DISTRIBUTED GARBAGE COLLECTION FOR LARGE-SCALE MOBILE ACTOR SYSTEMS

By

Wei-Jen Wang

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

Carlos A. Varela, Thesis Adviser

David Bacon, Member

Ana Milanova, Member

David Musser, Member

Boleslaw Szymanski, Member

Rensselaer Polytechnic Institute
Troy, New York

December 2006
(For Graduation Dec 2006)

© Copyright 2006
by
Wei-Jen Wang
All Rights Reserved

CONTENTS

LIST OF TABLES v

LIST OF FIGURES vi

ABSTRACT ix

1. Introduction 1
   - 1.1 Large-Scale Distributed Computing 1
   - 1.2 Garbage Collection (GC) 4
   - 1.3 Definition of Active and Passive Actor Garbage 5
   - 1.4 Distributed Mobile Actor Garbage Collection 7
   - 1.5 Problem Description 9
   - 1.6 Thesis Contributions 10
   - 1.7 Roadmap 11

2. Related Work 12
   - 2.1 Passive Object Garbage Collection for Shared Memory Systems . . . 12
     - 2.1.1 Sequential Garbage Collection Techniques 12
     - 2.1.2 Concurrent and Incremental Garbage Collection Algorithms . 14
   - 2.2 Distributed Passive Object Garbage Collection Algorithms 17
     - 2.2.1 Reference Counting 17
     - 2.2.2 Complete Distributed Tracing 21
     - 2.2.3 Server-Based Tracing 22
     - 2.2.4 Heuristic Migration and Tracing 22
     - 2.2.5 Group Tracing 25
     - 2.2.6 Distributed Generational Garbage Collection 25
     - 2.2.7 Snapshot-Based Tracing 26
   - 2.3 Equivalence of Passive Object Garbage Collection and Distributed Termination Detection 26
   - 2.4 Actor Garbage Collection 28
     - 2.4.1 Actor Marking Algorithms 28
     - 2.4.2 Distributed Actor Garbage Collection 30

3. Equivalence of Actor Garbage Collection and Passive Object Garbage Collection . . . 33
3.1 Transformation Methods . . . 33
3.1.1 Garbage in Passive Object Systems . . . 33
3.1.2 Garbage in Actor Systems . . . 34
3.1.3 Problem Equivalence . . . 36
3.1.4 Transformation by Direct Back Pointers to Unblocked Actors . 37
3.1.5 Transformation by Indirect Back Pointers to Unblocked Actors 37
3.2 Implementation of the Transformation Methods . . . 39
3.2.1 The Back Pointer Algorithm . . . 39
3.2.2 The N-Color Algorithm . . . 40
3.2.3 Complexity Analysis . . . 43
3.3 Proofs . . . 43
4. A Distributed Actor Garbage Collection Mechanism for Mobile Actor Systems . . . 47
4.1 The Pseudo-Root Approach . . . 47
4.1.1 The Live Unblocked Actor Principle . . . 47
4.1.2 Pseudo-Root Actor Garbage Collection . . . 48
4.1.3 Imprecise Inverse Reference Listing . . . 50
4.1.4 Actor Garbage Collection without the Live Unblocked Actor Principle . . . 51
4.1.5 Implementation of the Pseudo-Root Approach . . . 51
4.2 A Non-Intrusive Distributed Snapshot Algorithm for Mobile Actor Garbage Collection . . . 54
4.2.1 Causal Consistency . . . 55
4.2.2 Non-Intrusive Distributed Snapshot . . . 56
4.2.3 Discussion on Correctness . . . 59
5. Correctness of the Pseudo-Root Approach . . . 63
5.1 The Computing Model of the Pseudo-Root Approach . . . 63
5.2 Safety and Liveness Properties of the Pseudo-Root Approach . . . 67
5.3 The Pseudo-Root Approach Properties in a Distributed Environment 74
5.4 Actor Garbage Collection without the Live Unblocked Actor Principle 75

6. Correctness of the Distributed Snapshot Algorithm 78
6.1 The Computing Model of the Snapshot Algorithm 78
6.2 Safety 81
6.2.1 Local State Logging 81
6.2.2 Global Snapshot 82
6.3 Liveness 87
6.4 Proofs 88
6.4.1 Local State Logging Properties 88
6.4.2 Global Snapshot Algorithm Properties 91
7. Experimental Results 97
7.1 Actor Garbage Collection in SALSA 97
7.2 Actor GC Overhead with Respect to Application Tests 98
7.3 Overhead Breakdown 100
7.3.1 Overhead Breakdown of Local Actor GC in a Uniprocessor Environment 100
7.3.2 Overhead Breakdown of Local Actor GC in a Concurrent Environment 102
7.3.3 Overhead Breakdown of Actor GC in a Cluster Environment . 103
7.4 Scalability Test 104
7.4.1 The Maximum Likelihood Evaluation (MLE) Fitter 104
7.4.2 Results 106
8. Conclusions and Future Work 109
8.1 Conclusions 109
8.2 Future Work 110
8.2.1 Resource Access Restrictions and Security Policies 110
8.2.2 Large-Scale Applicability 111
8.2.3 Extension of the Distributed Snapshot Algorithm 111
8.2.4 Using Static Analysis for Actor Garbage Collection 113
LITERATURE CITED 114

LIST OF TABLES

7.1 Application performance on a dual-core processor Sun Blade 1000s machine (measured in seconds). . . . 100

7.2 Application performance in a distributed environment. Real time is measured in seconds. . . . 100

7.3 Actor creation (ms) and its overhead (%) in a uniprocessor environment. 101

7.4 Message passing (μs) and its overhead (%) in a uniprocessor environment. 101

7.5 Reference passing (ms) and its overhead (%) in a uniprocessor environment. . . . 101

7.6 Multiplication-division (Secs) and its overhead (%) in a uniprocessor environment, where each actor performs several loops, each of which contains a double-precision multiplication operation and a double-precision division operation. . . . 102

7.7 Actor creation (ms) and its overhead (%) in a quad-core processor environment. . . . 103

7.8 Message passing (μs) and its overhead (%) in a quad-core processor environment. . . . 103

7.9 Reference passing (ms) and its overhead (%) in a quad-core processor environment. . . . 103

7.10 Actor migration (ms) and its overhead (%) in a distributed environment. 103

7.11 Message passing (ms) and its overhead (%) in a distributed environment. 104

7.12 Reference passing (ms) and its overhead (%) in a distributed environment. 104

LIST OF FIGURES

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

1.2 Actors 3, 4, and 8 are live because they can potentially send messages to the root. Objects 3, 4, and 8 are garbage because they are not reachable from the root. 7

2.1 A comparison on distributed acyclic garbage collection algorithms. The figure explains how the reference sender process passes a reference of Object B to the target process (reference copy), and how a process deletes a reference of Object B and then notifies the object owner process (reference deletion). 19

3.1 An example of transformation by direct back pointers to unblocked actors and root actors. 37

3.2 An example of transformation by indirect back pointers to unblocked actors and root actors. 38

3.3 The back pointer algorithm. 40

3.4 An example of the back pointer algorithm where W stands for *White*, G stands for *Gray*, and B for *Black*. Only actors marked by Color B are live. 41

3.5 The N-color algorithm. 42

3.6 An example for the N-color algorithm. Actors marked by Colors 0, 2, or 3 are live. Actors colored -1 and 1 are garbage. 43

4.1 The left side of the figure shows a possible race condition of mutation and message passing. The right side of the figure illustrates both kinds of sender pseudo-root actors. 49

4.2 An example of pseudo-root actor garbage collection which maps the real state of the given system to a pseudo-root actor reference graph. 50

4.3 The actor garbage detection sub-protocols. 52

4.4 Time lines to illustrate late and early messages. At the left side, Actor a sends a message to Actor b at t_1 and then its state is recorded at t_a ($t_a > t_1$); the state of Actor b is recorded at t_b and then it receives the message at t_2 ($t_2 > t_b$). At the right side, the state of Actor a is recorded at t_a and then it sends a message to Actor b at t_1 ($t_1 > t_a$); Actor b receives it at t_2 and then its state is recorded at t_b ($t_b > t_2$). . . 55

4.5 An example of local state logging. The upper part demonstrates the actor reference graph in the real world, while the lower part illustrates how local state logging works. 58

4.6 The distributed snapshot algorithm. A meaningful global snapshot consists of the local snapshots of the computing nodes that reply 'OK'. . . 59

4.7 The local snapshot actor. 60

4.8 Nine possible migration detection scenarios. 61

5.1 Lemma 5.2.7: safe imprecise inverse references. 71

5.2 Theorem 5.3.3: one-step back-tracing safety. 75

5.3 Theorem 5.4.1: one-step back-tracing and forward validating safety. . . 76

6.1 Different phases of global synchronization. 83

6.2 The relationship of mutation operations, snapshots, and the snapshot-composition operation. There are two actor configuration sets in the figure — one is $\{S_{s,i} \mid m \geq i \geq 1\}$ at time t_x, and the other is $\{S_{e,i} \mid m \geq i \geq 1\}$ at time t_e, where $S_{s,i} \rightarrow^* S_{e,i}$ and t_x and t_e are defined in Figure 6.1 as time points. $S_S = (S_{s,1} \parallel S_{s,2} \parallel ... \parallel S_{s,m})$, and $S_E = (S_{e,1} \parallel S_{e,2} \parallel ... \parallel S_{e,m})$. 84

6.3 Proposition 6.2.10: consistency guarantee of remote references and inverse references. 85

6.4 Sub-case 2 of the induction step of the proof for Lemma 6.2.1. 90

6.5 Two different possible global actor configurations where $\exists S_k, S_m : S_{s,1} \rightarrow^* S_k \xrightarrow{a.MI} S_m \rightarrow^* S_{e,1}$. 93

7.1 Execution time per fit function call vs. the total number of processors. Four kinds of mechanisms are used to evaluate the implementation of our actor garbage collection algorithms. 107

7.2 Breakdown of the actor garbage collection mechanism. 108

ABSTRACT

Distributed actor garbage collection (GC) is a notoriously hard problem due to the nature of distributed actor systems — no common clock, no coherent global information, asynchronous and unordered message passing, autonomous behavior of actors, and counter-intuitive actor marking to identify live actors. Most existing distributed actor GC algorithms rely on First-In-First-Out (FIFO) communication, which constrains the actor model and is impractical with actor mobility; others depend on stop-the-world synchronization, which is intrusive and impractical for users' computations. Existing actor GC algorithms ignore actor mobility and resource access restrictions. Existing distributed passive object GC algorithms cannot be directly reused because of the different semantics of passive objects and actors.

To overcome the problems that existing algorithms cannot solve, this thesis presents a practical actor GC mechanism for distributed mobile actor systems. Our approach starts formalizing garbage actors, and then we show two different but similar transformation methods that prove the equivalence of actor GC and passive object GC. Two actor marking algorithms are derived from the transformation methods — *the back pointer algorithm* and *the N-color algorithm.* The back pointer algorithm has linear time complexity of $O(E+V)$ and extra space complexity of $O(E+V)$, and the N-color algorithm has time complexity of $O(E \lg^* M)$ and extra space complexity of $O(M)$, given that E is the number of references, V, the number of actors, and M, the number of unblocked and root actors. The N-color algorithm only requires scanning the reference graph once while the back pointer algorithm requires scanning it twice.

The thesis follows by describing our distributed mobile actor GC mechanism. It consists of 1) an asynchronous, non-FIFO reference listing based algorithm which supports hierarchical GC (local and global GC), 2) a new fault-tolerant, distributed snapshot-based algorithm which collects cyclic and acyclic garbage in a partial set of computing hosts, and 3) formal models and correctness proofs. Experimental results have confirmed that our approach is practical and scalable.

CHAPTER 1
Introduction

This chapter describes fundamental aspects of large-scale distributed computing, garbage collection, and distributed actor garbage collection. A short description of the problem to attack and the roadmap of the thesis are also included.

1.1 Large-Scale Distributed Computing

Distributed computing is a promising field: using a collection of distributed computing resources is more cost-effective than using a single powerful supercomputer. Performance of applications can be improved by concurrently running several processes in a coordinated way. Modern computing and networking resources have improved dramatically in the past decades, and thus the size of distributed computing has increased into a world-wide scale. However, scientists eventually realize that managing or developing distributed applications is not easy because of: 1) heterogeneous resources, 2) possible failures of resources and networking, 3) lack of shared memory and a global clock, and 4) races among processes. An emerging requirement of computing element mobility makes it more difficult — run-time reconfiguration of the distributed system [62] can improve overall performance of applications but complicates the design of resource management.

To overcome these rising problems of distributed computing, several object-oriented computation models are introduced such as CORBA [74], DCOM [32], and JAVA RMI [92]. These models make heterogeneous resources approachable for application development and run-time access. On the other hand, several formalized concurrent models keep being proposed since 1970s, such as CCS and π-Calculus by Milner [68, 82], the Actor Model of Computation by Hewitt [44, 2], Join-Calculus by Fournet and Gonthier [37], and Mobile Ambients by Cardelli and Gordon [19]. Different models may introduce different requirements of resource management because they support different semantics for computing. For example, systems based on π-calculus may need *distributed termination detection*, object-oriented systems

may require *passive object garbage collection*, and actor systems may demand *actor garbage collection*.

Nowadays the performance-cost ratio of both networking and computing resources keeps improving. These positive factors trigger the idea of large-scale computing, which consists of thousands or even millions of distributed resources all over the world. Unfortunately, the larger the computing scale is, the more difficult the management and application development becomes. To reduce the efforts for application developers and users, a friendly large-scale computing framework is absolutely required, in which they can access computing power and resources simply and transparently, without having to consider where the resources are and how the computation takes place — such as SETI@Home [91] and Folding@Home [112].

Grid computing [34] is one of the studies that tries to solve the large-scale computing problems by introducing the idea of virtual organizations [36], where resources are protected and accessed transparently by trusted groups of users and applications. Distributed system frameworks, such as Globus [34], Legion [43], and IOS [62], are enabling technologies to develop grid computing systems. For instance, a Globus-based system may follow the Open Grid Services Architecture (OGSA) [35], which uses an XML-based standards to support transparency and layered services for composability. Globus is an application toolkit, while Legion and IOS are object- and actor-based distributed virtual operating systems running on virtual machines [42].

An actor system is comprised of uniquely named, autonomous reactive objects, namely the *actors*. Communication of actors is *purely asynchronous, guaranteed, and fair* — they always send messages in a *non-blocking* manner. Even though messages may arrive unordered, they are eventually delivered. Furthermore, an actor can either send messages to its *acquaintances*, to whom it has explicit references, or some predefined special actors such as the output service. An actor consists of an encapsulated thread of control, its internal state, and a message box to buffer incoming messages. An actor is *unblocked* if it is processing a message or has messages in its message box; otherwise it is *blocked* waiting for incoming messages. An actor buffers incoming messages in its message box, and its thread of control keeps retrieving and

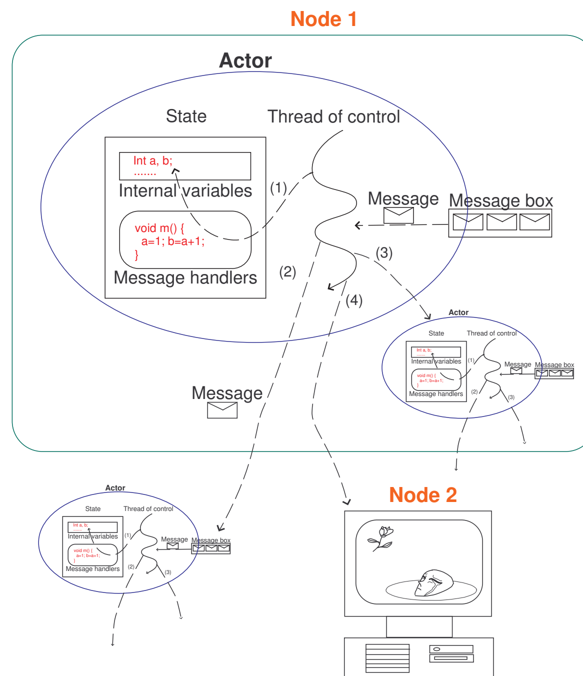

Figure 1.1: An actor is a reactive entity which communicates with others by asynchronous messages in a non-blocking manner. In response to an incoming message, it can use its thread of control to 1) modify its internal state, 2) send messages to other actors, 3) create actors, or 4) migrate to another computing node.

processing them. In response to an incoming message, an actor can use its thread of control to modify its encapsulated internal state, send messages to other actors, create actors, or migrate to another computing node (see Figure 1.1).

The actor model of computation provides natural semantics for distributed systems because it has several preferable features: purely asynchronous communication (non-blocking and unordered communication), and state encapsulation with an internal thread. With these features, a distributed system can be easily reconfigured at runtime — migration of an actor is as easy as migration of its encapsulated state and its message box, without worrying about state corruption. Compared to the synchronous *object method invocation model* (or the *Remote Procedure Call model*), the actor model is more concurrent because actors do not block for any

return value. Additionally, state encapsulation facilitates dynamic load balancing [62] by reallocating the actors during computation. Many programming languages have partial or total support for actor semantics, such as SALSA [100], ABCL [110], THAL [54], and Erlang [5]. Some libraries also support the actor model of computation, such as Actor Foundry [75] and ProActive [7] for Java, Broadway [90] for C++, IOS for MPI library [33], and Actalk [16] for Smalltalk.

1.2 Garbage Collection (GC)

Garbage collection (GC) is defined as a mechanism to reclaim memory space. The problem of garbage collection caught people's attention since the late 1950s when high-level programming languages became more and more popular. These high-level programming languages provide dynamic data structures, such as linked lists. As a consequence, the size of objects might not be fixed at compile time. Therefore, a mechanism to reclaim the memory space of garbage objects is required. Manual garbage collection can solve this problem if an application does not require a lot of dynamic memory allocation operations. As the size of the application becomes larger and more complex, automatic garbage collection becomes preferable. There are two reasons. Firstly, manual garbage collection is error-prone and thus causes memory security issues. Secondly, manual garbage collection cuts against high-level programming. From the perspective of software engineering, people should focus on the development of functionalities, not on irrelevant concerns. Currently, garbage collection usually refers to *automatic garbage collection*.

Objects can be either *passive* or *active*. Active objects relate to actors, and we use actors to refer to them. One major difference between actors and passive objects is the thread of control. A passive object is operated by *external threads*, which can create new objects, add new references, or delete references. If an object can be possibly manipulated by the external threads of control, it is live; otherwise it is garbage. On the other hand, an actor has an internal thread, which can only create new actors but cannot directly access other actors including the actors it created. Both kinds of objects can become garbage, and require a garbage collection mechanism to reclaim them. Without garbage collection, a system that supports

dynamic object creation may crash because the memory is eventually full of garbage. The actor system could be worse because garbage actors could consume computing power. The requirements of a garbage collection mechanism are listed below:

- Liveness: All garbage is collected eventually.
- Safety: No live objects can be collected.
- Efficiency: The overhead of garbage collection must be acceptable.

1.3 Definition of Active and Passive Actor Garbage

The definition of garbage actors relates to the idea of whether an actor is doing meaningful computation. Meaningful computation is defined as having the ability to communicate with any of the root actors. By assuming that *every actor has a reference to itself*, an actor is live if it can either possibly: 1) receive messages from the root actors or 2) send messages to the root actors. The ability of sending messages depends upon the state of an actor — blocked or unblocked. Only unblocked actors can send messages. The set of garbage actors is then defined as the complement of the set of live actors. To formally describe actor garbage collection, we introduce the following definitions:

- **Blocked actor**: An actor is blocked if it has no pending messages in its message box, nor any message being processed. Otherwise it is *unblocked*.
- **Reference**: A reference indicates an address of an actor. Actor a_p can only send messages to Actor a_q if a_p has a reference pointing to a_q, denoted as $\overline{a_p a_q}$.
- **Inverse reference**: An inverse reference is a conceptual reference in the counter-direction of an existing reference.
- **Acquaintance**: Let Actor a_p have a reference pointing to Actor a_q. a_q is an acquaintance of a_p, and a_p is an *inverse acquaintance* of a_q.
- **Root actor**: An actor is a root actor if it encapsulates a resource, or if it is a public service — such as I/O devices, web services, and databases.

The original definition of live actors is denotational because it uses the concept of "potential" message delivery and reception. To make it more operational, we assume instant message delivery and use the term "*potentially live*" [28] to define live actors.

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

With the concept of potentially live actors, we introduce the definition of *active garbage*, which refers to the set of actors which is both potentially live and garbage. Similarly, *passive garbage* refers to the set of actors which is not potentially live and is garbage. The difference between active garbage and passive garbage is that active garbage can change the actor reference graph. For instance, it can create/delete references, and send messages. Garbage can also be classified as *cyclic garbage* and *acyclic garbage*, depending on the referential relationship among actors. Without considering the self reference to a garbage computing element (object/actor), the garbage computing element is said to be cyclic garbage if it can follow references to come back to itself. Otherwise it is acyclic garbage.

Acyclic passive garbage can be identified by a count for incoming references. A zero count indicates that no one is referencing the computing element. If the computing element is blocked or naturally passive (such as passive objects), then it can be reclaimed safely. Algorithms based on this idea are called reference counting/listing algorithms. However, the algorithms cannot be used for cyclic garbage and active garbage, which requires a consistent local/global state to be identified.

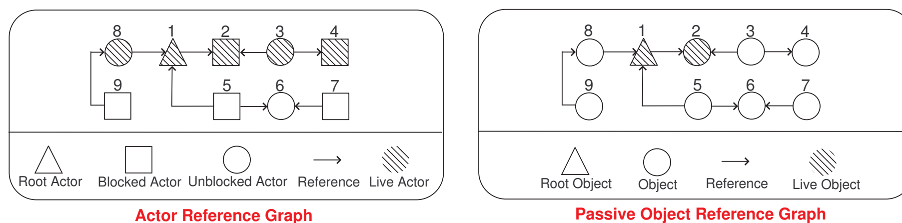

Figure 1.2: Actors 3, 4, and 8 are live because they can potentially send messages to the root. Objects 3, 4, and 8 are garbage because they are not reachable from the root.

1.4 Distributed Mobile Actor Garbage Collection

Uniprocessor and distributed garbage collection have been developed for decades. However, literature for *actor garbage collection* is scarce. The problem of actor garbage collection is different in nature from passive object garbage collection. Actor computations must be able to directly or indirectly provide results to a set of predefined output devices, such as consoles, printers, file systems, and databases. Actors may also obtain information from input devices, such as keyboards or sensors, to output results if they can communicate with them. As a result, the input or output devices are represented by a *root set of actors*. Actors that can directly or indirectly communicate with the root set of actors are live. Figure 1.2 illustrates a key difference between actor garbage collection and passive object garbage collection.

In actor-oriented programming languages, an actor must be able to access resources which are encapsulated in service actors. To access a resource, an actor requires its reference. This implies that actors keep persistent references to some special service actors — such as the file system service and the standard output service. Furthermore, an actor can explicitly create references to public services. For instance, an actor can dynamically convert a string into a reference to communicate with a service actor, analogous to accessing a web service by a web browser using a URL. However, the resource access assumption is not always true, such as in sandbox computing environments.

Previous distributed GC algorithms (including actor GC algorithms) rely on First-In-First-Out (FIFO) communication which simplifies detection of a consistent

global state. A distributed object GC algorithm either adopts: 1) a lightweight reference counting/listing approach which cannot collect distributed mutually referenced data structures (*cycles*), 2) a trace-based approach which requires a consistent state of a distributed system, or 3) a hybrid approach.

The concept of *migrating* actors complicates the design of actor communication — locality of actors can change, which means even simulated FIFO communication with message redelivery is impractical, or at least limits concurrency by unnecessarily waiting for message redelivery. FIFO communication is an assumption of existing distributed GC algorithms. For instance, distributed reference counting algorithms demand FIFO communication to ensure that a reference-deletion system message does not precede any application messages.

Global state detection is required in distributed actor garbage collection. There are two common approaches to use [1] — the *stop-the-world* approach which simply stops everything for garbage collection, and the *snapshot* approach which detects a causally consistent snapshot for garbage collection. Snapshot-based algorithms greatly fit in the need of actor systems because they have less interference with applications than the stop-the-world algorithms. However, when a snapshot-based algorithm is applied to a mobile actor system, it encounters difficulties detecting migrating actors because these actors are being transmitted over the network and are therefore hard to detect. The end result is that global snapshots may be missing actors or may contain duplicate actors; both views are inconsistent with the global system state. In-transit messages cause three additional difficulties to detect actor garbage in a distributed environment. Firstly, in-transit messages may carry references that can affect the global state. Secondly, a reference to an actor can be deleted while a message to it is in transit, because actor communication is non-blocking. This implies that there may exist a blocked actor such that nobody knows about it but it is live. In other words, following references to detect garbage is not reliable in actor systems. Thirdly, actors communicate with each other by asynchronous and non-First-In-First-Out (non-FIFO) messages, which violates preconditions of most existing garbage collection algorithms and snapshot algorithms.

[1]A modified Dijkstra's on-the-fly algorithm [31] could be a third possible solution for global actor garbage collection.

1.5 Problem Description

The purpose of this research is to design and to implement a practical distributed actor garbage collection mechanism for large-scale, mobile actor systems. This research is based on the SALSA programming language [100] and the World-Wide Computer (WWC) middleware [99].

We assume the distributed computing environment consists of numerous computing nodes, and the computing nodes are connected by a network. Applications running on the distributed computing environment follow the assumptions of the actor model of computation. Actors are the basic computing elements of applications, and always communicate by sending guaranteed and unordered asynchronous messages. The guaranteed message delivery is handled not by the garbage collection mechanism, but by the communication middleware of the WWC. In other words, hard failures are not considered. Although message delivery is unordered, partial message execution order can be expressed with *execution continuations*. This is a powerful feature provided by the SALSA programming language, and it does not violate the assumptions of the actor model. Each actor is assumed to keep references to some local roots — such as the standard input/output and the file system. However, the references to the roots may not exist in sandbox computing hosts.

Our distributed actor garbage collection approach consists of three major subcomponents:

1. Formalization of the actor garbage problem and the actor marking algorithms: This thesis provides a formal definition of the actor garbage collection problem, transformations from actor garbage collection into passive object garbage collection, and new actor marking algorithms.

2. A non-blocking, non-FIFO reference listing based algorithm: *the pseudo-root approach*. The algorithm keeps track of inverse references to support hierarchical garbage collection — independent collection of local and global garbage. Actor migration is supported as well. It can also identify distributed acyclic passive garbage. Furthermore, it considers in-transit messages in the reference graph and thus simplifies the problem of consistent global state detection.

3. A distributed snapshot-based actor garbage collection algorithm: The algorithm does not require First-In-First-Out or blocking communication, nor message logging. Furthermore, actor migration is allowed while capturing global snapshots and partial snapshots can be safely used to collect garbage, therefore not requiring comprehensive cooperation among all computing nodes.

1.6 Thesis Contributions

The thesis describes a comprehensive approach for distributed mobile actor garbage collection. The main contributions of the thesis are listed as follows:

1. The thesis provides a formal definition of the actor garbage collection problem.

2. The thesis provides two new transformation methods from actor garbage collection into passive object garbage collection and two corresponding new actor marking algorithms. Assume E is the number of references, V is the number of actors, and M is the number of unblocked and root actors. One of the marking algorithms only scans the reference graph twice, and it has linear time complexity of $O(E+V)$ and extra space complexity of $O(E+V)$; the other only scans the reference graph once, and it has time complexity of $O(E \lg^* M)$ and extra space complexity of $O(M)$.

3. The pseudo-root approach supports systems with asynchronous, non-FIFO communication and maps in-transit messages and references into the actor reference graph, making it unique in the family of reference counting/listing algorithms. Furthermore, it supports actor migration. Thus it is not intrusive and simplifies global state detection for actor garbage collection [105].

4. The thesis presents a non-intrusive distributed snapshot algorithm for actor garbage collection. Migration and all mutation operations are allowed while taking a snapshot. Consequently, it does not require comprehensive cooperation of all computing hosts in the distributed system. It can support multi-level hierarchical distributed actor garbage collection [106].

5. The thesis provides formal proofs of correctness.

6. The actor garbage collection algorithms are implemented and form an integral part of the SALSA programming language [109], which is used to develop grid/pervasive computing applications [104].

1.7 Roadmap

The remainder of the thesis is organized as follows: Chapter 2 reviews a set of important papers for both passive object garbage collection and actor garbage collection. Chapter 3 discusses the problem of actor garbage collection from the perspective of the graph problem. It describes the relationship among actor garbage collection and passive object garbage collection, shows two light-weight transformation methods to transform actor garbage collection to passive object garbage collection, and provides two corresponding novel actor marking algorithms. Chapter 4 shows how we handle mobile actor garbage collection in a distributed environment in a non-intrusive manner, where communication of both applications and systems is asynchronous and non-FIFO. Two major algorithms are provided — the pseudo-root approach and the distributed snapshot algorithm. Chapter 5 and Chapter 6 define formal computing models for the pseudo-root approach and the distributed snapshot algorithm respectively, and they also provide proofs of the safety and liveness properties. Chapter 7 shows some experimental results of our implementation. Chapter 8 contains concluding remarks and future work directions.

People who are interested in the definition and properties of actor garbage collection should read Chapters 1, 2, and 3. For those who are interested in the concept of distributed mobile actor garbage collection, Chapters 1, 2, 4, and 7 are recommended. Formal models and proofs of properties of Chapter 4 can be found in Chapters 5 and 6.

