Fully Concurrent Garbage Collection of Actors on Many-Core Machines

Sylvan Clebsch and Sophia Drossopoulou
Department of Computing, Imperial College, London
{sc5511, scd}@doc.ic.ac.uk

Abstract

Disposal of dead actors in actor-model languages is as important as disposal of unreachable objects in object-oriented languages. In current practice, programmers are required to either manually terminate actors, or they have to rely on garbage collection systems that monitor actor mutation through write barriers, thread coordination through locks etc. These techniques, however, prevent the collector from being fully concurrent.

We developed a protocol that allows garbage collection to run fully concurrently with all actors. The main challenges in concurrent garbage collection is the detection of cycles of sleeping actors in the actors graph, in the presence of concurrent mutation of this graph. Our protocol is solely built on message passing: it uses deferred direct reference counting, a dedicated actor for the detection of (cyclic) garbage, and a confirmation protocol (to deal with the mutation of the actor graph).

We present our ideas informally through an example, and then present a formal model, prove soundness and argue completeness. We have implemented the protocol as part of a runtime library. As a preliminary performance evaluation, we discuss the performance of our approach as currently used at a financial institution, and use four benchmarks from the literature to compare our approach with other actor-model systems. These preliminary results indicate that the overhead of our approach is small.

Categories and Subject Descriptors D.3.2 [**Programming Languages**]: Language Classifications - concurrent, distributed, and parallel languages; D.3.4 [**Programming Languages**]: Processors - Memory management (garbage collection).

Keywords actors; message passing; concurrency; many-core; garbage collection

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2374-1/13/10. . . $15.00.
http://dx.doi.org/10.1145/2509136.2509557

1. Introduction

The actor-model uses actors as the unit of computation. Actors encapsulate messaging, memory, and a thread of execution into a single entity, providing a powerful model for concurrent computation [1, 2].

Actor-model languages must know when an actor has terminated in order to free resources dedicated to the actor. Most existing actor-model languages and libraries do not attempt to solve this problem, instead requiring the programmer to explicitly manage every actor's lifetime [3–7]. The languages that do garbage collect actors require hardware features that adversely impact performance, such as cache coherency, expensive software techniques that may require hardware support, such as mutation monitoring through write barriers, and use approaches that are not based on the message passing paradigm at the heart of the actor-model [14, 15]. The very problems that actor-model programming excels at addressing (including concurrency, scalability, and simplicity) have made actor garbage collection problematic, due to the difficulty of observing the global state of a program. As a result, actor-model systems in applications which create many short-lived actors become either more difficult to program (when they require manually terminating actors) or encounter performance problems (when they have actor garbage collection that is not fully concurrent).

A language which does not provide garbage collection of actors will require a facility to explicitly terminate actors. This will also require the language to provide a default behaviour when a message is sent to a terminated actor, the ability to distinguish at runtime between terminated and non-terminated actors, and possibly notification mechanisms for actor termination.

In this paper, we present a technique for garbage collection of actors, which we call Message-based Actor Collection (*MAC*), that satisfies the following goals:

1. *Soundness:* the technique collects only dead actors.
2. *Completeness:* the technique collects all dead actors eventually.
3. *Concurrency:* the technique does not require a stop-the-world step, thread coordination, actor introspection, shared memory, read/write barriers or cache coherency.

When an actor has completed local execution and has no pending messages on its queue, it is *blocked*. An actor is *dead* if it is blocked and all actors that have a reference to it are blocked, transitively. Collection of dead actors depends on being able to collect closed cycles of blocked actors.

Our approach is inspired by previous work on distributed garbage collection of passive objects using distributed reference counting and a secondary mechanism to collect cyclic garbage [22–25]. Detection of cycles of objects is based on their *topology,* which is essentially the number of incoming references and the identities of all outgoing references. We adopt this approach so that the topology of an actor consists of the number of incoming references from actors, the set of outgoing references to actors, and a flag indicating whether the actor is blocked. A dedicated actor, called the *cycle detector*, keeps track of the actor topology and detects any cycles.

The challenge we face is that the true topology of an actor is a concept distributed across all of the actors: it changes not only when the actor mutates, but also when other actors mutate. An actor's view of its topology may be out of sync with the true topology, and the cycle detector's view of an actor's topology may be out of sync with the actor's view of its topology. This differs radically from previous work on distributed object cycle detection, where objects must either be immutable or cycle detection must monitor mutation [26–28].

Our technique uses the message passing paradigm at the heart of the actor-model: when an actor blocks, it sends a snapshot of its view of its topology to the cycle detector. The cycle detector in turn detects cycles based on its view of the topology of blocked actors. Because the cycle detector operates on its own view of the blocked actor topology rather than stopping execution or monitoring mutation, cycles may be detected based on a view of the topology that is out of date. This is overcome with a confirmation protocol that allows the cycle detector to determine whether or not its view of the blocked actor topology is the same as the true topology, without stopping execution, monitoring mutation, or examining any actor's heap.

Contributions The key contribution of this paper is a system for efficient concurrent garbage collection of actors. More specifically, we present:

- An informal explanation of Message-based Actor Collection (*MAC*).
- A formal model of garbage collection with *MAC* expressed through an operational semantics.
- A proof of soundness for *MAC*.
- Preliminary performance results which indicate *MAC* has a small overhead over manual collection - if any.

Garbage collection systems are often presented without soundness proofs [8, 9, 16, 18, 26–28]. Where proofs are provided, they often do not address cyclic garbage [24] or mutation [22, 23], or require synchronous collection [10]. We developed suitable abstractions to be able to make our soundness proof. These abstractions also helped us develop a simpler presentation of protocol based on the consistency of the perceived topologies.

Outline We discuss the background on garbage collection of actors in section 3. We present the design of our system informally in section 4, formalise it in section 5, and provide a proof of soundness in section 6. We report on our implementation in section 7, and conclude and discuss further work in section 8.

2. Motivation

Even though actors are extensively used in the distributed setting, they can address massively concurrent programming, a major challenge currently, attracting a significant amount of research. Moreover, actors are often used without distribution. For example, in [11] the software from repository [12] is studied. From around 750 programs, 16 are isolated as representative of "real-world actor programs". Of these, only 7 are distributed applications. Of these 7, only 3 use remote actors for distributed computation.

Within the concurrent setting, our work is best applicable to the style of concurrency where a multitude of lightweight actors are continuously created and discarded, rather than where actors are a few large entities that logically persist during program execution (e.g. vats). Applications of the latter style have less need for actor GC. Applications of the former style of concurrency are encountered in, for example, trading applications, social simulations, and network traffic analysis, and motivate the need to reclaim actors. Our system is currently in use in such an application (cf. section 7). Moreover, such a style of concurrency will be supported by the many cores forecast in hardware development [13].

3. Background on Garbage Collection of Actors

Actor Collection Existing actor-model languages and libraries use three approaches to garbage collection of actors.

The first approach is to require the programmer to manually terminate actors. Many existing actor-model languages and libraries, such as Erlang [3], Scala [4], AmbientTalk [5], SALSA 2.0 [6], Kilim [7], and Akka, do not garbage collect actors at all. All of these except Kilim support actors on

distributed nodes, although only SALSA supports manual migration of actors to new nodes. None support distributed scheduling or automatic migration.

The second approach is to transform the actor graph into an object graph and use a tracing garbage collector to collect actors [8–10], as done in ActorFoundry [14]. This requires shared memory, cache coherency, and a stop-the-world step. This approach allows actors to be collected using the same collector used for passive objects, but cannot be used across distributed nodes.

The third approach, used in SALSA 1.0 [15], uses reference listing (whereby an actor keeps a complete list of every other actor that references it) and monitoring of actor mutation to build conservative local snapshots which are assembled into a global snapshot. This requires write barriers for actor mutation (which requires shared memory and cache coherency), a global synchronisation agent, and coordination of local snapshots within an overlapping time range. These snapshots are used with the pseudo-root algorithm, which additionally requires acknowledgement messages for all asynchronous messages, inverse reference listing, and a multiple-message protocol for reference passing [16–18]. Like SALSA 2.0, SALSA 1.0 supports distributed nodes and manual actor migration.

None of these approaches provides a fully concurrent method for garbage collection of actors.

Distributed Passive Object Collection The literature on distributed passive object collection is vast, and so we will only briefly mention key differences. Our approach has been inspired not just by previous work in actor collection, but also by work in concurrent cycle detection [23] and distributed reference counting [22, 24–28] for passive object collection. Some of these approaches do not address cyclic garbage [24, 25]. Others require either immutable passive objects or a synchronisation mechanism between the cycle detector and the mutator, which makes them inapplicable to actor collection [22, 23, 26–28].

MAC differs significantly from previous work. Unlike distributed passive object collection, no restrictions on actor mutation or monitoring of mutation are required in order to detect cyclic garbage, and no reference listing, indirection cells, or diffusion trees (a technique whereby nodes keep a trail of object references they have passed, which can lead to zombie nodes) are required. Unlike the pseudo-root approach, acknowledgement messages are only required when actors are actually collected, no reference listing is required, no message round-trips are required, and no snapshot integration or time ranges are required. As a result, *MAC* requires significantly less overhead. Because *MAC* does not require thread coordination or cache coherency, it does not become less efficient as core count increases.

Some approaches to distributed passive object collection are fault-tolerant [21]. In order to make distributed garbage collection fault-tolerant, it is necessary to detect and handle failure and often also to track a global view of time. Our work is targeted at the many-core environment rather than the distributed environment and relies on guaranteed message delivery, which obviates the need for failure detection. In addition, our reliance on *causal messaging* (cf. section 4) obviates the need for a global view of time.

4. Message-based Actor Collection

In this section we explain Message-based Actor Collection (*MAC*) informally. We introduce our additions on top of the actor-model, including *causal messaging*, *external sets* for tracking potentially reachable actors, and the reference count invariant and the protocol for maintaining it. We also introduce our *cycle detector*, including *perceived cycles* for detecting possible cycles, and the *conf-ack protocol* for confirming perceived cycles. The operational semantics of *MAC* are formalised in section 5, and a proof of soundness is provided in section 6.

Actors The actor-model stipulates that an actor can [1, 2]:

1. Send a finite number of asynchronous, buffered messages to other actors, with guaranteed delivery but no ordering or fairness guarantees.
2. Select a behaviour to be executed in response to the next message.
3. Create a finite number of new actors.

We additionally require that an actor's message queue is FIFO ordered, and message delivery is *causal* (defined below). Moreover, each actor has a local heap. In this paper, we are only concerned with actor references, but in general the heap would also contain passive objects, and an actor would have a stack while performing local execution.

Application Messages We model application-level messages as a single message type (*APP*) that allows an actor ι_1 to send a set of actors ιs to another actor ι_2. In general, there would be multiple application message types, which could contain passive objects as well as actors.

Topology The *true topology* of the system is the directed graph of actor reachability. Because actors execute concurrently, it is not possible to efficiently track the true topology. Instead, each actor maintains a view of its own topology, consisting of a reference count (indicating the number of incoming graph edges) and an *external set* of potentially reachable actors (the outgoing edges).

