©Copyright by
Nalini Venkatasubramanian
1992

Hierarchical Garbage Collection in Scalable Distributed Systems

BY

NALINI VENKATASUBRAMANIAN

B.E., Bangalore University, 1989

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, 1992

Urbana, Illinois

HIERARCHICAL GARBAGE COLLECTION IN SCALABLE DISTRIBUTED SYSTEMS

Nalini Venkatasubramanian
Department of Computer Science
University of Illinois at Urbana-Champaign, 1992
Gul A. Agha, Advisor

Scalability, in the context of parallel computation, implies that given a program with sufficient parallelism, it will be possible to increase the performance of the system by adding physical resources. First, in order to discuss issues in scalable concurrent computing, we will develop a generalized model of parallel computation. Our model presents a hierarchical view of concurrency based on the *Actor* Model implemented on a highly concurrent machine. Second, in order to achieve high performance computing, new techniques must be developed for managing the computational resources of large numbers of concurrent processors. In this thesis, we deal with efficient memory management in scalable, parallel machines.

Automatic storage management is important in highly parallel programming environments because programmer controlled reclamation is highly prone to errors. Many of the deficiencies of existing storage reclamation schemes arise from the fact that they halt the computation process, preventing real-time response by the system. Our goal is to guarantee non-disruptive garbage collection (GC) which would permit the user process to proceed simultaneously with GC.

We propose a novel technique to handle the GC problem in multicomputers. The Hierarchical Distributed Garbage Collection scheme is an extension of Ungar's Generation Scavenging algorithm to multicomputers. The crux of the algorithm lies in obtaining a consistent state of the machine without disrupting ongoing computation. Significant features of the storage management scheme proposed include its applicability to a broad range of machines, compatibility with fault tolerance mechanisms, the ability to ensure a safe collection in the presence of a dynamically changing topology and real time response.

To my father

To live in hearts we leave behind is not to die

ACKNOWLEDGEMENTS

It has been a pleasure to work with Prof. Gul Agha, my thesis advisor. Gul has shown me what it means to be a kind person and enthusiastic scientist at the same time. I would like to express my deep respect for his knowledge and I appreciate his willingness to help me anytime I needed it. From numerous discussions on garbage collection to our weekly group lunches, I enjoyed it all.

I am grateful to Carolyn Talcott at Stanford University who was instrumental in the formalization of the abstract machine and correctness proofs. The formal model has been privileged to receive her thorough scrutiny. I wish to thank Prof. Sam Kamin for a number of discussions, especially the corridor ones. I would like to express my gratitude to Prof. Andrew Chien and Prof. L.V. Kale for their encouragement, advice and all the informative sessions. The members of of the *Open Systems Lab*, my officemates, Svend Frolund, Chris Houck, Wooyoung Kim, Shakuntala Miriyala, Rajendra Panwar, Anna Patterson and Dan Sturman have provided useful comments, and criticisms during the course of this work. I am happy to be a part of the group. Wooyoung, Chris and Dan, thanks for proofreading the thesis. Mike Lake, Rocco Martin and Rob Knauerhause, our manifolded garbage collection ventures were very useful to me. Thank you. In addition I wish to thank Manish Gupta, Nayeem Islam, Subrata Mitra and Brian Totty for their support and friendship. A big hug to Bonnie Howard for all her help, concern and pink balloons. Raju and Subrata, thanks for depositing my thesis and for encouraging me to be a chocolate nut.

My stay in Champaign-Urbana has been very memorable. The members of our Sunday night study circle, the children of Matthew House, have all in their own way helped me realize how fortunate I am and what joy there can be in sharing and giving. John, Lisa, Tamara and all my friends - Thank you.

I wish to acknowledge my collegues and managers at Hewlett-Packard for their cooperation.

Mummy, your urge for excellence has been a motivating force to me; thank you for convincing me that nothing is impossible, for helping me overcome my reluctance to try, for demonstrating that to me in your own ventures; and, above all, for giving up everything to be there for me.

Daddy, .., I don't know what to say. Your confidence in me is my most powerful shield.

Finally, I owe what I am to that ultimate power that has infused in me the desire to know *more*, to feel *more* and the strength to achieve *more*. I would feel lost without *It*.

TABLE OF CONTENTS

Chapter

Part 1: Resource Management in Scalable Systems 1

1 Introduction 2
1.1 Resource Management in Scalable Parallel Systems 2
1.2 Thesis Outline 5

2 A Survey of Memory Management Schemes 6
2.1 Traditional Memory Management Schemes for Sequential Machines 7
2.1.1 Mark and Sweep Algorithms 8
2.1.2 The Stop and Copy Scheme 8
2.1.3 Reference Counting Techniques 9
2.1.4 Baker's Copying Garbage Collector 11
2.1.5 Generational Garbage Collection 12
2.1.6 Ungar's Generation Scavenging 13
2.1.7 Moon's Ephemeral Garbage Collection 15
2.2 Multiprocessor Garbage Collection 16
2.2.1 Storage Management in Shared Memory Multiprocessors 17
2.2.2 Storage Management in Distributed Memory Multiprocessors 21
2.3 Improvisations 35
2.3.1 Virtual Memory Issues 35
2.3.2 Interactive and Real-Time Garbage Collection 36
2.3.3 Conservative Pointer Finding 39
2.3.4 Fault Tolerant Garbage Collection 40

3 Generalized Parallel Models 42
3.1 Universal Models Of Concurrency 42
3.2 The Actor Model 43
3.3 The Bulk Synchronous Parallel (BSP) Model 47
3.4 The Parallel Machine Interface 49
3.5 A Hierarchical View of Parallel Computation 51

4 Issues in Parallel and Distributed Machines 55
4.1 Storage Management Issues in Parallel Machines 56
4.1.1 A Shared Distributed Namespace 57
4.1.2 Locality 58

4.1.3 Load Balancing . . . 60
4.1.4 Network Activity . . . 61
4.1.5 Exploitation of Concurrency . . . 62
4.2 Goals of Distributed Memory Management Revisited . . . 62
4.3 Annotations for Storage Management . . . 64

Part 2: Hierarchical Distributed Memory Management . . . 67

5 Object Creation and Naming . . . 68
5.1 Object Creation . . . 68
5.2 Naming in Distributed Systems . . . 69
5.3 The Tiered Naming Scheme for Distributed Machines . . . 72
5.3.1 Organization of Global Memory . . . 75
5.4 Persistent Objects . . . 79
5.4.1 Implementing Persistence . . . 82
5.4.2 A Generational Approach to Persistence . . . 83

6 Distributed Generation Scavenging . . . 85
6.1 The Problem of Asynchrony in Distributed Garbage Collection . . . 85
6.2 Generational Collection of Actors . . . 89
6.3 The Distributed Generation Scavenging (DGS) Algorithm . . . 91
6.3.1 Assumptions of the Algorithm . . . 95
6.3.2 Phase 1: The Distributed Scavenge Phase . . . 96
6.3.3 Phase 2: The Local Clear Phase . . . 98
6.3.4 Post-GC Work . . . 101
6.4 Migration and Compaction . . . 101

7 Network Clearance in an Asynchronous System . . . 104
7.1 A Consistent State . . . 104
7.2 Assumptions of the Network Clearance Algorithm . . . 107
7.3 Definitions and Terminology . . . 108
7.4 Tagged Messages in the System . . . 109
7.5 Network Clearance Algorithms . . . 110
7.5.1 Bulldoze Messages . . . 111

8 Hierarchical Distributed Garbage Collection . . . 114
8.0.2 Specification of the Hierarchy . . . 115
8.1 Hierarchical Distributed Garbage Collection . . . 119
8.1.1 Outline of The HDGC Algorithm . . . 119
8.1.2 Assumptions of the HDGC Algorithm . . . 121
8.1.3 The HDGC Algorithm . . . 121
8.2 Extensions and Optimizations to the HDGC Algorithm . . . 126
8.2.1 Multiple Levels in the Hierarchy . . . 126
8.2.2 Fault Tolerance in the HDGC Algorithm . . . 128
8.2.3 Abstracting Local Garbage Collection . . . 128
8.3 Discussion . . . 128

Part 3: A Formal Model . . . 130

9 A Theoretical Framework for Memory Management . . . 131
9.1 Model of a Multicomputer System . . . 132
9.1.1 The Computation Model . . . 132
9.1.2 The Memory Model . . . 132
9.1.3 The Communication Model . . . 132
9.1.4 Timing Considerations . . . 133
9.1.5 System Considerations . . . 133
9.2 Terminology and Notation . . . 134
9.3 Formal Configuration Definition of the $\mathcal{AAM}$ . . . 135
9.3.1 Multicomputer System . . . 135
9.3.2 Meta-Actor System . . . 136
9.3.3 Base-Actor System . . . 138
9.3.4 Channel System . . . 140
9.4 Initial Configuration of the $\mathcal{AAM}$ . . . 141
9.4.1 Multicomputer System . . . 141
9.4.2 Meta-Actor System . . . 141
9.4.3 Base-Actor System . . . 141
9.4.4 Channel System . . . 142

10 Correctness Proof of the Hierarchical Distributed Garbage Collection Scheme . . . 143
10.1 Correctness Proofs of Distributed Algorithms . . . 143
10.1.1 Safety or Invariance Properties . . . 143
10.1.2 Liveness Properties . . . 144
10.2 Summary of the Generation Based Distributed Garbage Collection Scheme . . . 145
10.3 Postconditions to be Verified . . . 147
10.3.1 GC Step 1: Pre-GC Sweeps . . . 147
10.3.2 GC Step 2: DistributedScavenge Phase . . . 148
10.3.3 GC Step 3: Interphase Synchronization . . . 150
10.3.4 GC Step 4: LocalClear Phase . . . 151
10.3.5 GC Step 5: Post GC Sweeps . . . 152
10.3.6 Proof Strategy . . . 152
10.4 Lemmas used in the Correctness Proof . . . 153
10.5 Correctness Proof of the Hierarchical Distributed GC Scheme . . . 154
10.5.1 Invariant Properties of the Transition System . . . 154
10.5.2 Safety Properties . . . 155
10.5.3 Liveness Properties . . . 158
10.6 Correctness Proof of the Individual Lemmas in the Hierarchical Distributed Garbage Collection Scheme . . . 162
10.6.1 Initial Conditions . . . 162
10.6.2 GC Step 1: Pre-GC Sweeps: . . . 162
10.6.3 GC Step 2: DistributedScavenge Phase . . . 167
10.6.4 GC Step 3: Interphase Synchronization . . . 171
10.6.5 GC Step 4: LocalClear Phase . . . 172
10.6.6 GC Step 5: Post GC Sweeps . . . 173

11 Conclusions and Future Research Directions 175
11.1 Thesis Conclusions 175
11.2 Optimizations and Further Work 177

BIBLIOGRAPHY 180
APPENDIX

A Glossary of Terms and Definitions 188

B Pseudocode for the HDMM Scheme 203
B.1 General Object Functions 203
B.1.1 Variables and Data Structures 203
B.1.2 Actor Manipulation Functions 204
B.2 Base-Actor Specific Functions 207
B.3 Distributed Generation Scavenging 209
B.3.1 The DistributedScavenge Phase 209
B.3.2 The LocalClear Phase 210
B.4 Network Clearance Algorithms 211
B.4.1 Node Functions 211
B.4.2 Message Functions 214
B.4.3 The Broadcast Algorithm 214
B.4.4 The Forward Bulldoze Algorithm 215
B.4.5 The Backward Bulldoze Algorithm 215
B.5 Hierarchical Distributed Garbage Collection 216
B.5.1 Cluster Management Functions 216

LIST OF TABLES

3.1 Universal Models of Parallel Computation 51
3.2 Universal Models of Parallel Computation continued 52

5.1 Comparison of Virtual Memory and Object Based Memory Systems 72
5.2 Tag Bits Associated with the Regions in the Workable Heapspace 79

6.1 Moving Objects in Generations 93

8.1 Dedicated vs. Distributed Cluster Manager 117
8.2 Actions Taken by Objects Receiving Tagged Messages 125

LIST OF FIGURES

2.1 The Semi-space Garbage Collection Scheme 10
2.2 Ungar's Generation Scavenging Scheme 14
2.3 Remembered Set Element -Solution 1 25
2.4 Remembered Set Element -Solution 2 26
2.5 Garbage Collection of Actors 33

3.1 An Abstract Representation of Actor Transitions 44
3.2 Actor Event Diagrams 46
3.3 A Hierarchical View of Parallel Systems 53

5.1 Structure of the Global Nametable 74
5.2 Formats of the Global Nametable Fields 76
5.3 Logical Organization of Global Memory 77
5.4 Logical Organization of Scavenge Space 78
5.5 Persistence as a Class/Subclass Hierarchy 81

6.1 Handling Remote GC Messages – Scheme 1 86
6.2 Handling Remote GC Messages – Scheme 2a 87
6.3 Handling Remote GC Messages – Scheme 2b 88
6.4 Handling Remote GC Messages – Scheme 3 89
6.5 Global Garbage Collections in Progress 94
6.6 ScavengeSpace Transitions during a Global Garbage Collection 99

7.1 The Broadcast Wavefront 109
7.2 The Forward Bulldoze Wavefront 110
7.3 Object Receives the Forward Bulldoze Message 111
7.4 The Backward Bulldoze Wavefront 112
7.5 Object Receives the Backward Bulldoze Message 113

8.1 The Bulldoze Wavefront in Progress 124

Part 1: Resource Management in Scalable Systems

In this part of the thesis, we identify what we believe are some critical problems that must be addressed if scalable, highly parallel computing is to become a reality. A number of existing automatic storage management (reclamation) schemes for sequential and parallel machines have been reviewed. Most of the garbage collection schemes for multiprocessor systems require a global halt during GC and are not directly scalable. In order to discuss resource management in general and memory management in particular, we have argued the need for a generalized model of concurrency independent of programming or architectural details. The hierarchical view of a universal model of concurrency to be discussed in Chapter 3, is what we will be using in the rest of the thesis to develop the hierarchical distributed memory management scheme. Finally, we establish the goals and non-goals of scalable memory management after considering the various factors that will influence the performance of a scalable, parallel system.

Chapter 1

Introduction

In this chapter, we will introduce scalable, parallel systems and discuss some complexities that arise in resource management for such systems. We also state what we believe are necessary facilities for effective resource management. An outline of the rest of the thesis is presented in the end.

1.1 Resource Management in Scalable Parallel Systems

The goals of new-generation computing are two-fold: high performance architectures and powerful, user friendly programming environments. It has been clear for some time that parallel computation is going to provide a means for this breakthrough. Scalability, in the context of parallel computation, implies that given a program with sufficient parallelism, it will be possible to increase the performance of the system by adding physical resources. In a scalable system, no alteration of the application program is necessary to exploit the benefit of the added resources. Scalability is a fundamental requirement of *open systems*, i.e. of information systems which permit continuous influx of input from new and different sources [AH87].

Resource management involves techniques and mechanisms used to efficiently allocate, utilize and coordinate the resources of a system. New complexities arise when we need to perform resource management in parallel systems. Some of them are listed below.

- The communication network is one of the most critical of execution resources. In fact, the bottleneck in scaling concurrent computers is not limitations in the computational power

of the individual processors, but the costs and delays associated with transferring information from one processor to another [AWY89]. Thus, resource management strategies must try to reduce the communication traffic.

- Another source of complexity in parallel systems is that different resources may be needed simultaneously and these resources must all be available. Furthermore, in a large scale concurrent system, dynamic allocation of resources is necessary because computations may need to be automatically divided into large numbers of sub-computations. In actor systems [Agh86], such division happens each time concurrent sub-requests are made as a result of processing a given request [AH87]. With every fork in the computation, execution resources must be provided to the sub-computations created. Often the behavior of sub-computations cannot be determined in advance. Different strategies for subdividing resources lead to different results. An improper allocation may lead to a condition where a useful subcomputation cannot proceed due to the non-availability of resources, called the *dangling subcomputation problem.*

- Dynamically spawning off tasks may give rise to expanding resource requirements limited only by their physical availability. By serializing executions, resources can be exclusively allocated to the executing process and there are no resource contentions. However, little parallelism is exploited by this scheduling strategy. The other extreme aims at extracting all the parallelism inherent in the application by using a suitable programming paradigm. In such cases, studies have determined that many programs are "embarrasingly parallel"[Sar86]. Excessive parallelism leads to inordinate resource utilization and may have an even more serious outcome – deadlock due to unavailability of resources for any of the parallel tasks to continue execution. There is an obvious crossover point between the degree of parallelism exploited and effective resource utilization in a concurrent system. The goal, then, is to be able to set up a flexible *throttle* which will allow us to increase or reduce the degree of parallelism at will [Sar86, AC86].

- Another aspect of resource management is controlling the use of resources at the application level. Some problems exhibit an exponential growth in the number of possible paths which could lead to a solution; it is infeasible to explore all of them. This in turn means that differing amounts of resources must be allocated to different paths. The assessment

of how fruitful a particular path is changes over time as one assesses the intermediate results along the path [Agh85]. This is one reason resources need to be controlled and re-allocated dynamically.

- Other complications in a real implementation arise from the presence of prioritized execution of processes, conflict resolution, deadlock handling and synchronization.

Effective resource management in parallel machines is a combination of:

Static Analysis: Here, the text of the program is analyzed before execution at compile-time to detect information such as useless concurrency via dependence analysis, rough estimates of resource requirements, data placement and partitioning.

Heuristics: It is often sufficient to approach the resource management problem conservatively. In fact, mechanisms implemented to guarantee completeness or totality may degrade machine performance, thereby defeating the purpose of an efficient resource management strategy. Heuristics designed to achieve effective resource management concentrate on detecting sections of program execution critical to performance and target their restructuring efforts at those sections where the payoffs would be significantly large.

Reconfiguration strategies: The system must be capable of dynamically detecting "congestion points" and handling them, when appropriate by alteration of the existing system configuration.