CHAPTER 2
Related Work

This chapter discusses local passive object garbage collectors, acyclic passive object garbage detection protocols, distributed passive object garbage collection algorithms, and actor garbage collection algorithms. Good surveys for passive object garbage collection can be found in [48], [108], and [1].

2.1 Passive Object Garbage Collection for Shared Memory Systems

Passive object garbage collection was originated from sequential environments. Many techniques of sequential passive object garbage collection can apply to concurrent and distributed environments, and much related work reuses them in concurrent and distributed environments. Passive object garbage collection for shared memory systems includes: 1) sequential garbage collection and 2) concurrent and incremental garbage collection. This field has been developed for a long time. In the past, research on garbage collection was in the realm of functional programming languages, such as Lisp. Right now even imperative programming languages such as C, C++ [10] and Java [41] support garbage collection.

2.1.1 Sequential Garbage Collection Techniques

Techniques for sequential garbage collection can be roughly divided into four categories: 1) reference counting algorithms, 2) mark-and-sweep algorithms, 3) copying garbage collection algorithms, and 4) generational algorithms. Concepts of these techniques are widely used in concurrent and distributed systems.

Reference Counting Algorithms

Reference counting algorithms were used in early programming languages and current file system implementations. Reference counting algorithms, also referred to as *direct garbage collection algorithms*, are based on counting the number of in-

verse acquaintances (or incoming references) for each object. Such an object can be deleted safely if it has a zero count because the mutator (the thread of control of a user's application) cannot reach it from the roots. The advantages of reference counting algorithms are traditionally considered to be: 1) simplicity of implementation, and 2) low execution overhead of garbage collection. The drawback of reference counting is that it cannot collect cyclic garbage. For instance, two garbage objects can have references to each other, making each of them have a count of one. Such cyclic garbage cannot be reclaimed by naïve reference counting algorithms. There are several complex reference counting algorithms that detect cyclic garbage, but they either depend on special functional programming language patterns [39], strong assumptions of reference information, such as [18, 81], or expensive space and execution time requirements for extra traversal of the reference graph, such as [22], [60], and [6].

Mark-And-Sweep Algorithms

Mark-and-sweep algorithms are either based on breadth-first-search or depth-first-search algorithms, traversing and marking objects from the roots in order to identify live objects. Compared to reference counting algorithms, a naïve mark-and-sweep algorithm has an advantage of low space overhead because it requires only one extra bit for marking. Another advantage is that it can collect cyclic garbage naturally. A disadvantage is that it requires long pause times to do garbage collection, which is not practical for many systems such as real-time systems, interactive applications, and concurrent/distributed systems.

Copying Garbage Collection Algorithms

The concept of *copying* garbage collection algorithms is similar to that of mark-and- sweep algorithms. Conceptually, a copying garbage collection algorithm divides the memory storage into several regions. Every time the garbage collector is invoked, it traces objects from roots, and moves them into another region. When the traversal finishes, objects in the original region can be deleted safely. A typical example is the *semi-space* algorithm [21, 8]. This algorithm requires two predefined regions, *Fromspace* and *Tospace*. *Fromspace* is the region where objects reside and

Tospace is empty at the beginning. Iterative copying starts from the roots and follows the references to preserve every live object in *Tospace*. If no live object can be found, the algorithm terminates. The advantage of copying garbage collection algorithms is that they can compact the memory layout. The drawback of these algorithms is that object copying has a performance penalty.

Generational Garbage Collection Algorithms

Generational garbage collection algorithms are based on the assumption that young objects tend to become garbage young [96], with a late amendment that the youngest tends to be live [88, 87]. This kind of algorithm is widely used in current garbage collection. With the object life time assumption, memory space can be partitioned into several regions, each of which represents a different generation. When the garbage collector is invoked, only one region is scanned. A generational garbage collection algorithm can be either mark-and-sweep based or copying-based, but most of them are copying-based. To avoid cross-generation garbage collection, every long-lived surviving object must be eventually moved to the oldest region. Generational garbage collection algorithms are complex. Issues of generational garbage collection include: 1) how to determine the age of an object, 2) how to handle inter-generation references, 3) strategies of moving surviving objects after garbage collection. Famous examples include *Mature Object Spaces (MOS)* [46] (referred to as the *train* algorithm), *Appel's generational* collector [3], and *Moon's Ephemeral* Collector [70] with hardware garbage collection support. Generational garbage collection algorithms are widely used in business products. They have short pause times for garbage collection because only a small set of objects is scanned (*incrementality*). This feature makes them practical for real-time systems and user interactive applications.

2.1.2 Concurrent and Incremental Garbage Collection Algorithms

For a multi-processor, shared memory machine, long pause times for stop-the-world garbage collection are unacceptable because they restrict concurrency. To avoid long pause times, garbage collection must be able to run in parallel with users' applications. The parallel execution introduces a coherence problem since mutators compete with the garbage collector for reference reading and modification.

For instance, a mutator can create a new reference in Object o_a to another new object, Object o_b, while every reference of Object o_a is just visited by a mark-and-sweep based garbage collector. This situation can lead to erroneously reclamation of Object o_b since Object o_b has no chance to be visited again. Every concurrent and incremental garbage collector must take care of coherence. When garbage collection is started, two strategies can be used to avoid such a coherence problem:

1. a reference cannot be added to an object, each of whose references has been visited by the garbage collector, or

2. the original reference to an unvisited object cannot be removed.

A practical garbage collection algorithm for concurrent, shared memory systems can be either concurrent or incremental. *Concurrent garbage collection* was first introduced by Steele's compactifying algorithm [49], and widely discussed since Dijkstra and his colleagues had published the on-the-fly algorithm [31]. The concept of incremental garbage collection was introduced by Baker [8] in the context of copying garbage collection. Generational garbage collection algorithms and reference counting based algorithms are considered as incremental garbage collection algorithms as well. Concurrent garbage collection requires an independent thread of control for the garbage collector, while incremental garbage collection distributes the work of garbage collection among all other mutators and only a small portion of garbage is collected to ensure that performance and availability of users' applications is not affected.

The Mostly Parallel Garbage Collection Algorithm

The goal of the *mostly parallel garbage collection* algorithm [15] is to shorten the pause times of a mark-and-sweep based algorithm, which can also be extended to a copying-based algorithm. The algorithm is designated to cooperate with the virtual memory system in order to check if an object is modified during garbage collection. The algorithm has six steps: 1) clear mark bits, 2) clear all virtual dirty bits, 3) trace and mark from roots, 4) stop the world, 5) trace and mark from all marked objects on dirty pages, and 6) restart the world. The algorithm is easy to

understand and implement. Variations of the collector are reported in many papers such as [9], and for the Java virtual machines of IBM [76] and SUN [78].

Yuasa's Snapshot-At-Beginning Algorithm

Yuasa's algorithm [111] is a mark-and-sweep based algorithm for the Kyoto Lisp programming language. The concept of the *snapshot-at-the-beginning* strategy is to preserve every reference at the beginning of garbage collection. The algorithm requires a special reference updating function for mutators. For instance, a reference to some object, Object o_a, is updated to point to another object, Object o_b, by the reference updating function during garbage collection. If Object o_a has not yet been visited by the garbage collector, Object o_a is then moved to a special marking stack to prevent erroneous reclamation. New objects allocated during garbage collection are also preserved. The snapshot-at-beginning algorithm is very conservative because it only collects the garbage which is produced before taking the snapshot.

The On-The-Fly Algorithm

The *on-the-fly* algorithm was proposed in [31], and is widely discussed, implemented, and formally proved. The on-the-fly algorithm is a mark-and-sweep based algorithm, that uses a tricolor abstraction to describe the states of objects during garbage collection. In this algorithm, an object is painted either *Black*, *Gray*, or *White*. Color *Black* indicates that an object has been completely visited — every reference of that object is visited. Color *Gray* means that an object is visited but not all of its references are visited. Color *White* is the initial color of objects at the beginning of garbage collection. When garbage collection finishes, only *White* and *Black* are left where *White* denotes garbage and *Black* denotes live objects. During garbage collection, the reference updating function paints a newly referenced *White* object *Gray*. Atomic reference updating is required for correctly performing this algorithm.

Steele's Algorithm

Steele's algorithm [49] is also mark-and-sweep based. It uses locks to implement the reference updating function of the mutator. The concept of the algorithm

is to recheck an object if any reference of the object is modified during garbage collection. Newly created objects are checked as well during the marking phase.

Baker's Incremental Copying Algorithm

Baker proposed an *incremental copying* algorithm [8] by modifying Cheney's copying algorithm [21]. The algorithm uses two regions, *Tospace* and *Fromspace*. Objects are copied from *Fromspace* to *Tospace* while doing garbage collection. Newly created objects are located in *Tospace* and marked visited, making them exempt from garbage collection. The algorithm uses a read barrier to coordinate the mutator and the collector, and the background collection is interleaved with object allocation. A similar strategy can be found in [4], which uses virtual memory primitives instead of heavy cooperation of mutators.

2.2 Distributed Passive Object Garbage Collection Algorithms

The computing model of distributed passive object systems consists of a collection of processes, also known as *spaces* or *nodes*. Each process has one to many threads of control (mutators). Every object must reside in a process, and can be remotely referenced by other objects. Some systems, such as Emerald [50, 51] and Jocaml [23], support object migration by encapsulating objects in a message. The communication model for users' applications is mostly synchronous, in particular based on remote procedure calls. However, the background communication, such as garbage collection among processes, can be asynchronous with the FIFO order assumption, or even completely unordered with incorporation of time-stamp-based message redelivery mechanism (simulated FIFO communication).

In this section, several types of distributed garbage collection algorithms are reviewed and categorized according to their features.

2.2.1 Reference Counting

Distributed reference counting (or *listing*) algorithms, also known as distributed reference counting protocols, are inherently incremental and thus they do not re-

quire complex synchronization for garbage collection. Typically, they require that either a referenced object has to maintain a counter, or a reference table has to keep track of every incoming/outgoing inter-node reference. Whenever an object is referenced, its counter has to increment by one (or creates an inverse reference in the reference table). One of their common disadvantages is that they cannot reclaim cyclic garbage (mutually referenced objects) unless a hybrid strategy is used. These reference counting algorithms are similar to our pseudo-root approach but they tend to be more synchronous — most of them rely on First-In-First-Out (FIFO) communication or time-stamp-based FIFO (simulated FIFO) communication at the application level, and some of them are even totally synchronous by using remote-procedure-call. Since actor communication is defined as asynchronous, unordered, and message-driven, these algorithms cannot be reused directly by actor systems. Figure 2.1 compares the algorithms which we are going to discuss in this subsection, in terms of reference copying and deletion.

The Lermen-Maurer Protocol

The *Lermen-Maurer* protocol (see Figure 2.1.a) [59] is the first distributed reference counting algorithm. It requires three messages per reference duplication, and one per reference deletion. The application message with the reference to pass and the increment system message to the object owner process are sent simultaneously. To avoid premature reference deletion, the object owner process must send an acknowledgement to the application message receiver process. References can only be deleted if the total number of received acknowledgements for the reference is equal to the total number of receipts of the reference from application message passing. The Lermen-Maurer protocol requires FIFO communication.

The Weighted Reference Counting Protocol

The *weighted reference counting* protocol (see Figure 2.1.b) [11, 107] avoids sending increment messages, and only one decrement message is required for reference deletion in most cases. The algorithm requires that each object and each reference maintain its *weight*. The weight of an object is always equal to the sum of the total weights of the references to the object. A new object and the original

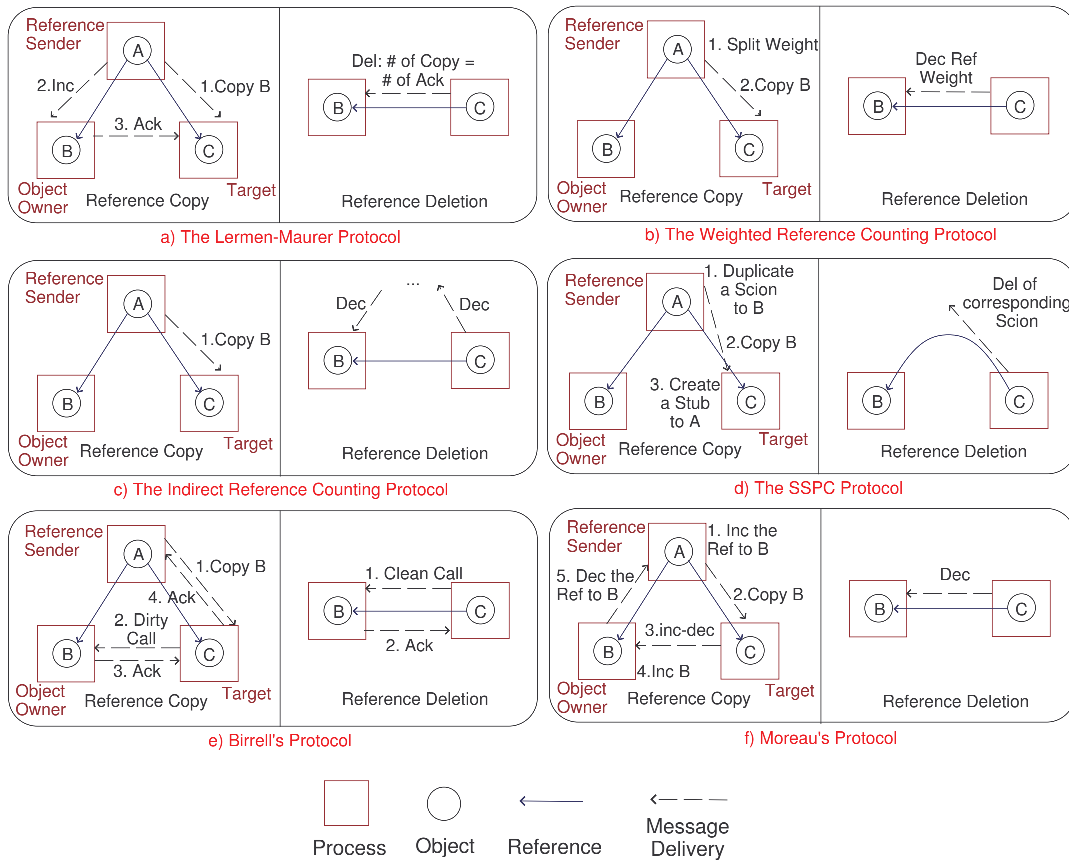

Figure 2.1: A comparison on distributed acyclic garbage collection algorithms. The figure explains how the reference sender process passes a reference of Object B to the target process (reference copy), and how a process deletes a reference of Object B and then notifies the object owner process (reference deletion).

reference pointing to it are created with a predefined maximum weight. When a reference is duplicated, the weight of the reference is divided into two positive new values whose sum is equal to the original weight of the reference. One value is assigned to the original reference and the other one to the new reference. When a reference is deleted, a decrement message containing the weight of the reference is sent to the object it points to. An object can be deleted safely when its weight drops to zero. However, an alternative approach must be used while duplicating a reference whose weight is 1, such as creating an indirect object with a value of maximum weight and then dividing the value by two to the original reference and

the copied reference. Corporaal et al in 1990 [26] use a similar approach using extra reference tables instead of indirect objects.

The Indirect Reference Counting Protocol

The *indirect reference counting* algorithm (see Figure 2.1.c), proposed by Piquer [77], avoids sending increment messages in order to minimize the number of extra messages to send. The algorithm maintains a special data structure for each reference which includes: 1) the number of duplications, 2) the parent pointer, and 3) the reference. The number of copies and the parent pointer are used to form an inverted diffusion tree to present the history of reference duplications. The root is the process that owns the object. The parent pointer is not used to point to the parent object, but for the history of reference duplication. Only the leaf references of the inverted diffusion tree can be deleted because the algorithm has to maintain the inverted diffusion tree structure. Object migration is done by changing the parent process to the migration destination. Deletion of references may produce extra messages. The algorithm may preserve zombie references because they are not leaf references.

The SSPC Protocol

The *SSPC (Stub-Scion Pair Chains)* protocol (see Figure 2.1.d) [85, 86] by Shapiro et al. is a reference listing based protocol which uses the techniques of time-stamps to ensure FIFO communication. Messages are re-delivered if the FIFO order is violated by comparing the time-stamps. Synchronous communication of users' applications is assumed, but the protocol itself is asynchronous. Whenever a reference is copied, an intermediate *stub-scion* pair is created to form an indirect path to connect the reference receiver process to the reference owner process. The SSPC protocol supports object migration by leaving an intermediate pointer at the original process to the newly migrated object. Another background protocol can be applied to reduce extra stub-scion pairs in a reference path to improve message delivery performance and robustness.

Birrell's Protocol

Birrell's protocol (see Figure 2.1.e) [12] is attributed as a remote-procedure-call, reference listing based algorithm, and thus it is a synchronous algorithm. It is used by Java RMI for distributed acyclic garbage collection. Upon reference duplication, the original call with reference parameters to the reference receiver process is made, following by another dirty call to the object owner process to create a *surrogate* (a remote inverse reference) which handles incoming requests to the referenced object. Reference deletion is implemented as a clean call which notifies the object owner process to remove the surrogate of the object. Moreau et al. [72] formalize the algorithm by a math model with correctness proofs, and also extend the algorithm with non-FIFO communication semantics.

Moreau's Protocol

Moreau's protocol (see Figure 2.1.f) [71] is a reference listing based protocol which requires point-to-point FIFO communication. Each process maintains a table for outgoing references, and each object maintains a counter of inverse references. Once a reference is passed to another process, the original process (sender) should increase the counter of the reference and then send the message out. If a process receives a message with a reference, it sends a special message *inc-dec* to the process where the referenced object resides. Upon receiving the inc-dec message, a process increases the counter of the referenced object and sends a *dec* message with the identity of the referenced object to the original process (sender). A process should then decrease the counter of the corresponding reference of the referenced object. Deleting references is similar to the Lermen-Maurer protocol.

2.2.2 Complete Distributed Tracing

A complete distributed tracing algorithm periodically starts a phase of global garbage collection. The major problems of complete distributed tracing are: 1) long pause times for global synchronization, and 2) the requirement of complete cooperation of all processes, which make it very sensitive to failures.

Hughe's Algorithm

Hughe's algorithm [47] uses global time-stamp propagation from roots. Garbage can be identified if an object has an earlier time-stamp than a certain global threshold. The algorithms requires a local garbage collector for each process, and it assumes: 1) a logical global clock, and 2) instantaneous message delivery. The algorithm uses stop-the-world synchronization, and a distributed termination detection algorithm to detect the end of global garbage collection.

2.2.3 Server-Based Tracing

A *server-based tracing* algorithm is hierarchical — it requires local garbage collection for each process and global garbage collection for the whole distributed system. The key feature of these algorithms is to identify distributed cyclic garbage either by physically or virtually migrating objects (partial reference graph) to one process, and let the process identify distributed garbage using a local object marking algorithm. Its major problem is that the server becomes a performance bottleneck in large-scale distributed systems.

The Ladin-Liskov Algorithm

The *Landin-Liskov* algorithm was first proposed in [61] and corrected in [55] by using a simplified version of Hughe's time-stamp algorithm to control the message reception order. The server clock is used as the global clock. Local garbage collection is performed at each process. A logically centralized server is designated to keep track of: 1) inter-process incoming and outgoing references, 2) the paths containing any inter-process incoming or outgoing reference, and 3) all local-root-reachable objects and references. The server runs a local mark-and-sweep algorithm to detect garbage and then notifies local garbage collectors.

2.2.4 Heuristic Migration and Tracing

Heuristics-based garbage collection was originated from Bishop's partitioned local garbage collection [13], in which the computing environment must support object mobility. Garbage collection is hierarchical — it requires local garbage collection. A heuristics-based algorithm starts from assuming an object is garbage,

then verifies it either by migration or distributed tracing. The problem of this approach is the penalty of a failed verification — it could be significant. Thus accurate heuristics is important to these algorithms. Most heuristics-based algorithms are time-based (the naïve approach) or distance-based.

The Distance-Based Heuristics and Migration Verification Algorithm

The *distance-based heuristics and migration verification* algorithm was presented in [63]. According to the original paper, the distance of an object is *"the minimum number of inter-node references in any path from a persistent root to that object, and the distance of an object unreachable from the persistent roots is infinity."* The persistent roots are real roots; remotely referenced objects are not. This algorithm requires distance propagation from every outgoing reference after each local garbage collection phase terminates. Once a remote object receives such a propagation message, it increases the distance inside the message by one, stores it, and then locally propagates the distance to its local descendants. Since garbage is detached from the persistent roots, its distance can only be increased. Once the distance of an object is greater than a predefined threshold, the object is suspected. The suspect objects and all its descendants are migrated to a single host to perform local garbage collection.

The Distance-Based Heuristics and Back Tracing Verification Algorithm

Back tracing verification was proposed by Fuchs [40], which assumes bi-directional references and ignores races among mutators and local garbage collectors. The idea was enhanced by Maheshwari and Liskov [64] with a reference management strategy. The *Maheshwari-Liskov* algorithm assumes a safe reference passing protocol, which can be found in many distributed reference listing algorithms. Only objects with outgoing references can be suspected because back tracing is defined to be started from an outgoing reference. No objects can be collected if the back tracing procedure identifies a root. Otherwise every back traced object is garbage.

The Min-Max Marking Heuristics Algorithm

The *min-max marking heuristics* algorithm was proposed in [57], and based on the work of [58] and [86]. The algorithm is complex, and requires four fields for mark propagation among processes and objects, including: 1) the distance, 2) the range of marking, 3) the mark generator identifier, and 4) the color. Only *White* and *Gray* color are used, and *White* is the initial color. Local roots and some incoming references (*scions*) are chosen to be the mark *generators*. The marking procedure follows a strict order by first sorting the local roots, and then the incoming references (scions) according to their marks. Outgoing references (*stubs*) store two marks, others only one. During a phase of local garbage collection, each object is marked twice. The first one follows the decreasing order, and then the increasing order. The incoming reference (scion) is marked by the *max* mark, and colored *Gray* if the *min* mark and the *max* mark are generated by two different sources (*generators*). If a generator receives its own mark with *White* color, there exists cyclic garbage. This approach cannot detect all cycles, and must use another approach, the *optimistic back-tracing* which creates many sub-generators to perform verification. A *sub-generator* is a generator that receives its *Gray* mark. During this phase, all incoming references (scions) that receive the same *Grey* mark become sub-generators. A sub-generator emits its *White* mark. When all sub-generators and the generator receive the *White* mark, the cycle can be removed safely. The paper [57] dos not provide correctness proofs.

The Trial Deletion Algorithm

Vestal proposed the *trial deletion* algorithm in [103]. The algorithm uses reference counting. Each object maintains an extra trial count. If an object is suspected as garbage, it is virtually deleted by sending a simulated decrement message to decrease its trial count. If the trial count of an object drops to zero, the object must be garbage. The algorithm does not work well if two trials happen in the same garbage chain, and it cannot detect all cycles.

2.2.5 Group Tracing

Group tracing algorithms are either based on static partitioning or heuristic dynamic partitioning. These algorithms are devised for large-scale distributed systems since garbage collection is limited to a small group of processes.

The Lang-Queinnec-Piquer Algorithm

The *Lang-Queinnec-Piquer* algorithm [56] uses a reference listing protocol and local garbage collectors. It starts from defining hierarchical groups for global garbage collection, and then identifies every object that participates in each of the groups. Roots and inter-group referenced objects are marked *hard*. Other intra-group referenced objects are marked *soft*. After identifying the hard objects, it performs global mark propagation with the help of local garbage collectors. The global propagation requires distributed termination detection. When the global marking is done, objects marked soft are cyclic garbage, and the group can be dismissed. This paper also suggests the use of predefined hierarchical groups. Two major problems occur in this approach: 1) the mutators must be stopped during the marking phase to maintain correctness [65], and 2) no inter-group migration is allowed after group identification. Since a garbage collection phase can take a long time, this approach is not scalable.

The Rodrigues-Jones Dynamic Partitioning Algorithm

The *Rodrigues-Jones dynamic partitioning* algorithm [80] also uses a reference listing protocol and local garbage collectors. The algorithm starts from suspecting an object as garbage, traces from it, and then forms a group for global garbage collection. During global marking, mutators must be suspended because it uses the mostly parallel garbage collection algorithm [15] for local garbage collection. Another problem is that overlapping partitions can occur, which either causes deadlocks, or prevents progress. The authors choose the latter approach.

2.2.6 Distributed Generational Garbage Collection

Hudson et al. [45] proposed a *generational* collector where the address space of each computing node is divided into several disjoint blocks (*cars*), and cars are

grouped together into several distributed *trains*. Each train represents a generation of objects, and forms a ring structure for distributed management. Objects can only move from an older generation car to a younger generation car, and the oldest car is eventually inspected. A car/train can be disposed of if there are no incoming inter-car/inter-train references to it. The granularity of cars/trains affects the performance.

2.2.7 Snapshot-Based Tracing

Snapshot-based algorithms are popular in actor garbage collection, but seldom used in distributed passive object garbage collection. A benefit of snapshot-based algorithms is that garbage tracing can be done in a concurrent manner, which enables mutation during the global marking phase with the tradeoff of extra space.

The Distributed Cycles Detection Algorithm

The *distributed cycles detection* algorithm (*DCDA*) [101] uses asynchronous local snapshots which have to be updated by local mutators to notify application changes to the snapshot. The algorithm starts from suspecting an object as garbage, and a heavy cycle detection message (*CDM*) is then traversed among the snapshots to see if a cycle exists. If any traversed object is modified by the mutators, the current activity for global garbage collection must abort.

2.3 Equivalence of Passive Object Garbage Collection and Distributed Termination Detection

Distributed termination detection [38, 29, 66] is a similar problem to passive object garbage collection, where processes are reactive and communicate with each other in a synchronous manner. A system consists of several processes, which are connected by unidirectional channels as a *strongly connected graph* — there exists a path from each process to every other process. A process is initially blocked (passive) or unblocked (active). Only unblocked processes can perform computations, including sending application messages. Blocked processes can become unblocked by receiving application messages. A system is ready for termination if 1) all processes

are blocked and 2) there are no messages in transit.

Distributed Termination Detection

Most distributed termination detection algorithms depend on specific network topologies, such as a computation tree [29] and a ring [30, 69]. For example, the *Dijkstra-Scholten* algorithm [29] requires that processes form a computation tree, which is also known as a *diffusing tree* because it always starts with a root process. An internal process is marked unblocked if any of its children is marked unblocked; otherwise it is marked blocked. The mark for a leaf process depends on its state. A system can be terminated if the root is marked blocked.

A *probe-based* (or *wave-based*) algorithm uses an initiator process to send system messages to pass to every process directly or indirectly. Dijkstra et al. [30] proposed such a probe-based algorithm for ring topologies, where the initiator process sends a *White* mark token which is a system message to its successor when it becomes blocked. Every process is numbered and can only pass messages to its successor, and the successor of the last process is the first process. All processes are initially *White*, and a *White* process passes the token it receives to its successor. Whenever an unblocked process sends an application message to another process, it becomes *Black*, and passes the received token by marking it *Black*. If the initiator process receives a *White* mark token, the system can be terminated correctly.

Szymanski et al. [94, 93] proposed a distributed termination detection algorithm which uses *distance and time heuristics*. The length of the shortest path from Process P_x to Process P_y is called a *distance* from Process P_x to Process P_y. The greatest distance between any two processes is called the *diameter*, which is used as the threshold to determine the termination condition. The initial shortest distance state of a process is 0. All processes have to stop periodically at a logical time barrier to calculate their *shortest distance states* from any unblocked process. A process sends its shortest distance state as a token to its successors and then reads tokens from its predecessors. Consequently, it sets its shortest distance state to be 0 if it is unblocked; otherwise, it chooses the minimum value of the input tokens and its current shortest distance state, and then increments the value by one as its new

shortest distance state. If the value exceeds the diameter of the system, the process can terminate correctly, and all processes are guaranteed to terminate at the same logical time.

Equivalence of Passive Object Garbage Collection and Distributed Termination Detection

Tel and Mattern [95] has shown that the distributed termination detection problem for distributed computations can be modeled as an instance of the garbage collection problem. Blackburn et al. [14] suggest a methodology to derive a distributed garbage collection algorithm from an existing distributed termination detection algorithm, in which the distributed garbage collection algorithm developers must design another algorithm to guarantee a consistent global state. All of the above algorithms cannot be reused directly in actor systems because actors and passive objects are different in nature.

2.4 Actor Garbage Collection

The definition of garbage actors is very different from passive object garbage collection. For instance, the marking phase of passive object garbage collection usually uses depth-first-search or breadth-first-search. Marking algorithms for actor garbage collection include Push-Pull [53], Is-Black [53], Dickman's graph partition merging algorithm [28], and the actor reference graph transformation algorithm [97, 98]. Most distributed actor garbage collection algorithms are snapshot-based due to the autonomous nature of actors.

2.4.1 Actor Marking Algorithms

This subsection introduces various marking algorithms of actor garbage collection. Most of them are not as intuitive as the marking algorithms of passive object garbage collection.

The Push-Pull and Is-Black Algorithms

The *Push-Pull* and *Is-Black* algorithms were proposed by Kafura, Mukherji, and Nelson [53]. The *Push-Pull* algorithm is based on Nelson's coloring rules [73],

and it has two major functions, *Pusher* and *Puller*. The algorithm uses set operations instead of mark propagation. Three different sets are used: *White*, *Gray*, and *Black*. Roots are initially put in *Black*, and others in *White*. *Black* means an actor is live, *Gray* means an actor is blocked but can communicate with a root if it can become unblocked, and *White* means that no live evidence is seen for an actor. *Pusher* and *Puller* keep pushing and pulling different actors in and out of different color sets. After the termination of the algorithm, actors in *Black* are live and others are garbage.

