FAULT-TOLERANT AND FAULT-RECOVERING GARBAGE COLLECTION FOR THE ACTOR MODEL: A COLLAGE-BASED APPROACH

BY

DAN PLYUKHIN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Doctoral Committee:

Professor Emeritus Gul Agha, Chair
Professor Indranil Gupta
Assistant Professor Tianyin Xu
Associate Professor Philipp Haller, KTH Royal Institute of Technology

Abstract

An *actor garbage collector (actor GC)* is a tool for automatically identifying actors that are safe to delete, and reclaiming their resources. Actor GC could be particularly useful in distributed applications, because programmers have difficulty reclaiming resources after faults such as crashed nodes or dropped messages. Unfortunately, faults are a pain point in existing actor GCs: in existing approaches, an actor on a crashed node with a reference to an actor on a healthy node will prevent the healthy actor—and its references—from ever being garbage collected. Moreover, existing GC algorithms have poor scalability in a distributed systems. This is because of the synchronization and message overhead they introduce by requiring causal delivery, or by introducing a large number of control messages. For these reasons, it has not been practical to add actor GC to popular frameworks like Akka and Erlang.

This thesis explores an emerging technique for actor GC, dubbed the *collage-based approach*. Collage-based GCs are capable of high performance because they do not dictate when an actor should participate in garbage collection, and by design they naturally make progress with only partial information. The thesis presents two collage-based GCs: PRL and CRGC. Both GCs are provably correct and impose no locks, memory barriers, or message ordering requirements. PRL uses distributed reference listing to collect acyclic garbage and allows node-local garbage collectors to detect distributed cyclic garbage via a lightweight gossip protocol. We then use insights from PRL to develop CRGC: the first actor GC capable of recovering from crashed nodes and dropped messages. We have formalized CRGC in TLA+ and implemented CRGC in Akka. Preliminary evaluation shows that CRGC imposes little overhead in practice and is capable of collecting actors that become garbage caused by crashed nodes.

For my parents, of course

Acknowledgments

The research for this thesis was conducted during my time at the University of Illinois and the University of Southern Denmark. My Ph.D. advisor, Gul Agha, developed the elegant formalism of the actor model and patiently shepherded me along the strange, meandering path to the completion of this thesis. I couldn't imagine a better supervisor than Gul, and if the reader can understand anything at all in this document, it is thanks to his guidance. I also warmly thank my supervisor, Fabrizio Montesi, at SDU for being open to this research and allowing it to incubate in Odense. I look forward to many more years of enthusiastic collaboration.

Implementing CRGC was challenging. I could not have done it without Charles Kuch and Jerry Wu, who spent a semester implementing PRL with me when they were undergraduates. The garbage collector presented in this thesis is heavily inspired by the details we uncovered during the development of that prototype. I also appreciate Patrik Nordwall for taking time out of the workday to help a lowly student understand Akka's Artery and Downing systems.

Many researchers have shaped the content of this thesis. The members of OSL—Dipayan Mukherjee, Atul Sandur, Kirill Mechitov, YoungMin Kwon, and Laine Taffin Altman—gave feedback to many iterations of this work. The members the ACP Section have all been wildly encouraging and curious—it's been a joy to be a part of that community. The attendees and reviewers of the *AGERE!* workshop, where I presented the first version of this work, were also very encouraging and gave me perspective. The delightful term "shadow graph" is stolen from a short conversation with Tobias Wrigstad at *AGERE!*

Finally, I thank my friends, family, and members of my community for their support: Matt, Alex, Kat, Lira, Sergei, Jamie, Mary Beth, Pedro, Rob, Arpan, Dan, Micky, Lovro, Valentino, Eva, Robert, Jonas, Alexey, and the WEFT 90.1 FM community radio station for East Central Illinois. And of course, I thank Mimi Hutchinson: the first person to show interest in my research, and the one who patiently collected the garbage left in its wake.

TABLE OF CONTENTS


uigc

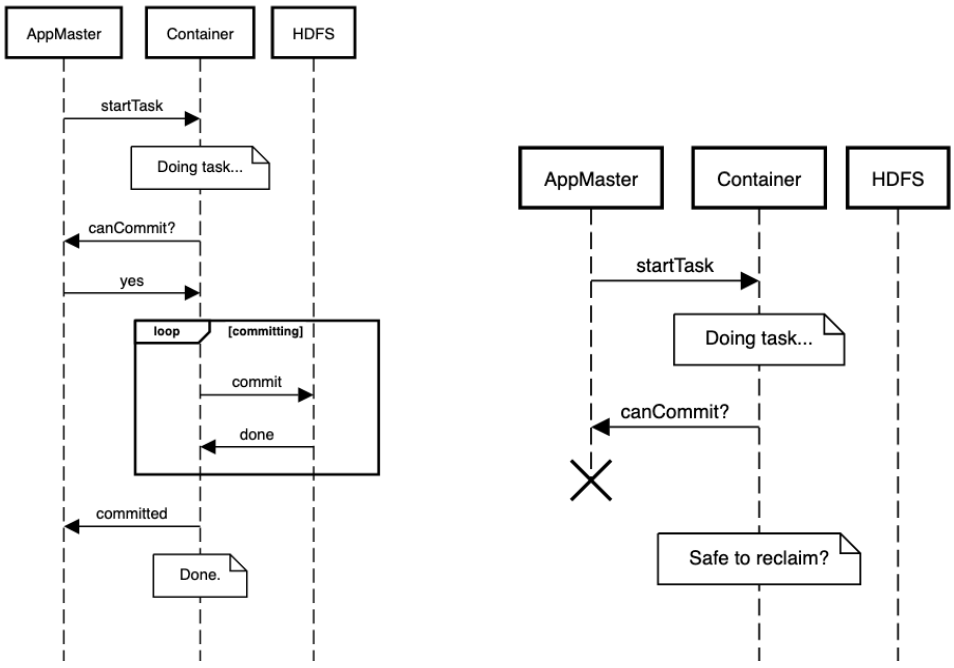
1

Introduction

Distributed systems consist of processes sharing a limited pool of *resources*. These resources could be containers, segments of memory, slices of CPU time, or file handles. The challenge of *distributed resource management* is to programmatically detect when resources are no longer needed by one part of the application so they can be reclaimed for use in another part of the application.

Figure 1.1 depicts a resource management problem based on the Hadoop YARN architecture [1]. In Figure 1.1a a node called the Application Master begins by assigning a task to a specific container. Once the task is completed, the container needs to be allocated to another task—but only after the Application Master and the distributed storage service have finished reading the results from the container. This problem is complicated by the fact that distributed systems have faults: messages might be dropped and remote nodes might fail unexpectedly at any time. Figure 1.1b shows one such case, where the Application Master failed before the protocol could complete. Programmers must not only anticipate such cases, but also choose a correct action to take. In this case, choosing to reclaim the container immediately could lead to an error if some process tries to access the task's results in the future. On the other hand, choosing to *never* reclaim the container would result in a *resource leak*. For safety and performance, programmers need to carefully determine the perfect moment to reclaim every resource.

We investigate the problem formally using the actor model of concurrency [3]. In the actor model, reactive processes known as *actors* execute concurrently and communicate by sending asynchronous messages to other actors. Actors encapsulate resources, so any part of the application needing the resource must do so by sending asynchronous messages to the enclosing actor. In the example above, we would encapsulate the container resource within a dedicated `Container` actor. Encapsulating resources this way has the benefit of controlling concurrent access (only the enclosing actor can directly use the resource) and simplifying resource cleanup (terminating the enclosing actor causes the resource to be reclaimed). The actor abstraction is widely used in industry; actor frameworks have been used to build NoSQL databases [4, 5], reactive streaming



(a) Non-faulty execution (b) Faulty execution

Figure 1.1: Two possible executions of a task commit protocol, based on Hadoop YARN [2].

services [6, 7], fault-tolerant message brokers [8], high-throughput web servers [9], low-latency financial applications [10], and scalable machine learning applications [11].

The actor model reduces the problem of *distributed resource management* to the problem of *distributed actor garbage collection*. Automatic actor garbage collectors (called *actor GCs* for short) [10, 12, 13, 14, 15, 16, 17, 18] can exploit structural properties of actor systems to decide when an actor is safe to terminate, thereby releasing its resources back to the system. In practice, programmers can use automatic actor GC in two ways:

1. *Entrusting all resource management to the GC.* This allows programmers to remove all resource management logic from their programs and also gives programmers more freedom in how they design systems.
2. *Using automatic GC to catch bugs.* In systems where explicit resource management is essential, actor GC can be used to detect resource leaks [19].

Despite the potential benefits, automatic actor GC is *not* provided by either of the two most popular actor frameworks, Akka and Erlang [20]. Instead, programmers are required to terminate actors manually. No study has yet been conducted on how many bugs could be prevented if these frameworks did provide actor GC—but a recent survey of Akka actor framework bugs reveals that manual actor termination does cause race conditions and explicit lifecycle bugs, which have a prevalence of 14.5% and 12.4% in the corpus, respectively [20].

Existing actor GCs have two limitations that prevent them from being used in popular actor frameworks:

1. *Performance:* Many actor GCs demand features, such as causal delivery [10] or exactly-once delivery [12], that popular frameworks cannot provide for performance reasons. Other GCs impose overhead through too many control messages [18] or untimely GC pauses [13].
2. *Fault recovery:* No actor GC so far is capable of detecting garbage produced by faults, such as the container in Figure 1.1.

The latter condition is particularly important, since it leaves the most difficult problem—reasoning about unexpected failures—up to the programmer, negating many of the benefits an actor GC could have.

Historically, most actor GCs have used a *top-down* approach, driven by the garbage collector. In top-down approaches, the garbage collector computes a consistent global snapshot [21] by collecting snapshots of each actor's local state. To ensure consistency (or approximate consistency) between the local snapshots, actors synchronize with one another using memory barriers [13] or message-passing protocols [12]. These synchronization mechanisms can be expensive and tend to interact poorly with distributed failures, such as dropped messages or crashed nodes.

This thesis develops an alternative, called the *collage-based* approach, which builds on the actor GC introduced by Clebsch and Drossopoulou [10]. In the collage-based approach, actors can send local snapshots to the garbage collector at any time. The local snapshots that a garbage collector has received so far form a *collage*, which is not necessarily consistent or global. The key idea of collage-based approaches is for the garbage collector to identify *subsets* of the collage that are consistent and to identify garbage actors within those subsets. The thesis improves on the Clebsch-Drossopoulou collector by removing the need for causal message delivery and centralized GC, and by recovering from crashed node and dropped message failures.

Chapter 2 gives an overview of the actor model and actor garbage; in particular, the chapter introduces *quiescent* garbage, which is the focus of this thesis. Chapter 3 surveys prior work in actor garbage collection. Part I shows how the collage-based approach can be combined with reference listing to collect quiescent garbage without relying on shared memory or causal delivery guarantees. Part II shows how a simplified version of the previous part can be developed into *fault-recovering* actor GC, and evaluates an implementation in Akka; eager readers with a little knowledge about actors can jump directly to Part II.

2

The Actor Model

2.1 Actors

Actors [3, 22] are sequential processes that communicate by message-passing. Each actor executes a Turing-complete script called its *behavior*, which dictates what actions to perform when a message is received. These actions include:

- Spawning new actors;
- Asynchronously sending messages to other actors;
- Performing local computation;
- Performing *effects*, such as file I/O; and
- Changing the actor's behavior.

Notably, actors do not use locks or shared memory; all communication and coordination is done via asynchronous message-passing.[1]

Figure 2.1 depicts a trivial actor program, using syntax based on the Akka actor framework [26]. When the program is started, a `PingActor` actor is spawned (line 23) and sent a `Start` message containing the integer 100. When the `PingActor` receives the message (line 5), it spawns a `PongActor` and thereby obtains a reference to that actor. It uses the reference to send `PongActor` a `Ping` message (line 8), containing a reference to the `PingActor` itself. When the `PongActor` receives the message (line 17), it prints a message to the console and sends a reply back to the `PingActor`. The actors proceed to send messages back and forth for 99 more iterations.

A detailed exposition of actors can be found in [3], with a formalization in [25]. For our purposes, there are three important properties to understand:

The shared-nothing principle Actors do not (observably) share state. This property ensures that two concurrently executing actors cannot interfere with one another and only communicate

[1] For performance reasons, programmers often *do* implement behaviors that use shared memory [23, 24]. These programs can still be considered "faithful" to the actor model, as long as the behaviors are *observationally equivalent* [25] to behaviors that do not use such features.

```
class PingActor extends Actor:
  var pingsLeft: Int
  var pong: ActorRef
  def receive =
    case Start(count) =>
      pingsRemaining = count
      pong = spawn(PongActor())
      pong ! Ping(self)
      pingsLeft = pingsLeft - 1
    case Pong() =>
      println("PING!")
      if pingsRemaining >= 0 then
        pong ! Ping(self)
        pingsLeft = pingsLeft - 1
```

```
class PongActor extends Actor:
  def receive =
    case Ping(replyTo) =>
      println("PONG!")
      replyTo ! Pong()

// The system starts here
def main =
  val ping = spawn(PingActor())
  ping ! Start(100)
```

Figure 2.1: Akka-style pseudocode for a pair of actors sending messages back and forth. The "bang" operator `a ! m` is used to asynchronously send message `m` to the actor that has address `a`.

with one another via message-passing. It also ensures that actors interact with local actors (actors residing on the same node) the same way they do with remote actors (those residing on a different node).

The event-driven principle At any time, an actor is either *busy* (performing actions to process a message) or *idle* (waiting for a message). As long as an actor is idle, it does not have any effect on the rest of the system.

Memory-safety For actor a to send a message to actor b, the sender must have a *reference* (or *mail address*) to the recipient. Such a reference could only have been obtained in one of two ways: either a spawned b, or a received a reference to b in a message from some other actor. References cannot be produced "from thin air"; this is analogous to the notion of memory-safety in programming languages.

2.2 Actor Garbage

Garbage in the actor model (i.e. *actor garbage*) is quite different from garbage in shared memory concurrency models (i.e. *object garbage*). In shared memory models, an object is garbage if it can never be accessed by a thread. Object GCs can therefore detect object garbage by first "marking" all the objects that are reachable by any thread, then reclaiming all the objects that were not reachable [27, 28]. But this is not sufficient for actor garbage. Each actor is, conceptually, its own thread of control. Thus, an actor can still send messages and perform effects, regardless of whether it is reachable from any other actor, as long as it has undelivered messages left to

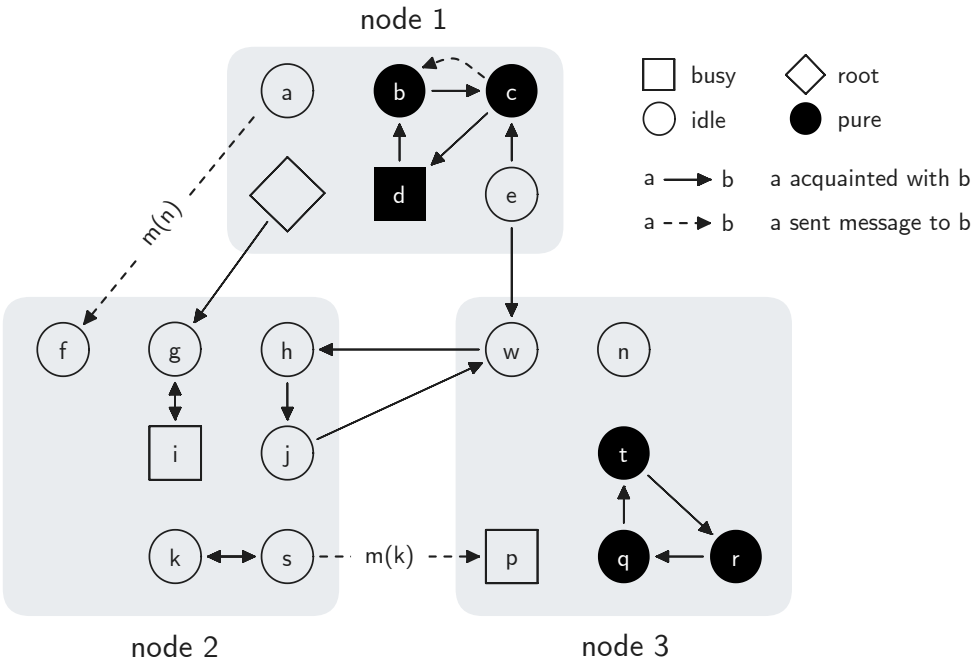


Figure 2.2: A global snapshot of an abstract actor system.

process.

To make a precise definition of actor garbage, we need the following terminology:

Configurations A *configuration* κ describes the global state (i.e. a consistent cut [21]) of an actor system during execution. A configuration consists of a set of nodes, a set of actors, and a set of undelivered messages. Each actor is located at a particular node. Each message consists of a *payload* (the contents of the message) and a *recipient* (the address of the recipient actor). Figure 2.2 depicts an example configuration.

Unblocked actors An actor a is *unblocked* in κ if it is busy (i.e. processing a message) or there is an undelivered message to a in κ. In Figure 2.2, actors b, d, f, p, q, r are unblocked and the rest are blocked.

Pure actors An actor is *pure* if it never performs effects (such as file I/O) and only spawns other pure actors. Pure actors can only affect the world by sending messages to *impure* actors. In Figure 2.2, actors b, c, d, q, r, t are pure and the rest are impure.

Root actors A root actor is one that must never be garbage collected, for example because it responds directly to user input. In Figure 2.2, root actors are denoted as triangles.

Potential acquaintances We say that *a potentially has a reference to b* in κ if either:

- a has a reference to b in κ; or
- a has an undelivered message in κ that contains a reference to b.

If a potentially has a reference to b, then we say b is a *potential acquaintance* of a and a is a *potential inverse acquaintance* of b. Figure 2.2 has many examples of potential acquaintances. In particular, k and s are potential acquaintances of one another; f potentially has a reference to n; and p potentially has a reference to k.

Potential reachability Actor a can *potentially reach* actor c in κ if either:

- a potentially has a reference to c in κ, or
- a potentially has a reference to some actor b that can potentially reach c in κ.

In Figure 2.2, e can potentially reach b, c, d, h, j, and w. Also notice that p can potentially reach both k and s.

Whereas *object garbage* is defined in terms of reachability from a thread, *actor garbage* is defined in terms of potential reachability by certain kinds of actors. We can define three main kinds of actor garbage below.

Acyclic garbage A non-root actor is *acyclic garbage* if it is blocked and has no potential inverse acquaintances. These actors are doomed to remain idle forever, due to the actor properties in Section 2.1:

1. Since actors are memory-safe, acyclic garbage actors can never become unblocked (no actor can ever send it a message).
2. Since actors are message-driven, acyclic garbage actors will remain idle forever and never affect the system (idle actors cannot spontaneously become busy).

In Figure 2.2, actors a and e are acyclic garbage but f and n are not. This is because f has an undelivered message and f will have a reference to n once that message is delivered.

Quiescent garbage A non-root actor is *quiescent* if it is blocked and only potentially reachable by other quiescent actors. This generalizes the definition of acyclic garbage to include *cyclic garbage* such as e, h, j, w in Figure 2.2. In quiescent garbage, each actor is permanently idle because no other actor can ever send it a message. Note that actors k, s are not quiescent garbage in Figure 2.2 (even though they are both blocked) because there is an unblocked actor p that potentially has a reference to k. Quiescent garbage arises when actors form cyclic data structures such as rings, doubly-linked lists, and supervision hierarchies [29].

Disconnected garbage A non-root actor is *disconnected garbage* if it is (1) pure and (2) can only potentially reach or be potentially reached by other disconnected actors. In Figure 2.2, actors q, t, r are disconnected garbage. Notice that b, c, d are not disconnected garbage because the simple garbage actor e has a reference to them. However, they will become disconnected garbage if e is garbage collected. Disconnected garbage can arise from a ring of pure actors that periodically send heartbeat messages to one another for fault-tolerance; the actors are never blocked, but they will never perform any useful work.

Actor garbage evidently exhibits more structure than ordinary object garbage. The story becomes even more interesting when we incorporate faults into the model. Briefly: notice that in Figure 2.2, the actors k, s are not quiescent garbage, but they would become garbage if message $m(k)$ is dropped by the network. Likewise, if node 1 crashes, then the non-garbage actors g, i both become quiescent garbage. No existing actor GC is capable of detecting g, i as garbage after a node failure.

The challenge of distributed actor GC is three-fold:

1. Obtaining a consistent view of the nodes without pausing them;
2. Obtaining a consistent view of the network without too many control messages; and
3. Minimizing the amount of data that nodes need to exchange to detect distributed garbage.

Chapter 3 describes prior strategies to achieve these goals, which use techniques derived from distributed snapshot algorithms [30] and shared memory garbage collection [28].

3

Related Work

Tables 3.1 to 3.3 list the major approaches to actor GC, along with their relative strengths and weaknesses. These approaches can be organized in the following way:

1. *Acyclic GCs:* Simple approaches that only detect acyclic garbage.
2. *Cyclic GCs:* Approaches that can collect cyclic garbage. They can be grouped according to how they ensure the garbage collector's view of the system is consistent:
   (a) *Snapshot-based GCs:* Approaches that require a consistent global snapshot of the distributed system before garbage can be collected.
   (b) *Trace-based GCs:* Approaches that use memory barriers [27, 28, 34] to overapproximate the set of live actors on a node.
   (c) *Collage-based GCs:* Approaches that record actor-local snapshots without coordination and subsequently identify which of the snapshots are consistent with one another.

It should be noted that all these GCs were implemented in incomparable actor frameworks, often with limited empirical evaluation, and several of the implementations have been lost to history. This unfortunately makes it impossible to rigorously compare the performance of the different approaches.

3.1 Fault Tolerance and Fault Recovery

A central theme in this thesis is how to collect distributed garbage despite unpredictable delays (slow nodes and network partitions) and faults (crashed nodes and dropped messages).

	Message order	Fault-tolerant?	Message-loss recovery?	Crashed-node recovery?
Reference counting [31, 32]	FIFO	✓	✗	✗
Reference listing [18, 33]	unordered	✓	✗	✗

Table 3.1: Comparison of acylic actor GCs.

	Proof?	Barrier-free?	Fault-tolerant?	Dropped message recovery?	Crashed-node recovery?
Venkatasubramanian et al [12]	✓	✓	✗	✗	✗
Puaut [14]	✗	✗	partly	✗	✗
Kafura et al [15]	✗	✗	✗	✗	✗
Vardhan-Agha [17]	✓	✗	✓	✗	✗

Table 3.2: Comparison of actor GCs that can collect both quiescent and disconnected garbage. All assume FIFO message delivery.

	Message order	Proof?	Barrier-free?	Fault-tolerant?	Dropped message recovery?	Crashed-node recovery?
Kamada et al [13]	none	✗	✗	✗	✗	✗
Wang-Varela [18]	none	✗	✗	✓	✗	✗
Clebsch-Drossopoulou [10]	causal	✓	✓	✓	✗	✗
PRL (Part I)	none	✓	✓	✓	✗	✗
CRGC (Part II)	none	✓	✓	✓	✓	✓

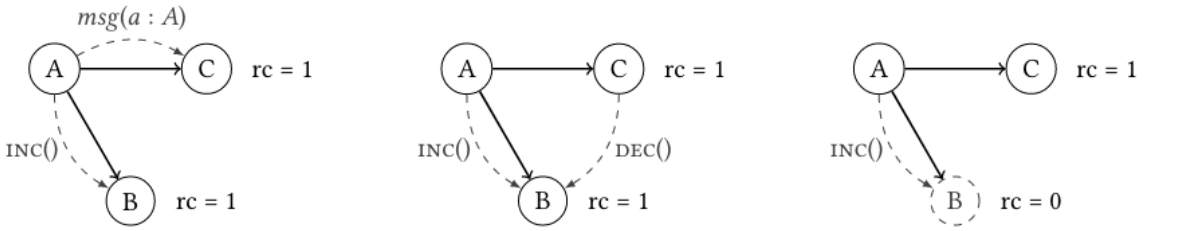
Table 3.3: Comparison of approaches that collect quiescent garbage.

The term "fault-tolerant" is used inconsistently in literature on distributed garbage collection [18, 35, 36, 37]. Some authors use it to mean that some garbage can be detected despite faults. Others use it to mean a stronger property, where actors can be garbage collected if they are only potentially reachable by actors on crashed nodes. We propose the following distinction:

> An actor GC is *fault-tolerant* if it can collect some distributed garbage despite faults. Dropped messages and crashed nodes may cause certain actors to never be garbage collected.
>
> An actor GC is *fault-recovering* if faults can cause actors to become garbage and be garbage collected. For example, actors that are never garbage in non-faulty executions may become garbage because all their potential inverse acquaintances have failed.

In this terminology, reference counting and reference listing are fault-tolerant because an actor can be garbage collected if all its past inverse acquaintances are on non-faulty nodes. In order to make such a GC fault-*recovering*, the runtime must detect and deactivate references held by faulty nodes [36].

Several actor GCs that can detect cyclic garbage are not fault-tolerant [12, 13, 15]. These GCs require (approximate) snapshots of every node in the system. If a node N does not take a local



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

Figure 3.1: Example of a race condition in naïve reference counting. Circles represent actors and "rc" is their corresponding reference count. Solid arrows represent existing references, while dashed grey arrows represent sent messages.

snapshot, the GC cannot rule out the possibility that N has references to every actor on every other node.

Several actor GCs that are not fault-tolerant can still collect *local* garbage despite any number of faulty nodes [13, 15]. These GCs do so by computing node-local approximate snapshots that determine whether an actor's address could leaked to an outside node. Actors that are only potentially reachable by local actors can therefore be garbage collected despite any number of failures. We do not consider such a GC to be fault-tolerant because any GC can be trivially modified to detect all local garbage despite failures.

Part I of this thesis introduces the first fault-tolerant cyclic actor GC that does not require message ordering, special actor topologies, or memory barriers. Part II builds on the previous part to make the first fault-*recovering* cyclic actor GC.

3.2 Acyclic GCs

Acyclic GCs are based on one of two related concepts: *reference counting* and *reference listing* [37]. Both approaches allow actors to garbage collect themselves when they detect that no other actor has a reference to them. Reference counting approaches achieve this by storing an integer at each actor, which is asynchronously incremented or decremented when new references are created or destroyed. In contrast, reference listing approaches achieve the same effect by maintaining a *list* of actors that have references.

A naive reference counting approach requires causal message delivery to prevent race conditions (see Figure 3.1). Optimizations such as weighted reference counting [31, 32] and indirect reference counting [38] eliminate the need for "increment" messages and can thereby remove the causal order requirement.

Reference listing imposes higher memory overhead than reference counting, but can be easier

to integrate with fault-tolerance and fault-recovery protocols. Birrell's reference listing [33, 39] requires a blocking step, preventing actors from using references until they have notified the target about their reference. Wang and Varela's reference listing [18] removes this restriction, but requires a significant number of acknowledgment messages: adding Wang and Varela's reference listing to an existing system adds 2–5 times as many messages—and even more when messages contain multiple references.

One benefit of reference tracking is that actors can be collected almost as soon as they become garbage. However, this only applies if "decrement" messages are sent as soon as their corresponding references are no longer needed. Certain languages, such as Rust and Pony, can ensure that references are garbage collected promptly and thereby trigger an automatic "decrement" message. Runtimes with tracing garbage collectors (such as the JVM or BEAM) do not provide this guarantee.

Another benefit of reference tracking is that it tolerates some faults; the only actors that can prevent an actor a from being garbage collected are the actors that actually hold references to a. However, dropped "decrement" messages can permanently prevent an actor from being collected, even though no other actor has a reference to it [37].

3.3 Cyclic GCs

To collect a cycle of actor garbage, actor GCs must first inspect the local state of each actor in the cycle. Conceptually, we can understand this as the GC computing a *snapshot* of each actor in the cycle, and combining the snapshots into a *collage*. The GC then inspects the collage to determine whether the actors are indeed garbage.

3.3.1 Snapshot-Based GCs

In snapshot-based GCs, the garbage collector ensures that the collage is consistent [21] or approximately consistent (thus forming a "global snapshot") so that live actors are not misdiagnosed as garbage. Venkatasubramanian et al introduced the first distributed actor GC that does not require pauses [12, 40, 41]. The GC requires actors to be arranged in a hierarchy of clusters, through which all messages are routed. The special properties of this hierarchy make it possible to compute a Chandy-Lamport-like global snapshot at the actor-level (rather than the node-level as in [15]). After computing the global snapshot, each actor would obtain a list of all its inverse acquaintances at the time of the snapshot. The actors would then perform a decentralized mark-and-sweep phase, passing messages to their acquaintances and inverse acquaintances so as to mark all the non-garbage actors. The primary limitations of this GC are the restrictive assump-

tions about actor hierarchies and the large number of messages that need to be exchanged.

In Chapter 2 we established that actor garbage is different from object garbage, so one cannot use "reachability from the root actors" to detect actor garbage. An alternative approach is to maintain a set of extra edges and nodes in the actor reference graph, making it so that actor garbage coincides with regular garbage; we call these *actor-to-object* GCs. Actor-to-object GCs were first proposed by Vardhan and Agha [17, 42]. Independently, Dickman sketched a similar approach but did not implement it or prove its correctness [16]. Subsequently, Wang et al presented an optimization that required fewer additional edges and no additional objects; this experimentally led to shorter graph traversal times [43].

One might assume that this approach would allow an actor framework, implemented on a garbage-collected runtime such as the JVM, to repurpose the existing object GC as an actor GC. This is not necessarily the case. Vardhan's original implementation was unable to use the JVM GC for actors because the ActorFoundry framework implemented actors as threads, which do not fall under the purview of the JVM GC [42]. Thus, in practice, distributed actor-to-object GCs still require a complete implementation of a distributed object GC [44].

Interestingly, it is possible to design a *non-distributed* actor framework that exploits the underlying runtime's garbage collector. Desell and Varela designed the SALSA Lite actor framework in such a way that *quiescent* actor garbage coincides with object garbage [45]. Unfortunately, it is unclear how their approach would generalize to distributed systems.

A fundamental limitation of actor-to-object GCs is that they require a collage to be *a priori* consistent in order to determine which actors are blocked and what their potential inverse acquaintances are. Computing such a snapshot can introduce high overhead, high pause times, and brittleness to crash faults.

3.3.2 Trace-Based GCs

Trace-based GCs are garbage collectors designed for use with the actor model that use techniques from concurrent tracing garbage collection [27, 28, 34]. All of the past approaches are *mark-and-sweep* GCs, detecting actor garbage by first *marking* the actors that might not be garbage and then collecting unmarked actors.

Kamada et al implemented an actor GC that only detects quiescent garbage [13]. Each node needs to pause actor execution to mark all the unblocked actors and root actors. The actors are then unpaused and execute concurrently while the GC marks them. Write barriers [27, 28, 34] are used to ensure that all non-garbage actors get marked. Whenever the GC encounters a reference to an external actor, it sends a "mark" message to the remote node, causing the external actor to become marked. Once all local marking has completed and all "mark" messages have

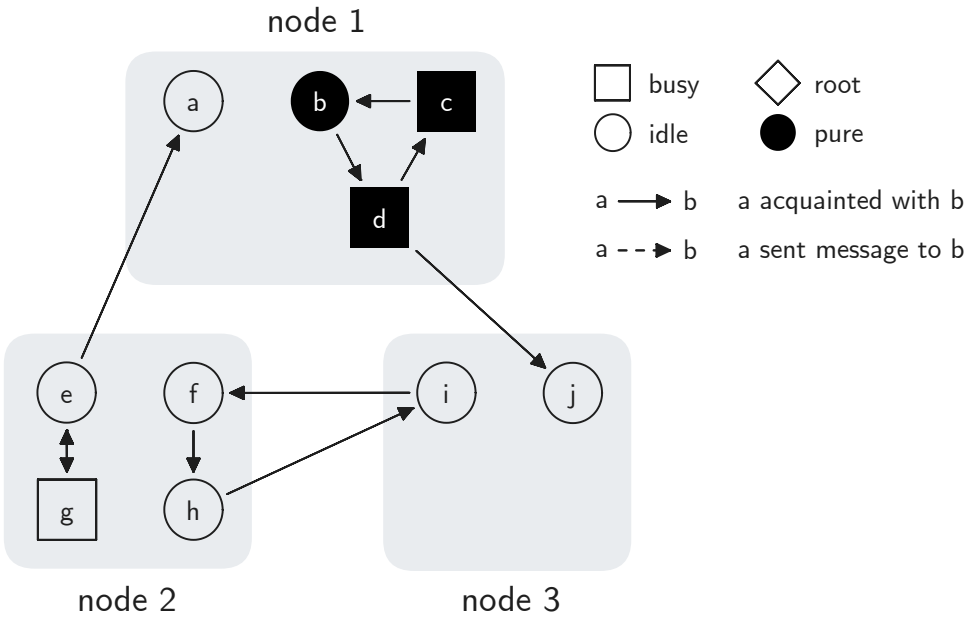


Figure 3.2: A snapshot of an actor system where failures may prevent distributed garbage collection.

been delivered, only the garbage actors will remain unmarked. A global termination detection algorithm [46] is used to detect that all "mark" messages have been delivered.

A fundamental limitation of Kamada et al's approach is that distributed GC requires cooperation from every node in the cluster. In Figure 3.2, f, h, i are garbage but they could not be collected without information from node 1, because some unblocked actor on node 1 might have a reference to one of those actors. Consequently, one slow node can prevent all distributed garbage from being collected.

Puaut proposed an algorithm with three parts [14]. First, nodes compute an approximate snapshot of their local actors using an incremental tracing technique. Second, in order to check that node-local snapshots are consistent with one another, each node maintains a vector clock and associates its local snapshot with a vector timestamp. Third, in order to compute a snapshot of the references *in transit* between nodes, each node tracks the set of references it has sent and received from the network. All this information is sent to a centralized GC process, which can detect distributed garbage only if the node-local snapshots happen to be consistent with one another. A key insight in Puaut's approach is that, when the network is FIFO, nodes can recover from message loss by examining the references they have sent/received and inferring which references were dropped.

Kafura et al proposed a similar algorithm to that of Puaut, but removed the centralized GC process and the need for vector timestamps [15]. They proposed using the Chandy-Lamport

algorithm to coordinate when nodes should compute their local snapshots and also to determine the state of the network. Nodes would then perform a decentralized worklist algorithm to mark all non-garbage actors in the cluster. This approach shares the limitation of Kamada et al, in which one slow node can prevent all distributed GC.

Wang and Varela proposed the first truly *fault-tolerant* object-to-actor GC, capable of detecting distributed actor garbage despite slow nodes [18, 47]. They achieved this by associating each actor with a *reference listing*; in Figure 3.2, this would allow nodes 2 and 3 to determine that f, h, i are not referenced by any actor in node 1, making it safe to garbage collect those actors. However, their reference listing scheme requires large numbers of acknowledgment messages for each application message, leading to high performance overhead.

3.3.3 *Collage-based GCs*

In collage-based GCs, actors record local snapshots at arbitrary times and it is left up to the garbage collector to determine which local snapshots are consistent with one another. The advantage of this approach is that it allows implementors to use heuristics to decide when (and how frequently) an actor should take a snapshot. Collage-based GCs typically do not assume that every actor has recorded a snapshot, so these approaches are naturally fault-tolerant.

Puaut's GC [14] can be seen as a collage-based GC in which *nodes* record local snapshots, rather than actors. However, it is not necessarily true that these node snapshots will ever form a consistent collage.

The first truly collage-based actor GC, *MAC*, was introduced by Clebsch and Drossopoulou [10]. In MAC, actors take snapshots when their mailbox is empty and send those snapshots to a centralized cyclic garbage collector. The garbage collector uses a simple request-reply protocol to identify sets of snapshots that are both consistent quiescent. This protocol appears to be a version of the Alagar-Venkatesan global snapshot protocol [48], generalized from a fixed set of processes to a dynamic set of actors. The main limitation of the Clebsch-Drossopoulou GC is its reliance on causal delivery between actors. Causal delivery in a distributed system imposes additional costs: the well-known strategy of using vector clocks requires $O(n)$ additional memory for each message, where n is the number of actors [49]. A more recent proposal, in which nodes must be arranged in a tree topology, does not impose this overhead but still prevents point-to-point communication between nodes [50]. Also, the centralized cycle detector is both a bottleneck and a single point of failure.

The actor GCs presented in Parts I and II build on MAC in two ways. Part I removes the causal message delivery and centralization requirements of MAC by adding data to each actor's local state and by using a novel form of reference listing. Part II takes a different approach, removing

the need for reference listing and making the garbage collector fault-recovering.

Part I

Coordination-Free Actor GC

Part I presents *PRL*: an actor GC based on *Proactive Reference Listing*. The primary advantage of PRL is that it is decentralized and naturally fault-tolerant: local garbage can be collected in a subsystem without communicating with the rest of the system. Systems can also cooperate to detect distributed garbage by exchanging minimal amounts of information. Garbage collection can be performed concurrently with the application, requires no locks or memory barriers, and imposes no message ordering constraints.

PRL works as follows. Actors implement the *PRL communication protocol* (Chapter 4) which maintains information about references and message counts in each actor's state. When an actor no longer needs a reference, it sends a message to the target actor to release the reference; this allows actors to detect when they become acyclic garbage. To detect cyclic quiescent garbage, each actor periodically sends a snapshot of its garbage collection information to a *garbage collector* actor. A garbage collector combines these snapshots into a *collage* and periodically searches the collage for actors that appear quiescent (Chapter 5). Once a garbage collector suspects that an actor is quiescent, it asks the actor to terminate. We prove that non-quiescent actors will never be garbage collected (Corollary 5.1) and, if every quiescent actor eventually sends a snapshot to the garbage collector, then all quiescent actors will eventually be collected (Theorem 5.3). We also show how a team of garbage collectors can detect cooperatively detect distributed garbage with little communication (Section 5.3).

Since PRL is defined in terms of the actor model, it is oblivious to details of a particular implementation (such as how sequential computations are represented or where actors are located). Our technique is therefore applicable to different actor frameworks; in particular, it may be implemented as a library. Moreover, it can also be applied to open systems, allowing an actor system using PRL to interoperate with a manually-collected actor system.

4

Proactive Reference Listing

This chapter presents the PRL communication protocol. We begin by motivating what kind of information a garbage collector would need to detect quiescent actors. Next we present the communication protocol more formally, using a two-level semantic model [51]. In this model, a *system-level* transition system interprets the operations performed by a user-facing *application-level* transition system. The application level defines the abstract operational semantics of the actor system from the user's perspective, including location transparency and fairness assumptions. Since PRL preserves this semantics, we leave out the application-level system; for a formalization, see [25]. The system-level transition system defines each actor's system-level state and what additional actions should be performed when the application level tried to do an operation, such as sending a message or spawning an actor. In the case of PRL, these operations sometimes cause additional system-level messages to be sent or for metadata to be added to an application-level message.

4.1 Overview

Ordinary actor systems allow an actor a to send a message to actor b if a has b's address. In PRL, actors must use *reference objects* (abbreviated *refobs*) instead; refobs combine a plain actor address (the address of the *target*) with additional metadata, such as the address of the refob's designated *owner*. A refob can only be used by its owner: in order for a to give b a reference to c, it explicitly creates a new refob owned by b. Once a refob is no longer needed, it is *deactivated* by its owner and removed from the owner's local state. These operations could be done manually at the application level or handled automatically in the runtime via a suitable API.

The PRL communication protocol enriches each actor's state with a list of refobs that it currently owns and associated message counts representing the number of messages sent using each refob. Each actor also maintains a subset of the refobs of which it is the target, together with associated message receive counts. Lastly, actors perform a form of "contact tracing" by maintaining

a subset of the refobs that they have created for other actors; we provide details about the bookkeeping later in this section.

The additional information above allows us to detect quiescence by inspecting actor snapshots. If a collage is consistent (in the sense of [21]) then we can use the "contact tracing" information to determine whether the set is *closed* under the potential inverse acquaintance relation (see Section 4.5). Then, given a consistent and closed collage, we can use the message counts to determine whether an actor is blocked. We can therefore find all the quiescent actors within a consistent collage.

In fact, PRL satisfies a stronger property: any collage that "appears quiescent" in the sense above is guaranteed to be consistent. Hence, given an arbitrary closed collage, it is possible to determine which of the corresponding actors have quiescent. This allows a great deal of freedom in how snapshots are collected. For instance, each actor could set its own recurring timeout for when to take the next snapshot. The duration of this timeout could be changed based on runtime information, such as how long the actor has been alive; this would amount to a *generational* approach to actor garbage collection [52].

4.1.1 Reference Objects

A refob is a triple (x, a, b), where a is the owner actor's address, b is the target actor's address, and x is a globally unique token. An actor can cheaply generate such a token by combining its address with a local sequence number, since actor systems already guarantee that each address is unique. We will stylize a triple (x, a, b) as $x : a \multimap b$. We will also sometimes refer to such a refob as simply x, since tokens act as unique identifiers.

When an actor a spawns an actor b (Figure 4.1 (1, 2)) the PRL protocol creates a new refob $x : a \multimap b$ that is stored in both a and b's system-level state, and a refob $w : b \multimap b$ in b's state. The refob x allows a to send application-level messages to b. These messages are denoted $\texttt{app}(x, R)$, where R is the set of refobs contained in the message that a has created for b. The refob y corresponds to the `self` variable present in some actor languages.

If a has active refobs $x : a \multimap b$ and $y : a \multimap c$, then it can create a new refob $z : b \multimap c$ by generating a token z. In addition to being sent to b, this refob must also temporarily be stored in a's system-level state and marked as "created using y" (Figure 4.1 (3)). Once b receives z, it must add the refob to its system-level state and mark it as "active" (Figure 4.1 (4)). Note that b can have multiple distinct refobs that reference the same actor in its state; this can be the result of, for example, several actors concurrently sending refobs to b. Transition rules for spawning actors and sending messages are given in Section 4.2.2.

Actor a may remove z from its state once it has sent a (system-level) `info` message informing

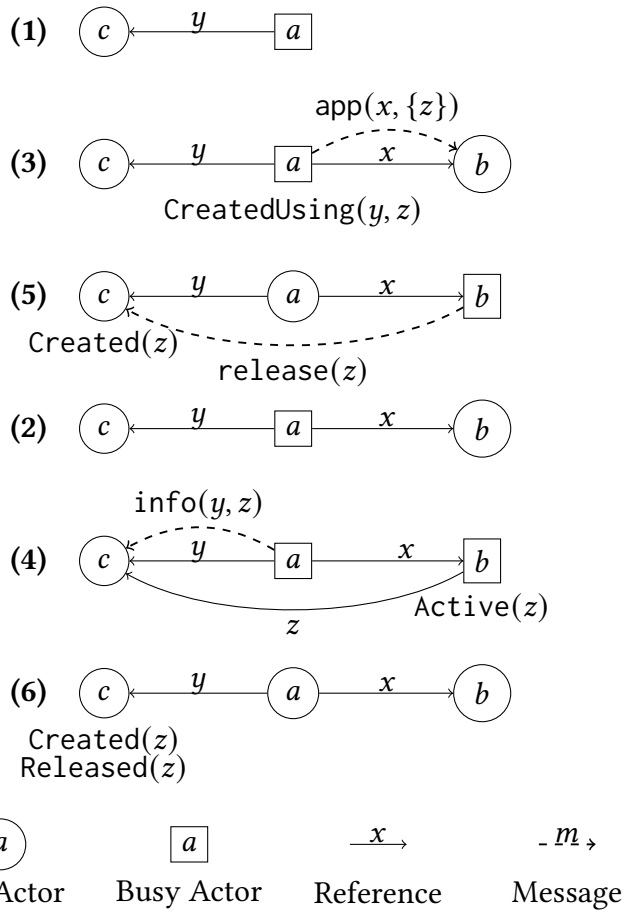


Figure 4.1: An example showing how refobs are created and destroyed. Below each actor we list all the "facts" related to z that are stored in its local state. Although not pictured in the figure, a also obtains facts $\texttt{Active}(x)$ and $\texttt{Active}(y)$ after spawning actors b and c, respectively. Likewise, actors b, c obtain facts $\texttt{Created}(x), \texttt{Created}(y)$, respectively, upon being spawned.

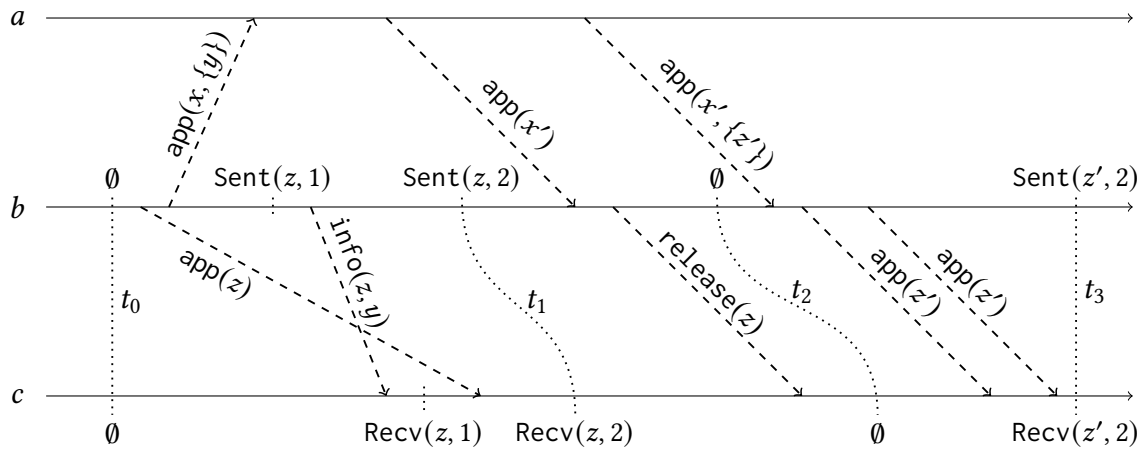


Figure 4.2: An event diagram for actors a, b, c, illustrating message counts and consistent snapshots. Dashed arrows represent messages and dotted lines represent mutually quiescent cuts. For a cut to be mutually quiescent, it is necessary (but not sufficient) that the message send and receive counts agree for all participants.

c about z (Figure 4.1 (4)). Similarly, when b no longer needs its refob for c, it can "deactivate" z by removing it from local state and sending c a (system-level) `release` message (Figure 4.1 (5)). Note that if b already has a refob $z : b \multimap c$ and then receives another $z' : b \multimap c$, then it can be more efficient to defer deactivating the extraneous z' until z is also no longer needed; this way, the `release` messages can be batched together.

When c receives an `info` message, it records that the refob has been created, and when c receives a `release` message, it records that the refob has been released (Figure 4.1 (6)). Note that these messages may arrive in any order. Once c has received both, it is permitted to remove all facts about the refob from its local state. Transition rules for these reference listing actions are given in Section 4.2.3.

Once a refob has been created, it cycles through four states: pending, active, inactive, or released. A refob $z : b \multimap c$ is said to be *pending* until it is received by its owner b. Once received, the refob is *active* until it is *deactivated* by its owner, at which point it becomes *inactive*. Finally, once c learns that z has been deactivated, the refob is said to be *released*. A refob that has not yet been released is *unreleased*.

Slightly amending the definition we gave in Chapter 2, we say that b is a *potential acquaintance* of a (and a is a *potential inverse acquaintance* of b) when there exists an unreleased refob $x : a \multimap b$. Thus, b becomes a potential acquaintance of a as soon as x is created, and only ceases to be an acquaintance once it has received a `release` message for every refob $y : a \multimap b$ that has been created so far.

4.1.2 Message Counts and Snapshots

For each refob $x : a \multimap b$, the owner a maintains a count of how many `app` and `info` messages have been sent along x; this count can be deleted when a deactivates x. Each message is annotated with the refob used to send it. Whenever b receives an `app` or `info` message along x, it correspondingly increments a receive count for x; this count can be deleted once x has been released. Thus the memory overhead of message counts is linear in the number of unreleased refobs. Figure 4.2 gives an example.

A snapshot is a copy of all the facts in an actor's system-level state at some point in time. We will assume throughout the paper that in every collage Q, each snapshot was taken by a different actor; such a set is also said to form a *cut*. Recall that a collage Q is *consistent* if no snapshot in Q causally precedes any other [21]; it is as if all the actors in Q took their snapshots simultaneously. Let us also say that Q is *mutually quiescent* if for all actors a, b in Q, all messages sent from a to b before a's snapshot were also received before b's snapshot. Notice that mutual quiescence is just a special case of consistency, in which all messages sent by actors in the cut to actors in the cut have been delivered. Moreover, if each actor in Q is idle and Q contains each actor's potential inverse acquaintances, then Q corresponds to a quiescent set of actors: every actor in Q is blocked and only potentially reachable by other blocked actors in Q.

One might therefore hope to check whether Q is mutually quiescent by simply comparing the message send and receive counts of all snapshots in Q. Clearly, if Q is mutually quiescent, then the participants' send and receive counts will agree for each $x : a \multimap b$ where $a, b \in Q$. However, the converse may be false for two reasons: out of order delivery of messages, and temporarily null message counts.

1. *Out of order delivery:* In Figure 4.2, a snapshot from b when $\texttt{Sent}(z, 1)$ is in its knowledge set would not be consistent with a snapshot from c when $\texttt{Received}(z, 1)$ is in its knowledge set. This is because the message $\texttt{info}(z, y)$ is sent after b's snapshot and received before c's snapshot. To guarantee this situation does not occur, we must be able to prove that b does not send any messages along z in the interval between b's snapshot and c's snapshot. In particular, this holds when c's snapshot happens before b's snapshot.
2. *Null message count:* Based on the available information, c's snapshot at t_0 in Figure 4.2 appears mutually quiescent with b's snapshot at t_2 and c's snapshot at t_2 appears mutually quiescent with b's snapshot at t_0—despite neither of these pairs being truly mutually quiescent. The problem is that when b's send count for $z : b \multimap c$ is null, it could be because b has not yet received z or because b has already deactivated z. Likewise, c's receive count could be null because it has not yet received any messages along z or because z has already been released.

To distinguish these scenarios, we incorporate the snapshots of c's other potential inverse acquaintances—such as a—into the snapshot set Q. In Section 4.5 we identify a distributed property called the *Chain Lemma* that must hold in any consistent collage closed under the potential inverse acquaintance relation. We show, in Chapter 5, that combining the Chain Lemma with message counts is sufficient to determine whether a collage is mutually quiescent.

4.2 Model

We use the letters a, b, c, d, e to denote actor addresses. Tokens are denoted x, y, z, with a special reserved token `null` for messages from external actors.

A *fact* is a value that takes one of the following forms: $\texttt{Created}(x)$, $\texttt{Released}(x)$, $\texttt{Active}(x)$, $\texttt{Unreleased}(x)$, $\texttt{CreatedUsing}(x, y)$, $\texttt{Sent}(x, n)$, or $\texttt{Received}(x, n)$ for some refobs x, y and natural number n. Each actor's state holds a set of facts about refobs and message counts called its *knowledge set*. We use ϕ, ψ to denote facts and Φ, Ψ to denote finite sets of facts. Each fact may be interpreted as a *predicate* that indicates the occurrence of some past event. Interpreting a set of facts Φ as a set of axioms, we write $\Phi \vdash \phi$ when ϕ is derivable by first-order logic from Φ with the following additional rules:

- If $(\nexists n \in \mathbb{N},\ \texttt{Sent}(x, n) \in \Phi)$ then $\Phi \vdash \texttt{Sent}(x, 0)$
- If $(\nexists n \in \mathbb{N},\ \texttt{Received}(x, n) \in \Phi)$ then $\Phi \vdash \texttt{Received}(x, 0)$
- If $\Phi \vdash \texttt{Created}(x) \wedge \neg\texttt{Released}(x)$ then $\Phi \vdash \texttt{Unreleased}(x)$
- If $\Phi \vdash \texttt{CreatedUsing}(x, y)$ then $\Phi \vdash \texttt{Created}(y)$

For convenience, we define a pair of functions $\mathit{incSent}(x, \Phi)$, $\mathit{incRecv}(x, \Phi)$ for incrementing message send/receive counts, as follows: If $\texttt{Sent}(x, n) \in \Phi$ for some n, then $\mathit{incSent}(x, \Phi) = (\Phi \setminus \{\texttt{Sent}(x, n)\}) \cup \{\texttt{Sent}(x, n + 1)\}$; otherwise, $\mathit{incSent}(x, \Phi) = \Phi \cup \{\texttt{Sent}(x, 1)\}$. Likewise for *incRecv* and `Received`.

Recall that an actor is either *busy* (processing a message) or *idle* (waiting for a message). An actor with knowledge set Φ is denoted $[\Phi]$ if it is busy and (Φ) if it is idle.

Our specification includes both *system messages* (also called *control messages*) and *application messages*. The former are automatically generated by the PRL protocol and handled at the system level, whereas the latter are explicitly created and consumed by user-defined behaviors. Application-level messages are denoted $\texttt{app}(x, R)$. The argument x is the refob used to send the message. The second argument R is a set of refobs created by the sender to be used by the destination actor. Any remaining application-specific data in the message is omitted in our notation.

The PRL communication protocol uses two kinds of system messages. $\texttt{info}(y, z, b)$ is a message sent from an actor a to an actor c, informing it that a new refob $z : b \multimap c$ was created using $y : a \multimap c$. $\texttt{release}(x, n)$ is a message sent from an actor a to an actor b, informing it that the

refob $x : a \multimap b$ has been deactivated and that a total of n messages have been sent along x.

A *configuration* $\langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$ is a quadruple $(\alpha, \mu, \rho, \chi)$ where: α is a mapping from actor addresses to knowledge sets; μ is a mapping from actor addresses to multisets of messages; and ρ, χ are sets of actor addresses. Actors in $\mathrm{dom}(\alpha)$ are *internal actors* and actors in χ are *external actors*; the two sets may not intersect. The mapping μ associates each actor with undelivered messages to that actor. Actors in ρ are *receptionists* [25]: an internal actor a becomes a receptionist when its address is exposed to an external actor. Subsequently, any external actor can obtain a's address and send it a message. We will ensure $\rho \subseteq \mathrm{dom}(\alpha)$ remains valid in any configuration that is derived from a configuration where the property holds (referred to as the locality laws in [53]).