The actor's view can disagree with the true topology. When an actor ι_1 sends a reference to itself to another actor ι_2, it can immediately update its reference count, maintaining agreement with the true topology. However, if ι_2 drops its reference to ι_1, ι_2 cannot directly mutate ι_1's reference count. Now ι_1's reference count is out of sync. To correct this, ι_2 sends a reference count decrement message (*DEC*) to ι_1. When ι_1 processes that message, it updates its view to restore agreement with the true topology.

Similarly, if ι_2 sends a reference to ι_1 to a third actor ι_3, it first sends a reference count increment message (*INC*) to ι_1. This *INC* represents the reference to ι_1 held by the message.

These *INC* and *DEC* messages allow the actor's view of its topology to be eventually consistent with the true topology.

Deferred Reference Counting The *external set* is an over-approximation of the set of actor references contained in some actor's heap. It differs from the heap in order to allow reference counting to be lazy. Rather than tracking all references from ι_1 to ι_2, a single reference exists if ι_2 appears one or more times in ι_1's heap. The external set contains all actors that have been in the actor's heap or received in a message since the last local garbage collection cycle. When an actor performs local garbage collection, the external set is compacted so as to contain only the actors remaining in the heap. Actors removed from the external set when it is compacted represent dropped references, and are sent *DEC*.

Similarly, when an actor ι_1 receives another actor ι_2 in a message, ι_1 adds ι_2 to its external set. If ι_2 is not present in ι_1's external set, the reference held by the message is transferred to ι_1 and ι_2's view of its topology remains in agreement with the true topology. If ι_2 is already present in ι_1's external set, ι_1 already has an outgoing edge to ι_2. To maintain the reference count invariant of ι_2, ι_1 sends *DEC* to ι_2, which allows ι_2 to eventually update its view of its own topology.

Our approach is based on, but differs from, deferred increments [19], where ephemeral reference count updates can be skipped, and update coalescing, where redundant reference count updates are combined for efficiency [20]. In our work, reference counts are not updated when references are created or destroyed on the stack or in the heap, but only when references are sent in messages and when local garbage collection indicates no references to an actor remain in a heap. The messages act as the mechanism for deferring increments and the external set in combination with local garbage collection acts as the mechanism for coalescing updates.

Cycle Detection As in any reference counting system, cyclic garbage cannot be collected by reference counting alone. Our system uses a *cycle detector* that has a message queue like an actor, and can both send and receive messages.

When an actor has no pending messages on its queue, it is *blocked*. When an actor blocks, it sends a block message (*BLK*) to the cycle detector containing the actor's view of its topology, i.e. its reference count and its external set. When a blocked actor processes a message, it becomes *unblocked* and sends an unblock message (*UNB*) to the cycle detector, informing the cycle detector that its view of that actor's topology is invalid and that actor is no longer blocked.

This allows the cycle detector to maintain a view of the topology of all blocked actors that is eventually consistent with each actor's view of its topology, which is in turn eventually consistent with the true topology.

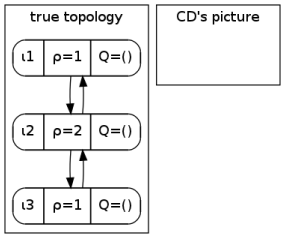
It would be possible but not efficient for application actors to perform cycle detection when no messages are pending on their queue (i.e. just before blocking): this would require every actor in the system to maintain a view of every other actor's topology, which for n actors would require n messages upon each block and unblock and duplication of blocked actor topology in every actor. A separate cycle detector reduces this to one message upon block or unblock regardless of the number of actors.

Dead Actors An actor is *dead* if it is blocked and all actors that have a reference to it are blocked, transitively. Because messaging is required to be *causal* (defined below), a blocked actor with a reference count of zero is unreachable by any other actor and is therefore dead (acyclic garbage).

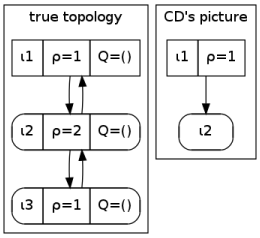
For cyclic garbage, the cycle detector uses a standard cycle detection algorithm to find isolated cycles in its view of the topology of blocked actors. However, the cycle detector's view of the topology may disagree with an actor's view of its topology (when a *BLK* or *UNB* message is on the cycle detector's queue but as yet unprocessed), and the actor's view of its topology may in turn disagree with the true topology (when an *INC* or *DEC* message is on the actor's queue but as yet unprocessed). If cyclic garbage is detected on the basis of a view of the topology that disagrees with the true topology, that cycle must not be collected. We call a cycle that has been detected a *perceived cycle* and a cycle that has been detected using a view of the topology that agrees with the true topology a *true cycle.*

Example 1. A perceived cycle that is not a true cycle. This is shown in figure 1.

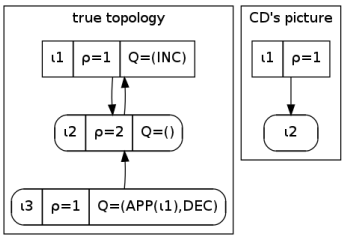
1. Given three actors (ι_1, ι_2 and ι_3), ι_1 and ι_2 reference each other and ι_2 and ι_3 reference each other.
2. ι_1 blocks, sending $BLK(\iota_1, 1, \{\iota_2\})$ to the cycle detector. When the cycle detector processes this, its view of the topology becomes $[\iota_1 \mapsto (1, \{\iota_2\})]$.
3. ι_2 wishes to send a reference to ι_1 to ι_3. It sends INC to ι_1 and then $APP(\iota_1)$ to ι_3. ι_2 then drops its reference to ι_3 , collects garbage locally, and sends DEC to ι_3. The cycle detector's view of the topology does not change.
4. ι_3 processes $APP(\iota_1)$, adding ι_1 to its external set. ι_3 then drops its reference to ι_2 , collects garbage locally, and sends DEC to ι_2.The cycle detector's view of the topology does not change.
5. ι_2 processes DEC, then blocks, sending $BLK(\iota_2, 1, \{\iota_1\})$ to the cycle detector. When the cycle detector processes this, its view of the topology becomes $[\iota_1 \mapsto (1, \{\iota_2\}), \iota_2 \mapsto (1, \{\iota_1\})]$.
6. The cycle detector perceives a cycle $\{\iota_1, \iota_2\}$, even though ι_1 is reachable from ι_3. This is because ι_1 has a pending INC that it has not processed.



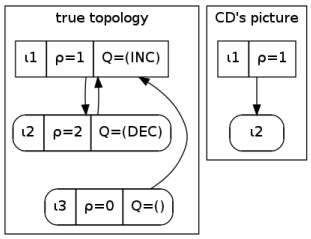
(a) Initial state, as in step 1



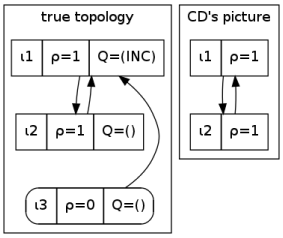
(b) ι_1 blocks, as in step 2



(c) ι_2 sends $\iota_3 \leftarrow APP(\iota_1)$ and drops ι_3, as in step 3



(d) ι_3 processes $APP(\iota_1)$ and drops ι_2, as in step 5



(e) ι_2 blocks, as in step 5. The perceived cycle is incorrect due to ι_1's pending *INC*.

Figure 1: Diagram of example 1. Boxes display the reference count (ρ), and queue (Q) of actors, with round corners indicating unblocked and square corners indicating blocked. The arrows indicate references, eg. ι_1 references ι_2, which implicitly shows the external set.

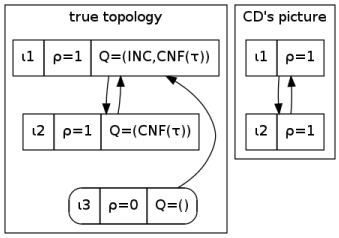
Conf-Ack Protocol When a perceived cycle is detected, the cycle detector must determine whether or not the view of the topology used to detect the cycle agrees with the true topology. To do so, we introduce a *conf-ack* step to our protocol. When the cycle detector detects a perceived cycle, it sends a confirm message (*CNF*) with a token uniquely identifying the perceived cycle to each actor in the cycle. When an actor receives *CNF*, it sends an acknowledgement message (*ACK*) with the token to the cycle detector without regard to the actor's view of its topology.

If the cycle detector receives *ACK* from an actor in a perceived cycle without receiving *UNB*, then that actor did not unblock between blocking and the detection of the perceived cycle, which tells us that the actor's view of its topology when the perceived cycle was detected was the same as the cycle detector's view of that actor's topology used to detect the perceived cycle. Such an actor is *confirmed.* Conversely, if an actor in a cycle changes state, it will send *UNB* before it sends *ACK*. Because messaging is causal, the cycle detector will receive the *UNB* before it receives the *ACK*. When the cycle detector receives *UNB* for an actor, it cancels all perceived cycles containing the newly unblocked actor, since they were detected with an incorrect view of that actor's topology.

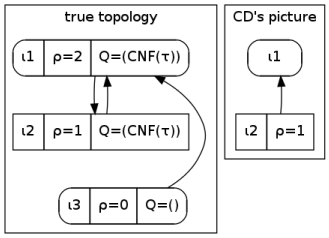
Further, if all actors in a perceived cycle are confirmed, then, at the time the cycle was detected, each actor in the cycle had a view of its topology that agreed with the true topology. As a result, the perceived cycle is a true cycle and can be collected.

Example 2. Expanding example 1 with the conf-ack protocol. This is shown in figure 2.

7. The cycle detector sends $CNF(\tau)$ to ι_1 and ι_2, where τ is a token uniquely identifying this perceived cycle.



(a) Cycle detector sends *CNF*, as in step 6.



(b) ι_1 unblocks, as in step 8. The perceived cycle is correctly cancelled, as in step 9.

Figure 2: Diagram of example 2. Boxes display the reference count (ρ), and queue (Q) of actors, with round corners indicating unblocked and square corners indicating blocked. The arrows indicate references, eg. ι_1 references ι_2, which implicitly shows the external set.

8. ι_1 processes the pending INC from example 1 before $CNF(\tau)$, due to causal messaging, and sends $UNB(\iota_1)$ to κ.
9. ι_1 processes $CNF(\tau)$ and sends $ACK(\iota_1, \tau)$ to the cycle detector.
10. κ processes $UNB(\iota_1)$ before $ACK(\iota_1, \tau)$, due to causal messaging, and correctly cancels the perceived cycle.

Explanation The conf-ack protocol works by providing the cycle detector with confirmation that the view of the topology used to detect a cycle (which was sent to the cycle detector as a snapshot of each actor's view of its topology) agreed with the true topology when the cycle was detected. This approach allows the cycle detector to work concurrently with other actors, without shared memory, locks, read/write barriers, cache coherency, or any other form of thread-coordination.