Is-Black initially marks roots *Black*, and other actors *White*. The second step is to mark all transitively root-reachable actors *Black*. The third step is to transitively follow references from an unblocked *White* actor to find if it can reach a *Black* actor. If so, mark *Black* every actor that is transitively reachable from that actor, and then restart the third step. If no such case is found, the algorithm terminates and all *Black* actors are live. Let N be the total number of actors, and E be the total number of references. The time complexity is $O(N \cdot E)$ for *Push-Pull*, and $O(N^2)$ for *Is-Black*. The extra space complexity is $O(N)$ for both of them.

Dickman's Partition Merging Algorithm

Dickman proposed the *partition merging* algorithm [28] which treats all unblocked actors as potential roots. The first part of the algorithm constructs several preliminary partitions of actors by traversing references from the potential roots, and then merges them into several large partitions by a special *X-node*. Actors in the same partition have the same status, which means all of them are either garbage or live. The second part of the algorithm then verifies each partition from the root actors using an Euler cycle traversal algorithm. The time complexity of the algorithm is $O(N + E)$, and the extra space complexity is $O(N + E)$.

The Actor Reference Graph Transformation Algorithm

The *actor reference graph transformation* algorithm was proposed in [97, 98]. The key concept of the algorithm is to perform a garbage-preserving transformation from the actor reference graph into a corresponding passive object reference graph. All roots are merged into a single root initially. Every actor is then transformed

into two objects — one represents the message box, and the other represents the actor. If an actor is an unblocked actor or a root, a reference is added from the message box object to the actor object. For any reference $\overline{a_p a_q}$, add three references in the transformed graph: 1) a reference from the a_p object to the a_q object, 2) a reference from the a_p object to the message box object of a_q, and 3) a reference from the message box object of a_q to the message box object of a_p. The root in the transformed reference graph is defined as the message box of the only root. A typical object marking algorithm is then used to identify live actors. Its time complexity is $O(N + E)$ and the extra space complexity is $O(N + E)$.

2.4.2 Distributed Actor Garbage Collection

Only a few distributed actor garbage collection algorithms can be found in the literature. In this subsection, four distributed actor garbage collection algorithms are reviewed. Three of them are snapshot-based, and one uses stop-the-world synchronization.

The most well known snapshot-based garbage collection algorithm was proposed by Yuasa [111] as part of the Kyoto Lisp concurrent programming language, designed for passive object garbage collection in shared memory systems. Distributed snapshot is more complicated because the requirement of detecting both the state of processes (actors) and channels (in-transit messages). Two kinds of snapshot algorithms are used in distributed garbage collection — the *Chandy-Lamport* algorithm [20] and the *uncoordinated snapshot* algorithm [89, 67].

The Global Push-Pull Algorithm

This global *Push-Pull* algorithm [52] uses hierarchical garbage collection — local garbage collection for each computing node, and a global synchronization agent located at some computing node. Once a global garbage collection phase begins, the Chandy-Lamport snapshot algorithm is used to take a coherent global snapshot. Distributed termination detection is required for the termination of distributed snapshot. Global marking by the global Push-Pull algorithm is performed when a global snapshot is available. Another distributed termination detection has to be used to detect the end of the global marking phase. The algorithm requires FIFO commu-

nication for Chandy-Lamport snapshot algorithm to record the state of channels, which violates the assumption of the actor model of computation.

The HDGC Algorithm

The *hierarchical distributed garbage collection* (*HDGC*) algorithm [102] is a Chandy-Lamport snapshot-based algorithm. It assumes a two-dimensional grid network topology, FIFO communication, and a centralized synchronization service — the *GC-root actor*. The FIFO communication assumption provides the ability to clear communication channels by a special *bulldoze* message, which is used for taking global snapshot. Any message sent in a cleared channel is marked as *new*. Inverse references are built during garbage collection for a distributed actor marking algorithm. The algorithm has five phases: 1) the *pre-GC* phase to start a global snapshot, 2) the *distributed scavenge* phase to identify live actors in the snapshot, 3) the *local-clear initiation* phase to notify each local collector to terminate the second phase, 4) the *local-clear* phase to reclaim garbage, and 5) the *post GC broadcast* phase to signal every computing node to terminate this phase of global garbage collection.

The Time-Stamp-Vector Snapshot Algorithm

Puaut proposed an asynchronous, *time-stamp-vector snapshot based* algorithm in [79], which uses a server for global garbage collection. Each computing node maintains a time-stamp vector to simulate a global clock. FIFO communication is assumed, but completely unordered communication is also possible by incorporation of time-stamp-based message redelivery mechanism. Local garbage collectors send local reference graphs to a centralized server, as well as the time-stamps of the last information received by the local garbage collectors. The server checks if the combined snapshot is consistent according to the time-stamps. If this is not true, nothing is done. Otherwise the server performs global garbage collection, and then notifies each local garbage collector about garbage. This approach is not scalable because: 1) the size of a message increases as the number of computing node increases, and 2) the probability of obtaining a consistent global snapshot decreases as the locality of references decreases.

The Distributed Actor Reference Graph Transformation Algorithm

The distributed actor reference graph transformation algorithm [97] uses local garbage collectors and a distributed mark-and-sweep algorithm. While performing distributed actor garbage collection, the local garbage collector obtains information from the local computing host where it resides, and then transforms the local actor reference graph into a passive object reference graph. If two actors, namely a_p and a_q, reside in different computing hosts and a_p has a reference to a_q, a message is sent to a_q's computing host to indicate that it is referenced by a_p. In such case, a_p has to be suspended until the message has been delivered. To guarantee safety, the algorithm, in general, requires message delivery to cause temporary suspension of the sender — whenever an actor sends a message to another actor, the actor must be suspended until the message has been delivered. After the completion of the transformation procedure, a suitable global marking algorithm such as Schelvis' algorithm [84] is selected to identify distributed garbage. To conclude, the algorithm and its implementation assume: 1) First-In-First-Out (FIFO) communication, 2) temporary suspension of the message sender, and 3) a stop-the-world approach for local garbage collectors.

CHAPTER 3
Equivalence of Actor Garbage Collection and Passive Object Garbage Collection

In this chapter, we define passive object garbage collection and actor garbage collection as graph problems. Then we discuss the actor garbage collection problem based on the reference graph without considering any distributed or concurrent computing issues such as in-transit messages. We will discuss: 1) transformation methods from actor garbage collection to passive object garbage collection, and 2) two actor marking algorithms derived from the transformation methods.

3.1 Transformation Methods

Both the passive object garbage collection and the actor garbage collection problems can be represented as graph problems. This section reveals that they can be transformed from one to the other. Properties and formal definitions of the transformation methods are provided, and proofs can be found in Section 3.3. This section will cover:

- the definition of passive object garbage collection,
- the definition of actor garbage collection,
- how passive object garbage collection can be transformed to actor garbage collection, and
- two transformation methods and their implementations (algorithms) from actor garbage collection to passive object garbage collection.

3.1.1 Garbage in Passive Object Systems

The essential concept of passive object garbage lies in the idea of the possibility of object manipulation. Objects that can be manipulated by the thread of control of the application are *live*; otherwise they are *garbage*. There are two

possible manipulations in passive object garbage collection — *direct* and *transitive* manipulations. Root objects are those which can be directly accessed by the thread of control, while transitively live objects are those transitively reachable from the root objects by following references. The problem of passive object garbage collection can be represented as a graph problem. To concisely describe the problem, we introduce *transitive reachability* $\rightsquigarrow$. The transitive reachability relation is *reflective* ($a \rightsquigarrow a$) and *transitive* ($(a \rightsquigarrow b) \wedge (b \rightsquigarrow c) \Rightarrow (a \rightsquigarrow c)$). Then we use it to define the passive object garbage collection problem.

Definition 3.1.1. *Transitive reachability.*
Entity (object or actor) o_q is transitively reachable from o_p, denoted by

$$o_p \rightsquigarrow o_q,$$

if and only if $o_p = o_q \vee (\exists o_u : \overline{o_p o_u} \wedge o_u \rightsquigarrow o_q)$ [2].
Otherwise, we say $o_p \not\rightsquigarrow o_q$.

Definition 3.1.2. *Live passive objects.*
Given a passive object reference graph $G = \langle V, E \rangle$, where V represents objects and E represents references, let R represent roots such that $R \subseteq V$: The problem of passive object garbage collection is to find the set of live objects, $Live_{object}(G, R)$, where

$$Live_{object}(G, R) \equiv \{o_{live} \mid \exists o_{root} : (o_{root} \in R \wedge o_{live} \in V \wedge o_{root} \rightsquigarrow o_{live})\}$$

3.1.2 Garbage in Actor Systems

The definition of actor garbage is related to the idea of whether an actor is doing *meaningful computation*, which is defined as having the ability to communicate with any of the *root actors*, where root actors are I/O services or public services such as web services and databases. We assume that *every actor/object has a reference to itself*, which is not necessary true in the actor model. The widely used definition of live actors [53] is based on the possibility of message reception from or message delivery to the root actors — a *live* actor is one which can either receive messages from the root actors or send messages to the root actors. The original definition of

[2]Notice that $\overline{o_p o_u}$ is defined as a reference from o_p to o_u (see Section 1.3).

live actors is denotational because it uses the concept of "potential" message delivery and reception. To make it more operational, the state of an actor (unblocked or blocked) and the referential relationship of actors must be used instead.

Definition 3.1.3. *Potential message delivery from a_p to a_q.*
Let the current system state be S. Potential message delivery from Actor a_p to Actor a_q (or message reception of a_q from a_p) is defined as:

$$\exists S_{future} : a_p \textit{ is unblocked and } a_p \rightsquigarrow a_q \textit{ at } S_{future},\ S \rightarrow^* S_{future}.$$

Now, consider two actors, a_p and a_q. If they are both transitively reachable from an unblocked actor or a root actor, namely a_{mid}, message delivery from Actor a_p to Actor a_q (or from a_q to a_p) is possible. The reason is that there exists a sequence of state transitions such that a_{mid} transitively makes a_p unblocked and transitively creates a directional path to a_q. As a result, $a_p \rightsquigarrow a_q$ is possible. The relationship of a_p and a_q can be expressed by the *may-talk-to* relation, defined as $\leftrightsquigarrow$ (Definition 3.1.4). It is also possible that a message can be delivered from a_p to another new actor a_r if ($a_p \leftrightsquigarrow a_q \wedge a_q \leftrightsquigarrow a_r$) because the unblocked actors can create a path to connect a_p and a_r. The generalized idea of the *may-transitively-talk-to* relation, $\leftrightsquigarrow^*$, is shown in Definition 3.1.5 to represent potential message delivery.

Definition 3.1.4. *May-talk-to $\leftrightsquigarrow$.*
Given an actor reference graph $G = \langle V, E \rangle$ and $\{a_p, a_q\} \subseteq V$, where V represents actors and E represents references, let R represent roots and U represent unblocked actors such that $R, U \subseteq V$, then:

$$a_p \leftrightsquigarrow a_q \iff \exists a_u : a_u \in (U \cup R) \wedge a_u \rightsquigarrow a_p \wedge a_u \rightsquigarrow a_q.$$

We call $\leftrightsquigarrow$ the may-talk-to relation.

Definition 3.1.5. *May-transitively-talk-to $\leftrightsquigarrow^*$.*
Following Definition 3.1.4,

$$a_p \leftrightsquigarrow^* a_q \iff \exists a_{mid} : a_p \leftrightsquigarrow a_q \vee (a_p \leftrightsquigarrow a_{mid} \wedge a_{mid} \leftrightsquigarrow^* a_q).$$

We call $\leftrightsquigarrow^*$ the may-transitively-talk-to relation.

The definition of the set of live actors can then be concisely rewritten by using the $\leftrightsquigarrow^*$ relation:

Definition 3.1.6. *Live actors.*
Given an actor reference graph $G = \langle V, E \rangle$, where V represents actors and E represents references, let R represent roots and U represent unblocked actors such that $R, U \subseteq V$. The problem of actor garbage collection is to find the set of live actors $Live_{actor}(G, R, U)$, where

$$Live_{actor}(G, R, U) \equiv \{a_{live} \mid \exists a_{root} : (a_{root} \in R \wedge a_{live} \in V \wedge a_{root} \leftrightsquigarrow^* a_{live})\}$$

3.1.3 Problem Equivalence

Transformation from Passive Object Garbage Collection to Actor Garbage Collection

Let the passive object reference graph be $G = \langle V, E \rangle$ and the set of roots be R. Let the transformed actor reference graph be $G' = \langle V', E' \rangle$, the set of roots be R', and U' be the set of unblocked actors. The problem of passive object garbage collection can be transformed into the problem of actor garbage collection by assigning $V' = V$, $E' = E$, $R' = R$ and $U' = \emptyset$. Then for any two objects o_r and o_q, we get $(o_r \rightsquigarrow o_q \wedge o_r \in R) \Longleftrightarrow (o_r \rightsquigarrow o_r \wedge o_r \rightsquigarrow o_q \wedge o_r \in R) \Longleftrightarrow (o_r \leftrightsquigarrow^* o_q \wedge o_r \in R)$. Therefore the set $Live_{object}(G, R) = Live_{actor}(G', R', U')$.

Transformation from Actor Garbage Collection to Passive Object Garbage Collection

Now, consider the backward transformation. Let the actor reference graph be $G = \langle V, E \rangle$, R be the roots and U be the unblocked actors. If there exist $G' = \langle V', E' \rangle$ and R' such that $Live_{actor}(G, R, U) = Live_{object}(G', R')$, we say the actor garbage collection problem can be transformed into the passive object garbage collection problem. The transformation problem has been solved and proven by Vardhan and Agha [98] by changing V, E, and R. In the following subsections, we will show that changing R is enough.

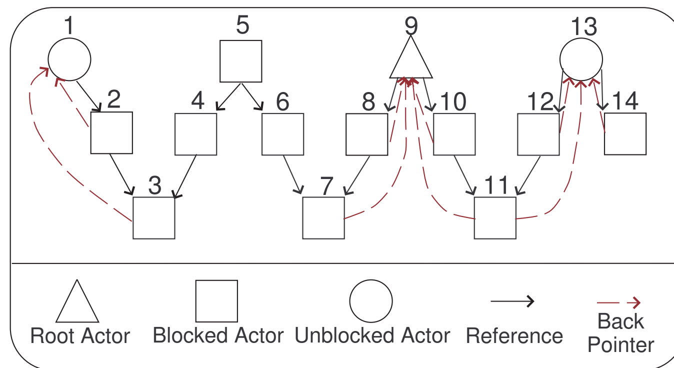

Figure 3.1: An example of transformation by direct back pointers to unblocked actors and root actors.

3.1.4 Transformation by Direct Back Pointers to Unblocked Actors

In this subsection we propose a much easier approach to transform actor garbage collection into passive object garbage collection, by making $E' = E \cup \{\overline{a_q a_u} \mid a_u \in (U \cup R) \wedge a_u \rightsquigarrow a_q\}$. See Figure 3.1 for example. Actors 2 and 3 have back pointers to Unblocked Actor 1 because they are reachable from Actor 1. Actor 11 has a back pointer to Root Actor 9 and another one to Unblocked Actor 13 for the same reason. Actor 3 does not have a back pointer to Actor 5 because Actor 5 is neither a root nor an unblocked actor. Notice that we use the term *back pointers* to describe the newly added references and to avoid ambiguity with the term *inverse references*. Theorem 3.1.7 shows that the direct back pointer transformation method is correct.

Theorem 3.1.7. *Direct back pointer transformation.*
Let the actor reference graph be $G = \langle V, E \rangle$, R be the set of roots, and U be the set of unblocked actors. Let $E' = E \cup \{\overline{a_q a_u} \mid a_u \in (U \cup R) \wedge a_u \rightsquigarrow a_q\}$, and $G' = \langle V, E' \rangle$ and $R' = R$.

$$Live_{actor}(G, R, U) = Live_{object}(G', R')$$

3.1.5 Transformation by Indirect Back Pointers to Unblocked Actors

In this subsection we propose another similar approach to transform actor garbage collection into passive object garbage collection, by making the reference

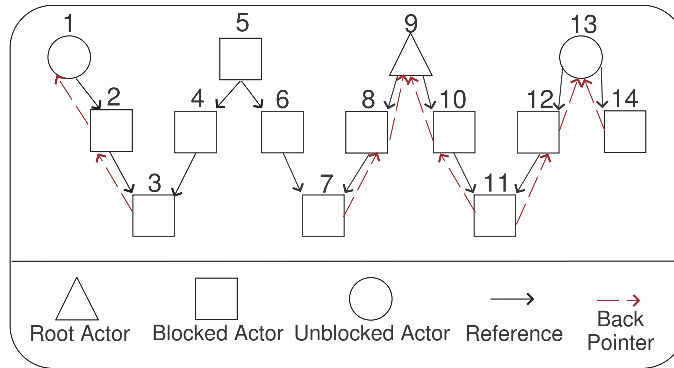

Figure 3.2: An example of transformation by indirect back pointers to unblocked actors and root actors.

set $E' = E \cup \{\overline{a_q a_p} \mid a_u \in (U \cup R) \wedge \overline{a_p a_q} \in E \wedge a_u \rightsquigarrow a_p\}$. See Figure 3.2 for example. Actor 2 has back pointers to Unblocked Actor 1 and Actor 3 has back pointers to Actor 2 because they are reachable from Actor 1. The newly added back pointers will create a corresponding counter-directional path of a path from an unblocked/root actor to another actor which is reachable from the unblocked/root actor. Similarly, Actor 11 has a new counter-directional path to Root Actor 9 and another one to Unblocked Actor 13.

Lemmas 3.1.8 and 3.1.9 are used to prove Theorem 3.1.10. Lemma 3.1.8 says that if an actor is reachable from an unblocked/root actor, the indirect back pointer transformation method guarantees that there exists an indirect inverse path from the actor to the unblocked/root actor. Lemma 3.1.9 says that the set of live objects in the transformed reference graph by the indirect back pointer transformation method is a subset of the set of live actors. Theorem 3.1.10 shows that the indirect back pointer transformation method is correct.

Lemma 3.1.8. *Backward reachability to the unblocked/root actors in the newly transformed graph.*
Let the actor reference graph be $G = \langle V, E \rangle$, R be the set of roots, and U be the set of unblocked actors. Let $E' = E \cup \{\overline{a_q a_p} | a_u \in (U \cup R) \wedge \overline{a_p a_q} \in E \wedge a_u \rightsquigarrow a_p\}$, and $G' = \langle V, E' \rangle$ and $R' = R$. Then in G',

$$\forall a_x, a_y : a_x \in (U \cup R) \wedge a_x \rightsquigarrow a_y \ in\ G \Longrightarrow a_x \in (U \cup R) \wedge a_y \rightsquigarrow a_x \ in\ G'.$$

Lemma 3.1.9. *$Live_{object}(G', R') \subseteq Live_{actor}(G, R, U)$.*
Let the actor reference graph be $G = \langle V, E \rangle$, R be the set of roots, and U be the set of unblocked actors. Let $E' = E \cup \{\overline{a_q a_p} \mid \exists a_u : a_u \in (U \cup R) \wedge \overline{a_p a_q} \in E \wedge a_u \rightsquigarrow a_p\}$, and $G' = \langle V, E' \rangle$ and $R' = R$. Then in G',

$$\forall a_x, a_y : a_x \in (R') \wedge a_x \rightsquigarrow a_y \text{ at } G' \implies a_x \in (U \cup R) \wedge a_y \leftrightsquigarrow^* a_x \text{ at } G,$$

that is, $Live_{object}(G', R') \subseteq Live_{actor}(G, R, U)$.

Theorem 3.1.10. *Indirect back pointer transformation.*
Let the actor reference graph be $G = \langle V, E \rangle$, R be the set of roots, and U be the set of unblocked actors. Let $E' = E \cup \{\overline{a_q a_p} \mid \exists a_u : a_u \in (U \cup R) \wedge \overline{a_p a_q} \in E \wedge a_u \rightsquigarrow a_p\}$, and $G' = \langle V, E' \rangle$ and $R' = R$.

$$Live_{actor}(G, R, U) = Live_{object}(G', R')$$

3.2 Implementation of the Transformation Methods

This section introduces two algorithms derived from the transformation methods of Theorem 3.1.7 and Theorem 3.1.10. They are the back pointer algorithm and the N-color algorithm.

3.2.1 The Back Pointer Algorithm

Either Theorem 3.1.7 or Theorem 3.1.10 can directly turn into an actor marking algorithm by simply adding new back pointers in the actor reference graph. Due to their similarity, we only select Theorem 3.1.7 to model the back pointer algorithm.

In the algorithm (see Figure 3.3), actors are all initially marked *White*. Then it first identifies all potentially live actors from root/unblocked actors by marking them *Gray* and simultaneously adding new back pointers in the graph. With the existing references and newly created corresponding back pointers, it transitively identifies live actors from roots and marks them *Black*. After the termination of the algorithm, any *Black* actor is live. An example is illustrated in Figure 3.4.

```
Algorithm Back_pointer
 1. all actors are initially marked White
 2. for each unblocked/root actor x do
 3.    if x.COLOR = White then
 4.       x.COLOR ← Gray
 5.       call DFS_potential_root_marking(x)
 6. for each root actor r do
 7.    if x.COLOR ≠ Black then
 8.       r.COLOR ← Black
 9.       call DFS_bidirection_marking(r)

Procedure DFS_potential_root_marking(Actor x)
 1. for each reference (x,y) held by x
 2.    if y.COLOR = White
 3.       build a back pointer of (x,y) in actor y
 4.       y.COLOR ← Gray
 5.       call DFS_potential_root_marking(y)

Procedure DFS_bidirection_marking(Actor x)
 1. for each reference (x,y) held by x do
 2.    if y.COLOR ≠ Black then
 3.       y.COLOR ← Black
 4.       call DFS_bidirection_marking(y)
 5. for each back pointer of (z,x) held by x do
 6.    if z.COLOR ≠ Gray
 7.       z.COLOR ← Black
 8.       call DFS_bidirection_marking(z)
```

Figure 3.3: The back pointer algorithm.

3.2.2 The N-Color Algorithm

The N-color algorithm (see Figure 3.5) comes from Theorem 3.1.7 with the strategy of turning "a back pointer to an unblocked/root actor" into a color. A root color is defined as 0. To avoid multiple colors in an actor, only one color is allowed in each actor. Once different colors conflict in an actor, we combine the colors by using disjoint set operations [24]. Since any color conflict implies the two representive unblocked/root actors have the relation of $\leftrightsquigarrow^*$, we can conclude that actors marked by any color in a disjoint set containing a root color are live.

Actors marked by the same colors may talk to each other because they are

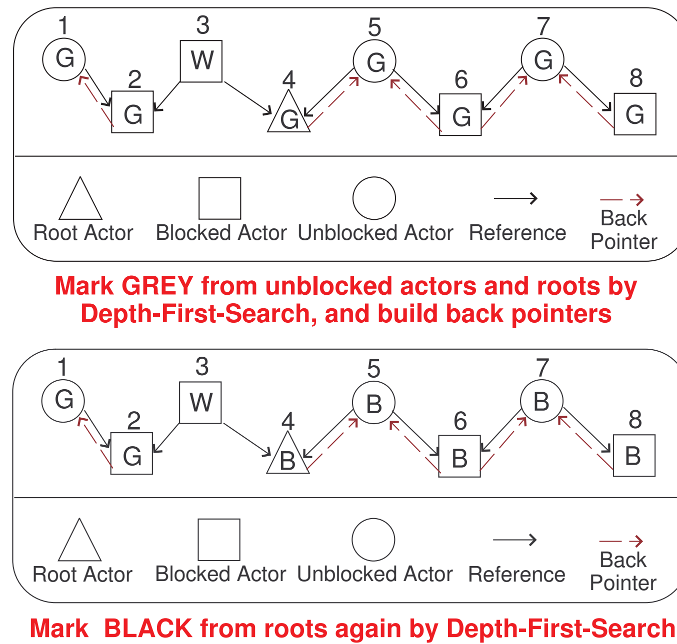

Figure 3.4: An example of the back pointer algorithm where W stands for *White*, G stands for *Gray*, and B for *Black*. Only actors marked by Color B are live.

directly reachable from the same unblocked actor. Color conflict implies the may-transitively-talk-to relationship. Therefore, combining conflict colors is equivalent to grouping the set of actors that may transitively talk to each other. If a root is included in this set, every actor in the set may transitively talk to the root. As a result, the algorithm is correct.

Let M be the number of unblocked actors. Three disjoint set operations are used by the algorithm:

1. `Create-Set(i)` to create a set containing only one element i. Its amortized time complexity is $O(1)$.

2. `Find-Set(i)` to find the set containing i. Its amortized time complexity is $O(\lg^* M)$ [3].

3. `Union(i,j)` to union one set containing i and another set containing j. Its

[3] $\lg^*$, defined in [25], is a function which grows very slowly. For example, $lg^*65536 = 4$ and $lg^*(2^{65536}) = 5$.

```
Algorithm N_color_marking
 1. all actors are initially marked −1
 2. for each root actor r do
 3.    r.COLOR ← 0
 4.    call DFS_marking(r,0)
 5. n ← the number of unblocked actors
 6. for i ← 0 to n do
 7.    Create−Set(i)
 8. currentColor ← 0
 9. for each unblocked actor x do
10.    currentColor ← currentColor + 1
11.    if x.color = −1 then
12.      x.color ← currentColor
13.      call DFS_marking(x,currentColor)

Procedure DFS_marking(Actor x, Color c)
 1. for each reference (x,y) held by x do
 2.    if y.COLOR = −1 then
 3.      y.COLOR ← c
 4.      call DFS_marking(y,c)
 5.    else if Find−Set(y.COLOR) ≠ Find−Set(c) then
 6.      Union(y.COLOR, c)
```

Figure 3.5: The N-color algorithm.

amortized time complexity is $O(\lg^* M)$.

The N-color marking algorithm is shown in Figure 3.5 with an example in Figure 3.6. It uses $M + 2$ colors to identify live actors. Let the initial color of non-root actors be -1, and the initial color of roots be 0. Other normal colors are ranging from 1 to M where each normal color represents a potentially live group. By using the DFS or BFS algorithm, the joint vertices of groups can be identified. The set operations make each color transitively point to the lowest color (including 0) it encounters during the marking phase. At the end of the marking phase, a color belonging to the set containing color 0 is a *root color*. Each actor marked by a root color is live.

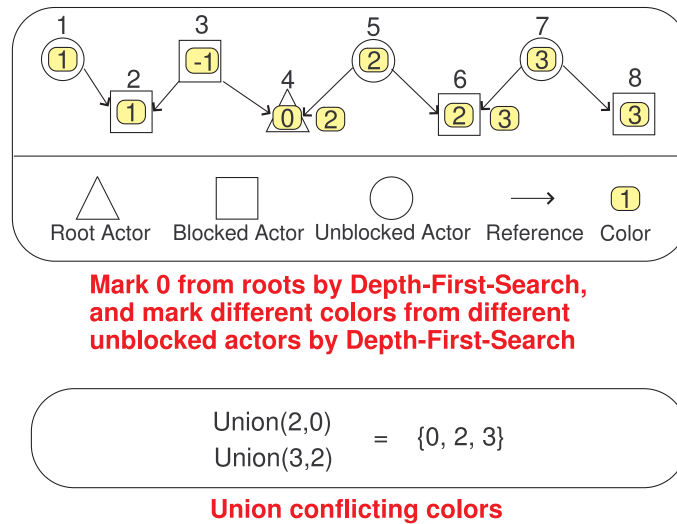

Figure 3.6: An example for the N-color algorithm. Actors marked by Colors 0, 2, or 3 are live. Actors colored -1 and 1 are garbage.

3.2.3 Complexity Analysis

Let N be the number of actors, E the number of references in the system, and M be the number of unblocked actors. The time complexity of the back pointer algorithm is $O(N + E)$, equal to the current known best [28]. The extra space complexity is $O(N + E)$ which is theoretically worse than the current best, $O(N)$ [53]. The back pointer algorithm requires scanning the reference graph twice.

The extra space complexity of the N-color algorithm is $O(M+N)$ where $O(M)$ is for the disjoint set operations and $O(N)$ is for marking, and thus makes it the best among the actor marking algorithms. The algorithm (Figure 3.5) requires tracing references in Lines 2–4 and Lines 9–13, making the time complexity $O(E + E \lg^* M) = O(E \lg^* M)$. Other pieces of the algorithm are at most $O(N)$, which means the time complexity of the algorithm is $O(N + E \lg^* M)$, very close to the current known best. The N-color algorithm is also the best among the actor marking algorithms for only scanning the reference graph once.

3.3 Proofs

Theorem 3.1.7 Direct back pointer transformation.

Let the actor reference graph be $G = \langle V, E \rangle$, R be the set of roots, and U be the set of unblocked actors. Let $E' = E \cup \{\overline{a_q a_u} \mid a_u \in (U \cup R) \wedge a_u \rightsquigarrow a_q\}$, and $G' = \langle V, E' \rangle$ and $R' = R$.

$$Live_{actor}(G, R, U) = Live_{object}(G', R')$$

Proof. Let $Live_{actor}(G, R, U) = \{a_{live} \mid \exists a_{root} : (a_{root} \in R \wedge a_{live} \in V \wedge a_{root} \leftrightsquigarrow^* a_{live})\}$ of G, and $Live_{object}(G', R') = \{o_{live} \mid \exists o_{root} : (o_{root} \in R' \wedge o_{live} \in V \wedge o_{root} \rightsquigarrow o_{live})\}$ of G'.

