CRGC: Fault-Recovering Actor Garbage Collection in Pekko

DAN PLYUKHIN, University of Southern Denmark, Denmark
GUL AGHA, University of Illinois Urbana-Champaign, USA
FABRIZIO MONTESI, University of Southern Denmark, Denmark

Actors are lightweight reactive processes that communicate by asynchronous message-passing. Actors address common problems like concurrency control and fault tolerance, but resource management remains challenging: in all four of the most popular actor frameworks (Pekko, Akka, Erlang, and Elixir) programmers must explicitly kill actors to free up resources. To simplify resource management, researchers have devised *actor garbage collectors (actor GCs)* that monitor the application and detect when actors are safe to kill. However, existing actor GCs are impractical for distributed systems where the network is unreliable and nodes can fail. The simplest actor GCs do not collect cyclic garbage, whereas more sophisticated actor GCs are not *fault-recovering*: dropped messages and crashed nodes can cause actors to become garbage that never gets collected.

We present Conflict-free Replicated Garbage Collection (CRGC): the first fault-recovering cyclic actor GC. In CRGC, actors and nodes record information locally and broadcast updates to the garbage collectors running on each node. CRGC does not require locks, explicit memory barriers, or any assumptions about message delivery order, except for reliable FIFO channels from actors to their local garbage collector. Moreover, CRGC is simple: we concisely present its operational semantics, which has been formalized in TLA$^+$, and prove both soundness (non-garbage actors are never killed) and completeness (all garbage actors are eventually killed, under reasonable assumptions). We also present a preliminary implementation in Apache Pekko and measure its performance using two actor benchmark suites. Our results show the performance overhead of CRGC is competitive with simpler approaches like weighted reference counting, while also being much more powerful.

CCS Concepts: • **Software and its engineering** → **Garbage collection**; • **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → **Distributed algorithms**.

Additional Key Words and Phrases: actors, actor model, fault tolerance, distributed systems, garbage collection

ACM Reference Format:
Dan Plyukhin, Gul Agha, and Fabrizio Montesi. 2025. CRGC: Fault-Recovering Actor Garbage Collection in Pekko. *Proc. ACM Program. Lang.* 9, PLDI, Article 185 (June 2025), 25 pages. https://doi.org/10.1145/3729288

1 Introduction

Distributed applications manage pools of resources like containers, memory, and connections to external APIs. In many such applications, resources are encapsulated within *actors*: lightweight reactive processes that communicate by asynchronous message-passing [1, 2]. Actors protect resources from unsafe concurrent access and guarantee that resources will be reclaimed when the actor is killed. However, deciding when to kill actors is a hard problem.

Figure 1 shows a simplified resource management problem based on Hadoop YARN [47]. In the figure, a manager actor (am) assigns work (startTask) to a worker actor (container) that encapsulates

Authors' Contact Information: Dan Plyukhin, dplyukhin@imada.sdu.dk, University of Southern Denmark, Odense, Denmark; Gul Agha, University of Illinois Urbana-Champaign, Urbana, IL, USA, agha@illinois.edu; Fabrizio Montesi, fmontesi@imada.sdu.dk, University of Southern Denmark, Odense, Denmark.

ACM 2475-1421/2025/6-ART185
https://doi.org/10.1145/3729288

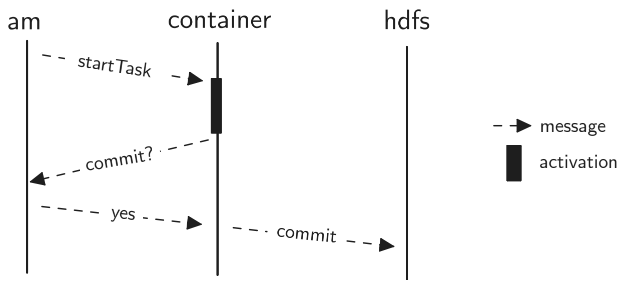

Fig. 1. A task commit protocol, based on Hadoop YARN [16].

a fixed set of CPU and memory resources. The worker completes the task and then asks am for confirmation (commit?) before writing to the distributed storage service (hdfs). But what if the worker does not receive a timely reply from its manager? There are two options:

(1) The worker could decide am's node crashed, in which case the worker's resources should be reclaimed. But if am's node hasn't really crashed, killing the worker could cause a bug.
(2) The worker could wait for am indefinitely, but then the worker's resources will never be reclaimed.

As a compromise, the worker could *passivate* [10, 29, 34] by persisting itself to disk—but moving actors between main memory and disk has a cost, and passivated actors still consume some resources. To be efficient, programmers must write code that detects when actors are safe to kill and anticipates all possible faults. Often this means reasoning about the global state of the application and communicating with remote nodes; unsurprisingly, such code is prone to distributed concurrency bugs [5, 26, 33].

Researchers have long since proposed *actor garbage collectors (actor GCs)* that automatically detect when actors are safe to kill [13, 15, 21, 23, 36–38, 40, 45, 49, 51]. But existing actor GCs are not *fault-recovering*: they will never reclaim container if am's node crashes in Figure 1. Although there are fault-recovering reference listing algorithms that could be applied to actors, these algorithms do not collect cyclic garbage (for instance, when a manager and a worker actor have references to one another) and the correctness of these algorithms is difficult to prove [32].

We present Conflict-free Replicated Garbage Collection (CRGC): the first fault-recovering cyclic actor GC. Our work addresses the following key problems:

How can the GC collect garbage resulting from crashed nodes, if a crashed node cannot be distinguished from a slow one? We observe that two of the most popular actor frameworks—Akka [27] and its open-source fork Pekko [3]—both use membership protocols to "exile" unresponsive nodes from the cluster after enough time passes [28, 41]. In these frameworks, if am's node is exiled before am can send its reply, then the reply will never be delivered. We show actor GCs can use information from membership protocols to safely recover after nodes have crashed—but one must have a precise understanding of those membership protocols to navigate the edge cases that arise in practice. For example, in both Akka and Pekko, a message from am can still be delivered after am's node is exiled if that message already reached its recipient's mailbox. Informed by real-world behavior, we develop a formal model for distributed actor systems with membership protocols, and we formally characterize which actors are safe to kill in the model. We then use the model to develop an operational semantics for CRGC.

How does the GC know which actors are garbage? The classical approach is to request a local snapshot from every actor in the application, imposing some form of synchronization to ensure snapshots are consistent with one another [12]. Such an approach is not ideal in large computing clusters, because one slow node can delay garbage collection for the rest of the cluster. In modern actor GCs, the actors themselves choose how often to send local snapshots, while the garbage collector scans the snapshots received so far and verifies if actors that "appear" to be garbage are truly garbage [8, 13, 37, 38]. A key limitation of modern actor GCs is that actors exchange certain "control" messages to maintain reference counts or reference listings, and the network must deliver these messages reliably. In CRGC, we remove the need for control messages by storing extra information in each actor's local state. Our approach allows CRGC to easily recover from dropped messages and to collect garbage without waiting for slow nodes, at the cost that acyclic garbage actors must wait to be killed by their local garbage collector instead of collecting themselves.

How does the GC recover from dropped messages and exiled nodes? In practice, messages between nodes always go through an "egress point" at the sending node to be serialized, and an "ingress point" at the receiving node to be deserialized. Following Puaut [40], egress and ingress points can be instrumented to track information about each message, allowing the runtime to infer when messages have been dropped without requiring message re-transmission. While Puaut's technique can already be used to recover from dropped messages, our key insight is that ingress points also integrate elegantly with membership protocols. Namely, after a node is exiled, the ingress points of that node's neighbors can be queried to form an "effective snapshot" of the node's actors. We prove that effective snapshots are sufficient for detecting all garbage ensuing from exiled nodes.

In summary, we make the following contributions.

(1) *A fault-aware model for reasoning about actor garbage.* Our model incorporates a high-level membership protocol, based on that of Akka and Pekko. Nodes may crash, messages can be dropped or reordered, and actors may halt and monitor one another for failure. We formalize the model's semantics with a TLA$^+$ specification, we define what it means for an actor to be garbage in the model, and we prove that garbage actors never receive messages.[1]

(2) *A fault-recovering actor GC.* We present CRGC and formalize its semantics with a TLA$^+$ specification. We also prove that CRGC is sound (non-garbage actors are never collected) and complete (all garbage actors are eventually collected) with respect to our model.

(3) *An implementation of CRGC in Apache Pekko* [3]. We present a preliminary Pekko frontend for *managed actors*: actors whose lifetimes are managed by the runtime instead of by the programmer.[2] Pekko has a large API that was not designed with actor GC in mind, so we report some open problems related to usability that may interest the broader research community. Our new frontend is agnostic about the underlying actor GC "engine", and currently supports preliminary implementations of CRGC and weighted reference counting [6, 52]. We evaluate CRGC with custom versions of the Savina [19] and ChatApp [9] actor benchmark suites, and find that CRGC has comparable performance overhead to weighted reference counting—despite being much more powerful.[3]

2 Related Work

Past approaches fit in three categories: *Acyclic GCs*, which cannot collect garbage actors that have references to one another; *Cyclic GCs*, which can collect these actors but are not fault-recovering; and *Passivation*, which is used for special kinds of actors that have infinite lifetimes.

[1]Full proofs and TLA$^+$ specifications for this paper can be found at github.com/dplyukhin/crgc-spec.
[2]The implementation can be found at github.com/dplyukhin/uigc-pekko.
[3]The benchmarks can be found at github.com/dplyukhin/uigc-bench.

Table 1. Comparison of cyclic actor GCs.

	Message order	Proof?	Fault-tolerant?	Dropped msg recovery?	Crashed node recovery?
Venkatasubramanian et al [49]	FIFO	✓	✗	✗	✗
Puaut [40]	FIFO	✗	✓	✓	✗
Kafura et al [21]	FIFO	✗	✗	✗	✗
Vardhan-Agha [45]	FIFO	✓	✓	✗	✗
Kamada et al [23]	none	✗	✗	✗	✗
Wang-Varela [51]	none	✗	✓	✗	✗
Clebsch-Drossopoulou [13]	causal	✓	✓	✗	✗
Plyukhin-Agha [38]	none	✓	✓	✗	✗
CRGC	none/FIFO	✓	✓	✓	✓

2.1 Acyclic GCs

Acyclic GCs allow actors to detect when no other actor has a reference to them. This is done either with reference counting [6, 35, 52] or reference listing [38, 51]; both approaches involve involve sending "control messages" when actors gain or lose references. Acyclic GCs are appealingly simple, but dropped control messages are difficult to recover from [7, 32, 42]. These GCs also cannot collect cyclic actor garbage, which occurs in supervision hierarchies where parent actors and their children have references to one another [4]. It is also possible to create cyclic garbage with monitoring, as we show in Section 3.