Configurations are denoted by κ, κ', κ_0, etc. If an actor address a (resp. a token x), does not occur in κ, then the address (resp. the token) is said to be *fresh*. We assume a facility for generating fresh addresses and tokens.

In order to express our transition rules in a pattern-matching style, we will employ the following shorthand. Let $\alpha, [\Phi]_a$ refer to a mapping α' where $\alpha'(a) = [\Phi]$ and $\alpha = \alpha'|_{\mathrm{dom}(\alpha')\setminus\{a\}}$. Similarly, let $\mu, [a \triangleleft m]$ refer to a mapping μ' where $m \in \mu'(a)$ and $\mu = \mu'|_{\mathrm{dom}(\mu')\setminus\{a\}} \cup \{a \mapsto \mu'(a)\setminus\{m\}\}$. Informally, the expression $\alpha, [\Phi]_a$ refers to a set of actors containing both α and the busy actor a (with knowledge set Φ); the expression $\mu, [a \triangleleft m]$ refers to the set of messages containing both μ and the message m (sent to actor a).

The rules of our transition system define atomic transitions from one configuration to another. Each transition rule has a label l, parameterized by some variables $\vec{x}$ that occur in the left- and right-hand configurations. Given a configuration κ, these parameters functionally determine the next configuration κ'. Given arguments $\vec{v}$, we write $\kappa \xrightarrow{l(\vec{v})} \kappa'$ to denote a semantic step from κ to κ' using rule $l(\vec{v})$.

We refer to a label with arguments $l(\vec{v})$ as an *event*, denoted e. a sequence of events is denoted π. If $\pi = e_1, \ldots, e_n$ then we write $\kappa \xrightarrow{\pi} \kappa'$ when $\kappa \xrightarrow{e_1} \kappa_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} \kappa'$. If there exists π such that $\kappa \xrightarrow{\pi} \kappa'$, then κ' is *derivable* from κ. An *execution path* (also called a *computation path* [25]) is a sequence of events $e_1, \ldots, e_n$ such that $\kappa_0 \xrightarrow{e_1} \kappa_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} \kappa_n$, where κ_0 is the initial configuration (Section 4.2.1). We say that a property holds *at time t* if it holds in κ_t. We will also employ the shorthand that α_t is the actor configuration at time t, i.e. $\kappa_t = \langle\!\langle\ \alpha_t \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$.

Note that the PRL communication protocol does not require a notion of a unique global time. We could have given a more general specification using concurrent rewriting [54], in which potential execution paths are partial orders of events. A given execution path in such a specification can be mapped to an execution path in our system by mapping its partial order to a total order which respects the ordering specified in the partial order. We refer to "time" as an ordinal corresponding to an arbitrary total order that is consistent with a partial order in a system's execution

SPAWN(x, a, b)

$$\langle\!\langle\ \alpha, [\Phi]_a \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \longrightarrow \langle\!\langle\ \alpha, [\Phi \cup \{\texttt{Active}(x : a \multimap b)\}]_a, [\Psi]_b \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

where x, y, b fresh
and $\Psi = \{\texttt{Created}(x : a \multimap b),\ \texttt{Created}(y : b \multimap b),\ \texttt{Active}(y : b \multimap b)\}$

SEND$(x, \vec{y}, \vec{z}, a, b, \vec{c})$

$$\langle\!\langle\ \alpha, [\Phi]_a \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \longrightarrow \langle\!\langle\ \alpha, [\mathit{incSent}(x, \Phi) \cup \Psi]_a \mid \mu, [b \triangleleft \mathsf{app}(x, R)]\ \rangle\!\rangle^{\rho}_{\chi}$$

where $\vec{z}$ fresh and $n = |\vec{y}| = |\vec{z}| = |\vec{c}|$
and $\Phi \vdash \texttt{Active}(x : a \multimap b)$ and $\forall i \leq n,\ \Phi \vdash \texttt{Active}(y_i : a \multimap c_i)$
and $R = \{z_i : b \multimap c_i \mid i \leq n\}$ and $\Psi = \{\texttt{CreatedUsing}(y_i, z_i) \mid i \leq n\}$

RECEIVE(x, b, R)

$$\langle\!\langle\ \alpha, (\Phi)_b \mid \mu, [b \triangleleft \mathsf{app}(x, R)]\ \rangle\!\rangle^{\rho}_{\chi} \longrightarrow \langle\!\langle\ \alpha, [\mathit{incRecv}(x, \Phi) \cup \Psi]_b \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

where $\Psi = \{\texttt{Active}(z) \mid z \in R\}$

IDLE(a)

$$\langle\!\langle\ \alpha, [\Phi]_a \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \longrightarrow \langle\!\langle\ \alpha, (\Phi)_a \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

Figure 4.3: Rules for standard actor interactions.

path (see [22, 55]). This allows us to prove various properties by induction on time t instead of by more complicated means.

4.2.1 *Initial Configuration*

The initial configuration κ_0 consists of a single actor in a busy state: $\langle\!\langle\ [\Phi]_a \mid \emptyset\ \rangle\!\rangle^{\emptyset}_{\{e\}}$, where $\Phi = \{\texttt{Active}(x : a \multimap e),\ \texttt{Created}(y : a \multimap a),\ \texttt{Active}(y : a \multimap a)\}$. The actor's knowledge set includes a refob to itself and a refob to an external actor e. a can become a receptionist by sending e a refob to itself. Henceforth, we will only consider configurations that are derivable from an initial configuration.

4.2.2 *Standard Actor Operations*

Figure 4.3 gives transition rules for standard actor operations, such as spawning actors and sending messages. Each of these rules corresponds a rule in the standard operational semantics

of actors [25]. Note that each rule is atomic, but can just as well be implemented as a sequence of several smaller steps without loss of generality because actors do not share state—see [25] for a formal proof.

The Spawn event allows a busy actor a to spawn a new actor b and creates two refobs, $x : a \multimap b$ and $y : b \multimap b$. Actor b is initialized with knowledge about x and y via the facts $\texttt{Created}(x)$ and $\texttt{Created}(y)$. The facts $\texttt{Active}(x)$ and $\texttt{Active}(y)$ allow a and b to immediately begin sending messages to b. Note that implementing Spawn does not require a synchronization protocol between a and b to construct $x : a \multimap b$. The parent a can pass both its address and the freshly generated token x to the constructor for b. Since actors typically know their own addresses, this allows b to construct the triple (x, a, b). Since the `spawn` call typically returns the address of the spawned actor, a can also create the same triple.

The Send event allows a busy actor a to send an application-level message to b containing a set of refobs $z_1, \ldots, z_n$ to actors $\vec{c} = c_1, \ldots, c_n$. Note that it is possible that $b = a$ or $c_i = a$ for some i in this sequence—i.e., an actor may send itself a message, or it may send b its a refob containing its own address. For each new refob z_i, we say that the message *contains* z_i. Any other data in the message besides these refobs is irrelevant to quiescence detection and therefore omitted. To send the message, a must have active refobs to both the target actor b and to every actor $c_1, \ldots, c_n$ referenced in the message. For each target c_i, a adds a fact $\texttt{CreatedUsing}(y_i, z_i)$ to its knowledge set; we say that a *created* z_i *using* y_i. Finally, a must increment its `Sent` count for the refob x used to send the message; we say that the message is sent *along* x.

The Receive event allows an idle actor b to become busy by consuming an application message sent to b. Before performing subsequent actions, b increments the receive count for x and adds all refobs in the message to its knowledge set.

Finally, the Idle event puts a busy actor into the idle state, enabling it to consume another message.

4.2.3 Release Protocol

Whenever an actor creates or receives a refob, it adds facts to its knowledge set. To remove these facts when they are no longer needed, actors can perform the *release protocol* defined in Figure 4.4. All of these rules are not present in the standard operational semantics of actors.

The SendInfo event allows a busy actor a to inform c about a refob $z : b \multimap c$ that it created using y; we say that the `info` message is sent *along* y and *contains* z. This event allows a to remove the fact $\texttt{CreatedUsing}(y, z)$ from its knowledge set. It is crucial that a also increments its `Sent` count for y to indicate an undelivered `info` message sent to c: it allows the garbage collector to detect when there are undelivered `info` messages, which contain refobs. This message is

$\textsc{SendInfo}(y, z, a, b, c)$

$$\langle\!\langle\ \alpha, [\Phi \cup \Psi]_a \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \longrightarrow \langle\!\langle\ \alpha, [\mathit{incSent}(y, \Phi)]_a \mid \mu, [c \triangleleft \texttt{info}(y, z, b)]\ \rangle\!\rangle^{\rho}_{\chi}$$

where $\Psi = \{\texttt{CreatedUsing}(y : a \multimap c, z : b \multimap c)\}$

$\textsc{Info}(y, z, b, c)$

$$\langle\!\langle\ \alpha, (\Phi)_c \mid \mu, [c \triangleleft \texttt{info}(y, z, b)]\ \rangle\!\rangle^{\rho}_{\chi} \longrightarrow \langle\!\langle\ \alpha, (\mathit{incRecv}(y, \Phi) \cup \Psi)_c \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

where $\Psi = \{\texttt{Created}(z : b \multimap c)\}$

$\textsc{SendRelease}(x, a, b)$

$$\langle\!\langle\ \alpha, [\Phi \cup \Psi]_a \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \longrightarrow \langle\!\langle\ \alpha, [\Phi]_a \mid \mu, [b \triangleleft \texttt{release}(x, n)]\ \rangle\!\rangle^{\rho}_{\chi}$$

where $\Psi = \{\texttt{Active}(x : a \multimap b), \texttt{Sent}(x, n)\}$
and $\nexists y,\ \texttt{CreatedUsing}(x, y) \in \Phi$

$\textsc{Release}(x, a, b)$

$$\langle\!\langle\ \alpha, (\Phi)_b \mid \mu, [b \triangleleft \texttt{release}(x, n)]\ \rangle\!\rangle^{\rho}_{\chi} \longrightarrow \langle\!\langle\ \alpha, (\Phi \cup \{\texttt{Released}(x)\})_b \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

only if $\Phi \vdash \texttt{Received}(x, n)$

$\textsc{Compaction}(x, b, c)$

$$\langle\!\langle\ \alpha, (\Phi \cup \Psi)_c \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \longrightarrow \langle\!\langle\ \alpha, (\Phi)_c \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

where $\Psi = \{\texttt{Created}(x : b \multimap c), \texttt{Released}(x : b \multimap c), \texttt{Received}(x, n)\}$
for some $n \in \mathbb{N}$
or $\Psi = \{\texttt{Created}(x : b \multimap c), \texttt{Released}(x : b \multimap c)\}$ and
$\forall n \in \mathbb{N},\ \texttt{Received}(x, n) \notin \Phi$

$\textsc{Snapshot}(a, \Phi)$

$$\langle\!\langle\ \alpha, (\Phi)_a \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \longrightarrow \langle\!\langle\ \alpha, (\Phi)_a \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

Figure 4.4: Rules for performing the release protocol.

delivered with the Info event, which adds the fact $\texttt{Created}(z : b \multimap c)$ to c's knowledge set and correspondingly increments c's Received count for y.

When an actor a no longer needs $x : a \multimap b$ for sending messages, a can deactivate x with the SendRelease event; we say that the release is sent *along* x. A precondition of this event is that

$\textsc{In}(x, a, R)$

$$\langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \longrightarrow \langle\!\langle\ \alpha \mid \mu, [a \triangleleft \mathsf{app}(x, R)]\ \rangle\!\rangle^{\rho}_{\chi \cup \chi'}$$

where $a \in \rho$ and $R = \{x_1 : a \multimap b_1, \ldots, x_n : a \multimap b_n\}$ and $x_1, \ldots, x_n$ fresh
and $\{b_1, \ldots, b_n\} \cap \mathrm{dom}(\alpha) \subseteq \rho$ and $\chi' = \{b_1, \ldots, b_n\} \setminus \mathrm{dom}(\alpha)$

$\textsc{Out}(x, b, R)$

$$\langle\!\langle\ \alpha \mid \mu, [b \triangleleft \mathsf{app}(x, R)]\ \rangle\!\rangle^{\rho}_{\chi} \longrightarrow \langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho \cup \rho'}_{\chi}$$

where $b \in \chi$ and $R = \{x_1 : b \multimap c_1, \ldots, x_n : b \multimap c_n\}$ and $\rho' = \{c_1, \ldots, c_n\} \cap \mathrm{dom}(\alpha)$

$\textsc{ReleaseOut}(x, b)$

$$\langle\!\langle\ \alpha \mid \mu, [b \triangleleft \mathtt{release}(x, n)]\ \rangle\!\rangle^{\rho}_{\chi \cup \{b\}} \longrightarrow \langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi \cup \{b\}}$$

$\textsc{InfoOut}(y, z, a, b, c)$

$$\langle\!\langle\ \alpha \mid \mu, [c \triangleleft \mathtt{info}(y, z, a, b)]\ \rangle\!\rangle^{\rho}_{\chi \cup \{c\}} \longrightarrow \langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi \cup \{c\}}$$

Figure 4.5: Rules for interacting with the outside world.

a has already sent messages to inform b about all the refobs it has created using x. In practice, an implementation may defer sending any `info` or `release` messages to a target b until all a's refobs to b are deactivated. This introduces a trade-off between the number of control messages and the rate of simple garbage detection (Section 4.5).

Each `release` message for a refob x includes a count n of the number of messages sent using x. This ensures that $\mathtt{release}(x, n)$ is only delivered after all the preceding messages sent along x have been delivered. Once the Release event can be executed, it adds the fact that x has been released to b's knowledge set. Once c has received both an `info` and `release` message for a refob x, it may remove facts about x from its knowledge set using the Compaction event.

Finally, the Snapshot event captures an idle actor's knowledge set. For simplicity, we have omitted the process of disseminating snapshots to a garbage collector. Although this event does not change the configuration, it allows us to prove properties about snapshot events at different points in time.

4.2.4 *Composition and Effects*

We give rules to dictate how internal actors interact with external actors in Figure 4.5. The In and Out rules correspond to similar rules in the standard operational semantics of actors.

External actors may or may not participate in the PRL protocol themselves. It would be routine (but tedious) to define the composition of two PRL systems and to give additional rules for exchanging `info` and `release` messages between them. For simplicity, we only define the bare minimum interaction between a system and its environment; all `release` and `info` messages sent to external actors are simply dropped by the ReleaseOut and InfoOut events. For a complete formalization of actor composition, see [25]. We do, however, explore how garbage collectors from different actor systems can cooperate to detect cycles of quiescent actors across the two systems (Section 5.3).

The In event allows an external actor to send an application-level message to a receptionist a containing a set of refobs R, all owned by a. If the external actor participates in PRL, the message is annotated as usual with the token x used to send the message. Otherwise, a special `null` token can be used instead. All targets in R that are not internal actors are added to the set of external actors.

The Out event delivers an application-level message to an external actor with a set of refobs R. All internal actors referenced in R become receptionists because their addresses have been exposed to the outside world.

4.3 Basic Properties

We now prove some basic properties of our model, both to help understand its semantics and to assist with later proofs.

Lemma 4.1. If b has undelivered messages along $x : a \multimap b$, then x is an unreleased refob.

Proof. There are three types of messages: $\texttt{app}(x, -)$, $\texttt{info}(x, -, -, -)$, and $\texttt{release}(x, -)$. All three messages can only be sent when x is active. Moreover, the Release rule ensures that they must all be delivered before x can be released. QED.

Lemma 4.2.

- Once $\texttt{CreatedUsing}(y : a \multimap c, z : b \multimap c)$ is added to a's knowledge set, it will not be removed until after a has sent an `info` message containing z to c.
- Once $\texttt{Created}(z : b \multimap c)$ is added to c's knowledge set, it will not be removed until after c has received the (unique) `release` message along z.
- Once $\texttt{Released}(z : b \multimap c)$ is added to c's knowledge set, it will not be removed until after c has received the (unique) `info` message containing z.

Proof. Immediate from the transition rules. QED.

The following lemma formalizes the argument made in Section 4.1. In our model, it is possible for the message counts of two actor snapshots to agree, and yet for there to be undelivered messages between the two actors. However, in the special case where no messages are sent during the interval between the two snapshots, we can indeed trust the message counts to accurately reflect the number of undelivered messages.

Lemma 4.3. Consider a refob $x : a \multimap b$. Let t_1, t_2 be times such that x has not yet been deactivated at t_1 and x has not yet been released at t_2. In particular, t_1 and t_2 may be before the creation time of x.

Suppose that $\alpha_{t_1}(a) \vdash \texttt{Sent}(x, n)$ and $\alpha_{t_2}(b) \vdash \texttt{Received}(x, m)$ and, if $t_1 < t_2$, that a does not send any messages along x during the interval $[t_1, t_2]$. Then the difference $\max(n - m, 0)$ is the number of messages sent along x before t_1 that were not received before t_2.

Proof. Since x is not deactivated at time t_1 and unreleased at time t_2, the message counts were never reset by the SendRelease or Compaction rules. Hence n is the number of messages a sent along x before t_1 and m is the number of messages b received along x before t_2. Hence $\max(n - m, 0)$ is the number of messages sent before t_1 and *not* received before t_2. QED.

4.4 Garbage

We can now operationally characterize actor garbage in our model. An actor a can *potentially receive a message* in κ if there is a sequence of events (possibly of length zero) leading from κ to a configuration κ' in which a has an undelivered message. We say that an actor is *quiescent* if it is idle and cannot potentially receive a message.

An actor is *blocked* if it satisfies three conditions: (1) it is idle, (2) it is not a receptionist, and (3) it has no undelivered messages; otherwise, it is *unblocked*. We define *potential reachability* as the reflexive transitive closure of the potential acquaintance relation. That is, a_1 can potentially reach a_n if and only if there is a sequence of unreleased refobs $(x_1 : a_1 \multimap a_2), \ldots, (x_n : a_{n-1} \multimap a_n)$; recall that a refob $x : a \multimap b$ is unreleased if its target b has not yet received a `release` message for x.

Notice that an actor can potentially receive a message if and only if it is potentially reachable from an unblocked actor. Hence an actor is quiescent if and only if it is only potentially reachable by blocked actors. A special case of this is *simple garbage*, in which an actor is blocked and has no potential inverse acquaintances besides itself.

We say that a set of actors S is *closed* at time t (with respect to the potential inverse acquaintance relation) if, whenever $b \in S$ and there is an unreleased refob $x : a \multimap b$ at time t, then also

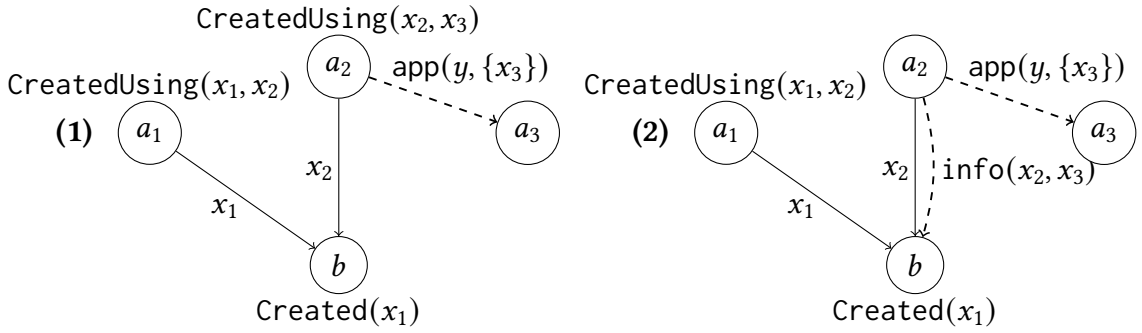


Figure 4.6: An example of a chain from b to x_3.

$a \in S$. The *closure* of a set of actors S' is the smallest closed superset of S'. Notice that the closure of a set of quiescent actors is also a set of quiescent actors.

4.5 Chain Lemma

To determine if an actor has quiescent, one must show that all of its potential inverse acquaintances have quiescent. This appears to pose a problem for quiescence detection, since actors cannot have a complete listing of all their potential inverse acquaintances without some synchronization: actors would need to consult their acquaintances before creating new references to them. In this section, we show that the PRL protocol provides a weaker guarantee that will nevertheless prove sufficient: knowledge about an actor's refobs is *distributed* across the system and there is always a "path" from the actor to any of its potential inverse acquaintances.

Let us construct a concrete example of such a path, depicted by Figure 4.6. Suppose that a_1 spawns b, gaining a refob $x_1 : a_1 \multimap b$. Then a_1 may use x_1 to create $x_2 : a_2 \multimap b$, which a_2 may receive and then use x_2 to create $x_3 : a_3 \multimap b$.

At this point, there are unreleased refobs owned by a_2 and a_3 that are not included in b's knowledge set. However, Figure 4.6 shows that the distributed knowledge of b, a_1, a_2 creates a "path" to all of b's potential inverse acquaintances. Since a_1 spawned b, b knows the fact $\texttt{Created}(x_1)$. Then when a_1 created x_2, it added the fact $\texttt{CreatedUsing}(x_1, x_2)$ to its knowledge set, and likewise a_2 added the fact $\texttt{CreatedUsing}(x_2, x_3)$; each fact points to another actor that owns an unreleased refob to b (Figure 4.6 (1)).

Since actors can remove `CreatedUsing` facts by sending `info` messages, we also consider (Figure 4.6 (2)) to be a "path" from b to a_3. But notice that, once b receives the `info` message, the fact $\texttt{Created}(x_3)$ will be added to its knowledge set and so there will be a "direct path" from b to a_3. We formalize this intuition with the notion of a *chain* in a given configuration $\langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$:

Definition 4.1. A *chain to* the refob $x : a \multimap b$ is a sequence of unreleased refobs $x_1 : a_1 \multimap b$, $\ldots, x_n : a_n \multimap b$ such that:

- $\alpha(b) \vdash \texttt{Created}(x_1 : a_1 \multimap b)$;
- For all $i < n$, either $\alpha(a_i) \vdash \texttt{CreatedUsing}(x_i, x_{i+1})$ or the message $[b \triangleleft \texttt{info}(x_i, x_{i+1})]$ is in transit; and
- $a_n = a$ and $x_n = x$.

We say that an actor b is *in the root set* if it is a receptionist or if there is an application message $\texttt{app}(x, R)$ in transit to an external actor with $b \in \text{targets}(R)$.

Lemma 4.4 (Chain Lemma)**.** Let b be an internal actor in κ. If b is not in the root set, then there is a chain to every unreleased refob $x : a \multimap b$. Otherwise, there is a chain to some refob $y : c \multimap b$ where c is an external actor.

Remark: When b is in the root set, not all of its unreleased refobs are guaranteed to have chains. This is because an external actor may send b's address to other receptionists without sending an `info` message to b.

Proof. We prove that the invariant holds in the initial configuration and at all subsequent times by induction on events $\kappa \xrightarrow{e} \kappa'$, omitting events that do not affect chains. Let $\kappa = \langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$ and $\kappa' = \langle\!\langle\ \alpha' \mid \mu'\ \rangle\!\rangle^{\rho'}_{\chi'}$.

In the initial configuration, the only refob to an internal actor is $y : a \multimap a$. Since a knows $\texttt{Created}(y : a \multimap a)$, the invariant is satisfied.

In the cases below, let x, y, z, a, b, c be free variables, not referencing the variables used in the statement of the lemma.

- $\textsc{Spawn}(x, a, b)$ creates a new unreleased refob $x : a \multimap b$, which satisfies the invariant because $\alpha'(b) \vdash \texttt{Created}(x : a \multimap b)$.
- $\textsc{Send}(x, \vec{y}, \vec{z}, a, b, \vec{c})$ creates a set of refobs R. Let $(z : b \multimap c) \in R$, created using $y : a \multimap c$.
  If c is already in the root set, then the invariant is trivially preserved. Otherwise, there must be a chain $(x_1 : a_1 \multimap c), \ldots, (x_n : a_n \multimap c)$ where $x_n = y$ and $a_n = a$. Then $x_1, \ldots, x_n, z$ is a chain in κ', since $\alpha'(a_n) \vdash \texttt{CreatedUsing}(x_n, z)$.
  If b is an internal actor, then this shows that every unreleased refob to c has a chain in κ'. Otherwise, c is in the root set in κ'. To see that the invariant still holds, notice that $z : b \multimap c$ is a witness of the desired chain.
- $\textsc{SendInfo}(y, z, a, b, c)$ removes the $\texttt{CreatedUsing}(y, z)$ fact but also sends $\texttt{info}(y, z, b)$, so chains are unaffected.
- $\textsc{Info}(y, z, b, c)$ delivers $\texttt{info}(y, z, b)$ to c and adds $\texttt{Created}(z : b \multimap c)$ to its knowledge set.

Suppose $z : b \multimap c$ is part of a chain $(x_1 : a_1 \multimap c), \ldots, (x_n : a_n \multimap c)$, i.e. $x_i = y$ and $x_{i+1} = z$ and $a_{i+1} = b$ for some $i < n$. Since $\alpha'(c) \vdash \texttt{Created}(x_{i+1} : a_{i+1} \multimap c)$, we still have a chain $x_{i+1}, \ldots, x_n$ in κ'.

- $\textsc{Release}(x, a, b)$ releases the refob $x : a \multimap b$. Since external actors never release their refobs, both a and b must be internal actors.
  Suppose the released refob was part of a chain $(x_1 : a_1 \multimap b), \ldots, (x_n : a_n \multimap b)$, i.e. $x_i = x$ and $a_i = a$ for some $i < n$. We will show that $x_{i+1}, \ldots, x_n$ is a chain in κ'.
  Before performing $\textsc{SendRelease}(x_i, a_i, b)$, a_i must have performed the $\textsc{Info}(x_i, x_{i+1}, a_{i+1}, b)$ event. Since the `info` message was sent along x_i, Lemma 4.1 ensures that the message must have been delivered before the present $\textsc{Release}$ event. Furthermore, since x_{i+1} is an unreleased refob in κ', Lemma 4.2 ensures that $\alpha'(b) \vdash \texttt{Created}(x_{i+1} : a_{i+1} \multimap b)$.
- $\textsc{In}(a, R)$ adds a message from an external actor to the internal actor a. This event can only create new refobs that point to receptionists, so it preserves the invariant.
- $\textsc{Out}(x, b, R)$ emits a message $\texttt{app}(x, R)$ to the external actor b. Since all targets in R are already in the root set, the invariant is preserved.

QED.

An immediate application of the Chain Lemma is to allow actors to detect when they are simple garbage. If any actor besides b owns an unreleased refob to b, then b must have a fact $\texttt{Created}(x : a \multimap b)$ in its knowledge set where $a \neq b$. Hence, if b has no such facts, then it must have no nontrivial potential inverse acquaintances. Moreover, since actors can only have undelivered messages along unreleased refobs, b also has no undelivered messages from any other actor; it can only have undelivered messages that it sent to itself. This gives us the following result:

Theorem 4.1. Suppose b is idle with knowledge set Φ, such that:

- Φ does not contain any facts of the form $\texttt{Created}(x : a \multimap b)$ where $a \neq b$; and
- for all facts $\texttt{Created}(x : b \multimap b) \in \Phi$, also $\Phi \vdash \texttt{Sent}(x, n) \wedge \texttt{Received}(x, n)$ for some n.

Then b is simple garbage.

5

Quiescence Detection

In the following chapter, we present the scheme for detecting non-simple quiescent actors in PRL. First, we define what it means for a collage to be *finalized* and prove that finalized sets correspond to closed sets of quiescent actors. This reduces quiescence detection to simply collecting snapshots at a garbage collector and periodically searching the collection for finalized subsets. We prove that such an approach is safe and live (under reasonable fairness assumptions). Next, in Section 5.2, we give an algorithm for finding the *maximum* finalized subset—the union of all finalized subsets—in an arbitrary collage. This gives each garbage collector an efficient procedure for detecting quiescent actors. Lastly, in Section 5.3, we show how a decentralized group of garbage collectors can cooperate to detect distributed garbage while exchanging minimal information. This makes PRL's quiescence detection scalable, parallelizable, and capable of making progress despite network partitions.

5.1 Consistent and Finalized Snapshots

Recall that when we speak of a collage Q, we assume each snapshot was taken by a different actor. We will therefore represent Q as a mapping from actor names to snapshots, with $Q(a)$ denoting a's snapshot in Q.

As shown in Chapter 4, actor snapshots taken at different times can result in conflicting accounts of the configuration. Hence, in general, an arbitrary collage Q does not accurately describe the current configuration. If a collage *does* accurately describe the configuration, we say that it is *consistent*. Formally:

Definition 5.1. Q is consistent at time t when $\forall \phi, \forall a \in \mathrm{dom}(Q),\ Q(a) \vdash \phi$ if and only if $\alpha_t(a) \vdash \phi$.

That is, the snapshot $Q(a)$ may not have been taken at time t—yet the contents of a's knowledge set at time t are the same as $Q(a)$. If Q is consistent at time t, then it is as if all the actors of $\mathrm{dom}(Q)$ took their snapshots at time t.

Another important notion is that of a quiescent actor's *final action*. We define this as the last non-snapshot event that an actor performs before becoming quiescent. Notice that an actor's final action can only be an IDLE, INFO, or RELEASE event. This is because quiescent actors are idle, and only these three events change an actor's status from busy to idle. Note also that the final action may come *strictly before* an actor becomes quiescent, since a blocked actor is only considered to be quiescent once all of its potential inverse acquaintances are quiescent.

We can now give a simple proof of our earlier claim that snapshots from quiescent actors are consistent. This property will allow us to treat finalized collages as if they were all taken at an instant in global time.

Lemma 5.1. Let S be a closed set of quiescent actors at time t_f. If every actor in S took a snapshot sometime after its final action, then the resulting set of snapshots Q is consistent at t_f.

Proof. a quiescent actor's knowledge set never changes. Moreover, an actor's knowledge set cannot change between the point of performing its final action and becoming quiescent. Hence each a's snapshot in Q agrees with its knowledge set at time t_f. QED.

5.1.1 Finalized sets

Given a consistent collage Q, is it possible to determine whether the actors $\mathrm{dom}(Q)$ are quiescent? Let us begin by giving an alternative characterization of quiescent actors.

Lemma 5.2. Let a be an actor at time t and let S be the *closure* of $\{a\}$. Then a is quiescent if and only if the actors of S are idle and *mutually quiescent*, i.e. there are no undelivered messages between actors of S.

Proof. By definition, a is quiescent if and only if every actor that can potentially reach a is blocked. Notice that the closure S is precisely the set of actors that can potentially reach a. Hence, if a is quiescent then the actors of S are blocked, i.e. idle and have no undelivered messages from any actor.

Conversely, let the actors of S be idle and mutually quiescent. Could any of them have undelivered messages from actors outside S? The basic property Lemma 4.1 shows that this cannot occur; any sender of such a message must be in S. Hence the actors of S are all blocked and a is quiescent. QED.

By the above lemma, we can only conclude that the actors of $\mathrm{dom}(Q)$ are quiescent if $\mathrm{dom}(Q)$ is closed, mutually quiescent, and all the actors are idle. This last condition is automatically satisfied by our communication protocol, since only idle actors take snapshots. In order to check the other two conditions, we will inspect the snapshots themselves.

To check that $\mathrm{dom}(Q)$ is closed, recall that every unreleased refob $x : a \multimap b$ has a chain, $(x_1 : a_1 \multimap b), \ldots, (x_n : a_n \multimap b)$. If $\mathrm{dom}(Q)$ is quiescent, then there can be no undelivered `info` messages. Hence x must satisfy the following predicate:

Definition 5.2. Let $Q \vdash \texttt{Chain}(x : a \multimap b)$ if there exist $(x_1 : a_1 \multimap b), \ldots, (x_n : a_n \multimap b)$ such that:

1. $Q \vdash \texttt{Created}(x_1)$ and $Q \nvdash \texttt{Released}(x_1)$;
2. For all $i < n$, $Q \vdash \texttt{CreatedUsing}(x_i, x_{i+1})$ and $Q \nvdash \texttt{Released}(x_{i+1})$;
3. $a_n = a$ and $x_n = x$.

Checking where $\mathrm{dom}(Q)$ is closed therefore amounts to checking that $Q \vdash \texttt{Chain}(x : a \multimap b)$ and $b \in Q$ always implies $a \in Q$.

To decide whether $\mathrm{dom}(Q)$ is mutually quiescent, we need to check that there are no undelivered messages along each unreleased refob $x : a \multimap b$ where $a, b \in \mathrm{dom}(Q)$. In a consistent collage, we can do so by inspecting the message counts of a and b for x. Hence x must satisfy the following predicate:

Definition 5.3. Let $Q \vdash \texttt{Relevant}(x : a \multimap b)$ if there exists n such that $Q \vdash \texttt{Active}(x) \wedge \texttt{Sent}(x, n) \wedge \texttt{Received}(x, n)$.

Together, we can use these predicates to characterize a collage from a closed, quiescent set of actors.

Definition 5.4. A collage Q is *finalized* if, for all $b \in \mathrm{dom}(Q)$ and for all $x : a \multimap b$,

1. $Q \vdash \texttt{Chain}(x)$ implies $a \in Q$; and
2. $Q \vdash \texttt{Chain}(x)$ implies $Q \vdash \texttt{Relevant}(x)$.

The first condition ensures that $\mathrm{dom}(Q)$ is closed; $Q \vdash \texttt{Chain}(x : a \multimap b)$ implies that a is a potential inverse acquaintance of b. The second condition ensures that $\mathrm{dom}(Q)$ is mutually quiescent: between any two actors $a, b \in \mathrm{dom}(Q)$, the message counts for any unreleased refob $x : a \multimap b$ must agree.

If finalized sets Q correspond to closed sets of quiescent actors S, then we would expect that a consistent snapshot of S is finalized. This is indeed the case:

Theorem 5.1. Let Q be a consistent collage at time t of a closed set of quiescent actors S. Then Q is finalized.

Proof. First, we show that if $b \in \mathrm{dom}(Q)$ and $x : a \multimap b$ is an unreleased refob at time t, then $Q \vdash \texttt{Chain}(x) \wedge \texttt{Active}(x) \wedge \texttt{Sent}(x, n) \wedge \texttt{Received}(x, n)$ for some n.

- $Q \vdash \texttt{Chain}(x)$ follows from Lemma 4.4 because b is blocked and S is closed.

- $Q \vdash \texttt{Active}(x)$ holds because x must be activated: if x were pending then a would be unblocked and if x were deactivated then b would be unblocked.
- $Q \vdash \texttt{Sent}(x, n) \wedge \texttt{Received}(x, n)$ holds because there are no undelivered messages between a and b at time t, so the send and receive counts of x at time t must agree.

Now it suffices to show that, if $Q \vdash \texttt{Chain}(x)$, then $x : a \multimap b$ is unreleased at time t. There are two cases: Either $Q(b) \vdash \texttt{Created}(x)$ or $Q(c) \vdash \texttt{CreatedUsing}(y, x)$ for some c, y. In both cases, x has been created before time t. Since Q is consistent and $Q(b) \nvdash \texttt{Released}(x)$, it follows from Lemma 4.2 that b has not yet received a `release` message for x. Hence x is unreleased at time t. QED.

By contrapositive, if Q is *not* finalized then it cannot be a consistent collage from a closed set of quiescent actors. Recall also that any collage from quiescent actors is guaranteed to be consistent (Lemma 5.1). Hence, if Q is not finalized, then either some actor in Q is not quiescent or $\mathrm{dom}(Q)$ is not closed. The latter case indicates that there is insufficient information to conclude whether the actors of $\mathrm{dom}(Q)$ are quiescent; they may or may not be reachable by an unblocked actor outside of $\mathrm{dom}(Q)$.

We now show that, surprisingly, the converse of Theorem 5.1 also holds: any finalized collage Q necessarily describes a closed set of quiescent actors, with each snapshot taken some point after the actor's final action. By Lemma 5.1, such a collage is also consistent.

Given a collage Q taken before some time t_f, we write Q_t to denote those snapshots in Q that were taken before time $t < t_f$. If $a \in \mathrm{dom}(Q)$, we denote the time of a's snapshot as t_a.

Theorem 5.2. Let Q be a finalized collage at time t_f. Then for all times t:

1. If $b \in \mathrm{dom}(Q_t)$ and $x : a \multimap b$ is unreleased, then $Q \vdash \texttt{Chain}(x)$.
2. The actors of Q_t are all blocked.

In particular $Q_t = Q$, when $t \geq t_f$.

Proof. Proof by induction on t. Notice that these two properties trivially hold in the initial configuration because $Q_0 = \emptyset$.

For the induction step, assume both properties hold at time t and call them IH-1 and IH-2, respectively. We show that IH-1 and IH-2 are preserved by any legal transition $\kappa \xrightarrow{e} \kappa'$.

Snapshot(b, Φ) Suppose $b \in \mathrm{dom}(Q)$ takes a snapshot at time t. We show that if $x : a \multimap b$ is unreleased at time t, then $Q \vdash \texttt{Chain}(x)$ and there are no undelivered messages along x from a to b. We do this with the help of two lemmas.

Lemma 5.3. If $Q \vdash \texttt{Chain}(x : a \multimap b)$, then x is unreleased at time t and there are no undelivered messages along x at time t. Moreover, if $t_a > t$, then there are no undelivered messages along x throughout the interval $[t, t_a]$.

Proof (Lemma). Since $Q \vdash \texttt{Relevant}(x : a \multimap b)$, we have $a \in \text{dom}(Q)$ and $Q \vdash \texttt{Active}(x)$ and $Q \vdash \texttt{Sent}(x, n) \wedge \texttt{Received}(x, n)$ for some n.

Consider the case when $t_a > t$. Since $Q(a) \vdash \texttt{Active}(x)$, x is not deactivated and therefore not released at t_a or t. Hence, by Lemma 4.3, every message sent along x before t_a was received before t. Since message sends precede receipts, each of those messages was sent before t. Hence there are no undelivered messages along x throughout $[t, t_a]$.

Now consider the case when $t_a < t$. Since $Q(a) \vdash \texttt{Active}(x)$, x is not deactivated and not released at t_a. By IH-2, a was blocked throughout the interval $[t_a, t]$, so it could not have sent a `release` message. Hence x is still not deactivated at t and therefore not released at t. By Lemma 4.3, all messages sent along x before t_a must have been delivered before t. Hence, there are no undelivered messages along x at time t. QED.

Lemma 5.4. Let $x_1 : a_1 \multimap b, \ldots, x_n : a_n \multimap b$ be a chain to $x : a \multimap b$ at t. Then $Q \vdash \texttt{Chain}(x)$.

Proof (Lemma). We prove by induction on the length of the chain that $Q \vdash \texttt{Chain}(x_i)$ for all $i \leq n$.
Base case: By the definition of a chain, $\alpha_t(b) \vdash \texttt{Created}(x_1)$ and $\alpha_t(b) \nvdash \texttt{Released}(x_1)$. Since b's snapshot happens at time t, we must have $Q(b) \vdash \texttt{Created}(x_1)$ and $Q(b) \nvdash \texttt{Released}(x_1)$.
Induction step: Assume $Q \vdash \texttt{Chain}(x_i)$. Notice that $Q \nvdash \texttt{Released}(x_{i+1})$ because x_{i+1} is unreleased at the time of b's snapshot. Hence, it suffices to show that $Q \vdash \texttt{CreatedUsing}(x_i, x_{i+1})$.

Since $Q \vdash \texttt{Relevant}(x_i)$, we must have $a_i \in \text{dom}(Q)$. Let t_i be the time of a_i's snapshot; we will show $\alpha_{t_i}(a_i) \vdash \texttt{CreatedUsing}(x_i, x_{i+1})$.

By the definition of a chain, either the message $[b \triangleleft \texttt{info}(x_i, x_{i+1})]$ is in transit at time t, or $\alpha_t(a_i) \vdash \texttt{CreatedUsing}(x_i, x_{i+1})$. But the first case is impossible by Lemma 5.3, so we only need to consider the latter.

Consider the case where $t_i > t$. Lemma 5.3 implies that a_i cannot perform the event SendInfo(x_i, x_{i+1}, a_{i+1}, b) during $[t, t_i]$. Hence $\alpha_{t_i}(a_i) \vdash \texttt{CreatedUsing}(x_i, x_{i+1})$.

Now consider the case where $t_i < t$. By IH-2, a_i must have been blocked throughout the interval $[t_i, t]$. Hence a_i could not have created any refobs during this interval, so x_{i+1} must have been created before t_i. This implies $\alpha_{t_i}(a_i) \vdash \texttt{CreatedUsing}(x_i, x_{i+1})$. QED.

Lemma 5.4 implies that b cannot be in the root set. If it were, then by the Chain Lemma there would be a refob $y : c \multimap b$ with a chain where c is an external actor. Since $Q \vdash \texttt{Chain}(y)$, there would need to be a snapshot from c in Q—but external actors do not take snapshots, so this is impossible.

Since b is not in the root set, there must be a chain to every unreleased refob $x : a \multimap b$. By Lemma 5.4, $Q \vdash \texttt{Chain}(x)$. By Lemma 5.3, there are no undelivered messages to b along x at time

t. Since b can only have undelivered messages along unreleased refobs (Lemma 4.1), the actor is indeed blocked.

Send$(x, \vec{y}, \vec{z}, a, b, \vec{c})$ In order to maintain IH-2, we must show that if $b \in \text{dom}(Q_t)$ then this event cannot occur. So suppose $b \in \text{dom}(Q_t)$. By IH-1, we must have $Q \vdash \texttt{Chain}(x : a \multimap b)$, so $a \in \text{dom}(Q)$. By IH-2, we moreover have $a \notin \text{dom}(Q_t)$—otherwise a would be blocked and unable to send this message. Since $Q \vdash \texttt{Relevant}(x)$, we must have $Q(a) \vdash \texttt{Sent}(x, n)$ and $Q(b) \vdash \texttt{Received}(x, n)$ for some n. Hence x is not deactivated at t_a and unreleased at t_b. By Lemma 4.3, every message sent before t_a is received before t_b. Hence a cannot send this message to b because $t_a > t > t_b$.

In order to maintain IH-1, we will show that if one of the refobs sent to b in this step is $z : b \multimap c$, where $c \in \text{dom}(Q_t)$, then $Q \vdash \texttt{Chain}(z)$. In the configuration that follows this Send event, $\texttt{CreatedUsing}(y, z)$ occurs in a's knowledge set. By the same argument as above, $a \in \text{dom}(Q) \backslash Q_t$ and $Q(a) \vdash \texttt{Sent}(y, n)$ and $Q(c) \vdash \texttt{Received}(y, n)$ for some n. Hence a cannot perform the SendInfo(y, z, a, b, c) event before t_a, so $Q(a) \vdash \texttt{CreatedUsing}(y, z)$. Since $Q \vdash \texttt{Chain}(y) \wedge \texttt{CreatedUsing}(y, z)$ and $Q \nvdash \texttt{Released}(z)$, we have $Q \vdash \texttt{Chain}(z)$.

SendInfo(y, z, a, b, c) By the same argument as above, $a \notin \text{dom}(Q_t)$ cannot send an `info` message to $b \in \text{dom}(Q_t)$ without violating message counts, so IH-2 is preserved.

SendRelease(x, a, b) Suppose that $a \notin \text{dom}(Q_t)$ and $b \in \text{dom}(Q_t)$. By IH-1, $Q \vdash \texttt{Chain}(x)$ at time t. Since $Q \vdash \texttt{Relevant}(x)$, it follows that $Q(a) \vdash \texttt{Active}(x)$. Hence a cannot deactivate x and IH-2 is preserved.

In(a, R) By IH-1, every potential inverse acquaintance of an actor in Q_t is also in Q. Hence none of the actors in Q_t is a receptionist and this rule does not affect the invariants.

Out(x, b, R) Suppose $(y : b \multimap c) \in R$ where $c \in \text{dom}(Q_t)$. Then y is unreleased and $Q \vdash \texttt{Chain}(y)$ and $b \in \text{dom}(Q)$. But this is impossible because b is an external actor and external actors do not take snapshots.

QED.

Corollary 5.1 (Safety)**.** If Q is a finalized collage at time t_f then the actors in Q are all quiescent at t_f.

Proof. Theorem 5.2 implies that, at t_f, all the actors in Q are blocked. Together with the fact that Q is finalized, it also implies that Q is closed under the potential inverse acquaintance relation

at t_f. Hence every actor that can potentially reach $b \in \mathrm{dom}(Q)$ at t_f is blocked, so by definition every $b \in \mathrm{dom}(Q)$ is quiescent at t_f. QED.

Recall that a garbage collector detects quiescent actors by receiving actor snapshots and periodically looking for finalized subsets. It is now simple to see that this algorithm is live, under reasonable fairness assumptions:

Theorem 5.3 (Liveness)**.** If every actor eventually takes a snapshot after performing an IDLE, INFO, or RELEASE event, then every quiescent actor is eventually part of a finalized collage.

Proof. If an actor a is quiescent, then the closure S of $\{a\}$ is a quiescent set of actors. Every actor eventually takes a snapshot after taking its final action and the resulting collage is consistent, by Lemma 5.1. Then Theorem 5.1 implies that the resulting snapshots is finalized. QED.

5.1.2 *Strongly finalized sets*

Note that our definition of finalized sets differs from the definition which originally appeared in [56]. This old definition, which we now call "strongly finalized", used $Q \vdash \texttt{Unreleased}(x)$ instead of $Q \vdash \texttt{Chain}(x)$:

Definition 5.5. a collage Q is *strongly finalized* if, for all $b \in \mathrm{dom}(Q)$ and for all $x : a \multimap b$, $Q \vdash \texttt{Unreleased}(x)$ implies $Q \vdash \texttt{Relevant}(x)$.

In fact, the two notions of finalized are equivalent. However, in the process of developing the theory in Sections 5.2 and 5.3, we found this old definition to be inconvenient.

Notice that any strongly finalized Q is also finalized because $Q \vdash \texttt{Chain}(x)$ implies $Q \vdash \texttt{Unreleased}(x)$. However, in an arbitrary set Q, $Q \vdash \texttt{Unreleased}(x)$ does not imply $Q \vdash \texttt{Chain}(x)$; there could exist $a, b, c \in \mathrm{dom}(Q)$ such that $Q(a) \vdash \texttt{CreatedUsing}(x : a \multimap c, y : b \multimap c)$ but $Q \nvdash \texttt{Chain}(x)$. Could such a situation occur in a finalized Q? We prove below that no, this is impossible:

Theorem 5.4. If Q is finalized then Q is strongly finalized.

Proof. We will show that if Q is finalized, then $Q \vdash \texttt{Unreleased}(x : a \multimap b)$ and $b \in \mathrm{dom}(Q)$ implies $Q \vdash \texttt{Chain}(x : a \multimap b)$.

Note that Q is consistent at some time t_f, since $\mathrm{dom}(Q)$ is a closed quiescent set of actors where each snapshot was taken after the actor's final action.

By definition, $Q \vdash \texttt{Unreleased}(x : a \multimap b)$ implies that $Q \nvdash \texttt{Released}(x)$ and either (1) $Q \vdash \texttt{Created}(x)$ or (2) $Q \vdash \texttt{Chain}(y) \wedge \texttt{CreatedUsing}(y, x)$ for some $y : c \multimap b$.

In case (1), we have a trivial chain $Q \vdash \texttt{Chain}(x)$.

In case (2), notice that we must have $Q \vdash \texttt{Active}(y)$ because $Q \vdash \texttt{Relevant}(y)$. Since Q is consistent, y must be an unreleased refob at t_f. Due to Lemma 4.4, there must be a chain of unreleased refobs $(y_1 : c_1 \multimap b), \ldots, (y_n : c_n \multimap b)$ at t_f. Again since Q is consistent, we must have $Q \vdash \texttt{Created}(y_1)$ and $Q \vdash \texttt{CreatedUsing}(y_i, y_{i+1})$ for each $i < n$ and $Q \nvdash \texttt{Released}(y_i)$ for each $i \leq n$. Hence $Q \vdash \texttt{Chain}(y)$. Since also $Q \vdash \texttt{CreatedUsing}(y, x)$ and $Q \nvdash \texttt{Released}(x)$, we can "extend" this chain to derive $Q \vdash \texttt{Chain}(x)$. QED.

Thanks to this theorem, we can use the two definitions interchangeably.

5.2 Maximal Finalized Subsets

In the previous section, we showed that a finalized collage corresponds to a closed set of quiescent actors. Hence, the problem of garbage collection reduces to finding all the finalized subsets of an arbitrary collage Q. In this section, we show that there is in fact a single largest finalized subset $Q_f \subseteq Q$ that contains all other finalized subsets. We then show that Q_f can be computed in linear time by removing all snapshots that cannot appear in a finalized subset.

Originally, we presented a slightly different algorithm in [56]. It operates similarly to the new algorithm, iteratively removing snapshots that appear not to be in a finalized subset. However, we subsequently discovered through model checking that the original algorithm could sometimes be overzealous: the presence of certain "stale" snapshots in Q can cause other snapshots to be unnecessarily removed. In other words, the computed set is finalized but not necessarily maximal. Nevertheless, since the original algorithm can have better cache locality than the new algorithm, it may be more practical for real systems. We present the algorithm again in Section 5.2.2 and go on to prove that it *eventually* detects all quiescent actors, under reasonable fairness conditions.

5.2.1 Chain Algorithm

Let us first show that a maximum finalized subset always exists. Notice that finalized sets are closed under union when they agree on $\text{dom}(Q_1) \cap \text{dom}(Q_2)$:

Lemma 5.5. Let Q_1, Q_2 be finalized collages that agree at their intersection, i.e. $\forall a \in \text{dom}(Q_1) \cap \text{dom}(Q_2),\ Q_1(a) = Q_2(a)$. Then $Q_1 \cup Q_2$ is also finalized.

Proof. Suppose there exists $x : a \multimap b$ such that $Q_1 \cup Q_2 \vdash \texttt{Chain}(x)$ and $Q_1 \cup Q_2 \nvdash \texttt{Relevant}(x)$. Let $(x_1 : a_1 \multimap b), \ldots, (x_n : a_n \multimap b)$ be the chain.

Let Q be either Q_1 or Q_2; we prove by induction on n that, if $b \in \text{dom}(Q)$, then $\forall i \leq n,\ Q \vdash \texttt{Chain}(x_i)$. If $n = 1$ then $Q(b) \vdash \texttt{Created}(x_1)$ and $Q(b) \nvdash \texttt{Released}(x_1)$ implies $Q \vdash \texttt{Chain}(x_1)$.

For $n > 1$, $Q \vdash \texttt{Chain}(x_{n-1})$ implies $Q \vdash \texttt{Relevant}(x_{n-1})$ since Q is finalized, and therefore $a_{n-1} \in \text{dom}(Q)$. Hence $Q(a_{n-1}) \vdash \texttt{CreatedUsing}(x_{n-1}, x_n)$, which implies $Q \vdash \texttt{Chain}(x_n)$.

Since each Q_1, Q_2 is finalized, we must therefore have $Q \vdash \texttt{Relevant}(x)$, i.e. $Q \vdash \texttt{Active}(x) \wedge \texttt{Sent}(x, n) \wedge \texttt{Received}(x, n)$ for some n. By definition of $(\vdash)$, it follows that $Q_1 \cup Q_2 \vdash \texttt{Relevant}(x)$. QED.

Any two finalized subsets of Q will satisfy the condition of this lemma. Hence the maximum finalized subset Q_f is the union of all finalized subsets of Q.

Next, we characterize which snapshots in Q *can* and *cannot* appear in a finalized subset of Q. To this end, we define the following useful concept:

Definition 5.6. We say that b *depends on* a in Q if $a = b$ or there is a sequence of one or more refobs $(x_1 : a_1 \multimap a_2), \ldots, (x_n : a_{n-1} \multimap a_n)$ where $a = a_1$ and $b = a_n$ and, for each $i < n$, $Q \vdash \texttt{Chain}(x_i : a_i \multimap a_{i+1})$. Hence the "depends on" relation is reflexive and transitive.

The following lemmas show that if there exists $a \in \text{dom}(Q)$ such that $a \notin \text{dom}(Q_f)$, then every b that depends on a in Q also cannot appear in Q_f.

Lemma 5.6. If $Q \vdash \texttt{Chain}(x : a \multimap b)$ then, for any finalized subset Q_f of Q, if $b \in \text{dom}(Q_f)$ then $Q_f \vdash \texttt{Chain}(x : a \multimap b)$.

Proof. Proof by induction on the length of the chain $(x_1 : a_1 \multimap b), \ldots, (x_n : a_n \multimap b)$.

If $n = 1$ then $Q(b) \vdash \texttt{Created}(x_1)$ and $Q(b) \nvdash \texttt{Released}(x_1)$. Hence any subset Q_f of Q must have $Q_f \vdash \texttt{Chain}(x_1)$.

If $n > 1$, assume $Q_f \vdash \texttt{Chain}(x_{n-1})$. Since Q_f is finalized, $a_{n-1} \in \text{dom}(Q_f)$. Since $Q(a_{n-1}) \vdash \texttt{CreatedUsing}(x_{n-1}, x_n)$ and $Q(b) \nvdash \texttt{Released}(x_n)$, it follows that $Q_f \vdash \texttt{Chain}(x_n)$. QED.

Lemma 5.7. If b depends on a in Q, then every finalized subset of Q containing b must also contain a.

Proof. If $a = b$ then the lemma trivially holds. We prove that this must hold for nontrivial sequences by induction on the length of the sequence $(x_1 : a_1 \multimap a_2), \ldots, (x_n : a_{n-1} \multimap a_n)$.

If $n = 1$ then $Q \vdash \texttt{Chain}(x : a \multimap b)$. Then $Q_f \vdash \texttt{Chain}(x : a \multimap b)$ for any finalized subset Q_f containing b. Since Q_f is finalized, we must also have $Q_f \vdash \texttt{Relevant}(x : a \multimap b)$ and therefore $a \in \text{dom}(Q_f)$.

For $n > 1$, assume any finalized subset containing a_{n-1} must also contain a_1. By the same argument as above, any finalized subset containing a_n must contain a_{n-1} and therefore also contain a_1. QED.