Now, consider the first case, $Live_{actor}(G, R, U) \subseteq Live_{object}(G', R')$. Let a_r and a_l be actors and $a_r \in R \wedge a_l \in V$. Then in G:

$a_r \leftrightsquigarrow^* a_l \Longrightarrow$

$\exists a_{mid,1}, a_{mid,2}, ..., a_{mid,n} : a_r \leftrightsquigarrow a_{mid,1} \leftrightsquigarrow a_{mid,2} \leftrightsquigarrow ... \leftrightsquigarrow a_{mid,n} \leftrightsquigarrow a_l \Longrightarrow$

$\exists a_{mid,1}, a_{mid,2}, ..., a_{mid,n}, a_{u,1}, a_{u,2}, ..., a_{u,n+1} : \{a_{u,1}, a_{u,2}, ..., a_{u,n}\} \subseteq (U \cup R) \wedge (a_{u,1} \rightsquigarrow a_r \wedge a_{u,1} \rightsquigarrow a_{mid,1}) \wedge (a_{u,2} \rightsquigarrow a_{mid,1} \wedge a_{u,2} \rightsquigarrow a_{mid,2}) \wedge ... \wedge ((a_{u,n+1} \rightsquigarrow a_{mid,n} \wedge a_{u,n+1} \rightsquigarrow a_l))$.

The above statement is true in G' because $E \subseteq E'$. Since $\forall a_x, a_y : a_x \in (U \cup R) \wedge a_y \in V \wedge a_x \rightsquigarrow a_y \Longrightarrow a_y \rightsquigarrow a_x$ in G', we know $\exists a_{mid,1}, a_{mid,2}, ..., a_{mid,n}, a_{u,1}, a_{u,2}, ..., a_{u,n+1} : \{a_{u,1}, a_{u,2}, ..., a_{u,n}\} \subseteq (U \cup R) \wedge a_r \rightsquigarrow a_{u,1} \rightsquigarrow a_{mid,1} \rightsquigarrow a_{u,2} \rightsquigarrow a_{mid,2} ... \rightsquigarrow a_{mid,n} \rightsquigarrow a_{u,n+1} \rightsquigarrow a_l$.

Therefore $Live_{actor}(G, R, U) \subseteq Live_{object}(G', R')$.

Now, consider the other case that $Live_{object}(G', R') \subseteq Live_{actor}(G, R, U)$.

For any a_r and a_l, $a_r \in R' \wedge a_l \in V' \wedge a_r \rightsquigarrow a_l$ in G',

let $\{\overline{a_{x,1} a_{u,1}}, \overline{a_{x,2} a_{u,2}}, ..., \overline{a_{x,n} a_{u,n}}\} \subseteq (E' - E)$ and be part of $a_r \rightsquigarrow a_l$ in G', such that $(a_r \rightsquigarrow a_{x,1} \wedge \overline{a_{x,1} a_{u,1}} \in (E' - E)) \wedge (a_{u,1} \rightsquigarrow a_{x,2} \wedge \overline{a_{x,2} a_{u,2}} \in (E' - E)) \wedge ...$ $(a_{u,n-1} \rightsquigarrow a_{x,n} \wedge \overline{a_{x,n} a_{u,n}} \in (E' - E)) \wedge a_{u,n} \rightsquigarrow a_l$.

Since a reference $\overline{a_q a_p}$ in $(E' - E)$ implies that $a_p \in (U \cup R) \wedge a_q \in V \wedge a_p \rightsquigarrow a_q$, we get $a_r \rightsquigarrow a_r \wedge (a_r \rightsquigarrow a_{x,1} \wedge a_{u,1} \rightsquigarrow a_{x,1}) \wedge (a_{u,1} \rightsquigarrow a_{x,2} \wedge a_{u,2} \rightsquigarrow a_{x,2}) \wedge ... \wedge (a_{u,n-1} \rightsquigarrow a_{x,n} \wedge a_{u,n} \rightsquigarrow a_{x,n}) \wedge a_{u,n} \rightsquigarrow a_l \Longrightarrow$

$a_r \leftrightsquigarrow a_{x,1} \wedge a_{x,1} \leftrightsquigarrow a_{x,2} \wedge ... \wedge a_{x,n-1} \leftrightsquigarrow a_{x,n} \wedge a_{x,n} \leftrightsquigarrow a_l \Longrightarrow$

$a_r \leftrightsquigarrow^* a_l$.

Therefore $Live_{object}(G', R') \subseteq Live_{actor}(G, R, U)$. □

Lemma 3.1.8 Backward reachability to the unblocked/root actors in the newly transformed graph.
Let the actor reference graph be $G = \langle V, E \rangle$, R be the set of roots, and U be the set of unblocked actors. Let $E' = E \cup \{\overline{a_q a_p} | a_u \in (U \cup R) \wedge \overline{a_p a_q} \in E \wedge a_u \rightsquigarrow a_p\}$, and $G' = \langle V, E' \rangle$ and $R' = R$. Then in G',

$$\forall a_x, a_y : a_x \in (U \cup R) \wedge a_x \rightsquigarrow a_y \text{ in } G \Longrightarrow a_x \in (U \cup R) \wedge a_y \rightsquigarrow a_x \text{ in } G'.$$

Proof. The lemma can be proven by induction. Let the minimal number of a reference set (a path) to make $a_x \rightsquigarrow a_y$ in G' be n.
Basis: Let $n = 0$. $a_x \in (U \cup R) \wedge a_x \rightsquigarrow a_x$ which is true.
Induction step: Assume the lemma is true for $n \leq k$. Then, consider $n = k + 1$.
Let $a_x \in (U \cup R) \wedge a_x \rightsquigarrow a_z \wedge \overline{a_z a_y}$ in G. Then we know $\overline{a_y a_z} \in E'$ by definition. By the induction step, $a_x \in (U \cup R) \wedge a_z \rightsquigarrow a_x$ in G'. Therefore, $a_x \in (U \cup R) \wedge a_z \rightsquigarrow a_x \wedge \overline{a_y a_z} \in E'$ in G' implies $a_x \in (U \cup R) \wedge a_y \rightsquigarrow a_x$ in G'. □

Lemma 3.1.9 $Live_{object}(G', R') \subseteq Live_{actor}(G, R, U)$.
Let the actor reference graph be $G = \langle V, E \rangle$, R be the set of roots, and U be the set of unblocked actors. Let $E' = E \cup \{\overline{a_q a_p} \mid \exists a_u : a_u \in (U \cup R) \wedge \overline{a_p a_q} \in E \wedge a_u \rightsquigarrow a_p\}$, and $G' = \langle V, E' \rangle$ and $R' = R$. Then in G',

$$\forall a_x, a_y : a_x \in (R') \wedge a_x \rightsquigarrow a_y \text{ at } G' \Longrightarrow a_x \in (U \cup R) \wedge a_y \leftrightsquigarrow^* a_x \text{ at } G,$$

that is, $Live_{object}(G', R') \subseteq Live_{actor}(G, R, U)$.

Proof. The lemma can be proven by induction.
Let $Path = \{\overline{a_x a_1}, \overline{a_1 a_2}, ..., \overline{a_{m-1} a_m}, \overline{a_m a_y}\} \subseteq E'$ such that $a_x \rightsquigarrow a_y$ and $|Path \cap (E' - E)| = n$.
Basis: Let $n = 0$. This implies that $Path \nsubseteq (E' - E)$. Since $(Path \subseteq E') \wedge (Path \nsubseteq (E' - E))$, we know $Path \subseteq E$. Therefore, $a_x \rightsquigarrow a_y$ at G $\Longrightarrow a_x \rightsquigarrow a_x \wedge a_x \rightsquigarrow a_y$ at G $\Longrightarrow a_x \leftrightsquigarrow^* a_y$ at G.

Induction step: Assume the lemma is true for $\exists k : 0 \leq n \leq k$. Then, consider $n = k + 1$.

$Path$ can be re-defined as $Path_1 \cup \{\overline{a_v a_{v+1}}\} \cup Path_2$, where $Path_1 = \{\overline{a_x a_1}, \overline{a_1 a_2}, ..., \overline{a_{v-1} a_v}\}$, $\overline{a_v a_{v+1}} \in E'$, and $Path_2 = \{\overline{a_{v+1} a_{v+2}}, ..., \overline{a_m a_y}\}$, such that
$(|Path_1 \cap (E' - E)| = k) \wedge (\overline{a_v a_{v+1}} \in (E' - E)) \wedge (|Path_2 \cap (E' - E)| = 0)$.
First, consider $\overline{a_v a_{v+1}}$:
$(\overline{a_v a_{v+1}} \in (E' - E)) \implies$
$\exists a_u : a_u \in (U \cup R) \wedge (a_u \rightsquigarrow a_{v+1}$ at G$) \wedge \overline{a_{v+1} a_v} \in E \implies$
$\exists a_u : a_u \in (U \cup R) \wedge a_u \rightsquigarrow a_v$ at G.
Second, consider $a_{v+1} \rightsquigarrow a_y$ and $\exists a_u : a_u \in (U \cup R) \wedge a_u \rightsquigarrow a_v$ at G:
$a_{v+1} \rightsquigarrow a_y$ and $\exists a_u : a_u \in (U \cup R) \wedge a_u \rightsquigarrow a_v$ at G $\implies$
$\exists a_u : a_u \rightsquigarrow a_v \wedge a_u \rightsquigarrow a_y$ at G $\implies$
$a_v \leftrightsquigarrow a_y$ at G.
By induction hypothesis, we know $a_x \leftrightsquigarrow^* a_v$ at G. Because $a_x \leftrightsquigarrow^* a_v \wedge a_v \leftrightsquigarrow^* a_y$ at G, $a_x \leftrightsquigarrow^* a_y$ at G is true for $n = k + 1$.
Therefore, the lemma is true by induction. □

Theorem 3.1.10 Indirect back pointer transformation.
Let the actor reference graph be $G = \langle V, E \rangle$, R be the set of roots, and U be the set of unblocked actors. Let $E' = E \cup \{\overline{a_q a_p} \mid a_u \in (U \cup R) \wedge \overline{a_p a_q} \in E \wedge a_u \rightsquigarrow a_p\}$, and $G' = \langle V, E' \rangle$ and $R' = R$.

$$Live_{actor}(G, R, U) = Live_{object}(G', R')$$

Proof. We can reuse the proof of $Live_{actor}(G, R, U) \subseteq Live_{object}(G', R')$ in Theorem 3.1.7 and Lemma 3.1.8 to prove that $Live_{actor}(G, R, U) \subseteq Live_{object}(G', R')$ in this theorem. By Lemma 3.1.9, we know $Live_{object}(G', R') \subseteq Live_{actor}(G, R, U)$. As a result, $Live_{actor}(G, R, U) = Live_{object}(G', R')$. □

CHAPTER 4
A Distributed Actor Garbage Collection Mechanism for Mobile Actor Systems

The chapter describes the proposed distributed mobile actor garbage collection mechanism. The main purpose of our approach is to be non-intrusive to users' applications. There are two parts in this chapter. The first part is the pseudo-root approach which is an asynchronous, non-FIFO reference listing algorithm to support hierarchical garbage collection. The second part is the distributed snapshot algorithm, which supports partial global garbage collection, that is, it requires coordination of only a subset of the computing hosts. Formalized computing models and correctness proofs can be found in Chapter 5 and Chapter 6.

4.1 The Pseudo-Root Approach

Together with *reference listing*, the core concept of the pseudo-root approach is to integrate message delivery and reference passing into the reference graph representation — using *sender pseudo-roots* and *protected references*. It thus enables the use of unordered, asynchronous communication.

The pseudo-root approach is mainly devised to support actor programming languages, which abide by the *live unblocked actor principle* — a principle which says every unblocked actor should be treated as a live actor. Nevertheless the pseudo-root approach can also be used by traditional actor marking algorithms without the assumption of the live unblocked actor principle.

4.1.1 The Live Unblocked Actor Principle

Without program analysis techniques, the ability of an actor to access resources provided by an actor-oriented programming language implies explicit reference creation to access service actors. The ability to access local service actors (e.g. the standard output) and explicit reference creation to public service actors make the following statement true: "*every actor has persistent references to root actors*". This

statement is important because it changes the meaning of actor garbage collection, making actor garbage collection similar to passive object garbage collection. It leads to the *live unblocked actor principle*, which says every unblocked actor is live. The live unblocked actor principle is easy to prove. Since each unblocked actor is: 1) an inverse acquaintance of the root actors and 2) defined as potentially live, it is live according to the definition of actor garbage collection.

With the live unblocked actor principle, every unblocked actor can be viewed as a root. Liveness of blocked actors depends on the transitive reachability from unblocked actors and root actors. Without considering in-transit messages, a blocked actor, which is transitively reachable from an unblocked actor or a root actor, is defined as potentially live. With persistent root references, such potentially live, blocked actors are live because they are inverse acquaintances of some root actors. This idea leads to the core concept of *pseudo-root* actor garbage collection.

4.1.2 Pseudo-Root Actor Garbage Collection

It is impossible to ignore in-transit messages in a distributed system and correctly perform actor garbage collection simply based on the actor reference graph. As a consequence, we introduce the concept of sender pseudo-root actors and protected references for actor garbage collection. The pseudo-root actor garbage collection starts actor garbage collection by identifying some live (not necessarily root) or even garbage actors as pseudo-roots. There are three kinds of pseudo-root actors: 1) root actors, 2) unblocked actors, and 3) *sender pseudo-root actors*. The sender pseudo-root actor refers to an actor which has sent a message which not yet been received. The goal of sender pseudo-roots is to prevent erroneous garbage collection of actors, either targets of in-transit messages or whose references are part of in-transit messages. A sender pseudo-root always contains at least one protected reference — a reference that has been used to deliver messages which are currently in transit, or a reference to represent an actor referenced by an in-transit message — which we call an *in-transit reference*. A protected reference cannot be deleted until the message sender knows the in-transit messages have been received correctly.

Asynchronous communication introduces the following problem (see the left

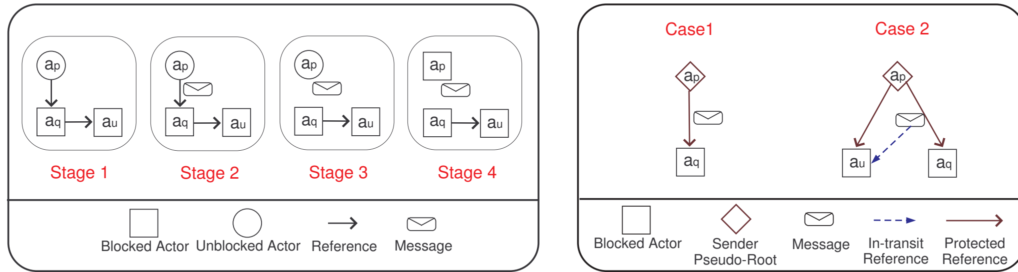

Figure 4.1: The left side of the figure shows a possible race condition of mutation and message passing. The right side of the figure illustrates both kinds of sender pseudo-root actors.

side of Figure 4.1): application messages from Actor a_p to Actor a_q can be in transit, but the reference held by Actor a_p can be removed. Stage 3 shows that Actor a_q and a_u are likely to be erroneously reclaimed, while Stage 4 shows that all of the actors are possibly erroneously reclaimed. Our solution is to temporarily keep the reference to Actor a_q undeleted and identify Actor a_p as live (Case 1 of the right side of Figure 4.1). This approach guarantees liveness of Actor a_q by tracing from Actor a_p. Actor a_p is named the *sender pseudo-root* because it has an in-transit message to Actor a_q and it is not a real root. Furthermore, it can be garbage but cannot be collected. The reference from a_p to a_q is protected and a_p is considered live until a_p knows that the in-transit message is delivered.

To prevent erroneous garbage collection, actors pointed by in-transit references must unconditionally remain live until the receiver receives the message. A similar solution can be re-used to guarantee the liveness of the referenced actor: the sender becomes a sender pseudo-root and keeps the reference to the referenced actor undeleted (Case 2).

Let us assume the live unblocked actor principle. Using pseudo-roots, *the persistent references to roots can be ignored*. Figure 4.2 illustrates an example of the mapping of pseudo-root actor garbage collection. We can now safely ignore: 1) dynamic creation of references to public services and 2) persistent references to local services. Notice that dynamic creation of references to public services implies persistent references to the root set.

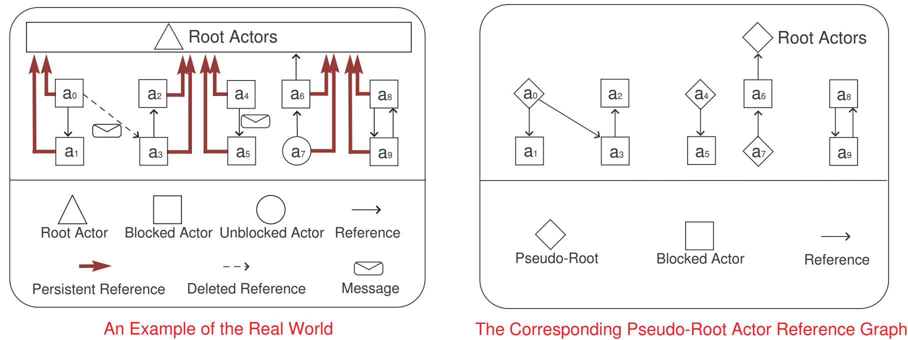

Figure 4.2: An example of pseudo-root actor garbage collection which maps the real state of the given system to a pseudo-root actor reference graph.

Take Figure 4.2 for instance. Actors a_1, a_2, and a_3 are live because they are transitively reachable from Actor a_0, which is directly identified as live by the live unblocked actor principle. Similarly, Actors a_4, a_5, a_7, and a_8 are live as well. The persistent references to the root set can be ignored in this case. Actors a_8 and a_9 are garbage because they cannot possibly become unblocked by receiving messages.

4.1.3 Imprecise Inverse Reference Listing

In a distributed environment, an inter-node referenced actor must be considered live from the perspective of local garbage collection because it can possibly receive a message from a remote actor. To know whether an actor is inter-node referenced, each actor should maintain inverse references to indicate if it is inter-node referenced. This approach is usually called *reference listing*. Maintaining precise inverse references in an asynchronous way is performance-expensive. Fortunately, imprecise inverse references are acceptable if all inter-node referenced actors can be identified as live — an inter-node referenced actor can be thought of as a new kind of pseudo-root actor (*the global pseudo-root*), or can be guaranteed to be transitively reachable from some local pseudo-root actor to ensure its liveness.

4.1.4 Actor Garbage Collection without the Live Unblocked Actor Principle

Without the live unblocked actor principle, the pseudo-root approach still supports actor garbage collection in a distributed environment by changing two definitions:

- Sender pseudo-roots should be considered as unblocked actors.
- Global pseudo-roots should be considered as roots, and thus we call them *global roots*. There are two kinds of global roots. The first kind of global root actor, namely *the unblocked-reachable global root*, is transitively reachable from an unblocked actor and has a reference pointing to a remote (or migrating) actor. It must be considered live because it can possibly send a message to a remote root actor. The second type of actor, *the remotely referenced global root*, has an inverse reference pointing to a remote actor. The pseudo-root approach cannot directly identify the unblocked-reachable global roots, which must be handled by an actor marking algorithm.

The pseudo-root approach in actor garbage collection also guarantees that if an actor is remotely referenced, either it may talk to an unblocked-reachable global root, or it is a remotely referenced global root. Once an actor marking algorithm follows the new definitions to identify live actors, all local garbage can be reclaimed, including the actors which are potentially live and yet garbage.

4.1.5 Implementation of the Pseudo-Root Approach

To implement the pseudo-root approach, we propose the *actor garbage detection protocol*. The actor garbage detection protocol, implemented as part of the SALSA programming language [100, 109], consists of four sub-protocols — the *asynchronous ACK protocol*, the *reference passing protocol*, the *migration protocol*, and the *reference deletion protocol*. Messages are divided into two categories — the *application messages* which require asynchronous acknowledgements, and the *system messages* that not always require any acknowledgement.

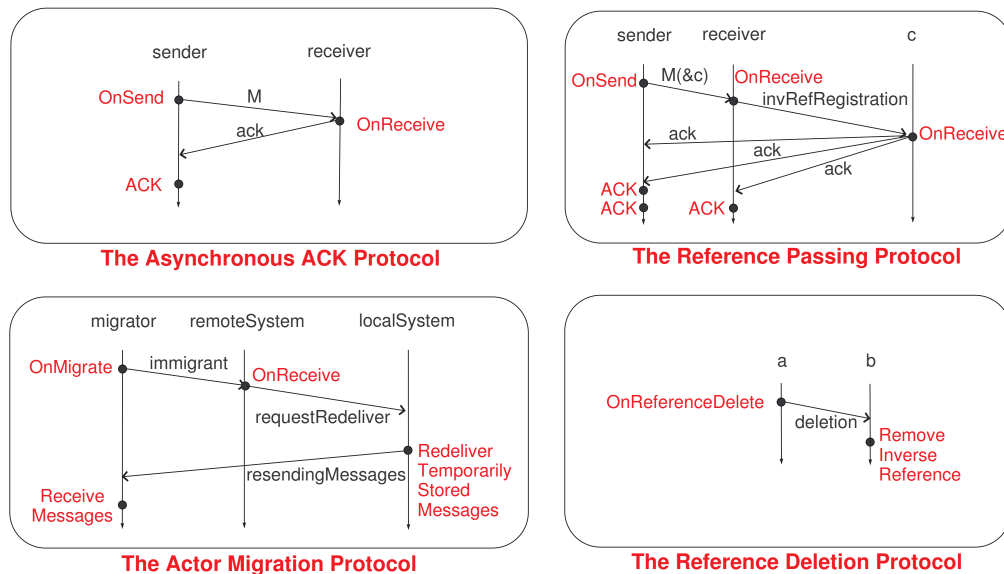

Figure 4.3: The actor garbage detection sub-protocols.

The Asynchronous ACK Protocol

The *asynchronous ACK protocol* is designed to help identifying sender pseudo-roots. Each reference maintains a counter, the *expected acknowledgement count.* A reference can be deleted only if its expected acknowledgement count is zero. An actor is a sender pseudo-root if the total expected acknowledgements of its references is greater than zero. The protocol is shown in the left upper part of Figure 4.3, in which Actor `sender` sends a message to Actor `receiver`. The event handler `OnSend` is triggered when an application message is sent; the event handler `OnReceive` is invoked when a message is received. If a message to receive requires an acknowledgement, the event handler `OnReceive` will generate an acknowledgement to the message sender. The message handler `ACK` is asynchronously executed by an actor to decrease the expected acknowledgement count of the reference to Actor `receiver` held by Actor `sender`. With the asynchronous ACK protocol, the garbage collector can identify sender pseudo-roots and protected references.

- A *sender pseudo-root* is one whose total expected acknowledgement count of its references is greater than zero.

- A *protected reference* is one whose expected acknowledgement count is greater than zero. A protected reference cannot be deleted.

The Reference Passing Protocol

The *reference passing protocol* specifies how to build inverse references in an asynchronous manner. The protocol is shown in the right upper side of Figure 4.3. A typical scenario of reference passing is to send a message `M` containing a reference to `c`, from `sender` to `receiver`. Reference $\overline{sender\ receiver}$ and Reference $\overline{sender\ c}$ are protected at the beginning by increasing their expected acknowledgement counts. Then `sender` sends the application message `M` to `receiver`. Right after `receiver` has received the message, it generates a special system message `invRefRegistration` to `c` to register the corresponding inverse reference of Reference $\overline{receiver\ c}$ in `c`. Requiring `invRefRegistration` to be acknowledged is to ensure that reference deletion of Reference $\overline{receiver\ c}$ always happens after `c` has built the corresponding inverse reference. Two special acknowledgements from `c` to `sender`, which can be combined into one, are then sent to decrease the counts of the protected references $\overline{sender\ c}$ and $\overline{sender\ receiver}$.

The Migration Protocol

The *migration protocol* is shown in the left lower side of Figure 4.3. Implementation of the protocol requires assistance from two special actors, `remoteSystem` at a remote computing node, and `localSystem` at the local computing node. An actor migrates by encoding itself into a message, and then delivers the message to `remoteSystem`. During this period, messages to the migrating actor are stored at `localSystem`. After migration, `localSystem` delivers the temporarily stored messages to the migrated actor asynchronously. Every migrating actor becomes a pseudo-root by increasing the expected acknowledgement count of its self reference. The migrating actor decreases the expected acknowledgement count of its self reference when it receives the temporarily stored messages. The protocol can be simply viewed as that the migrating actor sends a message to itself, and it will not receive the message until it has finished migration.

The Reference Deletion Protocol

The *reference deletion protocol* cleans corresponding inverse references of deleted references. The protocol is shown in the right lower side of Figure 4.3. A reference can be deleted if it is not protected — its expected acknowledgement count must be zero. The deletion automatically creates a system message to the actor to which the deleted reference points. The system message will trigger the corresponding inverse reference deletion.

4.2 A Non-Intrusive Distributed Snapshot Algorithm for Mobile Actor Garbage Collection

A snapshot algorithm executes in parallel with applications to obtain a consistent view of a system, also referred to as the *snapshot*. In a distributed system, active entities, such as MPI processes and actors, can send messages to affect each other, which means a snapshot algorithm must take care of both in-transit messages and each local state of the system. A snapshot must be causally consistent, but different variations of snapshot algorithms have different requirements of causal consistency. Actor garbage collection requires that no live actors can be collected (*safety*), and garbage be eventually collected (*liveness*). Sometimes the ability to collect garbage one chunk at a time (incrementality) is important when a system is large or it needs interactivity. Snapshots are used in garbage collection [102, 79, 52, 101], as well as in many other areas, such as distributed termination detection [69] and distributed checkpointing for execution rollback [17, 83]. In this section, we will introduce the problem of causal consistency, and then present a distributed snapshot algorithm to support actor garbage collection in a partial set of the computing hosts. Currently the algorithm only supports actor systems with the live unblocked actor principle. Actor systems without the live unblocked actor principle require the design of a security model. For instance, migrating actors can be treated as roots or unblocked actors depending on the security policy. Thus we leave it for future work.

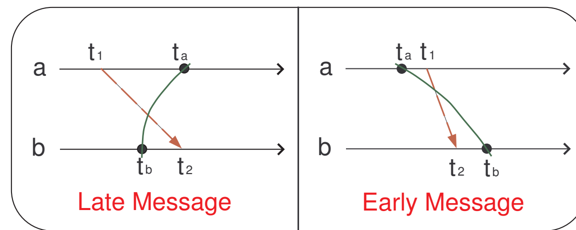

Figure 4.4: Time lines to illustrate late and early messages. At the left side, Actor a sends a message to Actor b at t_1 and then its state is recorded at t_a ($t_a > t_1$); the state of Actor b is recorded at t_b and then it receives the message at t_2 ($t_2 > t_b$). At the right side, the state of Actor a is recorded at t_a and then it sends a message to Actor b at t_1 ($t_1 > t_a$); Actor b receives it at t_2 and then its state is recorded at t_b ($t_b > t_2$).

4.2.1 Causal Consistency

A snapshot algorithm must guarantee causal consistency of the obtained snapshot. The problem of causal consistency can be expressed by the order of message sending, message reception, and local state logging. Let Actor a send an application message at time t_1 to a remote actor b, and Actor b receive the message at time t_2. Let t_a and t_b be the time points when a snapshot is taken for a and b respectively. Note that $t_2 > t_1$ is always true because of the causal relationship of message sending. There are two kinds of inconsistent snapshots caused by different orders of message delivery (refer to Figure 4.4):

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

- **Early message (inconsistent message)**: If $(t_a < t_1) \wedge (t_2 < t_b)$, the message is said to be *early*. It is causally inconsistent because a message produced by a future unrecorded state of Actor a affects the recorded state of Actor b.

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

4.2.2 Non-Intrusive Distributed Snapshot

Given a non-blocking, non-FIFO reference listing algorithm such as the *pseudo-root approach* in Section 4.1, many actor garbage collection problems can be simplified:

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

2. Remotely referenced actors can be identified by using inverse references.

3. Actor garbage collection does not stop applications.

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

garbage collection phase, but cannot be detected in that phase. Any garbage collector that uses this approach cannot detect floating garbage of `G1`, but it can detect the floating garbage in the next garbage collection cycle because garbage cannot become live any longer.

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

Local State Logging

Local state logging is triggered by a global synchronization agent, which requests the local garbage collector to form a closed group of actors and then starts to monitor mutation operations on that closed group. Newly created actors are automatically excluded from the closed group; migrating or migrated actors are segregated by the local snapshot procedure. Reference deletion is not logged because we want to ensure that a live actor remains live by following the original path from the beginning to the end of local state logging. The state logging procedure for local snapshot has to ensure that: 1) deleted references, including inverse references, are logged in the local snapshot, 2) migrating or migrated actors are segregated dynamically from the closed group and their acquaintances become remotely referenced. Figure 4.5 shows an example of how local state logging works. At the beginning of local state logging, Actor a is referenced by Actor c; Actor b is referenced by Actor a. Actor a and Actor b are put in a closed group for state logging. At the second stage, Actor a becomes unblocked to execute something, and the snapshot should detect the event. At the third stage, Actor a deletes Reference $\overline{ab}$. Although Actor b becomes garbage at this stage, it is live in the local snapshot because it is reachable

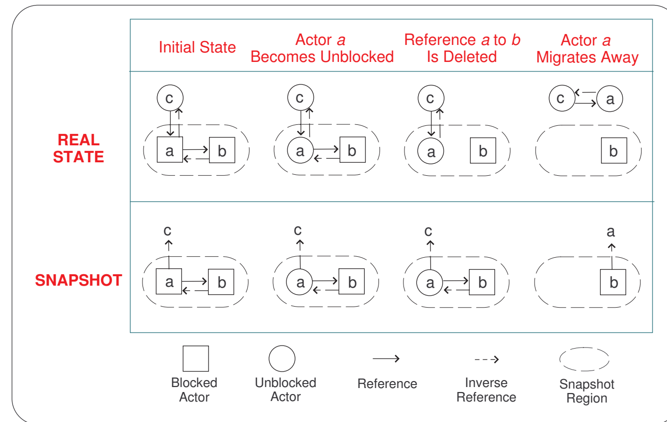

Figure 4.5: An example of local state logging. The upper part demonstrates the actor reference graph in the real world, while the lower part illustrates how local state logging works.

from a pseudo-root (unblocked) actor, Actor a. At the last stage, Actor a migrates away, and local snapshot should reflect the fact that Actor a is missing. Meanwhile, all its acquaintances should become remotely referenced because the local snapshot must not produce new garbage. At this stage, no actor in the local snapshot is garbage. Actor a is not garbage either because it does not belong to the closed group. The local state logging algorithm is modeled as a special actor which responds to a global garbage collection request from the global synchronization agent. Note that it does not stop any mutation operations, including migration.