2.2 Cyclic GCs

Early cyclic GCs used one of three broad approaches. The *global snapshot* approach [22, 48] computes an approximately-consistent global snapshot of the cluster and then searches the global snapshot for actor garbage. The *tracing* approach [21, 23, 40, 51] is based on traditional tracing collectors [20] but adapted to the actor setting. The *actor-to-object* approach [15, 45, 50] adds edges to the actor reference graph, so that actor garbage coincides with traditional garbage. All three approaches are appealingly similar to traditional GCs. However, they are often complex, lacking proofs of correctness, and involving costly distributed synchronization. Only Puaut's actor GC is capable of recovering from dropped messages [40]. These approaches are problematic because they assume the collector can get a global view of the cluster; any slow actors, threads, and nodes will cause delays and must be handled explicitly.

More recent work uses a *collage-based* approach, in which actors send snapshots of their local state to the garbage collector without any distributed coordination. The resulting set of snapshots, which we call a *collage*, is not necessarily global or consistent in the sense of Chandy and Lamport [12]. Nevertheless, collage-based GCs can use message-passing protocols [13] or information in the collage itself [37, 38] to identify subsets of the collage corresponding to garbage actors. Collage-based GCs do not assume that every actor has recorded a snapshot, so these approaches can naturally collect *some* garbage even when certain nodes are unresponsive.

The first collage-based GC was *MAC*, introduced by Clebsch and Drossopoulou in the Pony language [13, 14]. In MAC, actors use weighted reference counting to collect acyclic garbage and they send local snapshots to the *cycle detector* when their mailbox becomes empty. Crucially, MAC requires causal message delivery—which is not provided in popular distributed actor frameworks—and a centralized cycle detector. Plyukhin and Agha showed that both of these assumptions could be removed in *DRL*, a collage-based GC based on reference listing [37]. They also demonstrated that distributed garbage collection can be parallelized across a team of node-local garbage collectors [38]. However, neither MAC nor DRL is fault-recovering: actors referenced by crashed nodes can never be collected. The key issue is that crashed nodes hold important data, like the number of messages or references sent by the node, but that data can never be recovered.

CRGC can be seen as an optimized version of DRL where actors do not send control messages.Our key technical contribution is showing how to make actor GC fault-recovering. We do this by extending the model to take into account *cluster membership protocols* (Section 3) and adding basic instrumentation to each node's message ingress points (Section 4.1). With this extra machinery, healthy nodes can synthesize "effective snapshots" for crashed nodes, permitting more actors to be garbage collected than in DRL or MAC. We also introduce the notions of *shadow graphs* and *undo logs* for representing collages efficiently in memory (Section 4.2).

Table 1 compares cyclic actor GCs. *Message order* indicates if application messages between actors must be delivered in a certain order; CRGC does not require FIFO delivery in the theoretical model (Section 4) but we exploit it in our implementation (Section 5). *Fault-tolerant* indicates that the GC does not become stuck when nodes crash. *Fault-recovery* is a stronger property: it is the ability to collect actors that have become garbage because of a fault.

2.3 Virtual Actors and Passivation

A *virtual actor* [10, 29] is an actor with an infinite lifetime; virtual actors are automatically restarted if they throw an exception, and re-instantiated elsewhere if their original node crashes. In contrast, ordinary *actors* in Pekko and Akka (like the processes in Erlang and Elixir) have an explicit lifecycle and can be monitored for failure. CRGC is designed for the latter semantics, which offers lower overhead and greater control, at the cost of convenience.

Virtual actors can be passivated—i.e., removed from memory and persisted to disk—to conserve resources [29, 34]. However, choosing a passivation policy requires guesswork and risks thrashing (passivating too often and hurting performance) or underutilization (passivating too little and incurring unnecessary resource costs). Passivation is also unsuitable for dynamic workloads that spawn large numbers of actors, as in the Savina benchmark suite [19]. In the future, programmers could combine actor GC and passivation to reduce the burden on the programmer—using GC on short timescales and using passivation on long timescales. Actor GC is somewhat orthogonal to passivation, because some actors only receive messages infrequently and never become garbage.

3 Model

To present CRGC, we will need a model for distributed actor systems. Our model incorporates the kinds of faults seen in some earlier work [40, 46]—namely, crashed nodes and dropped or reordered messages—but it also includes mechanisms that have not been considered before. In particular, actors in our model may halt during execution due to uncaught exceptions, and actors may monitor one another for failure. Our model also includes a high-level *cluster membership protocol*, based on the protocols Pekko and Akka use to remove misbehaving nodes from the cluster; we use this protocol in Section 4 to make CRGC fault-recovering. We show how these failure scenarios and fault-handling mechanisms can lead to subtle, unintuitive behavior that arises in practice. Nevertheless, we can prove the cluster membership protocol has strong properties (Section 3.2) and actor garbage can be precisely characterized in terms of *quiescent* actors (Section 3.3).

We have formalized the semantics of our model with a TLA$^+$ specification. Similarly to the labeled transition systems used in prior work [13, 37, 38], TLA$^+$ specifications can be formatted concisely on a page and formally reasoned about, but they can also be executed by a computer up to bounded depth. More details can be found in the Supplement.

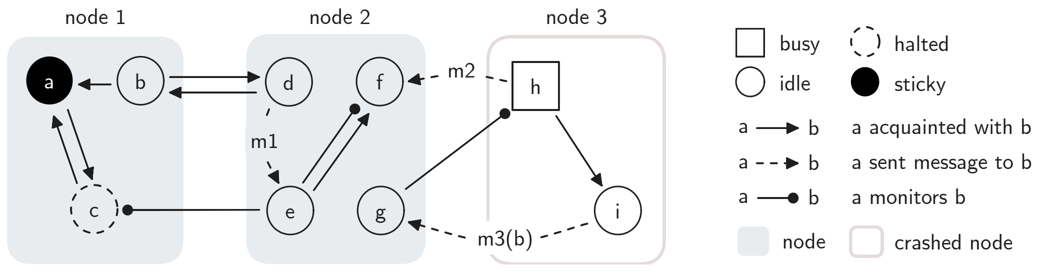

Fig. 2. A configuration in our model.

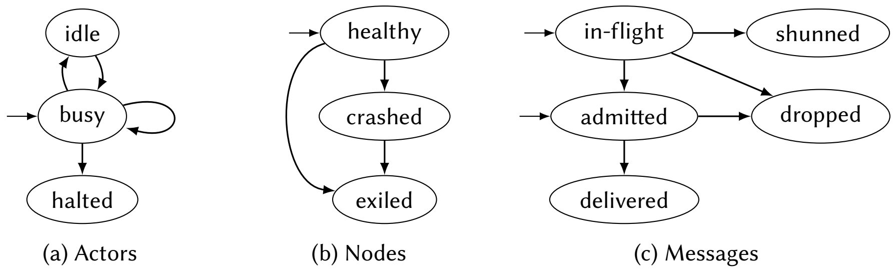

Fig. 3. State machines for actors, messages, and nodes.

3.1 Actors

The global state of the system at some instant is called a *configuration*. The initial configuration consists of a fixed set of nodes $N_1, \ldots, N_n$ and an actor a_0 located on an arbitrary node.

Actors may be *busy*, *idle*, or *halted*. A busy actor can perform computation, like sending messages and spawning new actors. An idle actor cannot do anything until some event causes it to become busy. A halted actor cannot do anything and cannot become busy. We explain these states in more detail below; a state machine for actors is depicted in Figure 3a.

Busy actors. A busy actor a can do any of the following:

- *Spawn actors.* Actor a creates a new actor b, thereby giving a a reference to b. Actor b may be spawned onto a different node than a, c.f. Section 3.2.
- *Send messages.* If a has a reference to b, then a can send a message to b. Messages may contain references to other actors (e.g., a can send b a reference to some third actor c).
- *Deactivate references.* If a has a reference to b, then the reference can be deactivated by removing the reference from a's local state.
- *Halt or become idle.* Actor a can become halted (e.g., due to an uncaught exception) or become idle (e.g., because it finished handling a message and is ready to receive a new one).
- *Begin/end monitoring an actor.* If a has a reference to b, then a can ask to be notified by the runtime system when b halts. Actor a will also be notified if b is exiled, c.f. Section 3.2.
- *Register/unregister as "sticky".* Sticky actors model the fact that some actors can spontaneously become busy (e.g., by setting a timeout or receiving messages from outside the system).

In Figure 2, actor e monitors c and f, and g monitors h. Actors e, f, and g have undelivered messages. The message from i to g contains a reference to b.

Idle actors. An idle actor can do the following:

- *Receive a message.* When a receives a message m, the actor becomes busy and adds all the references in m to a's local state.
- *Receive a failure notification.* If a is monitoring b and b halted, then a can become busy. Messages and failure notifications can arrive out of order, as shown in Figure 4.
- *Spontaneously wake up (if sticky).* Idle sticky actors can spontaneously become busy.

In Figure 2, actor e can receive a failure notification because it monitors a halted actor, and g will be able to receive a failure notification once h's node is exiled, c.f. Section 3.2. If g receives the message $m_3(b)$, it will become busy and be able to send a message to b. Actor a is idle, but it can become busy because it is sticky.

Messages. Message delivery is asynchronous. Every actor has a mailbox that stores undelivered messages. If a sends a message to some b on the same node, then a puts the message directly in b's mailbox. If b is on a remote node, then the message is initially *in flight* from one node to the other, before being *admitted* to b's node and added to its mailbox. This distinction between in-flight and admitted messages is crucial to how failures are handled.

Some actor GCs assume messages are delivered in a particular order. For instance, Pony's actor GC requires causal message delivery [13]. Pekko, Akka, Erlang, and Elixir only guarantee that messages are delivered in FIFO order [30]. Our actor GC will not impose *any* constrains on message order, except for FIFO delivery between actors and their local garbage collector (Section 5).

