq G$ be the set of actors that already took their snapshots before time t.*
If $In_t(G_t) \subseteq G^*$ then $Out_t(G_t) \subseteq G^*$.

Proof. We prove this by induction on events ordered by global time, starting from the the first snapshot of any actor in G.

If at time t some actor B is taking a snapshot, then the closure rule guarantees $In_t(\{B\}), Out_t(\{B\}) \subseteq G$. This also holds for every other actor in G_t, by the induction hypothesis.

If at time t, actor $B \in G$ spawned a new actor B' (and now $B' \in Out_t(\{B\})$), then by definition B' is in the creation closure G^*.

Suppose now that at time t, actor $B \in G_t$ receives $x : B \longrightarrow C$ from some A, where by hypothesis $A \in In_t(\{B\}) \subseteq G^*$.

If $A \in G \backslash G_t$ – that is, it has not yet taken a snapshot – then Requirement 4 guarantees that C will remember $x : B \longrightarrow C$ when it does take one.

Otherwise, $A \in G_t$. Then at the time t_s that x was sent, $C \in Out_{t_s}(\{A\})$. (At time t this is no longer guaranteed to be true, as A may have released C.)

1. If t_s was after A took a snapshot, then $C \in G^*$ by the induction hypothsis.
2. Otherwise, C must have been in A's memory at the time of its snapshot by Requirement 4, because at time t the source A still has a reference to B.

□

We may now prove the first half of soundness:

Lemma 5.2. *Let G be a closed set of snapshots at time T. Then $\forall T' > T$, $In_{T'}(G) \subseteq G^*$.*

Proof. The proof is by induction on events ordered by global time, starting from the first snapshot of any actor in G. We show for each time t that $In_t(G_t) \subseteq G^*$.

Suppose at time $t = t_B$ that $B \in G$ takes a snapshot. Then $In_{t_B}(\{B\}) \subseteq G$ follows from the closure rule, the chain lemma, and the invalidation rule.

Consider now the case where at time t a reference $y : C \longrightarrow B$ is created by actor A, where $B \in G_t$ and by hypothesis $A \in In_t(\{B\}) \subseteq G^*$. If A has not yet taken a snapshot, then Requirement 4 ensures that $C \in G$. Otherwise, it follows from Lemma 5.1 that $Out_t(\{A\}) \subseteq G^*$ and therefore $C \in G^*$. □

Theorem 5.3. (Soundness) *If G is a closed set of snapshots at time T, then the actors of G form a closed set.*

Proof. It suffices to show that $In_T(G) \subseteq G^*$ and $Out_T(G) \subseteq G^*$. The first fact was proved in Lemma 5.2. The second then follows from Lemma 5.1, since $G_T = G$. □

5.4 Quiescence Detection

Even if a closed set contains part of the root set, it may still contain garbage in the form of *quiescent actors*, as defined in Section 2. Fortunately, such subsets can be detected by storing the number of messages sent using $x : A \longrightarrow B$ and comparing it with the number processed by B. Such a count is also used to deal with unordered messages in Section 6.1.

Let G be a closed set of snapshots and $Q \subseteq G$ a set of snapshots which does intersect the root set and where $A \in Q$ implies every owner of A in G is also in Q. We argue by informal induction that if all the send/receive counts of snapshots in Q are in agreement, then the actors of Q are quiescent garbage.

Let A be the first actor in Q to have taken a snapshot, at time t_A. It must be blocked because all its owners are in Q, their snapshots come after t_A, and their message send counts are no higher than the receive counts of A. Consequently, A is only potentially unblocked if one of its owners is potentially unblocked.

For the next actor B we can make the same argument and observe:

- A could only send a message to B if one of A's owners became unblocked after t_A.
- B did not send any new messages to A during the interval (t_A, t_B).

Therefore B is only potentially unblocked if an owner of A or B in $Q \setminus \{A, B\}$ is potentially unblocked.

We may continue this way for each of the actors in Q and conclude that none of them are potentially unblocked. Consequently, the set is quiescent (and in fact it has been so since t_A).

6 Discussion

6.1 Unordered Messages

The algorithms above assume FIFO message ordering for simplicity and because most modern actor systems guarantee it. However, we can easily extend the algorithm to support unordered messaging, as in the theoretical Actor Model [2] and the SALSA language [15].

We can do this by attaching to each application-level message the token used to send that message. The sender counts the number of messages sent using that token, as does the receiver; note that the receiver need not know about the token beforehand. When the sender releases its references, it attaches the number of messages sent using those tokens. The target then delays processing the release message until it has received the expected number of messages.

6.2 Capabilities

Our treatment of reference tracking in Section 3 is richer than previous approaches like [11] because we assume that a reference can only be used by its designated owner. It is for this reason that actors cannot simply *duplicate* a reference as usual, and need to explicitly specify the recipient. While this restriction makes our definition slightly less flexible – and incompatible with existing RTS APIs – it is essential to proactive reference tracking, and we believe it captures the intended use case.