Causal Messaging In order to maintain the actor's reference count invariant, message delivery must be *causal.* When an actor ι_1 sends *INC* to an actor ι_2 before including it in a message to an actor ι_3, ι_2 must process that *INC* before any *DEC* message sent by ι_3. Each message is an *effect,* and every message the sending actor has previously sent or received is a *cause* of that effect. Messaging is causal if every cause is enqueued before the effect. Causality propagates forward: the causes of an effect are also causes for any secondary effect.

Example 3. Causal messaging.

1. ι_1 sends msg_1 to ι_2.
2. ι_1 sends msg_2 to ι_3 .
3. After receiving msg_2, ι_3 sends msg_3 to ι_2.
4. To preserve causality, ι_2 must receive msg_1 before msg_3.

Causality is easy to achieve in a many-core setting. Sending a message and enqueuing it at the destination can be done with a single atomic operation. As a result, causality is a natural consequence of lock-free, wait-free FIFO message queues, and has no overhead.

Consistency Model Our approach requires only weak memory consistency. In particular, when a message is sent, all writes to the contents of the message must be visible to the receiver of the message. This can be implemented with a release barrier on message send. On the x86 architecture, this release barrier is implicit on all writes, so no fence is required. Moreover, because *MAC* requires no shared memory other than the contents of messages, no consistency model is necessary for other writes, e.g. when the cycle detector updates its view of blocked actor topology.

5. Formal Model

In this section we present a formal model, expressed as an operational semantics for *MAC*. Types and identifier conventions are presented in figure 3, and the steps that rewrite the configuration are presented in figures 4 to 7.

A configuration contains a queue, a cycle detector and a set of actors. While an actor logically contains its own queue, we represent the queue as a global entity that maps an actor ID to a message sequence.

The cycle detector is composed of the cycle detector's view of the blocked actor topology (*PT*), the set of perceived cycles that are awaiting confirmation (*PC*), and the next token that will be used to identify a perceived cycle (τ). The cycle detector's identifier is κ.

Each actor is composed of its identifier, a view of its topology (ρ and ξ), its heap, and a flag indicating whether or not it is blocked. We treat an actor's heap as a set of actor IDs for convenience, but it stands in for a normal heap.

Messages can be sent and received by actors and the cycle detector. Each message is composed of a message identifier and arguments. The *APP* message represents all application level messages that are sent by actors, and its parameter represents the set of actor identifiers included in the message. All of the other messages are internal. They are used to describe the protocol, but would not be exposed in a programming language that used *MAC*.

$$
\begin{array}{rcll}
cfg & \in & Configuration & = & Queue \times CycleDetector \times Actors \\
Q & \in & Queue & = & (ID \cup \{\kappa\}) \rightarrow (Message)* \\
 & & Message & = & APP(\iota s)|INC|DEC| \\
 & & & & BLK(\iota, \rho, \xi)|UNB(\iota)|CNF(\tau)|ACK(\iota, \tau) \\
CD & \in & CycleDetector & = & PerceivedTopo \times PerceivedCycles \times Token \\
\kappa & \in & CycleDetectorID & & \\
PT & \in & PerceivedTopo & = & ID \rightarrow (RefCount \times ExSet) \\
PC & \in & PerceivedCycles & = & Token \rightarrow (ID \rightarrow Boolean) \\
as & \in & Actors & = & \mathcal{P}(Actor) \\
a & \in & Actor & = & ID \times RefCount \times ExSet \times Heap \times Blocked \\
\iota & \in & ID & & \\
\iota s & \in & IDs & = & \mathcal{P}(ID) \\
\xi & \in & ExSet & = & \mathcal{P}(ID) \\
h & \in & Heap & = & \mathcal{P}(ID) \\
\rho & \in & RefCount & = & Integer \\
\beta & \in & Blocked & = & Boolean \\
\tau & \in & Token & = & Integer
\end{array}
$$

Figure 3: Types and identifier conventions

In example 1, the initial configuration looks like this:

$$
\begin{array}{rcl}
cfg_1 & = & (Q_1, CD_1, \{a_1, a_2, a_3\})\, where \\
Q_1 & = & \varepsilon \\
CD_1 & = & (\varepsilon, \varepsilon, 0) \\
a_1 & = & (\iota_1, 1, \{\iota_2\}, \{\iota_2\}, false) \\
a_2 & = & (\iota_2, 2, \{\iota_1, \iota_3\}, \{\iota_1, \iota_3\}, false) \\
a_3 & = & (\iota_3, 1, \{\iota_2\}, \{\iota_2\}, false)
\end{array}
$$

Notation In this paper, we make use of some additional notation for convenience.

- We treat values in the context of sets as singleton sets, eg. $\xi \cup \iota$, $\iota s \setminus \iota$ have the expected meaning.
- We use set operations on the domains of mappings.
  - $x \in map \Leftrightarrow x \in dom(map)$
  - $map \setminus \{x_1..x_n\} \triangleq map[x_1 \mapsto \bot, ..x_n \mapsto \bot]$
- We use set operations between actors and actor identifiers.
  - $as \setminus \iota s \triangleq as \setminus \{a | a = (\iota, \_, \_, \_, \_) \wedge \iota \in \iota s\}$
  - $\iota \in as \Leftrightarrow (\iota, \_, \_, \_, \_) \in as$
- We use an index operation to examine a queue and an append operation to modify a queue.
  - $Q(\iota)[k]$ is the k^{th} message on $Q(\iota)$.
  - $Q(\iota){+}{+}msg$ appends msg to the end of $Q(\iota)$.
- We use $Push$, Pop, and $Unblock$ to manipulate the queue.
  - $Push(Q, \{\iota_1..\iota_n\}, msg) \triangleq$ $Q[\iota_1 \mapsto Q(\iota_1){+}{+}msg, ..\iota_n \mapsto Q(\iota_n){+}{+}msg]$
  - $Pop(Q, \iota) \triangleq Q', msg$ where $Q(\iota) = msg : rest$ and $Q' = Q[\iota \mapsto rest]$
  - $Unblock(Q, \iota, \beta) \triangleq \begin{cases} Push(Q, \kappa, UNB(\iota)) & if\ \beta \\ Q & if\ \neg\beta \end{cases}$
- We use $Closed$ to refer to a closed cycle of blocked actors in a perceived topology.
  - $Closed(\iota s, PT) \Leftrightarrow \left(\begin{array}{c} \forall \iota \in \iota s : \\ \forall \iota'. \iota \in PT(\iota') \downarrow_2 \rightarrow \iota' \in \iota s \wedge \\ PT(\iota) \downarrow_1 = \\ |\{\iota' | \iota' \in \iota s,\ \iota \in PT(\iota') \downarrow_2\}| \end{array} \right)$

We guarantee causality with FIFO message queues that provide both guaranteed and atomic delivery. This is expressed in the operational semantics by using a single operation on the queue ($Push$) to both send and enqueue a message. Using an intermediate container of messages that have been sent by an actor but not yet enqueued by the receiving actor would make delivery non-atomic, even though messages would still be FIFO ordered.

We will now discuss the operational semantics.

Actor Local Execution Rather than present a programming language for actors, the rules in figure 4 describe the effects of local execution on the entities of our protocol. As usual in concurrency, execution is non-deterministic. In each rule, the active actor is indicated by $(\iota, \rho, \xi, h, false)$.

CREATE Create a new actor. The newly created actor a' has identifier ι', a reference count of one (because the creating actor ι has a reference to it), an empty external set and heap, and is unblocked. The new actor a' is added to the set of actors, and its identifier ι' is added to the external set of the active actor.

SEND Send an *APP* message, possibly containing actor IDs, to another actor ι'. The active actor first sends an *INC* message each actor (other than the sender and the receiver) in ιs, and then sends $APP(\iota s)$ to ι'. If the sender includes itself in a message, it increments its own reference count.

ADDREF,DELREF Add and delete references to actors in its local heap, representing heap changes during program execution.

GC Garbage collect locally, compacting its external set. Actors that are removed from the external set (i.e. ιs) are sent *DEC*.

BLOCK When an actor finishes responding to a message and has no pending messages, it sends BLK to the cycle detector with a snapshot of the its topology and sets its blocked flag to $true$.

In example 1, step 2 applies rule BLOCK, rewriting the configuration to:

$$\begin{array}{rcl} cfg_2 & = & (Q_2, CD_1, \{a_1', a_2, a_3\})\, where \\ Q_2 & = & [\kappa \mapsto (BLK(\iota_1, 1, \{\iota_2\}))] \\ a_1' & = & (\iota_1, 1, \{\iota_2\}, \{\iota_2\}, true) \end{array}$$

It then applies rule RECVBLK (defined below), rewriting the configuration to:

$$\begin{array}{rcl} cfg_3 & = & (Q_3, CD_2, \{a_1', a_2, a_3\})\, where \\ Q_3 & = & \varepsilon \\ CD_2 & = & ([\iota_1 \mapsto (1, \{\iota_2\})], \varepsilon, 0) \end{array}$$

Step 3 then applies rules SEND, DELREF and GC, rewriting to:

$$\begin{array}{rcl} cfg_4 & = & (Q_4, CD_2, \{a_1', a_2', a_3\})\, where \\ Q_4 & = & [\iota_1 \mapsto (INC), \iota_3 \mapsto (APP(\iota_1), DEC)] \\ a_2' & = & (\iota_2, 2, \{\iota_1\}, \{\iota_1\}, false) \end{array}$$

Actor Message Receipt As shown in figure 5, an actor can receive messages regardless of whether or not it is blocked. An actor can receive four types of message:

RECVAPP When an actor receives an application message *APP*, each actor contained in the message (i.e. ιs) other than the receiver (i.e. $\iota s \setminus \iota$) that is already present in the receiving actor's external set (i.e. $(\iota s \setminus \iota) \cap \xi$) is sent *DEC*. Those not present in the external set are added to it. A blocked actor that receives *APP* unblocks.

RECVINC When an actor receives *INC,* it increments its reference count by one. A blocked actor that receives *INC* unblocks.

RECVDEC When an actor receives *DEC*, it decrements its reference count by one. A blocked actor that receives *DEC* unblocks.

RECVCNF When an actor receives *CNF*, it echoes the token in the message back to the cycle detector in an *ACK* message. A blocked actor that receives *CNF* does not unblock.

