© 2015 Shuheng Huang

PERFORMANCE OF A HIERARCHICAL DISTRIBUTED GARBAGE COLLECTION ALGORITHM IN ACTORFOUNDRY

BY

SHUHENG HUANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor Gul Agha

ABSTRACT

Automatic garbage collection is an essential feature so that programs can reclaim resources without the need for manual input. This feature is present in many modern languages and is a common subject of research. However, in parallel and distributed environments, programmer-controlled resource reclamation is highly error-prone. As the scale of programs increase, automatic garbage collection is of paramount importance for efficient and error-free execution.

Garbage collection in the context of actor systems is especially difficult because actors are active objects and may not be garbage even if there are no references to it. An additional difficulty is to perform garbage collection on active objects without halting the current computation.

This thesis implements one of the proposed algorithms which can solve the problem of garbage collection in distributed actor systems. This study also explores how parameters in this algorithm along with how the topology of an actor system affect the garbage collection. This was implemented on an existing actor framework in order to highlight key factors in the algorithm's performance. The design details and insights gained from the results of these tests are then discussed.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Professor Gul Agha, for his continuous support throughout the entire development of this thesis. His guidance, expertise, and patience were critical to the successful completion of this thesis.

I would also like to also thank Minas Charalambides who not only contributed greatly through thought-provoking conversations but has mentored me throughout the year. Working with Minas as a TA was an enlightening and humbling experience.

Finally, I would like to thank my parents and cousin for all their support over the last two years and my life in general. My time at the University of Illinois was not without its difficulties but my family has helped me through it all.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Overview 1
1.2 Thesis Outline 2

CHAPTER 2 BACKGROUND 3
2.1 The Actor Model 3
2.2 Garbage Collection 5

CHAPTER 3 PREVIOUS WORK IN GARBAGE COLLECTION . . 8
3.1 Garbage Collection Strategies 8
3.2 GC Variations 9
3.3 Languages 11
3.4 Actor System Garbage Collection 15
3.5 Hierarchical Distributed Garbage Collection 16

CHAPTER 4 IMPLEMENTATION 20
4.1 ActorFoundry 20
4.2 Architecture 21
4.3 HDGC in ActorFoundry 25
4.4 Listings 31

CHAPTER 5 RESULTS 33
5.1 Test Setup 33
5.2 Single Node 34
5.3 Multi-Node 42

CHAPTER 6 CONCLUSION 45
6.1 Future Work 46

REFERENCES 47

CHAPTER 1

INTRODUCTION

1.1 Overview

The main benefit of garbage collection is reclaiming resources to improve performance. However, manual garbage collection is not only highly error-prone but also imposes a non-trivial burden on the programmer. Automatic garbage collection alleviates both of these issues. The downside to garbage collection is that there are overhead storage costs for maintaining information about objects as well as computational costs to determine object status. As a system scales these drawbacks are overshadowed by the benefits automatic garbage collection brings.

In a parallel/distributed environment, objects and processes are continuously created and the need to reclaim these resources is of paramount importance. Garbage collection for the Actor Model [1] is difficult because in addition to the challenges of garbage collection the problem of distributed resources and active objects [2] need to be considered. However, traditional distributed garbage collection algorithms cannot be applied to the actors model because actors are active objects.

This thesis implements the Hierarchical Distributed Garbage Collection algorithm (HDGC) [3, 4] and explores how parameters in the algorithm affect the performance of the algorithm. This algorithm is implemented in the actor framework ActorFoundry [5] which is based on Java™. The main contribution of this thesis is in the implementation of the HDGC algorithm, analysis of the algorithm, and finally implications of the results for actor program design. The performance is evaluated for both local and distributed systems for a changing actor system topography.

1.2 Thesis Outline

The thesis is organized as follows. Section 2 covers the Actor Model and garbage collection of actor systems. Section 3 is a review of previous work in garbage collection and covers the HDGC algorithm proposed by Nalini Venkatasubramanian. Section 4 explains the implementation of the algorithm in ActorFoundry along with shortcomings and differences in the implementation versus the original algorithm. Section 5 presents the test setup along with the results along with an analysis of the results. Finally, chapter 6 outlines conclusions along with areas for potential future investigation.

CHAPTER 2

BACKGROUND

This chapter introduces the Actor Model as well as gives a definition of garbage in an actor system.

2.1 The Actor Model

2.1.1 History

The Actor Model is a model for concurrent computation through Actor objects which was first specified by Carl Hewitt [6] and later, further developed by Gul Agha [1]. The modern operational semantics of the Actor model follows closely from the work of Gul Agha. The paradigm of Actors provides many advantages in parallel and distributed application design over other models such as object-oriented design. As parallel and distributed computation has become widespread, the Actor model has also become more widely used in various languages and frameworks. A more in-depth review of the Actor model along with its design can be found by Rajesh Karmani [7].

2.1.2 Operational Semantics

In the Actor Model the universe consists of actor objects and messages. Each actor is a self-contained, autonomous agent which encapsulates its state, behavior, and processing power. The way actors communicate with each other and the outside world is through messages. Specifically, actors communicate with each other through a globally unique *name* assigned during creation and can only be communicated through messages. Figure 2.1 shows an implementation of the Actor Model with the processing power implemented as a thread. Actors are executed concurrently and messages are sent asyn-

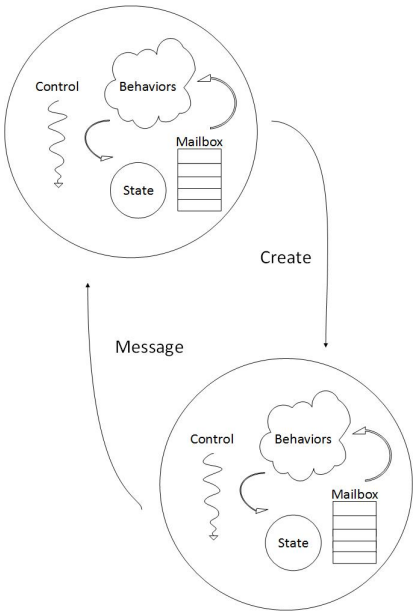

Figure 2.1: Actors are self-contained objects which contains its own state, behaviors, and control.

chronously. Messages are delivered to an internal mailbox and buffered until the actor processes the message.

The behavior and state of an actor is driven by the messages it receives. Messages are received and buffered in a mailbox and eventually executed. When an actor handles a message one or more of the following events may be triggered:

- Actor creation
- Modify its state
- Send a message to another actor

The Actor Model naturally lends itself to parallel and distributed applications because of the properties inherent to its design. Due to each actor having its own state which cannot be accessed or modified by any other actor there is no worry of unsafe concurrent access to shared memory. The *name* of an actor is location transparent and as such can be used the same whether in a local or distributed setting. Finally because messages are handled asynchronously and are buffered upon receiving with execution not guaranteed immediately upon arrival, actor programs can be used in a distributed setting without modifications.

Although there is a conceptual difference between the *name* of an actor and the *mailbox* of an actor, in practice they refer to the same thing. The *name* of an actor is a handle for the *mailbox* of the actor that a message is intended for. In the context of referring to an actor and message sending the terms *name* and *mailbox* will be used interchangeably in this thesis.

2.2 Garbage Collection

2.2.1 Passive Object Garbage Collection

Functional languages such as Lisp and Scheme have had automatic garbage collection for decades. Erlang is a functional language which is built around the Actor Model and does GC per process. Similarly, many object oriented languages also support garbage collection. Java, an object-oriented language, has built-in support for multiple automatic garbage collectors which all use some form of a tracing collector. Python on the other hand uses a reference counting scheme. A more detailed explanation of garbage collection schemes in various languages are covered in the following chapter.

Garbage collection for passive objects in object-oriented systems can be viewed as a graph problem. This graph is a directed graph where every object is a node and each reference to that object is an edge. The graph of objects and their references is called the *object-reference graph*. Root objects in this are those which can never be garbage such as static objects, input-output objects, and threads of control. These root objects are the starting point for the reference graph traversal. Objects which are unreachable from any root objects are unable to affect the execution of the program and can be safely garbage collected. Periodically the garbage collection process is executed and the resources for garbage objects are reclaimed.

However, garbage collection does have an associated cost both in terms of storage overhead and computation cost. Many garbage collection algorithms also requires a stop-the-world pause which pauses user computation. These costs along with details of the schemes are covered in chapter 3.

2.2.2 Garbage in Actor Systems

The above garbage collection scheme for object-oriented systems cannot be applied directly to actor systems. The root set for the object-reference graph in a object-oriented system includes threads of control but in the actor model each actor encapsulates its own thread of control. The root set in an actor system can only be actor themselves as state and objects are also encapsulated. The following definitions of garbage in an actor system are based on the work of Kafura et al. [8].

In an actor system the root set consists of the root actor which is the actor created at the beginning of the computation. This actor receives an initial message and as part of its behavior may create more actors. Similarly, actors which communicate with the external environment are equivalent to input-output objects and thus are part of the root set. Any other actors which are not known either directly or indirectly by a root actor can therefore be considered garbage as they cannot influence future computation.