Our definition comes from the observation that references can be thought of as a kind of *capability* [7, 9]: A combination of an actor name (called the *designation*) with some promise that messages sent will be delivered (called the *authority* to access the target). Capabilities, like references, can only be created by those already in possession of them – initially just the target actor and its creator – and are usually only meant to be used by a specific actor. It therefore makes sense for the act of capability creation, often called *delegation*, to specify who that new owner is at the moment of creation. In Section 4, this information was used to trace *who gave authority to whom* and to create the "chains" of delegation formalized in the Chain Lemma.

6.3 Passive Objects

Although our model assumes no shared state, it is common in practice for actors colocated on the same node to pass data structures by reference instead of by copying. This leads to at least two complications:

1. What if actor names are stored in the data structure?
2. When can shared passive data be reclaimed?

The soundness of our garbage detection scheme is contingent on using references to communicate instead of actor names. Since distinct owners cannot share the same reference and references cannot be used after being released, any data structure containing references must provide a means for producing new references the same way actors do. For passive objects protected by a monitor, the simplest solution is to treat them as a kind of actor and endow them with the same methods and fields as Listings 1 and 3. On the other hand, lock-free data structures would require greater care to ensure compatibility with actor GC.

The second problem is somewhat of a non-issue, since passive garbage could presumably be collected normally (i.e. by sequential GC) when all the actors with references to it have themselves been destroyed. One must take care, however, to release all references stored in the data structure when this occurs.

7 Related Work

7.1 Garbage Collection of Actors

Tracing garbage collectors for sequential languages cannot be directly applied to actor systems because objects determined "unreachable" by the former may not be garbage at all. In a multicore setting this issue can be worked around by specially designing the runtime using "stages", as in SALSA Lite [6]. Another clever approach is to transform the actor graph in such a way that garbage actors coincide with garbage objects [14, 16]. However, doing so requires maintaining expensive inverse acquaintance references, so no practical GC employs this method.

Even though the Pony language targets multiprocessors, its designers took a different approach to the above due to concerns about cache coherency and synchronization [5]. Our intent in the present work has been to adapt their garbage collection scheme, MAC, to a distributed setting by removing the dependency on causal message ordering. Whereas proactive reference tracking methods are more expensive than the weighted reference counting used by MAC, this cost might be offset (even on a multiprocessor) by the reduced number of control messages and snapshots sent to the cycle detector. Our cycle detection scheme is fundamentally different than theirs, as is our definition of actor garbage (see Section 2).

To the best of our knowledge, the only complete and fault-tolerant distributed actor GC implemented is that of SALSA 1.0 [15]. Unfortunately the protocol introduces a very high message overhead for the maintenance of reference listing and for compatibility with the quiescence-based garbage collector, which needs to monitor actors for mutation. The overhead of our GC is significantly lower because proactive reference tracking uses the Chain Lemma to achieve the guarantees of reference listing, and because our garbage detection scheme is reduced to snapshot collection.

7.2 Distributed Reference Tracking

The contributions of this paper are made possible by the good properties of a new distributed reference tracking scheme we call *proactive reference tracking*, inspired by the novel reference counting technique introduced by Lee in a shared memory context [8]. We briefly survey previous approaches to distributed referencing below.

The naïve extension of reference counting to actors requires causal message delivery to ensure that *increment* messages are received in time. The *weighted reference counting* (WRC) optimization, introduced in [3, 17] can be used (as in [5]) to reduce the number of control messages. In fact WRC can be used to remove the causal restriction entirely, but would require the creation of indirection cells through which messages need to be routed.

Our scheme is superficially similar but essentially inverse to the *indirect reference counting* [10] and *stub-scion pair chain* (SSP) [13] approaches. Both remove the causal ordering requirement by having references contain pointers back to the actor that created them, producing an *inverted diffusion tree*. In these approaches the creator of the reference is responsible for its release, rather than the reference's target.

Our approach essentially uses (non-inverted) diffusion trees, where actors keep track of what references they create and to whom they are sent.

8 Future Work

We plan to implement the present work and evaluate its performance on distributed and multicore systems. In order to make it applicable to general-purpose applications, the message and memory overhead must be kept low. The data structures used to implement memory and the frequency of UPDATE_INFO invocations may depend on an actor's behavior, so this should be evaluated across a variety of different benchmarks.

Unfortunately, it is not possible to add our GC to an existing codebase without modifying any methods: Unlike actor names, references cannot be freely passed around. Creating a new reference requires passing both the target and owner actor names to its constructor, and all references received in messages must have RECEIVE_REF invoked upon them. (However, RELEASE(x) could be triggered automatically when x is determined to be garbage by ordinary actor-local GC.) Tools and languages that perform these labors automatically would significantly improve the usability of our GC.