Resource Management issues can be divided into process management, memory management and I/O (input-output) management. In this thesis, we will deal with memory management issues in scalable, parallel systems. However, in order to discuss resource management issues without referring to any particular programming language or architecture, we must isolate features specific to parallelism from issues specific to the framework in which parallelism is being exploited . We will develop a universal model of concurrency which uses Actors, a concurrent object oriented programming (COOP) environment [Agh90]. Memory management in a COOP system involves allocation and distribution of memory objects and garbage collection of these objects when they are no longer needed and we describe a method to effectively achieve this in scalable, concurrent systems.

1.2 Thesis Outline

This thesis is divided into three parts. Part 1 (Chapters 2-4) will develop some background and framework to deal with the problem for efficient memory management for scalable, parallel systems. Chapter 2 is a survey of existing methodologies for memory management in parallel and distributed systems. In Chapter 3, we review programming models for concurrent computing. Specifically, we describe Actors, a model of scalable concurrent computation. We also evaluate three generalized models of concurrency and propose a new universal model of concurrency that presents a hierarchical view of parallel computation. A number of problems, such as naming, load balancing and locality arise in concurrent systems. Chapter 4 addresses these problems, solutions and tradeoffs.

In Part 2, we introduce and describe the hierarchical distributed memory management (HDMM) scheme. Object naming is an integral part of the HDMM scheme and we elaborate on this in Chapter 5. Chapter 6 introduces the *Distributed Generation Scavenging* (DGS) algorithm, an extension of Ungar's Generation Scavenging to distributed systems. In order to ensure the correctness of the DGS algorithm in a distributed system, it is necessary that all the individual processing units perform collection with respect to a consistent view of memory. In Chapter 7 we motivate the need for a consistent global state and introduce the network clearance algorithms. Chapter 8 integrates the Distributed Generation Scavenging and network clearance algorithms to give us the *Hierarchical Distributed Garbage Collection* algorithm. We will show how the HDGC algorithm is applied when multiple levels are present in the hierarchy and extensions to achieve fault-tolerance and real-time behavior.

Part 3 deals with more theoretical aspects of the HDMM scheme. Chapter 9 develops the Abstract Actor Machine $\mathcal{AAM}$, a theoretical model of an actor implementation on a multicomputer system. We present an semi-formal proof of correctness to the HDGC algorithm in Chapter 10 using the $\mathcal{AAM}$. The final chapter discusses some optimizations to our algorithm and future research directions in this area.

Appendix A is a glossary of terms and definitions. Appendix B gives a pseudo-code implementation of different modules in the hierarchical memory management scheme.

Happy reading!

Chapter 2

A Survey of Memory Management Schemes

We state the requirements of a memory management scheme for a general purpose computer system in this chapter followed by a review of existing techniques for memory management, specifically, garbage collection (GC) in uniprocessor machines. We also discussed issues involved in shared memory and distributed memory multiprocessor GC and surveys many proposed algorithms and implementations. Various improvisations of the traditional algorithms have been developed to deal with specific application requirements like real-time response, virtual memory design implications and fault tolerance. Section 2.5 contains a discussion of these improvisations.

Storage Management in a programming environment refers to the control exercised over the use of memory that holds representations of objects/ information. It is responsible for allocating space for the object when it is created and subsequently reusing this allocated space when it is no longer needed. An efficient storage management scheme plays a crucial role in enhancing the efficiency of a programming system. Memory which is not accessible is referred to as *garbage* and the reclamation of garbage is *garbage collection.* Automatic Storage Reclamation is a must in efficient, user friendly programming environments. The runtime system must be capable of recognizing memory shortages and reclaiming unused memory for reallocation. There have been a number of schemes and algorithms for performing garbage collection which we will describe

in this chapter [BH77, Ung84]. The following steps are either implicitly or explicitly performed in all the algorithms.

1. Identification of accessible objects.
2. Reclamation of the inaccessible memory objects.
3. Compaction of memory to improve locality.

Storage reclamation involves both temporal and spatial overhead. Some traditional garbage collection algorithms take time proportional to the size of memory which makes them inefficient as the size of memory increases. The additional time taken to detect and reclaim all the unreachable memory cells may also destroy the interactive response of the system. All these problems accompanied with the stringent time restrictions imposed by interactive programs necessitates efficient memory management mechanisms in these environments.

General requirements of a memory management scheme are –

1. Short and predictable response times
2. Minimal user interaction with memory internals.
3. Minimal hardware support.
4. Compatibility with memory hierarchy hardware, i.e., cache and virtual memory.
5. Small impact on execution overhead.
6. Capability to reclaim circular structures.

2.1 Traditional Memory Management Schemes for Sequential Machines

Garbage Collection in sequential machines has been well explored and a number of techniques and algorithms have been developed. Most sequential GC algorithms belong to one of three categories: *reference counting*, *mark and sweep* and *copying* collectors. The next step in the evolution of memory subsystems is the design of multiprocessor memory management schemes. As this process has derived heavily from existing uniprocessor schemes, we find it necessary to include an explanation of uniprocessor memory management schemes.

2.1.1 Mark and Sweep Algorithms

The mark and sweep algorithm detects garbage by halting the mutator and then performing two (optionally three) phases. A predetermined root set is used to *mark* all the reachable objects in the first phase, the second phase reclaims dead objects one at a time by walking down the entire memory space and a third, optional phase compacts memory to avoid fragmentation. This algorithm collects all available garbage and uses only one extra bit per word for reclamation purposes. However, the running time of mark and sweep is proportional to the size of memory making this method impractical for large memory sizes. Furthermore, marking algorithms often require a stack to perform the recursive searching process and this ties up memory space when we are already tight on storage. The Deutsch-Schorr-Waite algorithm uses link reversals to circumvent this difficulty but this does not bring about any changes in response times. The major drawback of the mark and sweep algorithm is that the slow and inefficient sweep phase has detrimental effects on the response time in an interactive environment.

2.1.2 The Stop and Copy Scheme

The stop-and-copy garbage collector divides the available memory in a system into two regions – the *fromspace* and the *tospace*. Only one of these regions, i.e., the fromspace, is available to the mutator at any time as illustrated in Figure 2.1. The copying algorithm [Che70, FY69] traverses all the reachable records in the from-space starting at a root pointer set and copies these records into a separate area of memory that is currently unused by the mutator. Forwarding pointers are placed in the from-space to redirect any references to the swapped record. The swapped records are contiguously placed in to-space and this precludes the need for a compaction phase. At the end of collection, the only records in to-space are the reachable ones. The from and to-space pointers are swapped and the mutator now works on the new from-space. As the to-space has a linear area of free memory available, the copying garbage collector does not need to maintain a free list. In other words, we get compaction for free. In this algorithm, the amount of work performed is proportional to the number of reachable cells unlike the mark-and-sweep algorithms where the amount of work performed is proportional to the size of memory. The performance of the copying algorithm is unaffected as the size of memory increases. In addition, the copied, but unscanned records in tospace form the queue for the traversal of other

reachable records. Therefore, we do not encounter the space overhead of maintaining a separate area (stack or queue) to perform the traversals.

2.1.3 Reference Counting Techniques

In *Reference Counting* schemes, every cell (object) in memory is associated with a field called the *reference count* which is a count of the number of references to the cell. The reference count of a cell is continuously updated as pointers to the cell are created or destroyed. When the reference count of a cell becomes 0, the object is no longer referenced and can be collected. Reference counting mechanisms are incremental in nature. Therefore, computation is not interrupted for a significant period of time. A major flaw, however, is their inability to handle circular garbage (because the reference count of any cell in a circular structure can never be zero). Also, when an object is collected, additional work is done in decrementing the reference counts of any child cells and this process can be potentially unbounded. It is possible to temporally distribute this work and utilize a free stack instead of a free list. When a cell is freed, it is pushed on to the free stack. A request for a free cell pops a cell off the stack and at this point, the reference counts of the cell's children are decremented and the children are pushed on the stack if their reference counts are decremented to zero. Each allocation then encounters a bounded overhead and this cost is paid only if needed. Attempts to modify reference counting algorithms to accommodate various implementation issues [Bev87, WW87, DB76, Gol89, IY90] will be discussed now.

Weighted Reference Counting

Here, each reference to an object has both a weight and the address of the object [Bev87, WW87]. The reference count is the sum of the weights of references to the object instead of the number of references. The object becomes garbage when this count goes to zero. When discarding an reference, a delete message with the weight of the reference is sent.

Deferred Reference Counting

The Deutsch-Bobrow [DB76] deferred reference counting algorithm ignores references from local variables. It therefore, spends less time manipulating reference counts as compared to the conventional reference count scheme.

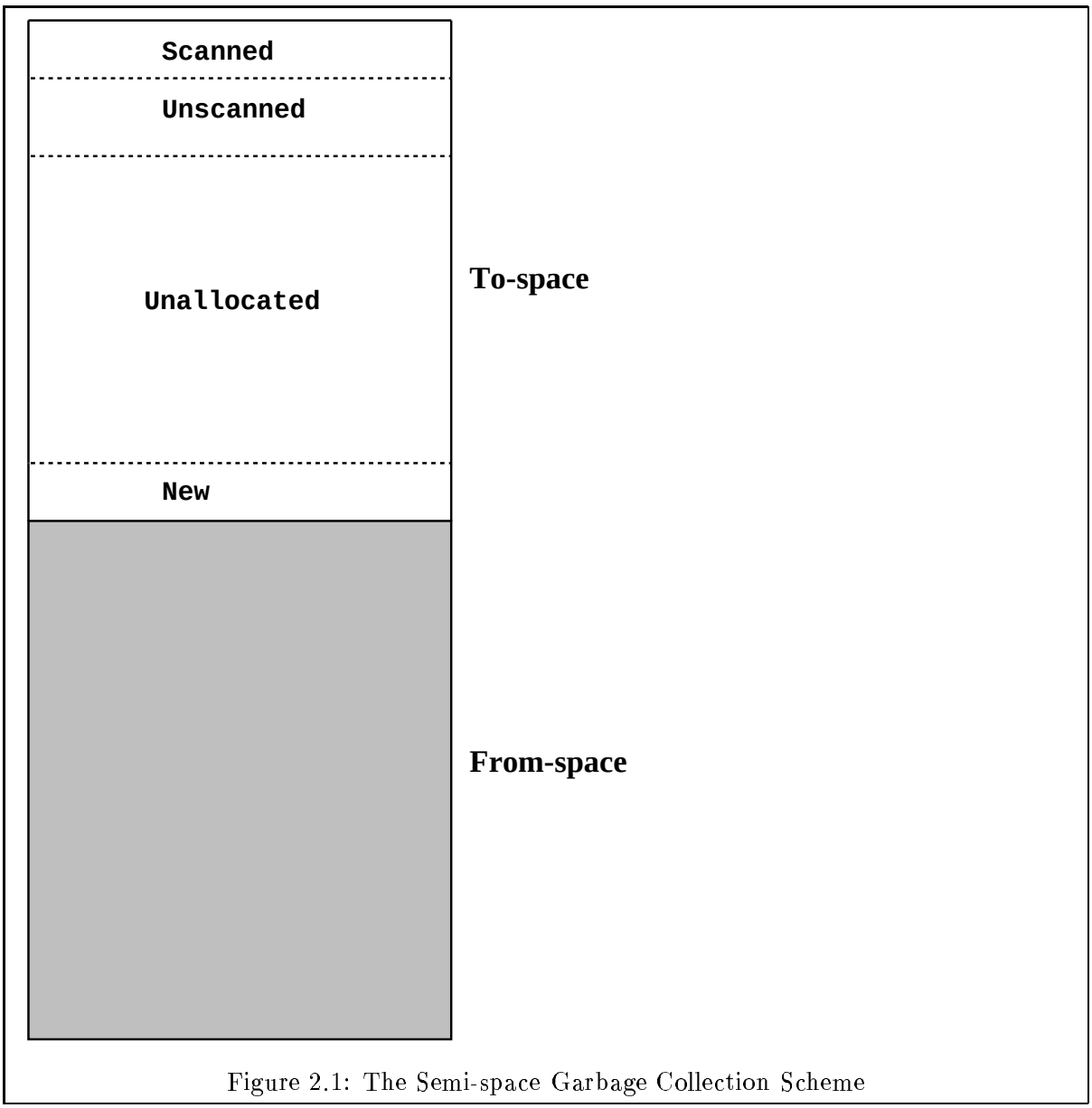


Figure 2.1: The Semi-space Garbage Collection Scheme

All the methods proposed above are inadequate if circular structures must be reclaimed. In most interactive environments, this is a reasonable expectation as a number of self-referential structures may be generated. Some of the methods used to handle cyclic structures in reference counting schemes are articulated below.

A simple minded solution to the problem of circular references is to use reference counting as the primary reclamation method to handle non-circular structures and resort to traditional garbage collection methods (mark and sweep, stop and copy etc.) in order to handle circular structures. Reference counting used in conjunction with other GC mechanisms can be used to improve the performance of those GC algorithms. For example, while using a stop-and-copy scheme, we can put off flipping spaces as long as the reference count system is able to recycle cells. Therefore, the frequency of a flip is reduced. *Group reference counting* [IY90] requires the programmer to define groups of objects that are expected to form cyclic structures. The system keeps a reference count for the group which is updated when some member of the group is referenced or discarded by an object which is not a member of the group. If the reference count of the group becomes zero, all the members of the group can be collected. Groups may themselves be members of a larger group.

2.1.4 Baker's Copying Garbage Collector

In the stop-and-copy algorithm [Che70, FY69], the collector stops the mutator when to-space is exhausted. It then causes a flip (from-space and to-space pointers are swapped) and copies all accessible objects into the new to-space. In Baker's *incremental copying algorithm*, when to-space is exhausted, the collector stops the mutator, causes a flip, but copies only the root objects and resumes mutator activity. Every time the mutator allocates a new object, a few more objects are copied into to-space. Therefore, the copying phase, which is the time-consuming phase of a stop-and-copy scheme is performed incrementally. Every time the mutator accesses an object, the pointer fetched must be checked to see if it points to the from or to-space. If it is a pointer to an element in the from-space, the object must be copied into to-space and the pointer must be updated.

2.1.5 Generational Garbage Collection

Generational garbage collection [LH83] is an extension of the copying garbage collector that takes advantage of two significant observations :

1. High rate of infant mortality - A young object is more likely to die than an object that has survived for a while. Intermediate results are held in newer structures rather than in older objects.

2. Newer records are more likely to point to older records than vice-versa. An older record points to a newer record only if it is altered (reassigned) after it is initialized. This flavor of reassignment is rather unlikely in many languages like declarative languages. Generational garbage collection is claimed to be highly suitable for such languages.

Records with a similar age are grouped together in a contiguous area of memory. Once we have segregated objects on the basis of their age, reclamation efforts can be concentrated on the youngest generation. The drawback of this scheme is what is known in garbage collection literature as the *Tenuring Problem*. Old objects that die may take a long time to be reclaimed. Even worse, they may never be reclaimed until the occurrence of an offline collection. Having many generations allows us to tailor the size of a given generation as well as the tenuring policies associated with the different generations in order to mitigate the effects of the tenuring problem. For instance, promoting new objects quickly keeps the pause times relatively small. The purpose of behind having multiple generations is that there is a strong possibility that an object will be scavenged when it is in a generation that can be collected efficiently. Promotion policies that have been implemented to date are mostly fixed-age policies. Attempts have been made at introducing feedback mediated tenuring policies which are expected to bring about dramatic improvements in the performance of the collector. In fact, measuring the amount of tenured garbage might be a good performance metric for evaluating a generation based garbage collection algorithm. Tunability is one of the salient features of generational algorithms.

Typical factors that one would be concerned about while assessing the performance of an interactive generational garbage collection algorithm are threshold size and promotion policies. Smaller thresholds result in more frequent, shorter collections. But, this also means that more objects are being promoted to the next generation and collection of the second and higher

generations would occur more frequently. Collecting higher generations results in significant pauses and clever promotion policies are necessary to mask this unavoidable tradeoff. The *copy-count* technique, where objects are promoted to a higher generation after they have survived a pre-determined number of scavenges in one generation is common solution to this problem. Appel argues that a garbage collector based on the generational method could potentially be so fast that allocation and not collection would prove to be a bottleneck and has proposed a scheme to alleviate this problem by fast record allocation is [App89].

2.1.6 Ungar's Generation Scavenging

Ungar's generation scavenging algorithm [Ung84] (see Figure 2.2) arose out of efforts to find an efficient, non-disruptive garbage collection algorithm for a uniprocessor implementation of Berkeley Smalltalk [Ung86]. Scavenging is the process of moving live objects from a currently active space to a new area and it involves a traversal of the graph of reachable objects starting from a root set. The Generation Scavenging (GS) Algorithm is based on the observation that nearly 95% of garbage cells can be reclaimed while they are still in the first generation. In the GS algorithm, objects are categorized as being either old or new. Old objects reside in an *old* or tenured space and do not participate in the marking and scavenging process.

New objects can be in one of the three areas that are actively involved in the garbage collection process. The three areas are

1. *NewSpace* - All new objects are created here. This is a comparatively large area.

2. *Past Survivor Space* - It has objects that have survived previous scavenges.

3. *Future Survivor Space* - This region is empty during program execution and is used during scavenging to copy live objects. Both the Past and Future Survivor spaces are small in size - this is in conformity with the assumption that these small areas are sufficient for the small, frequent garbage collections that will improve the interactive response of the system.