Monitoring. The semantics of monitoring is similar to message passing, but with some crucial differences. If a begins monitoring an actor and that actor fails—i.e., the actor halts or its node is exiled—then a is guaranteed to eventually receive a failure notification. Even if the monitored actor failed before monitoring began, a will still be notified. Hence monitoring cannot easily be modeled in terms of message passing alone.

3.2 Nodes

Every actor has a fixed location on some node. Nodes are either *healthy* or *crashed*; actors on healthy nodes execute normally, but actors on crashed nodes cannot take actions and their local state is unrecoverable. In Pekko, Akka, Erlang, and Elixir, actors can detect crashes with monitoring: if actor a is monitoring a remote actor b and b's node crashes, then a will eventually be notified. However, actor frameworks differ in how they handle transient failures. In Erlang and Elixir, a will be notified when b's node disconnects [43]; it is up to the application developer to decide if the node "really" crashed, or merely underwent a temporary network partition. In Pekko and Akka, nodes use a *cluster membership protocol* [28, 41] to force all misbehaving nodes out of the cluster—even nodes that have not really crashed. The latter semantics is crucial for fault-recovering actor GC, because otherwise we can never be certain that actors on a supposedly-crashed node will not return. We model cluster membership at a high level, abstracting implementation details and not assuming any kind of global synchronization.

3.2.1 *Cluster Membership.*

In Pekko and Akka, healthy nodes use failure detectors [11, 18, 28] to identify nodes that may have crashed. We assume all crashed nodes are eventually detected, but some healthy nodes may be misdiagnosed as crashed.

If N_1 suspects N_2 crashed, then N_1 can make the irrevocable decision to *shun* N_2. As a result, messages from N_2 will no longer be admitted into N_1 (although messages that have already been admitted can still be delivered). After N_1 shuns N_2, every other node must eventually either shun N_2 or else be shunned by N_1. When some group of nodes $\mathcal{G}_1$ has shunned all the remaining nodes $\mathcal{G}_2$, we say $\mathcal{G}_1$ has *exiled* $\mathcal{G}_2$: from the perspective of $\mathcal{G}_1$, the exiled nodes in $\mathcal{G}_2$ are indistinguishable from crashed nodes. The state machine for nodes is depicted in Figure 3b.

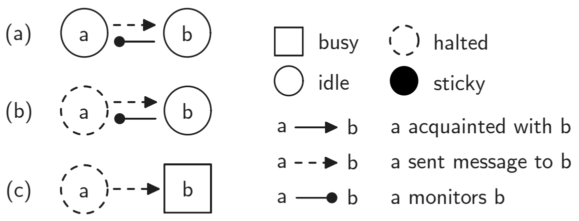

Fig. 4. A sequence of configurations showing how failure signals and messages can arrive out of order. In configuration (a), actor *b* monitors *a* and also has a deliverable message from *a*. In configuration (b), actor *a* halts. In configuration (c), *b* is notified that *a* halted before *b* receives the message.

Notice that nodes can crash while a node is being exiled from the system. For example, one node can shun another and immediately crash. Let us say a node is *faulty* if it crashes at some point in time. Assuming every execution has a non-faulty node, our model has the following properties:[4]

Lemma 3.1. *Every faulty node is eventually exiled.*

Lemma 3.2. *If N_1 shunned N_2 then eventually N_2 will be exiled or N_1 will be exiled.*

The cluster membership protocol can also result in "split-brain scenarios", where two groups of healthy nodes exile one another. For example, nodes 1 and 2 can exile each other in Figure 2. From the perspective of actor GC, these scenarios are irrelevant: the garbage collector in node 1 can proceed as if node 2 crashed, and vice versa. Hence, without loss of generality, we assume healthy nodes that have been exiled do not take actions.

3.2.2 Dropped Messages. We assume all undelivered messages can be dropped. In practice, messages in flight from one node to another can be dropped because a TCP connection resets. Messages sent between actors on the same node can also be dropped if the recipient has a bounded mailbox [30]. We assume a mechanism that eventually detects how many messages have been dropped and what references those messages contained. Pekko and Akka do not provide this mechanism natively, but it can be implemented using Puaut's technique [40]. The complete state machine for messages is depicted in Figure 3c.

3.3 Actor Garbage

In modern actor frameworks, any busy actor can potentially have observable effects like writing to a file. We therefore follow recent work [13, 38, 46] and define *garbage actors* to be actors that never become busy. But just by looking at a configuration at some instant in time, how do we predict which actors will never become busy? The problem is not as simple as in traditional GCs, where garbage objects are those that cannot be reached from the root set [20]. Fortunately, we can use properties of actor systems to characterize which actors will surely never become busy; these actors are said to be *quiescent*. But first, we need to define some properties about configurations:

Definition 3.3. A message is *deliverable* if either (1) it has been admitted, or (2) it is in-flight and the recipient's node has not shunned the sender's node. We say actor *a* has a deliverable message if *a* is the recipient of a deliverable message.

An actor *a* is *potentially acquainted* with actor *b* if *a* has a reference to *b* or *a* has a deliverable message containing a reference to *b*. In this situation, *b* is a *potential acquaintance* of *a* and *a* is a *potential inverse acquaintance* of *b*.

[4]All proofs are relegated to the Supplemenal Text, which can be found at github.com/dplyukhin/crgc-spec.

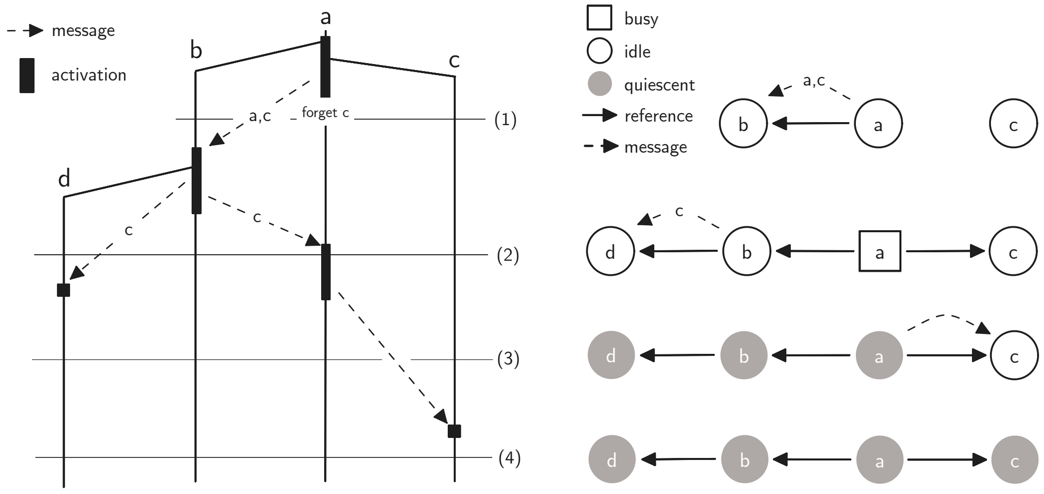

Fig. 5. A sample execution in our model. On the left is a time diagram with four consistent cuts. On the right we see the configuration in each of those four cuts.

An actor a is *blocked* if a is idle and has no deliverable messages. Otherwise, a is *unblocked*.
An actor a has *failed* if it is halted or its node is exiled. Otherwise, a is *healthy*.

In Figure 2, messages m_2 and m_3 are deliverable unless node 2 shuns node 3 before the messages are admitted. If m_3 is deliverable, then actor g is unblocked and potentially acquainted with b; this implies d is not garbage, because g could send a message to b and b could send a message to d. But if m_3 is dropped or node 2 shuns node 3 before m_3 is admitted, then the message is no longer deliverable, and b and d are garbage because they can never become busy.

Monitoring also has surprising effects on actor garbage. In Figure 2, since e monitors f and has a reference to it, the two actors have a cyclic dependency on one another: actor f is not garbage unless e is garbage (otherwise, e could send f a message), and e is not garbage unless f is garbage (otherwise, f could become busy and then halt, causing e to receive a failure notification).

In general, a healthy actor a can become busy if and only if one of the following can occur: a receives a message; a is sticky and spontaneously wakes up; a monitors an actor that halted; or a monitors an actor on a different node that was exiled. If an actor satisfies none of these properties in a given configuration, we say it is quiescent:

Definition 3.4. An actor b is *quiescent* if b has failed or all of the following hold:

1. b is blocked;
2. b is not sticky;
3. If any a is potentially acquainted with b, then a is quiescent; and
4. If any a is monitored by b, then a is quiescent, not failed, and located on the same node.

Figure 5 shows an execution where actors become quiescent. In (1), b is unblocked and potentially acquainted with a and c. In (2), a and d are unblocked and a has references to b and c. In (3), a, b, and d are quiescent because they are idle and so are their potential inverse acquaintances—but c is unblocked. In (4), all the actors are quiescent.

As expected, quiescence is a tight characterization for actor garbage in our model:

Theorem 3.5. *An actor a is garbage if and only if a is quiescent.*

Table 2. Information recorded locally by an actor a in CRGC.

Field	**Description**
a.status	Either “idle”, “busy”, or “halted”.
a.isSticky	TRUE if a is sticky; FALSE otherwise.
a.monitored	The set of actors currently monitored by a.
a.received	The number of messages that a has received.
a.sent(b)	The number of messages that a has sent to b.
a.created(b, c)	The number of references to c that a has sent to b.
a.deactivated(b)	The number of references to b that a has deactivated.

Thus, despite the numerous complications in our model—dropped messages, faulty actors, monitor signals arriving out-of-order with messages—we arrived at a definition of actor garbage that straightforwardly generalizes the definition from simpler models [38]. Crucially, our definition accounts for garbage resulting from faults: consider an idle, non-sticky actor a that does not monitor any other actors. If all potential inverse acquaintances of a are garbage or located on exiled nodes, then a is garbage.

4 Fault-Recovering Actor Garbage Collection

Now that we have seen how garbage actors coincide with quiescent actors, we can present CRGC: an actor GC that detects quiescent actors. We begin in Section 4.1 with an abstract mathematical view, in which actors record GC-related information in their local state and take snapshots of that state at arbitrary times. The garbage collector is simply a function whose input is a collage (a set of local snapshots) and whose output is a set of actors that “appear” quiescent in the collage. We prove garbage collection is sound (actors that appear quiescent are quiescent) and complete (quiescent actors eventually appear quiescent).