We can also use the notion of dependency to give a new characterization of finalized sets:

Definition 5.7. c is *finalized* in Q if, for all b on which c depends, for all $x : a \multimap b$, $Q \vdash \texttt{Chain}(x)$ implies $Q \vdash \texttt{Relevant}(x)$.

Lemma 5.8. c is finalized in Q if and only if c is in a finalized subset of Q.

Proof. If c is finalized in Q, let Q_f be a subset of Q containing only snapshots of actors on which c depends. To see that Q_f is finalized, first notice that each $b \in \mathrm{dom}(Q_f)$ has a sequence of refobs $(x_1 : a_1 \multimap a_2), \ldots, (x_n : a_{n-1} \multimap a_n)$ where $b = a_1$ and $c = a_n$ and $Q \vdash \texttt{Chain}(x_i : a_i \multimap a_{i+1}$ for each $i < n$. For any $(x : a \multimap b)$, $Q_f \vdash \texttt{Chain}(x : a \multimap b)$ implies $Q \vdash \texttt{Chain}(x : a \multimap b)$ and therefore c depends on a in Q. Hence $Q \vdash \texttt{Relevant}(x)$ and therefore $Q_f \vdash \texttt{Relevant}(x)$.

Conversely, let c be in a finalized subset Q_f and consider a sequence of refobs $(x_1 : a_1 \multimap a_2)$, $\ldots, (x_n : a_{n-1} \multimap a_n)$ where $Q \vdash \texttt{Chain}(x_i : a_i \multimap a_{i+1}$ for each $i < n$. Then $a_i \in \mathrm{dom}(Q_f)$ for each $i \leq n$. Since Q_f is finalized, $Q_f \vdash \texttt{Relevant}(x_i)$ for each $i < n$. Hence $Q \vdash \texttt{Relevant}(x_i)$ for each $i < n$. QED.

Since $c \in Q_f$ if and only if c is finalized in Q, it follows that $c \notin Q_f$ if and only if c is not finalized in Q. Hence, to find the maximum finalized subset of Q it suffices to remove every snapshot that is not finalized in Q.

Algorithm 5.1: Compute the largest finalized subset of Q

Data: Q is an arbitrary collage
Result: S_3 is the largest finalized subset of Q
$S_1 \leftarrow \{b \in \mathrm{dom}(Q) : \exists x,\ Q \vdash \texttt{Chain}(x : a \multimap b)$ and $Q \nvdash \texttt{Relevant}(x : a \multimap b)\}$;
$S_2 \leftarrow \{a \in \mathrm{dom}(Q) : \exists b \in S_1,\ a$ depends on $b\}$;
$S_3 \leftarrow \mathrm{dom}(Q) \setminus S_2$;

Theorem 5.5. Algorithm 5.1 computes the largest finalized subset of Q.

Proof. Clearly, an actor is not finalized in Q if it depends on one of the actors of S_1. Hence S_2 is precisely the set of all actors that are not finalized in Q. Its complement, S_3, is therefore the set of all finalized actors in Q. QED.

5.2.2 *Heuristic algorithm*

Although the algorithm above has $O(m)$ time complexity, where m is the number of unreleased refobs in Q, it can suffer from poor locality: finding every $x : a \multimap b$ such that $Q \vdash \texttt{Chain}(x)$ requires tracing a path from b to all of its potential inverse acquaintances using the chains of `CreatedUsing` facts.

One way to address this problem is to keep the `CreatedUsing` chains short, by having actors not keep the `CreatedUsing` fact in their knowledge set for long periods of time. In the extreme case, actors can immediately perform the SendInfo rule whenever they create a refob. This relieves the garbage collector from dealing with `CreatedUsing` chains entirely, at the cost of increased control messages between actors.

Another interesting approach is for the garbage collector to use a heuristic to find *some* finalized subset, not necessarily the largest one. For our heuristic, notice that $Q \vdash \texttt{Chain}(x)$ implies $Q \vdash \texttt{Unreleased}(x)$ in *any* collage Q. This motivates a new definition:

Definition 5.8. b *potentially depends* on a in Q if $a = b$ or there is a sequence of one or more refobs $(x_1 : a_1 \multimap a_2), \ldots, (x_n : a_{n-1} \multimap a_n)$ where $a = a_1$ and $b = a_n$ and, for each $i < n$, $Q \vdash \texttt{Unreleased}(x_i : a_i \multimap a_{i+1})$.

Notice that if b depends on a, then b also potentially depends on a; the latter is a coarser relation than the former.

Our heuristic algorithm is identical to the original, except that S_2 is the set of all actors that *potentially* depend on S_1. Since the "potentially depends" relation is coarser than the "depends" relation, every snapshot in the resulting set is necessarily in the maximum finalized subset.

Algorithm 5.2: Compute a finalized subset of Q

Data: Q is an arbitrary collage
Result: S_3 is a finalized subset of Q
$S_1 \leftarrow \{b \in \text{dom}(Q) : \exists x,\ Q \vdash \texttt{Chain}(x : a \multimap b) \text{ and } Q \nvdash \texttt{Relevant}(x : a \multimap b)\}$;
$S_2 \leftarrow \{a \in \text{dom}(Q) : \exists b \in S_1,\ a \text{ potentially depends on } b\}$;
$S_3 \leftarrow \text{dom}(Q) \setminus S_2$;

The following lemma shows that, indeed, only "stale" snapshots prevent the resulting set from being the largest finalized subset.

Lemma 5.9. Let Q be an arbitrary collage at time t, and Q_f the largest finalized subset of Q. Let Q' be another collage, all taken after time t, such that $\text{dom}(Q') \cap \text{dom}(Q) = \emptyset$.

Then for all $b \in \text{dom}(Q_f)$, for all $x : a \multimap b$, $Q' \cup Q_f \vdash \texttt{Unreleased}(x : a \multimap b)$ implies $Q' \cup Q_f \vdash \texttt{Chain}(x : a \multimap b)$.

Proof. Since Q_f is finalized, $Q_f \vdash \texttt{Unreleased}(x : a \multimap b)$ implies $Q_f \vdash \texttt{Chain}(x : a \multimap b)$. Moreover, Q_f is a consistent closed snapshot at all times $t' \geq t$. Hence, for ant $b \in \text{dom}(Q_f)$, if $x : a \multimap b$ is unreleased at time t' then $a \in \text{dom}(Q_f)$.

Now let $Q' \cup Q_f \vdash \texttt{Unreleased}(x : a \multimap b)$. By definition, this means $(Q' \cup Q_f)(b) \nvdash \texttt{Released}(x)$ and there exists some c such that $(Q' \cup Q_f)(c) \vdash \texttt{Created}(x)$.

If $c \in \mathrm{dom}(Q_f)$ then $Q_f \vdash \texttt{Unreleased}(x)$ and therefore $Q_f \vdash \texttt{Chain}(x)$ and therefore $Q' \cup Q_f \vdash \texttt{Chain}(x)$.

Now suppose $c \in \mathrm{dom}(Q') \setminus \mathrm{dom}(Q_f)$. This implies $c \neq b$ and therefore, for some $y : c \multimap b$, $Q'(c) \vdash \texttt{CreatedUsing}(y, x)$. This implies that $Q'(c) \vdash \texttt{Active}(y)$, so y is unreleased at the time of c's snapshot t_c. But since $\mathrm{dom}(Q_f)$ is closed at time t_c, this implies $c \in \mathrm{dom}(Q_f)$ after all; a contradiction. Hence $c \in \mathrm{dom}(Q_f)$, so $Q' \cup Q_f \vdash \texttt{Chain}(x)$ by the argument above. QED.

Hence, if every non-quiescent actor eventually takes a snapshot, a garbage collector running the heuristic algorithm will eventually detect all quiescent garbage.

5.3 Cooperative Garbage Collection

Up to this point we have assumed the existence of a single garbage collector that eventually receives all snapshots. However, there is no reason this must be a centralized entity. For instance, we can view a multicore actor system as a composition of n actor systems; one for each processor core. It would be natural to have a garbage collector for each system, dedicated to detecting and collecting quiescent actors in that system. To detect cycles quiescent sets of actors distributed across multiple systems, the garbage collectors can gossip their local snapshots amongst themselves; eventually every garbage collector will obtain enough snapshots to detect all local quiescent actors. Moreover, since the actor model is location-transparent, this same strategy extends to distributed multicore systems as well.

More formally, the cooperative garbage collection problem is for two garbage collectors, with disjoint snapshot sets Q_1, Q_2, to find maximal subsets $\hat{Q}_1 \subseteq Q_1$, $\hat{Q}_2 \subseteq Q_2$, such that $\hat{Q}_1 \cup \hat{Q}_2$ is finalized. For simplicity, we assume that neither Q_1 nor Q_2 has any finalized subsets, since such quiescent actors could be detected without cooperation. Although we only consider the two-party case here, the discussion naturally generalizes to n garbage collectors.

In this formalism, the simple strategy amounts to having the first garbage collector send its entire snapshot set Q_1 to the second garbage collector, and vice versa. This is clearly inefficient for two reasons. Firstly, the two garbage collectors must perform duplicate work to compute the maximum finalized subset of $Q_1 \cup Q_2$. Secondly, each snapshot set seems to contain significantly more information than is necessary to compute $\hat{Q}_1, \hat{Q}_2$; we might expect, for example, that it is only necessary to pass along snapshots from actors at the "border" of Q_1, Q_2 (e.g. the receptionists).

In this section, we address both of the above concerns. We begin by defining *potentially finalized* subsets of Q_1 and Q_2, which omit any snapshots that *a priori* cannot be finalized in $Q_1 \cup Q_2$. Every actor in a potentially finalized set Q depends on one or more of the receptionists in Q.

Hence, computing $\hat{Q}_1, \hat{Q}_2$ reduces to finding the finalized receptionists of Q_1, Q_2. With this insight, we then show how to compute *summaries* $\tilde{Q}_1, \tilde{Q}_2$ of Q_1, Q_2 such that the finalized receptionists in $\tilde{Q}_1 \cup \tilde{Q}_2$ coincide with those of $Q_1 \cup Q_2$. Garbage collectors can therefore simply exchange summaries to find the finalized receptionists. Since summaries can be significantly smaller than the original collage, this technique reduces the amount of data exchanged and reduces the amount of computation needed to detect finalized receptionists.

5.3.1 Potentially finalized sets

An actor a in Q_1 could potentially be finalized in $Q_1 \cup Q_2$ if there exists Q' disjoint from Q_1, such that a is finalized in $Q_1 \cup Q'$. This motivates the following definition:

Definition 5.9. c is *potentially finalized* in Q if, for all b that c depends on, $Q \vdash \texttt{Chain}(x : a \multimap b)$ implies either $Q \vdash \texttt{Relevant}(x)$ or $a \notin \mathrm{dom}(Q)$.

That is, c would be finalized in Q if it did not depend on some actors outside of Q. We say that a set Q is potentially finalized if every actor in Q is potentially finalized in Q.

Notice that if c is *not* potentially finalized in Q, then c depends on some b which has an irrelevant chain in Q. Such a c is guaranteed not to be finalized in $Q_1 \cup Q_2$, for any Q_2. This means that any actor in Q_i that is not potentially finalized in Q_i can safely be removed from consideration, since it can neither be finalized in Q_i nor $Q_1 \cup Q_2$.

Viewing $\mathrm{dom}(Q)$ as an actor system, we call b a *receptionist* in Q if $b \in \mathrm{dom}(Q)$ and $Q \vdash \texttt{Chain}(x : a \multimap b)$ and $a \notin \mathrm{dom}(Q)$. The following lemmas show that every $c \in Q_i$ depends on a receptionist.

Lemma 5.10. If $a, b \in \mathrm{dom}(Q_i)$ and $Q_i \vdash \texttt{Chain}(x : a \multimap b)$ then $Q_i \vdash \texttt{Relevant}(x)$.

Proof. Immediate from the assumption that Q_i is potentially finalized. QED.

Lemma 5.11. Every $c \in \mathrm{dom}(Q_i)$ depends on some $b \in \mathrm{dom}(Q_i)$ where $a \notin \mathrm{dom}(Q_i)$ and $Q_i \vdash \texttt{Chain}(x : a \multimap b)$.

Proof. Immediate from the assumption that Q_i has no finalized subsets. QED.

Moreover, $c \in \mathrm{dom}(Q_i)$ is finalized in $Q_1 \cup Q_2$ if the receptionists on which it depends are finalized in $Q_1 \cup Q_2$:

Lemma 5.12. Let $a \in \mathrm{dom}(Q_1)$ and $b \in \mathrm{dom}(Q_2)$, without loss of generality. If $Q_1 \cup Q_2 \vdash \texttt{Chain}(x : a \multimap b)$, then b is a receptionist in Q_2.

Proof. Let $(x_1 : a_1 \multimap b), \ldots, (x_n : a_n \multimap b)$ be the chain from b to x in $Q_1 \cup Q_2$. Since $a_n = a \in \mathrm{dom}(Q_1)$, there must be some $m \leq n$ such that $\forall i < m,\ a_i \in \mathrm{dom}(Q_2)$ and $a_m \in \mathrm{dom}(Q_1)$. Hence $Q_2 \vdash \texttt{Chain}(x_m)$ and $a_m \notin \mathrm{dom}(Q_2)$, so b is a receptionist in Q_2. QED.

Lemma 5.13. If $c \in \mathrm{dom}(Q_i)$ depends on a in $Q_1 \cup Q_2$, then either *(1)* c depends on a in Q_i, or *(2)* c depends on a receptionist b in Q_i and b depends on a in $Q_1 \cup Q_2$.

Proof. Let $(x_1 : a_1 \multimap a_2), \ldots, (x_n : a_{n-1} \multimap a_n)$ be the sequence of refobs from a to c. If $\forall i \leq n,\ a_i \in \mathrm{dom}(Q_i)$, then c depends on a in Q_i. Otherwise, let $m < n$ be the greatest index such that $a_m \notin \mathrm{dom}(Q_i)$; then $a_{m+1} \in \mathrm{dom}(Q_i)$ is a receptionist that depends on a and c depends on a_{m+1} in Q_i. QED.

Theorem 5.6. a non-receptionist $b \in \mathrm{dom}(Q_i)$ is finalized in $Q_1 \cup Q_2$ if and only if every receptionist on which b depends in Q_i is finalized in $Q_1 \cup Q_2$.

Proof. If $b \in \mathrm{dom}(Q_i)$ is finalized in $Q_1 \cup Q_2$ then every actor on which it depends must be finalized in $Q_1 \cup Q_2$. Since every actor on which b depends in Q_i is also depended upon in $Q_1 \cup Q_2$, the receptionists in particular must be finalized.

Conversely, let $c \in \mathrm{dom}(Q_i)$ and let every receptionist on which c depends in Q_i be finalized in $Q_1 \cup Q_2$. We show that, if c depends on b in $Q_1 \cup Q_2$ and $Q_1 \cup Q_2 \vdash \texttt{Chain}(x : a \multimap b)$, then $Q_1 \cup Q_2 \vdash \texttt{Relevant}(x)$. By the preceding lemma, there are two cases.

Case 1. $b \in \mathrm{dom}(Q_i)$ and c depends on b in Q_i. If b is a receptionist of Q_i then it is finalized by hypothesis; this implies $Q_1 \cup Q_2 \vdash \texttt{Relevant}(x)$ *a fortiori*. Otherwise, a must be in Q_i, so $Q_i \vdash \texttt{Relevant}(x)$ by Lemma 5.10.

Case 2. c depends on a receptionist b' in Q_i and b' that depends on b in $Q_1 \cup Q_2$. Then b must be finalized because b' is finalized. QED.

Corollary 5.2. a receptionist $b \in \mathrm{dom}(Q_2)$ is finalized in $Q_1 \cup Q_2$ if and only if $Q_1 \cup Q_2 \vdash \texttt{Chain}(x : a \multimap b)$ implies $Q_1 \cup Q_2 \vdash \texttt{Relevant}(x)$ and a is finalized.

This formalizes our intuition that snapshots from "internal actors" of Q_1 and Q_2 are unnecessary. It suffices to combine the snapshots of actors at the "boundary" (e.g. receptionists) with dependency information (i.e. which "boundary" actors depend on which receptionists).

5.3.2 Summaries

Based on the insight from the preceding section, our approach is to compute, for each Q_i, a smaller collage $\tilde{Q}_i$ called its *summary*. These summaries are designed so that (1) all receptionists in Q_i have snapshots in $\tilde{Q}_i$, and (2) a receptionist is finalized in $\tilde{Q}_1 \cup \tilde{Q}_2$ if and only if it is finalized

in $Q_1 \cup Q_2$. We achieve this by removing all facts about the "internal structure" of each Q_i and then adding new refobs to encode the dependency information of Q_i.

Definition 5.10. The *summary* $\tilde{Q}$ of Q is the least collage satisfying the following properties:

For any $x : a \multimap b$ where $a \in \mathrm{dom}(Q)$ and either b is a receptionist or $b \notin \mathrm{dom}(Q)$:

- If $Q(a) \vdash \texttt{Active}(x)$ then $\tilde{Q}(a) \vdash \texttt{Active}(x)$;
- If $Q(a) \vdash \texttt{CreatedUsing}(x, y)$ for some y then $\tilde{Q}(a) \vdash \texttt{CreatedUsing}(x, y)$.
- If $Q(a) \vdash \texttt{Sent}(x, n)$ then $\tilde{Q}(a) \vdash \texttt{Sent}(x, n)$;

For any $x : a \multimap b$ where b is a receptionist:

- If $Q(b) \vdash \texttt{Created}(x)$ then $\tilde{Q}(b) \vdash \texttt{Created}(x)$;
- If $Q(b) \vdash \texttt{Released}(x)$ then $\tilde{Q}(b) \vdash \texttt{Released}(x)$;
- If $Q(b) \vdash \texttt{Received}(x, n)$ then $\tilde{Q}(b) \vdash \texttt{Received}(x, n)$;

If $a, b \in \mathrm{dom}(\tilde{Q})$ and a is a receptionist and b depends on a, then $\tilde{Q}(a) \vdash \texttt{Active}(x)$ and $\tilde{Q}(b) \vdash \texttt{Created}(x)$ for some new, "fake" refob $x : a \multimap b$ with a fresh token x.

By this definition, both $Q_1 \cup Q_2$ and $\tilde{Q}_1 \cup \tilde{Q}_2$ agree about refobs $x : a \multimap b$ where the owner is in Q_1 (resp. Q_2) and the target is in Q_2 (resp. Q_1):

Lemma 5.14. Let $a \in \tilde{Q}_1$ and $b \in \tilde{Q}_2$. For any $x : a \multimap b$, if $Q_1 \cup Q_2 \vdash \texttt{Chain}(x)$ then $\tilde{Q}_1 \cup \tilde{Q}_2 \vdash \texttt{Chain}(x)$.

Proof. Let $Q_1 \cup Q_2 \vdash \texttt{Chain}(x)$ and let $(x_1 : a_1 \multimap b), \ldots, (x_n : a_n \multimap b)$ be the chain from b to x in $Q_1 \cup Q_2$. Notice that b is a receptionist in Q_2. Hence, by definition of the summary, $\tilde{Q}_2(b) \vdash \texttt{Created}(x_1)$. We now show that $\tilde{Q}_1 \cup \tilde{Q}_2 \vdash \texttt{CreatedUsing}(x_i, x_{i+1})$ for each $i < n$.

If $a_i \in \mathrm{dom}(Q_2)$ then $Q_2(a_i) \vdash \texttt{Active}(x_i) \wedge \texttt{CreatedUsing}(x_i, x_{i+1})$. Since b is a receptionist in Q_2, $\tilde{Q}_2(a_i) \vdash \texttt{Active}(x_i) \wedge \texttt{CreatedUsing}(x_i, x_{i+1})$.

Otherwise, $a_i \in \mathrm{dom}(Q_1)$ and therefore $Q_1(a_i) \vdash \texttt{Active}(x_i) \wedge \texttt{CreatedUsing}(x_i, x_{i+1})$. Since $b \notin \mathrm{dom}(Q_1)$, $\tilde{Q}_1(a_i) \vdash \texttt{Active}(x_i) \wedge \texttt{CreatedUsing}(x_i, x_{i+1})$. QED.

Lemma 5.15. Let $a \in \tilde{Q}_1$ and $b \in \tilde{Q}_2$ such that $Q_1 \cup Q_2 \vdash \texttt{Chain}(x : a \multimap b)$. Then $Q_1 \cup Q_2 \vdash \texttt{Relevant}(x)$ if and only if $\tilde{Q}_1 \cup \tilde{Q}_2 \vdash \texttt{Relevant}(x)$.

Proof. Let $Q_1 \cup Q_2 \vdash \texttt{Relevant}(x)$. Then there exists n such that $Q_1(a) \vdash \texttt{Active}(x) \wedge \texttt{Sent}(x, n)$ and $Q_2(b) \vdash \texttt{Received}(x, n)$. Since $a \in \mathrm{dom}(Q_1)$ and $b \notin \mathrm{dom}(Q_1)$, $\tilde{Q}_1(a) \vdash \texttt{Active}(x) \wedge \texttt{Sent}(x, n)$. Since b is a receptionist in Q_2, $\tilde{Q}_2(b) \vdash \texttt{Received}(x, n)$.

Conversely, let $\tilde{Q}_1 \cup \tilde{Q}_2 \vdash \texttt{Relevant}(x)$. Then there exists n such that $\tilde{Q}_1(a) \vdash \texttt{Active}(x) \wedge \texttt{Sent}(x, n)$ and $\tilde{Q}_2(b) \vdash \texttt{Received}(x, n)$. From the definition of $\tilde{Q}_1$ and $\tilde{Q}_2$, we must also have $Q_1(a) \vdash \texttt{Active}(x) \wedge \texttt{Sent}(x, n)$ and $Q_2(b) \vdash \texttt{Received}(x, n)$. QED.

The following lemma formalizes our understanding that the refobs in $\tilde{Q}$ serve to abbreviate the dependency information of Q:

Lemma 5.16. Let $a, b \in \tilde{Q}_1 \cup \tilde{Q}_2$. If $\tilde{Q}_1 \cup \tilde{Q}_2 \vdash \texttt{Chain}(x : a \multimap b)$ then either:

1. $Q_1 \cup Q_2 \vdash \texttt{Chain}(x : a \multimap b)$; or
2. Both a, b are in some Q_i and b depends on a in Q_i.

Proof. Let $(x_1 : a_1 \multimap b), \ldots, (x_n : a_n \multimap b)$ be the chain from b to x in $\tilde{Q}_1 \cup \tilde{Q}_2$. Then for some Q_i, we must have $\tilde{Q}_i(b) \vdash \texttt{Created}(x_1)$.

By construction of $\tilde{Q}_i$, this could either be the result of (1) b being a receptionist in Q_i or (2) a_1 being a receptionist in Q_i and b depending on a_1 in Q_i.

Case 1. In this case, we must have $Q_i(b) \vdash \texttt{Created}(x_1)$. Moreover, by construction of $\tilde{Q}_1$ and $\tilde{Q}_2$, $(\tilde{Q}_1 \cup \tilde{Q}_2)(a_j) \vdash \texttt{CreatedUsing}(x_j, x_{j+1})$ implies $(Q_1 \cup Q_2)(a_j) \vdash \texttt{CreatedUsing}(x_j, x_{j+1})$ for every $j < n$. Hence $Q_1 \cup Q_2 \vdash \texttt{Chain}(x)$.

Case 2. In the latter case, the chain can only have length 1 because x_1 is a "fake" refob. Hence $x = x_1$ and $a_1 = a$, so indeed b depends on a in Q_i. QED.

Corollary 5.3. If b depends on a in $\tilde{Q}_1 \cup \tilde{Q}_2$ then b depends on a in $Q_1 \cup Q_2$.

Conversely, we now show that all the important dependencies have been preserved - namely, which actors depend on which receptionists.

Lemma 5.17. Let $a, b \in \tilde{Q}_1 \cup \tilde{Q}_2$ and let a be a receptionist in Q_1. If b depends on a in $Q_1 \cup Q_2$ then b depends on a in $\tilde{Q}_1 \cup \tilde{Q}_2$.

Proof. Let $(x_1 : a_1 \multimap a_2), \ldots, (x_n : a_{n-1} \multimap a_n)$ be the sequence of refobs from a to b.

If $n = 0$ then the lemma is trivially satisfied.

If $\forall i \leq n,\ a_i \in \text{dom}(Q_1)$ then, by construction of $\tilde{Q}_1$, there exists $x : a \multimap b$ such that $\tilde{Q}_1 \vdash \texttt{Chain}(x)$.

For the general case, the sequence of refobs may pass between Q_1 and Q_2 multiple times. We partition the sequence $x_1, \ldots, x_n$ into a sequence of "runs" $\vec{x}_1, \ldots, \vec{x}_m$, such that:

1. For each refob $(x : a \multimap b)$ in the first run $\vec{x}_1$, the owner a is in Q_1; for each refob $(x : a \multimap b)$ in the second run $\vec{x}_2$, the owner a is in Q_2; for each refob $(x : a \multimap b)$ in the third run $\vec{x}_3$, the owner a is in Q_1; and so on.
2. The concatenation of $\vec{x}_1, \ldots, \vec{x}_m$ is $x_1, \ldots, x_n$.

For each run $\vec{x}_i$, we denote the owner of the first refob as b_i and the target of the last refob as c_i. Notice that every b_i is a receptionist. Hence, by construction of $\tilde{Q}_1$ and $\tilde{Q}_2$ there is, for each run $\vec{x}_i$, a refob $y_i : b_i \multimap c_i$ such that $\tilde{Q}_1 \cup \tilde{Q}_2 \vdash \texttt{Chain}(y_i)$. Then the sequence of refobs $y_1, \ldots, y_m$ witnesses the fact that b depends on a in $\tilde{Q}_1 \cup \tilde{Q}_2$. QED.

Finally, we can show that summaries are sound and complete for the intended purpose of finding finalized receptionists.

Theorem 5.7. Let $c \in \tilde{Q}_1 \cup \tilde{Q}_2$. Then c is finalized in $\tilde{Q}_1 \cup \tilde{Q}_2$ if and only if c is finalized in $Q_1 \cup Q_2$.

Proof. Let c be finalized in $\tilde{Q}_1 \cup \tilde{Q}_2$. We show that, if c depends on some b in $Q_1 \cup Q_2$ and $Q_1 \cup Q_2 \vdash \texttt{Chain}(x : a \multimap b)$, then $Q_1 \cup Q_2 \vdash \texttt{Relevant}(x)$.

If a, b are both in some Q_i, then $Q_i \vdash \texttt{Relevant}(x)$ because Q_i is potentially finalized.

Otherwise, let $a \in \text{dom}(Q_1)$ and $b \in \text{dom}(Q_2)$, without loss of generality. Since $Q_1 \cup Q_2 \vdash \texttt{Chain}(x)$, Lemma 5.14 implies $\tilde{Q}_1 \cup \tilde{Q}_2 \vdash \texttt{Chain}(x)$. Since b is a receptionist, $b \in \tilde{Q}_2$. Since $b, c \in \tilde{Q}_1 \cup \tilde{Q}_2$ and c depends on b in $Q_1 \cup Q_2$, c must also depend on b in $\tilde{Q}_1 \cup \tilde{Q}_2$ by Lemma 5.17. Hence, since c is finalized, $\tilde{Q}_1 \cup \tilde{Q}_2 \vdash \texttt{Relevant}(x)$. This implies, by Lemma 5.15, $Q_1 \cup Q_2 \vdash \texttt{Relevant}(x)$.

Conversely, let c be finalized in $Q_1 \cup Q_2$. We show that, if c depends on some b in $\tilde{Q}_1 \cup \tilde{Q}_2$ and $\tilde{Q}_1 \cup \tilde{Q}_2 \vdash \texttt{Chain}(x : a \multimap b)$, then $\tilde{Q}_1 \cup \tilde{Q}_2 \vdash \texttt{Relevant}(x)$. By Lemma 5.16, there are two cases:

Case 1: b depends on a in Q_i; the refob x is a "fake" reference created in the process of constructing $\tilde{Q}_i$. Then $\tilde{Q}_i(a) \vdash \texttt{Active}(x)$ and $\tilde{Q}_i \vdash \texttt{Sent}(x, 0) \wedge \texttt{Received}(x, 0)$ by construction.

Case 2: $Q_1 \cup Q_2 \vdash \texttt{Chain}(x)$. Since c is finalized and c depends on b in $Q_1 \cup Q_2$, we must have $Q_1 \cup Q_2 \vdash \texttt{Relevant}(x)$. Then, since $b, c \in \tilde{Q}_1 \cup \tilde{Q}_2$, by Lemma 5.15, it follows that $\tilde{Q}_1 \cup \tilde{Q}_2 \vdash \texttt{Relevant}(x)$. QED.

Hence a pair of garbage collectors can find quiescent actors in $Q_1 \cup Q_2$ by:

1. Garbage collecting all finalized actors in each Q_i;
2. Removing all actors not potentially finalized in each Q_i;
3. Computing the summary of the remaining collage and exchanging it with their partner;
4. Removing all potentially unfinalized snapshots in the pair of summaries $\tilde{Q}_1 \cup \tilde{Q}_2$ (optionally using the `Unreleased` heuristic from Section 5.2.2);
5. Garbage collecting all actors in Q_i that are reachable from a finalized receptionist in $\tilde{Q}_1 \cup \tilde{Q}_2$.

Alternatively, a set of garbage collectors could send their summaries to a parent collector, which uses the summaries to compute the finalized receptionists and sends this set to each child collector.

Part II

Fault-Recovering Actor GC

Part I presented *PRL*: an actor GC based on reference listing. PRL is sound despite message reordering and complete when actors always eventually take new snapshots. However, PRL loses its completeness property if nodes crash so that actors can no longer take snapshots, or if messages are dropped. PRL therefore relies on the underlying actor framework to guarantee nodes do not crash and messages are not dropped.

We now turn our attention to actor GC for *fault-exposing* actor frameworks, such as Akka [26] and Erlang [57]. Instead of masking failures, these frameworks give programmers the tools to detect faults and implement custom fault-handling policies [57]. For example, both Akka and Erlang provide a mechanism called *monitoring*: if a monitors b, then a will be notified when b appears to have failed. Erlang and Akka also allow programmers to set timeouts for detecting other kinds of failures: if a sets a timeout and the time limit expires before a response was received, then a message may have been dropped or some other error may have occurred.

Chapter 6 presents an abstract actor model for fault-exposing actor frameworks with monitoring, timeouts, crash failures, and message omission failures. We show that the definition of actor garbage in Part I is no longer adequate in this new model and give two new definitions: one conservative definition which we call *strong quiescence*, and another less conservative definition which we call *weak quiescence*. Lastly, we compare the failure detection semantics of Erlang and Akka, and argue that complete actor GC is impossible with Erlang-style semantics. The fault model we develop in this chapter is the first formalization of an Akka-style fault model.

Chapter 7 presents a sound and complete actor GC for the new fault model. This is the first cyclic actor GC to detect garbage that results from dropped messages or faulty actors. The new GC is presented in five stages. The first stage adapts a classic algorithm for global termination detection [46] for detecting actor garbage in fixed topologies. The second stage builds on the first stage, adding support for dynamic topologies. The third stage adds support for timeouts and monitoring. The fourth stage adds support for dropped messages and node failures. The fifth stage presents efficient data structures for identifying actor garbage and for recovering from faults.

Chapter 8 presents an implementation of the novel actor GC for Akka. We show how actor snapshots can be divided into fixed-size *entries* to reduce memory allocation, and how garbage collectors can exchange *delta graphs* to obtain eventually consistent collages. We conclude in Chapter 9 with an evaluation of our approach using the Savina actor benchmark suite [59] and a tunable distributed benchmark.

6

Fault Model

This chapter describes an abstract model for fault-prone actor systems. Our model is motivated by the semantics of Distributed Erlang (which has been formally specified [61, 62]) and Akka Cluster (which has only been described informally [63]). Our model explicitly incorporates *nodes* (Figure 6.1) at which actors are located; allows messages to be dropped; allows nodes to crash spontaneously; and allows nodes and messages to experience unbounded delays. Our model also incorporates distributed fault handling, based on Akka's cluster membership service [63]—a relaxed version of cluster membership in *Isis* [64]—which allows healthy nodes to *exile* suspected-faulty nodes, i.e. remove them from the cluster.

Our model is the first formalization of Akka-style fault detection and monitoring. The model accounts for several complicating factors, including:

1. Healthy nodes cannot exile suspected-faulty nodes from the cluster all at once; each healthy node asynchronously makes the decision to close its connection to the suspected-faulty node.
2. Healthy nodes may be exiled because they were misdiagnosed as crashed (e.g. if the node was slow to respond to a message).
3. Healthy nodes may crash while another (possibly healthy) node is being exiled from the system.
4. Actors may continue to receive messages from inverse acquaintances on exiled nodes if those messages were placed in its mailbox before the nodes were exiled.
5. Actors on healthy nodes monitoring actors on exiled nodes will eventually be notified that the monitored actors have been exiled.
6. Actors themselves can *halt* (e.g. because they threw an uncaught exception), causing the actors that monitor them to be notified.

Criteria 1–3 above are also addressed in the *Isis* group membership protocol [64], but Criteria 4–6 are unique to our model. We show that, despite all these complications, it is safe to reason about execution paths as if every healthy node *simultaneously* exiled the suspected-faulty nodes

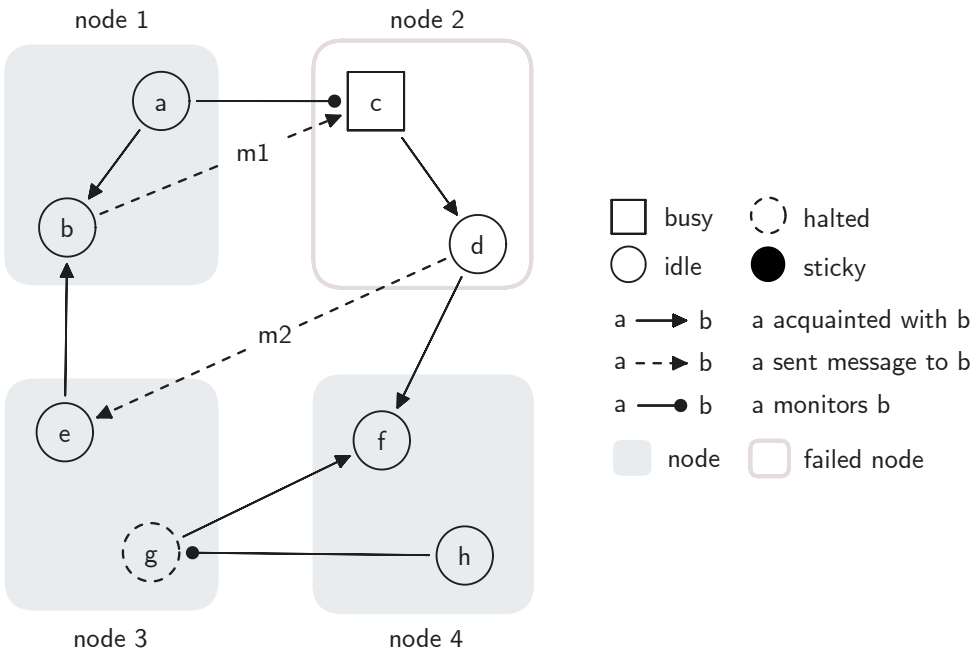


Figure 6.1: An example configuration with nodes, halted actors, and monitoring. Actors c and d are on a crashed node, causing them to "freeze" in their current state. Actor g has halted (perhaps because it threw an uncaught exception). If actors a and h do not fail, they will eventually be notified about the failures of c and g, respectively.

from the cluster. We exploit this property to develop a simplified version of the model, with respect to which we will prove the correctness of the actor GC in Chapter 7.

Sections 6.1 to 6.3 motivate the basic concepts in the model. Section 6.4 establishes notation for reasoning about execution paths in the model. Section 6.5 establishes important results about execution paths with faults and Section 6.6 defines actor garbage. The formal specification of the fault model is relegated to Appendix A.1.

6.1 Nodes

Our new model extends the model of Part I with the notion of *nodes*. Every actor is located on some node and never migrates from one node to another. A node can spontaneously *crash* or *fail*, causing all actors on the node to become frozen and unable to take additional actions. In a configuration κ, nodes that have not failed are said to be *healthy*. In an execution path σ, nodes are said to be *faulty* if they fail at some point in time and *non-faulty* if they never fail in σ.

When an actor on node N sends a message m to an actor on another node N', the message undergoes two stages. Initially, m is said to be *in-flight* from N to N'. In-flight messages can be reordered, dropped, or delayed for arbitrary lengths of time. If m arrives at N', we say that m

has been *admitted* to N'. Admitted messages can also be reordered, dropped, or delayed before being delivered to the recipient actor[2]. The semantic distinction between in-flight and admitted messages arises when one node suspects that another node is faulty.

Through some unspecified mechanism (e.g., ϕ-accrual [67]), node N' may begin suspecting that N has failed. When N' is confident enough that N has failed, N' can make the irrevocable decision to *shun* N. Once N' has shunned N, messages from N will no longer be admitted to N'. Subsequently, every other node N'' must eventually either shun N as well or else be shunned by N'. If some group of nodes $\mathcal{G}_1$ all shun the remaining nodes $\mathcal{G}_2$, we say that $\mathcal{G}_1$ has *exiled* $\mathcal{G}_2$. If this happens then, from the perspective of $\mathcal{G}_1$, it looks as if all the actors on $\mathcal{G}_2$ have halted and all in-flight messages from $\mathcal{G}_2$ were dropped. We formalize this observation in Theorem 6.1.

In practice, "split-brain scenarios" [63] may arise, where two groups of nodes $\mathcal{G}_1$ and $\mathcal{G}_2$ exile one another. Akka applications handle these scenarios by shutting down one of the two partitions [68] or by ignoring the issue, allowing both groups of nodes to operate independently. In any case, split-brain scenarios do not pose a problem for actor GC: nodes in $\mathcal{G}_1$ may collect garbage as if the nodes in $\mathcal{G}_2$ crashed, and nodes in $\mathcal{G}_2$ may collect garbage as if the nodes in $\mathcal{G}_1$ crashed. Without loss of generality, when $\mathcal{G}_1$ exiles $\mathcal{G}_2$ we may "take the perspective" of $\mathcal{G}_1$ and assume that the actors on $\mathcal{G}_2$ have halted.

In Appendix A.1, we model the basic transitions of actors with the following events:

- *Idle*(a): A busy actor becomes idle.
- *Spawn*(a, b, N): A busy actor a spawns actor b onto node N, assuming neither N nor a's node have shunned one another.
- *Deactivate*(a, b): Busy actor a no longer needs to send messages to b, so a removes all references to b from its local state.
- *Send*(a, b, m): Busy actor a sends message m to one of its acquaintances b. The message may contain references to a's acquaintances.
- *Receive*(a, m): Idle actor a receives an admitted message m, becoming busy.

We also add the following transitions to model the behavior of nodes and the underlying network:

- *Admit*(m): An in-flight message m from a non-shunned node is admitted to its destination.
- *Drop*(m): An in-flight or admitted message m is dropped.
- *Shun*(N_1, N_2): Node N_1 is shunned by non-faulty node N_2.

Once N_1 has been shunned by N_2, messages from N_1 can no longer be admitted and actors on N_1 cannot spawn children on N_2 (nor vice versa). In addition, once an actor's node is exiled, it can

[2]Our model is more lax than the Erlang model of Svensson and Fredlund [61], which only accounts for dropped messages between nodes and guarantees that messages are not delivered out-of-order. In Akka, messages between actors on a single machine can be dropped if actors use bounded mailboxes or a stack overflow occurs [65] but messages will not be delivered out-of-order. Other actor frameworks, such as SALSA [66], do allow messages to be delivered out-of-order.

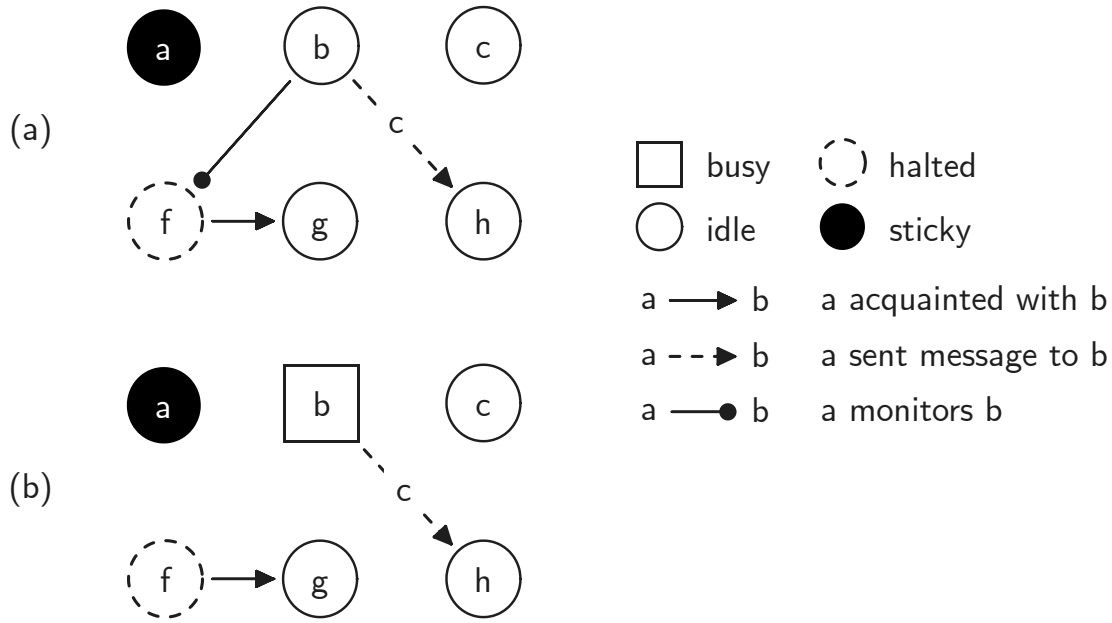


Figure 6.2: Depiction of monitoring in our model. In (a), actor b monitors actor f that has halted. In (b), b is notified that f halted and becomes busy.

no longer perform any events such as spawning or sending messages.

In Section 6.5, we prove that all failed nodes are eventually exiled, but healthy nodes may also be exiled erroneously—for instance because a temporary network partition caused the healthy node to be unreachable. Compared to *Isis* [64], the guarantees in our model are more relaxed: if N has shunned N' then in-flight messages from N' will not be admitted, but already-admitted messages can still be delivered to their target actor.

There are several complicating factors that we omit from the model. We do not consider nodes dynamically joining the configuration. We also do not consider actors migrating from one node to another; once an actor is spawned, it has a specific node that is its "home". However, actors on a node N can spawn actors onto another node N' (provided that N has not shunned N', nor vice versa). Venkatasubramanian and Talcott showed that actor migration can be *simulated* in a model with fixed actor locations [51] although their model does not account for faults.

6.2 Monitoring

Actors in our model can fail in two ways: (1) by *halting* while processing a message (e.g. throwing an exception or being manually garbage collected), or (2) by being *exiled*, i.e. being located on an exiled node. An actor that has not failed is said to be *healthy*. We say that an actor is *faulty* in an execution path if it fails at some point in time; otherwise it is *non-faulty*.

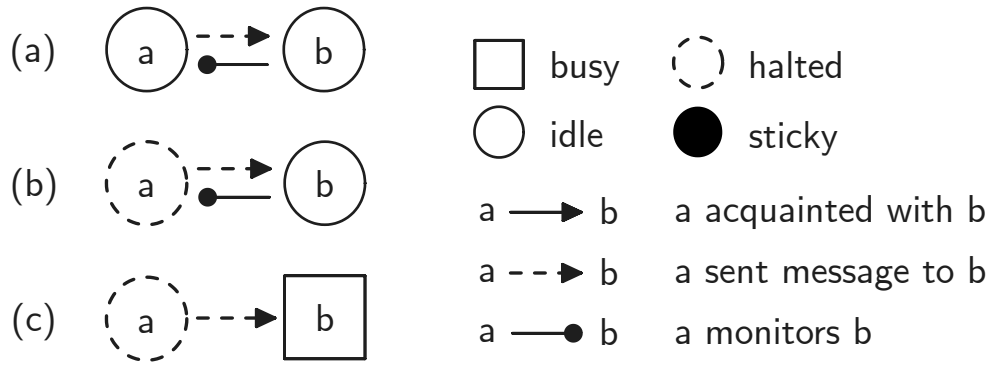


Figure 6.3: Depiction of failure signals arriving out-of-order with messages. In (a), actor b has an undelivered message from a and also monitors a. In (b), actor a halts. In (c), b is notified that a halted before b receives the message.

A busy actor a can request to be notified when an acquaintance b fails by *monitoring* b. If a is a healthy actor monitoring b and b has failed and a is always eventually idle, then a will eventually be notified that b has failed and become busy, as shown in Figure 6.2. However, we do not assume that this notification arrives after all messages from b to a have been delivered, as shown in Figure 6.3. This flexibility is consistent with Akka's message delivery guarantees [65].

In Appendix A.1, we model monitoring with four events:

- $Halt(a)$: The busy actor a halts.
- $Monitor(a, b)$: The busy actor a begins monitoring its acquaintance b.
- $Notify(a, b)$: The idle actor a is notified that b failed, causing a to become busy.
- $Unmonitor(a, b)$: The busy actor a stops monitoring b.

6.2.1 *Romeo-and-Juliet Problems*

Although Akka and Erlang both implement monitoring, the two frameworks have different semantics when healthy nodes are incorrectly marked as faulty. This is depicted in Figure 6.4, which adapts a well-known counterexample [69].

In Figure 6.4, actor j is inaccurately marked as faulty, causing a monitoring actor r to be notified and to halt. Subsequently, actor j is notified that r halted, causing j to halt. This execution path is possible in Erlang, which implements monitoring with unreliable failure detectors[3]. In contrast, this execution path is not possible in Akka because of Akka's group membership policy [64]: suspected-faulty nodes are exiled from the cluster, and monitoring actors are not notified until those nodes have been successfully exiled. However, it is possible for old messages from exiled

[3]Note that failure detectors in practice can either be *inaccurate* (actors on healthy nodes become exiled) or *incomplete* (actors on failed nodes never become exiled), not both [70].

actors to arrive at their destinations, as shown in Figure 6.3.

In Section 6.6, we argue that actors should be garbage collected if they are only potentially reachable by failed actors. But for an actor framework that uses Erlang-style monitoring to detect failures, we can never be sure if actors that *appear* faulty are truly faulty. To preserve soundness, an actor GC for such a model would need to assume that any apparently-failed actor could come back to life. By employing a group membership policy as in Akka, an actor GC can collect such garbage without violating soundness.

6.3 Sticky Actors and Timeouts

In Part I, we modeled an open system in which garbage-collected actors could communicate with actors in the outside world. Once an actor exposed its address to the outside world, that actor could no longer be garbage collected automatically. In our new model, we take a simplified approach: busy actors can *register* as sticky, allowing them to spontaneously wake up from idle state as if they received a message from an external actor. Busy sticky actors can also *unregister* as sticky to stop receiving such wakeup messages. Sticky actors generalize the notion of "root actors" used in other actor GCs [17].

Sticky actors are a simple way to model several patterns in practical Akka systems, such as timeouts. The drawback, compared to the model in Part I, is that an actor cannot register one of its *acquaintances* as a sticky actor by sending the acquaintance's address to an external actor.

In Appendix A.1, we model sticky actors with three events:

- *Register*(a): The busy actor a registers as a sticky actor.
- *Wakeup*(a): The idle sticky actor a becomes busy.
- *Unregister*(a): The busy sticky actor a unregisters as a sticky actor.

6.4 Configurations and Executions

The fault model is formalized as a TLA+ specification [71] in Appendix A.1.[4] The specification defines a labeled transition system (LTS) on configuration κ. The specification defines:

- An *Init* predicate, which defines the initial configuration κ_0; and
- A *Next* predicate, which defines the transition relation $\kappa \rightarrow \kappa'$ on configurations. The predicate is composed of events *Idle(a), Send(a,b,m), ...* that can occur in an execution path.

[4]The sources are available at https://github.com/dplyukhin/uigc-spec.

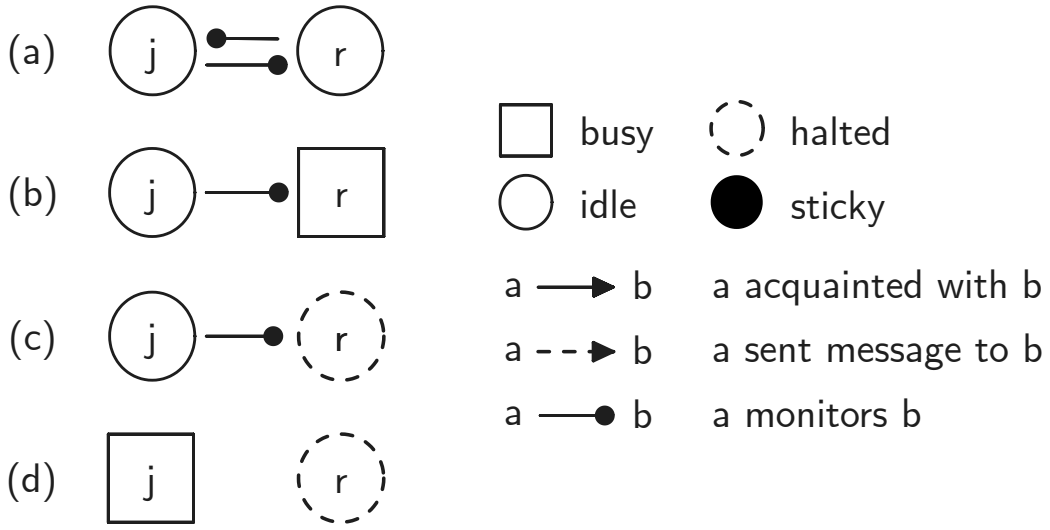


Figure 6.4: Depiction of the Romeo-and-Juliet problem, which occurs in Erlang-style models but not Akka-style models. In (a), actors r and j monitor one another. In (b), j is misdiagnosed as faulty, causing r to become busy. In (c), r halts. In (d), j becomes busy because r halted.

We write $\kappa \xrightarrow{e} \kappa'$ if κ' can be obtained from κ via event e. An event e is *legal* in κ if there exists κ' such that $\kappa \xrightarrow{e} \kappa'$. An execution path is a (possibly infinite) sequence of events $e_1, e_2, \ldots$ such that $\kappa_0 \xrightarrow{e_1} \kappa_1 \xrightarrow{e_2} \kappa_2 \xrightarrow{e_3} \ldots$. We then refer to κ_t as the *configuration at time* t.

Given a finite execution path σ, we say that the sequence of events σ' is a *legal extension* if the concatenation $\sigma \cdot \sigma'$ is an execution path. We write $cfg(\sigma)$ to denote the configuration resulting from finite σ. We say that configuration κ is *reachable* if there exists a finite execution path σ where $\kappa = cfg(\sigma)$. An execution path is *complete* if it is infinite or ends with a configuration κ for which there are no legal events.

We say that a property holds at time t if it holds in κ_t. We also use interval notation $[t, t']$, $[t, t')$, $(t, t']$, (t, t') with the usual meaning; for example, a property holds in the interval (t, t') if it holds at times $t + 1, \ldots, t' - 1$.

6.5 Faulty Execution Paths

Assume that every execution path σ has at least one non-faulty node, i.e. a node that never fails. We also assume the following fairness properties:

(FP1) Let σ be an execution path in which N_1 is non-faulty and N_2 is faulty. Then eventually N_1 shuns N_2.

(FP2) Let σ be an execution path in which N_1 has shunned N_2. If N_1 is non-faulty then every node N eventually shuns N_2 or is shunned by N_1.

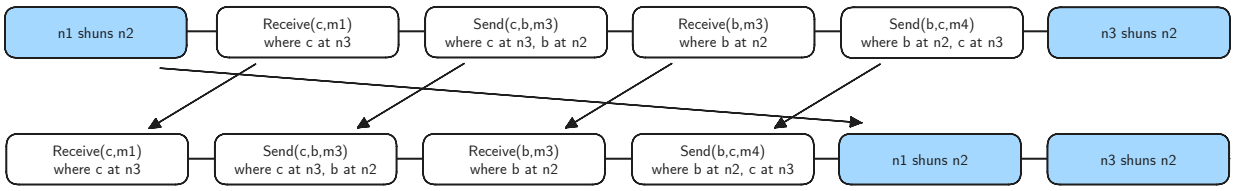


Figure 6.5: An example of two equivalent execution paths, where the bottom execution path is obtained by delaying the *Shun* event.

Note that nodes can fail at any time and healthy nodes can be shunned. For example, in a configuration with three nodes N_1, N_2, N_3, node N_1 can shun N_2 and then immediately fail. Alternatively, N_1 and N_2 can shun one another and then N_3 fails. Despite these complications, we prove that the shunning mechanism is well-behaved.

Lemma 6.1. Every faulty node is eventually exiled.

Proof. Immediate from FP1. QED.

Lemma 6.2. If N_1 has shunned N_2 then eventually N_2 will be exiled or N_1 will be exiled.

Proof. Consider any complete execution path σ in which N_1 shuns N_2. If N_1 fails in σ then eventually N_1 will be exiled, as shown in the preceding theorem. Let us therefore assume that N_1 never fails in σ.

For any prefix of σ, let $\mathcal{G}_1$ be the nodes that have not been shunned by N_1, let $\mathcal{G}_2$ be the set of nodes that have been shunned by N_1, and let $\mathcal{G}_2^*$ be the set of nodes shunned by all of $\mathcal{G}_1$. Notice that $N_1 \in \mathcal{G}_1$ and $\mathcal{G}_2^* \subseteq \mathcal{G}_2$ for any prefix of σ.

Let σ_0 be a prefix of σ in which N_1 shuns N_2. In σ_0, $N_2 \in \mathcal{G}_2$.

Let σ_1 be an extension of σ_0 in which every node either shuns N_2 or is shunned by N_1. This extension must exist, by FP2.