Global Snapshot Synchronization

The global synchronization agent is devised to coordinate a meaningful global snapshot among several computing nodes. Since each computing node logs local state independently, global synchronization must be used to ensure that no early messages or migrating actors can be received before the state logging starts. This goal can be achieved by enforcing the participating local snapshots to have a common overlapping time range during local state logging. An overlapping time range also ensures that no actor can appear more than once at the participating local snapshots with the help of local state logging. Let the common overlapping time range start at time t_1 and finish later to become available for global snapshot merging. The set

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

Figure 4.6: The distributed snapshot algorithm. A meaningful global snapshot consists of the local snapshots of the computing nodes that reply 'OK'.

of garbage at each local snapshot is fixed after t_1. Our algorithm also guarantees that the set of global garbage in the global snapshot, combined by the participating local snapshots, is fixed after t_1. To prevent some kind of temporary failures from stopping global garbage collection, the synchronization agent can use a time-out to keep the global snapshot going. The pseudo-code is shown in Figure 4.6 and Figure 4.7.

4.2.3 Discussion on Correctness

In any distributed system, the two most common scenarios to erroneously reclaim live actors are 1) the race between reference creation (passing) and reference deletion, and 2) detection of in-transit messages and references. These challenges are handled by the pseudo-root approach because 1) every in-transit reference and message has a representative reference in a pseudo-root actor (the message sender), and 2) the safety property of referenced actors.

Reference and pseudo-root state preservation by the snapshot algorithm guarantees the maximum reachability from pseudo-root actors and thus ensures safety of forward DFS or BFS marking. Early messages cannot happen before taking local state monitoring because the enforcement of the overlapping time range — every participating computing host has to start local state monitoring before the time

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

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

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

Figure 4.7: The local snapshot actor.

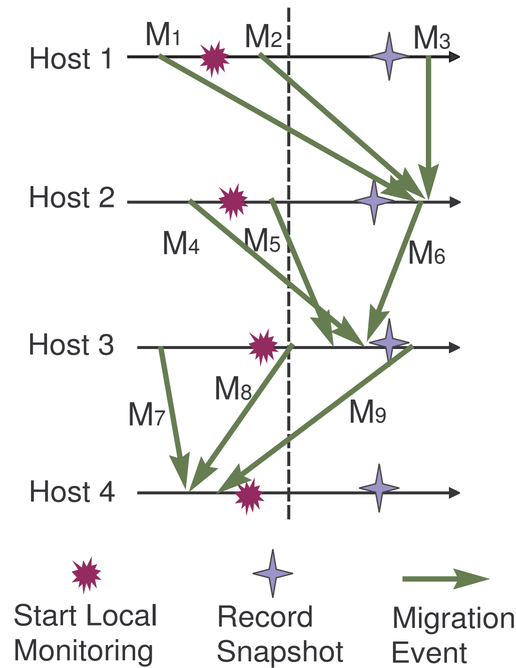

Figure 4.8: Nine possible migration detection scenarios.

that the first participating computing host finishes taking a local snapshot.

Snapshots obtained by the distributed snapshot algorithm are composable. With the overlapping time range, a path from any pseudo-root actor to any logged actor must exist in the merged global snapshot. To avoid inconsistency from duplicate actors, the snapshot algorithm must ensure that each logged actor is unique, which is also guaranteed by the enforcement of the overlapping time range. Figure 4.8 illustrates nine possible scenarios of migration events among different hosts. Events M_3 and M_7 are trivial. Events M_4, M_5, and M_6 do not create duplicate actors because local state monitoring does not add new actors. In either Event M_2 or Event M_8, the migrating actor cannot be detected by its original host. In Event M_1, no actor can be detected because the actor is migrating (in transit) while taking snapshot. Event M_9 is not possible because the global synchronization mechanism enforces a common overlapping time range. Therefore, the snapshot algorithm is safe.

The liveness property of the distributed snapshot algorithm is conditional —

each blocked actor must be selected for local state monitoring. The reason is that the set of blocked actors is a superset of garbage actors. With a periodical garbage collection event, garbage will be eventually reclaimed.

CHAPTER 5
Correctness of the Pseudo-Root Approach

In this chapter, we define the model of the pseudo-root approach, and then prove the safety and liveness properties of the pseudo-root approach.

5.1 The Computing Model of the Pseudo-Root Approach

To implement the concept of protected references, we introduce the data structure of actor references, which consists of three elements — the source and target addresses to describe which actor holds a reference to another actor, a counter to track the expected incoming acknowledgements, and a boolean variable to indicate if a reference has been deleted at the application level. Then we define a set of messages along with their explanations. These notations help define the actor system with the pseudo-root approach, which is an abstract machine with a given initial state, identified by a set of actor names, a mapping relations from actor references to their meta-data, a set of inverse references, and a set of in-transit messages.

Definition 5.1.1. *The meta-data map of actor references.*
A meta-data map (function) of actor references is described as

$$R \equiv \{(\overline{a_0 a_1} \mapsto \langle n_0, d_0 \rangle), (\overline{a_2 a_3} \mapsto \langle n_1, d_1 \rangle), ...\}.$$

Its domain, $dom(R) = \{\overline{a_0 a_1}, \overline{a_2 a_3}, ...\}$, is a set of actor references while its range, $\{\langle n, d \rangle \mid n \in \mathbb{N} \wedge d \in \{true, false\}\}$, is the meta-data of the actor reference, where

- *n is an integer to keep track of expected incoming acknowledgements, and*
- *$d \in \{true, false\}$, where $false$ denotes a normal reference, and $true$ denotes a reference deleted at the application level.*

Definition 5.1.2. *The extended meta-data map of actor references.*
For a given meta-data map of references R, $R' = R\{(\overline{a_i a_j} \mapsto \langle n, d \rangle)\}$ is defined as the extended map, such that $R'(\overline{a_i a_j}) = \langle n, d \rangle$ and $R'(x) = R(x)$ for $x \neq \overline{a_i a_j}$.

Definition 5.1.3. *Message*
Let x be a unique id. A message

msg	$\equiv$	$m_x\langle a_i, a_j\rangle$	*(Regular message from a_i to a_j.)*
	$\mid$	$mack_x\langle a_i, a_j\rangle$	*(Acknowledgement for Reference $\overline{a_i a_j}$.)*
	$\mid$	$mr_x\langle a_h, a_i, a_j\rangle$	*(Reference passing, from a_h to a_i with a reference to a_j.)*
	$\mid$	$mir_x\langle a_h, a_i, a_j\rangle$	*(Inverse reference registration, initialized by a_h to register $\overline{a_i a_j}$.)*
	$\mid$	$md_x\langle a_i, a_j\rangle$	*(Deletion message of the inverse reference of $\overline{a_i a_j}$.)*
	$\mid$	$mm_x\langle a_i\rangle$	*(Migration of a_i.)*

Definition 5.1.4. *Actor system configuration*
An actor system configuration

$$S \equiv \langle A, R, IR, M\rangle,$$

is a 4-tuple where

- *A is the set of actor names, $\{a_0, a_1, ...\}$.*
- *R is the map from actor references to their meta-data.*
- *IR is the set of inverse reference, $\{\overline{a_0 a_1}, \overline{a_2 a_3}, ...\}$ [4].*
- *M is the set of messages, $\{msg_0, msg_1, ...\}$.*

The initial state is defined as

$$S_0 \equiv \langle \{a_{init}\}, \{(\overline{a_{init} a_{init}} \mapsto \langle 0, false\rangle)\}, \{\overline{a_{init} a_{init}}\}, \emptyset\rangle.$$

A state of the actor system can change to another state by transition rules. Following the actor model, the transition rules of the proposed abstract machine can be classified into two categories. The first type consists of *the spontaneous transitions* by unblocked/root actors, including actor creation, reference passing, actor-exit (to migrate out), reference deletion at the application or the GC level,

[4] $\overline{a_0 a_1} \in IR$ implies that a_1 has an inverse reference to a_0.

and message sending. The other type of transitions consists of *the message-driven transitions*, including actor-enter (to migrate into), reference reception, inverse reference registration, acknowledgement, and inverse reference deletion. They are the consequent transitions triggered by the spontaneous transitions. We define two different reference deletion transition rules because of the idea of protected references, in which a deleted protected reference must remain visible to garbage collectors but invisible to applications — the application-level reference deletion transition makes the reference invisible to the application, while the GC-level reference deletion transition deletes the reference physically and will trigger the deletion of the corresponding inverse reference. The name of a transition rule follows the format, $op(p_1, p_2, ...)$, where op is the identification of the transition rule and $p_1, p_2, ...$ are parameters. The transition rules are shown below:

Definition 5.1.5. *Transitive relationship on actor system configurations. The transition notation, $\rightarrow$, is defined as:*

Spontaneous transitions:

- $AC(a_i, a_j)$*:* Actor creation of a_j by a_i.
  $\langle A, R, IR, M\rangle \xrightarrow{AC(a_i,a_j)}$
  $\langle A \cup \{a_j\}, R\{(\overline{a_i a_j} \mapsto \langle 0, false\rangle), (\overline{a_j a_j} \mapsto \langle 0, false\rangle)\}, IR \cup \{\overline{a_i a_j}, \overline{a_j a_j}\}, M\rangle$,
  where a_j is fresh $\wedge$ $a_i \in A$.
- $RC1(a_h, a_i, a_j)$*:* Reference passing from a_h to a_i with a reference to a_j.
  $\langle A, R\{(\overline{a_h a_i} \mapsto \langle n_0, false\rangle), (\overline{a_h a_j} \mapsto \langle n_1, false\rangle)\}, IR, M\rangle \xrightarrow{RC1(a_h,a_i,a_j)}$
  $\langle A, R\{(\overline{a_h a_i} \mapsto \langle n_0+1, false\rangle), (\overline{a_h a_j} \mapsto \langle n_1+1, false\rangle)\}, IR, M \cup \{mr_x\langle a_h, a_i, a_j\rangle\}\rangle$,
  where x is fresh $\wedge$ $\{a_h, a_i, a_j\} \subseteq A$.
- $MM1(a_i)$*:* Actor-exit of a_i.
  $\langle A, R\{(\overline{a_i a_i} \mapsto \langle n, false\rangle)\}, IR, M\rangle \xrightarrow{MM1(a_i)}$
  $\langle A - \{a_i\}, R\{(\overline{a_i a_i} \mapsto \langle n+1, false\rangle)\}, IR, M \cup \{mm_{x1}\langle a_i\rangle, mack_{x2}\langle a_i, a_i\rangle\}\rangle$,
  where $x1$ and $x2$ are fresh $\wedge$ $a_i \in A$.

- $MS1(a_i, a_j)$*:* Message sending from a_i to a_j.
  $\langle A, R\{(\overline{a_i a_j} \mapsto \langle n, false\rangle)\}, IR, M\rangle \xrightarrow{MS1(a_i,a_j)}$
  $\langle A, R\{(\overline{a_i a_j} \mapsto \langle n+1, false\rangle)\}, IR, M \cup \{m_x\langle a_i, a_j\rangle\}\rangle$,
  where x *is fresh* $\wedge$ $\{a_i, a_j\} \subseteq A$.

- $RDA(a_i, a_j)$*:* Application-level reference deletion of a_j at a_i.
  $\langle A, R\{(\overline{a_i a_j} \mapsto \langle n, false\rangle)\}, IR, M\rangle \xrightarrow{RDA(a_i,a_j)}$
  $\langle A, R\{(\overline{a_i a_j} \mapsto \langle n, true\rangle)\}, IR, M\rangle$,
  where $\{a_i, a_j\} \subseteq A \wedge a_i \neq a_j$. [5]

- $RD1(a_i, a_j)$*:* GC-level reference deletion of a_j at a_i.
  $\langle A, R\{(\overline{a_i a_j} \mapsto \langle 0, true\rangle)\}, IR, M\rangle \xrightarrow{RD1(a_i,a_j)}$
  $\langle A, R, IR, M \cup \{md_x\langle a_i, a_j\rangle\}\rangle$,
  where x *is fresh* $\wedge$ $\{a_i, a_j\} \subseteq A$. [6]

Message-driven transitions:

- $RC2(a_h, a_i, a_j)$*:* Reference reception of a_j by a_i, initialized by a_h.
  $\langle A, R, IR, M \cup \{mr_x\langle a_h, a_i, a_j\rangle\}\rangle \xrightarrow{RC2(a_h,a_i,a_j)}$
  $\langle A, R\{(\overline{a_i a_j} \mapsto \langle 1, false\rangle)\}, IR, M \cup \{mir_{x1}\langle a_h, a_i, a_j\rangle\}\rangle$,
  where $\overline{a_i a_j}$ *and* $x1$ *are fresh* $\wedge$ $\{a_h, a_i, a_j\} \subseteq A$.

- $RC3(a_h, a_i, a_j)$*:* Inverse reference registration at a_j from a_i, initialized by a_h.
  $\langle A, R, IR, M \cup \{mir_x\langle a_h, a_i, a_j\rangle\}\rangle \xrightarrow{RC3(a_h,a_i,a_j)}$
  $\langle A, R, IR \cup \{\overline{a_i a_j}\}, M \cup \{mack_{x1}\langle a_i, a_j\rangle, mack_{x2}\langle a_h, a_i\rangle, mack_{x3}\langle a_h, a_j\rangle\}\rangle$,
  where $x1$*,* $x2$*, and* $x3$ *are fresh* $\wedge$ $\{a_h, a_i, a_j\} \subseteq A$.

- $MM2(a_i)$*:* Actor-enter of a_i.
  $\langle A, R, IR, M \cup \{mm_{x1}\langle a_i\rangle\}\rangle \xrightarrow{MM2(a_i)}$
  $\langle A \cup \{a_i\}, R, IR, M\rangle$.

[5] We assume an actor always keeps a reference to itself and thus self-reference deletion is impossible.

[6] GC-level reference deletion only happens when the reference to delete is removed by the application and no longer protected.

- $MS2(a_i, a_j)$*:* Message reception of a_j from a_i.
  $\langle A, R, IR, M \cup \{m_x\langle a_i, a_j\rangle\}\rangle \xrightarrow{MS2(a_i,a_j)}$
  $\langle A, R, IR, M \cup \{mack_{x1}\langle a_i, a_j\rangle\}\rangle$,
  where $x1$ *is fresh* $\wedge \{a_i, a_j\} \subseteq A$.

- $RD2(a_i, a_j)$*:* Inverse reference deletion of $\overline{a_i a_j}$.
  $\langle A, R, IR \cup \{\overline{a_i a_j}\}, M \cup \{md_x\langle a_i, a_j\rangle\}\rangle \xrightarrow{RD2(a_i,a_j)}$
  $\langle A, R, IR, M\rangle$,
  where $\{a_i, a_j\} \subseteq A$.

- $ACK(a_i, a_j)$*:* Decreasing expected acknowledgements for $\overline{a_i a_j}$.
  $\langle A, R\{(\overline{a_i a_j} \mapsto \langle n, d\rangle)\}, IR, M \cup \{mack_x\langle a_i, a_j\rangle\}\rangle \xrightarrow{ACK(a_i,a_j)}$
  $\langle A, R\{(\overline{a_i a_j} \mapsto \langle n-1, d\rangle)\}, IR, M\rangle$,
  where $d \in \{true, false\} \wedge \{a_i, a_j\} \subseteq A$.

The pseudo-root actors play important roles in the pseudo-root approach since they are the starting point to identify live actors. Unblocked actors and root actors can be identified easily by garbage collectors, while the sender pseudo-root actors and the global pseudo-root actors are implementation dependent. As a consequence, definitions for the sender pseudo-root actors and the global pseudo-root actors are mandatory. We first define sender pseudo-roots, and we defer the definition of global pseudo-root actors (Definition 5.3.2) until we define the partial view of an actor system (Definition 5.3.1).

Definition 5.1.6. *Sender pseudo-root actors.*
An actor, a_p*, is a sender pseudo-root actor in an actor system* $\langle A, R, IR, M\rangle$ *if and only if*

$$\exists a_q : (R(\overline{a_p a_q}) = \langle n, d\rangle \wedge n > 0).$$

5.2 Safety and Liveness Properties of the Pseudo-Root Approach

The discrete timeline of the actor system can be identified by the order of executed transitions (events) because only transitions can change the state of the

actor system. The timeline helps chronological reasoning about an actor system. It is defined by a set of ordered time points:

Definition 5.2.1. *Timeline of an actor system.*
A timeline of a given actor system consists of time points, where a time point, t, *of the timeline of the given actor system S, is the time period from the time that the t_{th} transition occurs and right before the $(t+1)_{th}$ transition happens. The original point of the timeline is* 0, *representing the time period when the initial state S_0 exists.*

The computing model of the pseudo-root approach maintains *paired references* in most cases, which means an actor reference usually has a corresponding inverse reference. Under some circumstances, some references exist without their corresponding inverse references or some inverse references exist without their corresponding actor references. However, the pseudo-root approach guarantees the following property all the time — *if an actor a_q is pointed by an unpaired actor reference, then there exists a sender pseudo-root actor a_p which has a paired reference pointing to a_q.* To formally describe the property, we introduce the following definitions:

Definition 5.2.2. *Paired references, unpaired references, and unpaired inverse references of an actor.*
Let $S = \langle A, R, IR, M \rangle$ be the state of the actor system at time point t. Let $a_q \in A$.

- *Paired references to a_q at time t:*
  $PRS(S, a_q) \equiv \{\overline{a_p a_q} \mid \overline{a_p a_q} \in dom(R) \wedge \overline{a_p a_q} \in IR\}$.
- *Unpaired references to a_q at time t:*
  $URS(S, a_q) \equiv \{\overline{a_p a_q} \mid \overline{a_p a_q} \in dom(R) \wedge \overline{a_p a_q} \notin IR\}$.
- *Unpaired inverse references from a_q at time t:*
  $UIRS(S, a_q) \equiv \{\overline{a_p a_q} \mid \overline{a_p a_q} \notin dom(R) \wedge \overline{a_p a_q} \in IR\}$.

Definition 5.2.3. *Inverse reference registration messages and inverse reference deletion messages of an actor.*
Let $S = \langle A, R, IR, M \rangle$ be the state of the actor system at time point t. Let $a_q \in A$.

- *Inverse reference registration messages for a_q at time t:*
  $MIRS(S, a_q) \equiv \{mir_x\langle a_{p1}, a_{p2}, a_q\rangle \mid mir_x\langle a_{p1}, a_{p2}, a_q\rangle \in M\}$.
- *Inverse reference deletion messages for a_q at time t:*
  $MDS(S, a_q) \equiv \{md_x\langle a_p, a_q\rangle \mid md_x\langle a_p, a_q\rangle \in M\}$.

There are some invariants of the pseudo-root approach which can be used for theorem proofs. Lemma 5.2.4 says that the total number of $UIRS(S, a_q)$ is always equal to the total number of $MDS(S, a_q)$ which will be used in the liveness property proof.

Lemma 5.2.4. *Let $S = \langle A, R, IR, M\rangle$ be an actor system configuration and $a_q \in A$. In the actor system, the following equation is true:*

$$|UIRS(S, a_q)| = |MDS(S, a_q)|.$$

Proof. The only two transitions that can affect $|UIRS(S, a_q)|$ or $|MDS(S, a_q)|$ are $RD1$ and $RD2$. $RD1$ increments $|UIRS(S, a_q)|$ and $|MDS(S, a_q)|$ by one and $RD2$ decrements $|UIRS(S, a_q)|$ and $|MDS(S, a_q)|$ by one. Thus they don't affect the equation. Because the initial state of the actor system follows the equation and no transition can change the equation, $|UIRS(S, a_q)| = |MDS(S, a_q)|$ must be true. □

Lemma 5.2.5 says that the expected acknowledgement count of an actor reference can never be decremented to a negative number. In fact, it is equal to the total number of some set of messages. The equation can be used to prove Lemma 5.2.6, which says that a reference must exist if some messages exist.

Lemma 5.2.5. *The expected acknowledgement count equation.*
Let the configuration of the actor system be $S = \langle A, R, IR, M\rangle$. Let $R(\overline{a_i a_j}) = \langle n, d\rangle$. Then

$$\begin{aligned} n = \; & |\{mr_x\langle a_i, a_m, a_j\rangle \mid mr_x\langle a_i, a_m, a_j\rangle \in M\}| \; + \\ & |\{mr_x\langle a_i, a_j, a_m\rangle \mid mr_x\langle a_i, a_j, a_m\rangle \in M\}| \; + \\ & |\{mir_x\langle a_i, a_m, a_j\rangle \mid mir_x\langle a_i, a_m, a_j\rangle \in M\}| \; + \\ & |\{mir_x\langle a_i, a_j, a_m\rangle \mid mir_x\langle a_i, a_j, a_m\rangle \in M\}| \; + \\ & |\{mir_x\langle a_m, a_i, a_j\rangle \mid mir_x\langle a_m, a_i, a_j\rangle \in M\}| \; + \\ & |\{m_x\langle a_i, a_j\rangle \mid m_x\langle a_i, a_j\rangle \in M\}| \; + \\ & |\{mack_x\langle a_i, a_j\rangle \mid mack_x\langle a_i, a_j\rangle \in M\}| \end{aligned}$$

Proof. The proof can be achieved by examining the transitions and the initial state. The initial state $\langle \{a_{init}\}, \{(\overline{a_{init}a_{init}} \mapsto \langle 0, false\rangle)\}, \{\overline{a_{init}a_{init}}\}, \emptyset\rangle$ confirms the lemma. Now, consider the transitions:

- Transition $AC(a_i, a_j)$ creates a reference with $n = 0$ and makes the right side of the equation zero.
- Transition $RC1(a_i, a_m, a_j)$, $RC1(a_i, a_j, a_m)$, $MM1(a_i)$ where $i = j$, $MS1(a_i, a_j)$, or $RC2(a_m, a_i, a_j)$ increases both sides of the equation by one.
- Transition $RC2(a_i, a_j, a_m)$, $RC2(a_i, a_m, a_j)$, $RC3(a_i, a_j, a_m)$, $RC3(a_i, a_m, a_j)$, $RC3(a_m, a_i, a_j)$, or $MS2(a_i, a_j)$ converts a message to another message described in the right side of the equation, which still follows the equation.
- Transition $ACK(a_i, a_j)$ decreases both sides of the equation by one.
- Other transitions have no effect on the equation.

$\square$

Lemma 5.2.6 says that if there exists $mr_x\langle a_i, a_m, a_j\rangle$, $mr_x\langle a_i, a_j, a_m\rangle$ (an in-transit message containing a reference), $mir_x\langle a_i, a_m, a_j\rangle$, $mir_x\langle a_i, a_j, a_m\rangle$, $mir_x\langle a_m, a_i, a_j\rangle$ (an in-transit inverse reference registration message), $m_x\langle a_i, a_j\rangle$ (an in-transit regular message), or $mack_x\langle a_i, a_j\rangle$ (an in-transit acknowledgement), then there exist a protected reference $\overline{a_ia_j}$ and a pseudo-root a_i.

Lemma 5.2.6. *Existence of actor reference.*
Let the configuration of the actor system be $S = \langle A, R, IR, M\rangle$. Let the expected acknowledgement counter equation be:

$$\begin{aligned} n = {} & |\{mr_x\langle a_i, a_m, a_j\rangle \mid mr_x\langle a_i, a_m, a_j\rangle \in M\}| + \\ & |\{mr_x\langle a_i, a_j, a_m\rangle \mid mr_x\langle a_i, a_j, a_m\rangle \in M\}| + \\ & |\{mir_x\langle a_i, a_m, a_j\rangle \mid mir_x\langle a_i, a_m, a_j\rangle \in M\}| + \\ & |\{mir_x\langle a_i, a_j, a_m\rangle \mid mir_x\langle a_i, a_j, a_m\rangle \in M\}| + \\ & |\{mir_x\langle a_m, a_i, a_j\rangle \mid mir_x\langle a_m, a_i, a_j\rangle \in M\}| + \\ & |\{m_x\langle a_i, a_j\rangle \mid m_x\langle a_i, a_j\rangle \in M\}| + \\ & |\{mack_x\langle a_i, a_j\rangle \mid mack_x\langle a_i, a_j\rangle \in M\}| \end{aligned}$$

Then the following statement is true:

Figure 5.1: Lemma 5.2.7: safe imprecise inverse references.

$$n > 0 \implies R(\overline{a_i a_j}) = \langle n, d \rangle.$$

Proof. The lemma can be proven by contradiction. Let the current time of the system be t_e. Let $\overline{a_i a_j} \notin dom(R)$ at time t_e. Since $n > 0$ at t_e, $\exists t_s : \overline{a_i a_j} \in dom(R)$ at time t_s and $\overline{a_i a_j} \notin dom(R)$ at time t_{s+1}, such that $t_s < t_e$ and t_s is the closest point to t_e (no re-creation allowed). Since the reference has been deleted and never re-created during t_s to t_e, $n = 0$ at t_s and t_{s+1}, implying no message of the right side of the equation is in transit. Consequently, $n = 0$ at t_e which contradicts the assumption. Therefore, $\overline{a_i a_j} \in dom(R)$ at time t_e. By Lemma 5.2.5, $R(\overline{a_i a_j}) = \langle n, d \rangle$ for some d and proves the lemma. □

From the perspective of the safety property of a reference listing algorithm, a referenced actor, namely a_q, must have at least one inverse reference to figure out it is referenced. The pseudo-root approach guarantees Lemma 5.2.7, which is used to prove the safety property (Theorem 5.2.8). Then we will use Lemma 5.2.4 to prove the liveness property (Theorem 5.2.9). Figure 5.1 illustrates Lemma 5.2.7.

Lemma 5.2.7. *Safe imprecise inverse references.*
Let $S = \langle A, R, IR, M \rangle$ be an actor system configuration and $\{a_p, a_q\} \subseteq A$. The following statement is true at any time:

$$\overline{a_p a_q} \in URS(S, a_q) \implies$$
$$\exists a_0, a_1, ..., a_m : \overline{a_0 a_q} \in PRS(S, a_q) \wedge$$
$$(R(\overline{a_0 a_q}) = \langle n_0, d \rangle \wedge n_0 > 0) \wedge$$
$$(mir_{xi}\langle a_i, a_{i+1}, a_q \rangle \in M \text{ where } i = 0 \text{ to } m-1 \text{ and } a_m = a_p).$$

Proof. To prove the lemma by induction, we need to build a partial order of reference creation. The first references to Actor a_q are either created by the actor creation transition or at the initial state of the actor system because Actor a_q is the initial actor. With the existence of the first references to a_q, they can be duplicated by the reference passing transition $RC1$. Then the n_{th} $RC2$ transition of a_q, $RC2(a_{w1}, a_{w2}, a_q)$, creates the n_{th} reference to a_q, where $\{a_{w1}, a_{w2}\} \subseteq A$.

Basis: Prove the lemma for $n = 1$ (the first references).

First, the initial configuration of an actor system does not have any unpaired references. Second, the actor creation transition does not create unpaired references. Therefore, the basis is true.

Induction step: Assume $\exists k : n \leq k$ the lemma is true. Now, consider a $(k+1)_{th}$ reference in any actor system.

Let $RC1(a_u, a_p, a_q)$ be the transition to initialize reference creation of $\overline{a_p a_q}$, and $RC2(a_u, a_p, a_q)$ be the transition to create $\overline{a_p a_q}$. That is, $\exists S_s, S_1, S_2, S_3$: $S_s \xrightarrow{RC1(a_u,a_p,a_q)} S_1 \rightarrow^* S_2 \xrightarrow{RC2(a_u,a_p,a_q)} S_3 \rightarrow^* S$. Let the corresponding time of S_s, S_1, S_2, S_3, and S be t_s, t_1, t_2, t_3, and t. Now, consider $\overline{a_u a_q}$ at t_s, a reference whose order is less than $k + 1$. There are two possible cases:

- $\overline{a_u a_q} \in PRS(S_s, a_q)$ at t_s: Message $mr_{x1}\langle a_u, a_p, a_q\rangle$ created by Transition $RC1(a_u, a_p, a_q)$ will not be consumed until the system transits to S_3. Let $R(\overline{a_u a_q}) = \langle n_{u,q}, d\rangle$. According to Lemma 5.2.5, $n_{u,q} \geq 0$ between t_s and t_2, which also means $RD1(a_u, a_q)$ cannot happen. $S_2 \xrightarrow{RC2(a_u,a_p,a_q)} S_3$ makes Message $mr_{x1}\langle a_u, a_p, a_q\rangle$ to be consumed and then Message $mir_{x2}\langle a_u, a_p, a_q\rangle$ to be produced, which means $n_{u,q} \geq 1$ at t_3. We know $\overline{a_p a_q} \in URS(S, a_q)$ at t, which means $mir_{x2}\langle a_u, a_p, a_q\rangle$ is still at t, implying $n_{u,q} \geq 1$ by Lemma 5.2.5. Therefore, the following property is guaranteed at t: $\overline{a_u a_q} \in PRS(S, a_q) \wedge (R(\overline{a_u a_q}) = \langle n_{u,q}, d\rangle \wedge n_{u,q} > 0) \wedge (mir_{x2}\langle a_u, a_p, a_q\rangle) \in M$.