In example 1, step 4 applies rules RECVAPP, RECVDEC, DELREF and GC, rewriting to:

$$\begin{array}{rcl} cfg_5 & = & (Q_5, CD_2, \{a_1', a_2', a_3'\})\, where \\ Q_5 & = & [\iota_1 \mapsto (INC), \iota_2 \mapsto (DEC)] \\ a_3' & = & (\iota_3, 0, \{\iota_1\}, \{\iota_1\}, false) \end{array}$$

Step 5 then applies rules RECVDEC and BLOCK, rewriting to:

$$\begin{array}{rcl} cfg_6 & = & (Q_6, CD_2\{a_1', a_2'', a_3'\})\, where \\ Q_6 & = & [\iota_1 \mapsto (INC), \kappa \mapsto (BLK(\iota_2, 1, \{\iota_1\}))] \\ a_2' & = & (\iota_2, 1, \{\iota_1\}, \{\iota_1\}, true) \end{array}$$

It then applies rule RECVBLK (defined below), rewriting to:

$$\begin{array}{rcl} cfg_7 & = & (Q_7, CD_3, \{a_1', a_2'', a_3'\})\, where \\ Q_7 & = & [\iota_1 \mapsto (INC)] \\ CD_3 & = & ([\iota_1 \mapsto (1, \{\iota_2\}), \iota_2 \mapsto (1, \{\iota_1\})], \varepsilon, 0) \end{array}$$

Cycle Detector Local Execution We now consider the actions of the cycle detector. As shown in figure 6, the cycle detector can:

DETECT An isolated cycle of blocked actors is detected and mapped to a unique token. The actors in the newly detected perceived cycle are initially unconfirmed (mapped to $false$), and are therefore sent a *CNF* request.

COLLECT A dead cycle of confirmed actors is garbage collected. They are removed from the set of actors.

In example 1, step 6 applies part of rule DETECT, rewriting to:

$$\begin{array}{rcl} cfg_8 & = & (Q_7, CD_3, \{a_1', a_2'', a_3'\})\, where \\ CD_3 & = & ([\iota_1 \mapsto (1, \{\iota_2\}), \iota_2 \mapsto (1, \{\iota_1\})], \\ & & [0 \mapsto [\iota_1 \mapsto false, \iota_2 \mapsto false]], 1) \end{array}$$

And in example 2, step 7 applies the remainder of rule DETECT, rewriting to:

$$\begin{array}{rcl} cfg_9 & = & (Q_8, CD_3, \{a_1', a_2'', a_3'\})\, where \\ Q_8 & = & [\iota_1 \mapsto (INC, CNF(0)), \iota_2 \mapsto (CNF(0))] \end{array}$$

Cycle Detector Message Receipt As shown in figure 7, the cycle detector can receive three types of message:

RECVBLK The cycle detector maps the actor (ι) to the topology snapshot (ρ, ξ) in the message.

RECVUNB The cycle detector removes the actor (ι) from the map of perceived topology and removes all perceived cycles that contain the newly unblocked actor.

RECVACK If the perceived cycle identified by the token in the message still exists, the acknowledging actor is confirmed (mapped to $true$) in that perceived cycle.

$$\frac{\iota' \notin as \qquad a' = (\iota', 1, \emptyset, \emptyset, false)}{Q, CD, (\iota, \rho, \xi, h, false) \cup as \rightarrow Q, CD, \{(\iota, \rho, \xi \cup \iota', h, false), a'\} \cup as} \text{ (CREATE)}$$

$$\frac{\begin{array}{c}\iota' \in \xi \qquad \iota s \subseteq (\xi \cup \iota) \qquad \rho' = \begin{cases}\rho + 1 & if\ \iota \in \iota s \\ \rho & if\ \iota \notin \iota s\end{cases} \\ Q' = Push(Q, \iota s \setminus \{\iota, \iota'\}, INC) \qquad Q'' = Push(Q', \iota', APP(\iota s))\end{array}}{Q, CD, (\iota, \rho, \xi, h, false) \cup as \rightarrow Q'', CD, (\iota, \rho', \xi, h, false) \cup as} \text{ (SEND)}$$

$$\frac{\iota' \in \xi \cup \iota}{Q, CD, (\iota, \rho, \xi, h, false) \cup as \rightarrow Q, CD, (\iota, \rho, \xi, h \cup \iota', false) \cup as} \text{ (ADDREF)}$$

$$\frac{\iota' \in h}{Q, CD, (\iota, \rho, \xi, h, false) \cup as \rightarrow Q, CD, (\iota, \rho, \xi, h \setminus \iota', false) \cup as} \text{ (DELREF)}$$

$$\frac{\iota s \subseteq \{\iota' | \iota' \in \xi \wedge \iota' \notin h\} \qquad Q' = Push(Q, \iota s, DEC)}{Q, CD, (\iota, \rho, \xi, h, false) \cup as \rightarrow Q', CD, (\iota, \rho, \xi \setminus \iota s, h, false) \cup as} \text{ (GC)}$$

$$\frac{Q(\iota) = () \qquad Q' = Push(Q, \kappa, BLK(\iota, \rho, \xi))}{Q, CD, (\iota, \rho, \xi, h, false) \cup as \rightarrow Q', CD, (\iota, \rho, \xi, h, true) \cup as} \text{ (BLOCK)}$$

Figure 4: Operational semantics of actor local execution

$$\frac{\begin{array}{c}Q', APP(\iota s) = Pop(Q, \iota) \\ Q'' = Push(Q', (\iota s \setminus \iota) \cap \xi, DEC) \qquad Q''' = Unblock(Q'', \iota, \beta)\end{array}}{Q, CD, (\iota, \rho, \xi, h, \beta) \cup as \rightarrow Q''', CD, (\iota, \rho, \xi \cup (\iota s \setminus \iota), h', false) \cup as} \text{ (RECVAPP)}$$

$$\frac{Q', INC = Pop(Q, \iota) \qquad Q'' = Unblock(Q', \iota, \beta)}{Q, CD, (\iota, \rho, \xi, h, \beta) \cup as \rightarrow Q'', CD, (\iota, \rho + 1, \xi, h, false) \cup as} \text{ (RECVINC)}$$

$$\frac{Q', DEC = Pop(Q, \iota) \qquad Q'' = Unblock(Q', \iota, \beta)}{Q, CD, (\iota, \rho, \xi, h, \beta) \cup as \rightarrow Q'', CD, (\iota, \rho - 1, \xi, h, false) \cup as} \text{ (RECVDEC)}$$

$$\frac{\begin{array}{c}Q', CNF(\tau) = Pop(Q, \iota) \\ Q'' = Push(Q', \kappa, ACK(\iota, \tau))\end{array}}{Q, CD, (\iota, \rho, \xi, h, \beta) \cup as \rightarrow Q'', CD, (\iota, \rho, \xi, h, \beta) \cup as} \text{ (RECVCNF)}$$

Figure 5: Operational semantics of actor message receipt

$$\frac{\begin{array}{c}Closed(\{\iota_1..\iota_n\}, PT) \\ PC' = PC[\tau \mapsto [\iota_1 \mapsto false, ..\iota_n \mapsto false]] \\ Q' = Push(Q, \{\iota_1..\iota_n\}, CNF(\tau))\end{array}}{Q, (PT, PC, \tau), as \rightarrow Q', (PT, PC', \tau + 1), as} \text{ (DETECT)}$$

$$\frac{\begin{array}{c}\iota s = \{\iota_1..\iota_n\} = dom(PC(\tau')) \qquad Q_1 = Q \\ \forall i \in 1..n. PC(\tau')(\iota_i) \wedge Q_{i+1} = Push(Q_i, PT(\iota_i) \downarrow_2 \setminus \iota s, DEC)\end{array}}{Q, (PT, PC, \tau), as \rightarrow Q_{i+1}, (PT \setminus \iota s, PC \setminus \tau', \tau), as \setminus \iota s} \text{ (COLLECT)}$$

Figure 6: Operational semantics of cycle detector local execution

$$\frac{Q', BLK(\iota, \rho, \xi) = Pop(Q, \kappa)}{Q, (PT, PC, \tau), as \rightarrow Q', (PT[\iota \mapsto (\rho, \xi)], PC, \tau), as} \text{ (RECVBLK)}$$

$$\frac{Q', UNB(\iota) = Pop(Q, \kappa) \qquad PC' = PC \setminus \{\tau' | \iota \in PC(\tau')\}}{Q, (PT, PC, \tau), as \rightarrow Q', (PT \setminus \iota, PC', \tau), as} \text{ (RECVUNB)}$$

$$\frac{\begin{array}{c} Q', ACK(\iota, \tau') = Pop(Q, \kappa) \\ PC' = \begin{cases} PC[\tau' \mapsto PC(\tau')[\iota \mapsto true]] & if\ \tau' \in PC \\ PC & if\ \tau' \notin PC \end{cases} \end{array}}{Q, (PT, PC, \tau), as \rightarrow Q', (PT, PC', \tau), as} \text{ (RECVACK)}$$

Figure 7: Operational semantics of cycle detector message receipt

In example 2, step 8 applies step RECVINC, rewriting to:

$$\begin{array}{rcl} cfg_{10} & = & (Q_9, CD_3, \{a_1'', a_2'', a_3'\})\, where \\ Q_9 & = & [\iota_1 \mapsto (CNF(0)), \iota_2 \mapsto (CNF(0)), \\ & & \kappa \mapsto (UNB(\iota_1))] \\ a_1'' & = & (\iota_1, 2, \{\iota_2\}, \{\iota_2\}, false) \end{array}$$

Step 9 applies step RECVCNF, rewriting to:

$$\begin{array}{rcl} cfg_{11} & = & (Q_{10}, CD_3, \{a_1'', a_2'', a_3'\})\, where \\ Q_{10} & = & [\iota_2 \mapsto (CNF(0)), \\ & & \kappa \mapsto (UNB(\iota_1), ACK(\iota_1, 0))] \end{array}$$

Step 10 applies step RECVUNB, rewriting to:

$$\begin{array}{rcl} cfg_{12} & = & (Q_{11}, CD_4, \{a_1'', a_2'', a_3'\})\, where \\ Q_{11} & = & [\iota_2 \mapsto (CNF(0)), \kappa \mapsto (ACK(\iota_1, 0))] \\ CD_4 & = & ([\iota_2 \mapsto (1, \{\iota_1\})], \varepsilon, 1) \end{array}$$

Completeness If a cycle of blocked actors exists, each actor will have sent *BLK* to the cycle detector. The cycle detector will eventually execute RECVBLK for each blocked actor, and will eventually execute DETECT and begin a confirmation process that will result in executing COLLECT. This process is non-deterministic, but it is theoretically possible to detect a cycle as soon as it appears. If all actors are blocked, the system will find all cycles.

The program terminates when it is not possible to apply any rule. This occurs when no actors are executing (preventing any actor local execution rules from being applied), the queue is empty (preventing any actor or cycle detector message receipt rules from being applied), and no cycles are detected (preventing any cycle detector local execution rules from being applied).

Robustness As presented, *MAC* is sound and does not have exceptional conditions. However, the protocol is robust even if failure is introduced. If the cycle detector fails, cycles of dead actors will not be collected, but no live actor will be collected.

If an actor fails, the result depends on whether or not the cycle detector's view of the failed actor's topology is in agreement with the failed actor's view of its own topology. If it is, the failed actor can be considered blocked, and the system will function normally. If the cycle detector's view of the failed actor's topology is not in sync, then there is no way to determine what other actors the failed actor referenced. As a result, actors the failed actor held a reference to will not receive *DEC* messages for those references and will not be collected. However, it remains the case that the cycle detector will continue to collect other dead cycles, and no live actor will be collected.