For each $i > 1$, we define σ_{i+1} as follows:

1. If $\mathcal{G}_2 = \mathcal{G}_2^*$ in σ_i, then $\sigma_{i+1} = \sigma_i$;
2. Otherwise, let $N_3 \in \mathcal{G}_2 \setminus \mathcal{G}_2^*$ in σ_i. Let σ_{i+1} be an extension of σ_i in which each node either shuns N_3 or is shunned by N_1. Every node in $\mathcal{G}_1$ that does not shun N_3 is moved into $\mathcal{G}_2$, and the remaining nodes in $\mathcal{G}_1$ all shun N_3—so N_3 is moved into $\mathcal{G}_2^*$.

Since $\mathcal{G}_2^*$ strictly increases between each distinct σ_i, σ_{i+1} and the set of nodes is finite, there must exist some finite n where $\mathcal{G}_2 = \mathcal{G}_2^*$ in σ_n.

By definition, $\mathcal{G}_2 = \mathcal{G}_2^*$ in σ_n implies that every node in $\mathcal{G}_1$ has shunned every node in $\mathcal{G}_2$. Since $N_2 \in \mathcal{G}_2$, this implies that N_2 has been exiled. QED.

Next, we show that every complete execution path with shunning is equivalent to an execution path in which nodes are exiled atomically. The result is due to the following lemma, which states that a $Shun(N_1, N_2)$ event at time t can always be "pushed further back" to occur at time $t + 1$ instead—unless N_1 became exiled at time t. Intuitively, this is because an execution path in which $Shun(N_1, N_2)$ occurs is indistinguishable from an execution path with a network partition, where messages from N_1 are delayed or dropped. The execution paths become distinguishable once N_1 is exiled because actors on N_2 can be notified actors on N_1 have failed.

Lemma 6.3. Let κ be a reachable configuration, let $e = Shun(N_1, N_2)$, and let e' be an arbitrary event. If there exist κ', κ'' such that $\kappa \xrightarrow{e} \kappa' \xrightarrow{e'} \kappa''$ and N_1 is not exiled in κ', then there exists κ^* such that $\kappa \xrightarrow{e'} \kappa^* \xrightarrow{e} \kappa''$.

Proof. By inspecting the definitions of every event in Appendix A.1, one can see that the *Shun* event can safely be "swapped" with any event e' in an execution path, except if e' is a *Notify* event. Specifically, if $e' = Notify(a, b)$ where b is on a node that became exiled along with N_1, then there is a causal relationship between e and e'. Thus, for any e' where N_1 is not exiled in κ', the two events can be swapped. QED.

The above result relies on the fact that, if actor a monitors actor b and a's node shuns b's node, then a is not notified that b has failed until b's node is *exiled*. If a were notified as soon as a's node has *shunned* b's node, then the result would no longer hold.

Theorem 6.1. Let σ be an execution path in which $\mathcal{G}_1$ has been exiled by $\mathcal{G}_2$. Then σ is equivalent to an execution path σ' in which the nodes $\mathcal{G}_1$ were *atomically* exiled by $\mathcal{G}_2$: that is, $\sigma' = \sigma_1 \cdot \sigma_2 \cdot \sigma_3$ where σ_2 consists only of $Shun(N_1, N_2)$ events for $N_1 \in \mathcal{G}_1$ and $N_2 \in \mathcal{G}_2$.

Proof. By Lemma 6.3, σ' can be obtained by delaying all *Shun* events so that they occur as one atomic block. QED.

As a consequence of this theorem, we can simplify the model in Appendix A.1 by replacing the *Shun* event with an *Exile* event in which all nodes of $\mathcal{G}_2$ are shunned at once. We therefore do not need to distinguish between healthy exiled nodes and failed exiled nodes.

6.6 Actor Garbage

In Part I, we conservatively defined garbage actors to be actors that are both blocked and only potentially reachable by blocked actors. In our new model, this definition is both unsound and incomplete:

- The definition is *unsound* because an actor a may be blocked and only potentially reachable by blocked actors, but become busy because a monitors another actor b that halted. This is exemplified in Figure 6.1 by actor a, which is blocked and unreachable by any actor, but will become busy once notified that c has been exiled.
- The definition is *incomplete* because an actor a may be blocked and only potentially reachable by halted or exiled actors; such actors are not necessarily blocked, but they are *effectively* blocked because they cannot send messages to a. This is exemplified in Figure 6.1 by actor f, which is only potentially reachable by halted actor g and exiled actor d.

In this section, we generalize the definition of quiescence for our new model. We also discuss why the resulting definition may be too conservative a criterion for real-world applications, and develop a more liberal definition of actor garbage: *weak quiescence*.

6.6.1 Deliverable Messages

Before defining actor garbage in our new model, we must revisit the basic concepts of unblocked actors and potential acquaintances.

Recall that an actor a is unblocked in configuration κ if a is currently busy or there is a message in κ that could cause a to become busy. In Part I, any undelivered message addressed to a could cause a to become busy. In our new model, this is no longer the case. For instance, if the message is in-flight and a's node has shunned the sender's node, then the message will never be admitted and therefore never be delivered to its target. From the perspective of actor GC, such "undeliverable" messages may as well not exist.

This leads us to the notion of *deliverable* messages, which in turn is used to redefine blocked actors and potential acquaintances. These terms are formalized in Appendix A.1 by the sets $Blocked$, $Unblocked$, and the relations $pacqs(a)$, $piacqs(a)$.

Definition 6.1. An undelivered message is *deliverable* if it has not been dropped and either:

1. The message has already been admitted; or
2. The message is in-flight and the recipient's node has not shunned the sender's node.

Definition 6.2. An actor a is *blocked* if a is idle and has no deliverable messages. An actor that is busy or has deliverable messages is said to be *unblocked*.

Definition 6.3. 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. We also say that a is a *potential inverse acquaintance* of b.

6.6.2 Potentially Unblocked Actors

As in Part I, we assume the *live unblocked actor principle* [18]:

1. Any busy actor can affect the world, so busy actors are always live (i.e. non-garbage); and
2. Actors that are not busy cannot affect the world except by becoming busy.

Based on this principle, an actor is garbage if and only if it is not busy and there is no future configuration in which it can become busy.

Definition 6.4. Actor a is *live* in σ if a is healthy and there exists an extension of σ in which a is busy. Otherwise, a is *garbage* in σ.

In our model, a healthy actor a can become busy if and only if one of the following can occur:

1. a receives a message from some other actor;
2. a is a sticky actor and spontaneously wakes up;
3. a monitors an actor that halted; or
4. a monitors an actor on a different node that was exiled.

We remark that Definition 6.4 implies that failed actors are always garbage; case (1) can only occur if a is unblocked or a potential acquaintance of a non-garbage actor; case (3) can only occur if the monitored actor can become busy, i.e. the monitored actor is non-garbage; and case (4) can always occur if a monitors an actor on a different node, because nodes can always spontaneously halt.

We now characterize the live and garbage actors in terms of *configurations*.

Definition 6.5. A healthy a is *potentially unblocked* if one of the following holds:

1. a is unblocked;
2. a is a sticky actor;
3. There exists another actor b that is potentially unblocked and a is a potential acquaintance of b;
4. There exists another actor b that has failed or is potentially unblocked and a is monitoring b; or
5. There exists another actor b on a different node from a and a is monitoring b.

Theorem 6.2. If a is potentially unblocked then a is live.

Proof. Let a be potentially unblocked at the end of finite execution path σ. We show there exists an extension σ' such that a is busy in $\sigma \cdot \sigma'$ by induction on the definition of potentially unblocked actors:

- If a is unblocked then either a is busy or there exists an deliverable message to a. The first case immediately implies a is live. In the second case, a can become busy in the next step by receiving a message.

- If a is a sticky actor then a can become busy by receiving a message from an external actor.
- Let b be another actor that is potentially unblocked. By the induction hypothesis, there is an extension σ'' such that b is busy in $\sigma \cdot \sigma''$.
  - If b is potentially acquainted with a then there is an extension σ''' such that b is busy and has a reference to c in $\sigma \cdot \sigma'' \cdot \sigma'''$. At this point b can send a message to a, causing a to become unblocked.
  - If a is monitoring b then b may halt in $\sigma \cdot \sigma''$, causing a to become unblocked.
- If a monitors an actor b on a different node, then a can become busy if b is exiled and a is notified about the failure.

QED.

Theorem 6.3. If a is live then a is potentially unblocked.

Proof. Assume that a is live in σ at time t. We show that a is potentially unblocked at time t.

First we show, for all times t and actors a, if a is potentially unblocked at $t+1$ then a was also potentially unblocked at t or a did not exist at t. This follows by induction on the definition of potentially unblocked actors:

- If a is unblocked at time $t+1$ then either (1) a is unblocked at time t, (2) some busy b sent a message to a at time t, (3) a monitors an actor b that failed at time t, or (4) a was a sticky actor at time t.

  Cases (1) and (4) immediately imply that a was potentially unblocked at t.

  Case (2) implies that b was acquainted with a and live at time t, implying that a was potentially unblocked at time t.

  Case (3) implies that b was busy (and therefore live) at t or b was on a remote node and became exiled at $t+1$. Both cases imply that a was potentially unblocked at t.
- If a is a sticky actor at time $t+1$ and was not a sticky actor at time t, then a must have been busy (and therefore unblocked) at time t in order to register.
- Let b be another potentially unblocked actor at $t+1$. By the induction hypothesis, b was potentially unblocked at t.
  - If b is a potentially unblocked potential inverse acquaintance at $t+1$ and not at t, then b was not potentially acquainted with a. But this is only possible if b was sent a message containing a reference to a at $t+1$ by some c. This implies that c was live and acquainted with a at time t.
  - Suppose that at $t+1$, b is monitored by a and b is faulty or potentially unblocked, but this is not the case at t. Then either (1) b is not monitored by a at t, or (2) b is not faulty at t.

    Case (1) can only occur if a was busy at t and began monitoring b at $t+1$.

Case (2) can only occur if b halts or becomes exiled at $t + 1$. In either case, because b was monitored by a at $t + 1$, it must also have been monitored at t. Because b was either busy or on a remote node at $t + 1$, a was potentially unblocked at t.

- If b is on a different node from a and monitored at $t + 1$ but not t, then a was busy at t and began monitoring at $t + 1$.

By backward induction on time, it follows that any potentially unblocked actor a at time t' must also be potentially unblocked throughout the interval $[t_a, t']$, where t_a is the creation time of a.

The theorem follows by letting t' be the time that the live actor a becomes busy. Then a is potentially unblocked at t' and, by the property above, also potentially unblocked at t. QED.

Definition 6.6. An actor a is *strongly quiescent* if it is not potentially unblocked, i.e. a has failed or all of the following hold:

1. a is blocked;
2. a is not a sticky actor;
3. a's potential inverse acquaintances are strongly quiescent;
4. a only monitors actors that are strongly quiescent and have not failed; and
5. a does not monitor any remote actors.

Theorem 6.4. Let σ be a finite execution path with an actor a. Then a is idle throughout every legal extension σ' if and only if a is strongly quiescent at the end of σ.

Proof. Immediate from Theorems 6.2 and 6.3. QED.

The theorem above shows that *strong quiescence* is a tight characterization of the garbage actors in a configuration of our model. Despite numerous complications of our new model—dropped messages, faulty actors, monitor signals arriving out-of-order with messages—the characterization is a fairly straightforward generalization of quiescence from Part I.

6.6.3 *Weak Quiescence*

The above characterization of actor garbage is correct, but somewhat inconvenient from a practical point of view: actors that monitor remote actors can never be garbage collected. This could lead to resource leaks, as the following example illustrates.

Consider the case in Hadoop YARN [1] (Figure 6.6) where an "application master" actor *am* spawns a local "task manager" actor *tm* that monitors a remote "task" actor t; the application master sends work to the task manager, which in turn forwards those jobs to the task actor. When the application master becomes quiescent, we would hope that *am*, *tm*, t are all garbage collected.

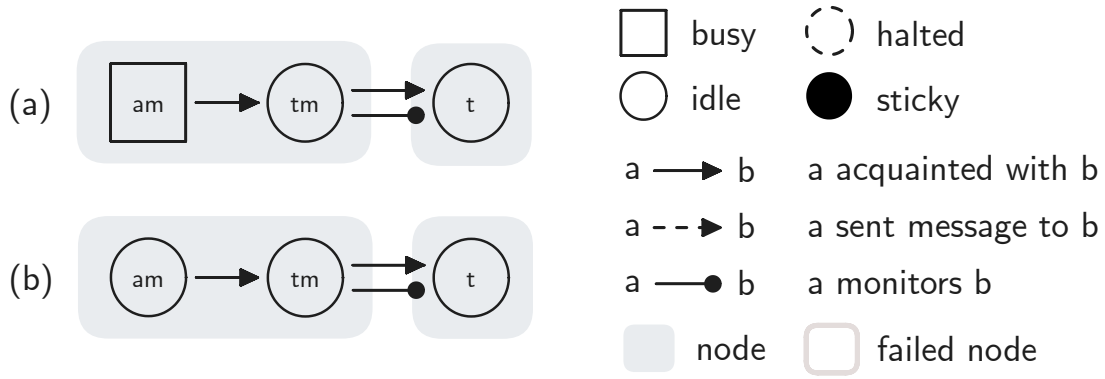


Figure 6.6: An example of a resource leak when relying on strong quiescence. In (a), actors *am*, *tm*, *t* are not garbage. In (b), *am* is strongly quiescent but *tm* and *t* prevent one another from becoming quiescent because they are on distinct nodes.

However, even though t can never become busy, it can always fail if its node crashes. Hence *tm*, which monitors t, will never be strongly quiescent. Moreover, actor t cannot be garbage collected because it is potentially reachable by the non-garbage actor *tm*. Thus *tm* and t prevent one another from being collected.

Intuitively, the cycle *tm*, t above should not be garbage collected while t is processing messages (because t may halt before the job is complete, requiring *tm* to take fault-recovery actions) but the actors *should* be collected once they have finished processing messages. In theory, it is possible for t's node to become exiled and for *tm* to be notified of the failure—but the useful work is presumably already done. We therefore offer the following principle:

> **The *weak garbage* principle:** *If an actor remains idle in every non-faulty execution path, then that actor is garbage.*

The drawback of this principle is that it introduces a race condition between the garbage collector and the fault detection mechanism. Suppose that:

1. Actor a on node N monitors actor b on node N'.
2. The garbage collector detects that a and b are weakly garbage.
3. Node N' is exiled.
4. Before a can be *collected* as garbage, a is notified that b failed and a becomes busy.

The usefulness of the weak garbage principle, and the possible problems caused by this race condition, warrants further study. In this thesis, we will content ourselves with formally characterizing "weak garbage" and showing how the actor GC in Chapter 7 can be adapted to detect either type of garbage.

Definition 6.7. Actor a is *weakly live* in σ if there exists a non-faulty extension of σ in which a is busy. Otherwise, a is *weakly garbage* in σ.

Definition 6.8. An actor a is *weakly potentially unblocked* if a is not halted or exiled and one of the following holds:

1. a is unblocked;
2. a is a sticky actor;
3. There exists another actor b that is weakly potentially unblocked and a is a potential acquaintance of b; or
4. There exists another actor b that is halted or exiled or weakly potentially unblocked and a is monitoring b.

The above definition is identical to Definition 6.5, except for the last condition. Likewise in the definition below:

Definition 6.9. An actor a is *weakly quiescent* if it is not weakly potentially unblocked, i.e. a has failed or all of the following hold:

1. a is blocked;
2. a is not a sticky actor;
3. a's potential inverse acquaintances are weakly quiescent; and
4. a only monitors actors that are also weakly quiescent and have not failed.

Notice that strong quiescence implies weak quiescence, but the converse is not true.

Theorem 6.5. Let σ be a finite execution path with an actor a. Then a is idle throughout every legal non-faulty extension σ' if and only if a is weakly quiescent at the end of σ.

Proof. The result follows by modifying the proofs of Theorem 6.3, Theorem 6.2, and Theorem 6.4 with *non-faulty* extensions and using Definitions 6.8 and 6.9. QED.

Notice, by comparing Theorems 6.4 and 6.5, that any actor GC detecting weak quiescence can be made to detect strong quiescence by not collecting the actors that monitor remote actors.

7

Fault-Recovering Actor GC

Part I presented *PRL*, an actor GC based on reference listing. Actors in PRL send control messages (*Info* and *Release*) to identify acyclic garbage and local snapshot messages to identify cyclic quiescent garbage. In this chapter, we develop a simplified version of PRL in which actors do not send control messages, leaving all garbage collection up to the GC. This has several advantages:

- Refobs (Chapter 4) and their unique tokens are no longer necessary.
- It suffices for actors to have a single message receive count, rather than message receive counts for each inverse acquaintance.
- Proofs are simplified because actor snapshots never lose information; an actor's snapshot is a summary of all the actions that the actor has performed so far.

We progressively build on this GC, incrementally adding support for sticky actors, monitoring, dropped messages, and exiled nodes:

1. Section 7.2 introduces the Static model, in which the set of actors and acquaintances is fixed. This simple model illustrates the basic intuitions and proof techniques that will be used in the following models.
2. Section 7.3 introduces the Dynamic model, which adds the ability to spawn actors and create or remove acquaintances. This model introduces *contact tracing* to determine whether two actors are acquainted.
3. Section 7.4 introduces the Monitors model, which adds the ability for actors to halt, monitor other actors, and register as sticky actors. This model shows that monitoring and sticky actors introduce additional cases to consider, but ultimately these cases do not significantly increase complexity.
4. Section 7.5 introduces the Exile model, which adds locations and dropped messages, as in the fault model introduced in Chapter 6. This model introduces the use of *ingress actors* and *ingress snapshots* to compensate for missing actorsnapshots or dropped messages.
5. Section 7.6 introduces data structures called *shadow graphs* and *undo logs* for efficiently identifying garbage in the Exile model.

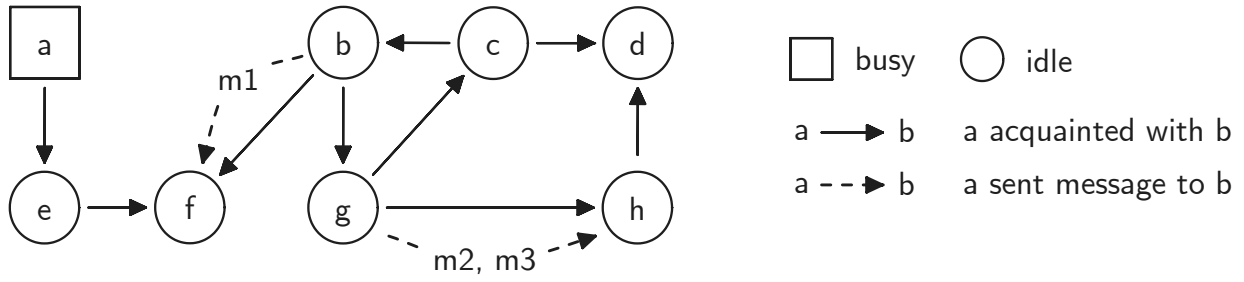


Figure 7.1: A global snapshot of a configuration in the STATIC model. Actors b, c, g are quiescent and a, d, e, f, h are not.

The models have all been formalized in TLA+ [71] and are presented in Appendices A.3 to A.8.[5]

7.1 The Collage-Based Approach

We assume that actors take local snapshots at arbitrary times without coordination. A collection of snapshots from distinct actors is called a *collage*. Formally, we represent a collage S as a partial map from actor names to actor snapshots, $S : \mathit{ActorName} \rightarrow \mathit{ActorState} \cup \{\text{NULL}\}$.

Informally, we say S is *consistent* if no snapshot in S "happens before" any other snapshot. We also say that a collage is *global* at time t if it contains snapshots from every actor at t. A global consistent collage is called a *global snapshot* [21]. Many existing actor GCs detect garbage by first computing a global snapshot [12, 17] and then searching the global snapshot for garbage. In this chapter, take an alternative approach that does not require collages to be *a priori* consistent or global. Instead, actors record information in their local state so that a garbage collector can find, within any collage S, a sub-collage Q that is consistent and corresponds to quiescent actors.

7.1.1 Notation

The specifications in Appendix A use square brackets to apply functions. In proofs, we will use parentheses instead; thus $\mathit{actors}[a]$ in the specification becomes $\mathit{actors}(a)$ in proofs.

We will abuse notation and use an actor name a to stand for its local state $\mathit{actors}(a)$; thus we will write $a.\text{created}(b, c)$ in place of $\mathit{actors}(a).\text{created}(b, c)$. To refer to a field in a's snapshot in collage S, we write $S(a).\text{created}(b, c)$.

We will also frequently "lift" properties about individual actors into properties about sets. For example, we say a set of actors Q is quiescent if every $a \in Q$ is quiescent.

[5] The sources are available at https://github.com/dplyukhin/uigc-spec.

7.2 Static Topologies

The STATIC model consists of n actors passing messages in a fixed (but arbitrary) topology. That is, actors in this model cannot spawn or send references in messages. In this section, we develop a definition of quiescence that specializes the definition in Section 6.6 and show that quiescent garbage can be detected using a variant of Mattern's channel counting algorithm [46].

7.2.1 Model

The STATIC model is a significantly simplified version of the model in Chapter 6. This model only includes four events:

1. $Idle(a)$: Busy actor a becomes idle, allowing it to receive a new message.
2. $Send(a, b, m)$: Busy actor a sends b a message m.
3. $Receive(a, m)$: Idle actor a receives a message m and becomes busy.
4. $Snapshot(a)$: Actor a takes a snapshot.

Notice we omit monitoring, locations, failures, sticky actors, etc. The initial configuration consists of n actors in an arbitrary topology; Figure 7.1 shows one such example. In this topology, we define the following terminology:

> Actor a is *acquainted* with b if a's local state contains a reference to b; we also say that a is an *inverse acquaintance* of b. The set $iacqs(b)$ ranges over the inverse acquaintances of b.
>
> *Reachability* is the transitive closure of the acquaintance relation.
>
> An actor a is *blocked* if it is idle and has no undelivered messages. Otherwise, a is said to be *unblocked*.

For example, actors a, f, h are unblocked in Figure 7.1 and actors b, c, g can reach b, c, d, f, g, and h.

We can specialize Definition 6.6 to the STATIC model:

Definition 7.1. An actor a is *quiescent* if:

1. a is blocked; and
2. Every inverse acquaintance of a is quiescent.

Using this definition, one can identify quiescent garbage by computing a consistent global snapshot and searching the snapshot for actors matching this definition. In Figure 7.1, only actors b, c, g are quiescent; actors a, f, h are not quiescent because they are unblocked and actors e, d are not quiescent because they are reachable by unblocked actors.

7.2.2 Apparent Quiescence

Even in this simple model, how can we identify quiescent actors without taking a consistent global snapshot? In this section, we give an approach for detecting quiescent actors given an *arbitrary* collage S in the Static model. Actors record the number of messages sent to each acquaintance and the total number of messages they received. We will prove *soundness* (if actors appear quiescent, they are actually quiescent) and *completeness* (if actors are actually quiescent, then they will eventually appear quiescent).

Our approach generalizes Mattern's *channel counting* algorithm [46]. The channel counting algorithm uses message send/receive counts to detect *global* quiescence, i.e. when *all* actors are quiescent. We build on the algorithm by (1) detecting quiescence of individual actors instead of quiescence of the entire system, and (2) only requiring actors to have a single *total* receive count instead of a count for each inverse acquaintance.

In our approach, each actor a tracks:

- a.status: An indicator of whether a is idle or busy;
- a.received: The total number of messages a received so far; and
- a.sent(b): The number of messages a sent to b so far.

In an arbitrary collage S, we define what it means to "appear" idle or "appear" blocked in the obvious way:

> a *appears idle* if $S(a)$.status = idle.
>
> b *appears blocked* if $S(b)$.received = $\sum_{a \in iacqs(b)} S(a)$.sent($b$).
>
> S is *closed* for b (with respect to inverse acquaintances) if $b \in \mathrm{dom}(S)$ implies $iacqs(b) \subseteq \mathrm{dom}(S)$.

Recall that S is not necessarily global; the notion of closed collages is needed so that appearing blocked is well-defined.

We can now define what it means for actors to appear quiescent similarly to Definition 7.1:

Definition 7.2. An actor a *appears quiescent* in collage S if:

1. S is closed for a;
2. a appears blocked; and
3. Every inverse acquaintance b of a appears quiescent.

Figure 7.2 depicts a sample collage S and the actors that appear quiescent within it. Notice that e does not appear quiescent because S is not closed for e, whereas h does not appear quiescent because it appears unblocked. Throughout the rest of this section, we use S to denote a collage and Q to denote a closed subset of dom(S) that appears quiescent.

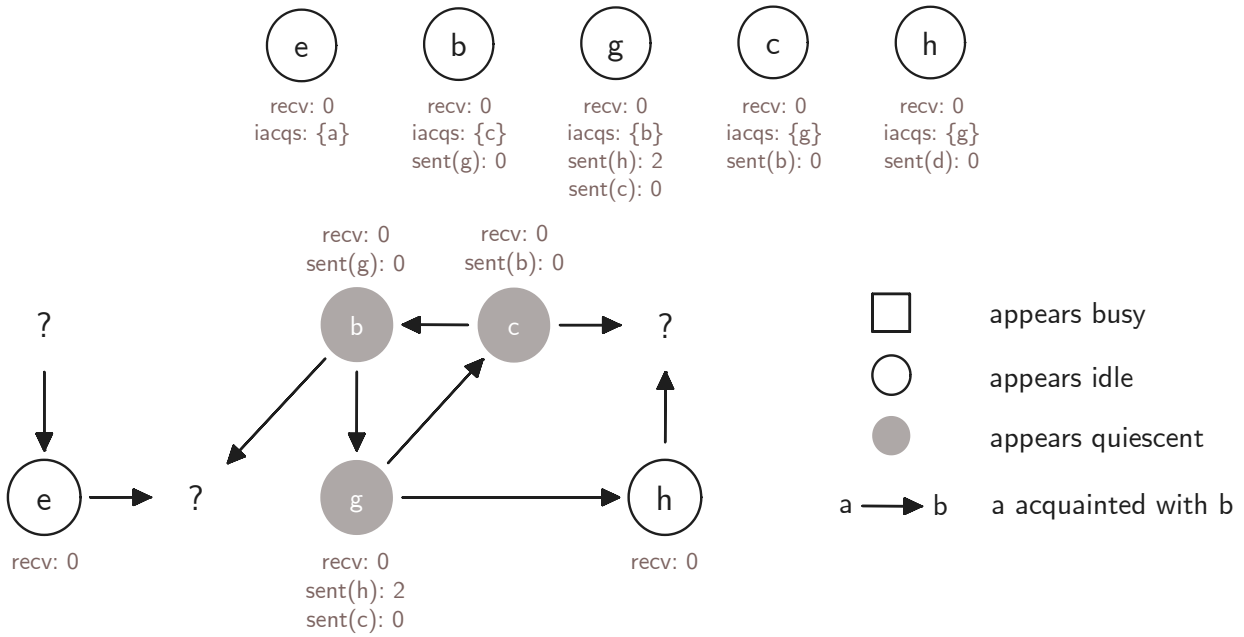


Figure 7.2: Two views of a collage STATIC model. The top diagram shows a collage as a collection of snapshots. The bottom diagram shows the graph induced by the collage; question marks represent actors that do not have snapshots in the collage.

It is perhaps obvious that if S is consistent then the apparently quiescent actors in S are indeed quiescent. But surprisingly, this property holds even if S is not *a priori* consistent because any subcollage that appears quiescent is *necessarily* consistent. The result holds by means of a simple invariant:

> (STATIC-INV): For each $a \in Q$, if a has taken a snapshot then a is idle.

To show that (STATIC-INV) implies consistency for Q, we first formalize consistency:

> Let $a, b \in \mathrm{dom}(S)$. There is a *forward-crossing message from a to b* if a sent a message before a's snapshot that b did not receive before b's snapshot. Likewise, there is a *backward-crossing message from a to b* if a sent a message after a's snapshot that b received before b's snapshot.
>
> A collage S is *consistent for a* if there are no backward-crossing messages to a. A collage S is *consistent* if there are no backward-crossing messages for any $a \in \mathrm{dom}(S)$.

Lemma 7.1. Let $b \in Q$ and let t be the time b recorded a snapshot. If (STATIC-INV) holds up to time t, then there are no forward- or backward-crossing messages to b.

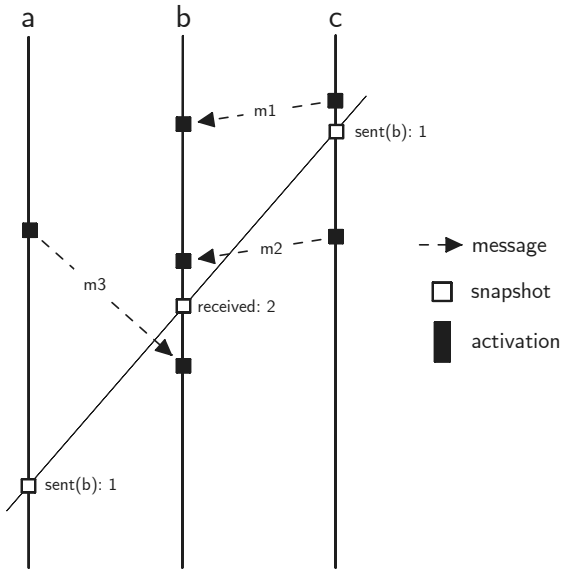


Figure 7.3: A time diagram depicting how backward-crossing messages can mask the existence of forward-crossing messages.

Proof. The fact that there are no backward-crossing messages follows immediately from the invariant: if an actor a took a snapshot before sending a message to b and the message was received before b took a snapshot, then a must have become busy before t. Since any a that could send b a message is in Q, this would violate the invariant.

For each actor a, let $\mathsf{recv}_{a,b}$ equal the number of messages b received from a before b's snapshot. In particular, because every a that could send a message to b is in Q, we have

$$\sum_{a \in \mathrm{dom}(S)} \mathsf{recv}_{a,b} = S(b).\mathsf{received}. \tag{7.1}$$

For every $a \in \mathrm{dom}(S)$, let $\mathsf{undelivered}_{a,b}$ equal the number of forward-crossing messages from a to b. Because there are no backward-crossing messages, every message b received from a before b's snapshot was sent before a's snapshot, i.e. $\mathsf{recv}_{a,b} + \mathsf{undelivered}_{a,b} = S(a).\mathsf{sent}(b)$. In particular:

$$S(b).\mathsf{received} + \sum_{a \in \mathrm{dom}(S)} \mathsf{undelivered}_{a,b} = \sum_{a \in \mathrm{dom}(S)} S(a).\mathsf{sent}(b) \tag{7.2}$$

Because b appears blocked, $S(b).\mathsf{received} = \sum_{a \in \mathrm{dom}(S)} S(a).\mathsf{sent}(b)$. Applying Equation (7.2), this can only be true if $\mathsf{undelivered}_{a,b} = 0$ for all $a \in \mathrm{dom}(S)$. QED.

The key idea of the soundness proof is to consider the first actor b that violates the invariant. The actor can only become busy from a forward-crossing message, but because b appears blocked

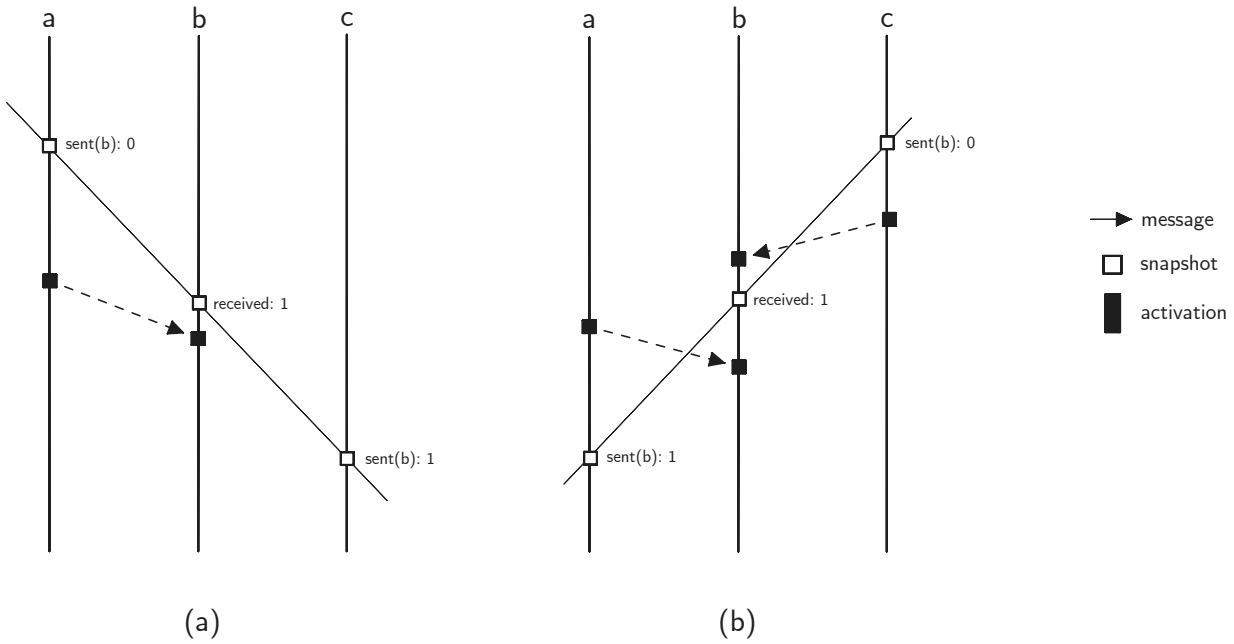


Figure 7.4: Time diagrams depicting how an actor b could become busy after taking a snapshot.

such a message can never arrive.

Theorem 7.1 (Soundness)**.** Let S be a collage and Q a closed subset of $\mathrm{dom}(S)$ that appears quiescent. Then Q is quiescent.

Proof. Let t be the first time that (Static-Inv) is violated. Then some $b \in Q$ took a snapshot and subsequently became busy at t.

In Static, actors only become busy by receiving messages. Let a be the actor that sent the message. Necessarily, $a \in \mathit{iacqs}(b) \subseteq \mathrm{dom}(S)$. Note that we may have $a = b$.

Suppose a took a snapshot before sending the message; this is depicted in Figure 7.4 (a). This violates the assumption that b was the first actor to become busy after taking a snapshot.

Suppose instead that a took a snapshot after sending the message, as shown in Figure 7.4 (b). By Lemma 7.1, this forward-crossing is only possible if there is a backward-crossing message from some c. But then c violates the assumption that b was the first actor to become busy after taking a snapshot. QED.

We conclude this section by showing that the collage-based approach eventually detects all quiescent garbage, under a very lax fairness assumption. This result is due to an important property of quiescent garbage: any collage collected from quiescent actors after they have become quiescent is necessarily consistent. Without this property, a garbage collector might never obtain a consistent view of the garbage actors.

Theorem 7.2 (Completeness)**.** Assume that every actor always eventually takes a snapshot. If a is quiescent at time t in execution path σ, then eventually a will appear quiescent.

Proof. Let Q be the closure of $\{a\}$ with respect to inverse acquaintances. Because a is quiescent at t, the closure Q is also quiescent. In any collage with snapshots from Q taken after t, all actors will appear blocked. QED.

7.3 Dynamic Topologies

In this section, we generalize the preceding approach to dynamic topologies. In a dynamic topology, actors can *create* references by spawning actors or sending messages that contain actor names. Actors can also *deactivate* references by removing an acquaintance's name from their local state. From the perspective of garbage collection, the key complication in this model is determining whether a collage is closed.

7.3.1 Model

The Dynamic model introduces the following events:

1. $Spawn(a, b)$: Busy actor a spawns an actor with fresh name b;
2. $Deactivate(a, b)$: Busy actor a removes all references to b from its local state; and
3. $Send(a, b, m)$: Sent messages can now contain references to the acquaintances of a.

Figure 7.5 demonstrates that Definition 7.1 no longer characterizes quiescent actors in the Dynamic model: at time (1), c is blocked and not reachable by any actor; yet at time (3) c is unblocked. The key problem is that *reachability* does not account for references in undelivered messages, such as the message sent from a to b at time (1).

To account for this, we introduce new terminology:

> A reference from a to b is said to be *owned by* a and *pointing to* b; we also call it a *b-reference*. If c sent the reference, then the reference was *created by* c.
>
> Actors may have references in three stages:
>
> 1. *Pending references:* references sent to a that have not yet been delivered.
> 2. *Active references:* references that a has received and not yet deactivated.
> 3. *Deactivated references:* references that a removed from local state.
>
> We say a is *potentially acquainted with* b if a has active or pending references to b. We also say a is a *potential inverse acquaintance* of b; the set $piacqs(b)$ ranges over the potential inverse acquaintances of b.
>
> *Potential reachability* is the transitive closure of the potential acquaintance relation.

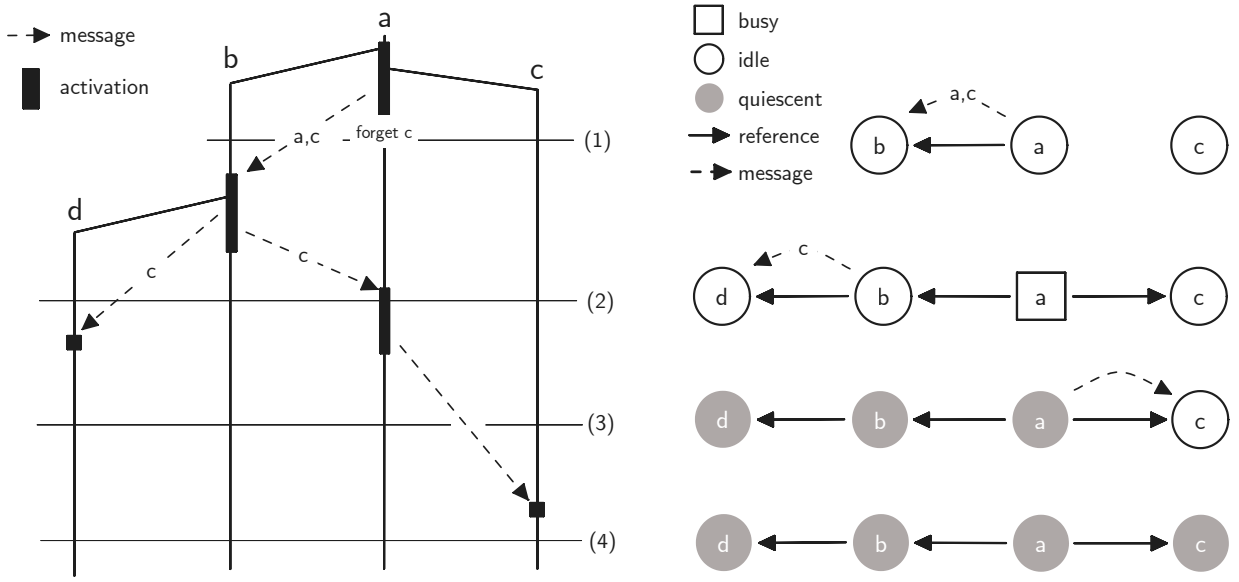


Figure 7.5: A sample execution in the DYNAMIC model. On the left is a time diagram. On the right we see the states of the actors at times (1), (2), (3), and (4).

In light of the new terminology, we see that c is not quiescent in Figure 7.5 (1) because c is *potentially* reachable by the unblocked actor b. We proceed to modify the definition of quiescence for the DYNAMIC model; the *emphasized* portion shows the difference compared to Definition 7.1.

Definition 7.3. An actor a is quiescent if:

1. a is blocked; and
2. Every *potential* inverse acquaintance of a is quiescent.

7.3.2 *Detecting Garbage*

In Section 7.2, we identified that a collage must be closed (with respect to inverse acquaintances) to appear quiescent. This poses a problem for the DYNAMIC model, in which actors no longer know their inverse acquaintances. Indeed, an actor a's address could possibly have been sent to any actor in the system. Figure 7.6 shows one such example, in which a non-quiescent actor b appears blocked because one of its inverse acquaintances, d, has not taken a snapshot.

Prior work has coped with this problem by requiring snapshots to be *global* [12] or by synchronously maintaining inverse reference listings for each actor [42]. In this section, we introduce a more lightweight approach: *contact tracing*.

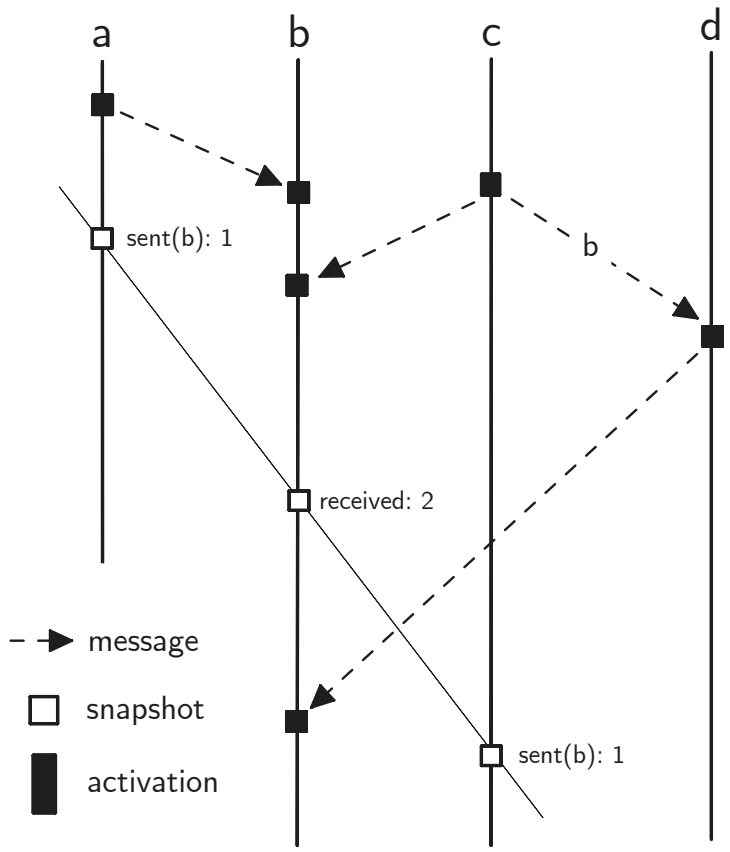


Figure 7.6: An execution illustrating the need for contact tracing.

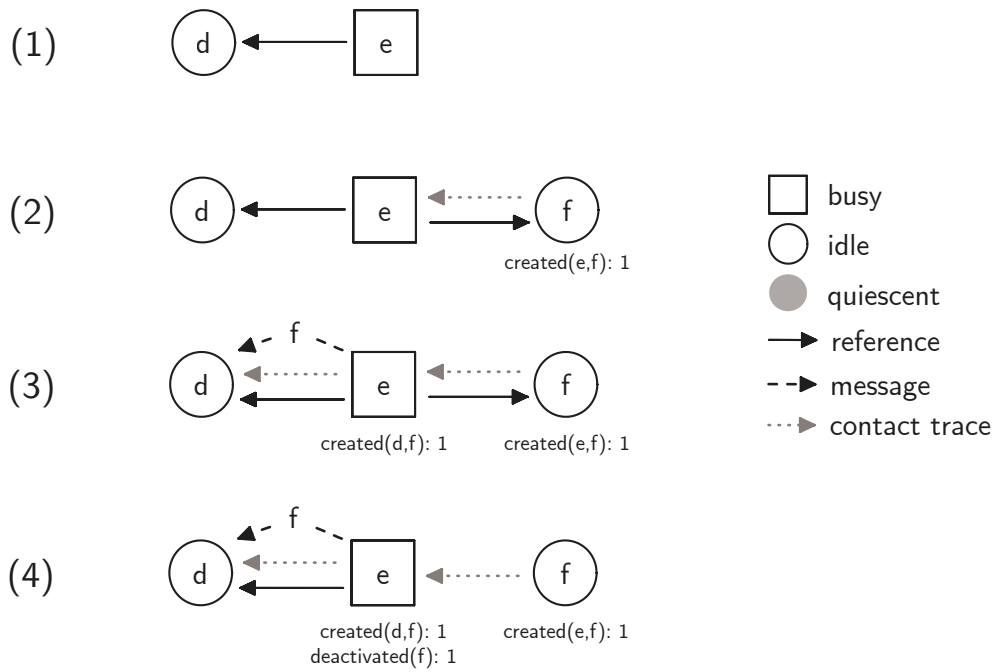


Figure 7.7: An execution with contact tracing.

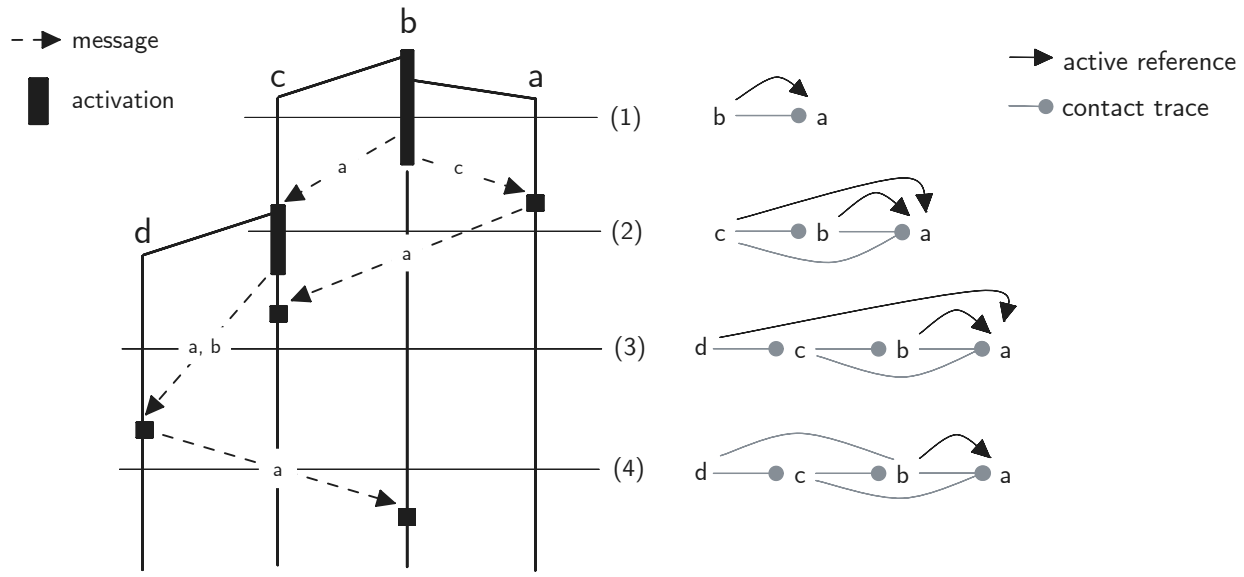


Figure 7.8: An execution in the DYNAMIC model with contact tracing. The right side of the figure depicts the contact tracing chains for actor a at times 1–4.

7.3.3 *Contact Tracing*

The key idea of contact tracing is for actors to record whenever they create new references. As a result, there is always a "contact tracing chain" from an actor to all its potential inverse acquaintances. We instrument each actor a with two new fields:

- $a.\text{created}(b, c)$: The number of c-references a sent to b.
- $a.\text{deactivated}(b)$: The number of b-references that a has deactivated.

Figure 7.7 shows a sample execution with contact tracing. Every actor a is spawned with a reference to itself, so $\text{created}(a, a) = 1$ in a's initial state. In addition, if a spawned b, we say the parent obtained a reference from the child, so $\text{created}(a, b) = 1$ in the b's initial state.

Figure 7.8 depicts an execution in which actors maintain contact tracing information. When b spawns a, there is a direct link from a to b. After b sends c a reference to a, there is a link in the chain from b to c. After a sends c a reference to itself, there is also a link in the chain from a to c. Subsequently, c sends d a reference to a and deactivates both of its a-references; despite the fact that c has no active references to a, there is still a contact tracing link from c to d. Finally, d sends b a reference to a and deactivates its reference to a; the only actor that can potentially reach a at time (4) is b, and there is indeed a path from a to b using the contact tracing information.

> We say that a is *hereto acquainted* with b at time t if a was acquainted with b at some time $t' \leq t$. We then call a an *inverse hereto acquaintance* of b. The set $\mathit{hiacqs}(b)$ ranges over b's hereto inverse acquaintances.

We say S is *hereto-closed* if $b \in \mathrm{dom}(S)$ implies $\mathit{hiacqs}(b) \subseteq \mathrm{dom}(S)$.

7.3.4 Apparent Quiescence

We now define what it means for a collage to "appear" closed, using contact tracing:

Actor a *appears hereto acquainted* with b in S if $\sum_{c \in \mathrm{dom}(S)} c.\mathsf{created}(a, b) > 0$.

Actor a *appears acquainted* with b if $\sum_{c \in \mathrm{dom}(S)} c.\mathsf{created}(a, b) > a.\mathsf{deactivated}(b)$.

A collage S *appears hereto-closed* if, for each $b \in \mathrm{dom}(S)$, all of b's apparent hereto inverse acquaintances are in S.

The hereto-closure of b includes snapshots from former acquaintances; these snapshots are needed because they contain contact tracing information and because they may have nonzero $\mathsf{sent}(b)$ counts.

Now we generalize the definition of apparently quiescent collages. The highlighted parts show the changes, compared to Definition 7.2.

Definition 7.4. Actor b appears quiescent in collage S if:

1. b is *appears hereto-closed* in S;
2. b appears blocked in S; and
3. If some a *appears acquainted* with b, then a appears quiescent as well.

Throughout the rest of this section, we use S to denote an arbitrary collage and Q to denote an arbitrary subset of $\mathrm{dom}(S)$ that appears hereto-closed and appears quiescent.

Interestingly, Definition 7.4 does not require b's former inverse acquaintances to appear quiescent. This reflects the fact that, if a is non-garbage and deactivates all its references to b, then b can be garbage collected without waiting for a to be collected.

7.3.5 Soundness

In Section 7.2.1, we proved soundness by means of an invariant. To account for dynamic topologies, we strengthen the invariant by adding an extra condition:

(Dynamic-Inv):

1. *(Idleness)* For each $a \in Q$, if a has taken a snapshot then a is not busy.
2. *(Closure)* For each $a \in \mathrm{dom}(S)$, if a has taken a snapshot and a is acquainted with some $b \in Q$, then $a \in Q$.

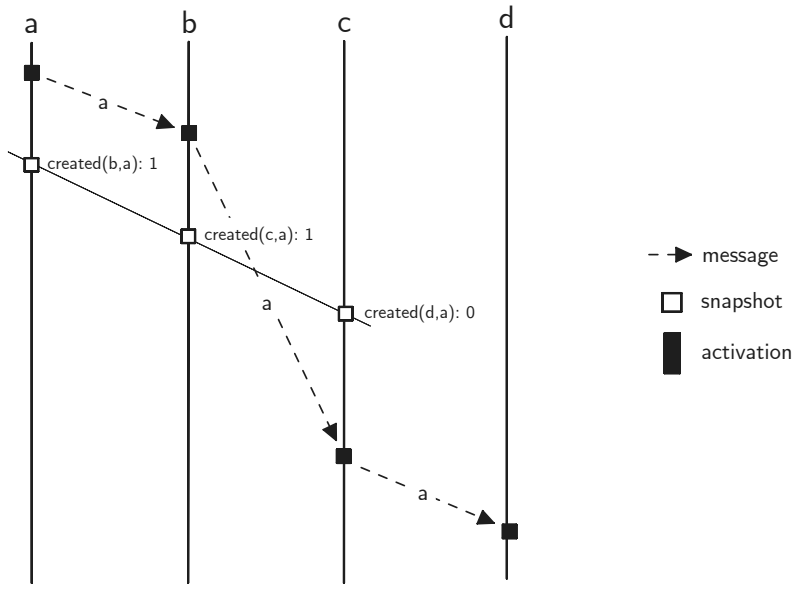


Figure 7.9: A time diagram showing an impossible execution, in which c creates a reference to an actor $a \in Q$ after c takes a snapshot.

We say that a collage S is hereto-closed *up to time* t for b if all b's hereto inverse acquaintances up to time t are contained in $\mathrm{dom}(S)$. The following lemma shows that, as long as the invariant holds, apparently quiescent collages that *appear* hereto-closed are indeed hereto-closed.

Lemma 7.2. If (Dynamic-Inv) holds up to t then S is hereto-closed for Q up to t.

Proof. Let $c \in Q$. By induction on time t, we show that every actor b with a reference to c at time t must have a snapshot in S.

At $t = 0$, there is only one actor a with a reference to itself. If $a \in Q$ then $a \in \mathrm{dom}(S)$ by definition.

Assuming the property holds up to time t, suppose actor b obtains a reference to c at time $t+1$. There are three possibilities.

1. b spawned c. Then c was spawned with $c.\mathrm{created}(b, c) > 0$ in its local state. This implies $S(c).\mathrm{created}(b, c) > 0$. Since Q appears quiescent, b must have a snapshot in S.
2. $b = c$ and c has just been spawned, thereby obtaining a reference to itself. Again, $c \in Q$ implies $b \in \mathrm{dom}(S)$.
3. b received a reference to c in a message, sent by some a. Let t_s be the time when a sent the message. Actor a must have been acquainted with c when it sent the message, so $a \in \mathrm{dom}(S)$ by the induction hypothesis. Notice that $a.\mathrm{created}(b, c) > 0$ at t_s. By (Dynamic-Inv), a could not have taken a snapshot before sending the message. Hence $S(a).\mathrm{created}(b, c) > 0$. Since Q appears quiescent, $b \in \mathrm{dom}(S)$.

QED.

The lemma above shows that executions such as Figure 7.9 are impossible: if $a \in Q$ and c obtains a reference to a, then c must be in Q as well—and therefore cannot receive a message after taking a snapshot.

The following lemma is a generalization of Lemma 7.1 for dynamic topologies, and has an identical proof; the key difference is that S is now *hereto*-closed, due to Lemma 7.2.

Lemma 7.3. Let $b \in Q$ and let t be the time that b took a snapshot. If (Dynamic-Inv) holds up to t then there are no forward- or backward-crossing messages to b.

Next we prove a variation of Lemma 7.3 for messages containing references. The lemma expresses a *topological* consistency property: if there are no backward-crossing messages containing references (as in Figure 7.10) then an actor's apparent acquaintances correspond to its actual acquaintances.