However, the abstract view above does not readily admit an efficient implementation. In Section 4.2, we show how collages can be represented efficiently as *shadow graphs* with a distinguished set of *pseudo-root* nodes; actors that appear quiescent in the collage coincide with nodes in the graph that are not reachable from a pseudo-root. This result establishes a pleasing correspondence between CRGC and traditional tracing garbage collectors.

We have formalized a model of CRGC and shadow graphs as TLA$^+$ specifications. All proofs in this section are written with respect to those specifications, the semantics of which are defined by set theory and the temporal logic of actions [25].

4.1 Garbage Collection with Collages

In this version of CRGC, actors record information in their local state and take snapshots of that information at arbitrary times. Nodes are equipped with *ingress points* that record information about messages admitted from the network and also take snapshots of this recorded state. A collection of local snapshots from distinct actors and ingress points is called a *collage*. We will show that, by recording the right kind of information in each snapshot, garbage collectors can detect quiescent actors without the need for locks or inter-actor synchronization protocols. This approach is fundamentally different from actor GCs that use consistent global snapshots [12] because collages are neither consistent nor global.

4.1.1 Actor Local State. Table 2 lists the information an actor needs to record in its local state. The status, isSticky, and monitored fields all reflect properties of an actor at a given time t. The remaining fields grow monotonically, summarizing all the actions a performed since it was first

Table 3. Information recorded by an ingress point $I_{N,N'}$ in CRGC.

Field	**Description**
$I_{N,N'}$.shunned	TRUE if N has been shunned by N'; FALSE otherwise.
$I_{N,N'}$.admittedMsgs(a)	The number of messages that N has sent to a and that have been admitted to N'.
$I_{N,N'}$.admittedRefs(a, b)	The number of references to b that N has sent to a and that have been admitted to N'.

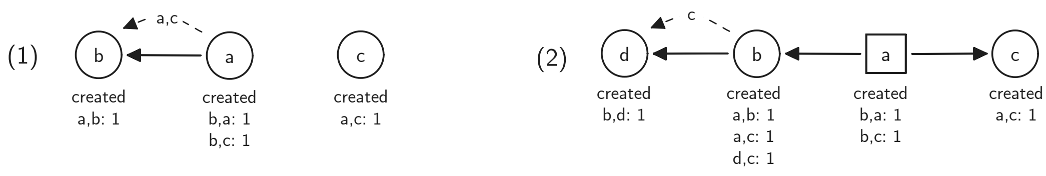

Fig. 6. Actors from Figure 5, annotated with CRGC creation counts.

spawned. In particular, the sent, created, and deactivated fields are growable associative arrays, mapping actors (or pairs of actors) to integers. In practice, these arrays impose a memory leak; we solve this problem in Sections 4.2 and 5. For now, we assume each actor maintains all the information in Table 2, and records this information whenever it takes a local snapshot. (Note that actors do not need locks to read and write local information atomically, because actors are single-threaded.)

Actors update their local state whenever they perform the events listed in Section 3.1. When actor a is first spawned by some actor b, the field a.created(b, a) is initialized to 1. (That is, a grants its parent a reference upon being spawned.) When a receives a message it sets a.status to "busy" and increments a.received. When a sends a message to b containing a reference to c, it increments a.sent(b) and a.created(b, c). When a no longer needs a reference to b, it increments a.deactivated(b).

Actors recover from dropped messages by updating their state as if the message was delivered normally. When an idle actor a learns that a message containing references to b and c was dropped, it increments a.received, a.deactivated(b), a.deactivated(c) and its status remains "idle". Thus a's state is the same as if the message was delivered and its references were immediately deactivated.

Of all these fields, a.created is the most unusual. It ensures there is always a "contact trace" from a to any actor that potentially has a reference to a [37]. This can be seen by annotating Figure 5 with creation counts, as shown in Figure 6, and focusing on actor c. Initially, c was spawned by a so c.created$(a, c) > 0$. Then a sends b a reference to c, and b sends d a reference to c, so a.created$(b, c) > 0$ and b.created$(d, c) > 0$. This contact tracing chain—from c's local state to a, from a's local state to b, and from b's local state to d—will allow the GC to check whether it has enough snapshots from enough actors to tell that c is quiescent.

4.1.2 *Ingress Points.*

An *ingress point* $I_{N,N'}$ is responsible for admitting messages from node N onto N'. Every pair of nodes has an ingress point located at the recipient, and we assume ingress points record the information in Table 3. If N' admits a message from N destined for a containing references to b and c, then N' increments $I_{N,N'}$.admittedMsgs(a), $I_{N,N'}$.admittedRefs(a, b), and $I_{N,N'}$.admittedRefs(a, c). The node also increments these fields if it learns such a message

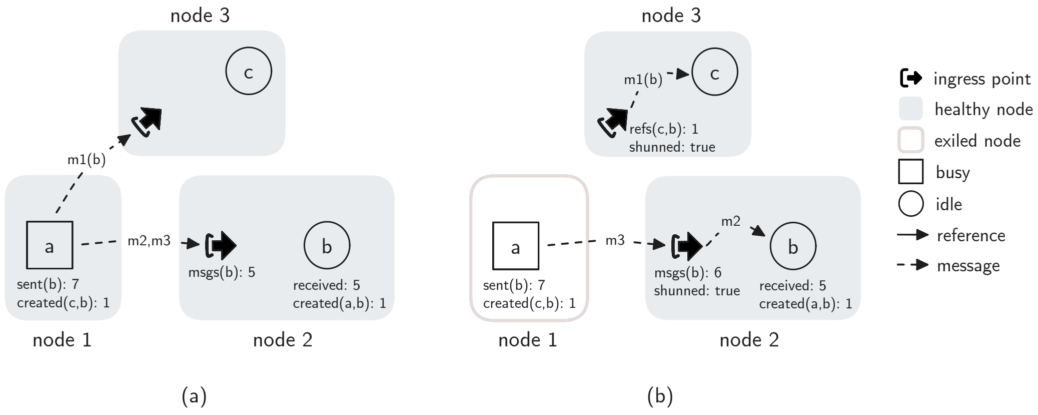

Fig. 7. An example with ingress points. For legibility, some information in actor and ingress states is not shown.

was dropped. In addition, if N' shuns N, then $I_{N,N'}$.shunned is eventually set to True; hence $I_{N,N'}$.shunned = True implies the other fields of $I_{N,N'}$ will never change again.

Figure 7 shows how ingress points make up for the fact that exiled actors cannot take snapshots. In Figure 7 (a), actor a's local state shows it sent seven messages to b and gave c a reference to b. In Figure 7 (b), actor a is exiled and can no longer take a snapshot—but the ingress points on nodes 2 and 3 can provide an "effective snapshot" of all a's effects.

Let S be a collage, i.e. a partial function mapping from actors and ingress points to local snapshots. Then we define what it means for an actor or node to appear exiled:

Definition 4.1. Node N *appears exiled* if the set of all nodes can be partitioned into two nontrivial groups $\mathcal{G}_1, \mathcal{G}_2$, where $N \in \mathcal{G}_1$ and $S(I_{N_1,N_2})$.shunned for each $N_1 \in \mathcal{G}_1, N_2 \in \mathcal{G}_2$.

Actor a *appears exiled* if a is located on a node that appears exiled.

Actor a has an *effective snapshot* in S if a has a snapshot in S or a appears exiled in S.

4.1.3 Apparent Quiescence. Given a collage S, we show how to check if an actor appears blocked (by tallying message send and receive counts) and how to find which actors appear acquainted with it (by tallying created and deactivated counts). We will use these notions to define which actors appear quiescent, as a straightforward analogue of Definition 3.4. Since S is not consistent, we cannot assume *a priori* that actors appearing have some property ever actually had that property [12, 13]. Nevertheless, we will prove that CRGC provides enough information that appearing quiescent is *sufficient* to imply quiescence (Theorem 4.5), and moreover every quiescent actor will eventually appear quiescent under reasonable assumptions (Theorem 4.6).

Counting Messages and References. In the absence of failures, accounting for messages and references is simple. We can define "the number of message sent to b according to S" as the sum $\sum_{a \in \text{dom}(S)} S(a).\text{sent}(b)$. If this total is equal to $S(b)$.received, we can say b appears blocked. Likewise, we can say b appears to have references to c if the number of references created ($\sum_{a \in \text{dom}(S)} S(a).\text{created}(b, c)$) exceeds the number of references b deactivated ($S(b).\text{deactivated}(c)$). However, we cannot guarantee that exiled actors will ever have a snapshot in S. Instead, as shown earlier, we must make up for the missing actor snapshots using snapshots from ingress points. This leads us to the revised definitions below.

Let $\mathcal{G}_1$ be the set of nodes that do not appear exiled, let $\mathcal{G}_2$ is the set of nodes that do appear exiled, and let N_b denote the location of actor b. For any pair of actors b, c, we define:

$$\mathsf{sent}(b) = \sum_{N_1 \in \mathcal{G}_1} \sum_{a \in S(N_1)} S(a).\mathsf{sent}(b) + \sum_{N_2 \in \mathcal{G}_2} S(I_{N_2,N_b}).\mathsf{admittedMsgs}(b)$$

$$\mathsf{created}(b,c) = \sum_{N_1 \in \mathcal{G}_1} \sum_{a \in S(N_1)} S(a).\mathsf{created}(b,c) + \sum_{N_2 \in \mathcal{G}_2} S(I_{N_2,N_b}).\mathsf{admittedRefs}(b,c)$$

$$\mathsf{received}(b) = \begin{cases} S(b).\mathsf{received} & \text{if } b \in \mathrm{dom}(S) \\ 0 & \text{otherwise} \end{cases}$$

$$\mathsf{deactivated}(b,c) = \begin{cases} S(b).\mathsf{deactivated}(c) & \text{if } b \in \mathrm{dom}(S) \\ 0 & \text{otherwise,} \end{cases}$$

where we write $a \in N$ if a is located on N, and $S(N)$ denotes the set of actors on N with snapshots in S. For example, the number of messages a sent to b according to S is computed by (1) adding up $S(a).\mathsf{sent}(b)$ for each actor a that doesn't appear exiled; and (2) adding up $S(I_{N,N_b}).\mathsf{admittedMsgs}(b)$ for each apparently exiled node N.