In order to formally define garbage in an actor system the following definitions are necessary. An *acquaintance* of actor A is an actor for which actor A knows the name of. Names are assigned at creation and can only be communicated through messages. However, acquaintances can be lost if the name is not kept. An *inverse-acquaintance* of actor A is an actor which holds the name for actor A. It then follows from this definition that if actor A is an acquaintance of actor B then actor B is an inverse-acquaintance of actor A.

An *active* actor is an actor which is either processing a message or has messages left that need to be processed. An *inactive* actor is an actor which is not currently processing a message nor has any messages to be processed. Any *inactive* actor which is not known either directly or indirectly by an active actor is *permanently inactive*.

The set of non-garbage actors in an actor system is called the *reachable set*. Actors in the *reachable set* are defined recursively as follows:

- Root actors are *reachable*.
- Acquaintances of live actors are *reachable*.
- Inverse-acquaintances of *reachable* actors which are not permanently inactive are *reachable*.

A result of this definition is the inverse-acquaintances relationship also influence reachability because actors which are not *reachable* from the root set but are not inactive are *reachable*. could send their *name* to a *reachable* acquaintance. In this case the topology changes and the previously unreachable actor is now *reachable*. However if the inverse-acquaintance is inactive then it is always unreachable because it can never communicate its *name* to any *reachable* actor.

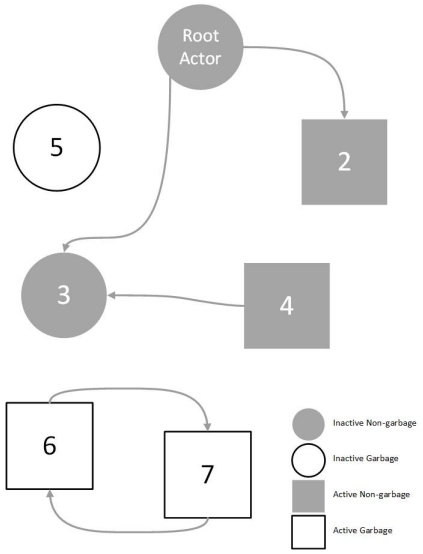

Figure 2.2: Inverse-acquaintance non-garbage example.

Figure 2.2 shows an example actor system with active and inactive actors with their GC labels. Actors 2 and 3 are non-garbage regardless of their status because they are both reachable from the root actor. Actor 4 is garbage because it is unreachable and inactive. Actor 5 is non-garbage despite not being directly reachable because it is active and can potentially communicate its address to a reachable actor. Actors 6 and 7 are both garbage despite being active because they are not reachable and can not become reachable.

From this definition, any unreachable actor is therefore garbage and can be safely collected. A key property of actors labeled garbage is that they will remain garbage and there is no way for them to become non-garbage. There is no way for the root actor to ever communicate with a garbage actor so once an actor is marked garbage by the above definition it will remain garbage forever.

CHAPTER 3

PREVIOUS WORK IN GARBAGE COLLECTION

This chapter begins with a summary of well established garbage collection strategies and common variations. Next is a review of garbage collectors in a number of languages. The section after describes work done in garbage collection of actor systems. Finally a description of the Hierarchical Distributed Garbage Collection algorithm which this thesis implements is explained.

3.1 Garbage Collection Strategies

3.1.1 Reference Counting

Reference counting is a common strategy for garbage collection due to its simplicity. In reference counting every object maintains a count of how many other objects hold a reference to it. This reference count is incremented whenever another reference is created and decremented when a reference is lost. Once the reference count for an object reaches zero that means that there are no objects which know of it and therefore it can be reclaimed as garbage. This particular trait of reference counting means that garbage is collected incrementally and the system is kept garbage free.

The naive implementation of this strategy has the advantage of being simple to implement and also being able to detect garbage as soon as there are no references to it. However, there is a high overhead of having to update the reference count continuously and a simple implementation is unable to detect cycles. A cycle of objects with references to each other will not have their reference count drop to zero so another method needs to be applied to detect these cycles.

Reference counting variants such as weighted reference counting and indirect reference counting are effective solutions to garbage collection in parallel

and distributed systems.

3.1.2 Mark-and-Sweep

The mark-and-sweep strategy for garbage collection was first described by John McCarthy in 1960 as part of a paper on the implementation of LISP [9]. In the original algorithm, mark-and-sweep is run when there is no more free memory to be allocated. Every base register (root set) is marked and from this every register (object) which can be reached is also marked. Once this sweep is finished, any unmarked register is considered to be free memory and can be reused.

This algorithm requires a stop-the-world pause and is only run periodically but is able to handle cycles directly. Compared to reference counting, mark-and-sweep is more computationally intensive due to a costly sweep of all objects. The cost is of a full mark-and-sweep is determined by the size of the heap.

3.2 GC Variations

Building on the reference counting and mark-and-sweep strategies there are numerous improvements and hybrid strategies. Both reference counting and mark-and-sweep in their most naive implementation have deficiencies which the following variations attempt to address.

3.2.1 Hybrid

A simple method of overcoming the downsides of both the mark-and-sweep algorithm and the reference counting algorithm is to utilize a hybrid scheme. An example of a hybrid garbage collector is the Python garbage collector [10]. Python uses a reference counting scheme but also has a periodic cyclic garbage collector. Additionally Python allows the programmer to specify objects which can only be manually garbage collected.

3.2.2 Compaction

In both reference counting and mark-and-sweep schemes after garbage is collected the heap is fragmented. This fragmentation makes future allocation of memory of a large block difficult. Compaction refers to the relocation of objects and free space to eliminate fragmentation of the heap. Some garbage collection algorithms include a separate step for compaction of the heap while others include as a side effect of the garbage collection.

3.2.3 Copy Collection

The copy collector scheme partitions the heap into two spaces; from-space and to-space. When garbage collection needs to be performed, all live objects are copied from one space to another and objects in the first space are garbage collected. This has the impact of also compacting the memory during the copy. A downside to this approach is that the heap is split in half.

3.2.4 Generational Garbage Collection

Hewitt and Lieberman first proposed Generational Garbage Collection [11] in 1980 which makes use of the temporal locality property to improve the efficiency of garbage collection. In generational garbage collection objects are partitioned into spaces based on duration alive and different GC policies are applied to the different sets. This is based on the heuristic that newer objects are more likely to be garbage whereas longer lived objects will remain non-garbage.

For example, objects initially created are placed in the first generation space (G1). Once a garbage collection is initiated, surviving objects will be moved to the second generation space. Garbage collection in G1 is performed much more frequently based on the temporal locality heuristic. Generational garbage collection not only exploits temporal locality but is faster than the traditional mark-and-sweep because the traced set is smaller. One problem with the generational approach is handling references which cross generation boundaries.

There are multiple models regarding when an object should be *tenured* from one space to another. However, it is argued by Clinger and Hansen [12]

that if an exponentially decreasing probability function is used to model an object's lifetime then how long an object has been alive does not give any insight towards how much longer it will live. A similar argument for why the age of an object is not a useful heuristic in predicting object lifetime was stated by Henry Baker [13]. Despite the potential erroneous underlying assumption, both papers state that generational garbage collection has many secondary effects which make it more efficient than non-generational schemes. Effects such as locality of references and shorter stop-the-world pauses are inherent in generational garbage collection and are improvements over non-generational garbage collection.

Many languages which use the generational garbage collection scheme use adaptive tenure policies. The garbage collector changes the tenure policy based on statistics the profiler learns about the application.

One optimization for the tenure policy is the concept of pre-tenure. Objects which are pre-tenured are directly allocated into the old generation space. This optimization reduces the number of objects which need to be traced and copied. A dynamic pre-tenuring optimization was proposed by Timothy Harris [14]. In dynamic pre-tenuring, statistics are gathered during runtime and objects are pre-tenured using this data. Objects are sampled and information such as class, life-time, how many allocated, and methods invoked are collected. Using this information it is possible to know, for example, that all objects of a given class always reach tenure and can directly be placed in that set.

3.3 Languages

3.3.1 Java

Depending on the version of Java there are multiple different garbage collectors which are available for use. These garbage collectors all use a mark-and-sweep strategy and differ mainly in the duration of stop-the-world pauses and resources required. In addition to the type of collector used, it is possible to specify the heap size and generation size. A detailed summary of the various garbage collectors was released by Sun Microsystems [15]

Java also features weak references which allows the programmer more con-

trol over the underlying garbage collection mechanism. Weak references are references which do not prevent garbage collection and when used to fetch objects may end up failing. There are four different levels of references ranging from strong, soft, weak, and phantom.