Moreover, failure of actors or the cycle detector does not jeopardise termination of the overall system. Namely, collection of all actors is not required in order to reach a quiescent state where no rules can be applied. This allows the program to terminate even when some dead actors have not been collected. As a result, failure results in uncollected dead actors but does not impact soundness or robustness.

Failure of individual messages, where a message is sent but not received while future messages from the same sender are successful, impacts the system differently depending on the message type. A failed *DEC* results in an actor with an excess reference count that will not be collected. A failed *CNF* or *ACK* message that pertains to a dead cycle results in the failure to collect that dead cycle, but if the message pertains to a live cycle, there is no impact on the system. A failed *BLK* message results in an actor never being collected if the actor is blocked from that point on, but has no impact on the system if the actor ever unblocks. A failed *APP* message will result in excess reference counts for actors in the message, with the result that those actors will not be collected.

The two messages that can impact soundness on failure are *INC* and *UNB*. A failed *INC* message results in an actor that has a reference count that is too low. As a result, the cycle detector may find perceived cycles that are smaller than the true cycle. If the actors in the perceived cycle are all blocked, the cycle may be collected while an unblocked actor retains a reference to a collected actor. A failed *UNB* message for an actor in a perceived cycle can cause the cycle to be incorrectly collected if all other actors in the cycle are blocked. The sender of the failed *UNB* message

will now respond with an *ACK* without having unblocked, and the cycle detector will incorrectly perceive it as having confirmed.

However, the actor-model requires guaranteed message delivery [2]. Failure of an individual message that cannot be corrected with buffering, retries, or other techniques, can thus be treated as failure of the sending actor. If a failed message results in all future messages from the sender also failing, no form of failure impacts either soundness or robustness.

6. Proof of Soundness

Outline To prove soundness, we will show that when every actor in a perceived cycle has confirmed, the perceived cycle is a true cycle. To do so, we will show in theorem 1 that if the cycle detector's view of the topology of actors in the perceived cycle is the true topology, the perceived cycle is a true cycle. Then we will show in theorem 2 that when a single actor confirms, the cycle detector's view of its topology, the actor's view of its topology, and the actor's true topology agreed at the time when the perceived cycle was detected. Finally, we will show in theorem 3 that when every actor has confirmed, the perceived cycle is a true cycle. We will present each with an informal proof here. Formal proofs are presented in the appendix.

As we already said in the introduction, the development of the soundness proof helped us better understand the algorithm itself and the central role of the relation between the different views of the topology. Moreover, our initial intuition about the reason the algorithm was correct was slightly wrong. Namely, instead of the property outlined in theorem 2 above, we thought that only after *all* actors had confirmed would we know that the cycle detector's view coincided with the true topology. The property from theorem 2 is stronger, and easier to prove. We believe that the concepts we developed for the proof of *MAC* will be useful to prove other protocols as well.

Detailed Arguments In order to express these theorems, we define $Topo$ as the actor's view of its topology, $TrueTopo$ as the true topology based on inspecting the heaps and queues of all actors, and $TrulyClosed$ as a property of a set of actors which holds when these actors form a closed cycle in the true topology. $TrueTopo$ allows only *CNF* messages in ι's queue because all other messages cause an actor to unblock when they are received. Because a $TrulyClosed$ cycle encompasses all references to all actors in the cycle, it is not possible for actors in a $TrulyClosed$ cycle to receive messages in the future.

Definition 1 (Topology, true topology, and true cycles)**.** Given $cfg = (Q, \_, as)$, we define:

- $Heap(\iota) \triangleq h \Leftrightarrow (\iota, \_, \_, h, \_) \in as$
- $Topo(\iota, cfg) \triangleq (\rho, \xi, \beta) \Leftrightarrow (\iota, \rho, \xi, \_, \beta) \in as$
- $$TrueTopo(\iota, cfg) \triangleq \begin{pmatrix} |\{\iota' | \iota' \in as, \iota \in Heap(\iota')\}| + \\ |\{(\iota', k) | Q(\iota')[k] = APP(\iota s), \\ \iota \in \iota s, \iota' \in as \setminus \iota\}|, \\ Heap(\iota) \setminus \iota, \\ Q(\iota) = CNF(\_)* \end{pmatrix}$$
- $$TrulyClosed(\iota s, cfg) \Leftrightarrow \\ \forall \iota \in \iota s : \forall \iota'.\iota \in Heap(\iota') \rightarrow \iota' \in \iota s \wedge Q(\iota) = \varepsilon$$

For example, after step 3 of example 1, $Topo(\iota_1, cfg) = (1, \{\iota_2\}, true)$, but $TrueTopo(\iota_1, cfg) = (2, \{\iota_2\}, false)$. This is because $Q(\iota_1) = INC$, which both indicates an additional reference (in this case, from ι_3) and that ι_1 has pending messages other than *CNF*, and so will unblock.

We require three things from a well-formed configuration. First, it maintains the reference count invariant that an actor's true reference count is equal to the actor's view of its own reference count, adjusted for *INC* and *DEC* messages in the actor's queue. Second, an actor identifier appears only once in the set of actors. Third, an actor in a perceived cycle ($PC(\tau)$) is also in the cycle detector's view of blocked actor topology (*PT*).

Definition 2 (A well-formed configuration)**.** We say that a configuration $cfg = (Q, (PT, PC, \_), as)$ is well-formed, formally $WF(cfg)$, if:

1. $$TrueTopo(\iota, cfg) \downarrow_1 = \begin{pmatrix} Topo(\iota, cfg) \downarrow_1 + \\ |\{k \mid Q(\iota)[k] = INC\}| - \\ |\{k \mid Q(\iota)[k] = DEC\}| \end{pmatrix}$$
2. $\forall \iota.|\{a \mid a \in as,\ a = (\iota, \_, \_, \_, \_)\}| \leq 1$
3. $\forall \tau \in PC.PC(\tau) \subseteq PT$

We now define a history of configurations. The history of configurations is ghost state that we use to denote the times at which various events took place. A history maps time 0 to a configuration that contains a single actor.

Definition 3 (History)**.** We define H, a history of configurations.

- $H \in History = Time \rightarrow Configuration$
- $H(0) = (\emptyset, (\emptyset, \emptyset, 0), \{(\iota, 0, \emptyset, \emptyset, false)\})$
- $H(t) \rightarrow H(t+1)$

Definition 4. We expect every configuration to be well-formed implicitly. The initial configuration is well-formed, and from lemma 1 in the appendix we know that execution preserves well-formedness.

With these definitions, we can present our first theorem. We establish that, for a given perceived cycle, if the cycle detector's view of the topology of the actors in the cycle is the same as their true topology, the perceived cycle is a true cycle.

Theorem 1 (A PC is truly closed if the CD's view of the topology is the true topology)**.** *Given a configuration* $cfg = (\_, (PC, PT, \_), \_)$*:*

$$\forall \iota \in PC(\tau).(PT(\iota), true) = TrueTopo(\iota, cfg) \Rightarrow TrulyClosed(dom(PC(\tau), cfg))$$

Proof. When the cycle detector detects a perceived cycle ($PC(\tau)$) based on the cycle detector's view of the blocked actor topology (*PT*), that cycle is closed ($Closed(dom(PC(\tau)), PT)$). If the cycle detector's view of a blocked actor's topology ($PT(\iota)$) is the same as the actors true topology ($TrueTopo(\iota)$), then we can substitute the true topology of the actor for the cycle detector's view of the actor's topology in the definition of $Closed$. If we do this for all actors in a perceived cycle, we arrive at the definition of $TrulyClosed$ for the perceived cycle. □

In order to be able to describe at which time certain events took place, we now define the elements of a configuration, topology, and events in the configuration history. The queue at time t in history H is referred to as Q^t_H, and the same notation is used for PT^t_H, PC^t_H, τ^t_H, and as^t_H. An actor ι's view of its topology at time t in history H is referred to as $Topo^t_H(\iota)$, and the same notation is used for $TrueTopo^t_H$, $Closed^t_H$ and $TrulyClosed^t_H$. Similarly, $Post^t_H(\iota, msg)$ denotes the time at which message msg was posted to actor ι, $Consume^t_H(\iota, msg)$ denotes the time at which message msg was consumed by actor ι, and $NewPC^t_H(\tau)$ denotes the time at which the perceived cycle identified by τ was detected.

Definition 5 (Configuration, topology and events in the history)**.** Given H and t, if $H(t) = (Q, (PT, PC, \tau), as)$ we define:

- The elements of a configuration at time t: $Q^t_H \triangleq Q$, $PT^t_H \triangleq PT$, $PC^t_H \triangleq PC$, $\tau^t_H \triangleq \tau$, $as^t_H \triangleq as$
- Actor and cycle topology at time t:
  - $Topo^t_H(\iota) \triangleq Topo(\iota, H(t))$
  - $TrueTopo^t_H(\iota) \triangleq TrueTopo(\iota, H(t))$
  - $Closed^t_H(\iota s) \triangleq Closed(\iota s, PT^t_H)$
  - $TrulyClosed^t_H(\iota s) \triangleq TrulyClosed(\iota s, H(t))$
- Predicates denoting the times at which events occurred.
  - $Post^t_H(\iota, msg) \Leftrightarrow Q^{t-1}_H(\iota) = q \wedge Q^t_H(\iota) = q.msg$
  - $Consume^t_H(\iota, msg) \Leftrightarrow Q^{t-1}_H(\iota) = msg.q \wedge Q^t_H(\iota) = q$
  - $NewPC^t_H(\tau) \Leftrightarrow \tau \notin PC^{t-1}_H \wedge \tau \in PC^t_H \wedge Closed^t_H(dom(PC^t_H(\tau)))$

For example, in example 1, $Post^t_H(\iota_1, INC)$ indicates the time when ι_2 sends *INC* to ι_1, and $Consume^t_H(\iota_3, APP(\iota s))$ indicates the time when ι_3 receives $APP(\iota s)$ from ι_2. Similarly, $NewPC^t_H(0)$ indicates the time when the cycle detector detects the cycle $\{\iota_1, \iota_2\}$.

In the appendix we present a series of short lemmas that establish the underlying behaviour of the system, revolving around FIFO message queues and the resultant ordering of events. Using these definitions and lemmas, we can present the remaining two theorems. First, we establish that confirmation from a single actor indicates that the confirmed actor's view of the its topology was the same as the cycle detector's view of that actor's topology when the perceived cycle was detected.