> Let S be a collage and $a, b \in \mathrm{dom}(S)$. There is a *forward-crossing c-reference from a to b* if a sent a message before a's snapshot that b received after b's snapshot, and the message contained a reference to c. Likewise, there is a *backward-crossing c-reference from a to b* if a sent a message after a's snapshot that b received before b's snapshot, and the message contained a reference to c.

Lemma 7.4. Let $b \in \mathrm{dom}(S)$ and let t be the time that b took a snapshot. If (Dynamic-Inv) holds up to t then there are no forward- or backward-crossing c-references to b, for any $c \in Q$. Furthermore, b appears acquainted with c if and only if b is actually acquainted with c at the time of b's snapshot.

Proof. The fact that there are no backward-crossing references follows from the invariant: Suppose actor a takes a snapshot at t_a; then a sends a c-reference to b at t_s; and then the reference is received before t. To send the reference, a must have had a reference to c at $t_s \in (t_a, t)$. By (Dynamic-Inv), $c \in Q$ implies $a \in Q$. But then (Dynamic-Inv) also implies a was idle at t_s, so the reference could not have been sent in the first place.

We now show that there are no forward-crossing c-references to b. Recall that b can obtain a reference to c in three ways:

1. Some a sent b a reference to c;
2. b spawned c; or
3. $b = c$, so b was spawned with a reference to itself.

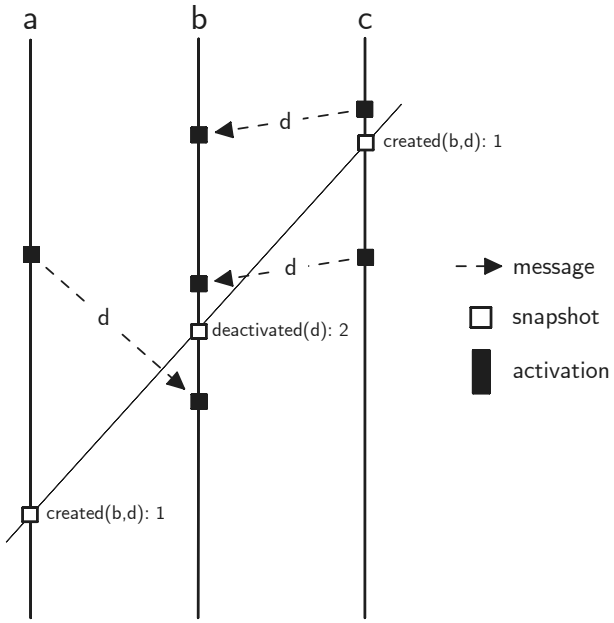


Figure 7.10: A time diagram with forward- and backward-crossing references. Message m_2 is a backward-crossing d-reference from c to b. Message m_3 is a forward-crossing d-reference from a to b. The backward-crossing d-reference causes b to appear as if it is not potentially acquainted with d.

For each actor a in the execution, let $\text{activated}_{a,b}$ equal the number of c-references that b received from a before t. Any c-reference that b received before t must have been sent by an actor a that was acquainted with c at some earlier time $t' < t_b$. By Lemma 7.2, $a \in \text{dom}(S)$. It follows that:

$$\text{self} + \sum_{a \in \text{dom}(S)} \text{activated}_{a,b} \geq S(b).\text{deactivated}(c), \tag{7.3}$$

where

$$\text{self} = \begin{cases} 1 & \text{if } b = c, \text{ or if } b \text{ spawned } c \text{ before } t \\ 0 & \text{otherwise} \end{cases} \tag{7.4}$$

That is, b could only have deactivated the c-references that it obtained from itself or actors in $\text{dom}(S)$.

For each $a \in \text{dom}(S)$, let $\text{crossing}_{a,b}$ equal the number of forward-crossing c-references from a to b. Hence, because we already proved there are no backward-crossing c-references,

$$S(a).\text{created}(b,c) = \begin{cases} \text{activated}_{a,b} + \text{crossing}_{a,b} & \text{if } a \neq b \\ \text{activated}_{b,b} + \text{crossing}_{b,b} + \text{self} & \text{otherwise} \end{cases} \tag{7.5}$$

There are two cases.

Case 1 ($\Rightarrow$). Assume b appears acquainted with c. Since $c \in Q$ and appears quiescent, $b \in Q$. Then Lemma 7.3 implies there are no forward-crossing messages to b. Hence, in particular there can be no forward-crossing c-references to b.

Next we show that b is acquainted with c at time t. Lemma 7.3 implies $\text{crossing}_{a,b} = 0$ for all a, so by Equation (7.5),

$$\sum_{a \in \text{dom}(S)} S(a).\text{created}(b, c) = \text{self} + \sum_{a \in \text{dom}(S)} \text{activated}_{a,b}. \tag{7.6}$$

Also, since b appears acquainted with c,

$$\sum_{a \in \text{dom}(S)} S(a).\text{created}(b, c) > S(b).\text{deactivated}. \tag{7.7}$$

Hence

$$\text{self} + \sum_{a \in \text{dom}(S)} \text{activated}_{a,b} > S(b).\text{deactivated}, \tag{7.8}$$

i.e. b has at least one reference to c at time t that has not been deactivated.

Case 2 ($\Leftarrow$). Assume b does not appear acquainted with c. By Equation (7.5):

$$S(b).\text{deactivated}(c) \geq \text{self} + \sum_{a \in \text{dom}(S)} (\text{activated}_{a,b} + \text{crossing}_{a,b}) \tag{7.9}$$

$$\geq \text{self} + \sum_{a \in \text{dom}(S)} \text{activated}_{a,b} \tag{7.10}$$

Combining the above inequality with Equation (7.3), we deduce that $\text{crossing}_{a,b} = 0$ for all a, i.e. there are no forward-crossing c-references. Moreover, every reference b obtained for c has been deactivated—so b is not acquainted with c at time t. QED.

Theorem 7.3 (Soundness)**.** Let S be a collage and let Q be a an apparently hereto-closed subset of $\text{dom}(S)$ that appears quiescent. Then Q is quiescent.

Proof. The result holds as a consequence of (Dynamic-Inv). We prove the invariant holds by contradiction. Let t be the first time the invariant is violated.

Now we consider the two ways the invariant could be violated at time t.

Case 1. An actor $b \in Q$ is busy after taking a snapshot. Because b was idle when it took a snapshot, it must have become busy at time t. In the Dynamic model, actors only become busy by receiving messages. But by Lemma 7.3, no such message could arrive.

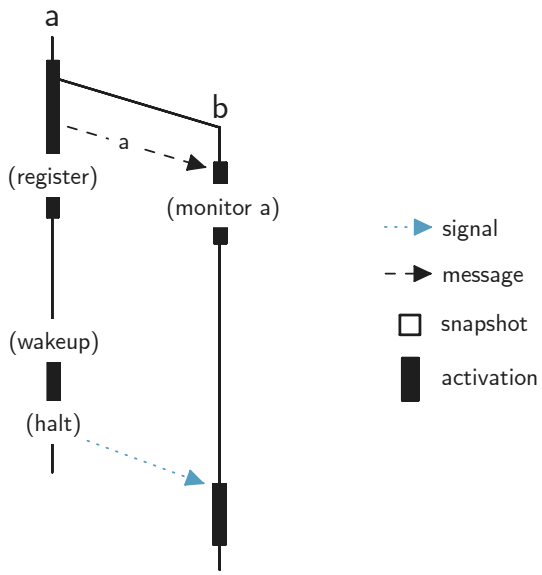


Figure 7.11: A sample execution in the MONITORS model. Actor a spawns b, registers as sticky, and becomes idle. Actor b begins monitoring a. Later, a spontaneously wakes up and then halts, causing b to become busy.

Case 2. An actor $b \in \text{dom}(S) \setminus Q$ has taken a snapshot and holds a reference to some $c \in Q$. By Lemma 7.4, b had no reference to c at the time of its snapshot and can only have obtained the reference from some $a \in \text{dom}(S)$ that sent the message after taking a snapshot. This contradicts the fact that t is the first time the invariant is violated.

Thus (DYNAMIC-INV) holds for all times t an all actors in Q remain idle after taking snapshots. QED.

7.4 Sticky Actors and Monitoring

In this section, we enrich the DYNAMIC model with halted actors, sticky actors, and monitoring. From the perspective of garbage collection, these features add new ways for actors to become busy (e.g. sticky actors spontaneously waking up) and they add new ways for actors to become garbage (e.g. if some of their potential inverse acquaintances have halted).

7.4.1 Model

The MONITORS model introduces the following events:

1. $Halt(a)$: Busy actor a halts. We assume that halted actors can take snapshots, unlike exiled actors (Section 7.5).

2. $Monitor(a, b)$: Busy actor a begins monitoring its acquaintance b.
3. $Notify(a, b)$: Idle actor a that monitors a halted actor b becomes busy. Also, a stops monitoring b.
4. $Unmonitor(a, b)$: Busy actor a stops monitoring b.
5. $Register(a)$: Busy actor a registers as a sticky actor.
6. $Wakeup(a)$: Sticky idle actor a becomes busy.
7. $Unregister(a)$: Busy sticky actor a unregisters as a sticky actor.

Figure 7.11 demonstrates that Definition 7.3 no longer characterizes quiescent actors in the Monitors model:

- Halted actors are *always* quiescent because they can never become busy.
- Sticky actors are *never* quiescent (unless they halted) because they can always spontaneously wake up.
- If b monitors a and a can halt, then b is not quiescent. Note that if a is quiescent then a cannot halt.

Fortunately, as observed in Chapter 6, it suffices to modify the Definition 7.3 to account for these complications in the obvious way:

Definition 7.5. An actor a is quiescent if *a is halted* or all of the following hold:

1. a is blocked;
2. *a is not sticky;*
3. Every potential inverse acquaintance of a is quiescent; and
4. *Every actor monitored by a is quiescent and not halted.*

7.4.2 Apparent Quiescence

To account for the new features in the Monitors model, each actor a records the following additional information:

- a.isSticky: Indicates whether a is sticky.
- a.monitored: The set of actors monitored by a.

a appears halted if $S(a)$.status is "halted".

a appears sticky if $S(a)$.isSticky is true.

a appears to monitor b if $b \in S(a)$.monitored.

Definition 7.4 now generalizes in the obvious way:

Definition 7.6. Actor b appears quiescent in collage S if *a appears halted* or all of the following hold:

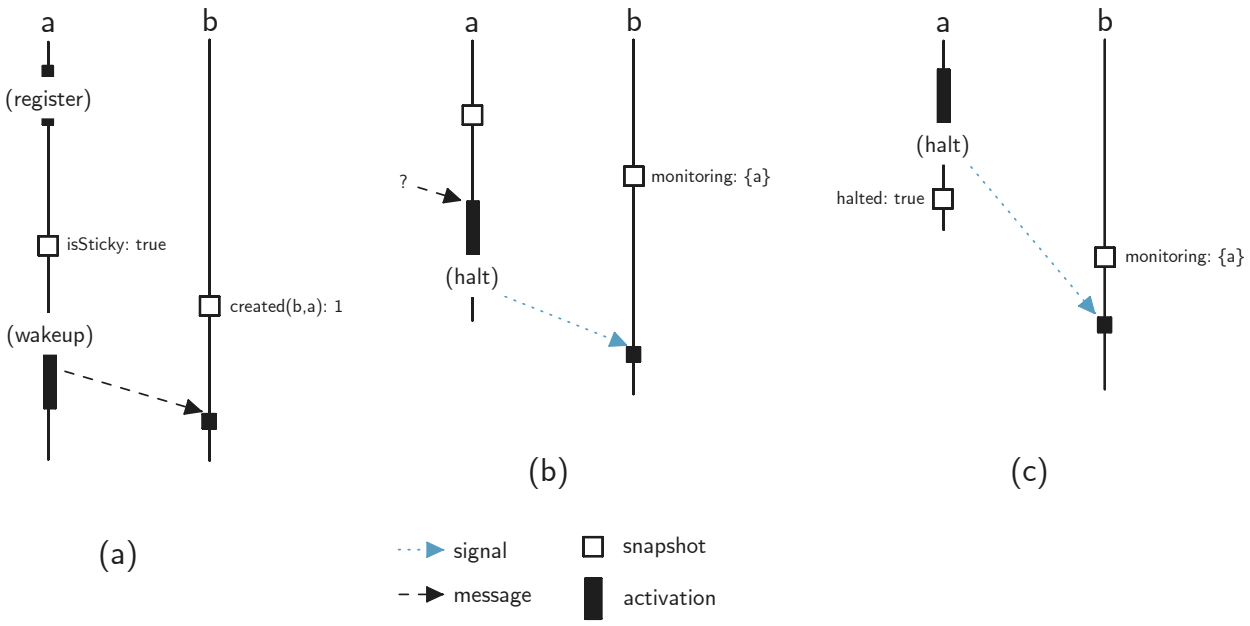


Figure 7.12: Three new ways that an actor can become busy after taking a snapshot.

1. b is appears hereto-closed in S;
2. b appears blocked;
3. *b does not appear sticky;*
4. If some a appears acquainted with b, then a appears quiescent; and
5. *If some a appears monitored by b, then a appears quiescent and does not appear halted.*

Adapting the soundness proof of Section 7.3 to Monitors is also straightforward. We use the same invariant (Dynamic-Inv) and Lemmas 7.2 to 7.4, which continue to hold without modification in the new model.

Figure 7.12 shows the three key cases to consider:

1. Actor a registers as sticky, takes a snapshot, and spontaneously wakes up.
2. Actor a, monitored by actor b, takes a snapshot and then halts.
3. Actor a, monitored by actor b, halts and then takes a snapshot.

The first case is addressed by the condition that, if an actor a appears sticky, neither a nor any of its apparent acquaintances can appear quiescent. The second case is addressed by the invariant (Dynamic-Inv): actors can only halt by first becoming busy, and an actor a that appears quiescent can never become busy. The third case is addressed by the condition that, if an actor a appears halted, then the actors that appear to monitor it cannot appear quiescent.

Theorem 7.4 (Soundness)**.** Let S be a collage and let Q be a an apparently hereto-closed subset of $\mathrm{dom}(S)$ that appears quiescent. Then Q is quiescent.

Proof. Similarly to the proof in Theorem 7.3, the result follows from the invariant (DYNAMIC-INV). We prove the invariant holds by contradiction. Let t be the first time the invariant is violated. We consider the two ways the invariant could be violated at time t.

Case 1. An actor $b \in Q$ is busy after taking a snapshot. Because b was idle when it took a snapshot, it must have become busy at time t. In the MONITORS model, actors can become busy in three ways:

1. *b received a message.* This is impossible by Lemma 7.3.
2. *b is sticky and received a wakeup signal.* This is impossible because b was not sticky at the time of its snapshot, so b would have to become busy at an earlier time $t' < t$ to register.
3. *b monitors an actor c and is notified that c halted.* This is only possible if c was busy at some $t' < t$ and then c halted. Notice that b must have been monitoring c at the time of b's snapshot so, because b appears quiescent, $c \in Q$. Because $c \in Q$ and b is the first actor in Q to become busy, c cannot have halted after taking a snapshot. Therefore c halted before taking a snapshot—but then c appears halted, contradicting the fact that b appears quiescent.

Case 2. Identical to Case 2 in the proof of Theorem 7.3.

Thus (DYNAMIC-INV) holds for all times t an all actors in Q remain idle after taking snapshots.

QED.

7.5 Dropped Messages and Exiled Nodes

In this section, we introduce faults—namely, dropped messages and exiled nodes—to the MONITORS model. From the perspective of the MONITORS GC, these faults are indistinguishable from arbitrary delays: dropped messages appear the same as undelivered messages and exiled actors appear the same as actors that (unfairly) never get to take a step. To detect garbage that results from faults, our approach will be to assume the existence of a high-level fault detection mechanism (Chapter 6) and to use this mechanism to inform the garbage collector after faults occur. Chapter 8 expounds on how these fault-detection mechanisms can be implemented.

7.5.1 Model

The EXILE model brings the MONITORS model in line with the fault model of Chapter 6. As in that chapter, every actor now has a permanent *location* and messages must explicitly be *admitted* to their destination node before they can be received. EXILE adds the events *Admit*, *Drop*, *Shun* and modifies the events *Receive*, *Spawn*, *Send*, *Notify*, *Snapshot*:

1. *Admit*(m): In-flight message m from N_1 to N_2 is admitted onto its destination N_2.

2. *Drop*(m): Message m is removed from the bag of undelivered messages. Note that both in-flight and admitted messages can be dropped.
3. *Shun*(N_1, N_2): Node N_1 is *shunned* by N_2, preventing messages from N_1 being admitted to N_2. This event also preventing actors being spawned from N_1 to N_2 or vice-versa. If a group of nodes $\mathcal{G}_1$ has been shunned by the remaining nodes $\mathcal{G}_2$, then the nodes and actors in $\mathcal{G}_1$ are said to be *exiled*.
4. *Receive*(a, m): Message m must now be *admitted* before it can be delivered to its target a.
5. *Spawn*(a, b, N): Actor a spawns b *onto node* N. The child's node can be any node that does not shun (and is not shunned by) a's node.
6. *Send*(a, b, m): Actor a sends b message m. If a and b are on the same node N, m is immediately marked as admitted.
7. *Notify*(a, b): Actor a is notified that b halted *or was exiled*.
8. *Snapshot*(a): Actor a takes a snapshot, *assuming a is not exiled*.

Note that we do not explicitly model *crashing* nodes, because they are subsumed by exiled nodes (Chapter 6).

In executions with dropped messages, collages from hereto-closed sets of quiescent actors will not necessarily appear quiescent. This can be seen in Figure 7.13:

1. a spawns actors b and c;
2. a sends c a reference to itself;
3. a becomes idle;
4. The message is dropped, leaving a, b, c quiescent.

The actors a, b, c can never appear quiescent because a sent c a message that c will never receive. The problem is that the snapshots cannot distinguish a configuration with dropped messages from a configuration with undelivered messages. The garbage collector therefore needs to detect when messages (and the references they contain) are dropped to collect all quiescent garbage.

In executions with exiled nodes, it is no longer possible in general to obtain a hereto-closed collage from quiescent actors. This can be seen in Figure 7.14:

1. a is an actor on node N_1;
2. a spawns b onto node N_2;
3. a's node becomes exiled;
4. b becomes idle.

Actor b is quiescent because it is not sticky, does not monitor any other actor, and cannot receive messages from any other actor. But in order for b to appear quiescent, the garbage collector must have snapshots from both a and b—this is impossible, because a can never take a snapshot.

Whereas recovering from dropped message faults is a relatively simple matter of detecting when messages are dropped, recovering from exiled node faults is more complex. It does not

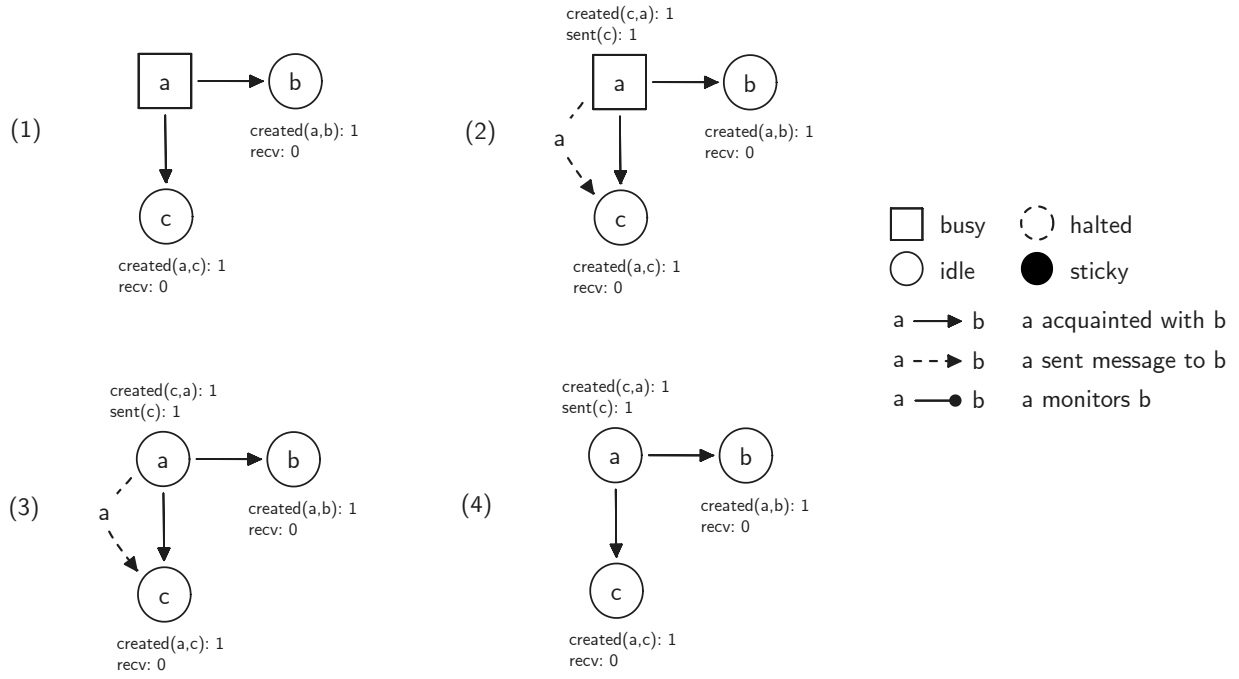


Figure 7.13: An execution in which a dropped message prevents actors from appearing quiescent.

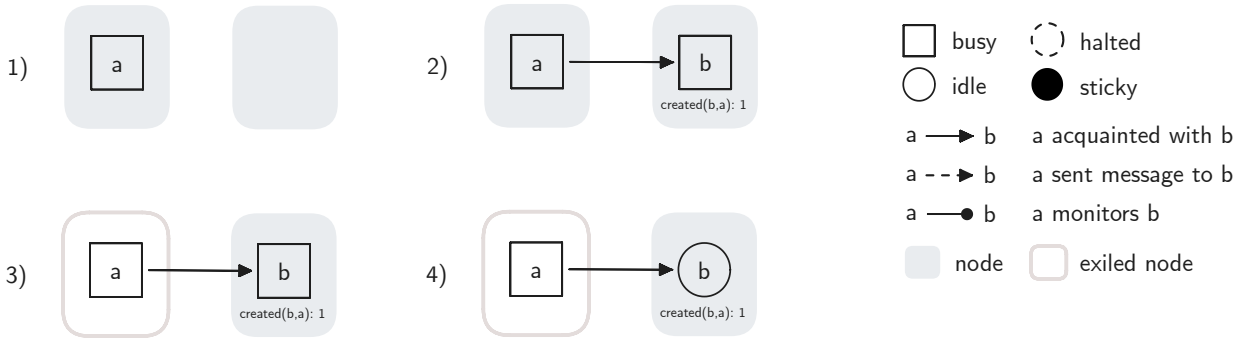


Figure 7.14: An execution in which an exiled node prevents actors from appearing quiescent.

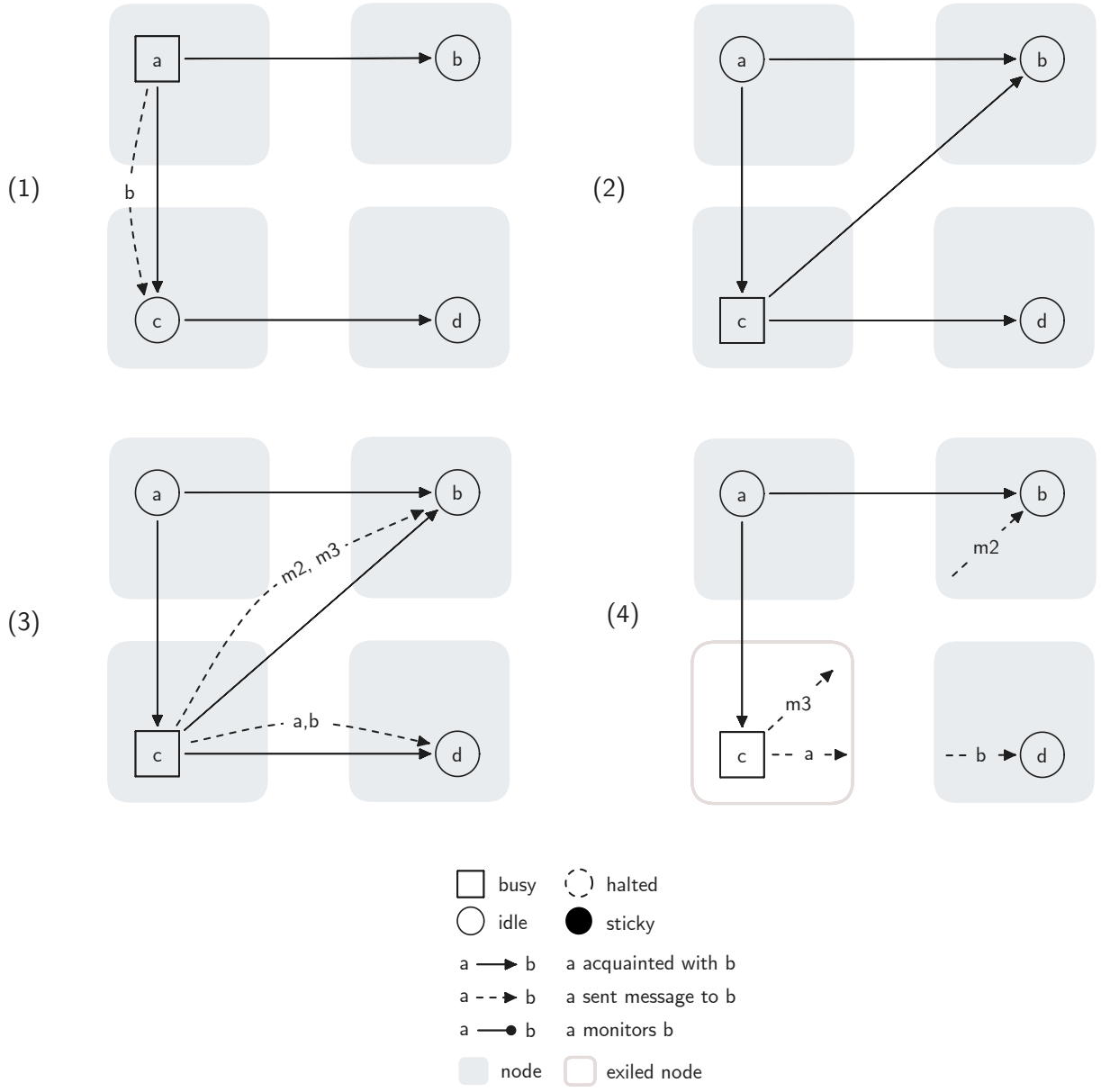


Figure 7.15: An execution in which an exiled node has sent messages, leaving quiescent actors.

suffice to know in Figure 7.14 that a has become exiled. We also need to know:

1. How many messages did a send to b before a became exiled?
2. How many b-references did a send before a became exiled?

This information, which would typically be available in a's snapshot, is inaccessible because a has been removed from the configuration.

Figure 7.15 shows a more complex example of actor garbage in the Exile model:

1. a sends c a reference to b;
2. c receives the reference and becomes busy;
3. c sends b two messages m_2, m_3 and sends d two messages, containing references to a and b;
4. c becomes exiled—but only m_2 and the reference b were admitted. Message m_3 and the reference a will never be delivered because c's node has been shunned by the nodes of b and d.

Our key insight is to observe that, in Figure 7.15, the number of messages that c sent before becoming exiled does not matter. The only messages that can be delivered to b and d are the *admitted* messages from c—messages that have already been delivered to their destination nodes. Our approach in the next section is therefore to instrument actor systems to track admitted messages, thereby compensating for the lack of snapshots from exiled nodes.

7.5.2 *Detecting Quiescence*

Our approach is to introduce mechanisms for the garbage collector to detect faults and then reduce faulty cases into non-faulty cases. For example, once an actor has been notified that a message was dropped, the actor's state will appear "as if" the message were delivered. Similarly, once the garbage collector has been notified that a group of nodes were exiled, it will collect garbage "as if" the exiled actors halted and never sent any messages after they were shunned.

Dropped Messages and Ingress Actors Our fault-detection mechanism is realized by a bag of dropped messages and a family of *ingress actors*.

In the Exile model, dropped messages are not simply removed from the configuration; they are moved to a new bag of *dropped messages*. We assume the implementation has some mechanism for eventually identifying all dropped messages. Actors are then eventually notified about dropped messages by incrementing their received count and incrementing their deactivated(b) count for each reference b the message contained. However, the actor never receives the payload of the message and never becomes busy as a result of the message. This mechanism is modeled by a new event in Exile:

1. *DetectDropped*(a, m): Idle actor a is notified that m was dropped, causing a's state to be updated but for a to remain idle.

Each pair of distinct nodes N_1, N_2 has a system-level *ingress actor* I_{N_1,N_2}, located on N_2, responsible for admitting messages from N_1. Ingress actors record the following information:

- $I_{N,N'}$.shunned: An indication of whether N has been shunned by N'. Once this field is set, messages from N can no longer be admitted to N'.
- $I_{N,N'}$.admittedMsgs(a): The number of messages sent to a, originating from N, that have been admitted.
- $I_{N,N'}$.admittedRefs(a, b): The number of references owned by a and pointing to b, originating from N, that have been admitted.

After a message from N to N' is detected to have dropped, $I_{N,N'}$ admits the message and then notifies the target actor.

Ingress actors, like ordinary actors, take uncoordinated snapshots that can be used by the garbage collector.

> We say node N *appears exiled* if there exist nontrivial groups of actors $\mathcal{G}_1, \mathcal{G}_2$ such that (1) $N \in \mathcal{G}_1$, (2) all nodes are either in $\mathcal{G}_1$ or $\mathcal{G}_2$, and (3) I_{N_1,N_2}.shunned for each $N_1 \in \mathcal{G}_1, N_2 \in \mathcal{G}_2$.
>
> We say a *appears exiled* if a is located on a node that appears exiled. If a appears exiled or appears halted, we say a *appears to have failed.*
>
> We say a has an *effective snapshot* if a has a snapshot in S or a appears exiled.
>
> Message m was *effectively received* if target actor a received m or detected that m was dropped.
>
> For convenience, we will often treat nodes N as sets of actors, so that $a \in N$ if a is located on N. We also write $S(N)$ to denote the actors on N that have taken snapshots, i.e. $\text{dom}(S) \cap N$.

As remarked at the start of Section 7.5.1, the actor GC of the Monitors model is sound in the face of failures. Our approach is therefore to treat each actor a as non-exiled until enough ingress actors have taken snapshots that a appears exiled. Subsequently, we use ingress snapshots in place of a's snapshot to collect any garbage produced when a failed.

Apparent Quiescence We now define what it means for a message to *appear* sent or received (or for a reference to appear created or deactivated) using snapshots from ingress actors. In the following definitions, $\mathcal{G}_1$ is the set of nodes that do not appear exiled, and $\mathcal{G}_2$ is the set of nodes that do appear exiled.

The *apparent send count* for b, located on node N, is

$$\mathsf{sent}(b) = \sum_{N_1 \in \mathcal{G}_1} \sum_{a \in S(N_1)} S(a).\mathsf{sent}(b) + \sum_{N_2 \in \mathcal{G}_2} S(I_{N_2,N}).\mathsf{admittedMsgs}(a). \tag{7.11}$$

The *apparent created count* for b, located on node N, pointing to c is

$$\mathsf{created}(b,c) = \sum_{N_1 \in \mathcal{G}_1} \sum_{a \in S(N_1)} S(a).\mathsf{created}(b,c) + \sum_{N_2 \in \mathcal{G}_2} S(I_{N_2,N}).\mathsf{admittedRefs}(b,c). \tag{7.12}$$

The *apparent receive count* for b is

$$\mathsf{received}(b) = \begin{cases} S(b).\mathsf{received} & \text{if } b \in \mathrm{dom}(S) \\ 0 & \text{otherwise.} \end{cases} \tag{7.13}$$

The *apparent deactivated count* for b to c is

$$\mathsf{deactivated}(b,c) = \begin{cases} S(b).\mathsf{deactivated}(c) & \text{if } b \in \mathrm{dom}(S) \\ 0 & \text{otherwise.} \end{cases} \tag{7.14}$$

Actor a *appears blocked* if it appears idle and its apparent send count is equal to its apparent receive count.

Actor a *appears hereto acquainted* with b if the apparent created count for a pointing to b is greater than zero.

Actor a *appears acquainted* with b if the apparent created count for a pointing to b is greater than the apparent deactivated count.

Collage S *appears hereto-closed* if, for each $b \in \mathrm{dom}(S)$, if a appears hereto acquainted with b then either $a \in \mathrm{dom}(S)$ or a appears exiled.

An actor *appears to have failed* if it appears halted or appears exiled.

Notice how snapshots from apparently exiled actors are replaced in $\mathsf{sent}(b)$ and $\mathsf{created}(b,c)$ by ingress snapshots. Using these definitions, we define "apparent" analogs of strongly and weakly quiescent garbage (Definitions 6.6 and 6.9):

Definition 7.7. Actor b appears *strongly* quiescent in collage S if *a appears to have failed* or all of the following hold:

1. b is appears hereto-closed in S;
2. b appears blocked;
3. b does not appear sticky;
4. If some a appears acquainted with b, then a appears *strongly* quiescent; and
5. If some a appears monitored by b, then a *does not appear to have failed*, appears *strongly* quiescent, and *is not located on a distinct node from* b.

Definition 7.8. Actor b appears *weakly* quiescent in collage S if a *appears to have failed* or all of the following hold:

1. b is appears hereto-closed in S;
2. b appears blocked;
3. b does not appear sticky;
4. If some a appears acquainted with b, then a appears *weakly* quiescent; and
5. If some a appears monitored by b, then a *does not appear to have failed* and appears *weakly* quiescent.

As before, we prove soundness via the invariant (Dynamic-Inv). However, Lemmas 7.2 to 7.4 must be modified to account for *effective* send and receive information.

> S is *hereto-closed* for $a \in \mathrm{dom}(S)$ if every hereto inverse acquaintance of a has an effective snapshot, i.e. a is either in $\mathrm{dom}(S)$ or appears exiled.

Lemma 7.5. If (Dynamic-Inv) holds up to t then S is hereto-closed for Q up to t.

Proof. Let $c \in Q$. By induction on time t, we show that every actor b with a reference to c at time t must either (a) have a snapshot in S, or (b) appear exiled in S.

At $t = 0$, there is only one actor a with a reference to itself. If $a \in Q$ then $a \in \mathrm{dom}(S)$ by definition.

Assuming the property holds up to time t, suppose actor b obtains a reference to c at time $t+1$. There are three possibilities.

1. b spawned c. Then c was spawned with $c.\mathrm{created}(b, c) > 0$ in its state. This implies $S(c).\mathrm{created}(b, c) > 0$. Since Q appears quiescent, b must either appear exiled or have a snapshot in S.
2. $b = c$ and c has just been spawned, thereby obtaining a reference to itself. Again, $c \in Q$ implies $b \in \mathrm{dom}(S)$.
3. b received a reference to c in a message, sent by some a. Let t_s be the time when a sent the message. Actor a must have been acquainted with c when it sent the message, so $a \in \mathrm{dom}(S)$ or a appears exiled by the induction hypothesis. Consider each case:

(a) Case 1 ($a \in \text{dom}(S)$): Notice that $a.\text{created}(b, c) > 0$ at t_s. By (Dynamic-Inv), a could not have taken a snapshot before sending the message. Hence $S(a).\text{created}(b, c) > 0$. Since Q appears quiescent, b would need to either appear exiled or have a snapshot in S.

(b) Case 2 (a appears exiled): If b also appears exiled, the property holds immediately. So suppose b does not appear exiled, implying that a and b are located on different nodes; call these nodes N_a and N_b, respectively. For b to receive the reference, the message must have been admitted to N_b. For a to appear exiled, S must have a snapshot from ingress actor I_{N_a,N_b} that was taken after N_b shunned N_a. Since the message was admitted before N_b shunned N_a, it follows that $S(I_{N_a,N_b}).\text{admittedRefs}(b, c) > 0$. Since Q appears quiescent and b does not appear exiled, b must have a snapshot in S.

QED.

We generalize the notions of forward- and backward-crossing messages to account for apparently exiled actors, which have no snapshot in S, and dropped messages:

> A message m from a to b is *forward-crossing* if b did not effectively receive m before taking a snapshot and one of the following holds:
>
> 1. $a \in \text{dom}(S)$ and m was sent before a's snapshot; or
> 2. a appears exiled and m was admitted.
>
> A message m from a to b is *backward-crossing* if $a \in \text{dom}(S)$ and m was sent after a took a snapshot and m was effectively received before b took a snapshot.

Lemma 7.6. Let $b \in Q$ and let t be the time that b took a snapshot. If (Dynamic-Inv) holds up to t then there are no forward- or backward-crossing messages to b.

Proof. First we show there are no backward-crossing messages. Suppose an actor a took a snapshot at t_a; then a sent a message to b at t_s; and then the message was received before t. Since $t_s \in (t_a, t)$ and a had a reference to b at t_s, (Dynamic-Inv) implies $a \in Q$. But then (Dynamic-Inv) also implies a was idle at t_s, so the message could not have been sent in the first place.

In the remainder, we show there are no forward-crossing messages up to time t. For each actor a in the execution, let $\text{recv}_{a,b}$ equal the number of messages b effectively received from a before t. By Lemma 7.5, the only actors that could have sent b such messages are in $\text{dom}(S)$ or appear exiled in S. Letting $\mathcal{G}_1$ be the set of nodes that do not appear exiled and $\mathcal{G}_2$ be the set of nodes that do appear exiled, it follows that:

$$\sum_{N \in \mathcal{G}_1} \sum_{a \in S(N)} \text{recv}_{a,b} + \sum_{N \in \mathcal{G}_2} \sum_{a \in N} \text{recv}_{a,b} = S(b).\text{received}. \tag{7.15}$$

For each a with an effective snapshot in S, let $\mathsf{crossing}_{a,b}$ equal the number of forward-crossing messages from a to b. If $a \in \mathrm{dom}(S)$, then:

$$S(a).\mathsf{sent}(b) = \mathsf{recv}_{a,b} + \mathsf{crossing}_{a,b} \tag{7.16}$$

because we already proved there are no backward-crossing messages from a to b. Likewise, for each apparently exiled node N:

$$S(I_{N,N_b}).\mathsf{admittedMsgs}(b) = \sum_{a \in N} (\mathsf{recv}_{a,b} + \mathsf{crossing}_{a,b}), \tag{7.17}$$

where N_b is the location of b. Combining Equations (7.16) and (7.17), we find:

$$\begin{aligned} &\sum_{N \in \mathcal{G}_1} \sum_{a \in S(N)} S(a).\mathsf{sent}(b) + \sum_{N_2 \in \mathcal{G}_2} S(I_{N_2,N}).\mathsf{admittedMsgs}(b) = \\ &\sum_{N \in \mathcal{G}_1} \sum_{a \in S(N)} (\mathsf{recv}_{a,b} + \mathsf{crossing}_{a,b}) + \sum_{N \in \mathcal{G}_2} \sum_{a \in N} (\mathsf{recv}_{a,b} + \mathsf{crossing}_{a,b}). \end{aligned} \tag{7.18}$$

Since b appears blocked, we have

$$\sum_{N \in \mathcal{G}_1} \sum_{a \in S(N)} S(a).\mathsf{sent}(b) + \sum_{N_2 \in \mathcal{G}_2} S(I_{N_2,N}).\mathsf{admittedMsgs}(a) = S(b).\mathsf{received}. \tag{7.19}$$

From Equations (7.15), (7.18) and (7.19), it follows that $\mathsf{crossing}_{a,b} = 0$ for every a. Hence b has no forward-crossing messages. QED.

Lemma 7.7. Let $b \in \mathrm{dom}(S)$ and let t be the time that b took a snapshot. If (Dynamic-Inv) holds up to t then, for all $c \in Q$:

1. There are no forward- or backward-crossing c-references to b.
2. Actor b appears acquainted with c if and only if b is *acquainted* with c at time t.

Proof. The fact that there are no backward-crossing references follows from the invariant: Suppose actor a takes a snapshot at t_a; then a sends a c-reference to b at t_s; and then the reference is effectively received before t. To send the reference, a must have had a reference to c at $t_s \in (t_a, t)$. By (Dynamic-Inv), $c \in Q$ implies $a \in Q$. But then (Dynamic-Inv) also implies a was idle at t_s, so the reference could not have been sent in the first place.

We now show that there are no forward-crossing c-references to b. Recall that b can obtain a reference to c in three ways:

1. Some a sent b a reference to c;
2. b spawned c; or

3. $b = c$, so b was spawned with a reference to itself.

For each actor a in the execution, let $\text{activated}_{a,b}$ equal the number of c-references that b effectively received from a before t. Any c-reference that b effectively received before t must have been sent by an actor a that was acquainted with c at some earlier time $t' < t_b$. By Lemma 7.5, actor a either has a snapshot in S or appears exiled in S. Letting $\mathcal{G}_1$ be the set of nodes that do not appear exiled and $\mathcal{G}_2$ be the set of nodes that do appear exiled, it follows that:

$$\text{self} + \sum_{N \in \mathcal{G}_1} \sum_{a \in S(N)} \text{activated}_{a,b} + \sum_{N \in \mathcal{G}_2} \sum_{a \in N} \text{activated}_{a,b} \geq S(b).\text{deactivated}(c), \tag{7.20}$$

where

$$\text{self} = \begin{cases} 1 & \text{if } b = c, \text{ or if } b \text{ spawned } c \text{ before } t \\ 0 & \text{otherwise} \end{cases} \tag{7.21}$$

That is, b could only have deactivated the c-references that it obtained from itself, actors in $\text{dom}(S)$, or actors that appear exiled.

For each a with an effective snapshot in S, let $\text{crossing}_{a,b}$ equal the number of forward-crossing c-references from a to b. Hence, because we already proved there are no backward-crossing c-references,

$$S(a).\text{created}(b, c) = \begin{cases} \text{activated}_{a,b} + \text{crossing}_{a,b} & \text{if } a \neq b \\ \text{activated}_{b,b} + \text{crossing}_{b,b} + \text{self} & \text{otherwise} \end{cases} \tag{7.22}$$

Likewise, for each apparently exiled node N,

$$S(I_{N,N_b}).\text{admittedRefs}(b, c) = \sum_{a \in N} (\text{activated}_{a,b} + \text{crossing}_{a,b}). \tag{7.23}$$

Combining Equations (7.22) and (7.23), we find:

$$\begin{aligned} &\sum_{N \in \mathcal{G}_1} \sum_{a \in S(N)} S(a).\text{created}(b, c) + \sum_{N_2 \in \mathcal{G}_2} S(I_{N_2,N_b}).\text{admittedRefs}(b, c) = \\ &\text{self} + \sum_{N \in \mathcal{G}_1} \sum_{a \in S(N)} (\text{activated}_{a,b} + \text{crossing}_{a,b}) + \sum_{N \in \mathcal{G}_2} \sum_{a \in N} (\text{activated}_{a,b} + \text{crossing}_{a,b}). \end{aligned} \tag{7.24}$$

There are two cases.

Case 1 ($\Rightarrow$). Assume b appears acquainted with c. Since $c \in Q$ and appears quiescent, $b \in Q$. Then Lemma 7.6 implies there are no forward-crossing messages to b. Hence, in particular there can be no forward-crossing c-references to b.

Next we show that b is acquainted with c at time t. Lemma 7.6 implies $\text{crossing}_{a,b} = 0$ for all a, so

$$\begin{aligned}\sum_{N\in\mathcal{G}_1}\sum_{a\in S(N)} S(a).\text{created}(b,c) + \sum_{N_2\in\mathcal{G}_2} S(I_{N_2,N_b}).\text{admittedRefs}(b,c) = \\ \text{self} + \sum_{N\in\mathcal{G}_1}\sum_{a\in S(N)} \text{activated}_{a,b} + \sum_{N\in\mathcal{G}_2}\sum_{a\in N} \text{activated}_{a,b}.\end{aligned} \tag{7.25}$$

Also, since b appears acquainted with c,

$$\sum_{N\in\mathcal{G}_1}\sum_{a\in S(N)} S(a).\text{created}(b,c) + \sum_{N_2\in\mathcal{G}_2} S(I_{N_2,N_b}).\text{admittedRefs}(b,c) > S(b).\text{deactivated}. \tag{7.26}$$

Hence

$$\text{self} + \sum_{N\in\mathcal{G}_1}\sum_{a\in S(N)} \text{activated}_{a,b} + \sum_{N\in\mathcal{G}_2}\sum_{a\in N} \text{activated}_{a,b} > S(b).\text{deactivated}, \tag{7.27}$$

i.e. b has at least one reference to c at time t that has not been deactivated.

Case 2 ($\Leftarrow$). Assume b does not appear acquainted with c. Applying Equation (7.24), this implies:

$$S(b).\text{deactivated}(c) \geq \tag{7.28}$$

$$\text{self} + \sum_{N\in\mathcal{G}_1}\sum_{a\in S(N)} (\text{activated}_{a,b} + \text{crossing}_{a,b}) + \sum_{N\in\mathcal{G}_2}\sum_{a\in N} (\text{activated}_{a,b} + \text{crossing}_{a,b}) \geq \tag{7.29}$$

$$\text{self} + \sum_{N\in\mathcal{G}_1}\sum_{a\in S(N)} \text{activated}_{a,b} + \sum_{N\in\mathcal{G}_2}\sum_{a\in N} \text{activated}_{a,b}. \tag{7.30}$$

Combining the above inequality with Equation (7.20), we deduce that $\text{crossing}_{a,b} = 0$ for all a, i.e. there are no forward-crossing c-references. Moreover, every reference b obtained for c has been deactivated—so b is not acquainted with c at time t. QED.

Theorem 7.5 (Strong Soundness)**.** Let S be a collage and let Q be a subset of $\text{dom}(S)$ that appears hereto-closed and appears strongly quiescent. Then Q is strongly quiescent.

Proof. The result follows from the invariant (Dynamic-Inv). We prove the invariant holds at all times by contradiction. Suppose t is the first time the invariant is violated. We consider the two ways the invariant could be violated at time t.

Case 1. An actor $b \in Q$ is busy and has taken a snapshot. Because b was idle when it took the snapshot, it must have become busy at time t. Recall that actors can become busy in four ways:

1. *b received a message.* By Lemma 7.5, there are two possibilities.
   (a) The message was sent at time $t_s < t$ by an actor $a \in \text{dom}(S)$. If a sent the message

before taking a snapshot, then the message is forward-crossing and therefore forbidden by Lemma 7.6. Otherwise, since (Dynamic-Inv) still held at time $t_s < t$, actor a must be in Q and must have been idle at t_s—making it impossible for a to have sent the message.

(b) The message was sent by an actor that appears exiled. Such a message is forward-crossing and therefore forbidden by Lemma 7.6.

2. *b is sticky and received a wakeup signal.* This is impossible because b was not sticky at the time of its snapshot, so b would have had to become busy at an earlier time $t' < t$ to become sticky.
3. *b monitors an actor c and is notified that c halted.* This is only possible if c was busy at some $t' < t$ and then c halted. Notice that b must have been monitoring c at the time of b's snapshot so, because b appears quiescent, $c \in Q$. Because $c \in Q$ and b is the first actor in Q to become busy, c cannot have halted after taking a snapshot. Therefore c halted before taking a snapshot—but then c appears halted, contradicting the fact that b appears quiescent.
4. *b monitors a remote actor c and is notified that c became exiled.* This is impossible because b did not monitor any remote actors at the time of its snapshot.

Case 2. An actor $b \in \mathrm{dom}(S) \setminus Q$ has taken a snapshot and holds a reference to some $c \in Q$. By Lemma 7.7, b could not have been acquainted with c at the time of b's snapshot, because then b would appear acquainted with c, implying $b \in Q$. Hence b received the reference to c at time t, sent by some actor a at time $t_s < t$. By Lemma 7.5, $a \in \mathrm{dom}(S)$ or a appears exiled. By Lemma 7.7, the reference cannot be forward-crossing; this implies a cannot appear exiled because b did not receive the message before taking a snapshot. So $a \in \mathrm{dom}(S)$.

Since a has a snapshot in S and the reference is not forward-crossing, a must have sent the message after its snapshot. This implies $a \in Q$ and that a was idle at t_s, by the invariant and the fact that a had a reference to c at t_s. Hence a could not have sent b the reference.

Thus (Dynamic-Inv) holds for all times t and all actors in Q remain idle after taking snapshots. This implies that, at time t_f when all actors in Q have taken snapshots, the actors of Q are all garbage.

QED.

Theorem 7.6 (Weak Soundness)**.** Let S be a collage and let Q be an apparently hereto-closed subset of $\mathrm{dom}(S)$ that appears weakly quiescent If, for each $a \in Q$, the actors monitored by a do not become exiled after a takes a snapshot, then Q is weakly quiescent.

Proof. Identical to the preceding proof, except for the case when a remote actor becomes exiled. This case is impossible by the assumption that monitored actors do not become exiled. QED.

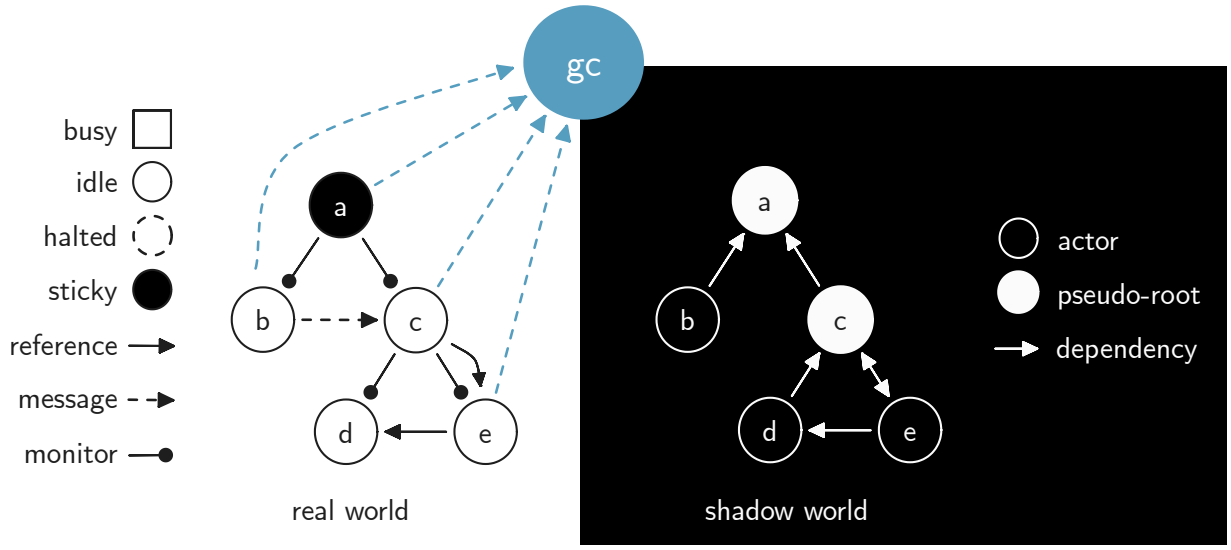


Figure 7.16: Actors (left) send snapshots to the garbage collector, which uses the snapshots to construct a shadow graph (right).

7.6 Shadow Graphs and Undo Logs

So far in the chapter, collages have been encoded as collections of snapshots. However, this encoding is not space efficient: since every actor records the number of messages sent to every other actor, the message counts collectively use $O(n^2)$ space. Similarly, reference creation counts use $O(n^3)$ space. This section presents *shadow graphs*, a more efficient way to represent collages (Figure 7.16). For now, let us ignore the failure scenarios handled in Section 7.5.1; failures will be handled with *undo logs*, presented in Section 7.6.2.

7.6.1 Shadow Graphs

A shadow graph is a collection of *shadows*—one for each actor occurring in the collage S. Formally:

Definition 7.9. Actor b *occurs in* collage S if any of the following hold:

1. $b \in \mathrm{dom}(S)$, or
2. There exists $a \in \mathrm{dom}(S)$ such that:
   (a) $\exists c,\ S(a).\mathrm{created}(b, c) > 0$;
   (b) $\exists c,\ S(a).\mathrm{created}(c, b) > 0$;
   (c) $S(a).\mathrm{sent}(b) > 0$;
   (d) $S(a).\mathrm{deactivated}(b) > 0$; or
   (e) $b \in S(a).\mathrm{monitored}$

Definition 7.10. Let b be any actor, not necessarily in the domain of S. We define the apparent message counts *according to the ordinary actors of* S as follows:

$$\mathsf{sent}_S(b) = \sum_{a \in \mathrm{dom}(S)} S(a).\mathsf{sent}(b) \tag{7.31}$$

$$\mathsf{created}_S(b, c) = \sum_{a \in \mathrm{dom}(S)} S(a).\mathsf{created}(b, c) \tag{7.32}$$

In contrast to the apparent message counts of Section 7.5.1, these counts do not use snapshots from ingress actors. They are therefore identical to the message counts in the Monitors model.

Definition 7.11. Let b be an actor and S a collage. Then the *shadow* of b is a 6-tuple s with the following components:

$$s.\mathsf{interned} = \begin{cases} \textsc{True} & \text{if } b \in \mathrm{dom}(S) \\ \textsc{False} & \text{otherwise} \end{cases} \tag{7.33}$$

$$s.\mathsf{status} = \begin{cases} S(b).\mathsf{status} & \text{if } b \in \mathrm{dom}(S) \\ \textsc{Undefined} & \text{otherwise} \end{cases} \tag{7.34}$$

$$s.\mathsf{isSticky} = \begin{cases} S(b).\mathsf{isSticky} & \text{if } b \in \mathrm{dom}(S) \\ \textsc{Undefined} & \text{otherwise} \end{cases} \tag{7.35}$$

$$s.\mathsf{watchers} = \{a : b \in S(a).\mathsf{monitored}\} \tag{7.36}$$

$$s.\mathsf{undelivered} = \mathsf{sent}_S(b) - \mathsf{received}(b) \tag{7.37}$$

$$s.\mathsf{references}(c) = \mathsf{created}_S(b, c) - \mathsf{deactivated}(b, c) \tag{7.38}$$

Definition 7.12. Let S be a collage. The *shadow graph* of S is a finite map G, associating the actors occurring in S with their shadows.