Java's generational garbage collectors also which offer programmers the choice of various tenure strategies. Work such as that done by Blackburn et al [16] show that pre-tenuring can easily be added to Java and can give a significant performance boost to the garbage collector. Blackburn concludes that because most Java programs have high lifetime homogeneity, pre-tenure predictions are very accurate. Only two statistics need to be collected about objects; age and time of death. From this information each allocation site is given a short, long, or immortal label. This label accurately predicts the behavior of objects created from each site and can be used as a pre-tenure heuristic.

3.3.2 Scala

Scala is programming language which offers both functional and object-oriented paradigms. Scala is compiled to Java bytecode and run on a JVM. Because Scala runs on a JVM, Scala uses the garbage collectors present in Java. Scala has featured support for actors through different libraries such as scala.actors, Akka, Lift, and Scalaz. Currently, Akka has become the standard for actors in Scala.

3.3.3 Akka

Akka [17] is a toolkit for Java and Scala and extends the two languages with an actor model implementation. However, the Akka toolkit features some functionality which deviates from the original Actor Model schematics. For example, it is possible to create a *pool* of actors that share the same mailbox.

The garbage collector for Akka relies on the underlying JVM and thus is unable to automatically collect actors for both Java and Scala implementations. Users must manually mark actors for garbage collection otherwise the memory is leaked. This is true for both local and distributed actors in Akka.

3.3.4 Erlang

Erlang [18] is a functional language designed by Ericsson in 1986 and was designed around concurrent/distributed systems using the Actor Model. Properties of the actor model such as message passing, location independent naming, unshared state, etc are all supported by design in Erlang. Functional language design such as immutable objects allows for faster and easier garbage collection. For example, due to immutable objects, in a generational scheme it is not possible for references to cross generation boundaries. Objects created can never point to objects newer than itself. Because objects cannot be modified the garbage collector does not need to worry about references changing during GC.

The garbage collector is Erlang is unique in that the garbage collection is per process and it is possible to specify the heap size of the process being spawned. This optimization is useful because garbage collection pauses are tiny and the sets traced are small. A programmer with knowledge of a process's lifetime and heap usage could potentially bypass garbage collection for that process.

One example garbage collector present in Erlang which is simple yet can be easily extended is the One-Pass Real-Time Generational Mark-Sweep Garbage Collector [19]. This garbage collector utilizes the design of the language to facilitate easier garbage collection. Assume the heap is organized with a high end and a low end such that objects are allocated in the first location available in the high end. Due to objects being immutable, all references for an object must point to objects which are lower than it in the heap. A sweep begins from the highend towards the low-end with the first object being marked as non-garbage. Any references encountered are followed and the object pointed to is marked. Any object encountered which is not marked is garbage and can be immediately reclaimed. This is because once the sweep moves to a lower portion of the heap there is possibility of that object being referenced to.

However, using the above algorithm means that memory locations once reclaimed cannot be reused due to the initial assumptions of older objects being placed higher. This problem can be fixed by keeping a history of allocated objects and traversing that rather than traversing the heap from high to low. Once objects are reclaimed the pointers in the history only need

to be changed in order to remain consistent. This mark-and-sweep scheme can be made generational by sweeping only the newer objects using some threshold for how much of the heap to sweep.

Despite the ease of garbage collection in Erlang, this only applies to memory allocated by Erlang actors and not actors themselves. The actor must be manually deallocated and cannot be automatically garbage collected.

3.3.5 Pony

Pony is a very new language which, like Erlang, is designed around the Actor Model concept. A unique feature of Pony is that the language can concurrently garbage collect actors without requiring manual termination. Additions to the Actor Model such as casual messaging help facilitate the garbage collection process.

The algorithm Pony-ORCA [20] takes advantage of the Pony language and its design for automatic actor garbage collection. Pony-ORCA maintains a reference count for both objects owned by actors and actors themselves. Garbage collection can be run for each actor independently and local objects with a reference count of zero can be collected due to implicit ownership. Similarly, when the reference for an actor is zero it can be safely garbage collected because Pony guarantees there are no other actor with a reference.

3.3.6 Orleans

Orleans [21] is a distributed actor framework much like Akka and Erlang. This very new framework was built entirely for distributed systems and scalability. The distributed model is hidden from the programmer and the Actor Model is used instead. Orleans does not have explicit actor creation and actors instances are created when message intended for said actor are sent. This design means that actor failure is transparent and additional actors are created automatically.

The Actor Model used by Orleans is a virtual "actor space" which allows actors to invoke any actors in the system. The location of the actor and instance creation are handled without programmer intervention. Garbage collection actors in this language is much simpler than in other languages.

Because there are no references to specific actors, any actor which is idle will eventually be garbage collected.

3.4 Actor System Garbage Collection

Kafura in addition to defining garbage in an actor system also presented the *Push-Pull Marking Algorithm* and *Is Black Algorithm*. The basis for both algorithms is the following color scheme for marking actors.

- White: Actors not reachable from a root actor.
- Gray: Actors reachable from a root actor, but cannot become active.
- Black: Non-garbage actors which are either root or are reachable from a root and potentially active.

The below coloring rules defined by Jeffrey Nelson [22] is be used to assign colors to actors:

Algorithm 3.4 Nelson's Coloring Rules

1. All actors are colored white, with the exception of root actors which are colored black.
2. Repeat the following rules until no more markings are made:

   Rule 1: Color black all acquaintances of black actors.

   Rule 2: Color black all inverse acquaintances of black actors if the inverse acquaintance is not blocked.

   Rule 3: Color gray all inverse acquaintances of black actors if the inverse acquaintance is blocked.

   Rule 4: Color black all inverse acquaintances of gray actors if the inverse acquaintance is not blocked.

   Rule 5: Color gray all inverse acquaintances of gray actors if the inverse acquaintance is blocked.
3. Actors colored black are not garbage. Gray and white actors are garbage and can be reclaimed.

The *Push-Pull Marking Algorithm* uses two routines, *Push* and *Pull*, to move actors between the three color sets. The pusher acts on actors in the white set and moves actors into either the gray set or the black set. If the white actor is active and an acquaintance of a black or gray actor then the actor is moved to the black set. If the white actor is inactive and an acquaintance of a black or gray actor then the actor is moved to the gray set. The puller examines actors in the black set and moves non-black acquaintances to the black set.

The *Is Black Algorithm* only uses two colors, black and white, but requires an additional *visit* field. First, all acquaintances of a black actor are colored black. Next, a depth first search is performed on all active white actors. If any acquaintance of an active white actor is black then the actor itself is moved to the black set. The *visit* field is used in this search to detect a cycle and terminate the search.

Vardan [23] also presents a distributed garbage collection algorithm which features a transformation of the actor graph into one which can be collected using a passive garbage collector. Previously it was stated that passive object garbage collectors cannot be applied to actors because actors are active objects. The transformation technique and proof of concept was implemented on ActorFoundry and uses the same definition of garbage as in Section 2.2.2.

More recently, Clebsch and Drossopoulou [24] present a concurrent actor garbage collection algorithm called Message-based Actor Collection(MAC). MAC uses a deferred reference counting scheme along with casual messaging to maintain correct reference counts for the actor system. There is an actor dedicated to the detection of cyclic garbage which reference counting alone cannot handle. Every actor maintains its own view of the topology of the system using reference counts and upon being blocked sends its view of the topology to the cyclic garbage detecting actor. This actor determines when blocked, cyclic actors are present and allows these cycles to be collected.

3.5 Hierarchical Distributed Garbage Collection

The Hierarchical Distributed Garbage Collection (HDGC) algorithm is a non-halting, distributed garbage collection algorithm. The HDGC algorithm uses the definition of garbage from Section 2.2.2 and uses a generational mark-

and-sweep strategy. The algorithm consists of two phases, Network Clearance and Distributed Generation Scavenge (DGS). The network communication channels are assumed to be FIFO and the topology of the distributed system is static. The distributed system is divided into clusters with the smallest unit being a single processor. Finally, the topology is assumed to be a two dimensional mesh.

To begin a garbage collection phase a message is broadcast to all nodes from the initiator. Next, a network clearance phase is done in order to take a snapshot of the global state at the time of garbage collection. This is necessary so that messages in transit are accounted for during garbage collection. The names of actors can be communicated through messages resulting in a dynamic toplogy. The snapshot is done by propagating a "forward bulldoze" message. Once the "forward" message reaches the last node in the mesh, the last node sends a "backwards bulldoze" message. Once the "backwards" message reaches the initiator the snapshot is complete. Messages are tagged *new* and *old* based on when the message was sent so that during garbage collector the state is consistent.

Figure 3.1: An in-progress forward message wave.

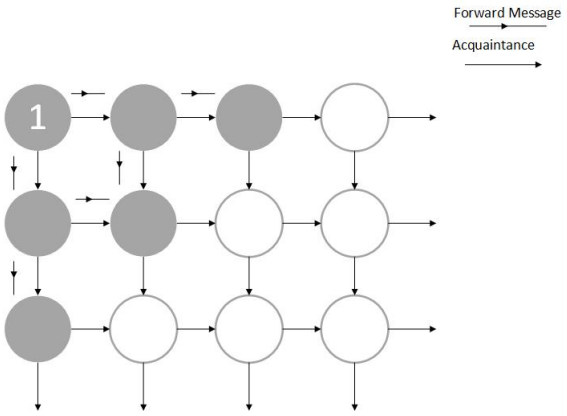

Figure 3.1 shows a "forward" wave in-progress. Each circle represents an individual node in the system. Node 1 began the message wave process. Nodes which are colored can only send *new* messages since they have received the "forward" message.