During a collection, live objects are copied from Newspace and Past Survivor Spaces into the Future Survivor Space and the two spaces (Past Survivor Space and Future Survivor Spaces) are swapped. When an object has survived a particular number of scavenges, it is tenured into

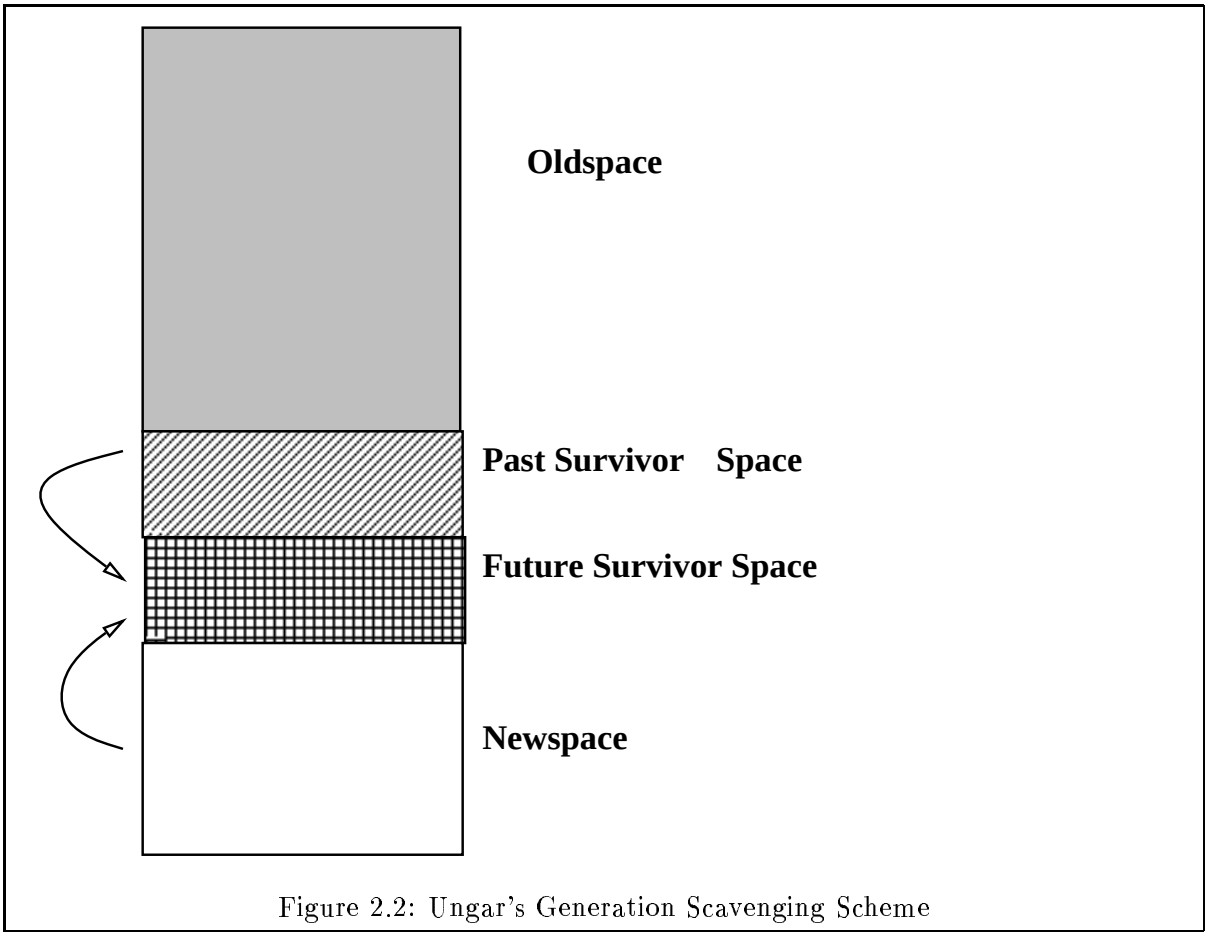


Figure 2.2: Ungar's Generation Scavenging Scheme

Oldspace. Any stores into Oldspace that reference the New or Survivor Spaces are placed in a *remembered set*. The remembered set is used as part of the root set along with the machine registers to determine the accessible objects. The remembered set is also scavenged to remove any invalid references from old objects to new ones.

Performance measurements of the Generation Scavenging algorithm in Berkeley Smalltalk II showed that only 1.5% of the total CPU time was spent in garbage collection. Space consumptions were of the order of 10% of main memory as all spaces were not required to be resident in main memory. Collection of Oldspace occurred rather infrequently (especially since the amount of space to be collected here is significantly large), say, every 3-8 hours by an offline mark-and-sweep. But, objects that die soon after being tenured into Oldspace are not reclaimed for a long time and block memory. The seriousness of the tenuring problem could be mitigated by imposing stringent tenuring policies that could be adaptive and/or take hints from the user programs.

2.1.7 Moon's Ephemeral Garbage Collection

This is a garbage collection mechanism designed for a high-performance Lisp system - A LISP machine with a large virtual memory [Moo84]. Early LISP systems introduced such long garbage collection overheads that programmers made substantial efforts to reduce the rate of creation. The price paid for this tailored efficiency is the loss of abstraction, consequently reducing readability. The Symbolics 3600 machine [Moo85] adopted an interesting mix of solutions to overcome the handicaps of existing storage management implementations. They include -

1. Incremental garbage collection - The Symbolics 3600 uses Baker's algorithm which is incremental in nature. The copying phase is temporally distributed and only one object is copied at a time. The 3600 however allocates virtual memory to spaces from a free pool.

2. Approximately Depth-first Copying - This alleviates the problems caused by poor locality of data structures, i.e., scattering of data by intervening garbage. Depth first scanning puts parent and children objects on the same page.

3. Ephemeral Objects - This strategy sorts objects on the basis of their expected lifetimes and focuses garbage collection efforts on objects that have a high probability of being garbage (ephemeral objects).

4. Tagged Architecture - and hardware assists facilitate efficient meshing of the implemented software strategies. Tags are required to distinguish between addresses and other values, primarily for the purpose of introducing indirections from the from-space to the to-space. A tagged architecture expedites this process by requiring that anyword in memory be tagged as a pointer or otherwise, not just LISP objects. The extra hardware required to implement this is about two percent of the CPU hardware and the time penalty incurred is zero as tag checking can be performed in parallel with computation.

2.2 Multiprocessor Garbage Collection

The desire to exploit concurrency among tasks in a job complicates storage management. This is due to the need for synchronization between the various tasks in order to ensure their correctness and consistency. Programming environments for highly parallel machines, it appears, are largely centered around dynamic resource management schemes which require sophisticated memory allocation and reclamation schemes. Furthermore, one can think of storage reclamation itself as a set of concurrent processes [SS91].

All this concurrency requires coordination between the parallel subtasks of a single job which in turn requires efficient and reliable communication. Deficiencies of traditional algorithms in a parallel setting have been detailed upon in [HK82]. Many of the insufficiencies arise from the fact that the traditional algorithms halt the computation process, preventing real-time response by the system. In other words, the collector (thread of control that is responsible for garbage collection) is initiated after the mutator(s) (threads of control corresponding to the application program) have been halted.

The main overhead in parallel GC occurs in maintaining intraprocessor and interprocessor references to an object. If this is not done, it is possible for an object to be collected even while a reference to it exists, violating the correctness criteria of the algorithm.

Automatic storage reclamation is important in interactive and real-time programming environments because programmer controlled reclamation is highly prone to errors. Further from an

aesthetic viewpoint, we would like to relegate low-level memory management issues to the system. Our goal is to guarantee non-disruptive garbage collection that would otherwise interfere with the mutators executing on the different processing units.

2.2.1 Storage Management in Shared Memory Multiprocessors

One of the first steps in isolating storage management issues from computation was to have the threads of computation and memory management tasks run simultaneously, but on different processing units. Also, locking mechanisms must be implemented to prevent two processors from simultaneously manipulating the same memory location.

Steele's Algorithm

This is one of the earliest parallel garbage collection algorithms [Ste75]. A dedicated storage reclamation processor handles garbage collection of a memory space that it shares with a mutator. In the general case, this algorithm could be extended to handle any number of mutators that share a single address space. Hence, this algorithm is applicable to a tightly coupled system. A number of similar on-the-fly mark-and-sweep based algorithms have been extensively to prove the correctness of the distributed GC process [BA84, DLM+78, Gri77, KS77].

Mul-T on the Encore Multimax

Mul-T is a parallel Lisp implementation designed to run on the Encore Multimax. T is itself a dialect of Lisp enhanced with parallel constructs. Mul-T differs from the standard T implementation in the handling of dynamic binding, storage management, task management, the inclusion of futures (from Multilisp) [Hal85] and implicit touches(to activate futures). It is more efficient than the Multilisp implementation as it uses the Orbit compiler [KKR+86] instead of a level of interpretation as in Multilisp.

There is a single global variable, *heap pointer* that points to a global heap. Each allocation request is satisfied locally out of a chunk of memory assigned to that processor. Large objects are allocated directly from the global heap to minimize fragmentation. When a local processor runs out of memory, it is replenished from the global heap. Garbage collection is invoked when we run out of space in the global heap. Mul-T implements a parallel stop-and-copy garbage

collector. The fact that it requires relatively fine grained synchronization argues against its implementation on available hardware. A parallel copying collector must atomically copy an object leaving a forwarding pointer behind at the old location.

Real Time Garbage Collection on Stock Multiprocessors

A garbage collector for the Firefly shared memory multiprocessor [TS87] designed at the DEC Research Center is described in [AEL88]. This collector based on Baker's incremental stop-and-copy algorithm takes advantage of virtual memory paging characteristics. Baker's algorithm assumes every address seen by the mutator to be in to-space. In order to ensure this, a check is made even if the object is already in to-space. If the referenced address is not in to-space, it is moved into to-space and the access is repeated.

In the scheme proposed by Ellis, Li and Appel, virtual memory updates only addresses that it knows are to-space references instead of testing every reference. To-space is divided into scanned and unscanned areas just as in Baker's algorithm. The virtual memory protection is set up in such a way that both the fromspace and unscanned portions of to-space are inaccessible. When a reference is made to an object in these areas, a page fault is generated. For faults that access the fromspace, the collector causes a move of the object from from-space to the end of the unscanned region of tospace. The access protection is not changed as the object is still in the unscanned region. For references into the unscanned portion of to-space, the page fault handler is modified to perform the following series of actions:

1. Scavenge all the objects on the faulted page, relocating the referenced objects in fromspace to the unscanned portion of tospace.

2. Leave a forwarding pointer in fromspace for all the moved objects.

3. Unprotect the faulted page.

A flip occurs when to-space gets filled to a threshold value. At this point, the remaining unscanned pages are collected.

The Berkeley SPUR Project

The focus of this project is to produce a shared memory RISC-multiprocessor specifically suited for Lisp implementations. SPURLisp has explicit constructs for concurrency and synchronization. These include:

1. **Processes** that represent concurrent tasks
2. **Mailboxes** that provide exclusion synchronization and interprocess communication.
3. **Signals** that are responsible for asynchronous interprocess communication.

Spur Lisp uses generational garbage collection as the basic storage reclamation scheme. This is not an incremental process and all processors halt during the scavenge operation. However, each processor maintains a free area from which any local allocations are done.

Multiprocessor Smalltalk on a Firefly Multiprocessor

Multiprocessor Smalltalk is an extension of the Berkeley implementation of the Smalltalk-80 language to execute on an experimental shared memory multiprocessor, the DEC-SRC Firefly. Three techniques have been identified to handle concurrency in Multiprocessor Smalltalk [PU88]:

1. Serialization of accesses
2. Replication of resources
3. Reorganization of process structure

Memory Management in Multiprocessor Smalltalk is serialized. The rationale for this is that the garbage collection mechanism chosen is not disruptive to the interactive response of the system.

The garbage collection mechanism used is *Generation Scavenging* [Ung84]. The presence of an explicit Newspace from which allocation of new objects can take place simplifies memory allocation. The Multiprocessor Smalltalk interpreter suspends all other activity when garbage collection is going on. This is necessary to ensure correctness, especially with forward references and indirection pointers across processors. The underlying operating system synchronizes all

the processing units with a global flag. A separate remembered set is maintained which records old objects that point to new ones.

Non-blocking Garbage Collection

Most of the shared memory multiprocessor garbage collection schemes we have seen require some synchronization mechanism to maintain consistency. This is not always possible, especially in the presence of asynchronous systems with faulty processes and arbitrary delays. Non-blocking storage management schemes [HM90] allow a process that is allocating or garbage collecting to be delayed for a substantial amount of time without blocking the other processes. The algorithm proposed in [HM90] does not busy-wait or use locks.

The underlying MIMD machine has a number of simultaneously executing asynchronous processes addressing a common shared memory. The primitive operations of this memory are *read, write* and an atomic *compare-and-swap* operation. Memory is divided into a number of contiguous regions, one for each process. A process can allocate or garbage collect exclusively within its own region, but can access other regions. Decentralization of control in this manner increases concurrency. Note that different memory management schemes can be applied to different regions. An object in memory is a list of versions, each version is a contiguous block of memory within a process's region. A *store* operation creates a new version without destroying an older version. Non-blocking updates using versioning allows us to perform multiple updates on objects without mutual exclusion. It also permits us to move objects in a copying collection without locking the object. Each contiguous memory region is partitioned into a single to-space, zero or more from-spaces and zero or more free-spaces. The copying algorithm is incremental in nature – old objects are moved from from-space to to-space while the computation process is going on. Once it has been determined that there are no more accessible objects in from-space and all pointers have been readjusted, the current from-space is added on to the set of free spaces. When the available memory in to-space reaches a threshold, it becomes a from-space and one of the free-spaces becomes the to-space.

It is not clear how the correct versions are accessed or updated. We suppose that every access is tagged with a system inserted version number.

2.2.2 Storage Management in Distributed Memory Multiprocessors

Coordination between the parallel subtasks of a single job brings about the necessity for communication. With state-of-the-art communication technology, we can say that intra-processor communication is less expensive than remote communication. Even with high-performance networks that communicate via wormhole routing and virtual channels, efforts must be made to reduce the amount of off-processor communication- the limiting factor here is not the speed of communication , but the bandwidth.

This is one of the primary motivations for distributed memory machines, others being for purposes of scalability.

In distributed architectures, each processing unit is associated with its own local memory. A global communication network connects the different processing units together. As intraprocessor communication is orders of magnitude faster than interprocessor communication, we are forced to exploit locality and manipulate data in the local memory as far as possible in a distributed memory machine. The unique features of distributed memory machines especially when integrated in a massively parallel framework complicate storage reclamation. Deficiencies of traditional algorithms in this setting have been detailed upon in [HK82]. There are two levels of garbage collection on distributed memory machines, i.e, global and local garbage collection. Global GC is a systemwide collection involving all processors in the system. Local GC is carried out independently on each processor without inhibiting the processing activities on another processor. We now discuss the costs associated with both local and global GC.

The interval of a global garbage collection is dictated by the first processing unit that needs memory. Other processors have to cooperate even if they have sufficient local memory to continue work. The overhead of synchronization - orchestrating a global start and stop could be substantial, especially when the collection process has to be real time. In other words, the deviations in the time taken by different processing elements to run out of space must be more well balanced. Hardware techniques like logic and signal lines or software mechanisms like barrier synchronization can be used to reduce the synchronization overhead.

Local garbage collection requires a reference management table to determine whether an object is wholly local or not. Local references will slow down if all accesses have to go through this reference management table. By separating local and remote references we can cause a

level of translation only if the object is non-local. Alternately, translation costs can be overcome with a hardware translation cache. The entries in this table should be periodically updated and eventually reclaimed. If a global reference table is not used, a mechanism for conversion between local and remote references must be available. With only local collection, reclamation of remotely referenced data must wait till the next collection. When a processing element starts local collection, it is difficult for other processors to access data on this processor.

We shall now assess the existing garbage collection schemes in terms of their suitability as a distributed garbage collection mechanism.

Reference Counting Collectors

While implementing reference counting schemes on a distributed memory machine, the ordering of references to the object must be preserved. Otherwise, it is possible for an object to be collected even while a reference to it exists. We will refer to object accesses as *incrementing* or *decrementing* accesses. Consider the following pathological case.

We have a popular object A in a mesh multicomputer that is being accessed by many other objects frequently. Say that a series of reference count manipulations have brought its reference count to 1. An incrementing access is issued from a distant object B. Following this a decrementing reference is issued by a neighboring object C. It is very likely that the decrementing reference reaches A first. If a garbage collection is initiated before B's reference can reach A, A will be destroyed leading to a *distributed dangling reference.*

The space overhead needed to maintain reference counts must be kept in mind, as allocation of new objects includes the allocation of reference count bits. The inability to handle either local or distributed cyclic garbage is another disadvantage of this approach.

Generational Reference Counting [Gol89] is a distributed storage reclamation scheme for loosely coupled multiprocessors. Here, each reference has a generation associated with it.The original reference to an object is a zero generation reference, any reference copied from the original reference is a first generation reference and so on. Each object has a ledger that keeps track of current references in each generation. The object can be reclaimed when there are no outstanding references to the object from any generation. This method has the drawback that circular structures cannot be collected.

Mark and Sweep Based Collectors

The simplest collectors stop all activity in all nodes during garbage collection. This is especially undesirable in interactive systems due to the disruptive and abrupt pauses that are caused. It is also undesirable in distributed systems where processing activity cannot be halted. Mark and Sweep collectors generally use a coloring algorithm to distinguish accessible objects from inaccessible ones. All circular structures are collected and the collection is complete (no garbage is left behind) These advantages are offset by the non-incremental and sequential nature of the collection. All objects, accessible or inaccessible, must be visited in the sweep phase and this overhead is disruptive especially in the presence of virtual memory. Most implementations modified to incorporate concurrency require cooperation between the processes and the collector. Even the concurrentized version repeatedly scans the entire memory.

Typically, a M and S collector will require the user processes to cooperate with the collector by setting the color bits when making an assignment. This may be expensive, especially across a network. We will shortly discuss different classes of GC schemes based on the mark and sweep strategy [Ali84].

Copying Collectors

Copying garbage collection in a parallel environment involves many subtle issues.