- $\overline{a_u a_q} \in URS(S_s, a_q)$ at t_s: There are two sub-cases:

  Sub-Case 1: Assume $(mir_x\langle a_u, a_p, a_q\rangle)$ is not consumed before t, and let $R(\overline{a_u a_q}) = \langle n_{u,q}, d\rangle$. According to Lemma 5.2.6, $n_{u,q} > 0$, and thus $\overline{a_u a_q}$ is at t. According to the induction assumption, $\exists a_0, a_1, ..., a_m : \overline{a_0 a_q} \in PRS(S, a_q)$ $\wedge$ $(R(\overline{a_0 a_q}) = \langle n_0, d\rangle \wedge n_0 > 0) \wedge (mir_{xi}\langle a_i, a_{i+1}, a_q\rangle \in M$ where $i = 0$ to $m-1$

and $a_m = a_u$). We know $\overline{a_pa_q} \in URS(S, a_q)$ at t, which means $mir_{x2}\langle a_u, a_p, a_q\rangle$ is at t and proves this sub-case.

Sub-Case 2: Assume $(mir_x\langle a_u, a_p, a_q\rangle)$ has been consumed between t_3 and t, which means $\forall S_x : \overline{a_ua_q} \in PRS(S_x, a_q)$ where S_x is an actor system configuration existing on a time point from t_3 to t. Let $R(\overline{a_ua_q}) = \langle n_{u,q}, d\rangle$. Since $\overline{a_pa_q} \in URS(S, a_q)$ is at t, $mir_{x2}\langle a_u, a_p, a_q\rangle$ is at t. By Lemma 5.2.5, $n_{u,q} > 0$. By Lemma 5.2.6, $R(\overline{a_ua_q}) = \langle n_{u,q}, d\rangle$. Therefore, the following property is true at t: $\overline{a_ua_q} \in PRS(S, a_q) \wedge (R(\overline{a_ua_q}) = \langle n_{u,q}, d\rangle \wedge n_{u,q} > 0) \wedge mir_{x2}\langle a_u, a_p, a_q\rangle \in M$.

To conclude, the lemma is true by induction.

□

Theorem 5.2.8. *Safety property.*
Let $S = \langle A, R, IR, M\rangle$ be an actor system configuration and $\{a_p, a_q\} \subseteq A$. Let the current time be t. The following statement is always true:

$$\overline{a_pa_q} \in dom(R) \Longrightarrow \exists a_u : \overline{a_ua_q} \in IR$$

Proof. If $\overline{a_pa_q} \in URS(S, a_q)$ at t is true, $\exists a_0 : \overline{a_0a_q} \in PRS(S, a_q)$ is also true by Lemma 5.2.7, implying $\overline{a_0a_q} \in IR$. Otherwise, $\overline{a_pa_q} \in PRS(S, a_q)$ at t must be true, implying $\overline{a_pa_q} \in IR$ is true. Both cases confirm the theorem. □

Theorem 5.2.9. *Liveness property.*
Let $S = \langle A, R, IR, M\rangle$ be an actor system configuration. Let $a_q \in A$ such that a_q is not a pseudo-root actor at current time point t. The following statement is true at any time:

$$\nexists a_p : R(\overline{a_pa_q}) = \langle n_p, d_p\rangle \text{ at } S \Longrightarrow \exists S_{future} : S \rightarrow^* S_{future} \wedge$$
$$(|UIRS(S_{future}, a_q)| = 0 \text{ at } S_{future}).$$

Proof. Let $S \rightarrow^* S_{future}$ where the time point of S_{future} is t_{future}. Since a_q is not a pseudo-root actor, it cannot execute any spontaneous transition. Thus $\nexists a_p : \overline{a_pa_q} \in dom(R)$ is always true for any state after S. By Lemma 5.2.4, $|UIRS(S, a_q)| = |MDS(S, a_q)|$ at t. Thus there exist $|UIRS(S, a_q)|$ inverse reference deletion messages of a_q, which will trigger $|UIRS(S, a_q)|$ inverse reference deletion transitions

of a_q. With the fairness assumption of the actor model of computation, all those inverse reference deletion transitions will eventually occur at a future state S_{future}, making $|UIRS(S_{future}, a_q)| = 0$. □

5.3 The Pseudo-Root Approach Properties in a Distributed Environment

The pseudo-root approach can be used to support local marking in a distributed environment, that is, performing actor marking in a *partial view* of an actor system to identify all local garbage. With the concept of the partial view, we can define the global pseudo-root actors which are treated as roots. The pseudo-root approach guarantees Theorem 5.3.3 to be true. It says that if an actor in a partial view is referenced by another actor outside the partial view, the actor is either a global pseudo-root actor or is directly reachable from a sender pseudo-root actor in the partial view (see Figure 5.2).

Definition 5.3.1. *Partial view of an actor system.*
Let $S = \langle A, R, IR, M\rangle$, and $V = \langle A', R', IR'\rangle$. V is a partial view of S if and only if

$A' \subseteq A\ \wedge$

$(\forall a_p : a_p \in A \wedge R(\overline{a_p a_q}) = \langle n_1, d_1\rangle \implies R'(\overline{a_p a_q}) = \langle n_2, d_2\rangle)\ \wedge$

$(\forall a_q : a_q \in A \wedge \overline{a_p a_q} \in IR \implies \overline{a_p a_q} \in IR')$.

Definition 5.3.2. *Global pseudo-root actor.*
Let V be a partial view of S, where $V = \langle A', R', IR'\rangle$. Actor a_q is a global pseudo-root actor of V if and only if

$$\exists a_p : a_p \notin A' \wedge a_q \in A' \wedge \overline{a_p a_q} \in IR'.$$

Theorem 5.3.3. *One-step back-tracing safety.*
Let V be a partial view of S, where $V = \langle A', R', IR'\rangle$ and $S = \langle A, R, IR, M\rangle$.

$$\exists a_p, a_q : a_p \notin A' \wedge a_q \in A' \wedge R(\overline{a_p a_q}) = \langle n, d\rangle \implies$$
$$\exists a_w : (\overline{a_w a_q} \in IR' \wedge a_w \notin A') \vee (R'(\overline{a_w a_q}) = \langle n', d'\rangle \wedge n' > 0 \wedge a_w \in A').$$

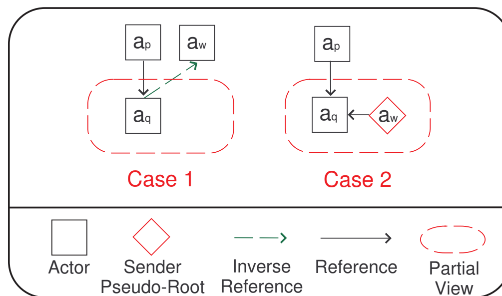

Figure 5.2: Theorem 5.3.3: one-step back-tracing safety.

Proof. Let the current time be t. Now, consider the following two cases:

- *Case 1:* Let $\overline{a_p a_q} \in PRS(S, a_q)$ at t, which means $(\overline{a_p a_q} \in IR)$. $a_q \in A'$ implies $(\overline{a_p a_q} \in IR')$.

- *Case 2:* Let $\overline{a_p a_q} \in URS(S, a_q)$ at t. By Lemma 5.2.7, $\exists a_0 : \overline{a_0 a_q} \in PRS(S, a_q)$ $\wedge$ $(\exists n_0 : R(\overline{a_0 a_q}) = \langle n_0, d \rangle \wedge n_0 > 0)$. Now condiser two sub-cases. First, $a_0 \in A'$ implies $(R'(\overline{a_0 a_q}) = \langle n_0, d \rangle \wedge n_0 > 0)$. Second, $a_0 \notin A'$ implies $\overline{a_0 a_q} \in IR'$.

Therefore, the theorem is true. ☐

5.4 Actor Garbage Collection without the Live Unblocked Actor Principle

The live unblocked actor principle is not always true in actor garbage collection. The pseudo-root approach also supports actor marking algorithms in this context. In a local host, we re-define sender pseudo-root actors as unblocked actors. Global pseudo-root actors are re-defined as global roots, consisting of the remotely referenced global roots and the unblocked-reachable global roots. Remotely referenced global roots are defined in Definition 5.3.2. Theorem 5.4.1 is the safety property of local actor marking. It says that if an actor is remotely referenced, it either has an inverse reference to figure out it is remotely referenced (the remotely

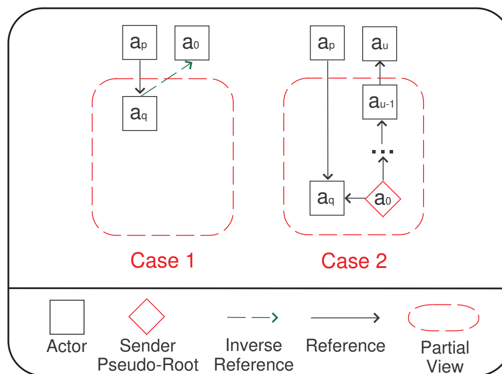

Figure 5.3: Theorem 5.4.1: one-step back-tracing and forward validating safety.

referenced global root), or it is reachable from an unblocked actor and the unblocked actor can reach an unblocked-reachable global root (see Figure 5.3).

Theorem 5.4.1. *One-step back-tracing and forward validating safety. Let V be a partial view of S, where $V = \langle A', R', IR' \rangle$ and $S = \langle A, R, IR, M \rangle$.*

$$\exists a_p, a_q : a_p \notin A' \wedge a_q \in A' \wedge R(\overline{a_p a_q}) = \langle n, d \rangle \implies$$
$$\exists a_0 : (\overline{a_0 a_q} \in IR' \wedge a_0 \notin A') \vee$$
$$(\exists a_1, a_2, ..., a_u : R'(\overline{a_0 a_q}) = \langle n_0, d_0 \rangle \wedge n_0 > 0 \wedge a_0 \in A'$$
$$\wedge\ R'(\overline{a_i a_{i+1}}) = \langle n_i, d_i \rangle \wedge a_u \notin A' \textit{ where } i = 1 \textit{ to } u - 1).$$

Proof. Let us consider two cases:

- *Case 1:* Let $\overline{a_p a_q} \in PRS(S, a_q)$, which means $(\overline{a_p a_q} \in IR)$. $a_q \in A'$ implies $(\overline{a_p a_q} \in IR')$.

- *Case 2:* Let $\overline{a_p a_q} \in URS(S, a_q)$. By Lemma 5.2.7, $\exists a_0, a_1, ..., a_m : \overline{a_0 a_q} \in PRS(S, a_q) \wedge (R(\overline{a_0 a_q}) = \langle n_0, d \rangle \wedge n_0 > 0) \wedge (mir_{xi}\langle a_i, a_{i+1}, a_q \rangle \in M$ where $i = 0$ to $m-1$ and $a_m = a_p)$, which implies $(R(\overline{a_i a_{i+1}}) = \langle n_i, d_i \rangle \wedge n_i > 0$ where $i = 1$ to $m-1$ and $a_m = a_p)$ by Lemma 5.2.6. Now condiser the following two sub-cases.

  First, let $a_0 \in A'$, which implies $R'(\overline{a_0 a_q}) = \langle n_0, d \rangle \wedge n_0 > 0$. Let $A_{0_to_u-1} = \{a_0, a_1, ..., au-1\} \subseteq A'$. Notice that $m \geq u \geq 0$ because $(a_0 \in A' \wedge a_m \notin$

A'). $R(\overline{a_i a_{i+1}}) = \langle n_i, d_i \rangle \wedge n_i > 0$ where $i = 1$ to $m - 1$ and $a_m = a_p$) and ($A_{0_to_u-1} \subseteq A'$) implies ($R'(\overline{a_i a_{i+1}}) = \langle n_i, d_i \rangle \wedge a_u \notin A'$ where $i = 1$ to $u - 1$). Thus the theorem is true for this sub-case.

Second, $a_0 \notin A'$ implies $\overline{a_0 a_q} \in IR'$.

Therefore, the theorem is true. □

CHAPTER 6
Correctness of the Distributed Snapshot Algorithm

In this chapter we describe a formal model of the distributed snapshot algorithm for actor garbage collection. First we define the model of local state logging, and then use the reachability relationship to prove safety and liveness of local garbage collection based on the local snapshot. We then formalize snapshot composition, and provide safety and liveness proofs for snapshot composition. Proofs of lemmas can be found in Section 6.4.

6.1 The Computing Model of the Snapshot Algorithm

Reachability from an actor to another actor is important for garbage collection. We defined it in Chapter 3 as the transitive reachability relation, $\rightsquigarrow$, and it will be used in this chapter. Then we will introduce the snapshot (actor configuration), some common terms for the snapshot, and the mutation operations on the snapshot.

Definition 6.1.1. *Actor configuration (snapshot).*
An actor configuration (snapshot),

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

is a 4-tuple where

- *V is a set of actor names.*
- *E is a set of references. $E = \{\overline{xy} \,|\, x \in V \wedge \overline{xy}$ is a reference.$\}$*
- *PS is a set of pseudo-roots. It consists of unblocked actors, roots, and sender pseudo-root actors, but excludes global pseudo-roots. $PS \subseteq V$.*
- *IR is a set of inverse references pointing to external actors. $IR = \{\overline{xy} \,|\, y \in V \wedge x \notin V\}$.*

Definition 6.1.2. *Receptionists, actor references, local actor references, and external inverse references.*
Let S be an actor configuration and Actor $a \in S.V$. Then we define the set of receptionists (remotely referenced actors) $S.RE$, actor references $a.ref$, local actor references $a.lref$, and external inverse references $a.xir$.

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

Definition 6.1.3. *Transitive relationship (mutation operation) on actor configurations.*
Let S be an actor configuration (snapshot) and Actor $a \in S.V$. Then, $\rightarrow$ is defined:

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

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

Definition 6.1.4. *Transitive state transition.*
Let S_1 and S_2 be actor configurations.

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

Definition 6.1.5. *Constrained reachability at actor configurations*
Let a and b be actor names, and S be an actor configuration.

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

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

Definition 6.1.6. *Constrained live actors at actor configurations.*
Let a be an actor name, and S be an actor configuration.

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

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

Definition 6.1.7. *Migration during snapshot state transition.*
Let a be an actor name. Let $S_s \rightarrow^* S_e$.

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

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

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

Proposition 6.1.8. *Initial state of MI operation.*

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

Proposition 6.1.9. *Initial state of CR operation.*

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

Proposition 6.1.10. *Initial state of CA operation.*

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

Proposition 6.1.11. *Initial state of MR operation.*

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

Proposition 6.1.12. *Initial state of IRR operation.*

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

6.2 Safety

In this section, we are going to present the safety property in both local and distributed computing environments.

6.2.1 Local State Logging

A local actor configuration never produces new garbage as the state mutates. The reason is that the model neither deletes references nor makes any actor blocked, except for actor migration. A migration operation breaks references and actors from the actor configuration. Meanwhile, it results in some actors to be referenced by an external actor, the migrating actor. We will prove that the local state logging model is correct. That is, we show that a migration operation does not affect reachability of actors from pseudo-roots, as formalized in Lemma 6.2.1. Therefore, migration does not add new garbage in the local snapshot.

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

Lemma 6.2.1. *Alternative guaranteed reachability for state transition.*
$(a \leadsto b\ \ at\ S_s) \wedge (a \not\leadsto b\ \ at\ S_e) \Longrightarrow$
$(Migrated(b, S_s, S_e)) \vee (\exists \overline{yx} : ((x \leadsto b\ \ at\ S_e) \wedge (\overline{yx} \in S_e.IR) \wedge (Migrated(y, S_s, S_e)))).$

With Lemma 6.2.1, we now prove that the set of garbage is stable in the actor configuration during local state logging, as shown in Theorem 6.2.5. Theorem 6.2.5

directly turns into Corollary 6.2.6, which guarantees a stable set of local garbage during local state logging.

Lemma 6.2.2. $Live(a)$ *at* $S_e \Longrightarrow Live(a)$ *at* S_s.

Lemma 6.2.3. $Migrated(a, S_s, S_e) \Longrightarrow Live(a)$ *at* S_s.

Lemma 6.2.4. $Live(a)$ *at* $S_s \Longrightarrow ((Live(a)$ *at* $S_e) \vee Migrated(a, S_s, S_e))$.

Theorem 6.2.5. *Coherent live actors in a local snapshot.*
$Live(a)$ *at* $S_s \Longleftrightarrow (Live(a)$ *at* $S_e) \vee (Migrated(a, S_s, S_e))$.

Proof. The proof is trivial by Lemma 6.2.2, 6.2.3, and 6.2.4. □

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

Corollary 6.2.6. *The stable property of the set of garbage actors of a local snapshot.*
$\neg Live(a)$ *at* $S_s \Longleftrightarrow (\neg Live(a)$ *at* $S_e) \wedge (\neg Migrated(a, S_s, S_e))$.

Proof. The proof is trivial by Theorem 6.2.5. □

6.2.2 Global Snapshot

Independent local state logging cannot reclaim global cyclic garbage. A coordinated action of local state logging is required to guarantee a causally consistent global snapshot. Let us assume that there are lots of computing nodes participating in a global snapshot activity. Figure 6.1 explains how global synchronization works. Now, consider the synchronization pseudo-code in Figure 4.6. Let t_s be the time the last computing node replies `YES` (line 6), and t_e the time the last computing node finishes `local_snapshot` (line 11). When a computing node finishes a `local_snapshot`, the local actor configuration should remain the same. Let $S_{s,i}$ be the actor configuration of the local group of the computing node i at time t_x, where $t_s \leq t_x \leq t_e$. Let $S_{e,i}$ be the local actor configuration at time t_e. Local actor configurations at t_e can be obtained easily for garbage collectors because they never

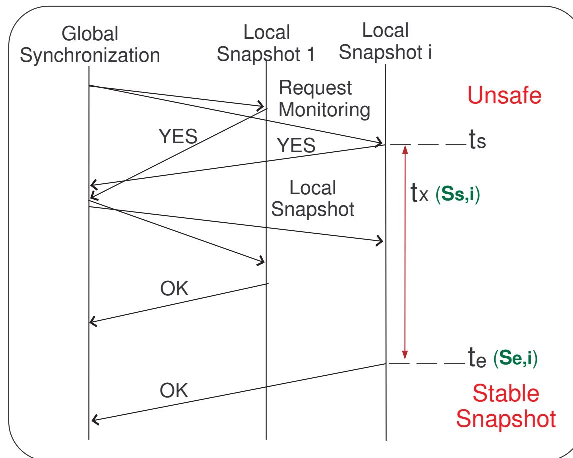

Figure 6.1: Different phases of global synchronization.

change again, while configurations at t_x are only used for proofs because they are volatile. Note that $S_{s,i} \rightarrow^* S_{e,i}$.

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

Lemma 6.2.7. *No actor appears more than once among coordinated local actor configurations.*
Let $S_1, S_2, ..., S_m$ *be coordinated local actor configurations.*
$\forall i, j : (S_i.V \cap S_j.V = \emptyset)$ *where* $(i \neq j) \wedge (m \geq i, j \geq 1)$.

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

Definition 6.2.8. *Real-world actor configuration.*
A real-world actor configuration,

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

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

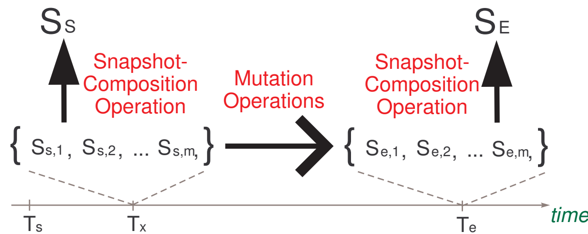

Figure 6.2: The relationship of mutation operations, snapshots, and the snapshot-composition operation. There are two actor configuration sets in the figure — one is $\{S_{s,i} \,|\, m \geq i \geq 1\}$ at time t_x, and the other is $\{S_{e,i} \,|\, m \geq i \geq 1\}$ at time t_e, where $S_{s,i} \rightarrow^* S_{e,i}$ and t_x and t_e are defined in Figure 6.1 as time points. $S_S = (S_{s,1} \parallel S_{s,2} \parallel ... \parallel S_{s,m})$, and $S_E = (S_{e,1} \parallel S_{e,2} \parallel ... \parallel S_{e,m})$.

Definition 6.2.9. *Binary snapshot-composition operation.*
Let $S_x = \langle V_x, E_x, PS_x, IR_x \rangle$ *and* $S_y = \langle V_y, E_y, PS_y, IR_y \rangle$*. Then*

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

where $V_x \cap V_y = \emptyset$*.*

The composition operation on actor configurations is *closed* ($\forall S_i, S_j : S_i \parallel S_j$ is also an actor configuration), *commutative* ($S_i \parallel S_j = S_j \parallel S_i$), and *associative* ($(S_i \parallel S_j) \parallel S_k = S_i \parallel (S_j \parallel S_k)$). The relationship of mutation operations, local snapshots, and the snapshot-composition operation are shown in Figure 6.2. Notice that S_S does not directly transit to S_E, and only the set $\{S_{e,i} \,|\, n \geq i \geq 1\}$ is observable.

The snapshot-composition operation can possibly identify new garbage which cannot be detected in each member of a snapshot set independently. However, the non-blocking, non-FIFO reference listing algorithm, such as the pseudo-root approach, has to guarantee Proposition 6.2.10 to solve inconsistency of two actor configurations (Figure 6.3) — one has a reference to the other one and the other one does not have a corresponding inverse reference. Then we prove that garbage in the global snapshot remains stable (safety).

Proposition 6.2.10. *Consistency guarantee of remote references and inverse references.*

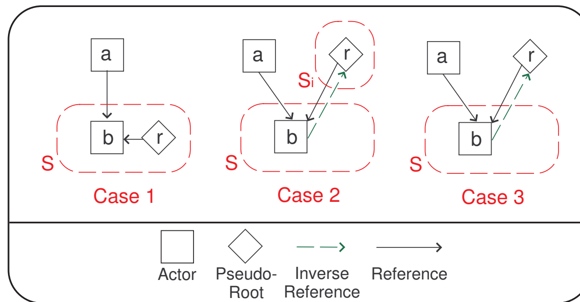

Figure 6.3: Proposition 6.2.10: consistency guarantee of remote references and inverse references.

Let a and b be actor names, S and S_1 ... S_m be coordinated actor configurations, and R be the real-world actor configuration, then

$$(a \in R.V) \wedge (b \in S.V) \wedge (\overline{ab} \in R.E) \wedge (\overline{ab} \notin S.IR) \implies$$
$$(\exists r : (r \in S.PS) \wedge (\overline{rb} \in S.E)) \vee$$
$$(\exists r, S_i : (r \in S_i.PS) \wedge (\overline{rb} \in S_i.E) \wedge (\overline{rb} \in S.IR) \textit{ where } m \geq i \geq 1) \vee$$
$$(\exists r : ((\overline{rb} \in S.IR) \wedge (\forall S_i : \overline{rb} \notin S_i.E) \textit{ where } m \geq i \geq 1)).$$

Let us assume the pseudo-root approach is used. Let Actors $\{a, b\} \subseteq R$ where R is the real-world configuration. Assume $S.V$ contains Actor b, and Actor a has a reference to Actor b in the real world. By Theorem 5.3.3 (the one-step back tracing safety), Actor b must have an outgoing inverse reference to a sender pseudo-root actor, or it is referenced by a local sender pseudo-root actor. Either case is detectable and reflected in S. This is because local state logging mimics the real-world computations, except reference deletion and state change from pseudo-root (unblocked) to blocked. Let the sender pseudo-root actor be r. If Actors r and b are in the same snapshot, their relationship is detectable. If r is in another participating snapshot S_i, then S_i must have the information of Reference $\overline{rb}$ and S has the information of the inverse reference of $\overline{rb}$. If Actor r is not in any of the participating snapshots, Reference $\overline{rb}$ is not detectable in any of the participating snapshots but S still contains the information of the inverse reference of $\overline{rb}$. Therefore, Proposition 6.2.10 must be true.

The following lemmas and theorems require the concept of *constrained paths*

to describe the relationship of actor reachability in a snapshot:

Definition 6.2.11. *Constrained path and its reference set.*
Given $S = \langle V, E, PS, IR \rangle$, *we define a constrained path* P *of* $a_1 \rightsquigarrow a_n$ *at* S,

$$P = \overline{a_1 a_2 ... a_n} \ \ at\ S$$

if and only if $\forall i : \overline{a_i a_{i+1}} \in S.E$ *where* $n > i \geq 1$, *and we define the reference set of* P *as*

$$PathSet(P) = \bigcup_{i=1}^{n-1} \{\overline{a_i a_{i+1}}\}.$$

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

Lemma 6.2.12. *Actors which are reachable from a migrated actor at a local snapshot are reachable from some global pseudo-root at the merged global snapshot.*
$(a \rightsquigarrow b\ at\ S_{s,1}) \wedge (a \not\rightsquigarrow b\ at\ S_{e,1}) \Longrightarrow$
$Migrated(b, S_{s,1}, S_{e,1}) \vee (\exists z : (z \in S_E.RE) \wedge (z \rightsquigarrow b\ at\ S_E))$.

Lemma 6.2.13. *If an actor is garbage at the beginning of the snapshot procedure, then the actor must be garbage in the real world.*
$((\neg Live(a)\ at\ S_S) \wedge (a \in S_S.V)) \Longrightarrow (\nexists y : (y \in R.PS) \wedge (y \rightsquigarrow a\ at\ R))$.

Lemma 6.2.14. $(Live(a)\ at\ S_S \wedge |PathSet(P).IR| = 0) \Longrightarrow (Live(a)\ at\ S_E)$.

Lemma 6.2.15. $Live(a)\ at\ S_S \Longrightarrow (Live(a)\ at\ S_E) \vee (\exists i : Migrated(a, S_{s,i}, S_{e,i}),$ $m \geq i \geq 1)$.

Lemma 6.2.16. $(Live(a)\ at\ S_E) \vee (\exists i : Migrated(a, S_{s,i}, S_{e,i}),\ m \geq i \geq 1) \Longrightarrow$ $Live(a)\ at\ S_S$.

Theorem 6.2.17. *Coherent live actors in a global merged actor configuration.*
$Live(a)\ at\ S_S \iff (Live(a)\ at\ S_E) \vee (\exists i : Migrated(a, S_{s,i}, S_{e,i}),\ m \geq i \geq 1).$

Proof. The proof is trivial by Lemma 6.2.15 and 6.2.16. □

Corollary 6.2.18. *The stable property of the set of garbage actors in a global partial snapshot.*
$\neg Live(a)\ at\ S_S \iff ((\neg Live(a)\ at\ S_E) \wedge (\nexists i : Migrated(a, S_{s,i}, S_{e,i}),\ m \geq i \geq 1).$

Proof. This follows trivially from Theorem 6.2.17. □

6.3 Liveness

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

Theorem 6.3.1. *Conditional liveness of local garbage collection.*
Let each local garbage collector be periodically activated, and use the local snapshot for actor garbage collection. If every non-pseudo-root actor is always selected, all local garbage is eventually identified by trace-based algorithms (DFS or BFS).

Proof. Since the set of local garbage actors is a subset of the local non-pseudo-root actors, all garbage actors are selected into the snapshot for garbage collection. According to Corollary 6.2.6, all garbage in this cycle can be identified and reclaimed in this cycle of garbage collection. The floating garbage produced during the current cycle can be identified at the next cycle because garbage actors cannot execute mutation operations. □

A global garbage collector can use a distributed tracing algorithm to identify global garbage, including distributed cycles. Liveness of global garbage collection based on the snapshot algorithm can be proven in a similar manner of Theorem 6.3.1, and thus we skip the proof.

Theorem 6.3.2. *Conditional liveness of distributed garbage collection that uses the snapshot algorithm.*

Let the global garbage collection mechanism be periodically activated, and use the proposed snapshot algorithm for distributed actor garbage collection. If every non-pseudo-root actor is always selected for snapshot, all garbage is eventually identified by distributed trace-based algorithms (DFS or BFS).

6.4 Proofs

In this section, we provide proofs for the proposed snapshot computing model. The proofs for local state logging properties can be found in Section 6.4.1; the proofs for global snapshot algorithm properties are described in Section 6.4.2.

6.4.1 Local State Logging Properties

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

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

Definition 6.4.1. *No-migration transitive state transition.*
Let S_1 and S_2 be actor configurations.

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

Similarly,

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

Lemma 6.2.1 Alternative guaranteed reachability for state transition.
$(a \rightsquigarrow b \ at \ S_s) \wedge (a \not\rightsquigarrow b \ at \ S_e) \Longrightarrow$
$(Migrated(b, S_s, S_e)) \vee (\exists \overline{yx} : (x \rightsquigarrow b \ at \ S_e) \wedge (\overline{yx} \in S_e.IR) \wedge (Migrated(y, S_s, S_e)))$.

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

- Case 1: Let $(b \notin S_e.V)$. Since the only operation to remove b is $b.MI$, then there must exist S_i, S_j $s.t.$ $S_s \rightarrow^* S_i \xrightarrow{b.MI} S_j \rightarrow^* S_e$, where $b \in S_i.V \wedge b \notin S_j.V$. Then $Migrated(b, S_s, S_e)$.