Following the snapshot, the distributed scavenge phase marks non-garbage actors as touched. Touched indicates that the actor is reachable and should not be garbage collected. Suspended (permanently inactive) and untouched (unreachable) states indicate the actor can be safely garbage collected. A

pseudo-code outline of the scavenge phase is reproduced below in Listing 3.1. The pseudo-code provides a rough outline of the algorithm with details regarding tags and counters omitted.

After the scavenge phase, another message is broadcast to indicate the end of the scavenge phase so that each node can then finalize garbage collection. This step consists of clearing the memory of garbage actors, reseting the state of alive actors, and incrementing the survived count. The survived count is used as a counter for tenure and is incremented for every surviving actor.

Listing 3.1: Actor Constructor Overloading

```
define traverseRoot(rootActor)
  set status of rootActor to touched
    for every actor A in rootset:
      set status of A to touched
      for every forward acquaintance facq of A:
        (scavengeForward(facq, A))
      for every inverse acquaintance iacq of A:
        (scavengeInverse(iacq, A))
      when (receiveAllAck(A)) (backPropogate(rootActor))
  when (receiveAllAck(rootActor)) (finish)

define scavengeForward(actor, sender)
  if actor is not touched then:
    set status of actor to touched
    for every forward acquaintance facq of actor:
      (scavengeForward(facq, actor))
    for every inverse acquaintance iacq of actor:
      (scavengeInverse(iacq, actor))
    when (receiveAllAck(actor)) (backPropogate(sender))

define scavengeInverse(actor, sender)
  if actor is active then:
    if actor is not touched then:
      set status of actor to touched
      for every forward acquaintance facq of actor:
        (scavengeForward(facq, actor))
      for every inverse acquaintance iacq of actor:
        (scavengeInverse(iacq, actor))
      when (receiveAllAck(actor)) (backPropogate(sender))
    endif
  else
    set status of actor to suspended
    for every inverse acquaintance iacq of actor:
      (scavengeInverse(iacq, actor))
  endif
```

CHAPTER 4

IMPLEMENTATION

This chapter starts with an introduction to ActorFoundry then covers parts of ActorFoundry which influence the implementation of the HDGC algorithm. Finally the implementation of the HDGC algorithm is covered.

4.1 ActorFoundry

ActorFoundry [5] is a actor framework which extends Java™ and was developed at the University of Illinois Urbana-Champaign by the Open Systems Laboratory. The features of the Actor Model discussed in Chapter 2 are all present in ActorFoundry.

The implementation of ActorFoundry uses various strategies to transform native Java™ into the semantic properties of the Actor Model. In order to efficiently encapsulate a thread of control for each actor, Actor foundry uses the Kilim framework [25, 26]. Kilim features a byte-code post-processor to map multiple actors to a single native thread for performance, allows user-level schedulers, and safe messaging between threads using "mailboxes". In addition to Kilim, ActorFoundry features a compiler which builds an "Executor" for each actor class. This is done to separate the interface of actors from the execution of an actor object.

4.1.1 Hello World Example

The code fragment in Listing 4.1 illustrates a couple of features of actor programs in ActorFoundry. An actor is defined using a class which extends the Actor class. In the above code there are two actor definitions; HelloActor and WorldActor. The initial actor that ActorFoundry should creates is the HelloActor and the initial message sent should be "boot" with no parameters.

Since the initial actor is specified either through command line or via a configuration file it is not possible to send object parameters initially.

Methods which the actor can execute by receiving messages are annotated with "@message". *create* is used to create a new Actor and returns an *ActorName* which is location transparent and can be passed around via messages. Notice that if the *ActorName* reference is not stored then there is no way for that actor to receive further messages. Finally, *call* and *send* are the two mechanisms used to send messages. *call* is a synchronous send and is blocking while *send* is asynchronous. In this example because "Hello" is sent synchronously to *stdout* there is no possibility of "World!Hello " to be output as a result of this program.

An in depth review of the ActorFoundry framework along with comparisons to other actor frameworks was done by Rajesh Karmani [27]. The following subsections briefly cover parts of ActorFoundry which were key to the implementation of the HDGC algorithm.

4.2 Architecture

ActorFoundry contains many features and includes all the standard actor semantics such as location independent naming, fair scheduling, encapsulation, transparent migration, etc. In the following section only the components relevant to the implementation of the HDGC algorithm are covered.

4.2.1 ActorManager

The ActorManager is the main component of the ActorFoundry framework. The ActorManager handles creation of new actors, delivery of messages, migration of actors, and holds references to services such as remote sessions and the the scheduler. The ActorManager is created by the ActorFoundry and the initial actor if one is specified along with its parameters is created.

There are two ActorManagers available in ActorFoundry but additional types can be created to extend the functionality provided. The LocalActorManager does not feature support for a distributed system but has performance optimizations for how messages are handled. The BasicActorManager features network capabilities such as remote actor creation, communication

with remote actors, and migration of actors. This thesis uses the BasicActorManager due to its network support.

The ActorManager in previous versions had a list of actors which it managed but in the latest version this was removed and the ownership of these actors was moved to the scheduler. However, the ActorManager still maintains a list of *Receptionists*. The *Receptionist* list consists of actors which are either not user created, created due to a remote ActorManager, or are actors whose references have been communicated to another node.

4.2.2 Scheduler

ActorFoundry uses Kilim threads which allows it to have a custom scheduler for said threads. It is inefficient for each actor to be handled by a native thread and could easily result in the creation of many short lived threads. However Kilim allows each actor to contain a thread of control called a *Task* and maps multiple *Tasks* to each native thread.

The most up to date scheduler available in ActorFoundry is the ContinuationBasedScheduler and thus the implementation of the HDGC algorithm was tested using this scheduler. The scheduler contains a pool of native threads along with a queue of Tasks each of which corresponds to an actor or a service. Whenever an actor is created it is scheduled with the scheduler. Services include network handling and cleanup handlers which are not actors but are scheduled in the same way. This design allows the cost of thread creation to be handled upfront and allows the scheduler to allocate runtime fairly. In order for a *Task* to be garbage collected by the JVM it is necessary to remove it from the scheduler queue.

4.2.3 Actor, ActorImpl, and ActorExecutor

In order to handle the actor abstraction ActorFoundry has a ActorName class, ActorImpl class, ActorExecutor class, and the user defined Actor classes. The user defined actors only need to extend the base Actor class to have access to actor functionality. In order to abstract the underlying implementation from the user, the ActorImpl and ActorExecutor are required. These two classes along with the scheduler is what allows actors in the ActorFoundry

framework to behave semantically as specified in the Actor Model.

The ActorName is a globally unique *name* generated when an actor is created. Actor creation is asynchronous and the ActorName is able to receive messages immediately even for remote actor creation.

The Actor class is extended to create a new type of actor as was shown in 4.1. Actors defined in this way can be created by the ActorManager through a *create* request and support a default constructor or an explicit constructor with parameters. Methods appended with "@message" are methods which can be called by another called using *call* or *send*.

The Listing 4.2 shows off a more complex actor which features a non-default constructor and method overloading. The remote ActorManager code is explained in 4.2.5 and highlights how the interface provided influences the topography of possible distributed actor systems.

Any objects created should be owned by the actor and although it is possible to pass embedded references around it is unsafe and breaks the actor abstraction. By default messages are passed by copy and a deep copy of the objects in the message is created. However, it is possible to pass messages by reference but it is up to the user to ensure safety. Additionally, manipulating threads within the actor description is potentially unsafe and breaks the actor semantics.

The Actor Executor is a post-processing compilation step done by the ActorFoundry compiler which creates an executor for each actor class. This is done so that all methods labeled with "@message" matched with a string name and array of *Objects*. Any message which has either a method name or *Object* parameters which do not match a known "@message" method results in an exception. This design allows actor methods to be *pausable* by the scheduler and also allows the defined actor methods to be executed indirectly through a message. The raw actor object is owned by the executor along with its state and is inspected using the Java reflection API to allow the above functionality.

The ActorImpl is a class which encapsulates the actor semantics and is the scheduler Task which is run endlessly. The main loop within ActorImpl checks for messages in its mailbox and then forwards the request to the executor. Other than the main loop, the ActorImpl handles the creation of actor requests such as *create*, *migrate*, *send* and passes those requests to the ActorManager. However, other than queuing new messages to the mailbox

of the actor, the thread which runs the ActorManager does not touch the state of individual ActorImpls.