Theorem 2 (A confirmed actor implies the CD's view of its topology, the actor's view of its topology, and its true topology agreed when the PC was detected)**.** $PC^t_H(\tau)(\iota) \Rightarrow \exists t_3 \leq t. NewPC^{t_3}_H(\tau) \wedge (PT^{t_3}_H(\iota), true) = Topo^{t_3}_H(\iota) = TrueTopo^{t_3}_H(\iota)$

Proof. Because actor ι is confirmed ($PC^t_H(\tau)(\iota)$), we know the cycle detector consumed an acknowledgement message from ι ($\exists t_5 \leq t. Consume^{t_5}_H(\kappa, ACK(\iota, \tau))$), and therefore ι consumed a confirmation message from the cycle detector ($\exists t_4 \leq t_5. Consume^{t_4}_H(\iota, CNF(\tau))$) and sent an acknowledgement message in response ($Post^{t_4}_H(\kappa, ACK(\iota, \tau))$). This in turn means the cycle detector sent the confirmation message ($\exists t_3 \leq t_4 \wedge Post^{t_3}_H(\iota, CNF(\tau))$), which means the perceived cycle containing ι was detected ($NewPC^{t_3}_H(\tau)$). This implies ι is in the cycle detector's view of the blocked topology ($\iota \in PT^{t_3}_H$), which means the cycle detector consumed a block message from ι ($\exists t_2 \leq t_3. Consume^{t_2}_H(\kappa, BLK(\iota, \rho, \xi))$) and has not consumed an unblock message from ι ($\forall t'. t_2 \leq t' \leq t. \neg Consume^{t'}_H(\kappa, UNB(\iota))$). This in turn indicates that ι sent a block message ($\exists t_1 \leq t_2. Post^{t_1}_H(\kappa, BLK(\iota, \rho, \xi))$) and did not send an unblock message before sending an acknowledgement message ($\forall t'. t_1 \leq t' \leq t_4. \neg Post^{t'}_H(\kappa, UNB(\iota))$), which means ι's view of its own topology did not change during that time ($\forall t'. t_1 \leq t' \leq t_4. Topo^{t_1}_H(\iota) = Topo^{t'}_H(\iota)$). Since the perceived cycle was detected on the basis of the topology in the block message and was detected before the acknowledgement message was sent, we know ι's view of its topology at the time the perceived cycle was detected was the same as the cycle detector's view of ι's topology ($\forall t'. t_2 \leq t' \leq t_4. (PT^{t'}_H(\iota), true) = Topo^{t'}_H(\iota)$).

If ι's view of its reference count was not the same as its true reference count ($Topo^{t_3}_H(\iota) \downarrow_1 \neq TrueTopo^{t_3}_H(\iota) \downarrow_1$), then, given the reference count invariant, ι's queue must contain either *INC* or *DEC* ($Q^{t_3}_H(\iota) \ni (INC \vee DEC)$). If that were true, we know that ι would consume *INC* or *DEC* before $CNF(\tau)$, which would mean ι would send an unblock message before sending an acknowledgement message, which we know to be untrue because ι is confirmed. Therefore, ι's view of its reference count is its true reference count.

If ι's view of its external set was not the same as its true external set ($Topo^{t_3}_H(\iota) \downarrow_2 \neq TrueTopo^{t_3}_H(\iota) \downarrow_2$), then ι must have taken a step that rewrites its external set. By case analysis, ι must unblock to change its external set,

which would mean ι would send an unblock message before sending an acknowledgement message, which we know to be untrue because ι is confirmed. Therefore, ι's view of its external set is its true external set.

If ι's view of its blocked state was not the same as its true blocked state ($Topo_H^{t_3}(\iota) \downarrow_3 \neq TrueTopo_H^{t_3}(\iota) \downarrow_3$), the ι's queue must contain a message other than *CNF* ($Q_H^{t_3}(\iota) \neq CNF(\_)*$). If this were true, we know that ι would consume a message other than *CNF* before $CNF(\tau)$, which would mean ι would send an unblock message before sending an acknowledgement message, which we know to be untrue because ι is confirmed. Therefore, ι's view of its blocked state is its true blocked state.

Therefore, for each actor ι in the perceived cycle, the cycle detector's view of ι's topology, ι's view of its topology and ι's true topology agree when the perceived cycle was detected ($\forall \iota \in PC_H^{t_3}(\tau).(PT_H^{t'}(\iota), true) = Topo_H^{t_3}(\iota) = TrueTopo_H^{t_3}(\iota)$). □

We now establish that confirmation from all actors indicates, for every actor in the cycle, that the actor's view of its topology is the same as the true topology of the actor. In combination with theorem 2, this tells us that the cycle detector's view of the topology of the actors in the cycle is also the same as the true topology. In combination with theorem 1, this tells us that the perceived cycle is a true cycle and can be safely collected.

Theorem 3 (A fully confirmed cycle is a true cycle)**.** $\forall \iota \in PC_H^t(\tau).PC_H^t(\tau)(\iota) \Rightarrow TrulyClosed_H^t(dom(PC_H^t(\tau)))$

Proof. Since every actor in the perceived cycle is confirmed, we know from theorem 2 that when the perceived cycle was detected ($\exists t_3 \leq t.NewPC_H^{t_3}(\tau)$) every for actor ι in the perceived cycle, the cycle detector's view of the topology of ι, ι's view of its topology, and ι's true topology agreed ($\forall \iota \in PC_H^{t_3}(\tau).(PT_H^{t_3}(\iota), true) = Topo_H^{t_3}(\iota) = TrueTopo_H^{t_3}(\iota)$). We know from theorem 1 that $PC(\tau)$ was therefore truly closed at time t_3 ($TrulyClosed_H^{t_3}(dom(PC_H^{t_3}(\tau)))$), and a truly closed cycle can be collected. □

7. Implementation

In this section we discuss the practical implications of implementing *MAC*. We first report on its current deployment in an actor runtime deployed at a large financial institution, where the good performance (linear speedup with the number of cores) indicates that *MAC* imposed a negligible performance overhead. We then discuss implementation considerations affecting the overhead of *MAC*, and some implementation details. Finally, we compare with four benchmarks proposed for Erlang, Scala, Akka, and libcppa, and find that *MAC's* performance is competitive.

We have implemented message-based actor collection as part of a runtime library written in C. The library also includes asynchronous messaging, work-stealing scheduling, memory allocation from per-actor heaps, precise passive object garbage collection, and an extension of the system described in this paper for collecting passive objects that have been passed or shared across actor heaps.

Actors are currently written in C on top of the runtime. We do not have a full actor-model programming language yet. Programming using our library does not offer full type safety and is thus error prone. However, the library and programs written in C are sufficient to investigate the performance of our approach.

Current deployment The library is currently in use at a large financial institution as the concurrency core of a high-throughput, low-latency application. The application processes thousands of requests per second under peak usage and each request potentially creates dozens of new actors. These actors are short-lived, surviving only for the duration of the request, which results in regular garbage collection of actors. The application is relatively long-lived, running for a week at a time. Performance has been nearly linear with core count for this application, resulting in approximately a 31.5x speed-up on a 32-core machine. These numbers indicate that hundreds of thousands of actors is a realistic number for some classes of production software, that short-lived actors are a useful approach to concurrency, and that our implementation of actor collection does not impede performance.

Implementation considerations Causal messaging is guaranteed on a single host (multi- or many-core) using FIFO ordered message queues with guaranteed atomic delivery, implemented as lock-free wait-free multiple-producer single-consumer queues. Sending a message requires approximately 10 nanoseconds. Because the only message that requires an acknowledgement is the confirmation message from the cycle detector, no message round trips are required during normal execution. During collection, a round trip is only required if the actor is indeed ready to be collected, in which case the *CNF* message requiring acknowledgment is the only message on the actor's queue, resulting in the fastest possible response. Otherwise, an *UNB* message associated with an earlier rule execution will be received by the cycle detector, short circuiting the round trip. As a result, the overhead of the Conf-Ack protocol is very low.

The overhead of *BLK* and *UNB* messages is similarly low, introducing only an additional 10 nanoseconds of latency when an actor unblocks and another 10 nanoseconds when an actor blocks. This cost is only paid when an actor has no other pending work. An optimisation we have made in the implementation allows an actor to notify the cycle detector of a reference count change when it processes *INC* or *DEC* without unblocking, accomplishing with one message what would otherwise take two.

The formalisation and proof lead directly to an additional optimisation in the implementation: as specified in the GC rule in the operational semantics, the subset of actors checked for cycles and the timing of those checks is non-

deterministic. This allows the cycle detector to defer detection, performing orders of magnitude less work. This can be seen in Table 5, where cycle detection attempts are significantly less common than *BLK* messages.

Weighted reference counting We have implemented a limited form of weighted reference counting to eliminate some of the *INC* and *DEC* messages required when sending and receiving *APP*. This requires the external set to become an *external map.* The external map associates actors with numbers, such that each actor can keep a *reference weight* for any other actor.

When an actor ι_0 receives an *APP* message containing an actor ι_1, the reference weight of ι_1 in the external map of ι_0 is incremented. As a result, when an actor receives an *APP* message, it no longer needs to send *DEC* messages to actors contained in the message that are already in the receiver's external map.

On the other hand, when ι_0 sends a reference to ι_1 to some actor ι_2 and the reference weight of ι_1 in the external map of ι_0 is greater than one, the reference weight is decremented and no *INC* message is sent. In that case, we say that a reference to ι_1 is *split* across ι_0 and ι_2. However, when ι_0 sends a reference to ι_1 to some actor ι_2 and the reference weight of ι_1 in the external map of ι_0 is one, then an *INC* message has to be sent to ι_1, because the single reference cannot be split. In this case, an *INC* message with an arbitrary additional reference weight is sent from ι_0 to ι_1. This additional reference weight is added to the reference weight of ι_1 in the external map of ι_0. When ι_1 receives the *INC* message, the additional reference weight is also added to the reference count of ι_1. In this way, the number of *INC* messages required is significantly reduced.

When the heap of ι_0 no longer contains a reference to ι_1, ι_0 sends a *DEC* message to ι_1 that includes the reference weight of ι_1 in the external map of ι_0. When ι_1 receives the *DEC* message, the reference weight is subtracted from the reference count of ι_1.

Finally, when an actor blocks, it includes its reference count and its external map in the *BLK* message sent to the cycle detector. The definitions of $Closed$ and $TrueTopo$ are changed to account for the reference weight in the external map, and cycle detection proceeds as before.

Benchmarks and preliminary comparisons with Erlang, Scala, Akka, and libccpa Currently, *MAC* can collect hundreds of thousands to millions of actors (in various cyclic topologies) per second on current x64 hardware (2.5 to 3.5 ghz, 4 to 32 cores). In comparison, the pseudo-root approach used in SALSA 1.0 collects thousands of actors per second on a dual-processor Solaris machine [16].