Finally, the fault-tolerance and fault-recovery properties of our algorithm warrant study in their own right, as a distributed system should have the capacity to detect garbage despite hardware failure and message loss.

Acknowledgments

This material is based on work supported in part by the National Science Foundation under grant NSF CCF 16-17401.

We are indebted to the anonymous referees for providing excellent feedback on the original version of this paper.

References

[1] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. 1997. A Foundation for Actor Computation. *J. Funct. Program.* 7, 1 (1997), 1–72. http://journals.cambridge.org/action/displayAbstract?aid=44065

[2] Gul A. Agha. 1990. *ACTORS - a model of concurrent computation in distributed systems.* MIT Press.

[3] David I Bevan. 1987. Distributed garbage collection using reference counting. In *International Conference on Parallel Architectures and Languages Europe.* Springer, 176–187.

[4] Sebastian Blessing, Sylvan Clebsch, and Sophia Drossopoulou. 2017. Tree topologies for causal message delivery. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2017, Vancouver, BC, Canada, October 23 - 27, 2017.* 1–10. https://doi.org/10.1145/3141834.3141835

[5] Sylvan Clebsch and Sophia Drossopoulou. 2013. Fully concurrent garbage collection of actors on many-core machines. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013.* 553–570. https://doi.org/10.1145/2509136.2509557

[6] Travis J. Desell and Carlos A. Varela. 2014. SALSA Lite: A Hash-Based Actor Runtime for Efficient Local Concurrency. In *Concurrent Objects and Beyond - Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday.* 144–166. https://doi.org/10.1007/978-3-662-44471-9_7

[7] Sophia Drossopoulou and James Noble. 2014. How to Break the Bank: Semantics of Capability Policies. In *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings.* 18–35. https://doi.org/10.1007/978-3-319-10181-1_2

[8] Hyonho Lee. 2010. Fast Local-Spin Abortable Mutual Exclusion with Bounded Space. In *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings.* 364–379. https://doi.org/10.1007/978-3-642-17653-1_27

[9] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. 2003. *Capability myths demolished.* Technical Report. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals.

[10] José M. Piquer. 1991. Indirect Reference Counting: A Distributed Garbage Collection Algorithm. In *PARLE '91: Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms, Eindhoven, The Netherlands, June 10-13, 1991, Proceedings.* 150–165. https://doi.org/10.1007/BFb0035102

[11] David Plainfossé and Marc Shapiro. 1995. A Survey of Distributed Garbage Collection Techniques. In *Memory Management, International Workshop IWMM 95, Kinross, UK, September 27-29, 1995, Proceedings.* 211–249. https://doi.org/10.1007/3-540-60368-9_26

[12] André Schiper, Jorge Eggli, and Alain Sandoz. 1989. A New Algorithm to Implement Causal Ordering. In *Distributed Algorithms, 3rd International Workshop, Nice, France, September 26-28, 1989, Proceedings.* 219–232. https://doi.org/10.1007/3-540-51687-5_45

[13] Marc Shapiro, Peter Dickman, and David Plainfossé. 1992. *SSP chains: Robust, distributed references supporting acyclic garbage collection.* Ph.D. Dissertation. inria.

[14] Abhay Vardhan and Gul Agha. 2002. Using passive object garbage collection algorithms for garbage collection of active objects. In *Proceedings of The Workshop on Memory Systems Performance (MSP 2002), June 16, 2002 and The International Symposium on Memory Management (ISMM 2002), June 20-21, 2002, Berlin, Germany.* 213–220. https://doi.org/10.1145/773039.512443

[15] Wei-Jen Wang and Carlos A. Varela. 2006. Distributed Garbage Collection for Mobile Actor Systems: The Pseudo Root Approach. In *Advances in Grid and Pervasive Computing, First International Conference, GPC 2006, Taichung, Taiwan, May 3-5, 2006, Proceedings.* 360–372. https://doi.org/10.1007/11745693_36

[16] Wei-Jen Wang, Carlos A. Varela, Fu-Hau Hsu, and Cheng-Hsien Tang. 2010. Actor Garbage Collection Using Vertex-Preserving Actor-to-Object Graph Transformations. In *Advances in Grid and Pervasive Computing, 5th International Conference, GPC 2010, Hualien, Taiwan, May 10-13, 2010. Proceedings.* 244–255. https://doi.org/10.1007/978-3-642-13067-0_28

[17] Paul Watson and Ian Watson. 1987. An Efficient Garbage Collection Scheme for Parallel Computer Architectures. In *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 15-19, 1987, Proceedings.* 432–443. https://doi.org/10.1007/3-540-17945-3_25