Naturally the access of the mailbox and even ActorName is not thread-safe and needs to be handled as such. The queuing of messages is not thread-safe because both the ActorImpl and the ActorManager modify it. The required synchronization, an object lock in Java™, of the mailbox negatively impacts performance. The creation of an actor needs to be an atomic operation so there is another lock for the duration of actor creation and initialization.

4.2.4 ActorRequest

An ActorMsgRequest represents a message sent from one actor to another. Information contained within this class includes the name of the sender, the name of the receiver, the method to be executed, and the arguments to the method.

4.2.5 Distributed Functionality

The way multi-node actor systems are created in ActorFoundry places strict limitations on the design potential of actor programs. ActorFoundry must be run without an initial actor and in an "open" state in order to create an ActorFoundry instance which can be used as part of a distributed system. Without the "open" state flag the scheduler will close the ActorFoundry instance once no actors in the systems are live. In the following sections, a "master" ActorFoundry instance refers to an instance with an initial actor while a "slave" ActorFoundry instance refers to one which waits for an initial message.

If an ActorRequest specifies a non-local actor or a creation request refers to a different ActorManager, the request is put in a service queue and handled asynchronously. Whenever an ActorName is communicated to a non-local ActorManager the ActorName is added to the *receptionist* list. Similarly, when an actor is created as a result of a remote creation request it is added to the *receptionist* list.

The "master" ActorFoundry should create an initial actor and pass it a message containing the IP of another node in the network. Although it is

possible to hardcode the ip of other nodes in the system, this design does not scale. However a downside of passing the IP to the initial actor is that it naturally leads to a star network topology. Using hardcoded ips it is possible to create longer chains within the topology but it suffers from maintenance and scalability issues.

In order to communicate with a remote ActorManager, the *YP* service is passed an IP and if successful returns an ActorManagerName. Due to the implementation of the network layer in ActorFoundry, once a ActorManager is discovered the ActorManagerName needs to passed around. This design makes it very difficult to create a connected topology and once again lends itself naturally to the star topology. Finally, ActorFoundry is unable to handle churn in the system and assumes reachability of all discovered remote ActorManagers.

4.3 HDGC in ActorFoundry

The HDGC algorithm implementation required changes to most of the parts of ActorFoundry covered in Section 4.2. The implementation is used as a proof of concept in order to perform tests on parameters in the HDGC algorithm.

4.3.1 ActorManager

Previously the ActorManager only held the list of *receptionist* actors since delivery of messages could be done through *ActorName* in the *ActorRequest*. The garbage collection algorithm requires access to all the local actors as well as a way to access their state. To allow this a list of local actors was added and ActorNames along with their ActorImpl is added for every local actor creation. Actors can be part of both the *receptionist* list as well as the *local* list.

The garbage collection process is implemented as a service within the ActorManager and is created during initialization of the Manager and immediately scheduled by the scheduler. For the entire duration that the ActorFoundry instance is alive, the garbage collection service runs in an infinite loop. The garbage collection thread runs periodically every 5 seconds and

sleeps otherwise.

The garbage collection thread is able to send the various garbage collection messages using actors in the local list. Additionally the garbage collection functionality added to ActorImpl is called by this thread rather than the ActorImpl itself. However, this means that all accesses to a mailbox must be synchronized because multiple threads are able to modify the mailbox. For a global garbage collection phase the GC thread also sends out remote ActorMsgRequests in order to coordinate with known remote ActorManagers.

4.3.2 ActorImpl

Functionality involving managing acquaintances, inspecting pending messages, and handling of garbage collection messages was added to ActorImpl. Although the functionality is part of the ActorImpl, the ActorManager thread calls them as opposed to the ActorImpl Task. The previously explained synchronization along with many additional messages for inverse acquaintances is the main overhead from the HDGC algorithm implementation.

In order to discover the acquaintances of an actor the ActorImpl needs to inspect the state of the actor as well as pending messages and the current executed message if one exists. The ActorExecutor contains a reference to the actor definition whose fields are inspected using the Reflection API. This inspection finds and records any ActorName or collection of ActorNames. Using a flag to signal local vs global garbage collection, this inspection adds either all acquaintances or only local ones. During regular execution of messages a copy of the message being handled is saved and is inspected if a garbage collection occurs during execution. Similarly, any messages in the mailbox are also inspected. The inspection of messages involves saving the sender as an inverse acquaintance and also inspecting parameters for any ActorNames or collection of ActorNames.

Previously the ActorManager delivers messages by queuing the message in the mailbox of receiving ActorImpl. Changes were made so that if the message is a garbage collection message send on behalf of the garbage collection service, it will be handled immediately rather than enqueued. The types of messages in ActorFoundry was extended and are described below.

4.3.3 ActorRequest

The messages sent by actors through an ActorMsgRequest have an additional color as well as type. The color is whether the message is new or old using the scheme descripted in the HDGC algorithm. Messages are intercepted before delivery and actions are performed according to the HDGC algorithm. An important design decision for the handling of ActorMsgRequests is that the ActorManager thread handles the messages rather than the ActorImpl. This is done so that the garbage collection process does not halt the user computation and is performed in the background.

ActorMsgRequest used to be messages sent from one another to another but with the introduction of garbage collection a message can be of the following types:

- "Normal" message.
- Garbage collection message.
- Inverse garbage collection message.
- Back propagation (backprop) message.
- Finish message.
- Inverse acquaintance message.
- Stop message.

A normal message is a message sent from an actor to another actor and does not need to be handled any differently from before.

The garbage collection message and inverse garbage collection is handled as specified by the HDGC algorithm according to their color. Similarly, the backprop message is the backwards propagation message which is used as an "ack" in the HDGC algorithm.

The finish message is used by root actors to notify the ActorManager that the current scavenge phase is complete. Although it is possible to reuse the backprop message for this purpose in order to keep the design clean these two types of notification messages are kept separate.

The inverse acquaintance message is sent from one actor to all of its acquaintances during the garbage collection phase so that its acquaintances

can add it as an inverse acquaintance. This message type is the source of a large number of messages and for large systems will flood the network. A new mechanism to handle the inverse acquaintance relationship is necessary for the HDGC algorithm to scale.

The stop message is a message sent by the ActorManager to an actor to end the execution loop. This is done so that when an actor is determined to be garbage it ends execution and can be garbage collected by the JVM in the future. The message is not intercepted and is handled by the ActorImpl itself because an actor with no pending messages will sleep until a message arrives.

4.3.4 Local vs Global Garbage Collection

Due to how ActorFoundry handles multi-node systems, the coordination required in the HDGC implementation requires additional bookkeeping. The knowledge of other ActorFoundry nodes is only acquired when an actor invokes the YP service to first determine the reachability of another ActorManager. Upon receiving a reply, the remote ActorManagerName is stored.

The number of local garbage collections run for every global garbage collection is a variable in the HDGC algorithm which this thesis explores. In a global garbage collection, all actors are explored and all garbage actors are collected in that phase. In a local garbage collection, actors whose name have been communicated to a remote ActorManager are treated as if they are root actors. This is done because it is not possible to determine locally if an actor is garbage and cannot be safely collected.

The garbage collection service on the master ActorFoundry node handles the coordination for garbage collection. When garbage collection is run the GC thread first determines if the current GC phase is a local or a global garbage collection. In both cases the GC thread broadcasts a message to all N slave ActorFoundry nodes notifying them to begin a local or global GC phase. After the broadcast a snapshot phase happens in order to clear messages in transit. The snapshot phase consists of the master node sending snapshot messages to all slaves and then waiting for an ack from all slave nodes.

In the case of a local GC phase, each node handles their respective garbage

collection without additional communication. In the case of a global GC phase, all GC threads proceed with capturing the state of local actors, sending inverse acquaintance messages, and scavenge phase the same as in local garbage collection but then waits. After the master GC thread has received all acks for its root actors it broadcasts a message indicating the scavenge phase is finished then continues as usual. Upon receiving this message, each slave node continues with marking actors as garbage and stopping the Task.

4.3.5 Garbage Collector Flow Summary

The following list is a high level overview of the steps in the garbage collector implementation for the "master" ActorManager. The list highlights some of the differences from the original HDGC algorithm due to being implemented on ActorFoundry.

1. Determine if the garbage collection phase should be local or global.
2. If global, send to all remote ActorManagers a message that a GC phase is beginning.
3. Send a snapshot message to all remote ActorManagers and wait for every ActorManager to reply.
4. For each local actor, capture the current acquaintances. For a local phase, do not add non-local actors.
5. For each local actor, send to each acquaintance an inverse acquaintance message. Touch root actors, tenured actors, and potentially receptionists.
6. For each actor that is touched send each acquaintance a GC message and each inverse acquaintance a INVGC message.
7. Wait for a finish message from the initial touched actors from this phase.
8. If global, broadcast a message indicating the scavenge phase is complete.