We use the counts above to define the collage's view of the cluster:

Definition 4.2. Actor a *appears acquainted* with b if $\mathsf{created}(a,b) > \mathsf{deactivated}(a,b)$.
Actor a *appears to monitor* b if $b \in S(a).\mathsf{monitored}$.
Actor a *appears idle*, *busy*, or *halted* if $S(a).\mathsf{status}$ = "idle", "busy", or "halted", respectively.
Actor a *appears to have failed* if a appears halted or exiled.
Actor a *appears blocked* if a appears idle and $\mathsf{sent}(a) = \mathsf{received}(a)$.

Closure. When does a collage S have enough snapshots to judge if a certain actor b is quiescent? Certainly a snapshot from b alone is not enough—we at least need to know how many messages were sent to b by its parent. In addition, if b or its parent gave other actors references to b, then we need snapshots from those actors as well. This leads us to the notion of closure, and how we can use contact tracing to check if a collage appears closed:

Definition 4.3. Actor a is *hereto-acquainted* with actor b at time t if a had a reference to b at some time $t' \leq t$. A collage S is *hereto-closed* for b if every actor hereto-acquainted with b has an effective snapshot.

Actor a *appears hereto-acquainted* with b if $\mathsf{created}(a,b) > 0$. Collage S *appears hereto-closed* for b if, for every a, $\mathsf{created}(a,b) > 0$ implies a has an effective snapshot.

Apparent Quiescence. Combining all the definitions up to this point, we can finally define what it means for an actor to "appear" quiescent. Formally, we adapt our definition of quiescence (Definition 3.4) as follows.

Definition 4.4. Actor b *appears quiescent* in collage S if b appears to have failed or all of the following hold:

1. S appears hereto-closed for b;
2. b appears blocked;
3. b does not appear sticky;
4. If any a appears acquainted with b, then a appears quiescent; and
5. If any a appears monitored by b, then a appears quiescent, does not appear to have failed, and is located on the same node as b.

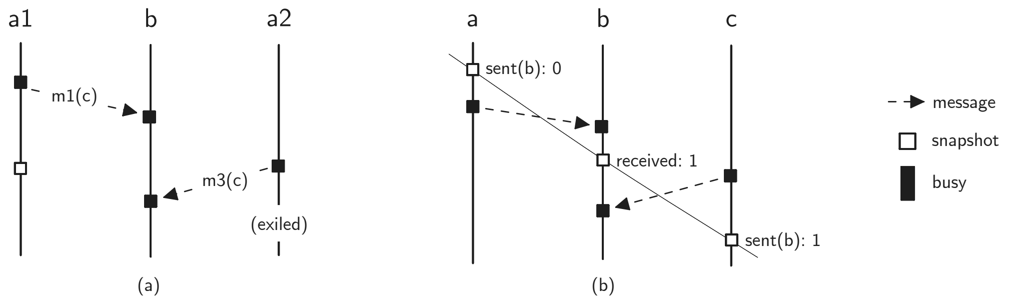

Fig. 8. Examples in the proof of CRGC soundness.

Up to this point, we have taken care to differentiate "apparent" properties from "real" ones because collages are not always consistent—for example, an actor that appears blocked might never have been blocked in reality. But by combining all the properties in the definition above, it so happens that actors appear quiescent *only* if they are truly quiescent. We sketch the key ideas below and give a complete proof in the Supplemental Material.

The property follows from a two-part invariant:

(IV1) If a appears quiescent and has taken a snapshot, then it is not busy.
(IV2) If a has taken a snapshot and has a reference to an actor that appears quiescent, then a appears quiescent.

As long as the invariant holds, any actor b with a reference to an apparently-quiescent actor c will have an effective snapshot. This is because only an actor with an effective snapshot could have given b the reference, and the reference must have been given before the snapshot was recorded. Figure 8 (a) shows an example: if actor a_1 had sent the reference after taking a snapshot, it would violate the invariant because a_1 would be a busy actor with a reference to an apparently quiescent actor. There is also no way for an apparently exiled actor like a_2 to have sent the reference after being exiled, because a_2's node has already been shunned.

Theorem 4.5 (Soundness). *Let S be a collage and let Q be a subset of $dom(S)$ that appears quiescent. Then Q is quiescent.*

Proof (sketch). By contradiction; the proof can be seen as a generalization of Mattern's channel counting argument [31]. Suppose b is the first actor to violate the invariant. There are two cases.

Case 1. Actor b appears quiescent and became busy after taking a snapshot. This could happen if:

1. *b received a message.* As we argued above, the message must have been sent by an actor a with an effective snapshot. It could not have been sent *after* the effective snapshot because b was the first actor to violate the invariant. This is shown in Figure 8 (b): actor a could only have sent the message after taking a snapshot if a violated the invariant before b did. Hence all messages sent to b up to this point are accounted for in S, and since b appears blocked, these messages must have all been received before b's snapshot; a contradiction.
2. *b is sticky and received a wakeup signal.* Impossible because we assumed b appears quiescent, so b was not sticky when it took a snapshot.
3. *b monitors an actor c and is notified that c halted.* Impossible because b only monitors local actors that appear quiescent, and c would need to become busy before it could halt.

Case 2. Let b be the first busy actor that has taken a snapshot, does not appear quiescent, and has a reference to some c that appears quiescent. The actor that gave the reference to b must have an

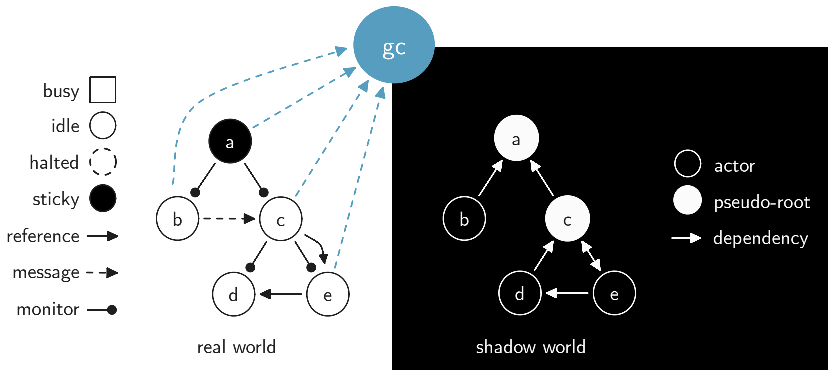

Fig. 9. Actors send snapshots to the garbage collector, which uses the snapshots to construct a shadow graph.

effective snapshot. The reference could not have been given after the sender's effective snapshot. Hence all c-references sent to b up to this point are accounted for in S, and since b does not appear quiescent, these references must have been deactivated before its snapshot; a contradiction. □

We also show that quiescent actors eventually appear quiescent, once the garbage collector has recent-enough snapshots and once notifications about dropped messages have been delivered.

THEOREM 4.6 (COMPLETENESS). *Assume that:*

1. *Every crashed node is eventually exiled, and at least one node is never exiled.*
2. *Ingress points and actors on healthy nodes will always eventually take a snapshot.*
3. *Healthy actors are eventually notified about dropped messages from healthy nodes.*

Then all quiescent actors eventually appear quiescent.

PROOF (SKETCH). Let A be the set of all actors at time t_0, and let $Q \subseteq A$ be the set of actors that are quiescent at t_0. Pick a time $t_1 > t_0$ when all exiled nodes appear exiled and all healthy actors in A have been notified about dropped messages from healthy nodes up to time t_0. Let S be a collage in which every actor in A either appears exiled or has taken a snapshot after t_1. Then for every $c \in Q$, $\text{sent}(c) = \text{received}(c)$; and for every a, $\text{created}(a, c) > \text{deactivated}(a, c)$ if and only if $a \in Q$. Hence all the actors in Q appear quiescent. □

4.2 Shadow Graphs and Undo Logs

In the previous section, we represented a collage S as a mapping from actors and ingress points to local snapshots. In this section we show that collages can be represented more efficiently as a pair of data structures: a shadow graph and a set of undo logs. In Section 5, we will show how shadow graphs and undo logs can be built incrementally by merging updates from actors and ingress points, so a full actor snapshot never needs to be stored.

4.2.1 Shadow Graphs. Given a collage S, a shadow graph is a directed multigraph with a node for each actor that occurs anywhere in S. Formally, *shadows* are defined in Figure 10 and a *shadow graph* is a map G from actors to their shadows. Shadows are the nodes in the shadow graph, and we say there is an edge from a to b if $G(a).\text{references}(b) > 0$ or if $b \in G(a).\text{watchers}$.

Shadow graphs have less information than collages. In a collage, the snapshot for actor a includes the number of messages a sent to each hereto-acquaintance b, as well as the number of references

$$s.\text{interned} = \begin{cases} \textsc{True} & \text{if } b \in \text{dom}(S) \\ \textsc{False} & \text{otherwise} \end{cases} \qquad s.\text{isSticky} = \begin{cases} S(b).\text{isSticky} & \text{if } b \in \text{dom}(S) \\ \textsc{Undefined} & \text{otherwise} \end{cases}$$

$$s.\text{status} = \begin{cases} S(b).\text{status} & \text{if } b \in \text{dom}(S) \\ \textsc{Undefined} & \text{otherwise} \end{cases} \qquad s.\text{watchers} = \{a : b \in S(a).\text{monitored}\}$$

$$s.\text{undelivered} = \sum_{a \in \text{dom}(S)} S(a).\text{sent}(b) - \text{received}(b)$$

$$s.\text{references}(c) = \sum_{a \in \text{dom}(S)} S(a).\text{created}(b, c) - \text{deactivated}(b, c)$$

Fig. 10. The definition of a shadow for actor b in collage S.

a created. In a shadow graph, message send and receive counts are collapsed into a single field, $G(b)$.undelivered. Likewise, reference creation and deactivation counts for b are collapsed into the map $G(b)$.references.

While the domain of a collage is the set of actors that have taken snapshots, the domain of a shadow graph is the set of actors that *occur* in the snapshots. For example, if actor a has a snapshot in which $S(a).\text{sent}(b) > 0$, then b has a node in the shadow graph; the bit $G(b)$.interned indicates whether b has a snapshot in S.

Although shadow graphs have less information than collages, they retain enough information to identify garbage. Borrowing terminology from Wang and Varela [51], we identify the shadows that are self-evidently not garbage:

Definition 4.7. Actor b is a *pseudo-root* in shadow graph G if any of the following hold:

1. $G(b)$.interned is False, $G(b)$.isSticky is True, $G(b)$.status is "busy", or $G(b).\text{undelivered} \neq 0$.
2. There exists a such that $b \in G(a)$.watchers and $G(a)$.status is "halted".