1. Correct manipulation of interprocessor pointer references.

2. Performance scalability of the memory management scheme.

In a distributed environment, if the object to be copied resides on a remote processor, the copying request must be issued to the remote processor. A reference table is needed to keep track of the current location of the object. It is not possible to predict when the remote processor will complete the copying request and return the new location of the object. Therefore, the turnaround time could be indefinite.

Two solutions have been suggested to solve this problem, neither of which require a reference management table.

1. Synchronous Readback or Two Phase copying:

The algorithm has two phases - a *copying* phase where objects are copied (possibly into the local memory of a remote PU), and a *restoring* phase where remote references are restored. All the processing elements transit each phase synchronously.

2. Asynchronous Writeback: When remote copying is requested, the address of the requester is sent along with the object to be copied. The new address is written back by the remote processor.

By implementing Baker's incremental algorithm on a distributed system, distributed copy collectors can be made real-time.

Generation Scavenging on Distributed Memory Machines

In extending a distributed copy collection scheme to realize generation scavenging, a new overhead must be faced; managing the *remembered set.* A remembered set is a set of references from older generations to the new generation. Some processors adopt a hardware mechanism for detecting a remembered reference (hardware tags, for example). An exception occurs when a remembered reference is detected and an exception handler adds this reference to the remembered set. [Z+87, Ung86]. Possible solutions for handling remote remembered references:

Solution 1: Record the remembered element (in oldspace) in the local memory of the oldspace as illustrated in Figure 2.3.

Advantages:-

1. Registering the remembered root pointer is local in the processor where such a reference is needed.
2. Removal of this reference is also easy; when the reference no longer exists, it can be removed - reducing the size of the remembered set.
3. No new communication primitives to register a remembered root pointer are necessary.

But, network traffic is incurred on every collection to request the copying of newspace remote objects. This overhead exists as long as the referenced object lies in Newspace.

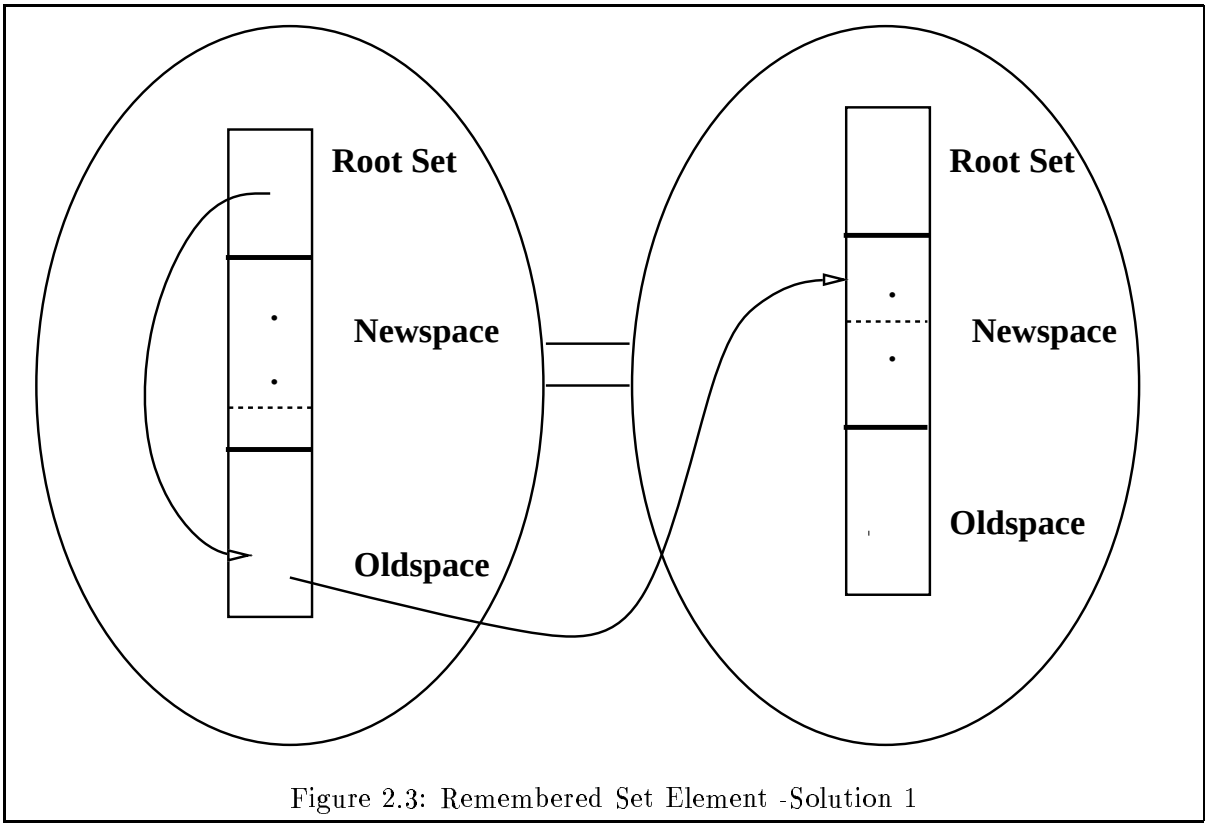


Figure 2.3: Remembered Set Element -Solution 1

Solution 2: Record the Newspace object that is pointed to by the Oldspace reference in the local memory of the Newspace element as shown in Figure 2.4.

Advantages

Network traffic occurs only once to register a remembered root pointer to a remote PE.

Disadvantages

1. The registration of the remembered root pointer requires a network communication.
2. Size of the root set increases monotonously.
3. Requires special primitives to register a remembered root pointer.

Hudak and Keller's Algorithm

The target machine for most of the algorithms suggested by Hudak and Keller [HK82] are highly parallel distributed machines. These algorithms have been derived from traditional mark-and-sweep techniques and scavenge, in addition to garbage objects, any irrelevant tasks.

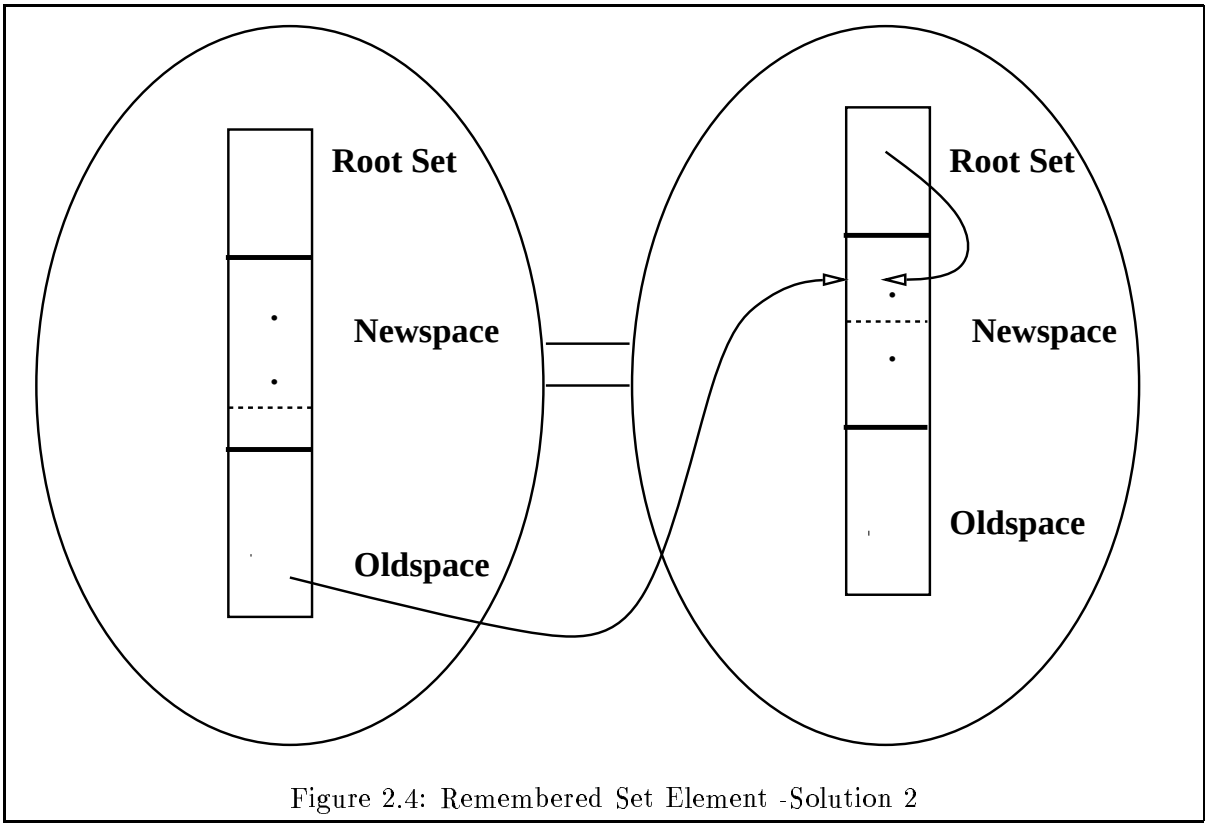


Figure 2.4: Remembered Set Element -Solution 2

The mark phase is a universal, distributed phase where every processor participates in the marking process.

Hudak's algorithms use a shared tree structure instead of a centralized stack to implement the mark phase of a conventional collector. This eliminates, to a large extent, the communication bottleneck caused by a single shared stack and portrays effectively the distributed nature of the parallel marking process. Completion of the marking process is indicated when the marking tree collapses to an empty tree - analogous to an empty stack.

The **Marking Tree** collection algorithm has two phases.

1. The marking phase is essentially a reformulation of the conventional marking scheme where each marking operation corresponds to the spawning of a mark task. Synchronization and termination mechanisms are built into this procedure. Each spawn task backpropagates in the marking tree and termination is detected when a spawn task is issued from the root node. The scheme use for marking is similar to Dijkstra's algorithm and nodes are marked white, black or grey.

2. The sweep phase has three subphases:

   (a) **Task Deletion:** This phase is responsible for deleting tasks that point to inaccessible nodes and prevents active nodes from spawning any irrelevant tasks.

   (b) **Node Reclamation:** Once the irrelevant tasks are removed, all the unreachable nodes are deleted and returned to the free list.

   (c) **Reinitialization:** This is a process of color reversal at each processor. Nodes may be colored black or white. Once node reclamation is done, the mutator at each processor sees only black accessible nodes. Color reversal is occurs at this point where each mutator now views these nodes as being colored white, indicating that it is ready for the next mark phase. A new mark phase can begin when all the processors (mutators) reach this state.

Drawbacks of the marking tree scheme include the lack of compaction of the resulting heap area. Merging a compaction phase into the sweep complicates issues in a distributed environment. The authors suggest combining the marking tree concept with that of a copying collector; a local copying collector for GC at each node accompanied by the marking tree algorithm to coordinate inter-processor references.

Ali's Algorithm

Ali [Ali84] has developed different classes of garbage collection schemes based on the mark and sweep strategy. They are the *global, local-global, distributed-local* and *distributed real-time* schemes.

Each processing unit (PU) handles any local garbage collection by a local mark and sweep. After a local collection has been performed, the processor informs all the other processors about remote pointers which must be treated as roots in the respective garbage collections. Synchronization requirements are greatly reduced as a large portion of the collection is done locally. Garbage collection messages can be batched together into large blocks. This results in an effective use of the communications medium. However, circular structures distributed across processors are never collected.

The global scheme is well-suited for applications with non real-time requirements. All user processes in the system are halted during garbage collection. The marking phase marks all

accessible objects in the system. Traversing the graph of reachable objects is both depth-first and breadth-first. Increasing the degree of "breadth-firstness" allows us to exploit more parallelism, but the resulting space utilization may be excessive. Depth-first traversal has a low space overhead and the degree to which a traversal must be depth first can be parameterized.

There are two ways of carrying out a global garbage collection.

1. The collection process can be controlled by a single *master processor*.
2. Alternately, the control of global garbage collection can be distributed over the entire system and any processor could be the *master*.

In either case, the master is responsible for synchronizing the start and finish of the marking phase. As the system needs to be frozen over a collection, a number of processing units are idle and this overhead cannot be tolerated in a massively parallel system.

In local-global schemes, the systems exploit locality and most memory requests are satisfied by a local collection. This reduces the frequency of global collection and the overhead associated. Global garbage collection is invoked only if a local collection cannot satisfy a memory request.

The distributed local scheme is designed to be used in systems with large numbers of processing units. Each processor can carry out a local garbage collection independent of other processors. Thus, the impact of frequently collecting processes on less active processes that do not require frequent collection is low. However, as objects and references to objects may be distributed across processors, interprocessor references must be maintained. At the end of a local collection, each PU informs all the other PU's to which it has remote references. Thus, each PU has a remote reference table indicating local objects referred to by remote PUs.

The fourth scheme of Ali, i.e., the *distributed real time* scheme is an extension of Baker's copying collector.

Hughes' Algorithm

Hughes' algorithm for distributed garbage collection is very similar to Ali's algorithm in that it performs local garbage collection at each processor. However, here each processor independently performs a global garbage collection (This can be done in parallel). Each global garbage collection marks nodes differently by stamping them with the time at which that collection started. A node is counted already marked if its time stamp is at or later than this time. The

root nodes of the collection are thus automatically considered marked. A local garbage collection marks the accessible nodes on that processor with the time stamp of the root nodes on that processor. Messages are sent only to those remote nodes whose time stamps exceed that of the local processor. Each processor keeps track of the earliest global collection for which it has work left to do. Termination of a task initiates the collection of nodes that are no longer needed. The response is therefore pseudo real-time as the garbage collection overhead is temporally distributed. When the task associated with garbage collection T has been completed, we can globally delete all nodes with time stamps less than T.

This algorithm assumes that the model embeds a notion of global time. But, it does not require a master processor or a global garbage collection halt. Local garbage collection causes all activities on that processor to stop. Hence this algorithm is *pseudo-realtime*. Furthermore, interprocessor communication does not involve housekeeping messages. This is handled individually by each processor.

DistributedSmalltalk

The DistributedSmalltalk [SB88] system consists of cooperating Smalltalk virtual machines distributed over a network that provides complete distribution transparency to the image level. There is a loose cooperation between hosts. Hosts are free to do a local garbage collection anytime. No remote hosts are involved during a local collection. Only remote references are gathered. In fact this information may not even reach the remote processor when the remote host is down. Therefore, remote references are used for the next garbage collection.

Apart from this, there are two global garbage collectors that handle external references :

1. A fast collector that does not reclaim cyclic structures and
2. a slower one that reclaims cyclic structures.

Both these schemes are variants of the mark-and-sweep strategy. This is not a suitable mechanism for communication intensive tasks where the remote reference table would be large in size.

Emerald

Emerald [H+87] presents a single unified model of computation for both distributed and non-distributed programming. The other motivating forces behind the design of this language are the introduction of fine-grained mobility in the system with sufficient language support to enhance performance and efficiency.

Emerald combines the uniform object model of Smalltalk with the distributed object model of Eden [TBLN85]. Emerald objects have a unique identity across the network as well as a local data representation. An optional **initially** section specifies the code that must be executed when the object is created. Unlike conventional object oriented languages, construction of Emerald objects occurs explicitly through a constructor and not via invocation. Objects in Emerald are strongly typed. Every variable and expression is typed and must conform to the declared type. But, as objects in Emerald may conform to more than one type, the view (operations that can be performed by the object) can be either narrowed or widened as desired.

Emerald has a two-collector storage reclamation scheme.

The Node Local Collector runs independently of other nodes. Many objects are created and referenced on a single node. Each object has a descriptor called the *Reference-Given-Out* bit that is set when an internode reference has been present at some point in time.

The Distributed Collector requires the nodes to cooperate to collect garbage spanning multiple nodes. It uses a modified mark and sweep algorithm. By having two sets of marking bits in each object, both local and distributed collection can go on simultaneously. Initially, a collecting process is started in every node and all nodes go on in parallel. For a non-local node, the collector sends a mark-grey message to the node on which the non-local object resides. This node processes the message and sends back an object-is-black message when it is done. All data areas accessible to every executable process are marked black before the process is allowed to run again. The marking phase is complete when there are no more grey objects left. It is possible to recheck for nodes that might have been unavailable using this strategy.

ACTRA - An Actor Based Implementation

The ACTRA project [TLP86] is an extension of Smalltalk to introduce actors [Hew80] which are abstractions for concurrent object oriented programming. An actor is an object that is associated both with a state and a behavior. Concurrency in actor-based systems is exploited because an actor can send messages to other actors without terminating its own execution. Actra is itself a Smalltalk virtual machine enhanced with real time support. Each processor in the system runs a copy of the Actra kernel. Garbage collection is entirely local and uses a variant of the generation scavenging algorithm. Entry tables hold interprocessor references to be used as part of the root set during GC. Each processor executes and performs GC at its own pace without interfering with the execution of other processors.

The J-Machine

The J-Machine [Dal90] is a scalable, highly parallel fine-grained MIMD machine consisting of special purpose processors that are loosely coupled by means of a high-speed, low-latency network. The J-Machine is designed to provide low-overhead system services to support the actor programming paradigm [Agh86]. It is fine grained in the sense that it supports fine-grained tasks and has fine-grained processing nodes i.e. nodes that cost less and have lower memory capacities than coarse-grained nodes. To efficiently coordinate the activities of the different fine grained processing elements, efficient and powerful system services are necessary. The challenges of building a fine-grained operating system are considerably different from those encountered in conventional distributed operating systems. The system services must be designed to handle fine-grained tasks that can context-switch rapidly. The system must also be capable of migrating and protecting these independent units.

The first version of the operating system called the JOSS or Jelly Bean Operating System [Tot87] was a basic object based storage model, where typed objects are the basic unit of storage. A more recent version of the operating system for the J-Machine is COSMOS [HTD90], based partly on JOSS.

Efficient and inexpensive task management is implemented by using the hardware task scheduling and dispatching facilities of the message driven processor [DCC+87]. The message driven processor is a high performance processing element designed for a message passing MIMD

machine. It is a memory-based architecture that permits very fast context switching and specialized hardware that reduces message reception overhead.