Notice that 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 a created. In a shadow graph, message counts are collapsed into a single field, $G(b).\mathsf{undelivered}$. Likewise, each reference created for b is accounted for in the map $G(b).\mathsf{references}$.

Shadow graphs also encode some information differently than collages. Whereas 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. We set the bit $G(b)$.interned to indicate whether b has a snapshot in S. In addition, the field $G(b)$.watchers is the inverse of $S(b)$.monitored; it stores the set of actors that *monitor* b, instead of the actors that are monitored by b.

Although shadow graphs have less information than collages, we now show that shadow graphs retain enough information to identify garbage. Our terminology is inspired by tracing garbage collectors, which proceed by identifying a "root set" and marking all the objects that are "reachable" from that set.

Definition 7.13. Actor b is a *pseudo-root* in shadow graph G if any of the following hold:

1. $G(b)$.interned is false;
2. $G(b)$.isSticky is true;
3. $G(b)$.status is "busy";
4. $G(b)$.undelivered $\neq 0$; or
5. $b \in G(a)$.watchers for some a, where $G(a)$.status is "halted".

Definition 7.14. Actor b is *marked* in shadow graph G if it does not appear faulty and any of the following hold:

1. b is a pseudo-root; or
2. There exists a marked actor $a \in \mathrm{dom}(G)$ such that:
   (a) $G(a)$.references$(b) > 0$; or
   (b) $G(a)$.watchers contains b.

If $b \in \mathrm{dom}(G)$ and b is not marked, we say b is *unmarked*.

In Figure 7.16, actors a and c are pseudo-roots; actors a, c, d, and e are marked; and b is unmarked.

In the absence of faults, the unmarked actors in G are the same actors that appear quiescent in S. The proof is a special case of Theorem 7.8, presented in the next section.

Theorem 7.7. Assume no actors appear exiled. Then actor b appears quiescent in collage S if and only if b is unmarked in its shadow graph G.

7.6.2 Undo Logs

When a node N is exiled, the snapshots from actors on N should be replaced with snapshots from ingress actors. 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*.

An undo log indicates how the shadow graph should be modified when a specific node is exiled. Undo logs are constructed from a collage S by combining actor snapshots with ingress snapshots.

In this section, we show how undo logs are constructed and how they can be used to *amend* a shadow graph when nodes are exiled. We conclude with a proof that the unmarked actors in the amended shadow graph correspond to the quiescent actors in the Exile model.

Definition 7.15. Let b be an actor and N a node. We define the message counts *according to* N as follows:

$$\mathsf{sent}_{S,N}(b) = \sum_{a \in S(N)} S(a).\mathsf{sent}(b) \tag{7.39}$$

$$\mathsf{created}_{S,N}(b,c) = \sum_{a \in S(N)} S(a).\mathsf{created}(b,c) \tag{7.40}$$

Definition 7.16. Given a collage S, we define the *undo log* L for node N to be a record with the following fields:

$$L.\mathsf{undeliverableMsgs}(b) = \mathsf{sent}_{S,N}(b) - S(I_{N,N_b}).\mathsf{admittedMsgs}(b) \tag{7.41}$$

$$L.\mathsf{undeliverableRefs}(b,c) = \mathsf{created}_{S,N}(b,c) - S(I_{N,N_b}).\mathsf{admittedRefs}(b,c), \tag{7.42}$$

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

Undo logs are the garbage collector's view of the network - namely, the messages sent from N to actor a that have not yet been admitted to a's node. Thus, whereas *ingress snapshots* grow without bound as the system evolves, we expect the undo logs to be small as long as snapshots from ordinary actors and ingress actors are "up to date".

Remarkably, these small undo logs are all we need to recover garbage after nodes have been exiled. Below we define the *amended* shadow graph, produced by merging the undo logs of apparently exiled nodes into the shadow graph presented in Section 7.6.1.

Definition 7.17. Let G be a shadow graph, let $N_1, \ldots, N_k$ be the set of nodes that appear exiled, and let $L_1, \ldots, L_k$ 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. $a \in \mathrm{dom}(\tilde{G})$ if $a \in \mathrm{dom}(G)$ and either:
   (a) a does not appear exiled, or
   (b) a appears exiled and $G(a).\mathsf{watchers}$ contains actors that do not appear faulty;
2. $\tilde{G}(a).\mathsf{status} = \mathsf{halted}$ if a appears exiled and $G(a).\mathsf{watchers}$ contains actors that do not appear faulty;

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 faulty; 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. In addition, the amended graph repairs reference counts and message counts to account for messages that will never be delivered.

We conclude this section with a proof that unmarked actors in $\tilde{G}$ correspond to the actors in S that appear quiescent.

Lemma 7.8. For each $b \in \mathrm{dom}(G)$, $\tilde{G}(b)$.undelivered = sent(b) − received(b).

Proof. By the definitions of shadow graphs, merging, and undo logs:

$$\tilde{G}(b).\mathsf{undelivered} = \mathsf{sent}_S(b) - \mathsf{received}(b) \tag{7.43}$$

$$- \sum_{N_2 \in \mathcal{G}_2} \left(\mathsf{sent}_{S,N}(b) - S(I_{N_2,N_b}).\mathsf{admittedMsgs}(b)\right) \tag{7.44}$$

where $\mathcal{G}_2$ is the set of nodes that appear exiled. Notice that $\mathsf{sent}_S(b)$ accounts for all nodes:

$$\mathsf{sent}_S(b) = \sum_{N_1 \in \mathcal{G}_1} \mathsf{sent}_{S,N_1}(b) + \sum_{N_2 \in \mathcal{G}_2} \mathsf{sent}_{S,N_2}(b). \tag{7.45}$$

Hence,

$$\tilde{G}(b).\mathsf{undelivered} = \sum_{N_1 \in \mathcal{G}_1} \mathsf{sent}_{S,N_1}(b) - \mathsf{received}(b) \tag{7.46}$$

$$+ \sum_{N_2 \in \mathcal{G}_2} S(I_{N_2,N}).\mathsf{admittedMsgs}(b). \tag{7.47}$$

Hence $\tilde{G}(b)$.undelivered is the difference between the apparent send count and the apparent receive count. QED.

Lemma 7.9. For each $b, c \in \mathrm{dom}(G)$,

$$\tilde{G}(b).\mathsf{references}(c) = \mathsf{created}(b, c) - \mathsf{deactivated}(b, c). \tag{7.48}$$

Proof. By the definitions of shadow graphs, merging, and undo logs:

$$\tilde{G}(b).\text{references}(c) = \text{created}_S(b,c) - \text{deactivated}(b,c) \\ - \sum_{N_2 \in \mathcal{G}_2} \left(\text{created}_{S,N_2}(b,c) - S(I_{N_2,N_b}).\text{admittedRefs}(b,c)\right) \quad (7.49)$$

where $\mathcal{G}_2$ is the set of nodes that appear exiled. Notice that $\text{created}_S(a,b)$ accounts for all nodes:

$$\text{created}(b,c) = \sum_{N_1 \in \mathcal{G}_1} \text{created}_{S,N_1}(a,b) + \sum_{N_2 \in \mathcal{G}_2} \text{created}_{S,N_2}(a,b) \quad (7.50)$$

Hence,

$$\tilde{G}(b).\text{references}(c) = \sum_{N_1 \in \mathcal{G}_1} \text{created}_{S,N_1}(a,b) - \text{deactivated}(b,c) \quad (7.51)$$

$$+ \sum_{N_2 \in \mathcal{G}_2} S(I_{N_2,N_b}).\text{admittedRefs}(b,c). \quad (7.52)$$

Hence $\tilde{G}(b).\text{references}(c)$ is the difference between the apparent created count and the apparent deactivated count. QED.

Lemma 7.10. For each b occurring in S and $c \in \text{dom}(\tilde{G})$, if $b \in \tilde{G}(c).\text{watchers}$ then $b \in \text{dom}(\tilde{G})$ and b appears to monitor c in S. Conversely, for each b, c occurring in S, if b appears to monitor c in S and b does not appear faulty, then $b, c \in \text{dom}(\tilde{G})$ and $b \in \tilde{G}(c).\text{watchers}$.

Proof. By definition, $b \in \tilde{G}(c).\text{watchers}$ implies b does not appear faulty and $c \in S(b).\text{monitored}$. Hence b appears to monitor c. Also, since b occurs in S and does not appear faulty, $b \in \text{dom}(\tilde{G})$.

Conversely, if b appears to monitor c in S then $b \in G(c).\text{watchers}$. Since b does not appear faulty, $b, c \in \text{dom}(\tilde{G})$ and $b \in \tilde{G}(c).\text{watchers}$. QED.

Lemma 7.11. Actor b appears hereto-closed in S if and only if every a with a reference to b in $\tilde{G}$ is interned.

Proof. We prove each direction by contrapositive.

If b does *not* appear hereto-closed in S, then there exists an a where $\text{created}(a,b) > 0$ and $a \notin \text{dom}(S)$. This implies $\text{deactivated}(a,b) = 0$ and a, b both occur in S; hence $\tilde{G}(a).\text{references}(b) > 0$ by Lemma 7.9. In addition, $a \notin \text{dom}(S)$ implies $\tilde{G}(a).\text{interned}$ is false. Hence there exists $a \in \text{dom}(\tilde{G})$ that is not interned and has a reference to b.

Conversely, if there exists a that is not interned in the shadow graph and $G(a).\text{references}(b) > 0$, then $a \notin \text{dom}(S)$ and a must appear acquainted with b in S. Hence a is an apparent hereto inverse acquaintance of b and $a \notin \text{dom}(S)$. QED.

Lemma 7.12. $\tilde{G}(a)$.status is halted if and only if $a \in \mathrm{dom}(\tilde{G})$ appears faulty.

Proof. If $\tilde{G}(a)$.status is halted then $S(a)$.status is halted or a appears exiled. Conversely, if $a \in \mathrm{dom}(\tilde{G})$ and a appears faulty then $\tilde{G}(a)$.status is halted by definition. QED.

Theorem 7.8. 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.

Proof. It suffices to prove the contrapositive, that b is marked if and only if b does not appear weakly quiescent.

($\Rightarrow$): By induction on the definition of marked actors. If b is a pseudo-root, then one of the following must hold:

1. $b \notin \mathrm{dom}(S)$;
2. b appears busy or sticky;
3. $\tilde{G}(b)$.undelivered > 0, implying b appears unblocked by Lemma 7.8; or
4. $b \in \tilde{G}(a)$.watchers for some a that appears faulty, implying b appears to monitor an apparently faulty actor by Lemma 7.10.

In each of these cases, b cannot appear weakly quiescent For the induction step, let a be a marked actor. By the induction hypothesis, a does not appear weakly quiescent. There are two cases:

1. $\tilde{G}(a)$.references$(b) > 0$. By Lemma 7.9, a appears acquainted with b.
2. $b \in \tilde{G}(a)$.watchers. By Lemma 7.10, b appears to monitor an actor that does not appear weakly quiescent.

In both cases, b cannot appear weakly quiescent.

($\Leftarrow$): If b does not appear weakly quiescent, then b appears non-faulty and one of the following must hold:

1. b does not appear hereto-closed;
2. b appears unblocked or sticky;
3. There exists a that does not appear weakly quiescent, and a appears acquainted with b;
4. There exists a that appears failed or does not appear weakly quiescent, and a appears to be monitored by b.

We proceed by induction on these cases.

Base case: If b does not appear hereto-closed, then Lemma 7.11 implies b is marked. If b appears unblocked or sticky, then Lemma 7.9 implies b is a pseudo-root.

Induction step: Let a be an actor that does not appear weakly quiescent. Hence a appears non-faulty and, by the induction hypothesis, a is marked. If a appears acquainted with b, then

$\tilde{G}(a)$.references(b) > 0 by Lemma 7.9, so b is marked. Otherwise, if a appears monitored by b, then $b \in \tilde{G}(a)$.watchers by Lemma 7.10, so b is marked.

Now let a be an actor that appears to have failed. If a appears monitored by b, then $b \in \tilde{G}(a)$.watchers by Lemma 7.10. Also, by Lemma 7.12, $\tilde{G}(a)$.status is halted. Hence b is a pseudo-root. QED.

8

Implementation

The previous chapter showed that quiescent actors can be detected by creating a shadow graph and tracing the graph to find actors that "appear" quiescent. This chapter presents *CRGC* (Conflict-Replicated Garbage Collection):[6] a high-performance fault-recovering actor GC for Akka that combines the collage-based approach of Chapter 7 with the theory of conflict-replicated data types (CRDTs) [73]. CRGC is currently capable of detecting distributed actor garbage and recovering from crashed node faults. Using the techniques in Chapter 7, it can also be extended to recover from dropped message faults and to support Akka's monitoring mechanism. The architecture uses the shadow graph and undo log data structures from the previous chapter and introduces two new concepts: *diary entries* and *delta graphs*.

8.1 Overview

The architecture of CRGC is depicted in Figures 8.1 to 8.3. Every node has a local garbage collector (GC) with its own shadow graph. Actors send incremental updates, called *diary entries* or simply *entries*, to be merged into their GC's shadow graph. Each GC periodically wakes up, traces its shadow graph, and kills any local actors that remain unmarked in the graph.

To detect distributed garbage, GCs also need entries from remote actors. However, broadcasting entries from all actors to all GCs would be data-intensive. Instead, actors only send entries to their *local* GCs, which combine batches of entries into efficient summaries called *delta graphs*.[7] GCs broadcast their delta graphs to one another and merge incoming delta graphs into their shadow graph, thereby obtaining eventually consistent views of the entire cluster.

For fault recovery, every node N has an ingress actor for each adjacent node N' in the cluster. Ingress actors track incoming messages as in the previous chapter, and broadcast diary entries to all the GCs. The GCs continually combine all incoming ingress entries and delta graphs to create

[6] Available at `https://github.com/dplyukhin/uigc`.

[7] Unlike the summaries in Section 5.3.2, delta graphs may contain snapshots from local actors that other nodes do not need to know about. Developing a more compact representation of delta graphs is a subject for future work.

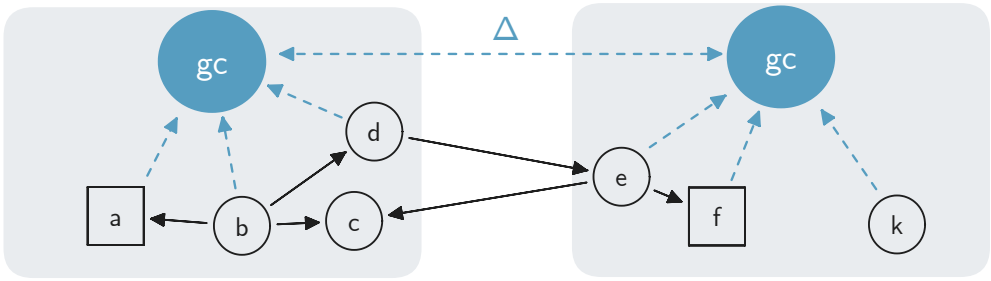


Figure 8.1: Two nodes using CRGC. Each node has a local garbage collector, and each actor sends entries to the garbage collector on its node. Garbage collectors exchange delta graphs.

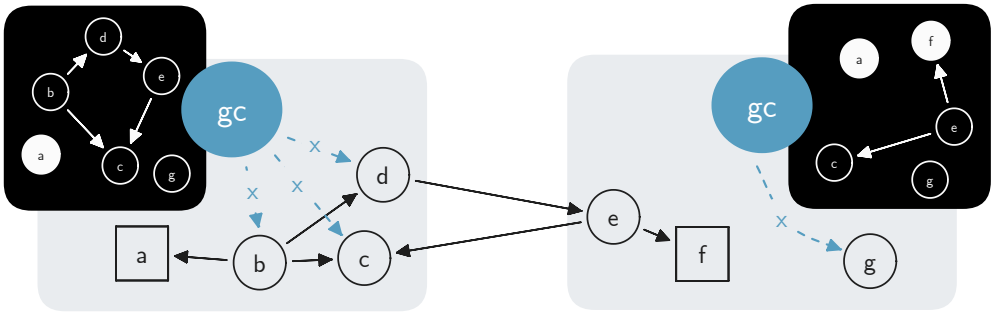


Figure 8.2: Two nodes using CRGC to collect garbage. Each GC has a shadow graph, representing its view of the cluster. When a GC finds one of its local actors is garbage, it asks the actor to stop.

undo logs—one undo log for each other node in the cluster. Thus, if node N' is exiled, then node N can recover from the failure by merging its undo log for N' into its shadow graph.

8.2 Diary Entries

In the previous chapter, each actor maintained a cumulative history of all the GC-related actions it performed. For example, if a sent b a reference, then a's state would record $a.\text{created}(b) > 0$ for eternity. This is inefficient because it allows actor states to grow without bound (causing a memory leak) and because each snapshot duplicates some of the information the actor sent in its previous snapshot.

In CRGC, actors maintain small, fixed-size diary entries where they record only their most recent GC-related actions. When an entry becomes full, the actor *finalizes* the entry by sending it to the actor's local garbage collector and allocating a fresh entry for future use. As long as entries are never dropped and always received in FIFO order, the set of entries a garbage collector received from actor a up to time t combines to form a snapshot of a at t.

Actors can also finalize entries before they are full. Doing so is necessary because the GC in

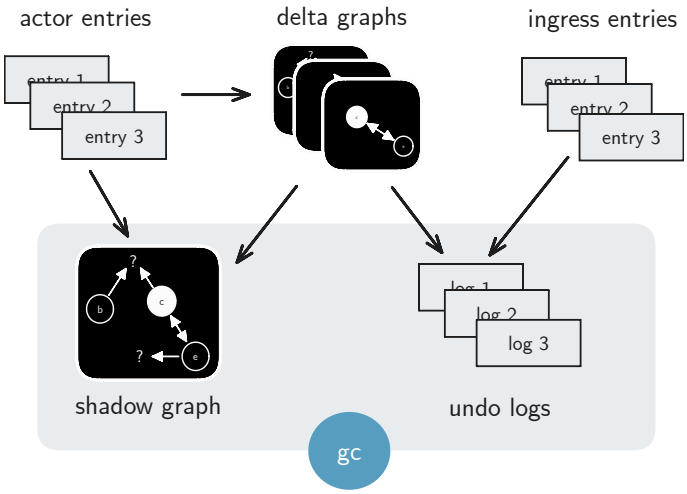


Figure 8.3: Dataflows in CRGC. Local actor entries are merged into the GC's shadow graph and used to produce delta graphs for other nodes. Delta graphs from remote nodes are merged into the shadow graph to detect distributed garbage. Delta graphs are also combined with ingress entries to produce undo logs.

Chapter 7 cannot detect that an actor b is quiescent unless the GC has a "recent enough" snapshot from b. In addition, if some non-quiescent actor a ever had a reference to b, the GC will not detect that b is quiescent until the GC knows the reference is deactivated. Hence, both quiescent and non-quiescent actors should always eventually finalize their entries.

Fortunately, our approach affords a great deal of flexibility in choosing policies for actors to finalize entries. The policy used in the current implementation is borrowed from MAC [10]: actors finalize entries whenever their mail queues become empty. Alternatively, we could choose a policy where the garbage collector explicitly asks actors to finalize their entries, or some hybrid of both policies.

8.2.1 Performance Optimizations

The design of CRGC and the flexibility of collage-based collection allow for interesting performance optimizations. In existing actor GCs, memory allocation is a source of overhead. This is particularly a problem for reference listing algorithms such as PRL (Part I) and the SALSA GC [18], because resizing the reference list requires freeing and allocating memory. Dynamic allocations take memory away from the application and create work for the object GC (e.g. the G1 collector on the JVM). By giving entries a fixed size in CRGC, nodes can use an object pool for entries and thereby eliminate the memory management cost of finalizing an entry.

In the current implementation of CRGC, there is only one garbage collector per node. This

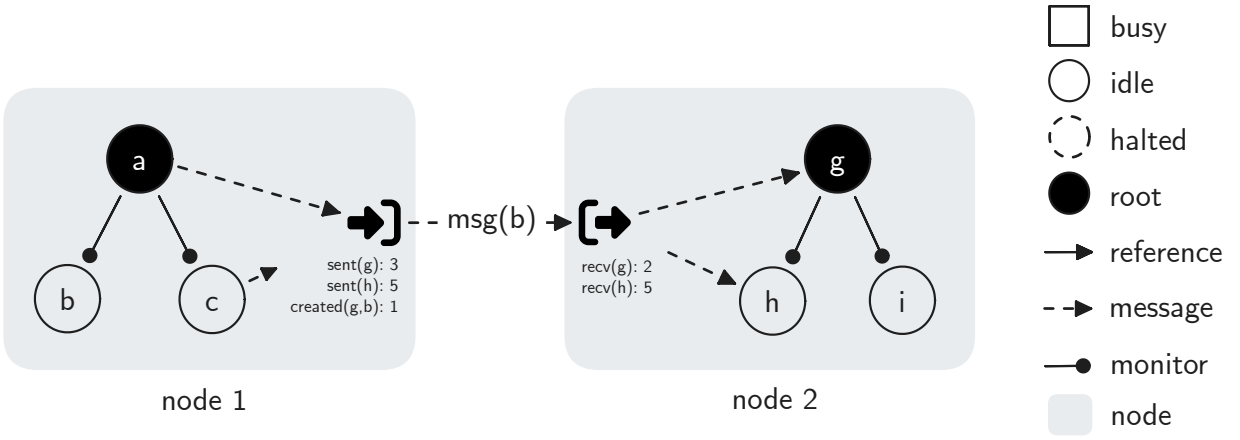


Figure 8.4: Ingress and egress actors in CRGC.

design has been adequate for our benchmarks, but it could lead to significant contention on the garbage collector's mail queue. Mailbox contention is a documented performance problem in Pony's cycle detector, MAC [74]. But CRGC, unlike MAC, does not require causal ordering and therefore does not require a centralized mail queue. To reduce contention, we can allocate multiple mail queues for the garbage collector. When actors finalize entries, they can use consistent hashing to select a mail queue. Thus the probability of two threads writing to the same queue is reduced, while ensuring that actor entries are received in FIFO order.

8.3 Shadow Graphs and Delta Graphs

Shadow graphs, introduced in Section 7.6, are an efficient representation of collages. Each node in the shadow graph, called a shadow, represents an actor in the collage. Shadows include fields to indicate whether they are pseudo-roots, and there is also a (weighted) edge from actor a's shadow to actor b's shadow if a appears acquainted with b. Garbage collectors can find quiescent actors by identifying pseudo-roots and tracing all the shadows reachable from those pseudo-roots; as in tracing garbage collectors [28], any unmarked actors in resulting graph are guaranteed to be garbage. However, in contrast to other concurrent garbage collectors, tracing the shadow graph can be done without read or write barriers because the shadow graph is only modified by the garbage collector itself.

Delta graphs are simply shadow graphs that can be serialized and sent to remote nodes. Delta graphs consume less memory than entries for the reasons listed in Section 7.6. Similarly to entries, delta graphs can be *merged* into shadow graphs: merging a delta graph Δ into a shadow graph G has the same effect as merging all the entries used to construct Δ directly into G.

8.4 Ingress and Egress Actors

Chapter 7 introduced the concept of an *ingress actor* that tracks the number of messages and references that have been admitted to a node. We also assumed that nodes had some mechanism for detecting dropped messages and references. In CRGC, this information is obtained by inserting ingress actors and *egress actors* (Figure 8.4) between every pair of nodes in the cluster. Ingress actors count incoming messages and references, as in Section 7.5.1; egress actors count *outgoing* messages and references, so that the two actors can cooperate to detect dropped messages.

When node N_1 sends a message to node N_2, the egress actor E_{N_1,N_2} at N_1 does the following:

1. Check the message recipient a and the set of references R in the message; and
2. Increment $E_{N_1,N_2}.\mathsf{sentMsgs}(a)$ and increment $E_{N_1,N_2}.\mathsf{sentRefs}(a, b)$ for each $b \in R$.

Likewise, when N_2 receives a message from N_1, the ingress actor I_{N_1,N_2} does the following:

1. Check the message recipient a and the set of references R in the message; and
2. Increment $I_{N_1,N_2}.\mathsf{admittedMsgs}(a)$ and increment $I_{N_1,N_2}.\mathsf{admittedRefs}(a, b)$ for each $b \in R$.

Thus at all times, the difference between $E_{N_1,N_2}.\mathsf{sentMsgs}(a)$ and $I_{N_1,N_2}.\mathsf{admittedMsgs}(a)$ is equal to the number of in-flight messages from N_1 to N_2 addressed to a. By using a communication protocol that guarantees messages are not delivered out-of-order, the egress actor and ingress actor can cooperate to determine how many messages and references were dropped [14].

9

Evaluation

In this section, we measure the overhead of CRGC on single machines using an established set of benchmarks—the *Savina* benchmark suite [59]—and evaluate the distributed performance of CRGC using a new *configurable* benchmark, designed to illustrate a variety of realistic workloads.

9.1 Savina Benchmarks

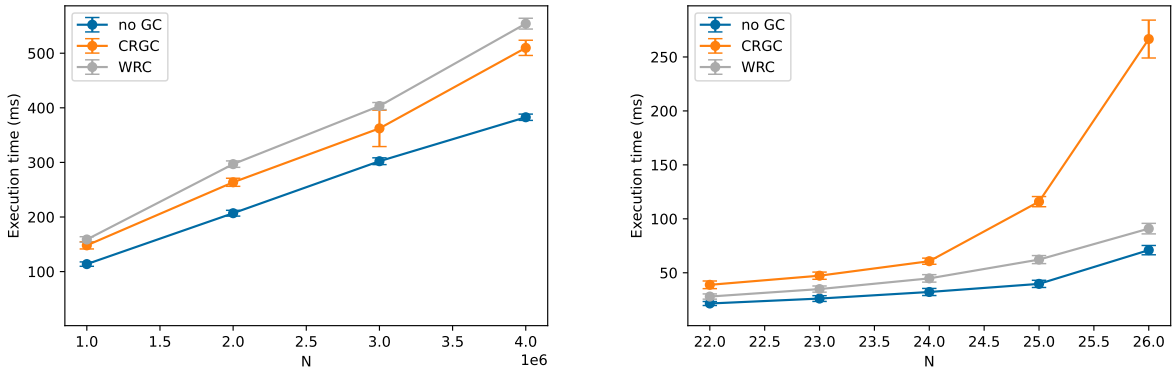
Savina is a well-established benchmark suite for actor frameworks [59]. The suite contains implementations of microbenchmarks, concurrency benchmarks, and parallelism benchmarks using a number of actor frameworks, including Akka.

Because few actor frameworks implement actor GC, all of the implemented benchmarks use manual actor garbage collection—and many do not produce any actor garbage at all. The purpose of the benchmarks is to measure the performance of various actor features, such as messaging throughput and rate of actor creation. By porting benchmarks from the Savina suite to use CRGC, we measure the worst-case performance overhead of using automatic garbage collection in applications where automatic garbage collection may not be necessary. The results in this section show that automatic garbage collection can have a non-negligible impact on certain microbenchmarks, but in practical benchmarks this performance impact vanishes.

In the original benchmarks, execution time included the time for all actors in the system to terminate themselves. As remarked by Blessing et al [60], this does not accurately reflect real actor systems and unfairly penalizes garbage collectors for running less often than necessary. This chapter therefore uses a slightly modified version of the Savina suite, in which termination time is excluded from measurements.

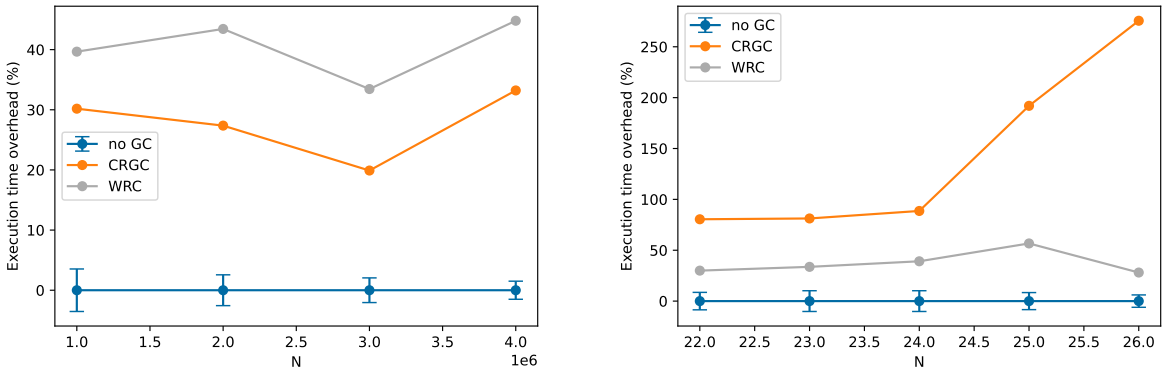
9.1.1 Microbenchmarks

Figures 9.1 and 9.2 show the impact of actor GC on two microbenchmarks, Count and Fibonacci. The Count microbenchmark measures the time to send N messages from a producer



(a) Count (b) Fibonacci

Figure 9.1: Execution times for Savina microbenchmarks.



(a) Count (b) Fibonacci

Figure 9.2: Execution time overheads for Savina microbenchmarks.

actor to a consumer actor. The Fibonacci benchmark computes the N-th Fibonacci number using a recursive actor tree computation: when an actor is asked to compute the N-th Fibonacci number, it spawns actors to compute the $N-1$ and $N-2$ Fibonacci numbers and returns the sum of the results. The Fibonacci benchmark therefore measures the rate of actor creation and deletion when actor GC is trivial.

As a baseline, we evaluated the benchmarks without garbage collection ("no GC") and with *weighted reference counting (WRC)* [32], a lightweight acyclic garbage collection scheme. Figures 9.2a and 9.2b report the overhead in execution time for WRC and CRGC, compared to the version without GC.

Results for the Count microbenchmark show that both CRGC and WRC have a nontrivial impact on the rate messages are sent and received. For WRC, we attribute this to the fact that actors need to check whether incoming messages contain any references. For CRGC, there are

a multitude of small costs. Sending a message usually only requires incrementing a counter associated with each reference, and receiving a message usually only requires incrementing an actor's receive count. However, when an actor's entry becomes full (i.e. the send or receive count is about to overflow) the actor *finalizes* its entry by sending it to the garbage collector and initializing a fresh entry. Actors also finalize their entries when their message queue becomes empty (similarly to Pony [10]) which happens frequently in Akka. Figure 9.1 shows that actor GC imposes overhead, but sending and receiving messages remains quite cheap.

Results for the FIBONACCI microbenchmark show that CRGC has a significant impact on the rate of actor creation, but the overhead is still within an order of magnitude of non-GC code. This overhead becomes vanishingly small in the QUICKSORT benchmark described in Section 9.1.2, which uses an identical tree-like parallelism structure but gives each actor nontrivial amounts of work to do. The overhead in FIBONACCI is due to the large number of entry messages that must be sent to and processed by the garbage collector.

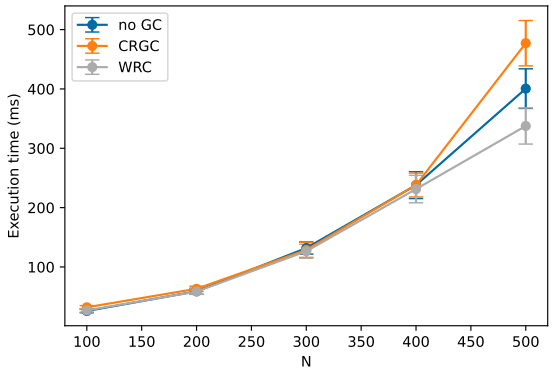
9.1.2 Benchmarks

Figures 9.3 and 9.4 show the performance overhead of CRGC on real parallel algorithms in the Savina suite. None of the benchmarks produce any actor garbage, so they measure the overhead of message-passing and reference-passing. Because these operations have low overhead in CRGC, the benchmarks all show similar performance between automatically garbage-collected and manually garbage-collected code.

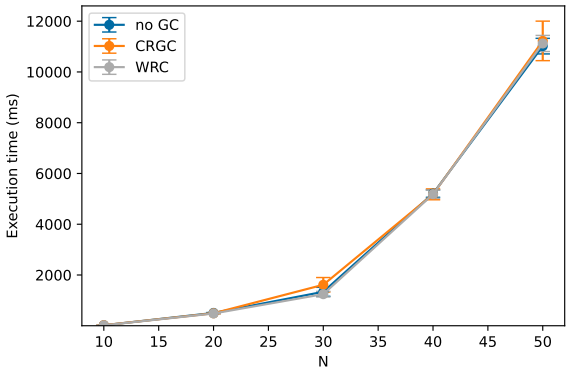
9.1.3 Message Counts

CRGC is closely related to Pony's MAC algorithm [10], but it is difficult to make a fair empirical comparison between the implementations: whereas CRGC is implemented as a library for Akka on the JVM, MAC is tightly integrated with the Pony actor runtime, which is implemented atop LLVM. We could write a MAC library for Akka and compare their execution times, but this would be unfair because the implementation needs to be finely tuned for good performance. We therefore compare the two approaches with respect to the *number of extra messages* imposed by each algorithm.

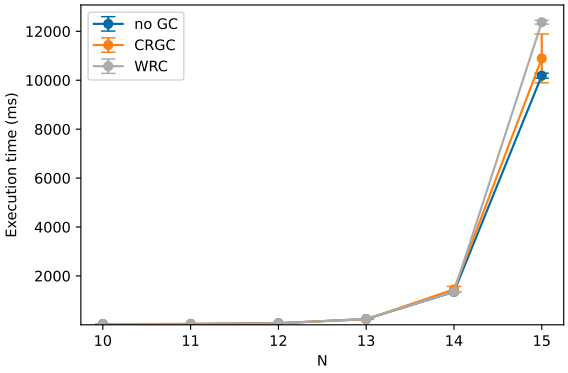
Figures 9.5 and 9.6 plot the number of control messages imposed by WRC, MAC, and CRGC in the Savina benchmark suite, relative to the number of application messages in each benchmark. Our MAC implementation does not actually detect actor garbage, but performs the same *CNF-ACK protocol* presented by Clebsch and Drossopoulou [10]. Unsurprisingly, MAC imposes more control messages than CRGC: whereas in CRGC it suffices for actors to send entries to the garbage collector, in MAC the garbage collector needs to send additional messages that confirm whether



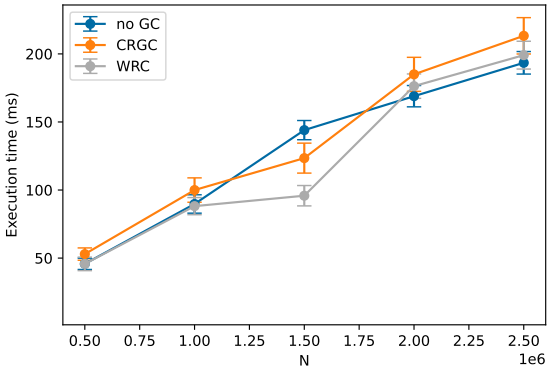
(a) All-Pairs Shortest Paths



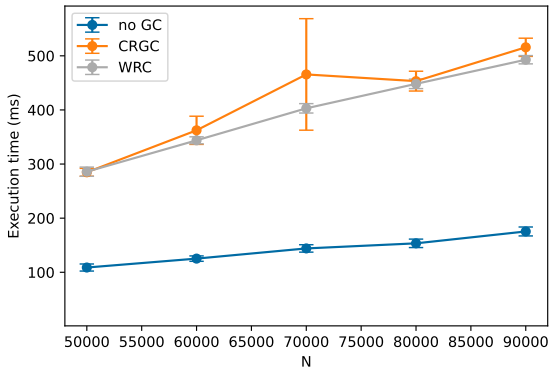
(b) A-Star



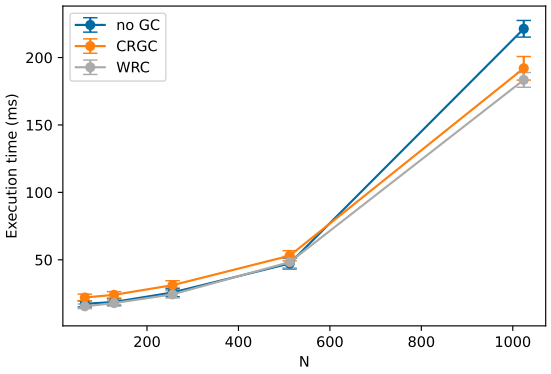
(c) N-Queens



(d) QuickSort

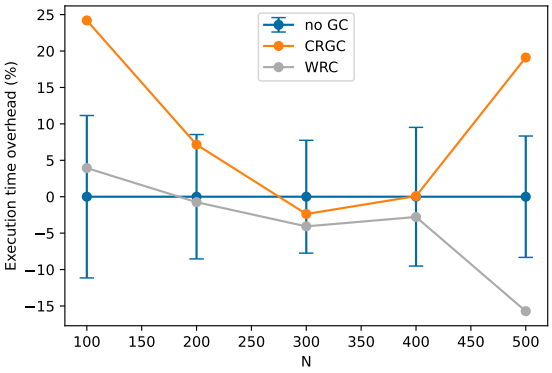


(e) RadixSort

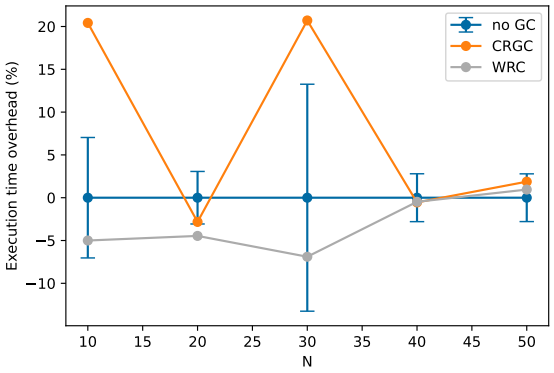


(f) RecMatMul

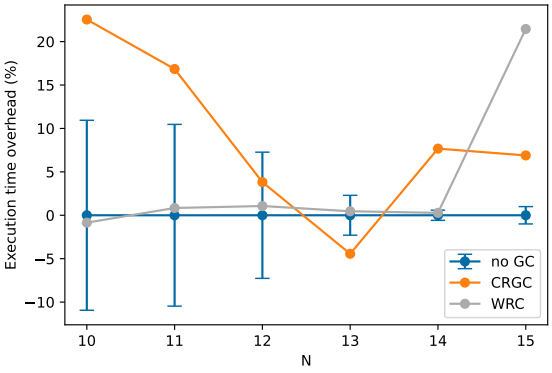
Figure 9.3: Execution times for Savina benchmarks.



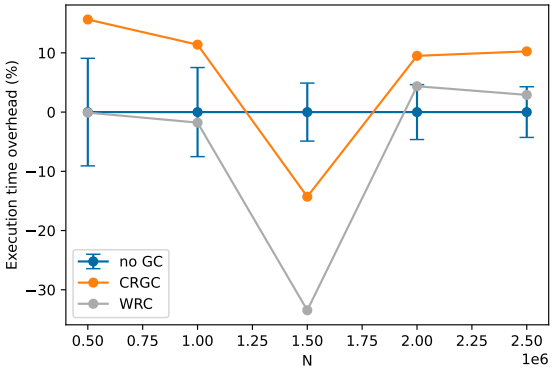
(a) All-Pairs Shortest Paths



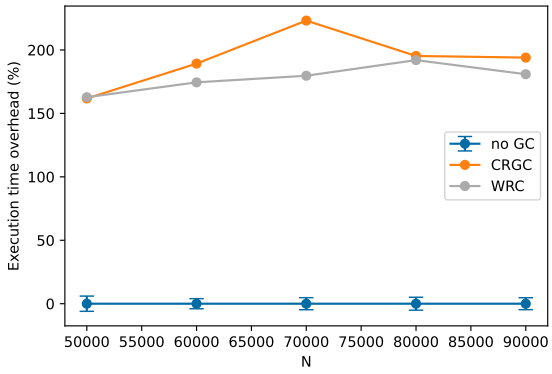
(b) A-Star



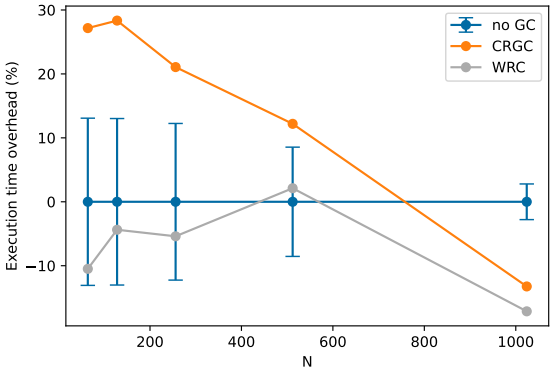
(c) N-Queens



(d) QuickSort

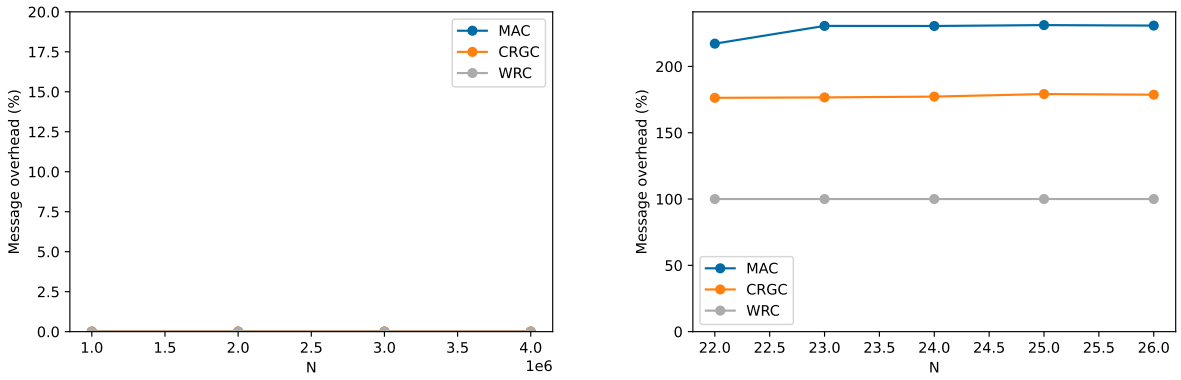


(e) RadixSort



(f) RecMatMul

Figure 9.4: Execution time overheads for Savina benchmarks.



(a) Count (b) Fibonacci

Figure 9.5: Message count overheads for Savina microbenchmarks.

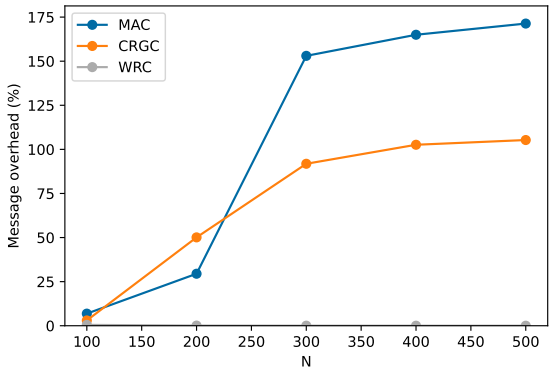
the actor is truly blocked. However, both approaches use significantly more messages than WRC, which only adds control messages when actors send or deactivate references.

9.2 RandomWorkers: A Configurable GC Benchmark

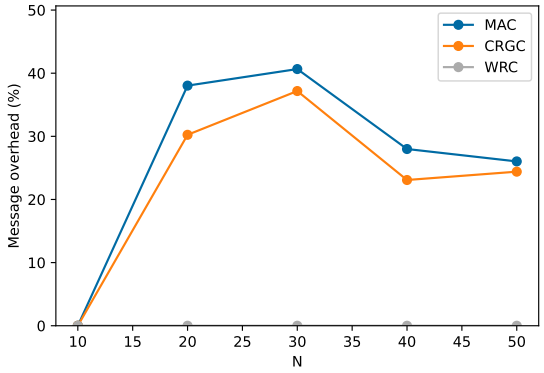
Although the benchmarks in the previous section serve to measure overhead, they are not realistic domains in which a programmer would use actor GC. All of the benchmarks are simple enough to make manual actor GC trivial, and many of the benchmarks do not generate actor garbage at all. Unfortunately, because actor GC is not widespread, there is no established corpus of algorithms that generate actor garbage in an unpredictable way. We therefore follow the methodology of Blessing et al. [60] and develop a *configurable* benchmark that can be tuned to represent many different kinds of workloads.

Blessing et al. proposed ChatApp [60], a single-machine benchmark that simulates a highly concurrent chat application. The benchmark consists of *directory actors*, which act as load balancers; *client actors*, which represent a remote client's current state; and *chat actors*, which represent the current state of a conversation between actors. By tuning parameters, such as the number of clients per directory or the likelihood of two actors starting a chat, the benchmark can illustrate multiple different realistic workloads. However, the benchmark makes it easy to predict when actors become garbage in order to accommodate actor frameworks that do not provide actor GC. This prevents the benchmark from expressing more sophisticated concurrency patterns, such as tree-based parallelism in the Quicksort benchmark.

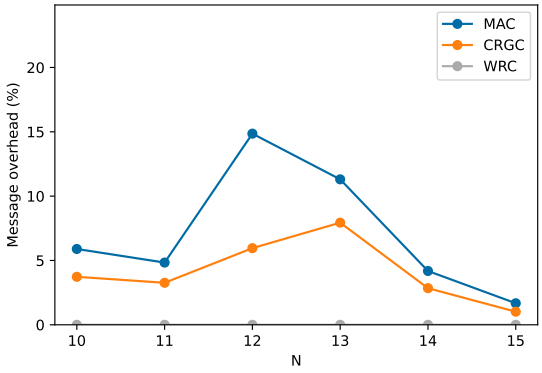
This section presents a generalization of the ChatApp benchmark, called RandomWorkers. RandomWorkers consists of N nodes with a `Manager` actor at each node. The central Manager



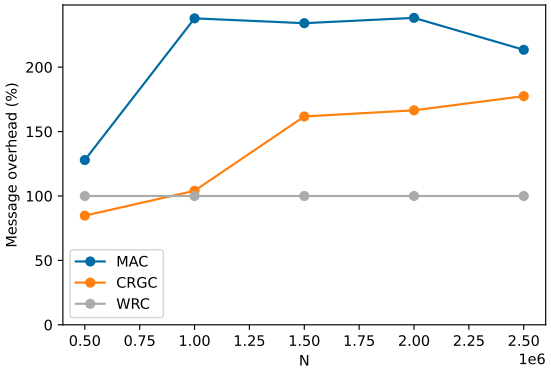
(a) All-Pairs Shortest Paths



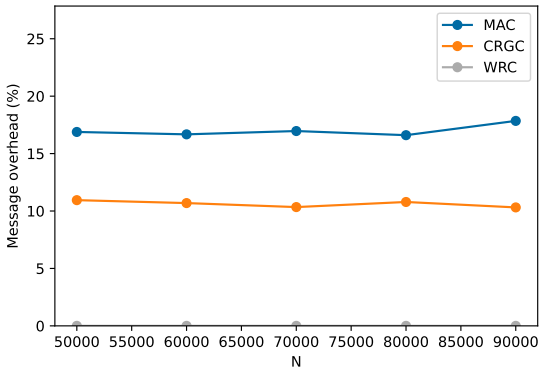
(b) A-Star



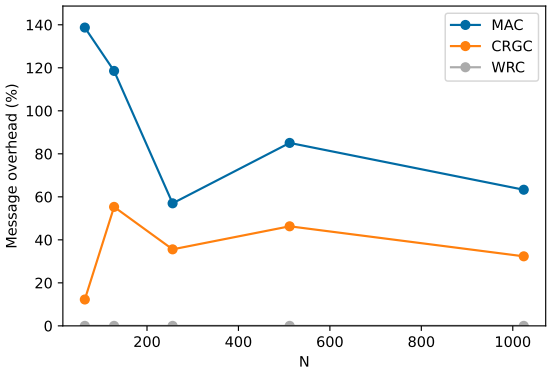
(c) N-Queens



(d) QuickSort



(e) RadixSort



(f) RecMatMul

Figure 9.6: Message count overheads for Savina benchmarks.

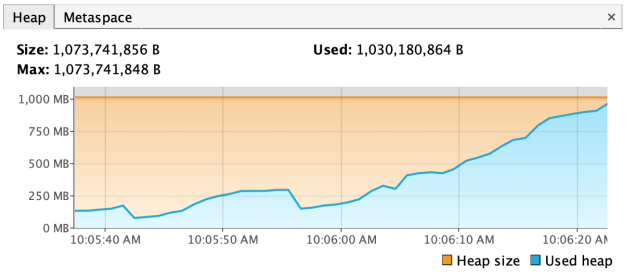


Figure 9.7: Heap usage by RandomWorkers when actor GC is not used.

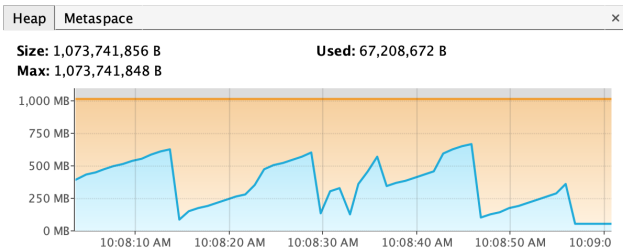


Figure 9.8: Heap usage by RandomWorkers with CRGC collection.

receives a continual stream of requests, at a configurable rate. Upon receiving a message, the `Manager` can:

1. Spawn a local `Worker` actor;
2. Send a remote `Manager` some references to `Workers`;
3. Send a `Worker` some references to other `Workers`;
4. Send a "work" message (containing a payload of random, but configurable, size) to a `Worker` actor;
5. Deactivate references to several `Worker` actors;
6. Deactivate references to *all* `Worker` actors.

The `Worker` actors, in turn, can also spawn new `Workers` and create references and deactivate references.

Different configurations of the RandomWorkers benchmark can simulate different types of applications:

1. The amount of work each `Worker` actor is given and the rate of requests to each `Manger` actor can simulate load on a system.
2. If actors have zero probability of creating references, then all garbage is acyclic garbage.
3. If actors have zero probability of creating references between remote workers and local workers, then there is no distributed garbage.
4. The likelihood for a `Manager` to deactivate references to *all* `Worker` actors affects the size of cyclic garbge.

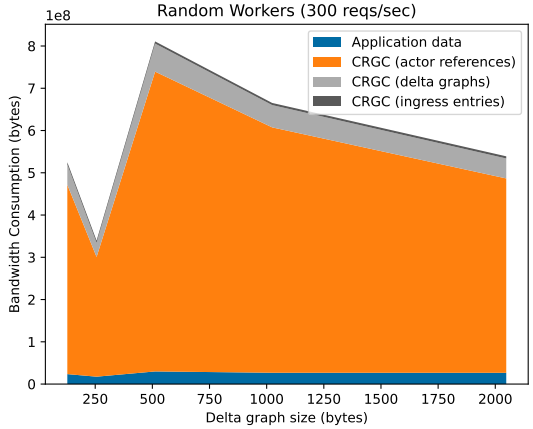
Figures 9.7 and 9.8 show how RandomWorkers introduces memory leaks into a system if actors are spawned without bound and never garbage-collected.

9.2.1 Bandwidth Usage

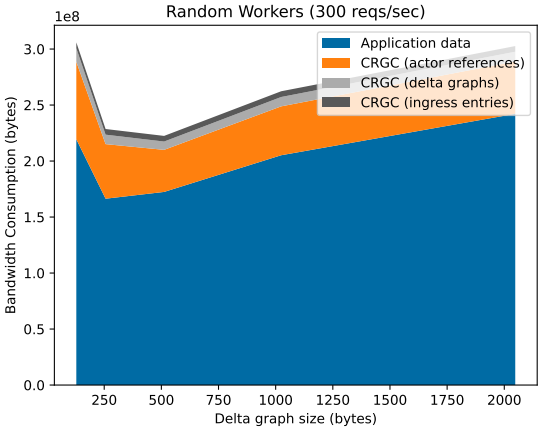
To collect distributed garbage, CRGC garbage collectors continually broadcast their ingress entries and delta graphs. Figure 9.9 shows the bandwidth used by RandomWorkers for three different configurations, broken down by source of overhead. The configurations are as follows:

- *Torture test (small messages):* Application messages contain a small payload between 0 and 50 bytes. Whenever a worker receives a message, the worker has a 30% chance of spawning another worker or creating a reference for another worker. Likewise, managers have a 50% chance of spawning a worker, a 50% chance of sending a message to a remote actor, and a 70% chance of sending a message to a local worker.
- *Torture test (medium messages):* A version of the torture test configuration where the message payload may be between 0 and 5KB.
- *Streaming:* An application that exchanges large amounts of data with a relatively static actor topology. Messages contain a payload between 0 and 5KB. Worker actors do not spawn or create references to one another. Whenever the manager actor receives a message, it has 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.

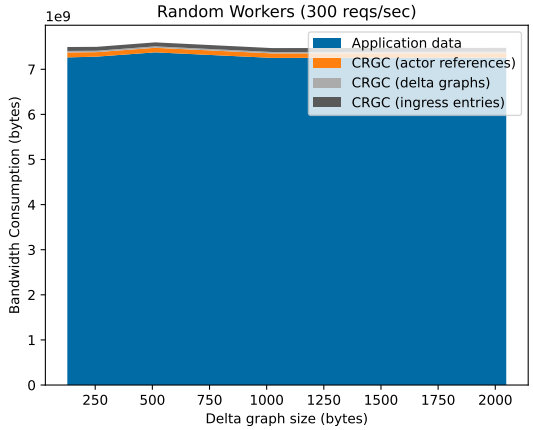
As expected, the bandwidth overhead for the torture test is very high (Figure 9.9a). However, the majority of this overhead is taken up by actor references stored in delta graphs. Akka represents actor references as fully qualified paths, each taking up dozens of bytes to serialize. In applications with many active actors per node, delta graphs will mention large numbers of actors and each actor reference will need to be serialized. This problem could be solved by giving each actor a compressed identifier for use in garbage collection, but the issue might not be pressing in practice: Figures 9.9b and 9.9c show that when application messages are larger and the number of actors is smaller, actor references (and CRGC as a whole) only impose a modest overhead.