Instead of searching for actors that appear quiescent in S, the garbage collector can build a shadow graph G and "mark" all the shadows reachable from a pseudo-root. For instance, in Figure 9, actors a and c are pseudo-roots; actors a, c, d, and e are marked; and b is unmarked. We will show that, in the absence of faults, the unmarked actors in G are the same actors that appear quiescent in S.

4.2.2 Undo Logs. When a node N is exiled, the snapshots from actors on N should be replaced with snapshots from ingress points. Doing so is straightforward if we encode a collage as a collection of snapshots. But when we encode a collage as a shadow graph, it is unclear how to roll back the effects of only those snapshots produced by N. For this, we introduce the concept of an undo log. Undo logs indicate how the shadow graph should be modified when a specific node is exiled.

Definition 4.8. Given a collage S, we define the *undo log* L for node N to be a record:

$$L.\text{undeliverableMsgs}(b) = \sum_{a \in S(N)} S(a).\text{sent}(b) - S(I_{N,N_b}).\text{admittedMsgs}(b) \tag{1}$$

$$L.\text{undeliverableRefs}(b, c) = \sum_{a \in S(N)} S(a).\text{created}(b, c) - S(I_{N,N_b}).\text{admittedRefs}(b, c), \tag{2}$$

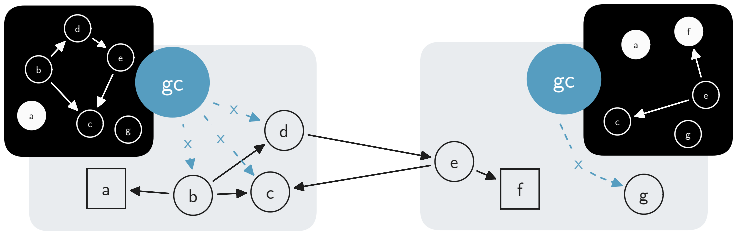

Fig. 11. Two nodes using CRGC to collect garbage. Each GC has a shadow graph, representing its view of the cluster. When a GC finds one of its local actors is garbage, it asks the actor to stop.

where N_b denotes the location of b. That is, L.undeliverableMsgs(b) is the number of messages that appear sent to b by N but not admitted; likewise, L.undeliverableRefs(b, c) is the number of references to c that appear sent to b by N but not admitted.

Whereas shadow graphs are the GC's view of the actors, undo logs are the GC's view of the network. Thus we expect the undo logs to be small as long as snapshots from ordinary actors and ingress points do not fall too far "out of sync" with one another. Remarkably, undo logs are all we need to recover garbage after nodes have been exiled.

Definition 4.9. Let G be a shadow graph, let $N_1, \dots, N_k$ be the set of nodes that appear exiled, and let $L_1, \dots, L_k$ be the undo logs for those nodes. We define the *amended shadow graph* as a shadow graph $\tilde{G}$ where, for each actor a:

1. $a \in \text{dom}(\tilde{G})$ if $a \in \text{dom}(G)$ and either (a) a does not appear exiled, or (b) a appears exiled and $G(a)$.watchers contains actors that do not appear failed;
2. $\tilde{G}(a)$.status = halted if a appears exiled and an actor in $G(a)$.watchers does not appear failed;
3. $\tilde{G}(a)$.undelivered = $G(a)$.undelivered − L.undeliverableMsgs(a);
4. $\tilde{G}(a)$.references(b) = $G(a)$.references(b) − L.undeliverableRefs(a, b);
5. $\tilde{G}(a)$.watchers is equal to $G(a)$.watchers, excluding any actors that appear failed; and
6. $\tilde{G}(a)$ is the same as $G(a)$ in all other cases.

The amended shadow graph removes shadows from exiled actors, except when those actors are monitored by non-faulty actors. The graph also repairs reference counts and message counts to account for messages that will never be delivered. As a result, the unmarked actors in the amended shadow graph coincide with the actors that appear quiescent in the collage.

Theorem 4.10 (Shadow Graph Equivalence). *Let S be a collage, let $\tilde{G}$ be its amended shadow graph, and let b be an interned actor that does not appear exiled. Then b is unmarked in $\tilde{G}$ if and only if b appears weakly quiescent in S.*

5 Implementation

We added CRGC to Apache Pekko, a popular actor library for Scala and Java. We chose Pekko because it has a built-in cluster membership service (unlike Erlang and Elixir) and supports distributed actors. However, a complete implementation of any actor GC in Pekko exceeds the scope of this work because Pekko has a large API that was not designed with garbage collection in mind. We focus here on the performance and correctness issues involved in implementing CRGC, and conclude the section with a survey of open challenges ranging from type safety to API design.

The architecture of our implementation is shown in Figure 11. Every node has a local garbage collector (GC). Each GC is an actor with its own shadow graph and set of undo logs; actors and ingress points send incremental updates to their local GC, which in turn broadcasts those updates to other GCs in the cluster. Each GC periodically wakes up, merges incoming entries into its shadow graph and undo logs, then traces its shadow graph and kills any actors on its node that appear to be garbage. When another node is exiled from the cluster, the GC uses its undo log for the exiled node to amend its shadow graph. We report the details of this architecture below.

5.1 Diary Entries

In Section 4, every actor's local state contained a cumulative history of every GC-related action it ever performed, and actors sent snapshots of this state to the garbage collector. This is inefficient because actor states and snapshots grow without bound and because the garbage collector needs to rebuild the shadow graph and undo logs from scratch whenever it receives an updated batch of snapshots. In our Pekko implementation, we converted this offline algorithm into an online one.

In our Pekko implementation, actors send the local GC incremental updates called *diary entries* (or simply *entries*). An actor's local state only contains information about actions the actor took since its last entry. When the GC receives an entry, it merges the entry into its shadow graph and undo logs. As long as entries from an actor are merged in FIFO order and never dropped, the resulting shadow graph is equivalent to a graph constructed by the offline algorithm in Section 4.2.

Users of CRGC can customize how often actors send entries. Sending entries more frequently can lower the time it takes for garbage to be detected, but increases the amount of time the GC spends merging entries into the graph. We implemented two simple policies: (1) *Wave:* Actors send entries every 50 milliseconds; and (2) *On-block:* Actors send entries when their mail queue is empty.

5.2 Ingress and Egress Points

In Pekko, a node's incoming and outgoing messages pass through a pipeline for serialization and other processing. For CRGC, we instrumented the incoming message pipeline with an ingress point for each node in the cluster, as described in Section 4.1.2. Ingress points, like ordinary actors, send entries to their local GC and their local state only records information collected since the last entry was sent.

To detect dropped messages between nodes, we use Puaut's approach [40]. Each outgoing message pipeline is instrumented with an *egress point*, which records the same data as ingress points (Table 3) but for outgoing messages. Egress points periodically flush their entry to the ingress point downstream. By comparing the number of messages sent by the egress point with the number of messages admitted by the ingress point, one can deduce how many messages were dropped and what references those messages contained.

5.3 An API for Actor GC

We created a version of Pekko's API for *managed actors*, whose lifetime is controlled by an actor GC. The API supports multiple GC backends, including preliminary implementations of CRGC and weighted reference counting (WRC). We chose to make the API backwards-compatible, so programmers can gradually migrate large codebases into the new API over time. This design raises interesting challenges that any production-ready actor GC will need to overcome.

References. In actor languages like Erlang, Elixir, or Pony [14], the garbage collector can scan an actor's heap to find actor references. But Pekko is a library, so our API is implemented in userspace. To track actor references when a sends b a reference to c, we ask programmers to explicitly *create* a new reference from b to c. When b receives the reference, it uses the JVM's `PhantomReference` mechanism to detect when the reference is no longer needed. This design is vulnerable to bugs if the programmer forgets to create a reference explicitly; researchers have proposed static analyses and type systems to catch accidental sharing of data of this sort between actors [17, 24], but no such system has yet been integrated into Pekko.

Monitoring. If actor a monitors b in Pekko, then a will be notified if b halted for any reason—including normal termination. This is a paradox for cyclic garbage collectors like CRGC: the local GC could identify a pair of quiescent actors a and b that monitor one another, but it could not kill a without waking up b and vice versa. Changing the semantics of monitoring would break backward-compatibility, so our implementation of CRGC does not collect this type of garbage.

Restarting. When an actor throws an uncaught exception, Pekko allows the actor to restart itself and all its children. This is a problem for actor GCs because every actor in Pekko is descended from some actor that could throw an exception; hence any garbage actor could become non-garbage again because its ancestor forced a restart. We propose that instead of restarting its children, an actor should spawn new children while the old ones are garbage collected. The net effect is the same, except any other actor with references to an old child will need to obtain references to a new child—e.g., by asking the parent.

6 Evaluation

Actor GC is a feature designed to reduce bugs and simplify programs. For a preliminary evaluation of CRGC, we would therefore like to know the cost of this feature in terms of application performance. We pose the following research questions:

RQ1. How does CRGC affect the performance of applications that do not need actor GC?
RQ2. How promptly does CRGC collect actor garbage?
RQ3. What is the network overhead of CRGC in a distributed application?

6.1 Savina Benchmarks

Savina is a multicore benchmark suite for actor frameworks [19], featuring microbenchmarks, concurrency benchmarks, and parallelism benchmarks. Most of the benchmarks in Savina produce no actor garbage at all, so the purpose of this evaluation is to measure the *overhead* of adding actor GC to an application where it is not needed. We provide two baselines: (1) a baseline with actor GC disabled, and (2) a baseline with *weighted reference counting (WRC)* [52], a simple and lightweight acyclic GC.

Experimental setup. The experiments were conducted on a 2.10 GHz Intel Xeon Gold 6130 SMP node, allocated 8 vCPUs and 48 GB RAM, running Ubuntu 24.04 and OpenJDK Temurin 17. To minimize the effect of unpredictable JVM GC pauses and JIT warmup times, we ran each benchmark for 20 iterations and six separate JVM invocations, and then filtered out the slowest 60% of benchmark run times. Each JVM instance was given 16 GB heap and used ZGC, a low-latency object garbage collector for the JVM. For each category of benchmarks, we computed the geometric mean slowdown by normalizing execution times with respect to the baseline, taking the geometric mean, and converting the result to a percentage value.

In the original Savina benchmarks, execution time included the time for all actors in the system to terminate themselves. As remarked by Blessing et al. [9], this does not accurately reflect real actor systems and unfairly penalizes garbage collectors that run less often than necessary. In the process of porting the benchmarks to use our API, we therefore modified them to exclude termination time from measurements.