The flexibility of a shared memory machine and locality and extensibility of a message passing machine are both exploited by means of a global object namespace. Each object is referred to by its name or unique ID and this allows relocation within the heap or across the network. A name-manager that associates virtual names with physical memory addresses is used to determine the location of virtually addressed objects.

The local memory at each node of the J-Machine can be accessed very rapidly. Part of this local memory serves as a heap. A heap manager handles allocation requests and allocates space from the top of free memory. Storage reclamation is done by a mark and sweep collector. Although the marking phase can be done in parallel, the mark phase requires the entire system to halt. The sweep phase can however, be done wholly in parallel. Compaction on local memory is extremely fast as it entails only an update of the address in the nametable.

Garbage Collection of Actors

The criterion commonly used to detect garbage in conventional garbage collectors is reachability. When dealing with active objects there is yet another factor that needs to be considered – computability, i.e, whether the object is active or may become active. Reachability criteria are further complicated by the fact that addresses of actors can be communicated making them reachable. Therefore garbage collection of actors (active objects in general) differs fundamentally from garbage collection in other systems. Collectors for actor-based languages have been developed by [KWN90, Nel89, Was90]. Most of these algorithms are of a mark-and-sweep flavor augmented with coloring algorithms. A garbage actor is defined as one that:

1. is not a root actor
2. is not an acquaintance of any other active actor
3. cannot potentially send a message to a root actor.

The collector is based on a set of coloring rules [Nel89] (see Figure **??**) implemented via a co-routine algorithm called the Push-Pull algorithm [KWN90].

The drawback of these systems is that they require the actor system to be halted during the entire marking and collection process. In a parallel system as mentioned earlier, this imposes

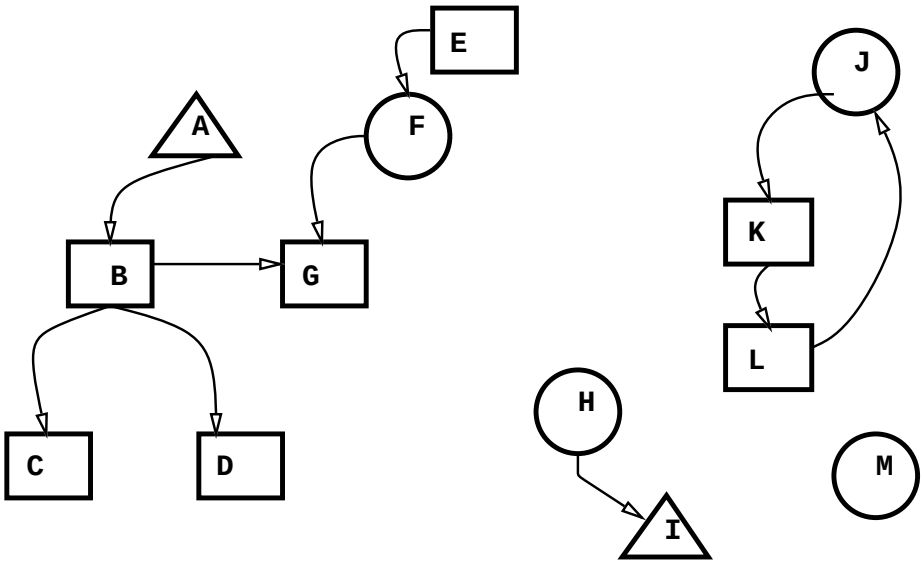


Figure 2.5: Garbage Collection of Actors

additional synchronization requirements while beginning the marking phase as well as while resuming execution.

Halstead's Reference Trees

This is a technique for garbage collection of computational objects. A computational object has both code and data associated with it. In addition , it specifies scheduling attributes. In the context of actors (computational agents that model distributed computation), an actor is garbage if it is not processing any messages, if it is not the target of some undelivered message and if it is not the acquaintance of any other actor(except itself). A mark and sweep algorithm proposed for the collection of actors starts by marking all actors that are not computable. The computable actors(those that are currently processing or those that are targets of messages not yet received) form the root set for the marking process. All acquaintances for these computable actors are marked and this is a recursive process. All unmarked actors can be reclaimed.

Halstead proposed the following scheme based on the notion of an actor reference tree [Hal78, Hal79]. An actor reference tree has a set of processors and connections to indicate references to actors. The reference tree for an object has as nodes processors that contain references to the object. Therefore, the reference tree is connected and acyclic.

Garbage collection is a tricky issue here as there may be more than one reference to the object. None of the referencing processors will collect an object that might possibly be referenced by other processors. While garbage collecting, the reference tree of an actor is reduced until it contains a single processor. Following this, local garbage collection at each processor determines if the actor is garbage. The problem with this scheme is that cyclic structures cannot be reclaimed in some cases.

The Mushroom Project - A Multiprocessor Smalltalk Machine

The Mushroom Project [Wil87] is aimed at developing a high-performance object oriented system. It involves the design of a shared memory multiprocessor oriented towards symbolic environments in general and Smalltalk in particular. It is based on a high speed bus that allows a shared memory distributed across processors. Mushroom supports an object based virtual memory, accessed by virtual pointer-offset addresses into a single object table. System performance can be improved by maximizing locality, i.e., the data required by a processing unit should be available in the local memory of that processing unit minimizing interprocessor references. An efficient virtual memory hierarchy is provided by swapping groups of objects between real memory and disk so that related objects eventually reside on the same page. The set of objects to be grouped together is determined dynamically based on memory access patterns. The technique of dynamic grouping on an object level [WWH87] guarantees that the fine-grained locality characteristic of symbolic processing is exploited. Simulations have shown that this mechanism has demonstrated significant improvements in the paging behavior. Tag bits are associated with every memory word that are used for run-time type checking. In addition, the tags can be used for garbage collection purposes , e.g., to keep track of reference counts or generation information. The garbage collection scheme envisioned for the Mushroom project is an integrated collection scheme, traditional mark and sweep for the cache, a stop-and-copy mechanism for main memory and reference tagging in virtual memory. As cache memory is relatively small, a full scale mark and sweep on the cache is not very time-consuming. No implementation or performance details of the integrated memory management scheme are available to date.

2.3 Improvisations

In this section, we will look at the following extensions to existing memory management schemes:

- Interactive and Real time Garbage Collection
- Virtual Memory Implications on GC
- Fault Tolerant Garbage Collection

2.3.1 Virtual Memory Issues

The fundamental assumption made here are that only part of the entire accessible address space is in use at any particular time and that the data used by a program has locality. Therefore, a garbage collection algorithm that is used in a virtual memory environment must preserve the locality of data structures in addition to collecting garbage. The storage manager must be capable of supplying virtual addresses to the user besides maintaining consistency of information by fetching and storing data from disk as and when needed.

With a virtual memory system, it is not desirable to traverse all of memory during garbage collection as this would entail accessing every location in virtual memory which can result in excessive overhead and thrashing. In early systems where this was the case, unpredictable and slow response times forced users to write programs that suppressed object creation. In fact, users were willing to pay the penalty of an occasional system crash and turn off garbage collection than put up with this unpredictable garbage collection overhead. The extent to which virtual memory and the storage reclamation strategies accommodate each other influences the performance of the garbage collection algorithm significantly.

Virtual Memory Performance of Traditional Garbage Collection Algorithms

Both reference counting and mark-and-sweep algorithms tend to perform poorly in VM systems due to an excessive number of page faults. This is because live objects are scattered sparsely all over memory after a collection.

A compaction phase can improve locality of reference in mark and sweep but will not improve the locality in a reference counting scheme. Reference counts themselves occupy space and have to be manipulated on each access. This can cause a small fraction of page faults and

decrease the data locality. Isolating the reference counts from the data object and updating reference counts separately is a viable solution to the above problem. It may be possible to delay updation of reference counts until the corresponding pages are brought into main memory.

Appel and Li studied [AL91] virtual memory primitives like page protection techniques that can be used for various purposes, especially garbage collection.

1. Concurrent Garbage Collection on shared memory machines:

   While using Baker's real time copying algorithm on shared memory machines, it becomes necessary to check every pointer reference to see if it is already in tospace. Otherwise we must move it into tospace from fromspace. By setting page protection bits to all objects in fromspace, references to these objects can be trapped and moved into tospace. This technique has been discussed in Section 2.4.

2. To build up the *Remembered Set* in generational garbage collection schemes:

   To recall, the remembered set contains elements in older generations that have pointers into younger generations. This can occur only if there is an assignment to an already existing record in the younger generation. Generally, such assignments are detected either during compilation or by using special purpose hardware. But, this can be expensive. With the addition of a special *dirty* bit, all assignments made to oldspace objects can be trapped and checked to see if they point to elements in younger generations. The trap handler can then add the address of the oldspace location onto the remembered set.

2.3.2 Interactive and Real-Time Garbage Collection

Interactive response times are usually evaluated in terms of the duration and frequency of pauses due to garbage collection. What is considered an acceptable interactive response is dependent on a number of factors like the system and application [Gab85]. Incremental collection avoids obvious pauses in execution. The availability of non-disruptive pauses is extremely crucial to the functioning of a multitasking system where we have a number of interactive tasks going on together simultaneously. However, garbage collection of one process might suspend other processes that are using the same address space. This is the model adopted by many LISP workstations where several LISP processes share a common workspace. Algorithms that have been proposed for interactive garbage collection belong to one of three categories.

1. Reference Counting

2. Copying/Tracing garbage collection

A conservative extension of these algorithms [BW88] can improve execution efficiency. As an example, we will look at an incremental garbage collection algorithm for functional languages [Hal85]. When an expression in a functional language is eagerly evaluated, the computation required to determine its value is started and this in turn sparks off other computations. A future is created for each expression, it is filled with a value when expression evaluation terminates and this value is used by other computations suspended on this information. An incremental garbage collector implementing the Baker-Hewitt algorithm [BH77] appears suitable in this case. All nodes have one of three colors - white, grey or black. We start with a root process colored grey (all other nodes are white) and trace the graph of objects from there. The collector chooses a grey process, colors it black and colors its children black in one indivisible step. We are done when there are no more grey nodes. The remnant white processes are garbage.

We have placed a lot of emphasis on short and predictable pause times. In practice, for many time-critical applications, it is not enough if the pauses are small. In *hard real-time systems*, all tasks that are in the job must be completed by a strict deadline. All operations associated with that task, including garbage collection are bounded by that deadline. The time for which the collector or mutator is halted should be strictly bounded. It is frequently impossible to halt mutator activity for an entire collection cycle. The activities of the mutator(s) and collector(s) need to be interspersed in some consistent manner.

In [WK90], some terms associated with real-time garbage collection have been defined. Some of them are:

Latency of a collector is the maximum time for which the mutator will remain suspended while the collector performs its elementary operations.

A Real-time garbage collection is a collection where the latency is bounded by a constant independent of the number of memory cells in use.

The above definition of a real-time collection has two important implications. Firstly, the collection process must be incremental i.e. the mutator and collector threads must be interleaved. If a mark-and-sweep based algorithm is used, both the marking and compaction phases must be incremental. Secondly, real-time garbage collection must also be concurrent.

The scheme should guarantee that the mutator has memory cells available while the reclamation process has not terminated.

Real-time garbage collection involves some subtle overheads:

- It involves extra synchronization costs to coordinate the actions of the mutator and the collector.
- Real-time algorithms reclaim less memory per collection than the corresponding batch algorithms.
- Pointer assignments involve a test to check if the object has already been moved.
- If garbage collections performed incrementally with marking, already marked objects might be collected while the marking process is still going on. The *floating cells* so generated cannot be reclaimed until the subsequent collection cycle.

Real Time List Processing

In a real time system, the time required by each of the elementary operations is bounded by a constant independent of the number of cells in use. In list processing languages, this is difficult to maintain and garbage collection is complicated by the presence of a number of small structures.

The Minsky-Knuth-Steele-Muller-Wadler method for realtime list GC where there are two processes, the mutator and the collector going on in parallel. The mutator is responsible for list manipulation. The collector has essentially three phases - Mark, Sweep and Relocate. This algorithm is *real-time* because the mutator can continue to execute while the mark and relocate phases proceed. This implies that the allocator cannot run out of free cells at any point of time.

There is no extra space requirement in order to process lists in real time as compared to non real-time algorithms. This is true even for large structures like arrays. Accessing elements is a little more expensive in real-time algorithms as checks for presence (in fromspace) are required. Creation and destructive update operations, however, require no extra time.

Generation Based Real-time Garbage Collection

Few of the proposed generation based algorithms are real-time (incremental) in nature. They do not guarantee that the mutator will never have to wait for an allocation request to be satisfied.

An algorithm that guarantees this has been proposed, described and analysed in [Ben90]. The algorithm has a local copying collector for each generation. But, if the ratio of reachable memory becomes high, a mark-and-sweep based collector is advocated.

Garbage collection takes place in a generation when any allocation is done in that generation. New objects are allocated int he youngest generation. Every allocation performs a small amount of garbage collection proportional to the size of the allocated object. When an object has survived a fixed number of collection in a generation, it is promoted to the next generation. This will in turn initiate allocation and garbage collection in that generation. Thus, it is possible that collection on all generations is being carried out simultaneously. Each generation has an *entry table* that holds references from older generations to that generation.

Real-Time Garbage Collection of Actors

Actors are dynamic units of computation and an actor can create other actors and create and delete acquaintances on the fly. In order to satisfy the real-time guarantee that all computation must be completed in bounded time, the collector must ignore any actors or acquaintances created or deleted after the start of a collection phase. The garbage collection algorithm must therefore work on a *frozen system* to ensure that the system operates on a consistent state.

[WK90] gives a detailed discussion of the issues involved in the real-time garbage collection of actors and presents a variant of Baker's algorithm to collect actors in real-time. The algorithm uses *epoch-numbers* or timestamps. Every garbage collection is associated with a fixed epoch-number. Each actor and acquaintance also carries along with its' creation time as an epoch number. During the process of collection, the algorithm marks only those objects whose epoch-number is less than the epoch-number of the collection. As in Baker's algorithm, only a fixed number of objects are copied from to-space to from-space on every memory operation.

2.3.3 Conservative Pointer Finding

In Conservative Pointer Finding [BW88], we treat every word of memory as if it might be a pointer. In most generational/semi-space schemes, every word of memory is tagged as being either a pointer or non-pointer. This is not a particularly reasonable assumption to make , as we are not always aware of this information, especially in languages like C. This implies that we have less than exact information and as a consequence, we may need to pay for this inaccuracy.

Generational Schemes are more efficient both in terms of space and time - because they are compacting by nature and achieve high performance. But, these schemes are highly language (and implementation) dependent. A positive feature of the conservative pointer finding scheme is that is it language independent. Most garbage collection techniques are contingent on the language or framework in which they are implemented and cannot be applied universally, a result of assumptions made about the behavior of pointers. A conservative scheme makes no such implications. This scheme, however precludes copying as it would be potentially disastrous to place a forwarding pointer into a location that had , say, an integer stored in it.

A novel approach suggested is to combine the two schemes [DWH+90] and this

1. provides a generalized theoretical framework for describing and reasoning about garbage collectors in general

2. paves the way for new implementations and combinations of garbage collection algorithms,

The new theory of garbage collection is based on notion of a partial collection (which retains a proper superset of the perfect collection). In other words, a partial collection recognizes and isolates garbage, but does not identify all the garbage. We can recognize the utility of a partial collection in two situations -

1. When we do not have the exact "points-to" relationship among memory cells

2. When all we want is to identify a subset of memory objects richer in collectible objects - i.e. we do not want to pay the overhead of traversing long-lived objects during each collection.

In the latter case, it is interesting to observe that we now have a generational scheme which is non-copying. Thus, we lose the compaction that we obtained for free in generational schemes.

2.3.4 Fault Tolerant Garbage Collection

Most distributed GC algorithms make very strong assumptions about the reliability of the computational system. They assume fail-proof nodes, reliable message delivery and atomic transactions. A mechanism for GC of passive objects which relaxes the contraints mentioned above is proposed in [SGP90]. The garbage detection protocol described relies only on local

information; effectively the only form of GC is local GC. The protocol records conservative approximations of outgoing and incoming references in object reference tables. The fault tolerant garbage detection protocol assumes fail-stop crashes; it allows for message loss but not message corruption. Traditional GC algorithms exhibit a varying degree of fault tolerance. Reference counting algorithms are not resilient to faults, a strong notion of equality must be maintained during the execution of a task. Tracing algorithms like mark-and-sweep and copying collectors, however, are more robust because accessibility is determined independently for every collection. Algorithms for garbage collection in the presence of faults have been discussed in [LL86, Ves87].

This summarizes the state of the art in parallel GC and ongoing research projects. We must note that many of the schemes described above are designed for a particular architecture or programming paradigm. While designing a memory management scheme for a highly parallel system, we must abstract away from such details. In [VMA91], we have discussed the inadequacy of existing models of computation for designing massively parallel systems. In the next chapter, we look into the need for a universal model of concurrency, compare three generalized models and propose a hierarchical view of concurrency.

Chapter 3

Generalized Parallel Models

In order to design scalable, parallel systems, we find the need for a generalized model of concurrency that will encompass all parameters of parallel computation, be it programming or architectural. We explain the utility of such a universal model of parallel computation in this chapter and analyze three models of concurrent computing – the *Actor* model, the *Bulk Synchronous Parallel Computer* and the *Parallel Machine Interface* in the following three sections. Finally, we propose a hierarchical view to parallel computation.