(a) Torture test (small messages)



(b) Torture test (medium messages)



(c) Streaming

Figure 9.9: Bandwidth usage for three configurations of RandomWorkers.

10

Conclusion

We have explored the collage-based approach to actor GC. Whereas traditional garbage collectors compute a distributed snapshot that is *a priori* global and consistent, a collage-based approach takes an arbitrary collection of snapshots and identifies a consistent subset *a posteriori*. The approach is remarkably general—it generalizes to models with dynamic topologies, monitoring for failure, message loss, and crashed nodes—and it is also flexible with respect to how it is implemented.

This thesis presented two actor GCs based on the collage-based approach: PRL and CRGC. The former allows acyclic garbage actors to collect themselves (using reference listing) and allows local garbage collectors to exchange only minimal data (by summarizing their local set of snapshots). In contrast, CRGC delegates all garbage collection to the node-local garbage collector and provides mechanisms for recovering from dropped messages and crashed nodes. One opportunity for future work is to find the best of both worlds, e.g. a summarization algorithm for CRGC like that of PRL, or an option to use reference listing when message delivery is guaranteed to reduce the load on the garbage collector. Scaling CRGC to large clusters will likely require more efficient protocols for exchanging garbage collection information. It would also be interesting to develop an actor GC that combines the collage-based approach with a more traditional snapshot-based approach, since the latter would be capable of collecting "disconnected" actor garbage introduced in Section 2.2.

The most remarkable feature of the collage-based approach is how well it decouples performance from correctness. Actors can take snapshots at any time, so we can tweak the frequency of those snapshots according to the needs of the actor or the application. We are also free to give local garbage collectors multiple message queues (reducing contention), to use gossip protocols for exchanging snapshots between nodes, or to use generational garbage collection. Collage-based GC allows researchers to experiment with optimizations without worrying about introducing memory leaks or use-after-free bugs.

Most importantly, the work we developed here has shown that *fully automatic* distributed

resource management could be more than just a pipe dream. After all, why would we expect that an automatic garbage collector could be *provably* capable of recovery when information has been irretrievably lost to the network or to a corrupted hard disk? The results are all due to the simplicity of the actor model and the generality of the collage-based approach. If we can now apply these results to real distributed applications, it could bring about serious improvements in program reliability and simplicity.

REFERENCES

[1] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*. Santa Clara California: ACM, Oct. 2013, pp. 1–16.

[2] "Apache Hadoop Amazon Web Services support – S3A Committers: Architecture and Implementation," https://hadoop.apache.org/docs/r3.1.0/hadoop-aws/tools/hadoop-aws/committer_architecture.html, 2018.

[3] G. Agha, "Concurrent object-oriented programming," *Communications of the ACM*, vol. 33, no. 9, pp. 125–141, Sep. 1990.

[4] "Riak," https://riak.com/index.html, 2024.

[5] "Apache CouchDB," https://couchdb.apache.org/, 2024.

[6] "Akka Streams Documentation • Akka Documentation," https://doc.akka.io/docs/akka/2.9.2/stream/index.html, 2024.

[7] "Alpakka Documentation," https://doc.akka.io/docs/alpakka/7.0.2/index.html, 2024.

[8] "RabbitMQ," https://www.rabbitmq.com/, 2024.

[9] "Akka HTTP," https://doc.akka.io/docs/akka-http/10.6.1/index.html, 2024.

[10] S. Clebsch and S. Drossopoulou, "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*. Indianapolis, Indiana, USA: ACM Press, 2013, pp. 553–570.

[11] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, A. C. Arpaci-Dusseau and G. Voelker, Eds. USENIX Association, 2018, pp. 561–577.

[12] N. Venkatasubramanian, G. Agha, and C. Talcott, "Scalable distributed garbage collection for systems of active objects," in *Memory Management*, Y. Bekkers and J. Cohen, Eds. Berlin/Heidelberg: Springer-Verlag, 1992, vol. 637, pp. 134–147.

[13] T. Kamada, S. Matsuoka, and A. Yonezawa, "Efficient parallel global garbage collection on massively parallel computers," in *Proceedings Supercomputing '94, Washington, DC, USA, November 14-18, 1994*, G. M. Johnson, Ed. IEEE Computer Society, 1994, pp. 79–88.

[14] I. Puaut, "A distributed garbage collector for active objects," in *PARLE'94 Parallel Architectures and Languages Europe*, G. Goos, J. Hartmanis, C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, vol. 817, pp. 539–552.

[15] D. Kafura, M. Mukherji, and D. Washabaugh, "Concurrent and distributed garbage collection of active objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 4, pp. 337–350, Apr. 1995.

[16] P. Dickman, "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*, ser. Lecture Notes in Computer Science, Ö. Babaoglu and K. Marzullo, Eds., vol. 1151. Springer, 1996, pp. 141–158.

[17] A. Vardhan and G. Agha, "Using passive object garbage collection algorithms for garbage collection of active objects," *ACM SIGPLAN Notices*, vol. 38, no. 2 supplement, p. 106, Feb. 2003.

[18] W.-J. Wang and C. A. Varela, "Distributed garbage collection for mobile actor systems: The pseudo root approach," in *Advances in Grid and Pervasive Computing*, Y.-C. Chung and J. E. Moreira, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 3947, pp. 360–372.

[19] H.-J. Boehm, "Space efficient conservative garbage collection," in *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, R. Cartwright, Ed. ACM, 1993, pp. 197–206.

[20] M. Bagherzadeh, N. Fireman, A. Shawesh, and R. Khatchadourian, "Actor concurrency bugs: A comprehensive study on symptoms, root causes, API usages, and differences," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.

[21] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[22] G. Agha, *ACTORS - A Model of Concurrent Computation in Distributed Systems*, ser. MIT Press Series in Artificial Intelligence. Cambridge, MA: MIT Press, 1990.

[23] P. Haller and A. Loiko, "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*. Amsterdam Netherlands: ACM, Oct. 2016, pp. 272–291.

[24] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil, "Deny capabilities for safe, fast actors," in *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. Pittsburgh PA USA: ACM, Oct. 2015, pp. 1–12.

[25] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *Journal of Functional Programming*, vol. 7, no. 1, pp. 1–72, Jan. 1997.

[26] "Akka," https://akka.io/, 2024.

[27] P. R. Wilson, "Uniprocessor garbage collection techniques," in *Memory Management, International Workshop IWMM 92, St. Malo, France, September 17-19, 1992, Proceedings*, ser. Lecture Notes in Computer Science, Y. Bekkers and J. Cohen, Eds., vol. 637. Springer, 1992, pp. 1–42.

[28] R. E. Jones, A. L. Hosking, and J. E. B. Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, ser. Chapman and Hall / CRC Applied Algorithms and Data Structures Series. CRC Press, 2011.

[29] J. Armstrong, R. Virding, and M. Williams, *Concurrent programming in ERLANG*. Prentice Hall, 1993.

[30] A. D. Kshemkalyani, M. Raynal, and M. Singhal, "An introduction to snapshot algorithms in distributed computing," *Distributed Syst. Eng.*, vol. 2, no. 4, pp. 224–233, 1995.

[31] D. Bevan, "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. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, vol. 259, pp. 176–187.

[32] P. Watson and I. Watson, "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. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, vol. 259, pp. 432–443.

[33] A. Birrell, D. Evers, G. Nelson, S. Owicki, and G. Wobber, "Distributed garbage collection for network objects," Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, Technical Report 116, 1993.

[34] P. P. Pirinen, "Barrier techniques for incremental tracing," in *International Symposium on Memory Management, ISMM '98, Vancouver, British Columbia, Canada, 17-19 October, 1998, Conference Proceedings*, S. L. P. Jones and R. E. Jones, Eds. ACM, 1998, pp. 20–25.

[35] B. Liskov and R. Ladin, "Highly-available distributed services and fault-tolerant distributed garbage collection," in *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing, Calgary, Alberta, Canada, August 11-13, 1986*, J. Y. Halpern, Ed. ACM, 1986, pp. 29–39.

[36] L. V. Mancini and S. K. Shrivastava, "Fault-tolerant reference counting for garbage collection in distributed systems," *Comput. J.*, vol. 34, no. 6, pp. 503–513, 1991.

[37] D. Plainfossé and M. Shapiro, "A survey of distributed garbage collection techniques," in *Memory Management, International Workshop IWMM 95, Kinross, UK, September 27-29, 1995, Proceedings*, 1995, pp. 211–249.

[38] J. M. Piquer, "Indirect reference counting: A distributed garbage collection algorithm," in *Parle '91 Parallel Architectures and Languages Europe*, E. H. L. Aarts, J. van Leeuwen, and M. Rem, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, vol. 505, pp. 150–165.

[39] L. Moreau, P. Dickman, and R. E. Jones, "Birrell's distributed reference listing revisited," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, pp. 1344–1395, 2005.

[40] N. Venkatasubramanian, "Hierarchical garbage collection in scalable distributed systems," M.S. thesis, University of Illinois at Urbana-Champaign, 1992.

[41] N. Venkatasubramanian and C. L. Talcott, "Integration of resource management activities in distributed systems," Stanford University, Technical Report, May 1999.

[42] A. Vardhan, "Distributed garbage collection of active objects: A transformation and its applications to java programming," in *MS Thesis, University of Illinois*, 1998.

[43] W.-J. Wang, C. Varela, F.-H. Hsu, and C.-H. Tang, "Actor garbage collection using vertex-preserving actor-to-object graph transformations," in *Advances in Grid and Pervasive Computing*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, P. Bellavista, R.-S. Chang, H.-C. Chao, S.-F. Lin, and P. M. A. Sloot, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6104, pp. 244–255.

[44] S. E. Abdullahi and G. A. Ringwood, "Garbage collecting the Internet: A survey of distributed garbage collection," *ACM Computing Surveys*, vol. 30, no. 3, pp. 330–373, Sep. 1998.

[45] T. J. Desell and C. A. Varela, "SALSA Lite: A hash-based actor runtime for efficient local concurrency," in *Concurrent Objects and Beyond - Papers Dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday*, ser. Lecture Notes in Computer Science, G. A. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, Eds., vol. 8665. Springer, 2014, pp. 144–166.

[46] F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing*, vol. 2, no. 3, pp. 161–175, Sep. 1987.

[47] W.-J. Wang, "Conservative snapshot-based actor garbage collection for distributed mobile actor systems," *Telecommunication Systems*, June 2011.

[48] S. Alagar and S. Venkatesan, "An optimal algorithm for distributed snapshots with causal message ordering," *Inf. Process. Lett.*, vol. 50, no. 6, pp. 311–316, 1994.

[49] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," *Australian Computer Science Communications*, vol. 10, no. 1, pp. 56–66, Feb. 1988.

[50] S. Blessing, S. Clebsch, and S. Drossopoulou, "Tree topologies for causal message delivery," in *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2017*. Vancouver, BC, Canada: ACM Press, 2017, pp. 1–10.

[51] N. Venkatasubramanian and C. Talcott, "Reasoning about meta level activities in open distributed systems," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing - PODC '95*. Ottowa, Ontario, Canada: ACM Press, 1995, pp. 144–152.

[52] H. Lieberman and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Communications of the ACM*, vol. 26, no. 6, pp. 419–429, 1983.

[53] C. Hewitt and H. G. Baker, "Laws for communicating parallel processes," in *Information Processing, Proceedings of the 7th IFIP Congress 1977, Toronto, Canada, August 8-12, 1977*, B. Gilchrist, Ed. North-Holland, 1977, pp. 987–992.

[54] J. Meseguer, "Conditional rewriting logic: Deduction, models and concurrency," in *Conditional and Typed Rewriting Systems, 2nd International CTRS Workshop, Montreal, Canada, June 11-14, 1990, Proceedings*, 1990, pp. 64–91.

[55] W. D. Clinger, "Foundations of actor semantics," Massachusetts Institute of Technology / Massachusetts Institute of Technology, USA, Tech. Rep., 1981.

[56] D. Plyukhin and G. Agha, "Scalable termination detection for distributed actor systems," in *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, ser. LIPIcs, I. Konnov and L. Kovács, Eds., vol. 171. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 11:1–11:23.

[57] J. Armstrong, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, Royal Institute of Technology, Stockholm, Sweden, 2003.

[58] G. Neiger and S. Toueg, "Automatically increasing the fault-tolerance of distributed systems," in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988*, D. Dolev, Ed. ACM, 1988, pp. 248–262.

[59] S. M. Imam and V. Sarkar, "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*. Portland, Oregon, USA: ACM Press, 2014, pp. 67–80.

[60] S. Blessing, K. Fernandez-Reyes, A. M. Yang, S. Drossopoulou, and T. Wrigstad, "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*. Athens, Greece: ACM Press, 2019, pp. 41–50.

[61] H. Svensson and L.-∀. Fredlund, "A more accurate semantics for distributed erlang," in *Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop*. Freiburg Germany: ACM, Oct. 2007, pp. 43–54.

[62] H. Svensson, L.-∀. Fredlund, and C. Benac Earle, "A unified semantics for future Erlang," in *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*. Baltimore Maryland USA: ACM, Sep. 2010, pp. 23–32.

[63] "Cluster Membership Service • Akka Documentation," https://doc.akka.io/docs/akka/2.9.2/typed/cluster-membership.html, 2024.

[64] A. M. Ricciardi and K. P. Birman, "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*. Montreal, Quebec, Canada: ACM Press, 1991, pp. 341–353.

[65] "Message Delivery Reliability • Akka Documentation," https://doc.akka.io/docs/akka/2.9.2/general/message-delivery-reliability.html, 2024.

[66] C. Varela and G. Agha, "Programming dynamically reconfigurable open systems with SALSA," *ACM SIGPLAN Notices*, vol. 36, no. 12, pp. 20–34, Dec. 2001.

[67] N. Hayashibara, X. Défago, R. Yared, and T. Katayama, "The ϕ Accrual Failure Detector," in *23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil*. IEEE Computer Society, 2004, pp. 66–78.

[68] "Split Brain Resolver • Akka Documentation," https://doc.akka.io/docs/akka/2.9.2/split-brain-resolver.html, 2024.

[69] W. Shakespeare, *Romeo and Juliet*, 1597.

[70] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

[71] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872–923, May 1994.

[72] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[73] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," in *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, ser. Lecture Notes in Computer Science, X. Défago, F. Petit, and V. Villain, Eds., vol. 6976. Springer, 2011, pp. 386–400.

[74] S. T. Allen, "Pony - About the Pony cycle detector," https://github.com/ponylang/ponyc/blob/fa8550ece582881e55e664e2bc0d8a5d8df12f4b/src/libponyrt/gc/cycle.c, Nov. 2022.

A

TLA+ Specifications

A.1 The Fault Model

MODULE *FaultModel*

EXTENDS *Integers*, *FiniteSets*, *Bags*, *TLC*

The model is parameterized by sets ranging over actor names and node identifiers:

CONSTANT

ActorName,	Every actor has a unique name; this set ranges over all names.
NodeID,	Every node has a unique identifier; this set ranges over all IDs.
null	A TLA+-specific constant, used to indicate values where a partial map is undefined.

A configuration is a 4-tuple (*actors*, *location*, *msgs*, *shunned*):

VARIABLE

actors,	A partial map from actor names to actor states (i.e. behaviors).
location,	A partial map from actor names to actor locations (i.e. nodes).
msgs,	A bag (i.e. multiset) of messages to be delivered.
shunned	A relation on nodes, such that $shunned[N1, N2]$ if N2 shuns N1.

A message is modeled as a record. The *origin* field indicates the node that produced the message; *admitted* indicates whether the message was admitted into the destination node; *target* indicates the name of the destination actor; and *refs* indicates the set of actor names contained inside the message.
We do not explicitly model the payload of the message (aside from the refs) because it is not relevant to garbage collection.

$Message \triangleq [$
$origin : NodeID,$
$admitted :$ BOOLEAN ,
$target : ActorName,$

$refs$: SUBSET $ActorName$

]

INITIALIZATION AND BASIC INVARIANTS

ActorState is a record that models the state of an actor:

- *status* indicates whether the actor is busy, idle, or halted.
- *isSticky* indicates whether the actor is sticky, i.e. able to spontaneously change state from "idle" to "busy".
- *active* is a map representing the number of references this actor has to every other actor.
- *monitored* is the set of actors monitored by this actor.

$ActorState \triangleq$ [

$status$: {"busy", "idle", "halted"},

$isSticky$: BOOLEAN ,

$active$: $[ActorName \to Nat]$,

$monitored$: SUBSET $ActorName$

]

TypeOK is an invariant that specifies the type of each component in the configuration.

In TLA+ syntax, a conjunction $e_1 \land e_2 \land e_3$ can be written as follows:

$\land\ e_1$

$\land\ e_2$

$\land\ e_3$

We will use this special syntax for better readability.

$TypeOK \triangleq$

$\land\ actors \in [ActorName \to ActorState \cup \{null\}]$

$\land\ location \in [ActorName \to NodeID \cup \{null\}]$

$\land\ BagToSet(msgs) \subseteq Message$

The above invariant states that *actors* and *location* are partial maps and that *msgs* is a bag of messages.

The initial configuration consists of an actor *a* located on node N. The actor is a busy sticky actor with one reference to itself.

The expression `(a :> 1) @@ [b ∈ ActorName |-> 0]` defines a function which maps each *b* to 0 except for *a*, which is mapped to 1.

$InitialConfiguration(a, N) \triangleq$

LET $state \triangleq$ [

$status$: "busy",

$isSticky$: TRUE,

$active : (a :> 1) @@ [b \in ActorName \mapsto 0]$

]

IN

$\land\ actors = (a :> state) @@ [b \in ActorName \mapsto null]$

$\land\ location = (a :> N) @@ [b \in ActorName \mapsto null]$

$\land\ msgs = EmptyBag$

actors[b] and *location[b]* are *null* (i.e. undefined) for every actor except *a*; we set *actors[a]* equal to *state* (defined above) and *location[a]* equal to node *N*.

DEFINITIONS

Below are TLA+ mechanisms for computing the largest subset of D that satisfies F, and for selecting fresh actor names.

$$LargestSubset(D,\ F(\_)) \triangleq D \setminus \text{CHOOSE } S \in \text{SUBSET } D : F(D \setminus S)$$

$$\begin{aligned} FreshActorName \triangleq\ & \text{IF } \exists\, a \in ActorName : actors[a] = null \\ & \text{THEN } \{\text{CHOOSE } a \in ActorName : actors[a] = null\} \\ & \text{ELSE } \{\} \end{aligned}$$

pdom(S) is the domain over which the partial function S is defined.

$$pdom(S) \triangleq \{a \in \text{DOMAIN } S : S[a] \neq null\}$$

The following functions are shorthand for manipulating bags of messages:

$put(bag,\ x)$	$\triangleq\ bag \oplus SetToBag(\{x\})$	Adds x to the bag.
$remove(bag,\ x)$	$\triangleq\ bag \ominus SetToBag(\{x\})$	Removes x from the bag.
$replace(bag,\ x,\ y)$	$\triangleq\ put(remove(bag,\ x),\ y)$	Replaces x with y in the bag.

We define the following sets to range over created, busy, idle, halted, and sticky actors.

$$\begin{aligned} Actors &\triangleq pdom(actors) \\ BusyActors &\triangleq \{a \in Actors : actors[a].status = \text{“busy”}\} \\ IdleActors &\triangleq \{a \in Actors : actors[a].status = \text{“idle”}\} \\ HaltedActors &\triangleq \{a \in Actors : actors[a].status = \text{“halted”}\} \\ StickyActors &\triangleq \{a \in Actors : actors[a].isSticky\} \end{aligned}$$

A message is admissible if it is not already admitted and the origin node is not shunned by the destination node.

$AdmissibleMsgs \triangleq \{m \in BagToSet(msgs) :$

$$\begin{aligned}
&\wedge \neg m.admitted \\
&\wedge \neg shunned[m.origin,\ location[m.target]]\}
\end{aligned}$$

$$AdmittedMsgs \triangleq \{m \in BagToSet(msgs) : m.admitted\}$$

deliverableTo(a) is the set of messages to an actor *a* is the set of messages *m* for which *a* is the target, and *m* has either been admitted or can be admitted. In-flight messages from shunned nodes are excluded from this set.
An actor's acquaintances acqs(a) are the set of actors for which it has references.
An actor's inverse acquaintances iacqs(b) are the actors for which it is an acquaintance.
An actor's potential acquaintances pacqs(a) are the actors for which it has a reference or can possibly receive a reference due to an undelivered message.
An actor's potential inverse acquaintances piacqs(a) are the actors for which it is a potential acquaintance.

$$\begin{aligned}
deliverableTo(a) \triangleq \{m \in BagToSet(msgs) :\ &\wedge m.target = a \\
&\wedge (m.admitted \vee m \in AdmissibleMsgs)\}
\end{aligned}$$

$$acqs(a) \triangleq \{b \in ActorName : actors[a].active[b] > 0\}$$
$$iacqs(b) \triangleq \{a \in Actors : b \in acqs(a)\}$$
$$pacqs(a) \triangleq \{b \in ActorName : b \in acqs(a) \vee \exists\, m \in deliverableTo(a) : b \in m.refs\}$$
$$piacqs(b) \triangleq \{a \in Actors : b \in pacqs(a)\}$$

$$admittedMsgsTo(a) \triangleq \{m \in deliverableTo(a) : m.admitted\}$$
$$monitoredBy(b) \triangleq actors[b].monitored$$

An actor is blocked if it is idle and has no deliverable messages. Otherwise, the actor is unblocked.

$$Blocked \triangleq \{a \in IdleActors : deliverableTo(a) = \{\}\}$$
$$Unblocked \triangleq Actors \setminus Blocked$$

The exiled nodes are the largest nontrivial faction where every non-exiled node has shunned every exiled node. Likewise, a faction of nodes G is apparently exiled if every node outside of G has taken an ingress snapshot in which every node of G is shunned.

$$\begin{aligned}
&ExiledNodes \triangleq \\
&\quad LargestSubset(NodeID,\ \textsc{lambda}\ G : \\
&\qquad \wedge G \neq NodeID \\
&\qquad \wedge \forall N1 \in G,\ N2 \in NodeID \setminus G : shunned[N1,\ N2] \\
&\quad)
\end{aligned}$$

$$NonExiledNodes \triangleq NodeID \setminus ExiledNodes$$
$$ExiledActors \triangleq \{a \in Actors : location[a] \in ExiledNodes\}$$
$$FailedActors \triangleq HaltedActors \cup ExiledActors$$
$$HealthyActors \triangleq Actors \setminus FailedActors$$

$$ShunnedBy(N2) \triangleq \{N1 \in NodeID : shunned[N1,\ N2]\}$$

$ShunnableBy(N1) \triangleq (NodeID \setminus \{N1\}) \setminus ShunnedBy(N1)$
$NeitherShuns(N1) \triangleq \{N2 \in NodeID : \neg shunned[N1, N2] \wedge \neg shunned[N2, N1]\}$

TRANSITIONS

This section of the model declares the events that may occur in an execution and how each event updates the configuration.
The events of an execution are defined as "actions" in TLA+:
"An action represents a relation between old states and new states, where the unprimed variables refer to the old state and the primed variables refer to the new state. Thus, y = x' + 1 is the relation asserting that the value of y in the old state is one greater than the value of x in the new state. An atomic operation of a concurrent program will be represented in TLA by an action." [Lamport 1994]
We first define each event (*Idle*, Spawn, Send, ...) individually, and then define the Next relation which specifies all the possible transitions a configuration can take.

$Idle(a) \triangleq$

A busy actor *a* can become idle by changing its status.
The notation below states that, as a result of the Idle event, the *actors* component of the configuration will change and the remaining components do not change.
Specifically, the new value *actors* is identical to the old value, except that the status of actor *a* is set to "idle".

$\wedge\ actors' = [actors \text{ EXCEPT } ![a].status = \text{“idle”}]$
$\wedge\ \text{UNCHANGED } \langle msgs, location, shunned \rangle$

$Spawn(a, b, N) \triangleq$

A busy actor *a* can spawn a fresh actor *b* onto a non-shunned node.

$\wedge\ actors' = [actors \text{ EXCEPT}$
$\quad ![a].active[b] = 1,$ The parent obtains a reference to the child.
$\quad ![b] = [$
$\qquad status : \text{“busy”},$ The child is busy,
$\qquad isSticky : \text{FALSE},$ not sticky,
$\qquad active : (b :> 1) @@ [c \in ActorName \mapsto null],$ has a reference to itself,
$\qquad monitored : \{\}$ and monitors nobody.
$\quad]]$
$\wedge\ location' = [location \text{ EXCEPT } ![b] = N]$
$\wedge\ \text{UNCHANGED } \langle msgs, shunned \rangle$

$Deactivate(a, b) \triangleq$

A busy actor can remove references from its state.

$\wedge\ actors' = [actors \text{ EXCEPT } ![a].active[b] = 0]$

$\land$ UNCHANGED $\langle location,\ msgs,\ shunned \rangle$

$Send(a,\ b,\ m) \triangleq$

A busy actor can send messages to its acquaintances.

$\land\ msgs' = put(msgs,\ m)$ Add message m to the msgs bag.

$\land$ UNCHANGED $\langle actors,\ location,\ shunned \rangle$

$Receive(a,\ m) \triangleq$

An idle actor can receive a message, becoming busy.

$\land\ actors' = [actors$ EXCEPT $![a].status =$ "busy"$]$

$\land\ msgs' = remove(msgs,\ m)$ Remove m from the msgs bag.

$\land$ UNCHANGED $\langle location,\ shunned \rangle$

$Halt(a) \triangleq$

Busy actors can halt.

$\land\ actors' = [actors$ EXCEPT $![a].status =$ "halted"$]$

$\land$ UNCHANGED $\langle location,\ msgs,\ shunned \rangle$

$Monitor(a,\ b) \triangleq$

Busy actors can monitor their acquaintances.

$\land\ actors' = [actors$ EXCEPT $![a].monitored = @ \cup \{b\}]$ Add b to the monitored set.

$\land$ UNCHANGED $\langle location,\ msgs,\ shunned \rangle$

$Notify(a,\ b) \triangleq$

Monitoring actors can become "busy" after the actors they monitor fail.

$\land\ actors' = [actors$ EXCEPT $![a].status =$ "busy", $![a].monitored = @ \setminus \{b\}]$

$\land$ UNCHANGED $\langle location,\ msgs,\ shunned \rangle$

$Unmonitor(a,\ b) \triangleq$

Busy actors can stop monitoring actors.

$\land\ actors' = [actors$ EXCEPT $![a].monitored = @ \setminus \{b\}]$

$\land$ UNCHANGED $\langle location,\ msgs,\ shunned \rangle$

$Register(a) \triangleq$

Actors can register as sticky to spontaneously be awoken from "idle" state.

$\land\ actors' = [actors$ EXCEPT $![a].isSticky =$ TRUE$]$

$\land$ UNCHANGED $\langle location,\ msgs,\ shunned \rangle$

$Wakeup(a) \triangleq$

A sticky actor can be awoken.

$\land\ actors' = [actors \text{ EXCEPT } ![a].status = \text{“busy”}]$

$\land$ UNCHANGED $\langle location,\ msgs,\ shunned \rangle$

$Unregister(a) \triangleq$

Actors can unregister as sticky.

$\land\ actors' = [actors \text{ EXCEPT } ![a].isSticky = \text{FALSE}]$

$\land$ UNCHANGED $\langle location,\ msgs,\ shunned \rangle$

$Admit(m) \triangleq$

In-flight messages can be admitted. If node N1 shuns node N2, then messages from N1 can no longer be delivered to N2 unless they are already admitted.

$\land\ msgs' = replace(msgs,\ m,\ [m \text{ EXCEPT } !.admitted = \text{TRUE}])$

$\land$ UNCHANGED $\langle actors,\ location,\ shunned \rangle$

$Drop(m) \triangleq$

Any message can spontaneously be dropped.

$\land\ msgs' = remove(msgs,\ m)$

$\land$ UNCHANGED $\langle actors,\ location,\ shunned \rangle$

$Shun(N1,\ N2) \triangleq$

A non-exiled node can shun another node.

$\land\ shunned' = [shunned \text{ EXCEPT } ![N1,\ N2] = \text{TRUE}]$

$\land$ UNCHANGED $\langle actors,\ msgs,\ location \rangle$

The following Exile event can safely be used in place of the Shun event to simplify reasoning and reduce the model checking state space. This is safe because, for any execution in which a group of nodes G1 all shuns another group G2, there is an equivalent execution in which all *Shun* events happen successively.

$Exile(G1,\ G2) \triangleq$

$\land\ shunned' = [N1 \in G1,\ N2 \in G2 \mapsto \text{TRUE}]\ @@\ shunned$

$\land$ UNCHANGED $\langle actors,\ msgs,\ location \rangle$

Init defines the initial configuration, choosing an arbitrary name and location for the initial actor.

$Init \triangleq InitialConfiguration($

CHOOSE $a \in ActorName$: TRUE, Choose an arbitrary name for the initial actor.

CHOOSE $N \in NodeID$: TRUE Choose an arbitrary location for the actor.

)

Next defines the transition relation on configurations, defined as a TLA action, such that configuration K1 can atomically transition to configuration K2 if the relation (K1)Next(K2) holds.
For example, let K1 be a configuration with two busy actors *a,b* and an idle actor *c*. Then K1 can transition to a configuration K2 in which *a* is busy and *b,c* are idle, because of the Idle transition below.

$$
\begin{aligned}
&Next \triangleq \\
&\quad \lor \exists\, a \in BusyActors \setminus ExiledActors : Idle(a) \\
&\quad \lor \exists\, a \in BusyActors \setminus ExiledActors : \exists\, b \in FreshActorName : \\
&\qquad \exists\, N \in NeitherShuns(location[a]) : Spawn(a,\ b,\ N) \\
&\quad \lor \exists\, a \in BusyActors \setminus ExiledActors : \exists\, b \in acqs(a) : Deactivate(a,\ b) \\
&\quad \lor \exists\, a \in BusyActors \setminus ExiledActors : \exists\, b \in acqs(a) : \exists\, refs \in \textsc{subset}\ acqs(a) : \\
&\qquad Send(a,\ b,\ [origin \mapsto location[a], \\
&\qquad\qquad admitted \mapsto location[b] = location[a], \\
&\qquad\qquad target \mapsto b, \\
&\qquad\qquad refs \mapsto refs]) \\
&\quad \lor \exists\, a \in IdleActors \setminus ExiledActors : \exists\, m \in admittedMsgsTo(a) : Receive(a,\ m) \\
&\quad \lor \exists\, a \in BusyActors \setminus ExiledActors : Halt(a) \\
&\quad \lor \exists\, a \in BusyActors \setminus ExiledActors : \exists\, b \in acqs(a) : Monitor(a,\ b) \\
&\quad \lor \exists\, a \in IdleActors \setminus ExiledActors : \exists\, b \in FailedActors \cap monitoredBy(a) : \\
&\qquad Notify(a,\ b) \\
&\quad \lor \exists\, a \in BusyActors \setminus ExiledActors : \exists\, b \in monitoredBy(a) : Unmonitor(a,\ b) \\
&\quad \lor \exists\, a \in (BusyActors \setminus StickyActors) \setminus ExiledActors : Register(a) \\
&\quad \lor \exists\, a \in (IdleActors \cap StickyActors) \setminus ExiledActors : Wakeup(a) \\
&\quad \lor \exists\, a \in (BusyActors \cap StickyActors) \setminus ExiledActors : Unregister(a) \\
&\quad \lor \exists\, m \in AdmissibleMsgs : location[m.target] \notin ExiledNodes \land Admit(m) \\
&\quad \lor \exists\, m \in AdmissibleMsgs \cup AdmittedMsgs : location[m.target] \notin ExiledNodes \land Drop(m) \\
&\quad \lor \exists\, N2 \in NonExiledNodes : \exists\, N1 \in ShunnableBy(N2) : Shun(N1,\ N2)
\end{aligned}
$$

GARBAGE

An actor is potentially unblocked if it is busy or can become busy. (Halted and exiled actors automatically fail this definition.) Similarly, an actor is potentially unblocked up-to-a-fault if it is busy or it can become busy in a non-faulty extension of this execution.

$$
\begin{aligned}
&isPotentiallyUnblockedUpToAFault(S) \triangleq \\
&\quad \land StickyActors \setminus FailedActors \subseteq S \\
&\quad \land Unblocked \setminus FailedActors \subseteq S
\end{aligned}
$$

$\land \forall\, a \in S,\, b \in HealthyActors :$
$\quad a \in piacqs(b) \implies b \in S$
$\land \forall\, a \in S \cup FailedActors,\, b \in HealthyActors :$
$\quad a \in monitoredBy(b) \implies b \in S$

An actor is potentially unblocked if it is potentially unblocked up-to-a-fault or it monitors any remote actor. This is because remote actors can always become exiled, causing the monitoring actor to be notified.

$isPotentiallyUnblocked(S) \triangleq$
$\quad \land\, isPotentiallyUnblockedUpToAFault(S)$
$\quad \land \forall\, a \in Actors,\, b \in HealthyActors :$
$\quad\quad \land\, (a \in monitoredBy(b) \land location[a] \neq location[b] \implies b \in S)$

An actor is quiescent if it is not potentially unblocked. Likewise for quiescence up-to-a-fault.

$PotentiallyUnblockedUpToAFault \triangleq$
$\quad \text{CHOOSE } S \in \text{SUBSET } HealthyActors : isPotentiallyUnblockedUpToAFault(S)$
$QuiescentUpToAFault \triangleq Actors \setminus PotentiallyUnblockedUpToAFault$

$PotentiallyUnblocked \triangleq$
$\quad \text{CHOOSE } S \in \text{SUBSET } HealthyActors : isPotentiallyUnblocked(S)$
$Quiescent \triangleq Actors \setminus PotentiallyUnblocked$

Both definitions characterize a subset of the idle actors. The difference between the definitions is that quiescence up-to-a-fault is only a stable property in non-faulty executions.

$QuiescentImpliesIdle \triangleq Quiescent \subseteq (IdleActors \cup FailedActors)$
$QuiescentUpToAFaultImpliesIdle \triangleq QuiescentUpToAFault \subseteq (IdleActors \cup FailedActors)$

A.2 Common Definitions

MODULE *Common*

EXTENDS *Integers*, *FiniteSets*, *Bags*, *TLC*

This module defines variables and functions used in all following models.

CONSTANT

ActorName	The names of participating actors.

VARIABLE

actors,	actors[a] is the state of actor *a*.

$msgs$,	msgs is a bag of all *undelivered* messages.
$snapshots$	snapshots[a] is a snapshot of some actor's state.

null is an arbitrary value used to signal that an expression was undefined.

CONSTANT $null$

Assuming map1 has type [D1 -> Nat] and map2 has type [D2 -> Nat] where D2 is a subset of D1, this operator increments every map1[a] by the value of map2[a].

$$map1 \mathbin{++} map2 \triangleq [a \in \text{DOMAIN } map1 \mapsto \text{IF } a \in \text{DOMAIN } map2 \text{ THEN } map1[a] + map2[a] \text{ ELSE } map1[a]]$$

$$map1 \mathbin{--} map2 \triangleq [a \in \text{DOMAIN } map1 \mapsto \text{IF } a \in \text{DOMAIN } map2 \text{ THEN } map1[a] - map2[a] \text{ ELSE } map1[a]]$$

Notation for manipulating bags, i.e. multisets. TLA+ represents bags as functions from a set of elements to a count of how many elements are in the bag.

$put(bag,\ x) \triangleq bag \oplus SetToBag(\{x\})$ Adds x to the bag.

$remove(bag,\ x) \triangleq bag \ominus SetToBag(\{x\})$ Removes x from the bag.

$replace(bag,\ x,\ y) \triangleq put(remove(bag,\ x),\ y)$ Replaces x with y in the bag.

RECURSIVE $removeAll(\_,\ \_)$ Removes all of S from the bag.

$removeAll(bag,\ S) \triangleq$

IF $S = \{\}$ THEN bag ELSE

LET $x \triangleq$ CHOOSE $x \in S$: TRUEIN

$removeAll(remove(bag,\ x),\ S \setminus \{x\})$

$removeWhere(bag,\ F(\_)) \triangleq$ Removes all elements satisfying F.

LET $S \triangleq \{x \in \text{DOMAIN } bag : F(x)\}$IN

$[x \in \text{DOMAIN } bag \setminus S \mapsto bag[x]]$

$selectWhere(bag,\ F(\_)) \triangleq$ Finds all elements satisfying F.

LET $S \triangleq \{x \in \text{DOMAIN } bag : F(x)\}$IN

$[x \in S \mapsto bag[x]]$

$BagUnionOfSets(bag) \triangleq$

Assuming *bag* is a bag of sets, this will produce a bag with an instance of *x* for each set in *bag* that contains *x*.

LET $Count(x) \triangleq BagCardinality(selectWhere(bag,\ \text{LAMBDA } s : x \in s))$IN

$[x \in \text{UNION DOMAIN } bag \mapsto Count(x)]$

Computes the sum $\sum_{x \in dom(f)} f(x)$.

RECURSIVE $sumOver(\_, \_)$

$sumOver(dom, map) \triangleq$ IF $dom = \{\}$ THEN 0 ELSE
  LET $x \triangleq$ CHOOSE $x \in dom$: TRUEIN
  $map[x] + sumOver(dom \setminus \{x\}, map)$

$sum(map) \triangleq sumOver(\text{DOMAIN } map, map)$

The domain over which the partial function S is defined.

$pdom(S) \triangleq \{a \in \text{DOMAIN } S : S[a] \neq null\}$

TLA+ mechanism for computing the largest subset of D that satisfies F.

$LargestSubset(D, F(\_)) \triangleq D \setminus \text{CHOOSE } S \in \text{SUBSET } D : F(D \setminus S)$

TLA+ mechanism for deterministically picking a fresh actor name. If ActorName is a finite set and all names have been exhausted, this operator produces the empty set.

$FreshActorName \triangleq$ IF $\exists\, a \in ActorName : actors[a] = null$
  THEN $\{$CHOOSE $a \in ActorName : actors[a] = null\}$
  ELSE $\{\}$

$Actors \triangleq pdom(actors)$

$Snapshots \triangleq pdom(snapshots)$

A.3 The Static Model

MODULE *Static*

EXTENDS *Common*, *Integers*, *FiniteSets*, *Bags*, *TLC*

ActorState represents the GC-relevant state of an actor. - status indicates whether the actor is currently processing a message. - received is the number of messages that this actor has received. - sent[b] is the number of messages this actor has sent to b. - acqs is the set of actors that this actor is acquainted with.

$ActorState \triangleq$ [
  $status$: {"busy", "idle"},
  $received$: Nat,
  $sent$: $[ActorName \to Nat]$,
  $acqs$: SUBSET $ActorName$

]

In this simple model, a message has only one field *target* representing the name of the destination actor. The payload of the message is omitted.

$$Message \triangleq [target : ActorName]$$

actors is a partial mapping from actor names to actor states.
snapshots is also a partial mapping from actor names to actor states.
msgs is a bag of messages.

$$\begin{aligned}
&TypeOK \triangleq \\
&\quad \land\ actors \quad \in [ActorName \to ActorState \cup \{null\}] \\
&\quad \land\ snapshots \quad \in [ActorName \to ActorState \cup \{null\}] \\
&\quad \land\ BagToSet(msgs) \subseteq Message
\end{aligned}$$

$$\begin{aligned}
&InitialActorState \triangleq [\\
&\quad status \quad \mapsto \text{“busy”}, \\
&\quad sent \quad \mapsto [b \in ActorName \mapsto 0], \\
&\quad received \quad \mapsto 0, \\
&\quad acqs \quad \mapsto \{\} \\
&]
\end{aligned}$$

$$\begin{aligned}
&InitialConfiguration(actor,\ actorState) \triangleq \\
&\quad \textsc{let}\ state \triangleq [actorState\ \textsc{except} \\
&\qquad\qquad !.acqs = \{actor\} \\
&\qquad\qquad] \\
&\quad \textsc{in} \\
&\quad \land\ msgs = EmptyBag \\
&\quad \land\ actors = [a \in ActorName \mapsto \textsc{if}\ a = actor\ \textsc{then}\ state\ \textsc{else}\ null] \\
&\quad \land\ snapshots = [a \in ActorName \mapsto null]
\end{aligned}$$

DEFINITIONS

$$\begin{aligned}
msgsTo(a) &\triangleq \{m \in BagToSet(msgs) : m.target = a\} \\
acqs(a) &\triangleq actors[a].acqs \\
iacqs(b) &\triangleq \{a \in Actors : b \in acqs(a)\}
\end{aligned}$$

$$\begin{aligned}
BusyActors &\triangleq \{a \in Actors : actors[a].status = \text{“busy”}\} \\
IdleActors &\triangleq \{a \in Actors : actors[a].status = \text{“idle”}\}
\end{aligned}$$

$Blocked \triangleq \{a \in IdleActors : msgsTo(a) = \{\}\}$
$Unblocked \triangleq Actors \setminus Blocked$

TRANSITIONS

$Idle(a) \triangleq$
$\land\ actors' = [actors \text{ EXCEPT } ![a].status = \text{“idle”}]$
$\land\ \text{UNCHANGED } \langle msgs, snapshots \rangle$

$Send(a, b, m) \triangleq$
$\land\ actors' = [actors \text{ EXCEPT } ![a].sent[b] = @ + 1]$
$\land\ msgs' = put(msgs, m)$
$\land\ \text{UNCHANGED } \langle snapshots \rangle$

$Receive(a, m) \triangleq$
$\land\ actors' = [actors \text{ EXCEPT } ![a].received = @ + 1, ![a].status = \text{“busy”}]$
$\land\ msgs' = remove(msgs, m)$
$\land\ \text{UNCHANGED } \langle snapshots \rangle$

$Snapshot(a) \triangleq$
$\land\ snapshots[a] = null$
$\land\ snapshots' = [snapshots \text{ EXCEPT } ![a] = actors[a]]$
$\land\ \text{UNCHANGED } \langle msgs, actors \rangle$

$Init \triangleq$
$InitialConfiguration(\text{CHOOSE } a \in ActorName : \text{TRUE}, InitialActorState)$

$Next \triangleq$
$\lor \exists\, a \in BusyActors : Idle(a)$
$\lor \exists\, a \in BusyActors : \exists\, b \in acqs(a) : Send(a, b, [target \mapsto b])$
$\lor \exists\, a \in IdleActors : \exists\, m \in msgsTo(a) : Receive(a, m)$
$\lor \exists\, a \in IdleActors \cup BusyActors : Snapshot(a)$

$PotentiallyUnblocked \triangleq$
$\text{CHOOSE } S \in \text{SUBSET } Actors : \forall\, a, b \in Actors :$
$\land\ (a \notin Blocked \implies a \in S)$
$\land\ (a \in S \land a \in iacqs(b) \implies b \in S)$

$Quiescent \triangleq Actors \setminus PotentiallyUnblocked$

$sent(b) \triangleq sum([a \in Snapshots \mapsto snapshots[a].sent[b]])$
$received(b) \triangleq \text{IF } b \in Snapshots \text{ THEN } snapshots[b].received \text{ ELSE } 0$

$AppearsIdle \triangleq \{a \in Snapshots : snapshots[a].status = \text{“idle”}\}$
$AppearsClosed \triangleq \{b \in Snapshots : iacqs(b) \subseteq Snapshots\}$
$AppearsBlocked \triangleq \{b \in AppearsIdle \cap AppearsClosed : sent(b) = received(b)\}$
$AppearsUnblocked \triangleq Snapshots \setminus AppearsBlocked$

$AppearsPotentiallyUnblocked \triangleq$
$\quad \text{CHOOSE } S \in \text{SUBSET } Snapshots : \forall\, a,\, b \in Snapshots :$
$\quad \land\, (a \notin AppearsBlocked \implies a \in S)$
$\quad \land\, (a \in S \land a \in iacqs(b) \implies b \in S)$

$AppearsQuiescent \triangleq Snapshots \setminus AppearsPotentiallyUnblocked$

A set of snapshots is insufficient for b if:

1.b's snapshot is out of date; or

2.b is reachable by an actor for which the snapshots are insufficient.

$SnapshotsInsufficient \triangleq$
$\quad \text{CHOOSE } S \in \text{SUBSET } Actors : \forall\, a \in Actors :$
$\quad \land\, actors[a] \neq snapshots[a] \implies a \in S$
$\quad \land\, \forall\, b \in Actors :$
$\quad\quad \land\, (a \in S \land a \in iacqs(b) \implies b \in S)$

$SnapshotsSufficient \triangleq Actors \setminus SnapshotsInsufficient$

The specification captures the following properties:
Soundness: Every actor that appears quiescent is indeed quiescent.
Completeness: Every quiescent actor with a sufficient set of snapshots will appear quiescent.

$Spec \triangleq (Quiescent \cap SnapshotsSufficient) = AppearsQuiescent$

A.4 The Dynamic Model

MODULE *Dynamic*

EXTENDS *Common*, *Integers*, *FiniteSets*, *Bags*, *TLC*

ActorState represents the GC-relevant state of an actor. - status indicates whether the actor is currently processing a message. - received is the number of messages that this actor has received. - sent[b] is the number of messages this actor has sent to b. - active[b] is the number of active references to b in the state. - deactivated[b] is the number of references to b that have been deactivated. - created[b,c] is the number of references to c that have been sent to b.

$$
\begin{array}{l}
ActorState \triangleq [\\
\quad status \quad : \{\text{“busy”}, \text{“idle”}\}, \\
\quad received \quad : Nat, \\
\quad sent \quad : [ActorName \to Nat], \\
\quad active \quad : [ActorName \to Nat], \\
\quad deactivated : [ActorName \to Nat], \\
\quad created \quad : [ActorName \times ActorName \to Nat] \\
]
\end{array}
$$

A message consists of (a) the name of the destination actor, and (b) a set of references to other actors. Any other data a message could contain is irrelevant for our purposes.

$$Message \triangleq [target : ActorName,\ refs : \textsc{subset}\ ActorName]$$

$$
\begin{array}{l}
TypeOK \triangleq \\
\quad \land\ actors \quad \in [ActorName \to ActorState \cup \{null\}] \\
\quad \land\ snapshots \quad \in [ActorName \to ActorState \cup \{null\}] \\
\quad \land\ BagToSet(msgs) \subseteq Message
\end{array}
$$

$$
\begin{array}{l}
InitialActorState \triangleq [\\
\quad status \quad \mapsto \text{“busy”}, \\
\quad sent \quad \mapsto [b \in ActorName \mapsto 0], \\
\quad received \quad \mapsto 0, \\
\quad active \quad \mapsto [b \in ActorName \mapsto 0], \\
\quad deactivated \mapsto [b \in ActorName \mapsto 0], \\
\quad created \quad \mapsto [b,\ c \in ActorName \mapsto 0] \\
]
\end{array}
$$

In the initial configuration, there is one busy actor with a reference to itself.

$$
\begin{array}{l}
InitialConfiguration(actor,\ actorState) \triangleq \\
\quad \textsc{let}\ state \triangleq [actorState\ \textsc{except} \\
\qquad\qquad !.active \ = @ \mathbin{++} (actor :> 1), \\
\qquad\qquad !.created = @ \mathbin{++} (\langle actor,\ actor\rangle :> 1) \\
\qquad\qquad]
\end{array}
$$

IN

$\land\ msgs = EmptyBag$

$\land\ actors = [a \in ActorName \mapsto \text{IF } a = actor \text{ THEN } state \text{ ELSE } null]$

$\land\ snapshots = [a \in ActorName \mapsto null]$

DEFINITIONS

$msgsTo(a) \triangleq \{m \in BagToSet(msgs) : m.target = a\}$

$acqs(a) \triangleq \{b \in ActorName : actors[a].active[b] > 0\}$

$iacqs(b) \triangleq \{a \in Actors : b \in acqs(a)\}$

$pacqs(a) \triangleq \{b \in ActorName : b \in acqs(a) \lor \exists\, m \in msgsTo(a) : b \in m.refs\}$

$piacqs(b) \triangleq \{a \in Actors : b \in pacqs(a)\}$

$pastAcqs(a) \triangleq \{b \in ActorName : actors[a].deactivated[b] > 0\}$

$pastIAcqs(b) \triangleq \{a \in Actors : b \in pastAcqs(a)\}$

$BusyActors \triangleq \{a \in Actors : actors[a].status = \text{“busy”}\}$

$IdleActors \triangleq \{a \in Actors : actors[a].status = \text{“idle”}\}$

$Blocked \triangleq \{a \in IdleActors : msgsTo(a) = \{\}\}$

$Unblocked \triangleq Actors \setminus Blocked$

TRANSITIONS

$Idle(a) \triangleq$

$\land\ actors' = [actors \text{ EXCEPT } ![a].status = \text{“idle”}]$

$\land$ UNCHANGED $\langle msgs, snapshots\rangle$

$Spawn(a, b, actorState) \triangleq$

$\land\ actors' = [actors \text{ EXCEPT}$

$![a].active[b] = 1,$ Parent has a reference to the child.

$![b] = [$

$actorState \text{ EXCEPT}$

$!.active = @ \text{ ++ } (b :> 1),$ Child has a reference to itself.

$!.created = @ \text{ ++ } (\langle b, b\rangle :> 1 \text{ @@ } \langle a, b\rangle :> 1)$ Child knows about both references.

$]$

$]$

$\land$ UNCHANGED $\langle snapshots, msgs\rangle$

$Deactivate(a,\ b) \triangleq$

$\land\ actors' = [actors$ EXCEPT

$![a].deactivated[b] = @ + actors[a].active[b],$

$![a].active[b] = 0$

$]$

$\land$ UNCHANGED $\langle msgs,\ snapshots\rangle$

$Send(a,\ b,\ m) \triangleq$

$\land\ actors' = [actors$ EXCEPT

$![a].sent[b] = @ + 1,$

$![a].created = @ ++ [\langle x,\ y\rangle \in \{b\} \times m.refs \mapsto 1]$

$]$

Add this message to the msgs bag.

$\land\ msgs' = put(msgs,\ m)$

$\land$ UNCHANGED $\langle snapshots\rangle$

$Receive(a,\ m) \triangleq$

$\land\quad actors' = [actors$ EXCEPT

$![a].active = @ ++ [c \in m.refs \mapsto 1],$

$![a].received = @ + 1,$

$![a].status =$ "busy"]

Remove m from the msgs bag.

$\land\ msgs' = remove(msgs,\ m)$

$\land$ UNCHANGED $\langle snapshots\rangle$

$Snapshot(a) \triangleq$

$\land\ snapshots[a] = null$

$\land\ snapshots' = [snapshots$ EXCEPT $![a] = actors[a]]$

$\land$ UNCHANGED $\langle msgs,\ actors\rangle$

$Init \triangleq$

$InitialConfiguration($CHOOSE $a \in ActorName$: TRUE, $InitialActorState)$

$Next \triangleq$

$\lor \exists\, a \in BusyActors : Idle(a)$

$\lor \exists\, a \in BusyActors : \exists\, b \in FreshActorName : Spawn(a,\ b,\ InitialActorState)$

$\lor \exists\, a \in BusyActors : \exists\, b \in acqs(a) : Deactivate(a,\ b)$

$\lor \exists\, a \in BusyActors : \exists\, b \in acqs(a) : \exists\, refs \in$ SUBSET $acqs(a)$:

$$
\begin{array}{l}
\quad Send(a,\ b,\ [target \mapsto b,\ refs \mapsto refs]) \\
\lor \exists\, a \in IdleActors : \exists\, m \in msgsTo(a) : Receive(a,\ m) \\
\lor \exists\, a \in IdleActors \cup BusyActors : Snapshot(a)
\end{array}
$$

$$
\begin{array}{l}
PotentiallyUnblocked \triangleq \\
\quad \textsc{choose}\ S \in \textsc{subset}\ Actors : \forall\, a,\ b \in Actors : \\
\quad \land\ (a \notin Blocked \implies a \in S) \\
\quad \land\ (a \in S \land a \in piacqs(b) \implies b \in S)
\end{array}
$$

$$
Quiescent \triangleq Actors \setminus PotentiallyUnblocked
$$

$$
\begin{array}{ll}
created(a,\ b) & \triangleq sum([c \in Snapshots \mapsto snapshots[c].created[a,\ b]]) \\
deactivated(a,\ b) & \triangleq \textsc{if}\ a \in Snapshots\ \textsc{then}\ snapshots[a].deactivated[b]\ \textsc{else}\ 0 \\
sent(b) & \triangleq sum([a \in Snapshots \mapsto snapshots[a].sent[b]]) \\
received(b) & \triangleq \textsc{if}\ b \in Snapshots\ \textsc{then}\ snapshots[b].received\ \textsc{else}\ 0
\end{array}
$$

$$
\begin{array}{ll}
heretoIAcqs(c) & \triangleq \{b \in ActorName : created(b,\ c) > 0\} \\
apparentIAcqs(c) & \triangleq \{b \in ActorName : created(b,\ c) > deactivated(b,\ c)\}
\end{array}
$$

$$
\begin{array}{ll}
AppearsIdle & \triangleq \{a \in Snapshots\ \ : snapshots[a].status = \text{“idle”}\} \\
AppearsClosed & \triangleq \{b \in Snapshots\ \ : heretoIAcqs(b) \subseteq Snapshots\} \\
AppearsBlocked & \triangleq \{b \in AppearsIdle \cap AppearsClosed : sent(b) = received(b)\} \\
AppearsUnblocked & \triangleq Snapshots \setminus AppearsBlocked
\end{array}
$$

$$
\begin{array}{l}
AppearsPotentiallyUnblocked \triangleq \\
\quad \textsc{choose}\ S \in \textsc{subset}\ Snapshots : \forall\, a,\ b \in Snapshots : \\
\quad \land\ (a \notin AppearsBlocked \implies a \in S) \\
\quad \land\ (a \in S \land a \in apparentIAcqs(b) \implies b \in S)
\end{array}
$$

$$
AppearsQuiescent \triangleq Snapshots \setminus AppearsPotentiallyUnblocked
$$

An actor's snapshot is up to date if its state has not changed since the last snapshot.

$$
SnapshotUpToDate(a) \triangleq actors[a] = snapshots[a]
$$

A snapshot from a past inverse acquaintance is recent enough if that the deactivated count in ths snapshot is up to date with the actual deactivated count.

$$
\begin{array}{l}
RecentEnough(a,\ b) \triangleq \\
\quad a \in Snapshots \land actors[a].deactivated[b] = snapshots[a].deactivated[b]
\end{array}
$$

A set of snapshots is insufficient for b if:

1. b's snapshot is out of date;
2. b has a previous inverse acquaintance whose snapshot is not recent enough; or
3. b is potentially reachable by an actor for which the snapshots are insufficient.

$$
\begin{aligned}
&SnapshotsInsufficient \triangleq \\
&\quad \text{CHOOSE } S \in \text{SUBSET } Actors : \forall\, a \in Actors : \\
&\quad \land\, (\neg SnapshotUpToDate(a) \implies a \in S) \\
&\quad \land\, \forall\, b \in Actors : \\
&\qquad \land\, (a \in pastIAcqs(b) \land \neg RecentEnough(a,\, b) \implies b \in S) \\
&\qquad \land\, (a \in S \land a \in piacqs(b) \implies b \in S)
\end{aligned}
$$

$$SnapshotsSufficient \triangleq Actors \setminus SnapshotsInsufficient$$

The specification captures the following properties:
1. Soundness: Every actor that appears quiescent is indeed quiescent.
2. Completeness: Every quiescent actor with a sufficient set of snapshots will appear quiescent.

$$Spec \triangleq (Quiescent \cap SnapshotsSufficient) = AppearsQuiescent$$

TEST CASES: These invariants do not hold, showing that interesting forms of garbage can indeed exist and be detected.