Microbenchmarks and Concurrency. Savina's microbenchmarks and concurrency benchmarks measure basic actor operations, which are designed to be very cheap; hence we expect any overhead

Table 4. Savina benchmarks.

		Average time (s)				**Average overhead (%)**			
	Benchmark	*No GC*	*WRC*	*CRGC-block*	*CRGC-wave*	*No GC (stdev)*	*WRC*	*CRGC-block*	*CRGC-wave*
Microbenchmarks	`big`	2.7	4.4	3.8	4.6	±8	63	42	70
	`chameneos`	1.7	1.8	1.7	2	±11	6	2	14
	`count`	1.6	1.6	1.7	2.7	±10	-1	3	67
	`fib`	0.1	0.11	0.14	0.15	±5	7	35	45
	`fjcreate`	1.3	1.6	1.4	1.3	±2	23	4	1
	`fjthrput`	0.39	0.93	0.77	1	±23	141	99	167
	`pingpong`	2.5	2.7	2.8	3	±2	8	12	18
	`threadring`	0.76	0.91	0.99	0.97	±4	19	29	27
	geomean						27	25	44
Concurrency	`banking`	0.091	0.11	0.13	0.13	±6	20	47	45
	`bndbuffer`	0.55	0.56	0.58	0.57	±1	1	4	2
	`cigsmok`	0.12	0.13	0.12	0.12	±11	8	-1	0
	`concdict`	0.76	0.81	0.83	0.87	±2	6	10	14
	`concsll`	30	30	33	33	±2	0	10	11
	`logmap`	0.17	0.16	0.17	0.24	±9	-6	1	45
	`philosopher`	1.1	1.2	1.2	1.3	±2	8	8	15
	geomean						5	10	18
Parallelism	`apsp`	0.8	0.78	0.82	0.81	±5	-1	3	1
	`astar`	1.9	1.6	1.8	1.6	±27	-12	-5	-11
	`bitonicsort`	0.11	0.15	0.11	0.17	±5	39	4	54
	`facloc`	0.27	0.28	0.4	0.4	±2	4	51	51
	`nqueenk`	0.63	0.67	0.67	0.67	±1	5	5	6
	`piprecision`	0.12	0.13	0.13	0.13	±2	0	0	0
	`quicksort`	0.46	0.46	0.45	0.46	±1	0	-1	0
	`radixsort`	1.3	1.3	1.3	1.8	±8	1	2	43
	`recmatmul`	0.44	0.45	0.44	0.45	±1	0	0	0
	`sieve`	0.19	0.19	0.2	0.2	±2	0	1	2
	`trapezoid`	0.21	0.21	0.21	0.21	±1	0	0	0
	`uct`	0.16	0.2	0.19	0.19	±7	25	22	19
	geomean						4	6	12

at all to have an outsized impact on performance. We also found the microbenchmarks highly noisy. For example, WRC performs poorly on `fjthrput`—but this benchmark measures the rate that a manager sends tasks to a fixed team of workers, so reference counting should not be expected to have a significant effect. Overall, there does not appear to be any consistent difference between CRGC and WRC across these benchmarks.

Parallelism. The only parallel benchmark in which CRGC performs decidedly worse than WRC is `facloc`. The benchmark features a "forwarding" behavior in which an actor a receives a reference to actor b, sends b a message, and then deactivates the reference. Each forwarded reference requires growing the "sent" field in a's entry (Table 2), causing the GC to spend more time processing entries. Some of these costs could be reduced by taking advantage of static typing information,

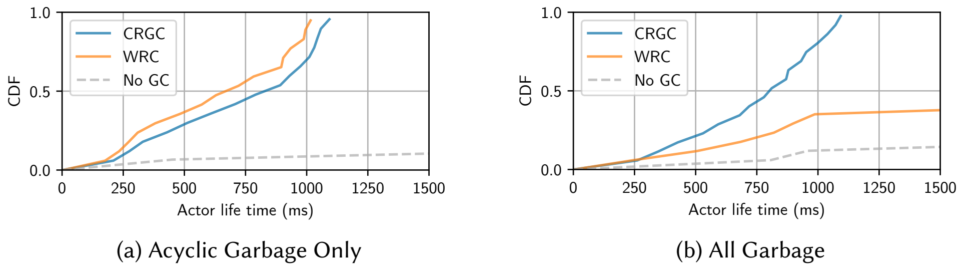

Fig. 12. CDFs plotting how long actors survive in the "torture test" configuration (shorter lifetimes are better).

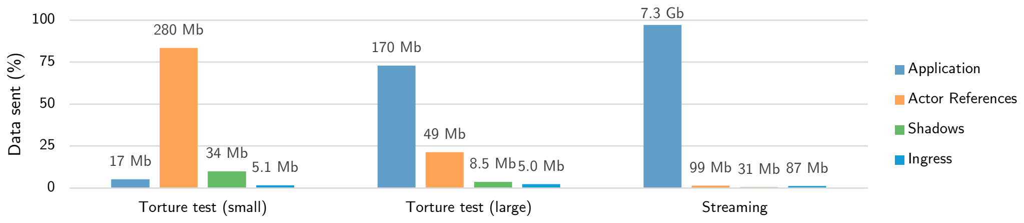

Fig. 13. Network overhead of CRGC three configurations of `RandomWorkers`.

like in Pony/Orca [14]. Without that information, we expect CRGC to have higher overhead in applications that pass references more frequently.

6.2 `RandomWorkers`: A Configurable GC Benchmark

The Savina benchmarks are useful for measuring overhead, but they are not realistic applications of CRGC because they are bound to a single node and they generate garbage in a predictable way. An alternative benchmark called `ChatApp` has been proposed [9], but it suffers the same limitations. We fill the gap with a new benchmark adapted from `ChatApp`, called `RandomWorkers`: a distributed benchmark that generates garbage randomly and can be configured to model different workloads.

`RandomWorkers` models a team of nodes handling a stream of requests from some frontend. Each node has a manager actor that can respond to requests by randomly (1) spawning a worker; (2) acquainting a manager or worker with a set of workers; or (3) sending work to a worker. Workers can also spawn workers and pass references.

`RandomWorkers` can be configured to model different applications. The benchmark can be made more or less dynamic by changing the probability of spawning actors or passing references. If the probability of passing references is zero, then all garbage has a tree topology. If the probability of passing references to a *remote* actor is zero, then there is no distributed garbage. We can also guarantee actor garbage is acyclic by imposing a lexicographic order on actor IDs, and ensuring actors are never given references to targets with a "smaller" name.

We evaluated CRGC with respect to three configurations of the benchmark:

- *Torture test (small messages):* Application messages contain a payload of 0 to 50 bytes. When a worker receives a message, the worker has a 30% chance of spawning or sending a reference. Likewise, managers have a 50% chance of spawning a worker, a 50% chance of sending to a remote actor, and a 70% chance of sending to a local worker.
- *Torture test (large messages):* As above, but the payload may be up to 5 KB.

- *Streaming:* As above, but with a less dynamic topology: payload sizes up to 5 KB; workers do not spawn or create references; managers have a 50% chance of sending a message to a remote actor and a 70% chance of sending to a local worker, but only a 1% chance of spawning a worker.

6.2.1 Garbage Collection Rates. We evaluate how promptly CRGC collects garbage in the *torture test (small)* configuration. To use WRC as a baseline, only acyclic garbage was generated. The experiment ran on a single node and the JVM GC was triggered every second.

Figure 12a shows the distribution of actor survival times as a CDF. When actor GC is disabled, an actor's survival time depends only on the time it was spawned. The figure shows CRGC is competitive with WRC, though lagging behind slightly; this is an expected result because reference counting actors can stop themselves immediately, whereas CRGC actors must wait for the local GC to trace the shadow graph. For completeness, Figure 12b shows the same experiment with *unrestricted* reference-passing. Here WRC introduces a memory leak, and the experiment will crash if left to run long enough; with CRGC, the experiment does not crash.

6.2.2 Network Overhead. We evaluate the network overhead of CRGC using three configurations of the benchmark, executed on a 3-node cluster. Figure 13 breaks down total network usage by cause. In *torture test (small)*, CRGC adds nearly 20× overhead. However, 88% of this overhead is taken up by actor references, which Pekko represents as fully qualified paths that take dozens of bytes to serialize. When the payloads of application messages are larger and the number of actors is smaller, CRGC imposes much less overhead; in *Streaming*, the overhead is less than 3%.

7 Conclusion

We presented a model for distributed actor systems with faults, and an actor GC that can collect actor garbage resulting from those faults. Our approach is simple enough to be proven correct and makes very few assumptions about the underlying implementation. We also implemented the actor GC in Apache Pekko and found it to have modest overhead, comparable to much simpler and less powerful approaches.

There is still work to be done before actor GCs can enter the mainstream. Our approach cannot be used in Erlang or Elixir unless these languages implement a cluster membership protocol matching the specification in Section 3. In actor libraries like Pekko and Akka, programmers will need support from type systems or static analysis to prevent bugs caused by unintentionally sharing actor references. Moreover, we will need to reevaluate how mechanisms like monitoring and restarting should work when programmers can rely on actor GC instead.

Another topic for future work is tuning CRGC itself. We observed that "forwarding" actors can add undue pressure to local garbage collection. CRGC's use of bandwidth is not optimal either: nodes broadcast all information about their local actors, even if those actors are not part of any distributed cyclic garbage. Methods of summarizing a node's heap have already been investigated [38, 44], but will require a redesign to match CRGC's approach.

Data Availability Statement

The TLA$^+$ specifications, Pekko implementation, and benchmarks for this article are available at https://doi.org/10.5281/zenodo.15049131 [39].

Acknowledgments

Partially supported by Villum Fonden (grant no. 29518). Co-funded by the European Union (ERC, CHORDS, 101124225). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Gul Agha. 1990. *ACTORS - A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA.

[2] Gul Agha. 1990. Concurrent Object-Oriented Programming. *Commun. ACM* 33, 9 (Sept. 1990), 125–141.

[3] Apache Software Foundation. 2025. Apache Pekko. https://pekko.apache.org/.

[4] Joe Armstrong. 2003. *Making Reliable Distributed Systems in the Presence of Software Errors.* Ph. D. Dissertation. Royal Institute of Technology, Stockholm, Sweden.