We have evaluated programs written against our runtime library with both manual actor termination and *MAC*. We present a summary of preliminary experimental results in Tables 1 to 4. A break down of cycle detector attempts

Language	Time (s)	Throughput (msg/s)
Erlang OTP	~9	~333,333
Erlang	~7	~428,571
Scala (react)	~9	~333,333
libcppa	~5.5	~545,454
MAC, disable CD	0.24	12,500,000
MAC, normal CD	0.24	12,500,000
MAC, force CD	0.24	12,500,000

Table 1: Message handling: 3 million messages, 2 cores

Language	Time (s)	Throughput (actors/s)
Erlang	~10	~52,429
Scala (react)	~10	~52,429
Scala (Akka)	~18	~29,127
libcppa	~18	~29,127
MAC, disable CD	2.9	180,788
MAC, normal CD	7.5	69,905
MAC, force CD	9.5	55,188

Table 2: Actor creation: 2^{19} actors, 4 cores

Language	Time (s)	Throughput (msgs/s)
Erlang	~16	~1,250,000
Scala (react)	~45	~444,444
Scala (Akka)	~30	~666,666
libcppa	~15	~1,333,333
MAC, disable CD	5.2	3,846,153
MAC, normal CD	5.2	3,846,153
MAC, force CD	5.2	3,846,153

Table 3: Mailbox performance: 20 million messages, 4 cores

Language	Time (s)	Throughput (msgs/s)
Erlang	~125	~400,000
Scala (react)	~120	~416,666
Scala (Akka)	~60	~833,333
libcppa	~80	~625,000
MAC, disable CD	45.7	1,094,091
MAC, normal CD	77.3	646,830
MAC, force CD	78.4	637,755

Table 4: Mixed scenario: 50 million messages plus factorisation, 4 cores

and successes and message counts for each test scenario is presented in Table 5.

These tests are taken from a series of benchmarks made available for a collection of existing actor languages and libraries [29] with publicly available source code [30]. The benchmarks are designed to stress specific aspects of actor-model languages such as message performance and actor creation performance. To match the hardware and methodology of the existing benchmark results, we executed the first test on a 2 core 2.67 GHz i7 and the other three tests on a 4 core 2.27 GHz Xeon, with the average of five runs being presented. In Tables 1 to 4, we present the results previously reported in the existing benchmarks [29] and add the results we obtained for *MAC*.

For *MAC*, we present results in three configurations: with cycle detection disabled, with cycle detection enabled (detecting termination via quiescence), and forcing cycle detection (detecting termination via all actors in the system having been collected). We include test results for forced cycle detection in order to evaluate worst-case behaviour.

The message handling benchmark spawns a counter actor and a worker actor. The worker actor sends three million messages to the counter actor asking it to increment its counter, followed by a single get-and-reset message retrieving the counter. This tests raw message performance, but not actor creation or concurrency.

The actor creation benchmark spawns 2^{19} actors, arranged in a doubly-linked tree, forming a single cyclic graph. This is a good stress test for the cycle detector.

The mailbox performance benchmark spawns a single receiver and twenty senders, each of which send one million messages to the receiver. This tests concurrent message performance, as the senders are scheduled simultaneously across cores.

The mixed scenario spawns twenty rings of fifty actors each. Each ring sends 500,000 messages around the ring while a worker actor performs expensive factorisation. This is repeated five times, resulting in fifty million messages and 100 factorisation runs. This tests combining expensive calculation with a heavy message load.

Discussion of preliminary implementation results We note that all MAC versions perform the same in message handling and in mailbox performance. This may be so because in both these tests the actors involved are constantly sending or receiving messages, and as a result they only block at the end of the test.

The preliminary results are highly encouraging. We chose to reuse existing benchmarks rather than design new ones in order to both provide a direct comparison between our work and existing actor-model languages and libraries and to avoid inadvertently tailoring benchmarks to our own approach. On the other hand, an aspect that might be underrepresented in these benchmarks is passing references to actors in messages. We plan to investigate more in future work.

8. Conclusion and Further Work

We have presented Message-based Actor Collection (*MAC*), a system for fully concurrent garbage collection of actors, including an operational semantics in section 5 and a proof of soundness in section 6. Specifically, we have addressed our goals:

1. *Soundness:* our three theorems show that after completing the conf-ack protocol, a perceived cycle is a true cycle and can be safely collected.
2. *Completeness:* our operational semantics show that all dead actors are eventually collected, allowing the system to terminate when all actors have been collected.
3. *Concurrency:* our technique is entirely message-based, and does not require clocks, timestamps, shared memory, locking, read/write barriers, or any particular threading or scheduling system.

The soundness result has been proven for *MAC* in the many-core setting. To transfer to the distributed setting, we will need to address the following issues: 1) causal messaging across distributed nodes, 2) potential message loss, 3) potential node failure.

We plan to fully address these issues in further work, but we argue here that *MAC* can be adapted to this setting. Namely, for (1) nodes can be structured as a tree, which can efficiently provide communication paths that are always causal. For (2), we can use a guaranteed delivery network protocol paired with a buffer of sent messages that can be used to replay messages when a connection is dropped and reestablished. This buffer can be reset by lazy, asynchronous acknowledgement of receipt of a batch of messages by a node. For (3), Erlang-like actor monitors can be coupled with periodic reference renewal to allow notification of failure and eventual consistency of actor reference counts.

We plan to investigate the application of *MAC* in a distributed environment. Specifically, we are interested in effi-

Test	CD Attempts	Cycles Collected	APP Msgs	BLK Msgs	UNBLK Msgs	Other MAC Msgs
Message handling	1	1	3,000,000	3	1	11
Actor creation	9	1	1,048,575	707,492	183,205	524,288
Mailbox performance	1	1	20,000,000	21	0	66
Mixed scenario	49	20	50,005,124	50,004,818	49,999,898	10,372

Table 5: Cycle detector and message statistics for tests for MAC, force CD

cient causal messaging across distributed nodes, improving distributed cycle detection by using multiple local cycle detectors, and robustness in the presence of message, actor, or node failure.

We also plan to extend this work to collect passive objects that are shared across actor heaps as well as separate collection of each heap. When an actor finishes handling an application message (whether or not additional messages are pending on the queue), that actor has no stack. By extending *MAC* so that it uses a message-based system for passive object collection, the point at which an actor finishes handling an application message establishes a safe-point without instrumentation. This will allow local and distributed passive object garbage collection without read or write barriers.

Acknowledgements

We are grateful to the anonymous referees for their pertinent and helpful feedback. We would like to thank the SLURP reading group at Imperial College London for their extensive feedback, as well as the anonymous referees at ECOOP and ESOP for their constructive comments on earlier versions of this paper. We would also like to thank Harry Richardson and Andrew McNeil for their helpful discussions of implementation considerations.

References

[1] G. Agha and C. Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In LNCS 1985.

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

[3] J. Armstrong. A history of Erlang. In HOPL III, 2007.

[4] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. In TCS 2008.

[5] T. Van Cutsem. Ambient References: Object Designation in Mobile Ad hoc Networks. PhD thesis, Vrije Universiteit Brussel, 2008.

[6] C. Varela, G. Agha, Wei-Jen Wang, et al. The SALSA Programming Language 2.0.0alpha Release Tutorial. Rensselaer Polytechnic Institute, 2009.

[7] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java (A Million Actors, Safe Zero-Copy Communication). In ECOOP 2008.

[8] D. Kafura, D. Washabaugh, J. Nelson. Garbage collection of actors. In OOPSLA 1990.

[9] A. Vardhan, G. Agha. Using passive object garbage collection algorithms for garbage collection of active objects. In ISMM 2002

[10] Wei-Jen Wang, et al. Actor Garbage Collection Using Vertex-Preserving Actor-to-Object Graph Transformations. In GPC 2010.

[11] S. Tasharofi, P. Dinges, R. Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In ECOOP 2013.

[12] `http://actor-applications.cs.illinois.edu/`

[13] `http://www.gotw.ca/publications/concurrency-ddj.htm`

[14] `http://osl.cs.uiuc.edu/af/`

[15] C. Varela and G. Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. In OOPSLA 2001.

[16] Wei-Jen Wang and C. Varela. Distributed Garbage Collection for Mobile Actor Systems: The Pseudo Root Approach. In GPC 2006.

[17] Wei-Jen Wang. Distributed Garbage Collection for Large-Scale Mobile Actor Systems. PhD thesis, Rensselaer Polytechnic Institute, 2006.

[18] Wei-Jen Wang. Conservative snapshot-based actor garbage collection for distributed mobile actor systems. In Telecommunication Systems, 2011.

[19] H. Baker. Minimizing reference count updating with deferred and anchored pointers for functional data structures. In ACM SIGPLAN, Sept. 1994.

[20] Y. Levanoni, E. Petrank. An On-the-Fly Reference-Counting Garbage Collector for Java. In OOPSLA 2001.

[21] M. Shapiro, D. Plainfossé. A Survey of Distributed Garbage Collection Techniques. In IWMM 1995.

[22] F. Dehne and R. Lins. Distributed Cyclic Reference Counting. In LNCS 1994.

[23] D. Bacon and V.T. Rajan. Concurrent Cycle Collection in Reference Counted Systems. In ECOOP 2001.

[24] L. Moreau and J. Duprat. A Construction of Distributed Reference Counting. In Acta Informatica 2001.

[25] L. Moreau, P. Dickman, and R. Jones. Birrell's Distributed Reference Listing Revisited. In TOPLAS 2005.

[26] R. Jones and R. Lins. Cyclic Weighted Reference Counting Without Delay. In PARLE 1993.

[27] R. Lins. Lazy Cyclic Reference Counting. In JUCS 2003.

[28] A. Formiga and R. Lins. A New Architecture for Concurrent Lazy Cyclic Reference Counting on Multi-Processor Systems. In JUCS 2007.

[29] `http://libcppa.blogspot.co.uk/search/label/benchmark`

[30] `https://github.com/Neverlord/cppa-benchmarks`

A. Lemmas

Lemma 1. *A well-formed configuration progresses to a well-formed configuration.*

- $cfg_0 \rightarrow cfg' \Rightarrow WF(cfg')$
- $WF(cfg) \wedge cfg \rightarrow cfg' \Rightarrow WF(cfg')$

Proof. By case analysis on the rewrite steps that can be applied to a configuration. Applying any step results in another well-formed configuration. □

Lemma 2. *A message in a queue implies the message was posted.*

$$Q^t_H(\iota) = q.msg.q' \Rightarrow \exists t' \leq t.Post^{t'}_H(\iota, msg)$$

Proof. By induction on t. The last step either appended msg to the queue, establishing the property, or msg was already present in the queue, in which case we apply the inductive hypothesis. □

Lemma 3. *Consuming a message implies the message was posted.*

$$Consume^t_H(\iota, msg) \Rightarrow \exists t' < t.Post^{t'}_H(\iota, msg)$$

Proof. Consuming a message at time t requires that $Q^t_H(\iota) = msg.q$, and thus by lemma 2, $\exists t' \leq t.Post^{t'}_H(\iota, msg)$. □

Lemma 4. *Messages are consumed in FIFO order.*

$$Post^t_H(\iota, msg) \wedge Post^{t+k}_H(\iota, msg') \wedge Consume^{t'}_H(\iota, msg') \Rightarrow \exists t''.t < t'' < t'.Consume^{t''}_H(\iota, msg)$$

Proof. By induction on t. Given that msg was posted before msg', and, using lemma 3, msg' was consumed after msg' was posted, we begin with the configuration at time t, when msg was posted. The last step consumed msg'. The previous step either consumed msg or msg was not on the queue. If msg was not on the queue, we apply the inductive hypothesis. □