3.1 Universal Models Of Concurrency

One can view a parallel computation model as a set of abstractions that capture the semantics and functionality of concurrent program execution. This problem has been approached from different perspectives and various models of concurrency have been proposed by language theorists, complexity analysts and architects of parallel machines. We will analyze some existing models with a view towards determining their suitability as a general purpose parallel programming model which is scalable. Some of these models are Actors[Agh90], the Bulk Synchronous Parallel(BSP) model [Val90] and the Parallel Machine Interface (PMI) [DW89]. A universal model facilitates the development of various applications without any concern for the underlying architectural configuration. It is also possible to insulate hardware and software developments from each other with a universal model. Thus, it is possible to derive the benefits of advances in hardware(software) without having to be overly concerned about equivalent advances in soft-

ware(hardware). Interestingly, the proposed models have similar underlying features. We will discuss the following classes of generalized models:

A Programming Model: *Actors* is a model of concurrent object oriented programming with active agents, put forth by language theorists.

A Complexity Model: The *Bulk Synchronous Parallel(BSP)* model proposed by Valiant is an intermediate model to bridge the gap between concurrent programming languages and parallel architectures. It deals with complexity issues involved in *efficient universality* and is a useful model for designers of parallel algorithms.

An Architectural Model: The Parallel Machine Interface studied by Bill Dally and his group at MIT is an implementation oriented machine model. The PMI aims at achieving efficient hardware implementations of primitive abstractions.

It is possible to measure the cost of implementing different operations in various programming models on the basis of the primitive operations on the each of these generalized models.

The Actor Model – centered around computational agents which are active, distributed in time and space provides a good abstraction for discussing concurrent computation in distributed systems and will be explained in the following section.

3.2 The Actor Model

The *actor model*, first proposed by Carl Hewitt [Hew77] and later developed by Agha [Agh86], captures the essence of concurrent computation in distributed systems at an abstract level. In the actor paradigm, the universe contains computational agents called *actors*, which are distributed in time and space. Each actor has a conceptual location (its *mail address*) and a *behavior* as illustrated in Figure 3.1.

The actor is itself a basic unit of encapsulation and the notion of a memory unit in the actor model is the memory allocated to store the local state of an actor and its behavior. Memory is allocated whenever an actor is created and is deallocated when it is determined to be inaccessible.

The only way one actor can influence the actions of another actor is to send the latter a communication. An actor can send another actor (or itself) a communication only if it knows

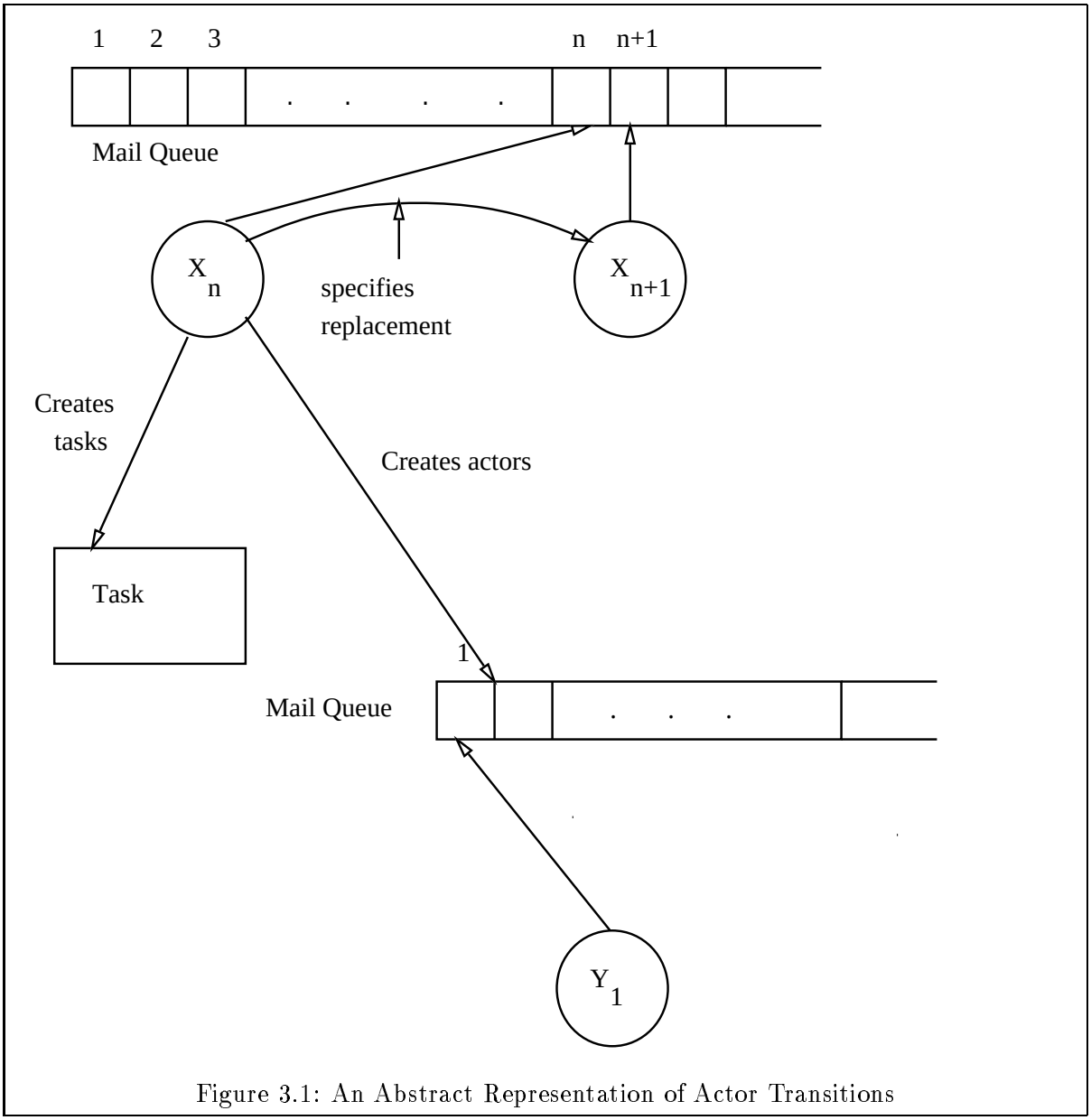


Figure 3.1: An Abstract Representation of Actor Transitions

the mail address of the recipient. On receiving a communication, an actor processes the message and as a result may cause one or more of the following events:

- Creation of a new actor
- Change its behavior
- Send a message to an existing actor

In asynchronous communication, the sender and receiver do not need to coordinate message delivery. If two messages sent to an actor arrive at their destination simultaneously, there must be some mechanism to serialize the incoming communications and execute both messages. To ensure this, every actor is equipped with a mailbox that queues any communications received. Therefore, communication in the actor model is *asynchronous* and *buffered*. The buffers may be bounded in size or infinite. Although Actors is inherently an asynchronous model, it is possible to simulate synchronous models as specializations of the actor model. A important characteristic of communication in the actor model is the ability to communicate mail addresses. Thus the interconnection topology of the system is capable of changing continuously. This adds to the reconfigurability and flexibility of the actor model; for example, it allows resource management decisions such as object to processor mapping to be directly programmed. Another property of the actor model is *Guarantee of Delivery*, i.e., messages in the system will eventually reach their destination actor. This implies that in-transit mail cannot be indefinitely buffered.

Actor execution is graphically expressed in terms of event-diagrams (See Figure 3.2). An event-diagram is a pictorial representation of the arrival orders of events within a thread of execution and the causal relationships between different threads of computation. The thick vertical line in Figure 3.2 represents the actor on a linear temporal scale with time floating from the top of the line to the bottom. Events that occurred earlier lie above those that occur later. Event diagrams bring out the concept of local time and local state in the actor model. In fact, it is believed that the notion of global time is itself quite constraining [Cli81]. Causality connections form the fundamental synchronization mechanism in the Actor model.

Arrival ordering imposes a notion of local time while activation ordering indicates communication patterns and causality among events. The combined ordering includes all possible dependencies.

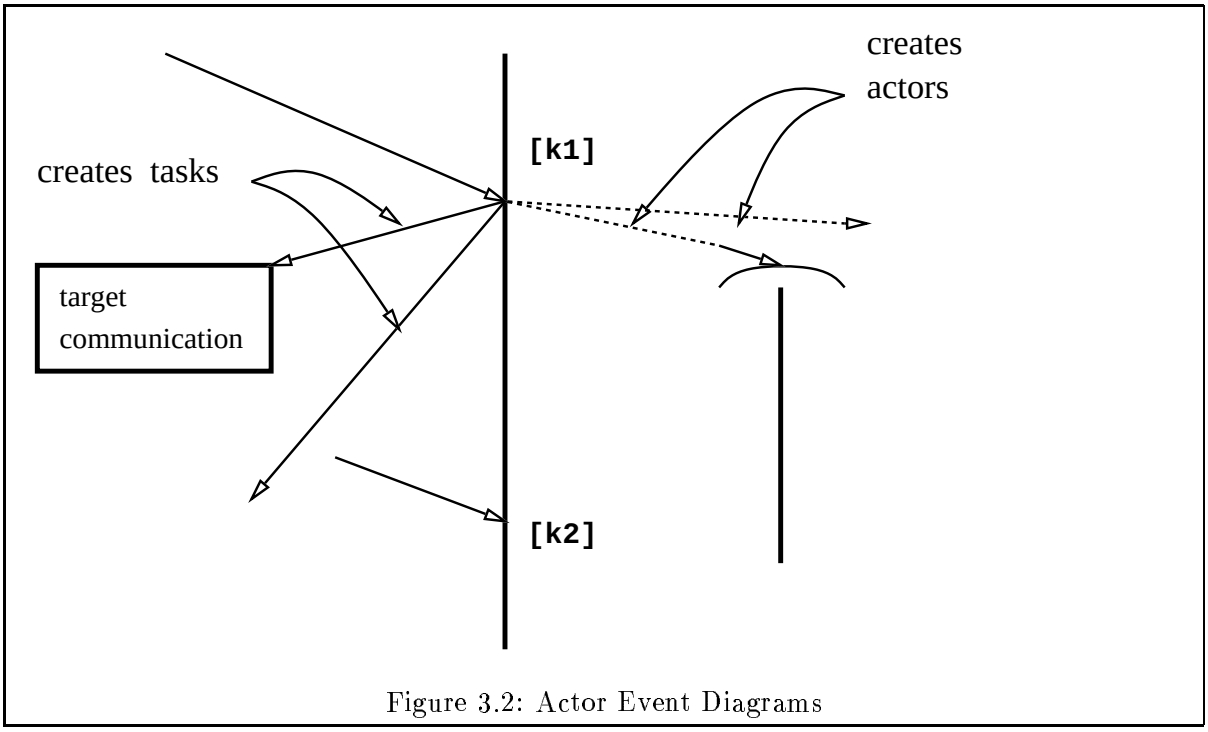


Figure 3.2: Actor Event Diagrams

The actor model is well suited for fine grained computation because of its dynamicity and its ability to create new actors inexpensively. It may sometimes be necessary to sequentialize task execution using one of the following methods:

1. **Introduce causality constraints** between different threads of execution. This does not reduce the grainsize of a task, but causes sequentiality that allows controlled resource allocation

2. **execute a given thread of control sequentially** instead of in a functional fashion. Either user defined or compiler derived annotations could be used to specify the sequentiality. For example, we could have both sequential and parallel versions of a recursive method. The parallel version creates new tasks for every recursive call. The sequential version is called when the argument to the recursive call falls below a certain value. The sequential method definition executes to completion without creating new tasks and executing its body sequentially.

Baude and Vidal-Naquet have showed that the asynchronous, message passing actor model is as powerful as the traditional PRAM model (used to analyze the complexity of parallel algorithms) which is a synchronous, shared memory model [BVN91]. In the PRAM model, program

execution is deterministic and can utilize an unbounded number of logical processing units. The logical processing units communicate via a global shared memory. Efficient parallel programs are defined as those programs that demonstrate an exponential speedup with a polynomial increase in the number of processing units. Such *efficiently parallelizable* programs are classified under the NC class of programs.

Efficiently actor parallelizable problems or NC_{actor} are defined as those problems for which there exists an actor program whose time complexity (the length of the execution chain created by the actor) is a polylogarithmic function of the input size and whose size complexity (a measure of the space needed to process the message) is a polynomial function of the input size.

The following result follows from the above discussion.

$$NC_{PRAM} \subseteq NC_{actor}$$

The simulation of Actors by PRAMS and vice-versa has also been demonstrated [BVN91].

In summary, the actor model is a convenient perspective for the programmer of the parallel machine due to its flexibility and simplicity (programming with abstractions). However, it does not model architectural issues; for instance, the overheads involved in message passing.

3.3 The Bulk Synchronous Parallel (BSP) Model

Before discussing implementation issues in parallel computation, we need to consider the complexity of parallel algorithms and programs.

Valiant proposed the Bulk Synchronous Parallel (BSP) Model as a suitable bridging gap between parallel languages and architectures.

The BSP model consists of three units:

- Components – which are computation or memory units
- Router – responsible for point to point message passing
- Periodic Synchronization – that performs synchronization of all or a subset of the processors with a periodicity L.

To achieve *efficient universality* results, we must be able to model the performance of the system as a relation. If h is the number of messages sent or received in a computation on a

given processor and g is the sum of all computations in the system per second divided by the number of data words delivered per second, it is assumed that all communications are delivered in time gh. Let the startup latency or cost be s. The router sends and is sent at most h messages in a superstep. Such supersteps are also called *h-relations*. Therefore, the cost to realize a h-relation is $gh + s$. This technique of modeling the performance of a system as a relation offers parameterized controllability and is one of the basic features of the BSP model that makes it amenable to complexity analysis.

Memory components in the BSP model are distributed across the processing components and therefore access to every computational unit is equally likely. To allow efficient symbolic to physical address mapping, the BSP model adopts a pseudo-random mapping or hashing mechanism. The assumption is that known hash functions are used and they can be computed locally and inexpensively.

Synchronization in the BSP model is carried out periodically using *bulk synchronization*. The process of bulk synchronization, from which the model derives its name, is as follows. Initially, the mean delay to execute each operation is roughly estimated. After the estimated time has elapsed, an elected processor sends out a sync detect signal to all the other processes along a spanning tree. If synchronization is achieved, we can proceed with the next phase of computation. Otherwise, a new timeslot is allocated and the process is repeated till synchronization is achieved.

In the BSP model, parallelism is extracted by the compiler. The tasks generated are executed in a sequential fashion but all the tasks generated can execute in parallel. The model is general enough to allow both SIMD and MIMD parallelism. The tasks involved in a bulk synchronization must wait until the synchronization has been achieved for further continuation of the executing task. However, it is possible for processing components to switch this mechanism off and execute without waiting for synchronization. As a consequence, task granularity in the BSP model is controllable by varying the periodicity of synchronization, L. As L increases, the granularity of the program also increases. The type of algorithms most naturally expressed using the BSP model are PRAM programs.

The BSP is a possible model for the designer of a parallel algorithm. Valiant demonstrates how the BSP model can be embedded on theoretical models like PRAMs as well as architectural models like networks of systems. However, the level of abstraction adopted makes it difficult to

perform optimizations that may be necessary for efficiency – specifically, no language interface or architectural issues are dealt with. The model assumes the feasibility of efficient bulk synchronization and thereby restricts communication costs. This feature will need to be addressed if the BSP model is used to design a realistically scalable system.

We have proceeded from the domain of abstractions (the Actor model) to the domain of complexity (the BSP model). The Parallel Machine Interface (PMI) presents a more architecturally focused view of concurrent computation.

3.4 The Parallel Machine Interface

In [DW89], a generalized machine model has been proposed to support various parallel models of computation, the *Parallel Machine Interface (PMI)*. Any machine model, sequential or parallel, must have abstractions representing hardware components, i.e., memory, instructions and instruction sequencing. One such general-purpose abstraction in a sequential machine that has been very successful is the notion of stack-based storage allocation. Complex modeling primitives lack flexibility, generality and are less amenable to optimizations and therefore, it is important to keep the modeling primitives simple and straightforward.

What are the kinds of abstractions needed to support a parallel model of computation? Three basic requirements of any parallel model are *communication, synchronization and naming*. Most proposed models of parallel computation like dataflow [AC86], static message passing, shared memory, parallel logic programming [GMN+88] and concurrent object oriented programming improvise on similar implementations these primitive abstractions. The goal is therefore to design efficient hardware implementations of these primitive mechanisms that are portable across different programming systems.

A parallel machine interface isolates the issues of programming models from the details of machine organization and implementation. This abstract model of parallel systems is then embellished with special features in order to support a particular programming paradigm efficiently on a specific target architecture. A parallel architecture interface, called Pi has been proposed based on the PMI [Wil90] and the implementation of several machine models has been illustrated. An abstract machine architecture has also been proposed for the same and we will use this abstract architecture implied by Pi in the discussion below.

The PMI views storage as a collection of data units or *segments* which are logically related. It makes no assumptions about how segment names are interpreted in an implementation. However, some translation mechanism must be built to support segment addressing and access. The cost of accessing a segment is directly dependent on the translation scheme chosen.

The PMI currently assumes a message passing model of communication, but mechanisms like communication via shared memory or RPC (remote procedure calls) can also be implemented. The communication network does not take a stand on routing and buffering mechanisms. As in the Actor model, the PMI assumes that all messages are eventually delivered (albeit an arbitrary delay due to network latency) and that there is no message order preservation. The network is capable of preventing message injection into the system (blocking processing units from injecting messages into the network).

The three primitive operations in a parallel system have been modeled as follows:

1. **Communication** is represented by means of a message send. Shared memory reads and writes are more complex mechanisms implemented in terms of message sends.

2. **Synchronization** is of two kinds. Data synchronization which is needed when there is a data dependence between executing tasks and control synchronization, which requires that a task must complete before another can begin execution. Annotations on memory segments serve as synchronization state indicators. A combination of the annotations and message passing access is used to demonstrate the feasibility of several kinds of synchronization like data, barrier and producer-consumer synchronization. The mechanism used to implement safety and correctness in synchronization is *actor firing* and progress after synchronization is guaranteed by *I-structure accesses* [ANP87].