This invariant fails, showing that the set of quiescent actors is nonempty.

$$GarbageExists \triangleq \neg(Quiescent = \{\})$$

This invariant fails, showing that quiescence can be detected and that it is possible to obtain a sufficient set of snapshots.

$$GarbageIsDetected \triangleq \neg(AppearsQuiescent = \{\})$$

An actor b can appear quiescent when a past inverse acquaintance a is not quiescent. This is because a has deactivated all its references to b.

$$
\begin{aligned}
&DeactivatedGarbage \triangleq \\
&\neg(\exists\, a,\, b \in Actors : a \neq b \land a \notin Quiescent \land b \in AppearsQuiescent \land \\
&\quad actors[a].active[b] = 0 \land actors[a].deactivated[b] > 0)
\end{aligned}
$$

A.5 The MONITORS Model

MODULE *Monitors*

This model extends the Dynamic model with sticky actors and monitoring.

EXTENDS $Common$, $Integers$, $FiniteSets$, $Bags$, TLC

Operators from the Dynamic model are imported within the D namespace.

$D \triangleq$ INSTANCE $Dynamic$

$ActorState \triangleq [$
$\quad status$: {"busy", "idle", "halted"}, NEW: Actors may become "halted".
$\quad received$: Nat,
$\quad sent$: $[ActorName \to Nat]$,
$\quad active$: $[ActorName \to Nat]$,
$\quad deactivated$: $[ActorName \to Nat]$,
$\quad created$: $[ActorName \times ActorName \to Nat]$,
$\quad monitored$: SUBSET $ActorName$, NEW: The set of actors monitored by this one.
$\quad isSticky$: BOOLEAN NEW: Indicates whether this actor is a sticky actor.
]

$TypeOK \triangleq$
$\quad \land\ actors \in [ActorName \to ActorState \cup \{null\}]$
$\quad \land\ snapshots \in [ActorName \to ActorState \cup \{null\}]$
$\quad \land\ BagToSet(msgs) \subseteq D!Message$

$InitialActorState \triangleq$
$\quad D!InitialActorState$ @@ [
$\quad\quad monitored \mapsto \{\}$,
$\quad\quad isSticky \mapsto$ FALSE
$\quad$]

$InitialConfiguration(actor, actorState) \triangleq$
$\quad D!InitialConfiguration(actor, [actorState$ EXCEPT $!.isSticky =$ TRUE$])$

DEFINITIONS

$msgsTo(a) \triangleq D!msgsTo(a)$
$acqs(a) \triangleq D!acqs(a)$
$iacqs(b) \triangleq D!iacqs(b)$
$pacqs(a) \triangleq D!pacqs(a)$
$piacqs(b) \triangleq D!piacqs(b)$

$pastAcqs(a) \triangleq D!pastAcqs(a)$
$pastIAcqs(b) \triangleq D!pastIAcqs(b)$
$monitoredBy(b) \triangleq actors[b].monitored$
$appearsMonitoredBy(a) \triangleq snapshots[a].monitored$

$BusyActors \triangleq D!BusyActors$
$IdleActors \triangleq D!IdleActors$
$Blocked \triangleq D!Blocked$
$Unblocked \triangleq D!Unblocked$
$HaltedActors \triangleq \{a \in Actors : actors[a].status = \text{“halted”}\}$
$AppearsHalted \triangleq \{a \in Snapshots : snapshots[a].status = \text{“halted”}\}$
$StickyActors \triangleq \{a \in Actors : actors[a].isSticky\}$
$AppearsSticky \triangleq \{a \in Snapshots : snapshots[a].isSticky\}$

TRANSITIONS

$Idle(a) \triangleq D!Idle(a)$
$Deactivate(a, b) \triangleq D!Deactivate(a, b)$
$Send(a, b, m) \triangleq D!Send(a, b, m)$
$Receive(a, m) \triangleq D!Receive(a, m)$
$Snapshot(a) \triangleq D!Snapshot(a)$
$Spawn(a, b, state) \triangleq D!Spawn(a, b, state)$

$Halt(a) \triangleq$
$\land\ actors' = [actors \text{ EXCEPT } ![a].status = \text{“halted”}]$ Mark the actor as halted.
$\land$ UNCHANGED $\langle msgs, snapshots\rangle$

$Monitor(a, b) \triangleq$
$\land\ actors' = [actors \text{ EXCEPT } ![a].monitored = @ \cup \{b\}]$ Add b to the monitored set.
$\land$ UNCHANGED $\langle msgs, snapshots\rangle$

$Notify(a, b) \triangleq$
$\land\ actors' = [actors \text{ EXCEPT }$ Mark the monitor as busy and remove b from the monitored set.
$\quad ![a].status = \text{“busy”}, ![a].monitored = @ \setminus \{b\}]$
$\land$ UNCHANGED $\langle msgs, snapshots\rangle$

$Unmonitor(a, b) \triangleq$
$\land\ actors' = [actors \text{ EXCEPT } ![a].monitored = @ \setminus \{b\}]$ Remove b from the monitored set.

$\land$ UNCHANGED $\langle msgs,\ snapshots \rangle$

$Register(a) \triangleq$

$\land\ actors' = [actors$ EXCEPT $![a].isSticky =$ TRUE$]$

$\land$ UNCHANGED $\langle msgs,\ snapshots \rangle$

$Wakeup(a) \triangleq$

$\land\ actors' = [actors$ EXCEPT $![a].status =$ "busy"$]$

$\land$ UNCHANGED $\langle msgs,\ snapshots \rangle$

$Unregister(a) \triangleq$

$\land\ actors' = [actors$ EXCEPT $![a].isSticky =$ FALSE$]$

$\land$ UNCHANGED $\langle msgs,\ snapshots \rangle$

$Init \triangleq$

$InitialConfiguration($CHOOSE $a \in ActorName :$ TRUE$,\ InitialActorState)$

$Next \triangleq$

$\lor \exists\, a \in BusyActors : Idle(a)$

$\lor \exists\, a \in BusyActors : \exists\, b \in FreshActorName : Spawn(a,\ b,\ InitialActorState)$

$\lor \exists\, a \in BusyActors : \exists\, b \in acqs(a) : Deactivate(a,\ b)$

$\lor \exists\, a \in BusyActors : \exists\, b \in acqs(a) : \exists\, refs \in$ SUBSET $acqs(a) :$

$\quad Send(a,\ b,\ [target \mapsto b,\ refs \mapsto refs])$

$\lor \exists\, a \in IdleActors : \exists\, m \in msgsTo(a) : Receive(a,\ m)$

$\lor \exists\, a \in IdleActors \cup BusyActors \cup HaltedActors : Snapshot(a)$

NEW: Halted actors can now take snapshots.

$\lor \exists\, a \in BusyActors : Halt(a)$

$\lor \exists\, a \in BusyActors : \exists\, b \in acqs(a) : Monitor(a,\ b)$

$\lor \exists\, a \in IdleActors\ \ : \exists\, b \in HaltedActors \cap monitoredBy(a) : Notify(a,\ b)$

$\lor \exists\, a \in BusyActors : \exists\, b \in monitoredBy(a) : Unmonitor(a,\ b)$

$\lor \exists\, a \in BusyActors \setminus StickyActors : Register(a)$

$\lor \exists\, a \in IdleActors\ \ \cap StickyActors : Wakeup(a)$

$\lor \exists\, a \in BusyActors \cap StickyActors : Unregister(a)$

Non-halted sticky actors and unblocked actors are not garbage. Non-halted actors that are potentially reachable by non-garbage are not garbage. Non-halted actors that monitor actors that can halt or have halted are not garbage.

$$
\begin{array}{l}
PotentiallyUnblocked \triangleq \\
\quad \textsc{choose}\ S \in \textsc{subset}\ Actors : \\
\quad \land\ (StickyActors \cup Unblocked) \setminus HaltedActors \subseteq S \\
\quad \land\ \forall\, a \in Actors,\ b \in Actors \setminus HaltedActors : \\
\qquad \land\ (a \in S \cap piacqs(b) \implies b \in S) \\
\qquad \land\ (a \in (S \cup HaltedActors) \cap monitoredBy(b) \implies b \in S)
\end{array}
$$

$$Quiescent \triangleq Actors \setminus PotentiallyUnblocked$$

$$
\begin{array}{l}
AppearsUnblocked \triangleq D!AppearsUnblocked \\
apparentIAcqs(b) \triangleq D!apparentIAcqs(b) \\
AppearsClosed \triangleq D!AppearsClosed \cap \\
\qquad\qquad \{b \in Snapshots : appearsMonitoredBy(b) \subseteq Snapshots\}
\end{array}
$$

Each clause in this definition corresponds to one in *PotentiallyUnblocked*—with one addition: if an actor a has potential inverse acquaintances or monitored actors that have not taken a snapshot, then a should be marked as potentially unblocked for safety.

$$
\begin{array}{l}
AppearsPotentiallyUnblocked \triangleq \\
\quad \textsc{choose}\ S \in \textsc{subset}\ Snapshots : \\
\quad \land\ Snapshots \setminus (AppearsClosed \cup AppearsHalted) \subseteq S \\
\quad \land\ (AppearsSticky \cup AppearsUnblocked) \setminus AppearsHalted \subseteq S \\
\quad \land\ \forall\, a \in Snapshots,\ b \in Snapshots \setminus AppearsHalted : \\
\qquad \land\ (a \in S \cap apparentIAcqs(b) \implies b \in S) \\
\qquad \land\ (a \in (S \cup AppearsHalted) \cap appearsMonitoredBy(b) \implies b \in S)
\end{array}
$$

$$AppearsQuiescent \triangleq Snapshots \setminus AppearsPotentiallyUnblocked$$

$$
\begin{array}{l}
SnapshotUpToDate(a) \triangleq D!SnapshotUpToDate(a) \\
RecentEnough(a,\ b) \triangleq D!RecentEnough(a,\ b)
\end{array}
$$

$$
\begin{array}{l}
SnapshotsInsufficient \triangleq \\
\quad \textsc{choose}\ S \in \textsc{subset}\ Actors : \forall\, a \in Actors : \\
\quad \land\ (\neg SnapshotUpToDate(a) \implies a \in S) \\
\quad \land\ \forall\, b \in Actors \setminus HaltedActors : \\
\qquad \land\ (a \in pastIAcqs(b) \land \neg RecentEnough(a,\ b) \implies b \in S) \\
\qquad \land\ (a \in S \land a \in piacqs(b) \implies b \in S) \\
\qquad \land\ (a \in S \land a \in monitoredBy(b) \implies b \in S)\ \textsc{new}
\end{array}
$$

$SnapshotsSufficient \triangleq Actors \setminus SnapshotsInsufficient$

$Spec \triangleq (Quiescent \cap SnapshotsSufficient) = AppearsQuiescent$

TEST CASES: These invariants do not hold because garbage can be detected.

This invariant fails, showing that the set of quiescent actors is nonempty.

$GarbageExists \triangleq \neg(Quiescent = \{\})$

This invariant fails, showing that quiescence can be detected and that it is possible to obtain a sufficient set of snapshots.

$GarbageIsDetected \triangleq \neg(AppearsQuiescent = \{\})$

This invariant fails, showing that quiescent actors can have halted inverse acquaintances.

$$HaltedGarbageIsDetected \triangleq \neg(\exists\, a,\, b \in Actors : a \neq b \land a \in HaltedActors \land b \in AppearsQuiescent \land a \in iacqs(b))$$

The previous soundness property no longer holds because actors can now become busy by receiving signals from halted actors or messages from external actors.

$OldSoundness \triangleq D!AppearsQuiescent \subseteq Quiescent$

The previous completeness property no longer holds because snapshots from monitored actors need to be up to date.

$OldCompleteness \triangleq (Quiescent \cap D!SnapshotsSufficient) \subseteq AppearsQuiescent$

A.6 The Exile Model

MODULE *Exile*

This model extends the Monitors model with dropped messages and faulty nodes.

EXTENDS *Common*, *Integers*, *FiniteSets*, *Bags*, *TLC*

Every node has a unique ID and every actor is located at some node. Every pair of nodes has an ingress actor that tracks when messages are dropped and when messages arrive at a node. Ingress actors can take snapshots. There is also a temporary holding area for messages that have been dropped but whose recipient has not yet learned of the drop.

CONSTANT *NodeID*

VARIABLE *location*, *ingress*, *ingressSnapshots*, *droppedMsgs*

$D \triangleq$ INSTANCE $Dynamic$
$M \triangleq$ INSTANCE $Monitors$

We add two fields to every message. *origin* indicates the node that produced the message and *admitted* indicates whether the message was admitted into the destination node by the ingress actor. All messages between actors on distinct nodes must be admitted before they can be received by the destination actor. Messages between actors on the same node are admitted by default.

$$
\begin{array}{l}
Message \triangleq [\\
\quad origin : NodeID, \\
\quad admitted : \text{BOOLEAN}, \\
\quad target : ActorName, \\
\quad refs : \text{SUBSET } ActorName \\
]
\end{array}
$$

INITIALIZATION AND BASIC INVARIANTS

$ActorState \triangleq M!ActorState$

$$
\begin{array}{l}
IngressState \triangleq [\\
\quad shunned \quad\quad : \text{BOOLEAN}, \\
\quad admittedMsgs : [ActorName \to Nat], \\
\quad admittedRefs \; : [ActorName \times ActorName \to Nat] \\
]
\end{array}
$$

The following invariant specifies the type of every variable in the configuration. It also asserts that every actor, once spawned, has a location; and every message in *droppedMsgs* must have first been admitted.

$$
\begin{array}{lll}
TypeOK \triangleq & & \\
\quad \land\ actors & \in [ActorName \to ActorState \cup \{null\}] & \\
\quad \land\ snapshots & \in [ActorName \to ActorState \cup \{null\}] & \\
\quad \land\ BagToSet(msgs) & \subseteq Message & \\
\quad \land\ BagToSet(droppedMsgs) & \subseteq Message & \\
\quad \land\ location & \in [ActorName \to NodeID \cup \{null\}] & \text{NEW} \\
\quad \land\ ingress & \in [NodeID \times NodeID \to IngressState] & \text{NEW} \\
\quad \land\ ingressSnapshots & \in [NodeID \times NodeID \to IngressState] & \text{NEW} \\
\quad \land \forall\, a \in Actors : location[a] \neq null & & \\
\quad \land \forall\, m \in BagToSet(droppedMsgs) : m.admitted & &
\end{array}
$$

$InitialActorState \triangleq M!InitialActorState$

$$
\begin{array}{l}
InitialIngressState \triangleq [\\
\quad shunned \mapsto \text{FALSE}, \\
\quad admittedMsgs \mapsto [a \in ActorName \mapsto 0], \\
\quad admittedRefs \mapsto [a, b \in ActorName \mapsto 0] \\
]
\end{array}
$$

$$
\begin{array}{l}
InitialConfiguration(initialActor, node, actorState) \triangleq \\
\quad \land M!InitialConfiguration(initialActor, actorState) \\
\quad \land ingress = [N1, N2 \in NodeID \mapsto InitialIngressState] \\
\quad \land ingressSnapshots = [N1, N2 \in NodeID \mapsto InitialIngressState] \\
\quad \land location = (initialActor :> node) @@ [a \in ActorName \mapsto null] \\
\quad \land droppedMsgs = EmptyBag
\end{array}
$$

DEFINITIONS

A message is admissible if it is not already admitted and the origin node is not shunned by the destination node.

$$
\begin{array}{l}
AdmissibleMsgs \triangleq \{m \in BagToSet(msgs) : \\
\quad \neg m.admitted \land \neg ingress[m.origin, location[m.target]].shunned\} \\
AdmittedMsgs \triangleq \{m \in BagToSet(msgs) : m.admitted\}
\end{array}
$$

Because inadmissible messages can never be delivered, we update the definition of *msgsTo* to exclude them. This causes several other definitions below to change in subtle ways. For example, an actor *a* is potentially acquainted with *b* if all there is an inadmissible message to *a* containing a reference to *b*.

$$
\begin{array}{ll}
msgsTo(a) & \triangleq \{m \in M!msgsTo(a) : m.admitted \lor m \in AdmissibleMsgs\} \\
acqs(a) & \triangleq M!acqs(a) \\
iacqs(b) & \triangleq M!iacqs(b) \\
pacqs(a) & \triangleq \{b \in ActorName : b \in acqs(a) \lor \exists\, m \in msgsTo(a) : b \in m.refs\} \\
piacqs(b) & \triangleq \{a \in Actors : b \in pacqs(a)\} \\
pastAcqs(a) & \triangleq M!pastAcqs(a) \\
pastIAcqs(b) & \triangleq M!pastIAcqs(b) \\
monitoredBy(b) & \triangleq M!monitoredBy(b) \\
appearsMonitoredBy(b) & \triangleq M!appearsMonitoredBy(b) \\
admittedMsgsTo(a) & \triangleq \{m \in msgsTo(a) : m.admitted\}
\end{array}
$$

Below, an actor can be blocked if all messages to it are inadmissible.

$$
\begin{array}{ll}
BusyActors & \triangleq M!BusyActors \\
IdleActors & \triangleq M!IdleActors
\end{array}
$$

$$
\begin{aligned}
&Blocked &&\triangleq \{a \in IdleActors : msgsTo(a) = \{\}\} \\
&Unblocked &&\triangleq Actors \setminus Blocked \\
&HaltedActors &&\triangleq M!HaltedActors \\
&AppearsHalted &&\triangleq M!AppearsHalted \\
&StickyActors &&\triangleq M!StickyActors \\
&AppearsSticky &&\triangleq M!AppearsSticky
\end{aligned}
$$

$$
\begin{aligned}
&ShunnedBy(N2) &&\triangleq \{N1 \in NodeID : ingress[N1,\ N2].shunned\} \\
&ShunnableBy(N1) &&\triangleq (NodeID \setminus \{N1\}) \setminus ShunnedBy(N1) \\
&NeitherShuns(N1) &&\triangleq \{N2 \in NodeID : \neg ingress[N1,\ N2].shunned \land \\
&&&\qquad\qquad \neg ingress[N2,\ N1].shunned\}
\end{aligned}
$$

The exiled nodes are the largest nontrivial faction where every non-exiled node has shunned every exiled node. Likewise, a faction of nodes G is apparently exiled if every node outside of G has taken an ingress snapshot in which every node of G is shunned.

$$
\begin{aligned}
&ExiledNodes \triangleq \\
&\quad LargestSubset(NodeID,\ \textsc{lambda}\ G : \\
&\qquad \land\ G \neq NodeID \\
&\qquad \land\ \forall\, N1 \in G,\ N2 \in NodeID \setminus G : ingress[N1,\ N2].shunned \\
&\quad)
\end{aligned}
$$

$$
\begin{aligned}
&ApparentlyExiledNodes \triangleq \\
&\quad LargestSubset(NodeID,\ \textsc{lambda}\ G : \\
&\qquad \land\ G \neq NodeID \\
&\qquad \land\ \forall\, N1 \in G,\ N2 \in NodeID \setminus G : ingressSnapshots[N1,\ N2].shunned \\
&\quad)
\end{aligned}
$$

$$
\begin{aligned}
&NonExiledNodes &&\triangleq NodeID \setminus ExiledNodes \\
&ExiledActors &&\triangleq \{a \in Actors : location[a] \in ExiledNodes\} \\
&NonExiledActors &&\triangleq Actors \setminus ExiledActors \\
&FailedActors &&\triangleq HaltedActors \cup ExiledActors \\
&HealthyActors &&\triangleq Actors \setminus FailedActors
\end{aligned}
$$

$$
\begin{aligned}
&ApparentlyNonExiledNodes &&\triangleq NodeID \setminus ApparentlyExiledNodes \\
&ApparentlyExiledActors &&\triangleq \{a \in Actors : location[a] \in ApparentlyExiledNodes\} \\
&ApparentlyNonExiledActors &&\triangleq Actors \setminus ApparentlyExiledActors \\
&AppearsFailed &&\triangleq M!AppearsHalted \cup ApparentlyExiledActors \\
&AppearsHealthy &&\triangleq Actors \setminus AppearsFailed
\end{aligned}
$$

$$
NonExiledSnapshots \triangleq Snapshots \setminus ApparentlyExiledActors
$$

$droppedMsgsTo(a) \triangleq \{m \in BagToSet(droppedMsgs) : m.target = a\}$
$droppedPIAcqs(b) \triangleq \{a \in Actors : \exists\, m \in droppedMsgsTo(a) : b \in m.refs\}$

TRANSITIONS

$Idle(a) \triangleq M!Idle(a)$
$\land$ UNCHANGED $\langle location, ingress, ingressSnapshots, droppedMsgs\rangle$

$Deactivate(a, b) \triangleq M!Deactivate(a, b)$
$\land$ UNCHANGED $\langle location, ingress, ingressSnapshots, droppedMsgs\rangle$

$Send(a, b, m) \triangleq M!Send(a, b, m)$
$\land$ UNCHANGED $\langle location, ingress, ingressSnapshots, droppedMsgs\rangle$

$Receive(a, m) \triangleq M!Receive(a, m)$
$\land$ UNCHANGED $\langle location, ingress, ingressSnapshots, droppedMsgs\rangle$

$Snapshot(a) \triangleq M!Snapshot(a)$
$\land$ UNCHANGED $\langle location, ingress, ingressSnapshots, droppedMsgs\rangle$

$Halt(a) \triangleq M!Halt(a)$
$\land$ UNCHANGED $\langle location, ingress, ingressSnapshots, droppedMsgs\rangle$

$Monitor(a, b) \triangleq M!Monitor(a, b)$
$\land$ UNCHANGED $\langle location, ingress, ingressSnapshots, droppedMsgs\rangle$

$Unmonitor(a, b) \triangleq M!Unmonitor(a, b)$
$\land$ UNCHANGED $\langle location, ingress, ingressSnapshots, droppedMsgs\rangle$

$Notify(a, b) \triangleq M!Notify(a, b)$
$\land$ UNCHANGED $\langle location, ingress, ingressSnapshots, droppedMsgs\rangle$

$Register(a) \triangleq M!Register(a)$
$\land$ UNCHANGED $\langle location, ingress, ingressSnapshots, droppedMsgs\rangle$

$Wakeup(a) \triangleq M!Wakeup(a)$
$\land$ UNCHANGED $\langle location, ingress, ingressSnapshots, droppedMsgs\rangle$

$Unregister(a) \triangleq M!Unregister(a)$
$\land$ UNCHANGED $\langle location, ingress, ingressSnapshots, droppedMsgs\rangle$

$Spawn(a, b, state, N) \triangleq$
$\land\ M!Spawn(a, b, state)$
$\land\ location' = [location$ EXCEPT $![b] = N]$
$\land$ UNCHANGED $\langle msgs, ingress, ingressSnapshots, droppedMsgs\rangle$

$Admit(m) \triangleq$
LET $a \triangleq m.target$

$N1 \triangleq m.origin$
$N2 \triangleq location[a]$
$B \triangleq [\langle b, c\rangle \in \{a\} \times m.refs \mapsto 1]$

IN

$\land\ ingress' = [ingress \text{ EXCEPT } ![N1, N2].admittedMsgs[a] = @ + 1,$
$\qquad\qquad ![N1, N2].admittedRefs = @ \mathbin{++} B]$
$\land\ msgs' = replace(msgs, m, [m \text{ EXCEPT } !.admitted = \text{TRUE}])$
$\land$ UNCHANGED $\langle actors, snapshots, ingressSnapshots, location, droppedMsgs\rangle$

Dropped messages are admitted (if necessary) by the recipient ingress actor and then added to the droppedMsgs bag.

$Drop(m) \triangleq$
$\land$ IF $\neg m.admitted$ THEN
LET $a \triangleq m.target$
$N1 \triangleq m.origin$
$N2 \triangleq location[a]$
$B \triangleq [\langle b, c\rangle \in \{a\} \times m.refs \mapsto 1]$
IN
$ingress' = [ingress \text{ EXCEPT } ![N1, N2].admittedMsgs[a] = @ + 1,$
$\qquad ![N1, N2].admittedRefs = @ \mathbin{++} B]$
ELSE UNCHANGED $\langle ingress\rangle$
$\land\ msgs' = remove(msgs, m)$
$\land\ droppedMsgs' = put(droppedMsgs, [m \text{ EXCEPT } !.admitted = \text{TRUE}])$
$\land$ UNCHANGED $\langle actors, snapshots, ingressSnapshots, location\rangle$

Some time after a message has been dropped, the recipient actor's local state is updated.

$DetectDropped(a, m) \triangleq$
$\land\ droppedMsgs' = remove(droppedMsgs, m)$
$\land\ actors' = [actors \text{ EXCEPT } ![a].received = @ + 1,$
$\qquad ![a].deactivated = @ \mathbin{++} [c \in m.refs \mapsto 1]]$
$\land$ UNCHANGED $\langle msgs, snapshots, ingress, ingressSnapshots, location\rangle$

When N2 shuns N1, the ingress actor at N2 is updated. If N1 is now exiled, we mark the actors as "exiled" to prevent them from taking snapshots.

$Shun(N1, N2) \triangleq$
$\land\ ingress' = [ingress \text{ EXCEPT } ![N1, N2].shunned = \text{TRUE}]$
$\land$ UNCHANGED $\langle actors, msgs, snapshots, ingressSnapshots, location, droppedMsgs\rangle$

To reduce the model checking state space, the *Shun* rule can be replaced with the following *Exile* rule. This is safe because, for any execution in which a faction G1 all shuns another faction G2, there is an equivalent execution in which all *Shun* events happen successively.

$$
\begin{array}{l}
Exile(G1,\ G2)\ \triangleq \\
\quad \land\ ingress' = \\
\qquad [N1 \in G1,\ N2 \in G2 \mapsto [ingress[N1,\ N2]\ \textsc{except}\ !.shunned = \textsc{true}]]\ @@\ ingress \\
\quad \land\ \textsc{unchanged}\ \langle actors,\ msgs,\ snapshots,\ ingressSnapshots,\ location,\ droppedMsgs\rangle
\end{array}
$$

Ingress actors can record snapshots.

$$
\begin{array}{l}
IngressSnapshot(N1,\ N2)\ \triangleq \\
\quad \land\ ingressSnapshots' = [ingressSnapshots\ \textsc{except}\ ![N1,\ N2] = ingress[N1,\ N2]] \\
\quad \land\ \textsc{unchanged}\ \langle actors,\ msgs,\ snapshots,\ ingress,\ location,\ droppedMsgs\rangle
\end{array}
$$

$$
\begin{array}{ll}
Init\ \triangleq\ InitialConfiguration(& \\
\quad \textsc{choose}\ a \in ActorName : \textsc{true}, & \text{An arbitrary name for the initial actor.} \\
\quad \textsc{choose}\ n \in NodeID : \textsc{true}, & \text{An arbitrary location for the actor.} \\
\quad InitialActorState & \text{The initial actor's state.} \\
)
\end{array}
$$

Several transition rules have been updated to account for locations. In addition, every rule is modified so that exiled actors no longer take actions.

$$
\begin{array}{l}
Next\ \triangleq \\
\quad \lor \exists\, a\ \in BusyActors \setminus ExiledActors\ : Idle(a) \\
\quad \lor \exists\, a\ \in BusyActors \setminus ExiledActors\ : \exists\, b \in FreshActorName : \\
\qquad \exists\, N \in NeitherShuns(location[a]) : Spawn(a,\ b,\ InitialActorState,\ N) \\
\qquad\quad \text{UPDATE: Actors are spawned onto a specific (non-shunned) node.} \\
\quad \lor \exists\, a \in BusyActors \setminus ExiledActors : \exists\, b \in acqs(a) : Deactivate(a,\ b) \\
\quad \lor \exists\, a \in BusyActors \setminus ExiledActors : \exists\, b \in acqs(a) : \exists\, refs \in \textsc{subset}\ acqs(a) : \\
\qquad Send(a,\ b,\ [origin \mapsto location[a],\ admitted \mapsto location[b] = location[a], \\
\qquad\qquad\qquad target \mapsto b,\ refs \mapsto refs]) \\
\qquad\quad \text{UPDATE: Messages are tagged with node locations.} \\
\quad \lor \exists\, a \in IdleActors \setminus ExiledActors : \exists\, m \in admittedMsgsTo(a) : Receive(a,\ m) \\
\quad \lor \exists\, a \in Actors \setminus ExiledActors : Snapshot(a) \\
\quad \lor \exists\, a \in BusyActors \setminus ExiledActors : Halt(a) \\
\quad \lor \exists\, a \in BusyActors \setminus ExiledActors : \exists\, b \in acqs(a) : Monitor(a,\ b) \\
\quad \lor \exists\, a \in IdleActors\ \setminus ExiledActors : \exists\, b \in FailedActors \cap monitoredBy(a) :
\end{array}
$$

$Notify(a,\ b)$

UPDATE: Actors are notified when monitored actors are exiled.

$\lor \exists\, a \in BusyActors \setminus ExiledActors : \exists\, b \in monitoredBy(a) : Unmonitor(a,\ b)$

$\lor \exists\, a \in (BusyActors \setminus StickyActors) \setminus ExiledActors : Register(a)$

$\lor \exists\, a \in (IdleActors \cap StickyActors) \setminus ExiledActors : Wakeup(a)$

$\lor \exists\, a \in (BusyActors \cap StickyActors) \setminus ExiledActors : Unregister(a)$

$\lor \exists\, m \in AdmissibleMsgs : location[m.target] \notin ExiledNodes \land Admit(m)$ NEW

$\lor \exists\, m \in AdmissibleMsgs \cup AdmittedMsgs : location[m.target] \notin ExiledNodes \land Drop(m)$ NEW

$\lor \exists\, a \in IdleActors \setminus ExiledActors : \exists\, m \in droppedMsgsTo(a) :$

$DetectDropped(m.target,\ m)$ NEW

$\lor \exists\, N1 \in NodeID : \exists\, N2 \in NonExiledNodes :$

$ingress[N1,\ N2] \neq ingressSnapshots[N1,\ N2] \land IngressSnapshot(N1,\ N2)$ NEW

To reduce the TLA+ search space, ingress actors do not take snapshots if

their state has not changed.

$\lor \exists\, N2 \in NonExiledNodes : \exists\, N1 \in ShunnableBy(N2) : Shun(N1,\ N2)$ NEW

ACTUAL GARBAGE

An actor is potentially unblocked if it is busy or can become busy. (Halted and exiled actors automatically fail this definition.) Similarly, an actor is potentially unblocked up-to-a-fault if it is busy or it can become busy in a non-faulty extension of this execution.

$isPotentiallyUnblockedUpToAFault(S) \triangleq$

$\land\ StickyActors \setminus FailedActors \subseteq S$

$\land\ Unblocked \setminus FailedActors \subseteq S$

$\land \forall\, a \in S,\ b \in HealthyActors :$

$a \in piacqs(b) \implies b \in S$

$\land \forall\, a \in S \cup FailedActors,\ b \in HealthyActors :$

$a \in monitoredBy(b) \implies b \in S$

NEW: An actor is not garbage if it monitors an exiled actor.

An actor is potentially unblocked if it is potentially unblocked up-to-a-fault or it monitors any remote actor. This is because remote actors can always become exiled, causing the monitoring actor to be notified.

$isPotentiallyUnblocked(S) \triangleq$

$\land\ isPotentiallyUnblockedUpToAFault(S)$

$\land \forall\, a \in Actors,\ b \in HealthyActors :$

$$\land\ (a \in monitoredBy(b) \land location[a] \neq location[b] \implies b \in S)$$

An actor is quiescent if it is not potentially unblocked. Likewise for quiescence up-to-a-fault.

$$PotentiallyUnblockedUpToAFault \triangleq$$
$$\quad \textsc{choose}\ S \in \textsc{subset}\ HealthyActors : isPotentiallyUnblockedUpToAFault(S)$$
$$QuiescentUpToAFault \triangleq Actors \setminus PotentiallyUnblockedUpToAFault$$

$$PotentiallyUnblocked \triangleq$$
$$\quad \textsc{choose}\ S \in \textsc{subset}\ HealthyActors : isPotentiallyUnblocked(S)$$
$$Quiescent \triangleq Actors \setminus PotentiallyUnblocked$$

Both definitions characterize a subset of the idle actors. The difference between the definitions is that quiescence up-to-a-fault is only a stable property in non-faulty executions.

$$QuiescentImpliesIdle \triangleq Quiescent \subseteq (IdleActors \cup FailedActors)$$
$$QuiescentUpToAFaultImpliesIdle \triangleq QuiescentUpToAFault \subseteq (IdleActors \cup FailedActors)$$

APPARENT GARBAGE

The effective created count is the sum of (a) the created counts recorded by non-exiled actors and (b) the created counts recorded by ingress actors for exiled nodes.

$$created(a,\ b) \triangleq$$
$$\quad sum([c \in NonExiledSnapshots \mapsto snapshots[c].created[a,\ b]]) +$$
$$\quad sum([N1 \in ApparentlyExiledNodes,\ N2 \in NodeID \setminus ApparentlyExiledNodes \mapsto$$
$$\qquad ingressSnapshots[N1,\ N2].admittedRefs[a,\ b]])$$

$$deactivated(a,\ b) \triangleq D!deactivated(a,\ b)$$

Once an actor a is exiled, the number of messages that a sent effectively to some b is equal to the number of messages admitted by the ingress actor at b's node. Thus the effective total send count for b is the sum of the send counts from non-exiled actors and the number of messages for b that entered the ingress actor from apparently exiled nodes. Note that dropped messages to b are implicitly included in the sum.

$$sent(b) \triangleq$$
$$\quad sum([a \in NonExiledSnapshots \mapsto snapshots[a].sent[b]]) +$$
$$\quad sum([N1 \in ApparentlyExiledNodes \mapsto ingressSnapshots[N1,\ location[b]].admittedMsgs[b]])$$

$$received(b) \triangleq D!received(b)$$

Hereto inverse acquaintances now incorporate ingress snapshot information. Once an actor appears exiled, it is no longer considered a hereto inverse acquaintance.

$$heretoIAcqs(c) \triangleq \{b \in Actors : created(b,\ c) > 0\}$$

$$apparentIAcqs(c) \triangleq \{b \in Actors : created(b, c) > deactivated(b, c)\}$$

$$AppearsIdle \triangleq \{a \in NonExiledSnapshots : snapshots[a].status = \text{“idle”}\}$$

$$\begin{aligned}
AppearsClosed \triangleq\ & \{b \in NonExiledSnapshots : \\
& \land heretoIAcqs(b) \subseteq Snapshots \cup ApparentlyExiledActors \\
& \land appearsMonitoredBy(b) \subseteq Snapshots \cup ApparentlyExiledActors\}
\end{aligned}$$

$$AppearsBlocked \triangleq \{b \in NonExiledSnapshots \cap AppearsIdle : sent(b) = received(b)\}$$

$$AppearsUnblocked \triangleq NonExiledSnapshots \setminus AppearsBlocked$$

$$\begin{aligned}
& appearsPotentiallyUnblockedUpToAFault(S) \triangleq \\
& \quad \land Snapshots \setminus (AppearsClosed \cup AppearsFailed) \subseteq S \\
& \quad \land AppearsSticky \setminus AppearsFailed \subseteq S \\
& \qquad \text{NEW: Exiled actors still appear potentially unblocked.} \\
& \quad \land AppearsUnblocked \setminus AppearsFailed \subseteq S \\
& \quad \land \forall\, a \in S,\ b \in Snapshots \setminus AppearsFailed : \\
& \qquad a \in apparentIAcqs(b) \implies b \in S \\
& \quad \land \forall\, a \in S \cup AppearsFailed,\ b \in Snapshots \setminus AppearsFailed : \\
& \qquad a \in appearsMonitoredBy(b) \implies b \in S \\
& \qquad\quad \text{NEW: An actor is not garbage if it monitors an exiled actor.}
\end{aligned}$$

$$\begin{aligned}
& appearsPotentiallyUnblocked(S) \triangleq \\
& \quad \land appearsPotentiallyUnblockedUpToAFault(S) \\
& \quad \land \forall\, a \in Actors,\ b \in Snapshots \setminus AppearsFailed : \\
& \qquad \land (a \in appearsMonitoredBy(b) \land location[a] \neq location[b] \implies b \in S) \\
& \qquad\quad \text{NEW: Actors that monitor remote actors are not garbage.}
\end{aligned}$$

$$\begin{aligned}
& AppearsPotentiallyUnblockedUpToAFault \triangleq \\
& \quad \textsc{choose}\ S \in \textsc{subset}\ Snapshots \setminus AppearsFailed : \\
& \quad appearsPotentiallyUnblockedUpToAFault(S)
\end{aligned}$$

$$\begin{aligned}
& AppearsQuiescentUpToAFault \triangleq \\
& \quad Snapshots \setminus AppearsPotentiallyUnblockedUpToAFault
\end{aligned}$$

$$\begin{aligned}
& AppearsPotentiallyUnblocked \triangleq \\
& \quad \textsc{choose}\ S \in \textsc{subset}\ Snapshots \setminus AppearsFailed : \\
& \quad appearsPotentiallyUnblocked(S)
\end{aligned}$$

$$\begin{aligned}
& AppearsQuiescent \triangleq \\
& \quad Snapshots \setminus AppearsPotentiallyUnblocked
\end{aligned}$$

SOUNDNESS AND COMPLETENESS PROPERTIES

Exiled actors may need to appear exiled in order for all quiescent garbage to be detected.

$SnapshotUpToDate(a) \triangleq$

IF $a \in ExiledActors$ THEN $a \in ApparentlyExiledActors$ ELSE

IF $a \in HaltedActors$ THEN $a \in AppearsHalted$ ELSE $M!SnapshotUpToDate(a)$

$RecentEnough(a, b) \triangleq$

IF $a \in ExiledActors$ THEN $a \in ApparentlyExiledActors$ ELSE

IF $a \in HaltedActors$ THEN $a \in AppearsHalted$ ELSE $M!RecentEnough(a, b)$

$BeingExiled(a) \triangleq$

$\exists N \in NonExiledNodes :$

$location[a] \in ShunnedBy(N) \land location[a] \notin ExiledNodes$

$SnapshotsInsufficient \triangleq$

CHOOSE $S \in$ SUBSET $Actors :$

$\land \forall N1, N2 \in ApparentlyNonExiledNodes : N1 \in ShunnedBy(N2) \implies$

$Actors \subseteq S$

NEW: Shunning creates garbage actors that might not be detected

until those nodes are apparently exiled.

$\land \forall b \in Actors :$

$\land \neg SnapshotUpToDate(b) \implies b \in S$

$\land b \in HealthyActors \land droppedMsgsTo(b) \neq \{\} \implies b \in S$

NEW: Actors may need to be notified about dropped references.

$\land \forall a \in Actors :$

$\land a \in pastIAcqs(b) \land \neg RecentEnough(a, b) \implies b \in S$

$\land a \in S \land a \in piacqs(b) \implies b \in S$

$\land a \in S \land a \in monitoredBy(b) \implies b \in S$

$\land a \in S \land a \in droppedPIAcqs(b) \implies b \in S$

NEW: Recipients of dropped messages containing references to b

may need to have sufficient snapshots.

$SnapshotsSufficient \triangleq Actors \setminus SnapshotsInsufficient$

The specification states that a non-exiled actor appears quiescent if and only if it is actually quiescent and there are sufficient snapshots to diagnose quiescence.

$Spec \triangleq$

$\land AppearsQuiescent \subseteq Quiescent$

$\land Quiescent \subseteq AppearsQuiescent \cup SnapshotsInsufficient \cup ExiledActors$

For quiescence up-to-a-fault, the simple specification above is not sufficient. This is because an actor that is quiescent up-to-a-fault can become busy if it monitors a remote actor that became exiled.

$$
\begin{aligned}
&SpecUpToAFault \triangleq \\
&\quad (\forall\, a \in AppearsQuiescentUpToAFault : \forall\, b \in appearsMonitoredBy(a) : b \notin ExiledActors) \\
&\quad \Longrightarrow \\
&\quad \land\ AppearsQuiescentUpToAFault \subseteq QuiescentUpToAFault \\
&\quad \land\ QuiescentUpToAFault \subseteq \\
&\qquad AppearsQuiescentUpToAFault \cup SnapshotsInsufficient \cup ExiledActors
\end{aligned}
$$

TEST CASES: These invariants do not hold because garbage can be detected.

$$ActorsCanBeSpawned \triangleq Cardinality(Actors) < 4$$
$$MessagesCanBeReceived \triangleq \forall\, a \in Actors : actors[a].received = 0$$
$$ActorsCanBeExiled \triangleq \forall\, a \in Actors : a \notin ExiledActors$$
$$SelfMessagesCanBeReceived \triangleq \forall\, a \in Actors : actors[a].received = 0 \lor Cardinality(Actors) > 1$$

This invariant fails, showing that the set of quiescent actors is nonempty.

$$GarbageExists \triangleq Quiescent = \{\}$$
$$HealthyGarbageExists \triangleq Quiescent \cap HealthyActors = \{\}$$

$$GarbageUpToAFaultExists \triangleq QuiescentUpToAFault = \{\}$$
$$HealthyGarbageUpToAFaultExists \triangleq QuiescentUpToAFault \cap HealthyActors = \{\}$$

This invariant fails, showing that quiescence can be detected and that it is possible to obtain a sufficient set of snapshots.

$$GarbageIsDetected \triangleq AppearsQuiescent = \{\}$$
$$HealthyGarbageIsDetected \triangleq AppearsQuiescent \setminus AppearsHalted = \{\}$$
$$GarbageIsDetectedUpToAFault \triangleq AppearsQuiescentUpToAFault = \{\}$$
$$HealthyGarbageIsDetectedUpToAFault \triangleq AppearsQuiescentUpToAFault \setminus AppearsHalted = \{\}$$

$$DistinctGarbageUpToAFault \triangleq AppearsQuiescentUpToAFault = AppearsQuiescent$$

This invariant fails, showing that quiescent actors can have halted inverse acquaintances.

$$
\begin{aligned}
&ExiledGarbageIsDetected \triangleq \\
&\quad \neg(\exists\, a,\, b \in Actors : a \neq b \land a \in ExiledActors \land b \in AppearsQuiescent \land \\
&\qquad a \in iacqs(b))
\end{aligned}
$$

This invariant fails, showing that "quiescence up to a fault" is a strict superset of quiescence.

$GarbageUpToAFault \triangleq AppearsQuiescentUpToAFault \subseteq AppearsQuiescent$

A.7 The SHADOWS Model

MODULE *Shadows*

EXTENDS *Common*, *Integers*, *FiniteSets*, *Bags*, *TLC*

$D \triangleq$ INSTANCE *Dynamic*

$M \triangleq$ INSTANCE *Monitors*

SHADOW GRAPHS

A Shadow is a node in the shadow graph. Each Shadow in the graph corresponds to an actor that has taken a snapshot or is referenced in another actor's snapshot.

- *interned* indicates whether this actor has taken a snapshot. If
- *interned* is FALSE, the values of *sticky* and *status* are meaningless.
- *sticky* indicates whether the actor was sticky in its latest snapshot.
- *status* indicates the status of the actor in its latest snapshot.
- *undelivered* is the number of messages that appear sent but not received.
- *references* is the number of references that appear created but not deactivated.
- *watchers* is the set of actors that appear to monitor this actor.

$Shadow \triangleq$ [
  $interned$: BOOLEAN ,
  $sticky$: BOOLEAN ,
  $status$: {"idle", "busy", "halted"},
  $undelivered$: Int,
  $references$: [$ActorName \to Int$],
  $watchers$: SUBSET $ActorName$
]

Shadow graphs are represented here as an indexed collection of shadows.

$ShadowGraph \triangleq [ActorName \to Shadow \cup \{null\}]$

$undelivered(b) \triangleq D!sent(b) - D!received(b)$

$references(a, b) \triangleq D!created(a, b) - D!deactivated(a, b)$

$watches(a,\ b) \triangleq\ a \in Snapshots \land b \in snapshots[a].monitored$

This is the domain of the shadow graph. An actor is in the shadow graph if it occurs in a snapshot.

$$
\begin{array}{l}
Shadows \triangleq \\
\quad \{c \in ActorName : \\
\qquad \lor\ c \in Snapshots \\
\qquad \lor \exists\, a \in Snapshots : \exists\, b \in ActorName : snapshots[a].created[b,\ c] > 0 \\
\qquad \lor \exists\, a \in Snapshots : \exists\, b \in ActorName : snapshots[a].created[c,\ b] > 0 \\
\qquad \lor \exists\, a \in Snapshots : snapshots[a].deactivated[c] > 0 \\
\qquad \lor \exists\, a \in Snapshots : snapshots[a].sent[c] > 0 \\
\qquad \lor \exists\, a \in Snapshots : c \in snapshots[a].monitored \\
\quad \}
\end{array}
$$

This is the shadow graph representation of the collage stored in *snapshots*.

$$
\begin{array}{l}
shadows \triangleq \\
\quad [b \in Shadows \mapsto \\
\qquad [\\
\qquad\quad interned \quad \mapsto b \in Snapshots, \\
\qquad\quad sticky \quad \mapsto \text{IF } b \in Snapshots \text{ THEN } snapshots[b].isSticky \text{ ELSE } \text{FALSE}, \\
\qquad\quad status \quad \mapsto \text{IF } b \in Snapshots \text{ THEN } snapshots[b].status \text{ ELSE } \text{“idle”}, \\
\qquad\quad undelivered \quad \mapsto undelivered(b), \\
\qquad\quad references \quad \mapsto [c \in ActorName \mapsto references(b,\ c)], \\
\qquad\quad watchers \quad \mapsto \{a \in ActorName : watches(a,\ b)\} \\
\qquad] \\
\quad]
\end{array}
$$

$$
\begin{array}{l}
AppearsFaulty(G) \triangleq \\
\quad \{a \in \text{DOMAIN } G : G[a].status = \text{“halted”}\}
\end{array}
$$

$$
\begin{array}{l}
PseudoRoots(G) \triangleq \\
\quad \{a \in \text{DOMAIN } G \setminus AppearsFaulty(G) : \\
\qquad \neg G[a].interned \lor G[a].sticky \lor G[a].status = \text{“busy”} \lor G[a].undelivered \neq 0 \lor \\
\qquad \exists\, b \in \text{DOMAIN } G : G[b].status = \text{“halted”} \land a \in G[b].watchers \\
\quad \}
\end{array}
$$

$$
\begin{array}{l}
acquaintances(G,\ a) \triangleq \\
\quad \{b \in \text{DOMAIN } G : G[a].references[b] > 0\}
\end{array}
$$

$watchers(G,\ a) \triangleq \{b \in \text{DOMAIN } G : b \in G[a].watchers\}$

In the shadow graph G, an actor is marked iff 0. It is a pseudo-root; 1. A potentially unblocked actor appears acquainted with it; or 2. A potentially unblocked actor is monitored by it.

$marked(G) \triangleq$
  CHOOSE $S \in$ SUBSET (DOMAIN G) $\setminus$ $AppearsFaulty(G)$:
  $\land$ $PseudoRoots(G) \subseteq S$
  $\land \forall\, a \in S$:
    $acquaintances(G,\ a) \setminus AppearsFaulty(G) \subseteq S$
  $\land \forall\, a \in S$:
    $watchers(G,\ a) \setminus AppearsFaulty(G) \subseteq S$

$unmarked(G) \triangleq$ (DOMAIN G) $\setminus$ $marked(G)$

MODEL

Alone, shadow graphs characterize the garbage in the Monitors model. To find garbage in the Exile model, we need undo logs.

$Init \triangleq M!Init$
$Next \triangleq M!Next$

PROPERTIES

$TypeOK \triangleq \forall\, a \in Shadows : shadows[a] \in Shadow$

$Spec \triangleq unmarked(shadows) = M!AppearsQuiescent$

A.8 The UndoLogs Model

MODULE *UndoLogs*

EXTENDS *Common*, *Integers*, *FiniteSets*, *Bags*, *TLC*

CONSTANT *NodeID*
VARIABLE *location*, *ingress*, *ingressSnapshots*, *droppedMsgs*

$D \triangleq$ INSTANCE *Dynamic*
$M \triangleq$ INSTANCE *Monitors*
$E \triangleq$ INSTANCE *Exile*

$S \triangleq \text{INSTANCE } Shadows$

UNDO LOGS

An undo log for node N indicates how to recover from the exile of node N.

$$
\begin{aligned}
&UndoLog \triangleq [\\
&\quad node : NodeID,\\
&\quad undeliverableMsgs : [ActorName \to Int],\\
&\quad undeliverableRefs \ : [ActorName \times ActorName \to Int]\\
&]
\end{aligned}
$$

$$snapshotsFrom(N) \triangleq \{a \in Snapshots : location[a] = N\}$$

The number of messages sent to actor *b* by actors on node N, according to the collage.

$$sent(N,\ b) \triangleq sum([a \in snapshotsFrom(N) \mapsto snapshots[a].sent[b]])$$

The number of references owned by *a* pointing to *b* created by node N, according to the collage.

$$created(N,\ a,\ b) \triangleq sum([c \in snapshotsFrom(N) \mapsto snapshots[c].created[a,\ b]])$$

The number of messages sent to *b* originating from N1 that have been admitted to their destination, according to the ingress actors' snapshots.

$$
\begin{aligned}
&admittedMsgs(N1,\ b) \triangleq\\
&\quad \text{LET } N2 \triangleq location[b] \text{ IN}\\
&\quad \text{IF } \langle N1,\ N2\rangle \in \text{DOMAIN } ingressSnapshots \text{ THEN}\\
&\qquad ingressSnapshots[N1,\ N2].admittedMsgs[b]\\
&\quad \text{ELSE } 0
\end{aligned}
$$

The number of references owned by *a* pointing to *b* created by N1 that have been admitted to their destination, according to the ingress actors' snapshots.

$$
\begin{aligned}
&admittedRefs(N1,\ b,\ c) \triangleq\\
&\quad \text{LET } N2 \triangleq location[b] \text{ IN}\\
&\quad \text{IF } \langle N1,\ N2\rangle \in \text{DOMAIN } ingressSnapshots \text{ THEN}\\
&\qquad ingressSnapshots[N1,\ N2].admittedRefs[b,\ c]\\
&\quad \text{ELSE } 0
\end{aligned}
$$

The undo logs for each node N.

$$
\begin{aligned}
&undo \triangleq\\
&\quad [N \in NodeID \mapsto\\
&\qquad [
\end{aligned}
$$

```
        node ↦ N,
        undeliverableMsgs ↦
          [b ∈ ActorName ↦
            sent(N, b) − admittedMsgs(N, b)],
        undeliverableRefs ↦
          [⟨b, c⟩ ∈ ActorName × ActorName ↦
            created(N, b, c) − admittedRefs(N, b, c)]
      ]
  ]

undeliverableMsgs ≜
  [b ∈ ActorName ↦
    sum([N ∈ E!ApparentlyExiledNodes ↦ undo[N].undeliverableMsgs[b]])
  ]

undeliverableRefs ≜
  [⟨b, c⟩ ∈ ActorName × ActorName ↦
    sum([N ∈ E!ApparentlyExiledNodes ↦ undo[N].undeliverableRefs[b, c]])
  ]

AmendedShadows ≜
  {a ∈ ActorName :
    ∧ a ∈ S!Shadows
    ∧ (a ∉ E!ApparentlyExiledActors ∨ S!shadows[a].watchers ≠ {})
  }
```

The shadow graph, amended using finalized undo logs.

```
amendedShadows ≜ [b ∈ AmendedShadows ↦ [
  interned   ↦ S!shadows[b].interned,
  sticky     ↦ S!shadows[b].sticky,
  watchers   ↦ S!shadows[b].watchers \ E!ApparentlyExiledActors,
  status     ↦ IF b ∈ E!ApparentlyExiledActors THEN "halted" ELSE S!shadows[b].status,
  undelivered ↦ S!shadows[b].undelivered − undeliverableMsgs[b],
  references ↦ [c ∈ ActorName ↦ S!shadows[b].references[c] − undeliverableRefs[b, c]]
]]
```

MODEL

$Init \triangleq E!Init$
$Next \triangleq E!Next$

PROPERTIES

$TypeOK \triangleq$
$\quad \land \quad undo \in [NodeID \to UndoLog]$
$\quad \land \quad \forall\, a \quad \in AmendedShadows : amendedShadows[a] \in S!Shadow$

$Spec \triangleq$
$S!unmarked(amendedShadows) \setminus E!ApparentlyExiledActors =$
$E!AppearsQuiescent \setminus E!ApparentlyExiledActors$