9. Any actor that is untouched or suspended is labeled as garbage. Increase survivedGC counter for all survivors.

10. Send garbage actors a stop message, remove them from the scheduler, and remove all references to them.

11. Wait 5 seconds before looping again.

4.3.6 Optimizations

The implementation requires the state of the actor to be captured anew each garbage collection phase. This is an expensive process and is unnecessary in a system where the topology infrequently changes. A potential solution is to detect when an acquaintance is gained or lost and update the topology accordingly. Although this was not implemented completely in this thesis, an approximation was used to estimate the effect this optimization would have. Additional book-keeping in the local collect phase is needed to remove references to the garbage actor from all acquaintance and inverse acquaintance lists.

When an actor gains or loses an acquaintance it sends a message to the acquaintance which is intercepted by the ActorManager. Using this message the ActorManager removes the acquaintance reference and inverse acquaintance reference from the respective ActorImpls.

4.4 Listings

Listing 4.1: Hello World in ActorFoundry

```
public class HelloActor extends Actor {
  ActorName acq = null;

  @message
  public void boot() throws RemoteCodeException {
    call(stdout, "println", "Hello ");
    acq = create(WorldActor.class);
    send(acq, "hello", self());
  }
}

public class WorldActor extends Actor {
  ActorName acq = null;

  @message
  public void hello(ActorName name) throws RemoteCodeException {
    acq = name;
    send(stdout, "println", "World!");
  }
}
```

Listing 4.2: Actor Constructor & Overloading

```
public class FibActor extends Actor {
  ActorName parent = null;

  public FibActor(ActorName p) {
    parent = p;
  }

  @message
  public void boot(String remote, int val) throws
   RemoteCodeException{
    ActorName child1 = null;
    try {
      ActorManagerName remoteManager = (ActorManagerName)
        invokeService(YP.name, "ypLookupRemoteManager", remote);
      newChild1 = create(remoteManager, FibActor.class, self());
    } catch (Exception e) {
        ...
    }
    ...
  }

  @message
  public void boot(int val) {
    ...
  }

  ...
}
```

CHAPTER 5

RESULTS

This chapter covers the test setup and the parameters which are used in this parametric study. The results include local and distributed tests. Finally the relevance of these results are discussed along with implications for actor program design.

5.1 Test Setup

The test setup was structured to represent an "embarrassingly parallel" problem using an actor system. In general these problems involve a splitting of input with little if any communication between different branches of computation which all report intermediate back to a parent actor. The topology of these sort of systems is a tree structure with limited interconnects between workers.

This is the general setup for most of the tests run because it can be used to isolate the parameters and observe the effects on the runtime. A few tests were run which did not require this sort of topology. In this sort of topology the number of actors can be grown easily and reachability easily managed. Additionally, this sort of topology provides a lower-bound on the performance because many problems can be decomposed to this organization. This provides a way to easily estimate the impact of the GC implementation for many existing actor systems.

A simple example of an "embarrassingly parallel" problem would be a naive implementation of Fibonacci. This actor system would consist of a root actor which receives a integer N which is *Nth* Fibonacci number to calculate. This root actor then creates 2 children which are responsible for calculating *N-1* and *N-2*. This continues until a base threshold is reached and the result is sent to the parent. Each parent upon receiving results from both children

will propagate the sum back to its parent. This continues until the root actor receives the final result and computation completes.

The specific topology used in the tests for single node and the multi-node system are explained in their respective sections below. This topology caching optimization was tested separately while the default strategy is used for every other result.

5.2 Single Node

Tests for single node were done on a single laptop computer with background processes running. 1024MB memory was allocated for each test.

5.2.1 Effect on User Computation

The first test run was Fibonacci for N=29 with and without garbage collection. This test was to determine the impact of garbage collection on user computation. Although garbage collection is run concurrently and does not block the user computation it uses up a thread that could otherwise be used for computation. Additionally the increase in heap usage from the HDGC algorithm forces the JVM garbage collector to run more frequently which includes stop-the-world pauses. Garbage collection was run once during the computation of Fibonacci N=29.

Table 5.1: Single Node Fibonacci N=29

GC_TOTAL	NO_GC	TOTAL NO_GC	GC_THREAD
3708 ms	3330ms	378ms	424ms

The results in Table 5.1 are the average of three runs to account for any variability. The results show that despite not having a stop-the-world pause the garbage collector still imposes a cost on user computation. The GC thread required 424ms to finish garbage collection and added 378ms to the user computation. However, this is only a difference of 46ms which is negligible. Although not as expensive as a full stop-the-world pause which includes context switches, the garbage collection is still very costly.

Next, the same test was run for quicksort. The implementation of quicksort also uses a naive "divide-and-conquer" organization of actors. However the implementation differs by using not copying the array being sorted. This object is shared between the actors and although supported in ActorFoundry, does not follow the original Actor Model schematics.

Table 5.2: Single Node Quicksort 500000

GC_TOTAL	NO_GC	TOTAL NO_GC	GC_THREAD
5784 ms	4544ms	1240ms	1372ms

The results in Table 5.2 show the runtime of quicksort for an array of 500000 integers with and without GC. The increase in runtime due to garbage collection was 1240ms which is lower than the overall runtime of the GC thread. However the difference is negligible and shows that the user computation is slowed down by nearly the entire GC runtime.

5.2.2 Tenure Policy

The following tests feature an initial actor which creates 100 local children and each child creates another 100 children. The root actor then enters a loop for 15 iterations. The children of the root actor will be referred to as the 1st generation and the children of those actors will be referred to as the 2nd generation.

In each iteration each 1st generation child is sent a loop message. The 1st generation child then removes references to 15 2nd generation children and creates 15 children. This keeps the number of reachable actors the same but introduces churn into the system. Additionally, the root creates 15 "garbage cycles". A garbage cycle actor creates a child, stores it, and sends it a message with its own ActorName. The resulting actors is a garbage cycle of size 2. The root actor does not save references to the created garbage cycles.

Table 5.3: Single Node Tenure Result

time(ms)	TENURE_2	TENURE_4	TENURE_N+1
ITER0	310	337	319
ITER1	184	144	180
ITER2	112	149	117
ITER3	121	134	135
ITER4	149	131	112
ITER5	150	128	114
ITER6	129	138	116
ITER7	129	147	131
ITER8	125	165	140
ITER9	135	174	137
ITER10	130	152	135
ITER11	139	192	141
ITER12	115	146	144
ITER13	120	200	134
ITER14	124	161	133
TOTAL	2172	2498	2188

The results in Table 5.3 are how long a GC phase takes using three different tenure policies. The *TENURE_N+1* column represents the policy where tenure begins at 1 but increases by 1 for the next tenure threshold. Actors which are tenured act as if they are part of the root set which means there are fewer actors to explore in the scavenge phase. However, the cost of capturing the actor state along with looping over local actors in the implementation is still present. Having a lower tenure threshold is more efficient in this implementation because it keeps the number of actors in the system low.

5.2.3 Number of Actors

The previous test highlighted the fact that the number of actors in the system is an important variable. The following test uses the same setup as above but with varying numbers of children. Churn, tenure, and cycles remain fixed at 15, 4, and 15 respectively.

Table 5.4: Single Node Num Actor Result

time(ms)	100/100	150/150	200/200	300/300	400/400
ITER0	342	452	623	1359	2037
ITER1	145	284	452	1018	1749
ITER2	109	244	441	857	1707
ITER3	115	276	634	892	1862
ITER4	109	255	598	983	2053
ITER5	116	262	793	956	1835
ITER6	126	261	517	956	2001
ITER7	127	277	439	1145	1905
ITER8	130	259	414	1242	1986
TOTAL	1319	2570	4911	9408	17135
RATIO	7.658	8.814	8.186	9.598	9.361

The results in Table 5.4 show three tests run with varying number of children. This number means that the test for *100/100* has initially 10101 actors, *150/150* has 22651 actors, *200/200* has 40201 actors, *300/300* has 90301 actors, and *400/400* has 160401 actors. The ratio row is the total GC thread time divided by the number of initial actors. The ratios show that the relationship between number of actors and GC runtime is roughly linear. The Figure 5.1 shows clearly that the runtime of the implementation scales linearly with the number of actors.

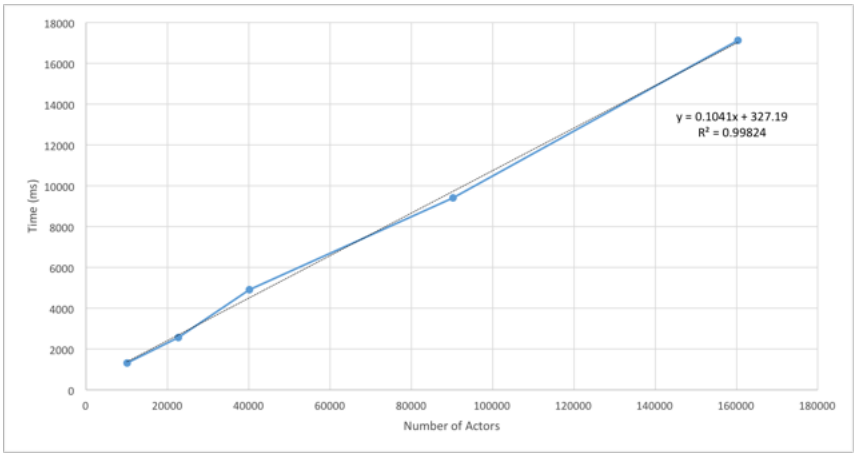

Figure 5.1: GC Runtime versus Number of Actors

A one-way analysis of variance test was done on Table 5.4. This test was

done on the 5 columns with each row being an observation. The results show an F-value of 473.8 and a P-value less than 0.01 which shows that the results are statistically significant.