[5] Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. 2020. Actor Concurrency Bugs: A Comprehensive Study on Symptoms, Root Causes, API Usages, and Differences. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 214 (Nov. 2020). doi:10.1145/3428282

[6] Di Bevan. 1987. Distributed Garbage Collection Using Reference Counting. In *PARLE Parallel Architectures and Languages Europe*, G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, J. W. Bakker, A. J. Nijman, and P. C. Treleaven (Eds.). Vol. 259. Springer Berlin Heidelberg, Berlin, Heidelberg, 176–187. doi:10.1007/3-540-17945-3_10

[7] A Birrell, D Evers, G Nelson, S Owicki, and G Wobber. 1993. *Distributed Garbage Collection for Network Objects.* Technical Report 116. Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301.

[8] Sebastian Blessing. 2013. *A String of Ponies: Transparent Distributed Programming with Actors.* Master's thesis. Imperial College, London, United Kingdom.

[9] Sebastian Blessing, Kiko Fernandez-Reyes, Albert Mingkun Yang, Sophia Drossopoulou, and Tobias Wrigstad. 2019. Run, Actor, Run: Towards Cross-Actor Language Benchmarking. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2019.* ACM Press, Athens, Greece, 41–50. doi:10.1145/3358499.3361224

[10] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11.* ACM Press, Cascais, Portugal, 1–14. doi:10.1145/2038916.2038932

[11] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (March 1996), 225–267. doi:10.1145/226643.226647

[12] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems* 3, 1 (Feb. 1985), 63–75. doi:10.1145/214451.214456

[13] 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 '13.* ACM Press, Indianapolis, Indiana, USA, 553–570. doi:10.1145/2509136.2509557

[14] Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. 2017. Orca: GC and Type System Co-Design for Actor Languages. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 1–28. doi:10.1145/3133896

[15] Peter Dickman. 1996. Incremental, Distributed Orphan Detection and Actor Garbage Collection Using Graph Partitioning and Euler Cycles. In *Distributed Algorithms, 10th International Workshop, WDAG '96, Bologna, Italy, October 9-11, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1151)*, Özalp Babaoglu and Keith Marzullo (Eds.). Springer, 141–158. doi:10.1007/3-540-61769-8_10

[16] Hadoop S3A 2025. S3A Committers: Architecture and Implementation. https://hadoop.apache.org/docs/r3.1.0/hadoop-aws/tools/hadoop-aws/committer_architecture.html.

[17] Philipp Haller and Alex Loiko. 2016. LaCasa: Lightweight Affinity and Object Capabilities in Scala. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* ACM, Amsterdam Netherlands, 272–291. doi:10.1145/2983990.2984042

[18] Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama. 2004. The Φ Accrual Failure Detector. In *23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil.* IEEE Computer Society, 66–78. doi:10.1109/RELDIS.2004.1353004

[19] Shams M. Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control - AGERE! '14.* ACM Press, Portland, Oregon, USA, 67–80. doi:10.1145/2687357.2687368

[20] Richard E. Jones, Antony L. Hosking, and J. Eliot B. Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* CRC Press.

[21] D. Kafura, M. Mukherji, and D.M. Washabaugh. 1995. Concurrent and Distributed Garbage Collection of Active Objects. *IEEE Transactions on Parallel and Distributed Systems* 6, 4 (April 1995), 337–350. doi:10.1109/71.372788

[22] Dennis G. Kafura, Douglas Washabaugh, and Jeff Nelson. 1990. Garbage Collection of Actors. In *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming, OOPSLA/ECOOP 1990, Ottawa, Canada, October 21-25, 1990, Proceedings*, Akinori Yonezawa (Ed.). ACM, 126–134. doi:10.1145/97945.97961

[23] Tomio Kamada, Satoshi Matsuoka, and Akinori Yonezawa. 1994. Efficient Parallel Global Garbage Collection on Massively Parallel Computers. In *Proceedings Supercomputing '94, Washington, DC, USA, November 14-18, 1994*, Gary M. Johnson (Ed.). IEEE Computer Society, 79–88. doi:10.1109/SUPERC.1994.344268

[24] Rajesh K. Karmani, Amin Shali, and Gul Agha. 2009. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ 2009, Calgary, Alberta, Canada, August 27-28, 2009*, Ben Stephenson and Christian W. Probst (Eds.). ACM, 11–20. doi:10.1145/1596655.1596658

[25] Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems* 16, 3 (May 1994), 872–923. doi:10.1145/177492.177726

[26] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '16*. ACM Press, Atlanta, Georgia, USA, 517–530. doi:10.1145/2872362.2872374

[27] Lightbend. 2025. Akka. https://akka.io/.

[28] Lightbend. 2025. Akka Documentation: Cluster Membership Service. https://doc.akka.io/docs/akka/2.10.2/typed/cluster-membership.html.

[29] Lightbend. 2025. Akka Documentation: Cluster Sharding. https://doc.akka.io/docs/akka/2.10.2/typed/cluster-sharding.html.

[30] Lightbend. 2025. Akka Documentation: Message Delivery Reliability. https://doc.akka.io/docs/akka/2.10.2/general/message-delivery-reliability.html.

[31] Friedemann Mattern. 1987. Algorithms for Distributed Termination Detection. *Distributed Computing* 2, 3 (Sept. 1987), 161–175. doi:10.1007/BF01782776

[32] Luc Moreau, Peter Dickman, and Richard E. Jones. 2005. Birrell's Distributed Reference Listing Revisited. *ACM Trans. Program. Lang. Syst.* 27, 6 (2005), 1344–1395. doi:10.1145/1108970.1108976

[33] MR-4099 2012. [MAPREDUCE-4099] ApplicationMaster May Fail to Remove Staging Directory - ASF JIRA. https://issues.apache.org/jira/browse/MAPREDUCE-4099.

[34] Orleans project authors. 2024. Activation Garbage Collection. https://github.com/dotnet/docs/blob/0ba574212ac88fe70b590547384eae7a158d4527/docs/orleans/host/configuration-guide/activation-collection.md.

[35] José M. Piquer. 1991. Indirect Reference Counting: A Distributed Garbage Collection Algorithm. In *Parle '91 Parallel Architectures and Languages Europe*, Emile H. L. Aarts, Jan van Leeuwen, and Martin Rem (Eds.). Vol. 505. Springer Berlin Heidelberg, Berlin, Heidelberg, 150–165. doi:10.1007/978-3-662-25209-3_11

[36] 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 2018*. ACM Press, Boston, MA, USA, 44–53. doi:10.1145/3281366.3281368

[37] Dan Plyukhin and Gul Agha. 2020. Scalable Termination Detection for Distributed Actor Systems. In *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference) (LIPIcs, Vol. 171)*, Igor Konnov and Laura Kovács (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:23. doi:10.4230/LIPIcs.CONCUR.2020.11

[38] Dan Plyukhin and Gul Agha. 2022. A Scalable Algorithm for Decentralized Actor Termination Detection. *Log. Methods Comput. Sci.* 18, 1 (2022). doi:10.46298/lmcs-18(1:39)2022

[39] Dan Plyukhin, Gul Agha, and Fabrizio Montesi. 2025. CRGC: Fault-Recovering Actor Garbage Collection in Pekko (Artifact). https://doi.org/10.5281/zenodo.15049131.

[40] Isabelle Puaut. 1994. A Distributed Garbage Collector for Active Objects. In *PARLE'94 Parallel Architectures and Languages Europe*, Gerhard Goos, Juris Hartmanis, Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis (Eds.). Vol. 817. Springer Berlin Heidelberg, Berlin, Heidelberg, 539–552. doi:10.1007/3-540-58184-7_129

[41] Aleta M. Ricciardi and Kenneth P. Birman. 1991. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing - PODC '91*. ACM Press, Montreal, Quebec, Canada, 341–353. doi:10.1145/112600.112628

[42] Marc Shapiro, Peter Dickman, and David Plainfosse. 1993. SSP Chains : Robust, Distributed References Supporting Acyclic Garbage Collection.

[43] Hans Svensson and Lars-Åke Fredlund. 2007. A More Accurate Semantics for Distributed Erlang. In *Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop*. ACM, Freiburg Germany, 43–54. doi:10.1145/1292520.1292528

[44] Abhay Vardhan. 1998. *Distributed Garbage Collection of Active Objects: A Transformation and Its Applications to Java Programming*. Master's thesis. University of Illinois at Urbana-Champaign, Urbana, IL.

[45] Abhay Vardhan and Gul Agha. 2003. Using Passive Object Garbage Collection Algorithms for Garbage Collection of Active Objects. *ACM SIGPLAN Notices* 38, 2 supplement (Feb. 2003), 106. doi:10.1145/773039.512443

[46] Carlos Varela and Gul Agha. 2001. Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIGPLAN Notices* 36, 12 (Dec. 2001), 20–34. doi:10.1145/583960.583964

[47] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*. ACM, Santa Clara California, 1–16. doi:10.1145/2523616.2523633

[48] Nalini Venkatasubramanian. 1992. *Hierarchical Garbage Collection in Scalable Distributed Systems*. Master's thesis. University of Illinois at Urbana-Champaign.

[49] Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. 1992. Scalable Distributed Garbage Collection for Systems of Active Objects. In *Memory Management*, Yves Bekkers and Jacques Cohen (Eds.). Vol. 637. Springer-Verlag, Berlin/Heidelberg, 134–147. doi:10.1007/BFb0017187

[50] Wei-Jen Wang, Carlos 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*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Paolo Bellavista, Ruay-Shiung Chang, Han-Chieh Chao, Shin-Feng Lin, and Peter M. A. Sloot (Eds.). Vol. 6104. Springer Berlin Heidelberg, Berlin, Heidelberg, 244–255. doi:10.1007/978-3-642-13067-0_28

[51] 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*, Yeh-Ching Chung and José E. Moreira (Eds.). Vol. 3947. Springer Berlin Heidelberg, Berlin, Heidelberg, 360–372. doi:10.1007/11745693_36

[52] Paul Watson and Ian Watson. 1987. An Efficient Garbage Collection Scheme for Parallel Computer Architectures. In *PARLE Parallel Architectures and Languages Europe*, G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, J. W. Bakker, A. J. Nijman, and P. C. Treleaven (Eds.). Vol. 259. Springer Berlin Heidelberg, Berlin, Heidelberg, 432–443. doi:10.1007/3-540-17945-3_25

Received 2024-11-15; accepted 2025-03-06