3. **Naming** issues in most models can be implemented as some form of translation from a logical name to a physical address in the memory of the system. The model assumes that the translation is embedded within the message injection and reception mechanisms.

The Pi model defines primitives to exploit spatial locality that provide information regarding the relative proximity of objects to a particular node. Task allocation (memory required for a task execution) in the Pi model is allocated when a message is received. Memory requirements that cannot be determined statically is allocated as additional segments during runtime. A

	BSP	*Actors*	*PMI*
Storage Model	memory and components	Actors as Abstract Data Types (ADT's)	Logical collection of data units called *segments*
Communication	asynchronous point to point	asynchronous point to point	asynchronous point to point
Synchronization	Bulk Synchronization	causality and history-sensitivity	access attributes

Table 3.1: Universal Models of Parallel Computation

task is executed until termination unless a higher priority task is scheduled for execution or a synchronization is needed. Multiple task queues are used for task prioritization.

The abstract machine implied by the Pi model is fine-grained and consists of a set of nodes interconnected by means of a communication network. However, the Pi model can also support large-grained computation. Fine-grained systems are *upward-compatible* with larger grained systems. In other words, architectures for fine grained computation can support larger grainsizes efficiently but not vice versa. With this model, it is possible to define operations to measure the cost of implementing the different operations in various programming models on the basis of the primitive operations on the PMI. In fact, it has been observed that the costs of the PMI model the actual machine costs very well.

Tables 3.1 and 3.2 summarize the three different universal models we have considered. As is evident from all the models, some attributes of parallel computation are mandatory to any model irrespective of whether it has an analytical or architectural flavor, i.e, memory, communication and synchronization. However, the degree of specification with respect to other attributes depends largely on the level of abstraction that the model is trying to achieve.

3.5 A Hierarchical View of Parallel Computation

Good scalability characteristics are achieved only if the problem size scales up with the number of processors. In other words, merely increasing the number of processors keeping the problem size constant is expensive. Furthermore, scalability and complexity analysis on these

	BSP	*Actors*	*PMI*
Naming	Pseudo-random mapping symbolic names to physical addresses	Unique actor names	Nametable for address translation
Process/Object Creation	unclear	dynamic creation	dynamic creation
Expressing Parallelism	Sequential tasks specified	Parallelism default implicit synchronization	fine-grained threads explicit synchronization
Locality	Hash function distributes memory components	Actors are a unit of locality	Primitives to support spatial locality
Process Topology	unclear	dynamic configuration	dynamic configuration
Atomicity of Primitive Operations	depends on periodicity of synchronization	Handled by programmer defined constructs	Task execution is non-preemptive modulo prioritization
Granularity	Controlled by varying the periodicity of synchronization	Flexible grainsize	Variable granularity
Nondeterminism	No claims	Arrival order nondeterminism	Not addressed
Fairness	Not addressed	Weak Fairness due to guarantee of mail delivery	Not addressed
Migration	Not addressed	Not addressed	Mapping tables handle address translation
Abstraction Level	Algorithmic	Programming Language	Architectural

Table 3.2: Universal Models of Parallel Computation continued

models must also consider the implications of the algorithm and its mapping on the underlying architecture [SKAT91]. A *hierarchical view* of a parallel machine is shown in Figure 3.3.

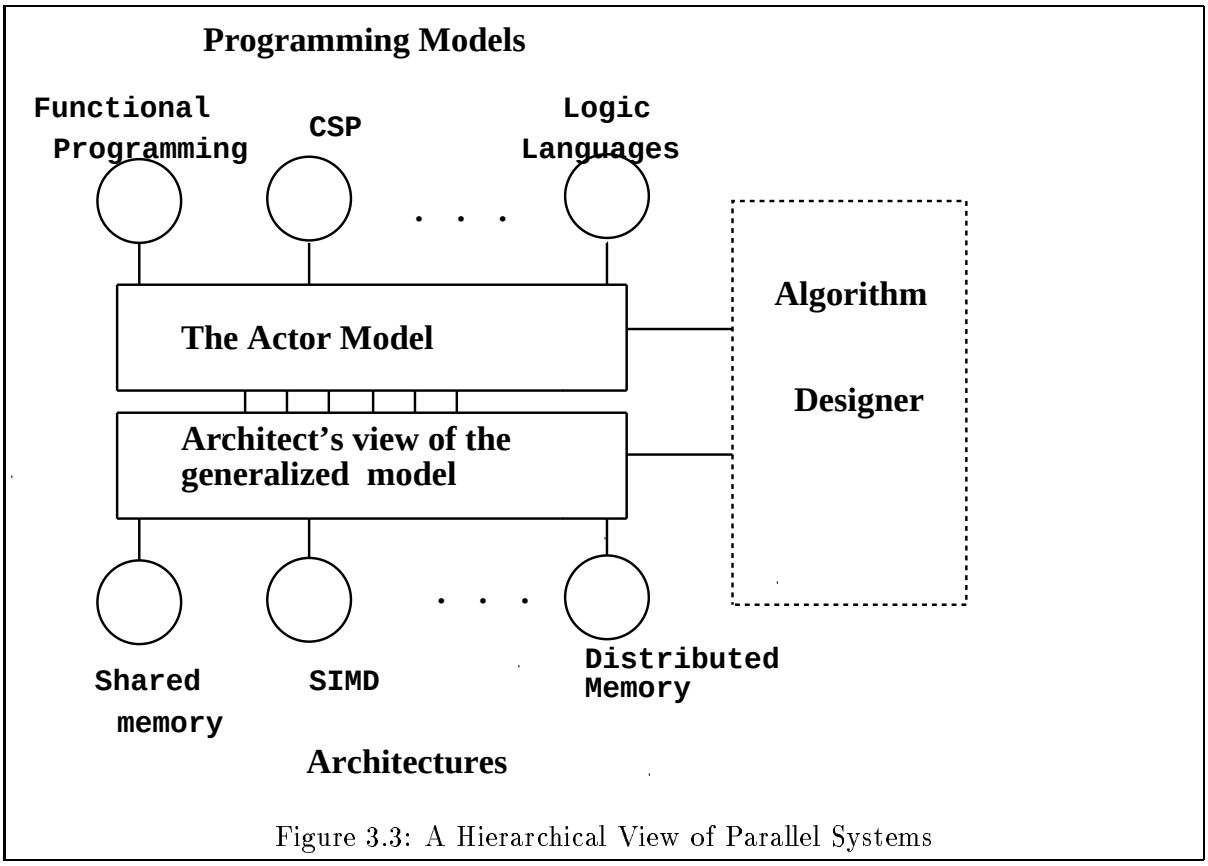


Figure 3.3: A Hierarchical View of Parallel Systems

The purpose of the hierarchical model is to provide layers of abstraction in a parallel system. When dealing with high level issues like algorithm design or programming language issues, we would like to abstract away from implementation or architectural details like the communication subsystem or resource management. However, in order to discuss resource management issues without referring to any particular programming language or architecture, we must isolate features specific to parallelism from issues specific to the framework in which parallelism is being exploited. The hierarchical view in Figure 3.3 illustrates the different levels at which a parallel system can be viewed for different purposes.

The hierarchy has three levels. They are:

Parallel Programmer's View: Uses the Actor model as a basis, with distributed agents that communicate via messages.

Systems Software Designer: Uses a bridging model that maps the conceptual agents on to a parallel system. This is where the operating system schemas sit. The hierarchical resource management capability is built into this layer.

Architect/Hardware Designer: Uses the PMI model that incorporates a hardware abstraction of message passing and I-structure like synchronization. Local resource management schemes (context switching, local garbage collection and memory allocation) are implemented at this level.

We consider storage management in parallel machines with the Actor Model as the programmer's view of the world and the Parallel Machine Interface as the underlying model of a highly concurrent machine. Using the hierarchical view, the user/designer of a parallel machine need only address problems that are specified at his or her level of interest.

Chapter 4

Issues in Parallel and Distributed Machines

In this chapter, we will address issues that are of concern to the designer of the resource management system for a highly parallel machine. We will recall the goals of storage management which we had established in an earlier chapter and reconsider them in the presence of various additional factors like naming criteria, load balancing and locality concerns, to name a few. After explaining the tradeoffs involved in handling these problems, we will reformulate the goals of a distributed storage management scheme. The runtime system could be assisted with annotations from the user or compiler and we will discuss a few annotations for memory management in the final section.

In Chapter 2, we formulated the goals of a storage reclamation algorithm. They are:

1. Short and predictable response times
2. Minimal user interaction with memory internals.
3. Require minimal hardware support.
4. Compatibility with memory hierarchy hardware, i.e., cache and virtual memory.
5. Causes only a small impact on execution overhead.
6. Capability to reclaim circular structures.

We have explained why automatic storage management is an important feature of memory management for parallel machines. Many of the tasks involved in memory management are delegated to the runtime system. The object based storage model we will develop assumes the following operating systems support mechanisms :

1. A mechanism that can allocate and deallocate contiguous blocks of memory.
2. Abstractions needed to access objects in memory with a single virtual address across a distributed object namespace. This facility is crucial to support object migration and easy inter-processor accessing of objects.
3. A network transparent runtime environment.
4. Effective support for relocation of objects either within the local heap or across the network. The migration of a process should interfere as little as possible with the execution of the process and with the functioning of the system as a whole. As in the Distributed V System [Che88] designed at Stanford University, a migrated program should have no residual dependencies. A residual dependency is said to exist from node a to node b if the execution of a requires the execution of b residing on a separate node [Che88].

4.1 Storage Management Issues in Parallel Machines

In the introduction (see Chapter 1), we spelled out the complexities involved in resource management for scalable, parallel systems. We will now look at some specific design issues and their implications on the memory management scheme. Specifically, we will deal with the following issues

- Distributed Naming
- Locality
- Load Balancing and Object Mobility
- Network Activity
- Exploitation of Concurrency

4.1.1 A Shared Distributed Namespace

When we have a single address space across all nodes in a network, all (entities) in the system are referred to by an address that is uniform over the entire system. This address remains unchanged even if the memory cells reside in the local memory of physically remote processing units. Such an address space is called a *distributed shared address space* and is the basis of the recent NUMA (Non uniform memory access) architectures [LLG+89]. A system which has a shared address space distributed across the local memories of distinct processing units is said to have a *distributed shared memory* organization.

With objects as the basic units of memory management, every object is known throughout the system by a unique *object name*. The object name of an object remains unchanged irrespective of the current location of the object in the system, giving us a uniform, *distributed namespace* over the entire system. A distributed shared namespace is desirable, primarily for scalability reasons. Adding new objects to memory or a new processing unit with its objects will not require a total redesign of the object naming and management scheme. A shared namespace also brings us additional flexibility in resource handling. It allows us to migrate objects to support locality, load balancing or garbage collection. The virtual address or name of an object has to be mapped to a corresponding physical location (processor number and offset in the local memory) in order to reference the object. Name translation is achieved through a translation table called an *object nametable*, much like the page tables in a virtual memory system. The object nametable is the crosspoint of the virtual naming - physical addressing relationship.

Interprocessor communication in a distributed system can be modeled as a series of message sends. These messages are routed by means of a fast, efficient routing network to their destinations. Local communication is accomplished through primitive messages, i.e., messages that an object sends to itself. The concept of a distributed shared namespace lends itself well to the exploitation of concurrency and supports the idea of a uniform access mechanism. It also extends our virtual address space across all nodes in the network with a uniform naming convention[1].

The cost of the flexibility available with a distributed namespace is the overhead associated with name translation. Each access to the object must query the nametable and it is therefore

[1] Name generation or unique ID generation is not a hard problem to solve in itself, and we shall not get into the intricacies of those policies here.

vital that all translations be completed with a relatively low latency. A hardware cache of frequently accessed nametable objects can be implemented, but we have to ensure a fairly high hit rate to achieve good performance. In order to expedite object access, a node may maintain a hint about where an object currently resides [Tot87], which in turn may maintain another hint and so on. But, a limit must be imposed on the number of indirections allowed. The nametable itself must be decentralized or it could become a bottleneck and degrade system performance. In a prototype architecture, the J-Machine [DW89], the nametable is distributed across the nodes of the system to prevent centralization, thereby reducing contentions to access this nametable. Furthermore, memory is a finite resource and as the nametable must contain an entry of every object in the system it could potentially consume excessive memory.

Efficient nametable management and quick translation schemes are currently hot research topics. Our belief is that we should avoid translation for objects that reside locally and as far as possible for remote objects through intelligent compilation techniques and optimizations. In the next chapter, we will discuss naming schemes in a distributed system in more detail.

4.1.2 Locality

A task is said to exhibit *locality* when frequently communicating processes are placed on the same node (processing unit) in a multiprocessor system. Frequent communication between processes on different processors increases network contentions. Minimizing network traffic is important even in the presence of high speed networks, where the network bandwidth is the major limiting constraint on communication. Locality directed scheduling policies reduce interprocessor communication, thereby decreasing network traffic.

Memory locality is a property that deals with the pattern of a program's references to memory. References that are tightly grouped together in terms of addresses are said to have spatial locality. References close together in time are said to have temporal locality. Memory locality is a concern for both distributed systems as well as non-distributed systems with virtual memory. The implications of memory locality can be seen at all levels in the memory hierarchy like the cache and virtual memory. The effectiveness of caching objects is determined by the degree of locality in the program. This is because poor locality in the cache causes frequent cache misses and new objects have to be paged into the cache. This increases not only the information transfer overhead due to page replacements but also the overhead needed to maintain cache

coherence. Hence the cache miss rate is an important measure of the temporal locality of a program. Poor locality of reference in virtual memory systems introduces unnecessary waits on page faults. By using an adaptive memory management algorithm [Cou88], page-wait times can be reduced by a significant factor. In traditional systems, task (process) locality and data locality have to be dealt with separately. However, when dealing with objects, the process and the memory it manipulates are encapsulated into a single unit and we have to only deal with object locality.

Garbage collection algorithms strongly influence the degree of locality of a program. Zorn presents an attempt to quantify this influence in Lisp programs by using a technique called stack simulation [Zor89]. He also draws a distinction between *program locality* and *collector locality*. Program locality is the locality inherent in the user program and collector locality is the reference locality during garbage collection. Note that interactive GC algorithms exhibit poor collector locality.

Three distinct classes of locality have been identified in user programs:

1. Temporary objects that are usually intermediates during computation.

2. Fairly long lived objects whose lifetime is a significant portion of the program's lifetime.

3. Permanent objects like the runtime system's routines and structures

A well tuned generational collector will place the latter two classes in distinct higher generations. A large fraction of references to the newest generation are initialization operations. References to the older generations are largely load accesses. Techniques to avoid fetching the original contents (only to overwrite it) in such a case are described in [PS89]. If compaction is carried out during promotion, older generations exhibit better locality. As the collection algorithms have to access the accessible locations, they exhibit poor reference locality. The order of traversal, i.e. depth or breadth first affects the degree of locality to a large extent. Better locality is exploited with depth-first traversal, because, the depth first order exploits object relationships in a natural manner. An object is directly related to its descendants, but not necessarily to a breadth first neighbor. If the collection algorithm is incremental, collector and program references are interspersed resulting in poor locality. All this suggests that mark and sweep algorithms exhibit fairly good collector locality and simulation studies substantiate this.

In a highly parallel fine grained system, programs must exploit locality in order to reduce the communication bandwidth requirement. Computation in highly parallel message passing systems is data-oriented [Chi90] and data placement strategies critically affect system performance. We shall now discuss three techniques to improve locality in highly parallel machines; programmer defined, static and dynamic object grouping.

Programmer based object grouping: Object Oriented languages designed for distributed processing like Emerald [Hut87] have primitives that allow us to place tasks that communicate frequently on the same node. In the last section of this chapter, we will discuss annotations that may be placed either by the programmer or the compiler to assist in object placement and locality.

Static object grouping: Static techniques for grouping objects to improve locality characteristics involves compiler directed analysis of the program text to generate clusters of objects that must be scheduled on the same node. The compiler has a better view of the global information required to make placement decisions and therefore, compiler based data partitioning is reliable. Compiler based locality can also utilize information about the configuration of the target architecture and perform useful optimizations to increase performance. However, it is not possible for the compiler to extract object interactions intended by the programmer in all instances. Conservative assumptions need to be made during compile time analysis due to the absence of runtime information and this may lead to scheduling patterns that could be far from optimal. Also, static grouping of objects is not sufficient to eliminate the paging problems of large persistent structures in object oriented programming.

Dynamic object grouping: Dynamic grouping of objects in virtual space in a Smalltalk system [WWH87] attempt to keep related objects on the same page.

4.1.3 Load Balancing

Load balancing is the task of keeping the processors of a parallel machine uniformly busy.

For effective load balancing, each processor needs to know the degree of load in every other processor, or at least, some controller needs to maintain load information in order to make load balancing decisions. In the latter case, there is likely to be a contention for this centralized

resource (i.e. the controller). If there is a static interconnection scheme between processors in a system, each processor need only maintain nearest neighbor information.

Locality and load-balancing are issues that are in direct confrontation with each other [Ath87]. Locality aims at placing logically related objects in close physical proximity, thereby mitigating the frequent communication costs. Load balancing efforts are geared towards splitting and distributing work and do not encourage "clubbing" of computational nodes or threads. In order to ensure satisfactory progress, the above two conflicting factors must work in harmony to yield just the right amount of locality needed to mask the communication latency. The degree of object mobility required to achieve this balance and mechanisms to deal with this have been explored in [Jul89]. Emerald is an object oriented language that integrates fine grained mobility of objects. Apart from load-balancing, object mobility benefits the performance and utilization of the communication network, facilitates system reconfigurability and increases the degree of availability in the network. Fine-grained mobility makes distributed garbage collection easier by moving objects to nodes where their references exist. The introduction of language primitives for mobility necessitates a great deal of cooperation between the compiler and the run-time system.