5.2.4 Churn

The following test results show how churn in the system affects the GC runtime. The topology remains the same as in the previous tests. Other than churn the variables tenure, cycles, num-children are 4, 15, and 100 respectively.

Table 5.5: Single Node Churn Result

time(ms)	CHURN_15	CHURN_50	CHURN_100
ITER0	342	360	340
ITER1	145	178	226
ITER2	109	146	230
ITER3	115	150	189
ITER4	109	170	206
ITER5	116	163	230
ITER6	126	180	290
ITER7	127	173	207
ITER8	130	176	183
TOTAL	1319	1696	2101

The results in Table 5.5 show three tests with varying amounts of churn. As stated previously, churn is how many 2nd generation children are removed and also how many additional children are created. The results show that high churn increases the GC runtime because there are more actors to explore and also remove.

A one-way analysis of variance test was done on Table 5.5. This test was done on the 3 columns with each row being an observation. However the first row was excluded because the first iteration in this test is without any churn with the same setup. The results show an F-value of 40.2 and a P-value less than 0.01 which shows that the results are statistically significant.

5.2.5 Topologies

The following test results show how the number of acquaintances affects the GC runtime. In order to test this we created a scale free network where actors know an arbitrary number of other actors based on the Poisson distribution. In this type of topology it is difficult to incrementally create garbage because a few number of random links will cause every node to be reachable from the root. The number of connections a parameter which can severely limit the scalability of this implementation and unlike previous tests with a minimum of 10101 actors, this test has a very limited number of actors.

Table 5.6: Single Node Scale Free Poisson Distribution

time(ms)	300	500	1000
$\lambda = \frac{1}{10}N$	238	634.3	4183.3
$\lambda = \frac{1}{3}N$	449.3	1690	13592.3
$\lambda = \frac{1}{2}N$	650.1	2501.3	22295.7

The Poisson distribution was used to determine the number of acquaintances an actor should have. Once the root actor creates all N actors, it generates a number from the Poisson distribution for each of the N Actors. Each number is used to randomly select acquaintances for each actor. After acquaintances have been generated for all N actors, the root actor also generates a number and removes extra acquaintances. In this way the topology where each actor knows an arbitrary number of acquaintances following the Poisson distribution is created.

The results in Table 5.6 show tests with 300, 500, and 1000 actors. The λ value for the Poisson distribution is a fraction of the total number of actors. With a large number of connections, this implementation quickly becomes unusable due to the increased scavenge time. The Figure 5.2 shows the trend of increasing the number of connections in a system.

Next, a ring topology actor system was created. The ring topology is the same as a linked list of actors with the last actor in the link knowing the first actor. The number of acquaintances is equal to the number of actors in the ring.

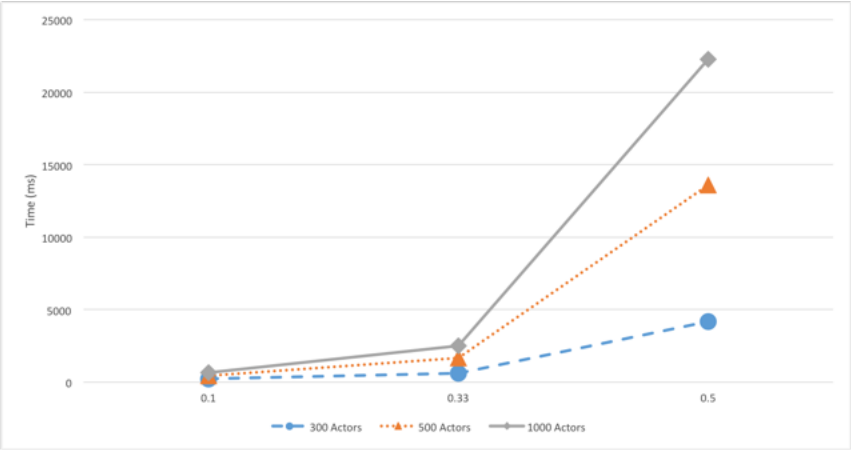

Figure 5.2: GC Runtime versus Poisson Lambda Value

Table 5.7: Single Node Ring Topology

time(ms)	N=10001	N=20001	N=40001	N=80001
ITER0	284	351	461	899

The results in Table 5.7 show GC runtimes for rings of size 10001, 20001, 40001, and 80001. Although the runtime does increase the topology also plays an important role. Unlike the results for a tree structure, because there are fewer backprop messages required in the scavenge phase the runtime scales. The ring topology is the lower bound for the least number of connections between actors in an actor system.

Finally, a 2D mesh topology actor system was created. The mesh is a NxN matrix of actors where each actor knows its immediate four neighbors. The root actor creates all the actors and sends each actor the ActorNames of its adjacent neighbors. Finally the root actor stores a reference to the first actor created.

Table 5.8: Single Node 2D Mesh Topology

time(ms)	60x60	80x80	100x100
ITER0	2518	8177	24907

The results in Table 5.8 are for meshes of 3601, 6401, and 10001 actors. Compared to the ring topology the mesh topology has many more connections. The label phase scales linearly with the number of actors while the scavenge phase scales exponentially. The results for tree, scale-free, ring, and

mesh show that HDGC implementation performs best for sparsely connected topology.

5.2.6 Topology Caching

The following test results use the topology cache optimization. The main benefit of this strategy is not having to use reflection to reinspect the state of each actor for every GC phase. The setup uses the same parameters as the tests from Section 5.2.4. However, because the optimization is faked using messages the performance measurement is an estimate and an actual implementation could have fairly different results.

Table 5.9: Single Node Topology Caching Result

time(ms)	CHURN_50	CHURN_100	CHURN_200	CHURN_400
ITER0	50	60	62	65
ITER1	43	59	103	122
ITER2	23	27	75	76
ITER3	18	27	58	70
ITER4	12	24	46	65
ITER5	36	46	62	107
ITER6	21	61	52	95
ITER7	25	37	68	66
ITER8	30	36	66	81
TOTAL	258	377	592	747

The results in Table 5.9 show GC runtimes with varying amounts of churn in the system. The tests with churn higher than 100 are where acquaintances are needlessly gained and lost. Even in the case where churn is higher than the number of actors the runtimes are very low. These results show that the cost of using reflection to inspect the state of an object is extremely expensive compared to sending messages.

5.3 Multi-Node

Tests were done using two computers connected via LAN but have different hardware and OS. The transport protocol used in ActorFoundry was UDP as that was the most up to date. The UDP implementation in ActorFoundry has guaranteed delivery of messages. Both computers had 1024MB allocated for all tests.

5.3.1 Local to Global GC Ratio

The following test varies the number of local garbage collection phases run for each global garbage collection phase. The local topology is the same as before but additional children and cycles are created remotely. In addition to 100 local children, the root actor creates 30 remote children that each create 100 2nd generation children. In each iteration 15 remote cycles are created as well. A remote cycle is a cycle that spans multiple nodes. These cycles are unable to be collected until the next global garbage collection phase.

Table 5.10: Two Node Local-Global Ratio Test

time(ms)	GLOB3	GLOB4	GLOB5
ITER0	1469	1491	1496
ITER1	174	138	158
ITER2	105	133	136
ITER3	104	120	130
ITER4	1042	119	167
ITER5	119	1213	170
ITER6	127	124	1312
ITER7	138	128	186
ITER8	1013	158	149
ITER9	167	132	145
ITER10	105	1136	175
AVG_LOC	129.8	131.5	157.3
AVG_GLO	1174.7	1280	1404

The results in Table 5.10 show that the implementation of HDGC in ActorFoundry performs poorly when references cross node boundaries. Many

non-local messages are sent during a global garbage collection for both the snapshot and scavenge phases. The implementation sends many inverse acquaintance messages during the snapshot phase and GC/INVGC messages during the scavenge phase. Messages to non-local actors are extremely expensive. As the local-global ratio increases, more remote cycles are created and need to be collected which is reflected in the increasing average global GC runtime. The local GC runtime increases as well but is similar to the values from a single node which is expected.

5.3.2 Two Node Churn

The following test features the same setup as the previous one except for the churn parameter and includes a new remote churn. In this test remote children are discarded every global GC phase and no additional remote children are created. 15/1 represents a system with 15 local churn and 1 remote churn. Similarly, 15/2 represents 15 local churn and 2 remote churn.