- Case 2: Let $(b \in S_e.V)$. Because $(a \not\leadsto b\ at\ S_e)$, there exists a reference at S_s and it is removed by a MI operation during state transitions, making $a \not\leadsto b$. This case can be proven by induction. Let the number of performed MI operations be n during state transitions where $n \geq 1$.

  Basis: Prove that the statement is true for $n = 1$.

  Let $S_s \rightarrow^* S_i \xrightarrow{y.MI} S_j \rightarrow^* S_e$ where $((a \leadsto y) \wedge (x \leadsto b) \wedge (\overline{yx} \in S_i.E))$ and $(a \not\leadsto b\ at\ S_j)$. Therefore, 1) $x \in S_j.RE$, 2) $x \leadsto b\ at\ S_j$, and 3) $\overline{yx} \in S_j.IR$. Because no more MI is performed, $((x \leadsto b\ at\ S_e) \wedge (\overline{yx} \in S_e.IR) \wedge (S_s \rightarrow^* S_i \xrightarrow{y.MI} S_j \rightarrow^* S_e))$ which proves the basis.

  Induction step: Assume the statement is true for n where $k \geq n \geq 1$, and show that it is true for $n = k + 1$.

  Let S_i be the state right before the last MI is performed. That is, $\exists y : S_i \xrightarrow{y.MI} S_j$. Because $(b \in S_e.V)$, $b.MI$ cannot be the last MI operation.

  Let $\overline{yx}$ be the reference removed by the last MI, $y.MI$. According to the induction hypothesis, $\exists \overline{zw}, S_k, S_m : ((w \leadsto y \leadsto x \leadsto b\ at\ S_i) \wedge (\overline{zw} \in S_i.IR) \wedge (S_s \rightarrow^* S_k \xrightarrow{z.MI} S_m \rightarrow^* S_i))$. Now, consider the following sub-cases:

  Sub-case 1: Let $(w \leadsto b\ at\ S_j)$, which means $((w \leadsto b\ at\ S_j) \wedge (\overline{zw} \in S_j.IR) \wedge (S_s \rightarrow^* S_k \xrightarrow{z.MI} S_m \rightarrow^* S_j))$. Since $(S_j \xrightarrow{(\neq MI)}{}^* S_e)$ do not remove any references or actor names, $((w \leadsto b\ at\ S_e) \wedge (\overline{zw} \in S_e.IR) \wedge (S_s \rightarrow^* S_k \xrightarrow{z.MI} S_m \rightarrow^* S_e))$. Thus the lemma is true for Sub-case 1.

  Sub-case 2: The concept of this sub-case is shown in Figure 6.4. Let $(w \not\leadsto b\ at\ S_j)$. Since $((w \leadsto b\ at\ S_i) \wedge (w \not\leadsto b\ at\ S_j))$ and only one MI is performed during $(S_i \xrightarrow{y.MI} S_j \xrightarrow{(\neq MI)}{}^* S_e)$, the statement can be proven by re-using the basis.

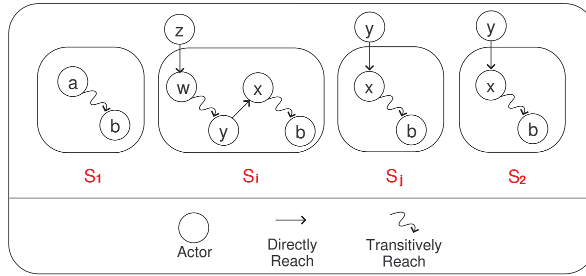

Figure 6.4: Sub-case 2 of the induction step of the proof for Lemma 6.2.1.

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

□

Lemma 6.2.2 $Live(a)\ at\ S_e \implies Live(a)\ at\ S_s$.

Proof. The statement can be proven by contradiction. Assume the statement is wrong, which means $(\neg Live(a)\ at\ S_s) \implies \forall y : (y \in (S_s.PS \cup S_s.RE)) \wedge (y \not\rightsquigarrow a\ at\ S_s)$. According to Proposition 6.1.8, 6.1.9, 6.1.10, 6.1.11, and 6.1.12, a cannot execute any mutation operation, and must remain the same state at S_e. By proposition 6.1.9, y cannot create a reference to a because $(Live(a)\ at\ S_s)$ is required. Therefore, $(\forall y : (y \in (S_e.PS \cup S_e.RE)) \wedge (y \not\rightsquigarrow a \text{ at } S_e)) \implies (\neg Live(a)\ at\ S_e)$, which contradicts the premise. □

Lemma 6.2.3 $Migrated(a, S_s, S_e) \implies Live(a)\ at\ S_s$.

Proof. $(Migrated(a, S_s, S_e)$ implies $(\exists S_i, S_j : S_s \rightarrow^* S_i \xrightarrow{a.MI} S_j \rightarrow^* S_e))$. By Proposition 6.1.8, $a \in S_i.PS$. By Definition 6.1.5 and 6.1.6, $Live(a)\ at\ S_i$. By Lemma 6.2.2, $Live(a)\ at\ S_s$. □

Lemma 6.2.4 $Live(a)\ at\ S_s \implies ((Live(a)\ at\ S_e) \vee Migrated(a, S_s, S_e))$.

Proof. By Definition 6.1.5, $Live(a)\ at\ S_e \iff \exists x : (x \in (S_e.PS \cup S_e.RE)) \wedge (x \rightsquigarrow a\ at\ S_e)$. By Definition 6.1.7, $Migrated(a, S_s, S_e) \iff (\exists S_i, S_j : S_s \rightarrow^* S_i \xrightarrow{a.MI} S_j \rightarrow^* S_e)$. Assume the statement is wrong, which means:

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

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

Let $z \in S_s.V$ and $(z \in (S_s.PS \cup S_s.RE)) \wedge (z \rightsquigarrow a\ at\ S_s)$ from the premise. One can conclude from Assumption 1 and Assumption 2 that $(z \in (S_e.PS \cup S_e.RE)) \wedge (z \not\rightsquigarrow a\ at\ S_e)$. By using Lemma 6.2.1, we know that $(z \rightsquigarrow a\ at\ S_s) \wedge (z \not\rightsquigarrow a\ at\ S_e)$ implies either:

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

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

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

6.4.2 Global Snapshot Algorithm Properties

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

Lemma 6.2.7 No actor appears more than once among coordinated local actor configurations.

Let $S_1, S_2, ..., S_m$ be coordinated local actor configurations.
$\forall i, j : (S_i.V \cap S_j.V = \emptyset)$ where $(i \neq j) \wedge (m \geq i, j \geq 1)$.

Proof. Only migration can cause an actor to appear twice in different locations. Let computing node i and j be two arbitrary computing nodes whose final local actor configurations are S_i and S_j respectively. Let a be in computing node i, and then migrate to j. Also let t_{si} be the time computing node i starts state logging, t_{ei} the time computing node i finishes state logging, t_{sj} the time computing node j starts state logging, and t_{ej} the time computing node j finishes state logging. Let the

time to migrate from node i be t_{smig}, and to arrive at node j be t_{emig}. Now, let us consider all possible cases that a will appear in both S_i and S_j.

Case 1: $t_{smig} > t_{ei}$.

Now, consider three sub-cases:

Sub-case 1.1: $t_{emig} > t_{ej}$.

a cannot be logged in S_j.

Sub-case 1.2: $t_{ej} \geq t_{emig} \geq t_{sj}$.

S_j cannot include any new actor during this period of time.

Sub-case 1.3: $t_{sj} > t_{emig}$ (an early message of migration).

Considering $t_{sj} > t_{emig}$, $t_{emig} > t_{smig}$, and $t_{smig} > t_{ei}$, we get $t_{sj} > t_{ei}$. However, the global synchronization mechanism guarantees that $t_{ei} \geq t_{sj}$ because snapshot termination must wait for a consensus of all participating computing nodes. Therefore, this sub-case is impossible due to a contradiction.

Case 2: $t_{ei} \geq t_{smig} \geq t_{si}$.

The MI operation guarantees that $\{a\} \notin S_i.V$.

Case 3: $t_{si} > t_{smig}$.

a cannot be logged in S_i.

$\square$

Lemma 6.2.12 Actors which are reachable from a migrated actor at a local snapshot are reachable from some global pseudo-root at the merged global snapshot.

$(a \rightsquigarrow b \ at\ S_{s,1}) \wedge (a \not\rightsquigarrow b \ at\ S_{e,1}) \Longrightarrow$

$Migrated(b, S_{s,1}, S_{e,1}) \vee (\exists z : (z \in S_E.RE) \wedge (z \rightsquigarrow b \ at\ S_E))$.

Proof. According to Lemma 6.2.1, one of the following statements is true.

Case 1: $\exists S_j, S_k : S_{s,1} \rightarrow^* S_j \xrightarrow{b.MI} S_k \rightarrow^* S_{e,1}$ by Definition 6.1.7.

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

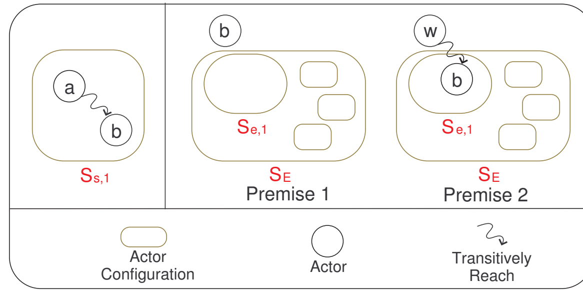

Figure 6.5: Two different possible global actor configurations where $\exists S_k, S_m : S_{s,1} \rightarrow^* S_k \xrightarrow{a.MI} S_m \rightarrow^* S_{e,1}$.

Figure 6.5 helps understand the proof. If Case 1 is true, the statement to prove is also true. Now, consider Case 2. Let $S'_E = (S_{e,2} \parallel S_{e,3} \parallel ... \parallel S_{e,m})$. Let S_j, S_k be the actor configurations and $\overline{wx}$ be the reference to make Case 2 true. Because $w \in S_{s,1}.V$ and no duplicate actor name is allowed (Lemma 6.2.7), we know ($w \notin S_{s,i}.V$ where $m \geq i > 1$). Consequently, $w \notin S_{e,i}.V$ where $m \geq i > 1$ because no mutation operation can add any new actor name. Since ($w \notin S_{e,i}.V$ where $m \geq i > 1$), we get ($\overline{wx} \notin S_{e,i}.E$ where $\forall i : m \geq i > 1$), which also implies that $\overline{wx} \notin S'_E.E$. From Case 2 we know ($\overline{wx} \in S_{e,1}.IR$), and thus we get $\overline{wx} \in (S_{e,1}.IR - S'_E.E)$.

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

Lemma 6.2.13 $((\neg Live(a)\ at\ S_S) \wedge (a \in S_S.V)) \implies (\nexists y : (y \in R.PS) \wedge (y \rightsquigarrow a\ at\ R))$.

Proof. Assume the lemma is wrong, which means $(\exists y : (y \in R.PS) \wedge (y \rightsquigarrow a\ at\ R)) \implies (\exists w, x, y : (y \rightsquigarrow w\ at\ R) \wedge (\overline{wx} \notin S_S.E) \wedge (x \rightsquigarrow a\ at\ S_S))$. Since $\overline{wx} \in S_S.IR \implies ((x \in S_S.RE) \wedge (x \rightsquigarrow a\ at\ S_S)) \implies Live(a)\ at\ S_S$ which contradicts the premise, we know $\overline{wx} \notin S_S.IR$. According to Proposition 6.2.10, $(\exists z : (z \in S_S.IR \cup S_S.PS) \wedge (z \rightsquigarrow a\ at\ S_S)) \implies (Live(a)\ at\ S_S)$ which also contradicts the premise. □

Lemma 6.2.14 $(Live(a)\ at\ S_S \wedge |PathSet(P).IR| = 0) \implies (Live(a)\ at\ S_E)$.

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

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

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

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

Case 2: Now, consider the case that $x \not\rightsquigarrow a \; at \; S_{e,1}$. According to Lemma 6.2.1, either $Migrated(a, S_{s,1}, S_{e,1})$ which means $\exists S_{ma}, S_{mb} : S_{s,1} \rightarrow^{*} S_{ma} \xrightarrow{a.MI} S_{mb} \rightarrow^{*} S_{e,1}$, or
$\exists S_{ma}, S_{mb}, \overline{cw} : ((w \rightsquigarrow a \; at \; S_{e,1}) \wedge (\overline{cw} \in S_{e,1}.IR) \wedge (S_{s,1} \rightarrow^{*} S_{ma} \xrightarrow{c.MI} S_{mb} \rightarrow^{*} S_{e,1}))$. Because $(c \in S_{s,1}.V)$, we know $(c \notin S_{s,i}.V$ where $m \geq i \geq 2)$, which also means that $c \notin S_{e,i}.V$ where $m \geq i \geq 2$. $(c \notin S_{e,i}.V$ where $m \geq i \geq 1)$ implies $(\forall d : \overline{cd} \notin S_{e,i}.E$ where $m \geq i \geq 1)$. Since $\overline{cw} \in S_{s,1}.IR$ and $(\forall d : \overline{cd} \notin S_{e,i}.E$ where $m \geq i \geq 1)$, we get $\overline{cw} \in S_E.IR$ which also means $w \in S_E.RE$. Therefore, $(w \in (S_E.PS \cup S_E.RE)) \wedge (w \rightsquigarrow a \; at \; S_E)$, implying $Live(a) \; at \; S_E$.

□

Lemma 6.2.15 $Live(a) \; at \; S_S \Longrightarrow (Live(a) \; at \; S_E) \vee (\exists i : Migrated(a, S_{s,i}, S_{e,i}),$ $m \geq i \geq 1)$.

Proof. Let $a \in S_{s,1}.V$ without losing generality. Let $|PathSet(P).IR| = n$.
Basis: The case of $n = 0$ is proven by Lemma 6.2.14.
Induction step: For each $k \geq 0$ assume the statement is true, and prove the case

$n = k + 1$.

Let $\overline{pq}$ be any inter-snapshot reference of $x \rightsquigarrow a\ at\ S_S$ s.t. $((x \rightsquigarrow p\ at\ S_S) \wedge (q \in S_{s,1}.V) \wedge (\exists j : p \in S_{s,j}.V$ where $m \geq j \geq 2) \wedge (q \rightsquigarrow a\ at\ S_{s,1}))$. Consequently, the induction assumption indicates either 1) $Live(p)\ at\ S_E$, or 2) $\exists i, S_{ms,i}, S_{me,i} : (S_{s,i} \rightarrow^* S_{ms,i} \xrightarrow{p.MI} S_{me,i} \rightarrow^* S_{e,i})$, $m \geq i \geq 1$ by Definition 6.1.7.

These cases are discussed as follows:

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

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

Sub-case 1.2: Let $q \not\rightsquigarrow a\ at\ S_{e,1}$. By Lemma 6.2.12, we know

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

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

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

Sub-case 2.1.1: Assume $\overline{pq} \in S_{ms,1}.IR$. Since $q \in S_{e,1}$, we get $\overline{pq} \in S_{e,1}.IR$. Because $p \in S_{s,i}.V$ and no duplicate actor name is allowed (Lemma 6.2.7), we know ($p \notin S_{s,j}.V$ where $m \geq j \geq 1$). Consequently, $p \notin S_{e,i}.V$ where $m \geq i \geq 1$ because no mutation operation can add any new actor name. Since ($p \notin S_{e,i}.V$ where $m \geq i \geq 1$), we get ($\overline{pq} \notin S_{e,i}.E$ where $m \geq i \geq 1$). Therefore, $(q \in S_E.RE \wedge q \rightsquigarrow a\ at\ \ S_E) \Longrightarrow Live(a)\ at\ S_E$.

Sub-case 2.1.2: Assume $\overline{pq} \notin S_{ms,1}.IR$. According to Proposition 6.2.10, there are three cases to consider as follows:

- Let $(\exists r : (r \in S_{ms,1}.PS) \wedge (\overline{rq} \in S_{ms,1}.E))$ be true. If $(\exists S_{rms,1}, S_{rme,1} : S_{ms,1} \rightarrow^* S_{rms,1} \xrightarrow{r.MI} S_{rms,1} \rightarrow^* S_{e,1})$ is true, we can prove the statement

by Lemma 6.2.12. Otherwise, $r.MI$ has never been executed. Therefore, $((r \in S_{e,1}.PS) \wedge (r \rightsquigarrow q \rightsquigarrow a \ at \ S_{e,1}))$, which implies $Live(a) \ at \ S_E$.

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

Sub-case 2.2: Let $q \not\rightsquigarrow a \ at \ S_{e,1}$. Sub-case 2.2 can be proven by reusing the proof of Lemma 6.2.14.

We conclude the statement is true by induction.

□

Lemma 6.2.16 $(Live(a) \ at \ S_E) \vee (\exists i : Migrated(a, S_{s,i}, S_{e,i}),\ m \geq i \geq 1) \Longrightarrow Live(a) \ at \ S_S$.

Proof. The lemma can be proven by contradiction. Assume $\neg Live(a) \ at \ S_S$ where $a \in S_S.V$. By Lemma 6.2.13, we get $(\forall y : (y \in R.PS) \wedge (y \not\rightsquigarrow a \ at \ R))$. Since local state logging corresponds to the real world computing, Actor a cannot execute any mutation operations, which means $\neg Live(a) \ at \ S_E$ and thus contradicts the premise. □

CHAPTER 7
Experimental Results

The proposed distributed mobile actor garbage collection algorithms are implemented in the SALSA programming language [109], where the pseudo-root approach is part of the core SALSA library and the distributed snapshot algorithm is implemented as an optional, logically centralized service. In this chapter, we use several types of applications to measure the performance of our implementation of the mobile actor garbage collection mechanism.

7.1 Actor Garbage Collection in SALSA

SALSA (*Simple Actor Language, System and Architecture*) is an actor-oriented programming language designed and implemented to introduce the benefits of the actor model while keeping the advantages of object-oriented programming. Abstractions include actors (active objects), asynchronous message passing, universal naming, migration, and advanced coordination constructs for concurrency. SALSA is pre-processed into Java and hence preserves many of Java's useful object oriented concepts — mainly, encapsulation, inheritance, and polymorphism. Because each actor maintains an encapsulated state, SALSA uses strict pass-by-value communication. [7] SALSA abstractions enable the development of dynamically reconfigurable applications. A SALSA program consists of universal actors that can migrate to other computing nodes at run-time.

SALSA provides actor garbage collection, which identifies actor garbage and let the Java garbage collector reclaim it. In case of active garbage, the thread of control of a garbage actor is destroyed first, and then the rest is handled by the Java garbage collector. Intra-actor garbage produced by updating references (the assignment operation) is handled by the Java garbage collector as well.

[7] Actor references (Actor names) are first-class and therefore they can also be passed by value.

Actor GC Mechanism to Measure

To understand the impact of actor garbage collection, we measure actor garbage collection using four different mechanisms: *NO-GC*, *GDP*, *LGC*, and *CDGC*. By using these mechanisms, we can understand the overhead each actor garbage collection algorithm imposes on the actor system. The mechanisms are described as follows:

- *No-GC*: Data structures and algorithms for actor garbage collection are not used.
- *GDP*: The local garbage collector is not activated. Only the *garbage detection protocol* (the implementation of the pseudo-root approach) is used.
- *LGC*: The local garbage collector is activated every n seconds or in the case of insufficient memory (n=2 for the tests in this chapter).
- *CDGC*: The logically centralized garbage collector is activated every m seconds or in the case of insufficient memory (m=20 for the tests in this chapter).

7.2 Actor GC Overhead with Respect to Application Tests

We developed three different benchmark applications using *SALSA 1.0.0* to measure the impact of our local actor garbage collection mechanism. These applications are *Fibonacci number (Fib)*, *N queens number (NQ)*, and *Matrix multiplication (MX)*. Each application is executed on a dual-core processor Sun Blade 1000s machine, equipped with two 750 MHz processors and 2 GB of RAM. The operating system used was SunOS 5.10 and the Java VM was Java HotSpot Client VM (build 1.4.1). The applications are described as follows:

- Fibonacci number (Fib): Fibonacci number, abbreviated as *Fib*, takes one argument k and then computes the k-th Fibonacci number concurrently. It is a coordinated tree-structure computation. When $k \leq 30$, the application sequentially computes the k-th Fibonacci number.
- N queens number (NQ): N queens number, abbreviated as *NQ*, takes one argument to calculate the total solutions of the N queens problem by creating $(N-1) \times (N-2)$ actors for parallel execution and one actor for coordination.

- Matrix multiplication (MX): Matrix multiplication, abbreviated as *MX*, requires two files for application arguments, each of which contains a matrix. The application calculates one matrix multiplication of the given two matrices.

We also developed four distributed benchmark applications. They are performed on four dual-core processor Sun Blade 1000s machines. The distributed benchmark applications are described as follows:

- Distributed Fibonacci number with locality (Dfibl): *Dfibl* optimizes the number of inter-node messages by locating four sub-computing-trees at each computing node.

- Distributed Fibonacci number without locality (Dfibn): *Dfibn* distributes the actors in a breadth-first-search manner.

- Distributed N queens number (DNQ): *DNQ* equally distributes the actors to four computing nodes.

- Distributed Matrix multiplication (DMX): *DMX* divides the first input matrix into four sub-matrices, sends the sub-matrices and the second matrix to four computing nodes, performs one matrix multiplication operation, and then merges the data at the computing node that initializes the computation.

The local experimental results are shown in Table 7.1, and the distributed results are in Table 7.2. Each result of a benchmark application is the average of ten execution times. Notice that *Real* represents the total real execution time to get the computing result, while *CPU* represents the total CPU time of both processors to get the computing result. CPU time can be bigger than Real time because the machine to test has two CPUs and CPU time is equal to the sum of the individual CPU time. The average GDP real time overhead of local experimental results is 20.5%; the average GDP CPU time overhead of local experimental results is 16%; the average LGC+GDP Real time overhead of local experimental results is 24%; the average LGC+GDP CPU time overhead of local experimental results is 19%; the average LGC+GDP+CDGC Real time overhead of experimental results is 19%;

Table 7.1: Application performance on a dual-core processor Sun Blade 1000s machine (measured in seconds).

Application(Arguments) /Number of Actors	NO-GC (Real/CPU)	GDP (Real/CPU)	LGC+GDP (Real/CPU)	GDP OVERHEAD (Real/CPU)	LGC+GDP OVERHEAD (Real/CPU)
Fib(38)/109	1.70/2.57	2.14/3.07	2.13/3.08	25%/20%	25%/20%
Fib(41)/465	5.09/8.66	6.20/10.21	6.63/10.54	22%/18%	30%/22%
NQ(13)/133	1.84/2.16	2.42/2.55	2.55/2.79	32%/18%	39%/29%
NQ(15)/183	6.72/11.58	7.63/12.84	7.97/13.30	14%/11%	19%/15%
MX(100^2)/3	1.84/1.93	2.16/2.24	2.16/2.24	17%/16%	17%/16%
MX(150^2)/3	2.63/2.84	2.97/3.17	3.03/3.20	13%/11%	15%/13%

Table 7.2: Application performance in a distributed environment. Real time is measured in seconds.

Application(Arguments) /Number of Actors	NO-GC	LGC+GDP+CDGC	LGC+GDP+CDGC OVERHEAD
Dfibl(39)/177	1.722	2.091	21%
Dfibl(42)/753	3.974	4.957	25%
Dfibn(39)/177	3.216	3.761	17%
Dfibn(42)/753	8.527	9.940	17%
DNQ(16)/211	13.120	17.531	34%
DNQ(18)/273	426.151	461.757	8%
DMX(100^2)/5	6.165	6.715	9%
DMX(150^2)/5	39.011	38.955	0%

7.3 Overhead Breakdown

There are totally four kinds of mutation operations in actor garbage collection: *actor creation*, *actor reference passing*, *actor reference deletion*, *and actor migration*. To understand if the actor garbage collection mechanism is intrusive or not, we also measure the effect on loops containing multiplication and division operations. Since the extra execution time required for actor reference deletion and actor reclamation is handled by Java virtual machines, we do not provide any overhead breakdown for them. However, their overheads should be proportional to the overhead of actor creation because the Java garbage collector needs to reclaim extra objects [8] produced by actor creation. All testing applications used in this section were developed using *SALSA 1.1.1*.

7.3.1 Overhead Breakdown of Local Actor GC in a Uniprocessor Environment

The experiments were performed on a Dell Inspiron 600m laptop, equipped with a 1.6 GHz Intel Pentium M processor and 512 MB of RAM. The operating

[8] The total size of the required objects is fixed in the experiments.

Table 7.3: Actor creation (ms) and its overhead (%) in a uniprocessor environment.

ACTOR CREATION	NO-GC	GDP	LGC	GDP OVERHEAD	LGC+GDP OVERHEAD
EMPTY ACTOR	0.872	1.001	1.382	15%	58%
1 KB STATE	1.412	1.423	1.743	1%	23%
10 KB STATE	1.873	1.913	2.443	6%	30%

Table 7.4: Message passing (μs) and its overhead (%) in a uniprocessor environment.

MESSAGE PASSING	NO-GC	GDP	LGC	GDP OVERHEAD	LGC+GDP OVERHEAD
EMPTY PARAMETERS	6.97	7.13	7.212	2%	3%
1 KB PARAMETERS	57.68	59.08	59.68	2%	3%
10 KB PARAMETERS	59.3	59.28	60.7	0%	2%

system used was Microsoft Windows XP, and the Java VM was Java HotSpot(TM) Client VM (build 1.5.0_06-b05, mixed mode). Experimental results are shown in Tables 7.3, 7.4, 7.5, and 7.6.

Table 7.3 shows the overhead of actor creation. The overhead is relatively huge while creating a small actor because each actor maintains a data structure for the actor referential relationship. The ratio of the extra data structure for actor garbage collection goes down while the size of a newly created actor increases. Table 7.4 shows the overhead of message passing. Table 7.5 shows the overhead of reference passing. The overhead is almost 100% for local actor garbage collection because each reference passing event generates at least 3 extra messages. Table 7.6 shows the overhead of a regular computation, represented by the multiplication-division computation whose overhead is negligible. The minus-overhead could be attributed to the behavior of the operating system or the Java VM. A -1% overhead should be considered as a precision problem of measurement.

Table 7.5: Reference passing (ms) and its overhead (%) in a uniprocessor environment.

REFERENCE PASSING	NO-GC	GDP	LGC	GDP OVERHEAD	LGC+GDP OVERHEAD
1 REFERENCE	0.549	1.094	1.118	99%	104%
10 REFERENCES	0.577	1.156	1.202	94%	102%
100 REFERENCES	0.691	1.348	1.384	95%	100%

Table 7.6: Multiplication-division (Secs) and its overhead (%) in a uniprocessor environment, where each actor performs several loops, each of which contains a double-precision multiplication operation and a double-precision division operation.

MUL-DIV	NO-GC	GDP	LGC	GDP OVERHEAD	LGC+GDP OVERHEAD
3×10^8 LOOPS IN 1 ACTOR	7.971	7.971	8.181	0%	3%
3×10^7 LOOPS IN 10 ACTOR	7.971	7.972	8.116	0%	2%
3×10^6 LOOPS IN 100 ACTOR	8.052	8.082	8.146	0%	1%
3×10^5 LOOPS IN 1000 ACTOR	8.692	8.603	8.933	-1%	3%

7.3.2 Overhead Breakdown of Local Actor GC in a Concurrent Environment

We used the same mechanisms and applications in Subsection 7.3.1 to measure the overhead of local actor garbage collection in a concurrent environment. The experiments were performed on an IBM pSeries 655 machine, equipped with four 1.7 GHz IBM POWER4 processors and 4 GB of RAM. The operating system used was IBM AIX 5.3, and the Java VM was Classic VM (build 1.4.2, J2RE 1.4.2 IBM AIX 5L for PowerPC). Experimental results are shown in Tables 7.7, 7.8, and 7.9. We decided not to show the experimental results of the multiplication-division (ms) and its overhead because it highly depends on the scheduling policy of the operating system and the Java VM — in some cases the system with actor garbage collection can outperform the system without actor garbage collection by almost 100%, while in some cases the system with actor garbage collection can have more than 50% overhead.

The actor garbage collection overhead in the concurrent system shows a significant improvement compared to the uniprocessor environment, shown in Tables 7.7, 7.8, and 7.9. We attribute it to the scheduling policy of the native operating system — the operating system wisely uses another idle processor for concurrent actor garbage collection, which avoids CPU time contention with users' applications. The minus-overhead in Table 7.7 shows that the scheduling policy of the operating system (or the Java VM) can have high influence on the performance of concurrent applications.

Table 7.7: Actor creation (ms) and its overhead (%) in a quad-core processor environment.

ACTOR CREATION	NO-GC	GDP	LGC	GDP OVERHEAD	LGC+GDP OVERHEAD
EMPTY ACTOR	7.594	6.222	6.43	-18%	-15%
1 KB STATE	4.611	4.717	5.191	2%	13%
10 KB STATE	4.446	5.015	5.442	13%	22%

Table 7.8: Message passing (μs) and its overhead (%) in a quad-core processor environment.

MESSAGE PASSING	NO-GC	GDP	LGC	GDP OVERHEAD	LGC+GDP OVERHEAD
EMPTY PARAMETERS	9.914	9.924	9.948	0%	0%
1 KB PARAMETERS	86.6	85.73	87.64	-1%	1%
10 KB PARAMETERS	86.74	86.15	86.44	-1%	0%

7.3.3 Overhead Breakdown of Actor GC in a Cluster Environment

The overhead of actor migration, message passing, and reference passing are evaluated in this subsection. We used two IBM pSeries 655 machines (intra-cluster) to measure the overhead of actor garbage collection. Experiments results are shown in Tables 7.10, 7.11, and 7.12. The major overhead still comes from reference passing, on average 18% (Table 7.12). The overhead shown in Tables 7.10 and 7.11 is negligible.