Lemma 5. *Actors that are blocked and have not posted unblock have not changed their view of their topology.*

$$Topo^t_H(\iota) \downarrow_3 \wedge \forall t' \geq t.\neg Post^{t'}_H(\kappa, UNB(\iota)) \Rightarrow Topo^t_H(\iota) = Topo^{t'}_H(\iota)$$

Proof. By case analysis. We begin with a blocked actor at time t. No step can be made in which UNB is sent, and the actor's view of its topology does not change in any step in which UNB is not sent. □

Lemma 6. *Actors that have posted block and have not posted unblock have not changed their view of their topology.*

$$Post^t_H(\kappa, BLK(\iota, \rho, \xi)) \wedge \forall t' \geq t.\neg Post^{t'}_H(\kappa, UNB(\iota)) \Rightarrow \forall t''.t \leq t'' \leq t'.Topo^t_H(\iota) = Topo^{t''}_H(\iota)$$

Proof. By case analysis. Given lemma 5, only one step (unblocked actors with empty queues) sets an actor's blocked state to true, and that step sends BLK. □

Lemma 7. *Actors in PT have blocked and have not unblocked.*

$$\iota \in PT^t_H \Rightarrow \exists t' \leq t.Consume^{t'}_H(\kappa, BLK(\iota, \rho, \xi)) \wedge \forall t''.t' \leq t'' \leq t.\neg Consume^{t''}_H(\kappa, UNB(\iota)) \wedge PT^{t''}_H(\iota) = (\rho, \xi)$$

Proof. By induction on t. The last step could have consumed BLK for the actor, establishing the property. It could not have consumed UNB, since that would remove the actor from PT. Any other step leaves PT unchanged, and we apply the inductive hypothesis. □

Lemma 8. *Actors that are sent CNF must be members of a perceived cycle that has just been detected and vice versa.*

$$Post^t_H(\iota, CNF(\tau)) \Leftrightarrow \iota \in PC^t_H(\tau) \wedge NewPC^t_H(\tau)$$

Proof. By case analysis. Only one step sends CNF, and that step detects a new perceived cycle and sends CNF only to the members of that cycle. □

Lemma 9. *Actors that send ACK must have consumed CNF and vice versa.*

$$Post^t_H(\kappa, ACK(\iota, \tau)) \Leftrightarrow Consume^t_H(\iota, CNF(\tau))$$

Proof. By case analysis. Only one step posts ACK, and that step consumes CNF for the same token. □

Lemma 10. *Confirming actors consumed CNF earlier.*

$$Consume^t_H(\kappa, ACK(\iota, \tau)) \Rightarrow \exists t' \leq t.Consume^{t'}_H(\iota, CNF(\tau))$$

Proof. If $Consume^t_H(\kappa, ACK(\iota, \tau))$ then by lemma 3 $\exists t'.Post^{t'}_H(\kappa, ACK(\iota, \tau))$, and thus by lemma 9 $\exists t'' \leq t.Consume^{t''}_H(\iota, CNF(\tau))$. □

Lemma 11. *The CD consumed ACK for every confirmed actor in a PC.*

$$PC^t_H(\tau)(\iota) \Rightarrow \exists t' \leq t.Consume^{t'}_H(\kappa, ACK(\iota, \tau))$$

Proof. By case analysis. Only one step maps ι to $true$ in $PC(\tau)$, and that step consumes $ACK(\iota, \tau)$. □

Lemma 12. *A PC is uniquely identified by its token.*
$NewPC^{t}_{H}(\tau) \wedge NewPC^{t'}_{H}(\tau) \Rightarrow t = t'$

Proof. By case analysis. Only one step creates a new PC, and that step uses a unique token. Any PC identified by a given token is the same PC, created in the same step. □

B. Theorems

Theorem 1 (A PC is truly closed if the CD's view of the topology is the true topology)**.** *Given a configuration $cfg = (\_, (PC, PT, \_), \_)$:*
$\forall \iota \in PC(\tau).(PT(\iota), true) = TrueTopo(\iota, cfg) \Rightarrow TrulyClosed(dom(PC(\tau), cfg))$

Proof. Assume that:
(A) $Closed(dom(PC(\tau)), PT)$
Expanding the definition of closed, we get:
(B) $\forall \iota \in PC(\tau) : \forall \iota'.\iota \in PT(\iota') \downarrow_2 \rightarrow \iota' \in PC(\tau) \wedge PT(\iota) \downarrow_1 = |\{\iota' \mid \iota' \in PC(\tau),\ \iota \in PT(\iota') \downarrow_2\}|$
Given $\forall \iota \in PT(\tau).(PT(\iota), true) = TrueTopo(\iota, cfg)$, we substitute $TrueTopo(\iota, cfg)$ for $PT(\iota)$ in (B), and get:
(C) $\forall \iota \in PC(\tau) : \forall \iota'.\iota \in TrueTopo(\iota', cfg) \downarrow_2 \rightarrow \iota' \in PC(\tau) \wedge TrueTopo(\iota, cfg) \downarrow_1 = |\{\iota' \mid \iota' \in PC(\tau) \wedge \iota \in TrueTopo(\iota', cfg) \downarrow_2\}|$
Given (A) and (C), we arrive at the definition of $TrulyClosed(dom(PC(\tau)), cfg)$. □

Theorem 2 (A confirmed actor implies the CD's view of its topology, the actor's view of its topology, and its true topology agreed when the PC was detected)**.** $PC^{t}_{H}(\tau)(\iota) \Rightarrow \exists t_3 \leq t.NewPC^{t_3}_{H}(\tau) \wedge (PT^{t_3}_{H}(\iota), true) = Topo^{t_3}_{H}(\iota) = TrueTopo^{t_3}_{H}(\iota)$

Proof. Given $PC^{t}_{H}(\tau)(\iota)$ and lemma 11, we get:
(A) $\exists t_5 \leq t.Consume^{t_5}_{H}(\kappa, ACK(\iota, \tau))$
From (A) and lemma 10, we get:
(B) $\exists t_4 \leq t_5.Consume^{t_4}_{H}(\iota, CNF(\tau))$
From (A) and (B) and lemma 3, we get:
(C) $\exists t_3 \leq t_4.Post^{t_3}_{H}(\iota, CNF(\tau))$
From (C) and lemma 8, we get:
(D) $\iota \in PC^{t_3}_{H}(\tau) \wedge NewPC^{t_3}_{H}(\tau)$
From (D) we get:
(E) $\iota \in PT^{t_3}_{H}$
From (E) and lemma 7 we get:
(F1) $\exists t_2 \leq t_3.Consume^{t_2}_{H}(\kappa, BLK(\iota, \rho, \xi))$
(F2) $\forall t'.t_2 \leq t' \leq t.\neg Consume^{t'}_{H}(\kappa, UNB(\iota))$
(F3) $\forall t'.t_2 \leq t' \leq t.PT^{t'}_{H}(\iota) = (\rho, \xi)$
From (F1), (F2) and lemma 3, we get:
(G1) $\exists t_1 \leq t_2.Post^{t_1}_{H}(\kappa, BLK(\iota, \rho, \xi))$
(G2) $\forall t'.t_1 \leq t' \leq t_4.\neg Post^{t'}_{H}(\kappa, UNB(\iota))$
From (G1), (G2) and lemma 6, we get:
(H) $\forall t'.t_1 \leq t' \leq t_4.Topo^{t_1}_{H}(\iota) = Topo^{t'}_{H}(\iota)$
From (F3), (G1) and (H), we get:
(I) $\forall t'.t_2 \leq t' \leq t_4.(PT^{t'}_{H}(\iota), true) = Topo^{t'}_{H}(\iota)$
If $Topo^{t_3}_{H}(\iota) \neq TrueTopo^{t_3}_{H}(\iota)$, one of the following must be true:
(J1) $Topo^{t_3}_{H}(\iota) \downarrow_1 \neq TrueTopo^{t_3}_{H}(\iota) \downarrow_1$
Given the reference count invariant, this is only possible if $Q^{t_3}_{H}(\iota) \ni INC$ or $Q^{t_3}_{H}(\iota) \ni DEC$. If either of these were true, by lemma 8 and lemma 4, INC or DEC would be consumed by ι before $CNF(\tau)$, which would mean $\exists t'.t_1 \leq t' \leq t_4.Post^{t'}_{H}(\kappa, UNB(\iota))$, which we know from to be untrue from (G2). Therefore (J1) cannot be true.
(J2) $Topo^{t_3}_{H}(\iota) \downarrow_2 \neq TrueTopo^{t_3}_{H}(\iota) \downarrow_2$
By case analysis, ι must unblock to change $Topo^{t_3}_{H}(\iota) \downarrow_2$, which would mean $\exists t'.t_1 \leq t' \leq t_{4,\iota}.Post^{t'}_{H}(\kappa, UNB(\iota))$, which we know to be untrue from (G2). Therefore (J2) cannot be true.
(J3) $Topo^{t_3}_{H}(\iota) \downarrow_3 \neq TrueTopo^{t_3}_{H}(\iota) \downarrow_3$
From theorem 2, we know $Topo^{t_3}_{H}(\iota) \downarrow_3$. If $\neg TrueTopo^{t_3}_{H}(\iota) \downarrow_3$, then $Q^{t_3}_{H}(\iota) \neq CNF(\_)*$. If this were true, by lemma 8 and lemma 4, messages other than $CNF(\_)$ would be consumed by ι before $CNF(\tau)$, which would mean $\exists t'.t_1 \leq t' \leq t_{4,\iota}.Post^{t'}_{H}(\kappa, UNB(\iota))$, which we know from to be untrue from (G2). Therefore (J3) cannot be true.
From (J1), (J2) and (J3) we get:
(K) $Topo^{t_3}_{H}(\iota) = TrueTopo^{t_3}_{H}(\iota)$
From (C), (D), (I) and (K), we establish $\exists t_3 \leq t.NewPC^{t_3}_{H}(\tau) \wedge (PT^{t_3}_{H}(\iota), true) = Topo^{t_3}_{H}(\iota) = TrueTopo^{t_3}_{H}(\iota)$. □

Theorem 3 (A fully confirmed cycle is a true cycle)**.** $\forall \iota \in PC^{t}_{H}(\tau).PC^{t}_{H}(\tau)(\iota) \Rightarrow TrulyClosed^{t}_{H}(dom(PC^{t}_{H}(\tau)))$

Proof. From theorem 2 and lemma 12, we get:
(A1) $\exists t_3.NewPC^{t_3}_{H}(\tau)$
(A2) $\forall \iota \in PC^{t_3}_{H}(\tau).(PT^{t_3}_{H}(\iota), true) = Topo^{t_3}_{H}(\iota) = TrueTopo^{t_3}_{H}(\iota)$
From (A1), (A2) and theorem 1, we establish $TrulyClosed^{t}_{H}(dom(PC^{t}_{H}(\tau)))$. □