Table 5.11: Two Node Churn Test

time(ms)	15/1	15/2
ITER0	1468	1659
ITER1	184	245
ITER2	130	173
ITER3	153	149
ITER4	149	219
ITER5	910	841
ITER6	156	163
ITER7	124	115
ITER8	113	145
ITER9	111	159
ITER10	1011	1029
AVG_LOC	140	171
AVG_GLO	1129.6	1176.3

The results in Table 5.11 show that although fewer remote references should improve efficiency there are many other effects which influence the global GC runtime. In iteration 0 the two system are exactly the same yet

there is a percentage difference of 12.2%. However the effect of the reduction in cross-node references is apparent in iteration 5. By removing remote references there are fewer cross-node inverse acquaintance messages as well as scavenge messages so the global GC is faster than the above test setup.

5.3.3 Ring and Mesh Topology

Similar to the single node topology test, these tests create a two node ring topology and a two node mesh topology. The ring topology has half of the ring on one node and the other half on the other. The ring is in one direction so only one actor on a node knows one non-local node. The mesh topology has half of the mesh on one node and the other half on the other. This means that there are N non-local acquaintances for each half of the mesh.

Table 5.12: Two Node Ring Topology

time(ms)	N=2501	N=5001	N=10001
1-Node	172	212	268
2-Node	210	272	398

Table 5.13: Two Node Mesh Topology

time(ms)	40x40	50x50	60x60
1-Node	564	1175	2362
2-Node	835	1950	4622

The results in Table 5.12 and Table 5.13 show the GC runtime for a global GC phase. In the ring topology regardless of the size of the ring the number of non-local acquaintances remains the same. The results show that the cost of this non-local acquaintance is roughly static regardless of ring size. In the mesh topology the number of non-local acquaintances increases linearly with the size of the mesh. However as the mesh size increases the GC runtime increases dramatically due to the increase of non-local acquaintances. These results indicate that in order for the HDGC implementation to scale it is necessary to create an actor system with as few non-local connection as possible.

CHAPTER 6

CONCLUSION

This thesis features an implementation of the Hierarchical Distributed Garbage Collection Algorithm on ActorFoundry. This implementation is then used to perform a parametric test on parameters in the HDGC algorithm. Parameters explored include the topology, churn, and tenure policy. The HDGC algorithm is a garbage collector which does not require stop-the-world pauses and utilizes the heuristic that young objects die early through a generational mark-and-sweep strategy.

The design of the ActorFoundry framework influenced the implementation greatly. In order to maintain an up to date list of acquaintances and inverse acquaintances, the state of each actor must be examined which is a major bottleneck in the implementation. The three main costs are determining acquaintance and inverse acquaintance relationships, number of connections in the system, and cross-node messages.

A potential optimization to the HDGC algorithm implementation by caching the topology was proposed and mock implementation was tested. The results for the mock implementation show that this could allow the implementation to scale with number of actors effectively. This could potentially mitigates the cost of determining acquaintance and inverse acquaintance relationships. However, it is possible that this optimization could still reduce the GC performance depending on how frequently changes actor in the actor topology. The performance can be reduced if the cost of handling all the updates to the cached topology is higher than the cost of determining the topology each time the GC is run.

The number of acquaintances in a system is also a parameter which heavily impacts the performance of the HDGC implementation. With a high degree of interconnectivity the scavenge phase of the algorithm completely dominates all other costs. A fully connected system of actors renders the implementation completely unusable due to the non-linear scaling. The topology

of the actor system also plays a role in determining the cost of the scavenge phase. The HDGC implementation handles the scavenge phase in a depth-first way so topologies which have long chains results in a large call stack.

The parametric study results indicate that the ideal topology of an actor system in ActorFoundry is to maintain as few remote references as possible in a specific topology. This result shows that "divide-and-conquer" topologies are the most efficient by reducing cross-node communication. Another effect of this topology is that there are few connections required overall because communication is only through parent-child. Finally, the results show that it is better to have a reusable pool of actor because the garbage collection runtime scales with the number of actors.

6.1 Future Work

Below is a list of potential areas for future work:

- Grouping cross-node inverse acquaintance messages can drastically reduce the amount of messages which need to be sent.
- Due to the synchronization required to inspect the state of an actor it may be more efficient have actors handle the state themselves. This means that user computation is halted but due to synchronization it may still be more efficient.
- The generational strategy could use profiling or pre-tenuring to reduce the amount of scavenge messages sent.
- Different local garbage collectors can be be used in conjunction with the HDGC global collector.
- The cost of inverse acquaintance messages is a major bottleneck for ActorFoundry. Implementing HDGC in a language with a different representation of acquaintances and inverse acquaintances could drastically improve the performance.
- A full implementation of the topology caching optimization would allow efficient scaling with number of total actors.

REFERENCES

[1] G. A. Agha, *ACTORS - a model of concurrent computation in distributed systems*, ser. MIT Press series in artificial intelligence. MIT Press, 1990.

[2] R. Greg Lavender and Douglas C. Schmidt, "Active object – an object behavioral pattern for concurrent programming," 1995.

[3] G. A. Nalini Venkatasubramanian and C. Talcott, "Hierarchical garbage collection in scalable distributed systems," Dept. of Computer Science, University of Illinois at Urbana-Champaign, Technical Report UIUCDCS-R-92-1740, April 1992.

[4] N. Venkatasubramanian, "Hierarchical garbage collection in scalable distributed systems," in *University of Illinois at Urbana-Champaign*, 1992.

[5] "The actor foundry," 1998. [Online]. Available: http://osl.cs.illinois.edu/software/actor-foundry

[6] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973. [Online]. Available: http://dl.acm.org/citation.cfm?id=1624775.1624804 pp. 235–245.

[7] R. K. Karmani and G. Agha, "Actors," in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed., 2011, pp. 1–11.

[8] D. Kafura, D. Washabaugh, and J. Nelson, "Garbage collection of actors," *SIGPLAN Not.*, vol. 25, no. 10, pp. 126–134, September 1990. [Online]. Available: http://doi.acm.org/10.1145/97946.97961

[9] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Commun. ACM*, vol. 3, no. 4, pp. 184–195, Apr. 1960. [Online]. Available: http://doi.acm.org/10.1145/367177.367199

[10] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. iUniverse, 2000.

[11] H. Lieberman and C. Hewitt, "A real time garbage collector that can recover temporary storage quickly." DTIC Document, Tech. Rep., 1980.

[12] W. D. Clinger and L. T. Hansen, "Generational garbage collection and the radioactive decay model," in *ACM SIGPLAN Notices*, vol. 32, no. 5. ACM, 1997, pp. 97–108.

[13] H. G. Baker, "Infant mortality and generational garbage collection," *ACM Sigplan Notices*, vol. 28, no. 4, pp. 55–57, 1993.

[14] T. L. Harris, "Dynamic adaptive pre-tenuring," in *ACM SIGPLAN Notices*, vol. 36, no. 1. ACM, 2000, pp. 127–136.

[15] "Memory management in the java hotspot^{TM} virtual machine," White Paper, Sun Microsystems, April 2006.

[16] S. M. Blackburn, J. Cavazos, S. Singhai, A. Khan, K. S. McKinley, J. E. B. Moss, and S. Smolensky, "Profile-driven pretenuring for java," in *OOPSLA00 Companion, 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA00), Minneapolis, MN, USA*, 2000, pp. 129–139.

[17] "Akka documentation," May 2015. [Online]. Available: http://doc.akka.io/docs/akka/2.3.11/

[18] A. Joe, "Programming erlang: Software for a concurrent world," *Pragmatic Bookshelf*, 2007.

[19] J. Armstrong and R. Virding, "One pass real-time generational mark-sweep garbage collection," in *Memory Management*. Springer, 1995, pp. 313–322.

[20] S. Clebsch, S. Blessing, J. Franco, and S. Drossopoulou, "Ownership and reference counting based garbage collection in the actor world."

[21] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, "Orleans: distributed virtual actors for programmability and scalability," MSR Technical Report (MSR-TR-2014-41, 24). http://aka. ms/Ykyqft, Tech. Rep.

[22] J. E. Nelson, "Automatic, incremental, on-the-fly garbage collection of actors," 1989.

[23] A. Vardhan, "Distributed garbage collection of active objects: A transformation and it's applications to java programming," M.S. thesis, University of Illinois at Urbana-Champaign, Illinois, 1998.

[24] S. Clebsch and S. Drossopoulou, "Fully concurrent garbage collection of actors on many-core machines," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 553–570, 2013.

[25] S. Srinivasan, "A thread of ones own," in *Workshop on New Horizons in Compilers*, vol. 4, 2006.

[26] S. Srinivasan and A. Mycroft, "Kilim: Isolation-typed actors for java," in *ECOOP 2008–Object-Oriented Programming*. Springer, 2008, pp. 104–128.

[27] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the jvm platform: a comparative analysis," in *PPPJ*, B. Stephenson and C. W. Probst, Eds., 2009, pp. 11–20.