Table 7.9: Reference passing (ms) and its overhead (%) in a quad-core processor environment.

REFERENCE PASSING	NO-GC	GDP	LGC	GDP OVERHEAD	LGC+GDP OVERHEAD
1 REFERENCE	0.818	0.948	0.957	16%	17%
10 REFERENCES	0.852	0.985	0.993	16%	17%
100 REFERENCES	0.929	1.067	1.113	15%	20%

Table 7.10: Actor migration (ms) and its overhead (%) in a distributed environment.

ACTOR CREATION	NO-GC	GDP	LGC	GDP OVERHEAD	LGC+GDP OVERHEAD
EMPTY ACTOR	158.875	157.755	158.7	-1%	0%
1 KB STATE	160.005	159.15	158.91	-1%	-1%
10 KB STATE	159.855	159.605	159.095	0%	0%

Table 7.11: Message passing (ms) and its overhead (%) in a distributed environment.

MESSAGE PASSING	NO-GC	GDP	LGC	GDP OVERHEAD	LGC+GDP OVERHEAD
EMPTY PARAMETERS	0.149	0.149	0.149	0%	0%
1 KB PARAMETERS	0.251	0.251	0.251	0%	0%
10 KB PARAMETERS	0.336	0.337	0.336	0%	0%

Table 7.12: Reference passing (ms) and its overhead (%) in a distributed environment.

REFERENCE PASSING	NO-GC	GDP	LGC	GDP OVERHEAD	LGC+GDP OVERHEAD
1 REFERENCE	1.063	1.237	1.236	16%	16%
10 REFERENCES	1.152	1.322	1.326	15%	15%
100 REFERENCES	1.221	1.398	1.409	15%	15%

7.4 Scalability Test

In this section, we evaluate the overhead of our mobile actor garbage collection mechanism using a scalable application, namely the *maximum likelihood evaluation fitter* (*MLE fitter*), to evaluate a large set of data, where *likelihood* is defined as the product of the probabilities of observing each event (the input data set) given a set of fit parameters. We used *SALSA 1.1.1* to develop the application.

7.4.1 The Maximum Likelihood Evaluation (MLE) Fitter

According to particle physics, *particles* which make up our universe have wave-like behavior and thus their identities and properties can be determined by *partial wave analysis (PWA)* [27]. To discover the identities and properties of particles, and the forces and interactions between these particles, scientists use a particle accelerator to create a high energy collision of particles. The collision may produce a spray of particles. Some of the particles can live long enough to be observed, while some may decay [9] into other kinds of particles after an extremely short time, making them impossible to be observed. The existence of the short lived particles can only be inferred from correlations in the final state particles into which they decay. There are many ways of reaching the final system through various intermediate states, and

[9]Particle decay refers to the transformation of a fundamental particle into other fundamental particles.

each possibility must be considered. By varying the amount of each intermediate state to fit the observed final state, PWA can determine the identities of the short-lived particles.

The purpose of the fits is to find the most probable intermediate states. The fits are done using *maximum likelihood evaluation (MLE)*. The likelihood is defined as the product of the probabilities of observing each event given a set of fit parameters. In practice, people usually use the negative logarithm likelihood to find the minimum value, which represents the maximum likelihood. In our case, the equation is described as:

$$-\ln(\mathcal{L}) = -\sum_{i}^{n} \ln\left(\left|\psi_{\alpha}^{p}\psi_{\alpha}^{d}(\tau_i)\right|^2\right) - n\psi_{\alpha}^{p}\Psi_{\alpha\alpha'}\psi_{\alpha'}^{p*} \tag{7.1}$$

where the sum over i runs over all events in the data set, and the sums over the repeated αs, the fit parameter index, are implicit. The ψ_{α}^{p} are the complex fit parameters, related to the amount of the intermediate state α produced. The $\psi_{\alpha}^{d}(\tau_i)$ is the identity (*quantum amplitude*) for the ith event with angles τ_i assuming intermediate state α. The possibility of non-interfering data has been ignored here. While physically important, it is a detail which further complicates the expression for the likelihood, yet serves no illustrative purpose for the discussion at hand. The second term on the right hand side is the normalization integral, where any known inefficiencies of the detector are taken into account. The total number of events in the data being fit is n; and $\Psi_{\alpha\alpha'}$ is the result of the normalization integral, done numerically before the fit is performed.

We used the *simplex* algorithm, which finds the most probable fit and iteratively improves the output, in our implementation of the MLE fitter. Finding the best fit parameters to a typical data set requires a given set of initial fit parameters and hundreds or even thousands of trials. In our case the MLE fitter evaluates the maximum likelihood of a given set of complex amplitudes and the observed events from collisions of particles. The execution time of the MLE fitter can be improved by using distributed computing if calculating the summation term of the negative logarithm likelihood equation requires a long time to finish. For example, if n in

Equation 7.1 is large enough ($> 10^5$).

7.4.2 Results

The test was performed in a cluster consisting of 15 computing hosts, consisting of three 4-dual-core 2.2 GHz Opteron machines (8 processors each) with 32 GB of RAM and twelve 4-single-core 2.2 GHz Opteron machines with 16 GB of RAM. The logically centralized actor garbage collector was running on another 2-single-core 2.2 GHz Opteron machine. The operating system used was Linux 2.6.15, and the Java VM was Java HotSpot Server VM (build 1.5.0_08-b03, mixed mode).

We used a set of 9053×2^6 events and 7 complex number parameters as the input for the MLE fitter. The MLE fitter used a static load balancing approach to distribute data to each *theater* [10], where for each processor a theater is started to host actors. The MLE fitter is sensitive to any delay at any theater because the execution time per fit function call is the longest execution time per fit function call among all the participating theaters. Our experiments show that the overall overhead of our implementations is on average 24% for 64 or more theaters, and on average 13% for 32 or less theaters (see Figures 7.1 and 7.2). The MLE fitter running on 64 or more theaters has relatively bad performance, which can be attributed to the execution time per function call being down to 1 second, making it relatively sensitive to any kind of interruption.

[10] A theater can be thought of as the virtual machine of the SALSA programming language.

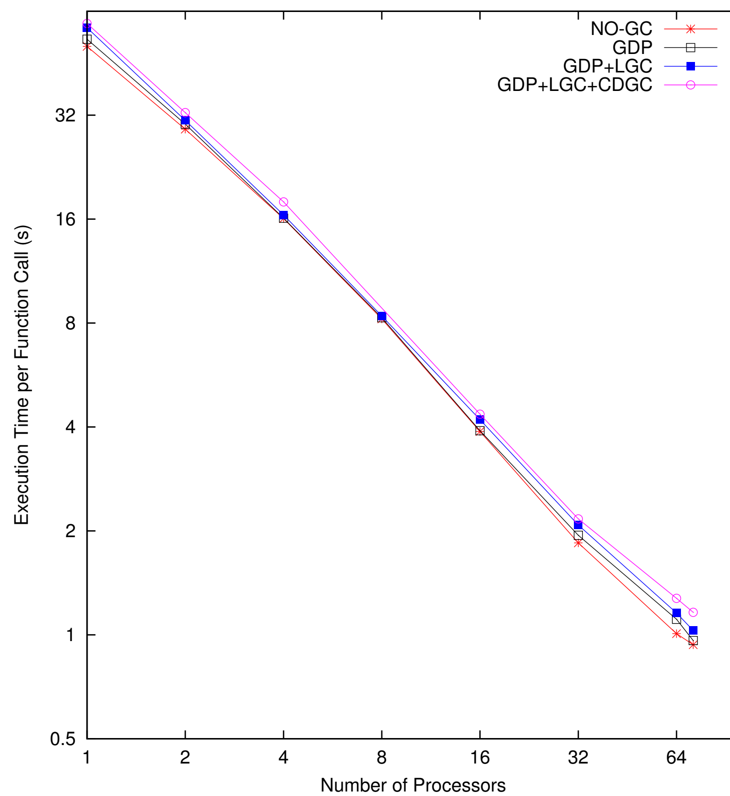

Figure 7.1: Execution time per fit function call vs. the total number of processors. Four kinds of mechanisms are used to evaluate the implementation of our actor garbage collection algorithms.

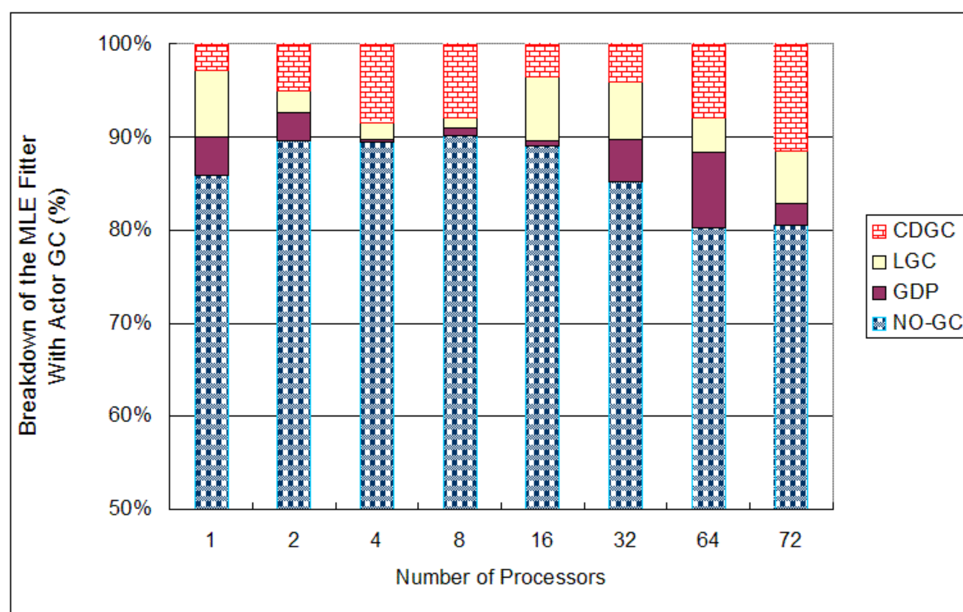

Figure 7.2: Breakdown of the actor garbage collection mechanism.

CHAPTER 8
Conclusions and Future Work

8.1 Conclusions

We have proposed a theory for distributed mobile actor garbage collection. We first described the definition of actor garbage collection and two actor marking algorithms. Then we described how and why our approach can handle mobile actor garbage even though the communication of both the applications and the systems is asynchronous and non-FIFO. The computing models, properties, and proofs were provided as well.

In the first part of the thesis, we redefined actor garbage from the perspective of the reference graph, we showed transformation methods between actor garbage collection and passive object garbage collection, and we devised two actor marking algorithms. The first one is the back pointer algorithm which scans the reference graph only twice for marking, and has linear time complexity of $O(V+E)$ and extra space complexity $O(V+E)$. The other one is the N-color algorithm which relies on disjoint set operations. It only requires one extra marking variable in each actor, and it only scans the reference graph once. It has time complexity of $O(E \lg^* M)$ where M is the number of unblocked actors and $\lg^* M$ is very close to a constant.

The second part of the thesis described how our approach can handle distributed mobile actor garbage collection in a non-intrusive manner. We introduced the concept of pseudo-roots, making actor garbage collection easy to implement. We also formally described the reference listing based algorithm — the pseudo-root approach. Unlike existing actor garbage collection algorithms, the proposed algorithm does not require FIFO communication or stop-the-world synchronization. Furthermore, it supports actor migration. With the help of the pseudo root approach, the proposed distributed snapshot algorithm can detect a casually consistent snapshot for actor garbage collection by two global synchronization events even when some computing hosts are uncooperative. As a consequence, it can tolerate partial failures in the actor systems. Snapshots can be composed in any order and thus the

algorithm can support multi-level hierarchical distributed actor garbage collection. We provided formal computing models and used them to reason about and prove correctness properties of the mobile actor garbage collection algorithms.

8.2 Future Work

This section provides several possible directions for future work.

8.2.1 Resource Access Restrictions and Security Policies

Future research focuses on the idea of resource access restrictions, which is part of distributed resource management and security policies. These issues are essential to the design philosophy of a distributed system. For example, an unblocked actor with migration ability could be live or garbage depending on different resource access restrictions. If the actor is universally prohibited from accessing roots, it can be garbage. On the other hand, it is definitely live if it can possibly migrate to a computing host that supports root accesses. More precisely, an actor in a fully restricted sandbox environment should be considered live if it can possibly migrate to a computing host that provides root accesses, such as input/output services. Another example is that an unblocked actor is live if it can create root actors. All these factors affect the design of distributed actor garbage collection. By applying the resource access restrictions to actors, the live unblocked actor principle may not be true — not every actor has references to the root actors, and potentially live actors can be garbage. Thus they can change the semantics of actor garbage collection.

From the perspective of actor garbage collection, the only interest of the security model is whether an actor is able to access a root through any reference, including any persistent reference [11]. For example, an actor may directly obtain a persistent reference to the local output service while it is created. We said that an actor is *fully restricted* if it does not have any persistent reference to any root, nor

[11] A persistent reference is one which can be obtained through actor migration to a remote computing host or actor creation in a local computing host. The persistent reference cannot be deleted while the actor holding the reference is in the computing host where the actor obtains the persistent reference.

can it obtain such persistent references through actor migration or actor creation. Actors which are not fully restricted are equal to those following the live unblocked actor principle. Notice that the set of active garbage is a subset of fully restricted actors.

Actor migration can be used for runtime system reconfiguration, which can help dynamic load-balancing. It is preferable to have the ability to collect mobile active garbage and the middleware support of automatic actor migration in today's grid or pervasive computing environments — the ability to collect mobile active garbage can save computing power while the middleware support of automatic actor migration can potentially improve total execution time of an application. Without program analysis techniques, a possible solution to support both of them is to guarantee these mobile actors to be always fully restricted. This can be done by only migrating actors among sandbox computing hosts that do not provide any persistent reference.

8.2.2 Large-Scale Applicability

Testing the distributed actor garbage collection algorithms by using more applications in large-scale distributed environments is necessary to further evaluate scalability and performance. The experimental results confirmed that the proposed distributed actor garbage collection algorithms can scale to hundreds of processors. Our next goal is to test it on a distributed environment, consisting of thousands of processors which are connected by a wider area network. Furthermore, streamlining the implementation of the local reference passing mechanism should be a priority for efficiency.

8.2.3 Extension of the Distributed Snapshot Algorithm

The thesis only considers the live unblocked actor principle for the distributed snapshot algorithm. We conjecture that the computing model of the distributed snapshot algorithm can support a system without the live unblocked principle by only adding a new set of roots, which is a subset of the pseudo-root set, PS. However, unblocked actors with migration ability must be considered as roots, or must

be temporarily immobile while taking a snapshot. Otherwise, they are hard to detect and may cause a race condition — no actor gets collected because the snapshot algorithm cannot capture the migrating actors even though the migrating actors are active garbage. Similar problems can happen for actor creation because active garbage may keep creating garbage actors. To conclude, precisely specified privileges of actors (or applications) are required to collect active garbage by the proposed distributed snapshot algorithm.

A possible solution to extend the distributed snapshot algorithm is: 1) to treat all mobile unblocked actors as roots (pseudo-roots) and 2) to forbid actors creating root actors arbitrarily. There are two advantages of this solution. First, it is still non-intrusive. Second, it is correct if there exists any possibility that an actor can obtain persistent references through actor migration. However, the first rule of the solution precludes the ability to collect active garbage even though actors only migrate among sandbox computing hosts which do not provide any persistent root references.

Another possible solution to extend the distributed snapshot algorithm is to make the algorithm intrusive — the ability of actor migration is restricted while taking a snapshot, and arbitrary root actor creation is forbidden as well. Newly created actors can be ignored. If an actor exists in the actor system before the global snapshot procedure starts, it must be selected for local state monitoring even though it is currently migrating to another computing host. To record the global snapshot correctly, two approaches can be used. The first approach is to maintain a snapshot of the migrating actor in the original computing host before the actor migrates, and the migrating actor cannot perform any mutation operation (including migration) unless the original computing confirms that the actor is no longer under local state monitoring. The second approach is to prolong the required time of taking a global snapshot until all migrating actors finish actor migration. Those newly migrated actors cannot perform any computation unless they are put into some actor groups for local state monitoring. Actor migration must be temporarily denied while taking a global snapshot.

8.2.4 Using Static Analysis for Actor Garbage Collection

Static program analysis enables the possibility to collect active garbage even if the application runs on a computing environment with the live unblocked actor principle. For example, actors that do not use any of the persistent references are candidates for active garbage. Furthermore, static program analysis can detect some simple scenarios of distributed infinite loops among actors that cannot potentially use any of the persistent references. In such case, those actors can be reclaimed (terminated) immediately. Static program analysis can also identify the static referential relationship among different types of actors, which can be potentially used to infer the runtime referential relationship. The master-worker application is a typical example — the worker types of actors are garbage if the master types of actors become garbage.

LITERATURE CITED

[1] Saleh E. Abdullahi and A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330–373, 1998.

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

[3] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[4] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

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

[6] David F. Bacon and V. T. Rajan. Concurrent Cycle Collection in Reference Counted Systems. In Jørgen Lindskov Knudsen, editor, *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001*, pages 207–235, Budapest, June 2001.

[7] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[8] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[9] Katherine Barabash, Yoav Ossia, and Erez Petrank. Mostly concurrent garbage collection revisited. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, pages 255–268, Anaheim, CA, November 2003. ACM Press.

[10] Joel F. Bartlett. Mostly-Copying garbage collection picks up generations and C++. Technical Note TN–12, DEC Western Research Laboratory, Palo Alto, CA, October 1989.

[11] David I. Bevan. Distributed garbage collection using reference counting. In *PARLE'87*, volume 258/259 of *Lecture Notes in Computer Science*, pages 176–187, Eindhoven, The Netherlands, June 1987. Springer-Verlag.

[12] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed garbage collection for network objects. Technical Report 116, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1993.

[13] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, MIT Laboratory for Computer Science, May 1977. Technical report MIT/LCS/TR–178.

[14] Stephen M. Blackburn, Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, David S. Munro, and John Zigman. Starting with termination: a methodology for building distributed garbage collection algorithms. *Aust. Comput. Sci. Commun.*, 23(1):20–28, 2001.

[15] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[16] J.-P. Briot. Actalk: a testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP'89)*, pages 109–129. Cambridge University Press, 1989.

[17] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. In *Proceedings of the 2003 ACM SIGPLAN Symposium on Principles of Parallel Programming (PPoPP-03)*, pages 84–94. ACM Press, 2003.

[18] David R. Brownbridge. Cyclic reference counting for combinator machines. In Jean-Pierre Jouannaud, editor, *Record of the 1985 Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 273–288, Nancy, France, September 1985. Springer-Verlag.

[19] Luca Cardelli and Andrew Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, June 2000.

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

[21] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[22] T. W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.

[23] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA/MA'99)*, pages 22–29, Palm Springs, California, October 1999.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 21, pages 498–522. MIT Press/McGraw-Hill, second edition, 2001.

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 3.2, pages 51–56. MIT Press/McGraw-Hill, second edition, 2001.

[26] H. Corporaal, T. Veldman, and A.J. van de Goor. An efficient reference weight-based garbage collection method for distributed systems. In *Proceedings of the PARBASE-90 Conference*, pages 463–465, Miami Beach, March 1990. IEEE Press.

[27] John P. Cummings and Dennis P. Weygand. An Object-Oriented Approach to Partial Wave Analysis. *ArXiv Physics e-prints*, page 24 pp, September 2003.

[28] Peter Dickman. Incremental, distributed orphan detection and actor garbage collection using graph partitioning and Euler cycles. In *WDAG'96*, volume 1151 of *Lecture Notes in Computer Science*, pages 141–158, Bologna, October 1996. Springer-Verlag.

[29] E. W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[30] Edsger W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gastern. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, 1983.

[31] Edsger W. Dijkstra, Leslie Lamport, Alain J. Martin, Carel S. Scholten, and Elisabeth F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[32] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.

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

[34] I Foster and C Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure Second Edition*. Morgan Kaufman, 2004.

[35] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. *Grid Computing: Making the Global Infrastructure a Reality*, chapter The Physiology of the Grid, pages 217–249. Wiley, 2003.

[36] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputing Applications*, 15(3):200–222, 2002.

[37] Cedric Fournet and Georges Gonthier. The Reflexive CHAM and the Join-Calculus. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, 1996.

[38] Nissim Francez. Distributed termination. *ACM Trans. Program. Lang. Syst.*, 2(1):42–55, 1980.

[39] Daniel P. Friedman and David S. Wise. Reference counting can manage the circular environments of mutual recursion. *Information Processing Letters*, 8(1):41–45, January 1979.

[40] Matthew Fuchs. Garbage collection on an open network. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 251–265, Kinross, Scotland, September 1995. Springer-Verlag.

[41] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[42] A. S. Grimshaw, M. A. Humphrey, and A. Natrajan. A philosophical and technical comparison of Legion and Globus. *IBM J. Res. Dev.*, 48(2):233–254, 2004.

[43] Andrew S. Grimshaw, Wm. A. Wulf, and CORPORATE The Legion Team. The legion vision of a worldwide virtual computer. *Commun. ACM*, 40(1):39–45, 1997.

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

[45] Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, and David S. Munro. Garbage collecting the world: One car at a time. *SIGPLAN Not.*, 32(10):162–175, 1997.

[46] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, Lecture Notes in

Computer Science, pages 388–403, St Malo, France, September 1992. Springer-Verlag.

[47] John Hughes. A distributed garbage collection algorithm. In *Record of the 1985 Conference on Functional Programming and Computer Architecture*, volume 201 of *LNCS*, pages 256–272, Nancy, France, September 1985. Springer-Verlag.

[48] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[49] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, 1975.

[50] Eric Jul, Henry M. Levy, Norman C. Hutchinson, and Andrew P. Black. Fine-grained mobility in the Emerald system. *TOCS*, 6(1):109–133, 1988.

[51] Neils-Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 103–115, St Malo, France, September 1992. Springer-Verlag.

[52] Dennis Kafura, Manibrata Mukherji, and Douglas Washabaugh. Concurrent and distributed garbage collection of active objects. *IEEE TPDS*, 6(4):337–350, April 1995.

[53] Dennis Kafura, Doug Washabaugh, and Jeff Nelson. Garbage collection of actors. In *OOPSLA'90 ACM Conference on Object-Oriented Systems, Languages and Applications*, pages 126–134. ACM Press, October 1990.

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

[55] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *International Conference on Distributed Computing Systems*, pages 708–715, Yokohama, June 1992.

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

[57] Fabrice Le Fessant. Detecting distributed cycles of garbage in large-scale systems. In *Principles of Distributed Computing (PODC)*, pages 200–209, Rhodes Island, August 2001.

[58] Fabrice Le Fessant, Ian Piumarta, and Marc Shapiro. An implementation for complete asynchronous distributed garbage collection. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 152–161, Montreal, June 1998. ACM Press.

[59] C. Lermen and Dieter Maurer. A protocol for distributed reference counting. In *ACM Symposium on Lisp and Functional Programming*, ACM SIGPLAN Notices, pages 343–350, Cambridge, MA, August 1986. ACM Press.

[60] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, 1992.

[61] Barbara Liskov and Rivka Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In J. Halpern, editor, *Proceedings of the Fifth Annual ACM Symposium on the Principles on Distributed Computing*, pages 29–39, Calgary, August 1986. ACM Press.

[62] Kaoutar El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela. The Internet Operating System: Middleware for adaptive distributed computing. *International Journal of High Performance Computing Applications (IJHPCA), Special Issue on Scheduling Techniques for Large-Scale Distributed Platforms*, 20(4):467–480, 2006.

[63] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. In *PODC'95 Principles of Distributed Computing*, pages 57–63, 1995.

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

[65] Umesh Maheshwari and Barbara Liskov. Partitioned garbage collection of a large object store. In *Proceedings of SIGMOD'97*, pages 313–323, 1997.

[66] Jeff Matocha and Tracy Camp. A taxonomy of distributed termination detection algorithms. *J. Syst. Softw.*, 43(3):207–221, 1998.

[67] Friedemann Mattern. Virtual time and global states of distributed systems. In *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, October 1989.

[68] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

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

[70] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246. ACM, August 1984.

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

[72] Luc Moreau, Peter Dickman, and Richard Jones. Birrell's distributed reference listing revisited. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1344–1395, 2005.

[73] Jeffrey E. Nelson. Automatic, incremental, on-the-fly garbage collection of actors. Master's thesis, Virginia Polytechnic Institute and State University, 1989.

[74] Object Management Group. CORBA services: Common object services specification version 2. Technical report, Object Management Group, June 1997. http://www.omg.org/corba/.

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

[76] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 129–140, Berlin, June 2002. ACM Press.

[77] José M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *PARLE'91*, volume 505 of *Lecture Notes in Computer Science*, pages 150–165, Eindhoven, The Netherlands, June 1991. Springer-Verlag.

[78] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Tony Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, pages 143–154, Minneapolis, MN, October 2000. ACM Press.

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

[80] Helena Rodrigues and Richard Jones. A cyclic distributed garbage collector for Network Objects. In *WDAG'96*, volume 1151 of *Lecture Notes in Computer Science*, pages 123–140, Bologna, October 1996. Springer-Verlag.

[81] Jon D. Salkild. Implementation and analysis of two reference counting algorithms. Master's thesis, University College, London, 1987.

[82] Davide Sangiorgi. *Communicating and Mobile Systems: the π-calculus*, Robin Milner, Cambridge University Press, Cambridge, 1999, 174 pages, ISBN 0-521-64320-1. *Science of Computer Programming*, 38(1–3):151–153, August 2000.

[83] Y. Sato, Michiko Inoue, Toshimitsu Masuzawa, and Hideo Fujiwara. A snapshot algorithm for distributed mobile systems. In *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, pages 734–743. IEEE Computer Society, 1996.

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

[85] Marc Shapiro. A fault-tolerant, scalable, low-overhead distributed garbage collection protocol. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 208–217, Pisa, September 1991.

[86] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapports de Recherche 1799, INRIA, November 1992.

[87] Darko Stefanovic, Matthew Hertz, Stephen Blackburn, Kathryn McKinley, and J. Eliot Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*, pages 25–36, Berlin, June 2002.

[88] Darko Stefanovic, Kathryn S. McKinley, and J. Eliot B. Moss. On models for object lifetime distributions. In Tony Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, pages 137–142, Minneapolis, MN, October 2000. ACM Press.

[89] Yemini S. Strom R. E. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

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

[91] W.T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A New Major SETI Project based on project SERENDIP data and 100,000 Personal Computers. In *Proceedings of the Fifth International Conference on Bioastronomy*. Editrice Compositori, Bologna, Italy, 1997.

[92] Sun Microsystems Inc. – JavaSoft. Remote Method Invocation Specification, 1996. Work in progress. http://www.javasoft.com/products/jdk/rmi/.

[93] Boleslaw K. Szymanski, Yuan Shi, and Noah S. Prywes. Synchronized distributed termination. *IEEE Trans. Software Eng.*, 11(10):1136–1140, 1985.

[94] Boleslaw K. Szymanski, Yuan Shi, and Noah S. Prywes. Terminating iterative solution of simultaneous equations in distributed message passing systems. In *Proceedings of the Fourth Annual ACM Symposium on the Principles of Distributed Computing*, pages 287–292, Amsterdam, Netherlands, 1985.

[95] Gerard Tel and Friedemann Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Transactions on Programming Languages and Systems*, 15(1):1–35, January 1993.

[96] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

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

[98] Abhay Vardhan and Gul Agha. Using passive object garbage collection algorithms. In *ISMM'02*, ACM SIGPLAN Notices, pages 106–113, Berlin, June 2002. ACM Press.

[99] Carlos A. Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, U. of Illinois at Urbana-Champaign, May 2001.

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

[101] Luis Veiga and Paulo Ferreira. Asynchronous complete distributed garbage collection. In Ozalp Babaoglu and Keith Marzullo, editors, *IPDPS 2005*, Denver, Colorado, USA, April 2005.

[102] N. Venkatasubramanian, G. Agha, and C. Talcott. Scalable distributed garbage collection for systems of active objects. In *IWMM'92*, volume 637 of *Lecture Notes in Computer Science*, pages 134–147. Springer-Verlag, 1992.

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

[104] Wei-Jen Wang, Kaoutar El Maghraoui, John Cummings, Jim Napolitano, Boleslaw K. Szymanski, and Carlos A. Varela. A middleware framework for maximum likelihood evaluation over dynamic grids. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 8 pp, Amsterdam, Netherlands, December 2006.

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

[106] Wei-Jen Wang and Carlos A. Varela. A non-blocking snapshot algorithm for distributed garbage collection of mobile active objects. Technical Report 06-15, Dept. of Computer Science, R.P.I., October 2006. Submitted to IEEE TPDS.

[107] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87*, volume 258/259 of *Lecture Notes in Computer Science*, pages 432–443, Eindhoven, The Netherlands, June 1987. Springer-Verlag.

[108] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994.

[109] Worldwide Computing Laboratory. The SALSA Programming Language, 2006. http://wcl.cs.rpi.edu/salsa/.

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

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

[112] Bojan Zagrovic, Christopher D. Snow, Michael R. Shirts, and Vijay S. Pande. Simulation of Folding of a Small Alpha-helical Protein in Atomistic Detail using Worldwide Distributed Computing. *Journal of Molecular Biology*, 323:927–937, 2002.