4.1.4 Network Activity

One of the goals of network behavior for a system is to keep the links of the network uniformly busy with active data objects. Non-uniform distribution of objects may result in bottlenecks at frequently accessed nodes, thereby delaying other communications. To achieve this, message dispatch and reception rates of the producer and consumer of the communication be compatible. Consider the scenario where we have a producer-consumer relationship between objects. In order to exploit concurrency, we assign them to different nodes. By placing the nodes adjacent to each other in a mesh, we have a single link between them and this may result in non-uniform traffic in the network. This suggests that we place the two objects on nodes that are far apart. But, we must keep in mind that message reception at the consumer is serialized and placing the nodes far apart in the network would block the network which is a shared resource in the system. We therefore, now have a contradictory argument that presses for locality. Thus, locality factors play an important role in network behavior in a parallel system.

We will now see how garbage collection in a distributed system can influence the degree of network traffic. In a distributed system, we are forced to exploit locality in all tasks, including garbage collection. Thus, most distributed systems have both local and global garbage collections. Local collection does not give rise to network traffic. Global garbage collection is communication intensive as each traversal may involve a message send. Communication is also necessary to maintain interprocessor references and for synchronization. This may result in an unacceptable amount of network traffic. The pros and cons of local and global garbage collection in a distributed system have been dealt with in [KT90].

4.1.5 Exploitation of Concurrency

By distributing tasks across the nodes of the machine we can expose the concurrency present in a program. Programs can turn out to be *embarassingly* parallel and this can reduce performance or even more critically, result in a deadlock. Object Oriented programming allows us to manipulate the level at which abstractions are specified to suit the requirements of the runtime system. With the introduction of programming paradigms where parallelism is inherent, we can think about GC algorithms where the actions of the mutator and collector can proceed concurrently. Furthermore, the mutator(s) and collector threads themselves possess some amount of concurrency [SS91].

In order to make a parallel system commercially attractive it should have the following features – correctness, efficiency and simplicity of use. Now that we have considered the tradeoffs and conflicts that we must deal with in a highly parallel system, we need to rethink our initial aims in designing a memory management subsystem, keeping in mind the necessities of the parallel system as a whole. In the light of the issues discussed above, we will reformulate the goals of storage management for parallel systems and the associated garbage collection mechanism.

4.2 Goals of Distributed Memory Management Revisited

Our storage reclamation scheme aims at satisfying the following requirements:

1. The algorithm must guarantee a good interactive response, i.e., pause times in an interactive environment must be small and predictable.

2. There must be no global garbage collection pause i.e the universe of processors must never halt in order for the algorithm to execute. This decision was made keeping in mind that we are now dealing with massively parallel machines; stopping task execution on all processing units may be impossible in many applications, especially time critical applications. Local stops on each processor are permitted as long as they do not destroy a satisfactory interactive response.

3. Try to retain locality that already exists.

4. We believe that distributing collection efforts both spatially and temporally will enable us to exploit more parallelism during collection as well as guarantee satisfactory (good) interactive response times. Spatial distribution of garbage collection refers to the participation of all processors in the collection. Temporal distribution involves the utilization of idle or less loaded processors to handle collection irrespective of whether they have pending allocation requests or not.

5. Our collector must be able to reclaim self-referential structures.

6. Must support task migration.

7. Ability to support persistent objects.

Our storage reclamation scheme relaxes a few constraints of a proper garbage collection scheme:

1. Our garbage collection scheme is conservative; in other words, the proposed scheme recognizes and isolates garbage, but does not identify all the garbage. We believe that for efficiency reasons, minimizing collection time is more important than ensuring the completeness of the process.

2. Our algorithm is not inherently compacting. Our current efforts are directed towards a "compaction with locality" approach. Very much like dynamic grouping in virtual memory in the Mushroom project .

We intend to model collection times and judge the performance of the garbage collector by using a metric called the *resource consumption metric* [SKAT91]. The isoefficiency metric

[KR87] evaluates a parallel algorithm on the basis of the speedup obtained. This metric is based on the assumption that the entire machine is dedicated to executing this single job and the time taken for completion is then a good enough measure of performance. When more than one problem is to be solved on the machine, we would like to ensure resource utilization by efficiently time-sharing the resources so that even if one of the tasks leaves resources idle, the other task can utilize them.

A parallel processor with p processing units has pT cycles available in time T. The *resource consumption* of a parallel algorithm is the sum total of all machine cycles consumed by the parallel processor. The remaining cycles can be used by other algorithms executing on the same machine. We could model the user task and the storage management process as two independent jobs running on the machine and then model the performance of the collector using the *resource efficiency metric*.

4.3 Annotations for Storage Management

We propose three annotations that serve as hints to the runtime system during storage allocation and reclamation. These annotations are intended to blend with the object tenuring scheme proposed in the next section. They are, however, general enough to be applicable to any language and programming environment. These annotations are hints and the runtime system can ignore them if it chooses to. The annotations themselves may be either user-defined or compiler derived.

1. **aggregate (object1 object2 .. objectn):** indicates that these objects must be allocated contiguously in order to preserve locality of reference. This is especially useful in the storage management framework because objects can be promoted to higher generations and preservation of locality is carried through all generations reducing the fragmentation at every stage.

2. **persistent(object):** Persistent objects are objects that outlive the program that creates them. Typically, the user knows that the object he is creating or manipulating is a persistent object and an annotation to declare object persistence is a useful feature in a language. Declaration of persistence in objects is discussed in Section 5.5.

3. **sticky(obj):** This annotation is used when allocating an object that is long-lived. The decision as to what is long-lived is left to the programmer or compiler. Most large structures (that are not lazily constructed) like arrays are intended to be long-lived and we can avoid the overhead of promoting the large volume of structures through the different generations. As this information is available statically, we allocate the initial portion of the heap to the persistent and sticky objects and divide the remaining heap space among other generations as required.

One might wonder if annotations introduce an additional level of constraints in forcing the user to deal with memory allocation and object scheduling details when the main purpose of object oriented programming is to abstract the user from representational details. We believe that it is reasonable to have user annotations for resource management purposes as long as the user is still looking at the objects from a logical perspective, i.e. he is not aware of any representational details. Annotations do not sacrifice correctness, at worst we would do some extra work and reduce throughput. By requiring the user to group objects that should co-exist on a processor, we are introducing a notion of location into the language. But, from the user's point of view, the information needed is logical in nature and does not include implementation details.

Some open questions related to annotations in object oriented languages are:

1. How much useful information can we derive from a compile time analysis?
2. How often will the programmer be able to use these annotations?
3. What is the cost of a badly placed annotation?
4. What effect does this have on the semantics of the language?

In this chapter, we have made an attempt at bringing out some resource management issues in parallel machines, their interaction with each other and with the memory management subsystem. This is by no means comprehensive, and there are a number of subtle problems that have not been discussed here. But, we believe that many critical issues have been addressed before we begin the task of designing a memory management subsystem for a scalable, parallel machine. This effort has also assisted us in isolating the goals from the non-goals of efficient memory management for parallel machines. The challenges of automatic storage management have propelled a number of efforts in tailoring existing collection algorithms to suit the specifics

of the target architecture or paradigm being supported. This has been reflected in using a particular memory configuration, dictating virtual memory bounds and the like. In Part 2 of the thesis, we will look at non-stop GC where a distributed, shared virtual namespace is a monolithic logical unit.

Part 2: Hierarchical Distributed Memory Management

In Part 2 of this thesis, we look into the major components of an object management strategy; object creation, object naming and object reclamation. While designing mechanisms for performing these tasks, we must retain universality in the object management scheme that will enable us to deliver efficient, scalable and well-managed concurrency.

We hope to deliver efficient universality in the Hierarchical Distributed Garbage Collection (HDGC) scheme in many ways. Firstly, there are no granularity constraints placed on the actors/objects. In fact, Hierarchical Distributed Resource Management (HDRM) techniques, discussed in Chapter 8 allow us to exploit granularity restrictions to our advantage, by allowing us to alter or fine tune resource management strategies for different grainsizes. Secondly, no architectural restrictions are placed on the system, as long as every PU has its physical local memory. This is one of the key requirements for scalability. Thirdly, HDRM can be used for any programming paradigm, static or dynamic. Extensibility does not occur naturally in static systems and our belief is that the power of dynamic reconfiguration must be embedded in a system if we are to have efficient scalability.

Chapter 5

Object Creation and Naming

In this chapter, we will develop the portion of a memory management system that deals with memory allocation. Object creation in the actor model is dynamic. It therefore follows that space for newly created actor/object has to be allocated at runtime. In the next section, we will see how to deal with object creation in scalable systems. When an object is created, it has to be allotted a unique name by which it is referred to throughout the system. To support large systems with various physical storage mechanisms, we require uniform naming strategies for an object irrespective of the form in which it is stored. Naming is an integral part of object creation and the following section explains the problems involved in object naming for distributed systems. In a system that permits migration of objects, a scheme has to be implemented to let the referents of an object know where the object currently resides. If the system allows highly mobile objects, communication overheads become excessive. A tiered naming scheme like the one described in Section 5.3 will be necessary. Support for persistent objects is embedded in our tiered naming scheme and we will talk about persistent objects and their implementation in the final section of this chapter.

5.1 Object Creation

Dynamic object creation is an important requirement for scalability. It means that one can tailor the number of tasks generated based on problem size and resource availability. Some applications grow exponentially at run-time. Resources can be allocated or deallocated as computation evolves. Predicting the resource requirements of a problem as well as the resource

availability at a particular point in time cannot be done statically: Dynamic extensibility is necessary for efficiently implementing scalable concurrent systems. We will assume that an object is created in the local memory of its creating node. However once the object has been created and named, the object is free to move around. Dynamic creation alone is not sufficient to obtain efficient concurrency. The ability to alter or reconfigure the system of computations plays a crucial role in determining the effectiveness of a dynamic resource management scheme. Reconfigurability in actor systems is built into the mail system which allows mail addresses of actors to be communicated.

Shared memory computers have multiple processors that share a global memory. For efficiency reasons, each processor also has a local cache. Large numbers of processors create increased contention for access to the global memory. The contention for shared information increases as the computing resources increase. As a result, shared memory architectures are not scalable [Dal86]. Distributed memory machines are scalable but accessing remote objects causes a large overhead if each processor has its own address space due to constraints on the amount of physical memory accessible on a node. We believe that *distributed shared memory* with universal addressing is a suitable memory configuration for scalable, concurrent machines. We will call this architecture a *distributed shared memory architecture*. Distributed shared memory gives us the scalability of a distributed memory machine as well as the flexibility afforded by a shared address space. Recently, this approach to memory design is being adopted by a number of multiprocessor NUMA (non-uniform memory access) machines where different PUs in the systems have varying access times to different pieces of information in memory [LLG+89].

5.2 Naming in Distributed Systems

A mechanism for identification lies in the core of all distributed systems. Identification mechanisms are required at various levels in the system like programming and architectural design. System entities at these levels are designated individual identities known as variables, addresses, identifiers and so on. What we require is a unified way of referring to any entity in the distributed system, be it an operating system process, a database entry, or a temporary object that has been newly created during processing. Unified identification mechanisms can be used for synchronization, object sharing, error recovery, communication and transaction processing.

Other functionalities that could be incorporated into a naming scheme include persistence and protection.

In conventional multiprocessor design, a virtual memory system supports the abstraction of a single large address space. In the object based memory management scheme we are going to design, however, naming strategies are used to handle a shared object namespace. The tiered translation mechanism is equivalent to the page-table address mapping schemes.

In an object based system, every object or entity in the distributed system has an identity. The object is referred to by a location independent object identifier (OID). We will call such a location independent reference a *name*. A name does not have any spatial attributes in a virtual or physical address space. A *virtual address* is a reference to an object; it has some spatial dependences; for example, virtual addresses are a linear sequence of addresses that are not tied to consecutive memory locations. A physical address is a location dependent reference and is usually an offset value into the physical main memory. The region in memory where a name or address is valid (scope of the name or address) is called a *namespace* or *address-space*, respectively. In a large system, the number of objects can be very large and therefore this results in a large namespace. In order to ensure that every object in the system has a globally unique name, OID's must be system generated. System generated names could be composed of different fields that specify properties of the object, e.g., persistence.

Accessing an object occurs by means of a reference to the object. A reference could mean a logical name or a physical address. If the object implied by a name is to be accessed, a mapping table or an associative mapping mechanism is consulted. Given the name of an object, a name-server returns a lower level address which may be another virtual address or the actual physical address of the object. The different levels of address mapping are translation mechanisms between different levels of identification.

Naming strategies for object based decentralized systems can be classified as:

Absolute naming: Here the physical address of the object is the name of the object. The route to access the object is explicitly specified in the name and no translation tables are necessary.

Flat Naming: The name does not maintain any information in its structure. Every routing node in the system must maintain sufficient information to handle all possible addresses.

Large mapping tables are stored on every node. In a system with a large number of objects, this is not feasible. But, as information regarding object location is held in the routing nodes, the object can freely migrate at will. Efficient broadcast facilities will greatly improve the performance of the naming scheme.

Hierarchical or Structured Naming: Typically, a hierarchical name consists of many fields concatenated together. Information is already present in the structure of the name. Thus, less information needs to be maintained at the individual nodes in the system. At every stage in the routing process, the routing algorithm on a routing node can decide the direction in which the message needs to be routed. It is no longer necessary to maintain the path to every accessible node at the router. This decision process is continued until the destination is reached.

Sarkar and Gehringer [SG90] have identified different addressing strategies based on factors like the number of address spaces per system, the number of address mapping tiers, when and how often mapping functions are performed and capability based addressing. Having a uniform (single) address space over the entire system incorporates sharing into the system in a very natural way. We will briefly describe some tradeoffs that arise.

In an object based system, the unit of memory transfer is an object. With fine-grained objects, the average object size is around the size of a nametable entry [SG90]. As the number of individual objects in such systems tend to be large, the size of the mapping tables involved would be potentially huge. Therefore, it would not be cost-effective to have multiple levels of address mapping. Short relative addresses constitute less overhead and are suitable for a smaller address space. Monads is a system where a shared address space accompanied with capability-based addressing has rendered fine-grained object protection [KR89].

In page based systems, however, the unit of memory transfer is a page. In this case, there may be many objects on a page; objects may begin anywhere in a page and may cross page boundaries. Multiple levels (greater than 2) of addressing would be useful in this case.

Object mobility plays an important factor in deciding the naming scheme and the number of levels in a naming hierarchy. The advantages of using object names or identifiers over using the physical address of an object directly are firstly, support for migration and secondly, support for different classes and levels of persistence. This is especially true if the objects are highly

	Virtual Memory Systems	*Object Based Memory Systems*
Fundamental unit of memory management	Pages	Objects
Fundamental unit of memory access	Locations	Objects
Fundamental unit of sharing	Pages	Objects
Mapping virtual names to physical addresses	Page Tables	Object Tables
Migration	Usually not an issue	Object Migration
Replication	Page Replication	Object Replication

Table 5.1: Comparison of Virtual Memory and Object Based Memory Systems

mobile. However, if the objects are relatively stationary, hints about the actual physical address of the object will avoid additional levels of translation and thus reduces overhead.

A comparison of traditional virtual memory based systems vs. object based memory management schemes is illustrated in Table 5.1.

Many open design issues have been identified and the following questions should be considered while designing a suitable naming scheme for a distributed system.

- What should the name reflect?
- How does the targeted size of a system affect the number of stages in the naming hierarchy?
- What is the difference between a shared object namespace and a shared address space? How can one simulate the other? What are the trade-offs and overheads involved?

5.3 The Tiered Naming Scheme for Distributed Machines

The memory model we have adopted is the distributed shared memory model. Mapping object names to physical address may be done either in a single level of translation or by a tiered scheme. A tiered scheme for address mapping translates one form of reference to another.

Address mapping tiers can be used to build virtual memory on top of an existing memory hierarchy. One of the main characteristics of a tiered naming scheme is *location independence* which solves a number of load balancing issues in multiprocessor systems. References to the objects need not be scanned when the object migrates. Tiered address schemes can also provide protection (via capabilities) and selective sharing of address spaces among processes.

In the tiered naming scheme that we develop here, every object is addressed by a globally known *object name* that remains unchanged irrespective of object migration. The object-name is itself system-generated. The birth-node of the object (processor on which the object was created) is encoded into the object-name. Every object in the system has a *virtual address* that is comprised of the processor number on which the object currently resides as well as a relative address which is used to distinguish objects on the same PU. The *physical address* is the actual memory offset on a particular processor where the object resides.

Every object is addressed by a globally known object-name that remains unchanged irrespective of object migration. Object name to virtual address translation is accomplished via a translation table called the *global distributed nametable*. which is distributed across all the processing units in the system. The translation of a virtual address of an object on a processor to an actual physical address is accomplished via a local translation table on a processor, called the *local nametable*.

The format of the variable fields used in a nametable entry is shown in Figure 5.2. We associate a one byte tag field associated with the global nametable entry for every object. The runtime system is responsible for the proper interpretation of these tag bits.

The distributed global nametable has some critical, frequently accessed information encoded into it. This helps us avoid traversing multiple levels of tables to access this information. For example, during the process of generational GC, it is necessary to know if an accessible object is in NewSpace and therefore, its referents may be scavenged. This check occurs for every access made by the garbage collector. If this information is encoded into the nametable instead of in the object, the required information is available during the first name translation stage. The global table entries supply more information than the virtual/physical address of the object given the object name. The object-name is itself system-generated. The birth-node of the object (processor on which the object was created) is encoded into the object-name. The serial number distinguishes objects created by the same processing unit. The virtual address consists

Virtual Name	Tag		Virtual/Physical Address	
	P	GEN	Pid	Address
	0	001	1	1100
	0	101	5	1232
. .		. .		. .
	1	110	356	1590

GLOBAL NAMETABLE

P - Persistance Bit
Gen - G_2 G_1 G_0

Figure 5.1: Structure of the Global Nametable

of two subfields: the PID field specifies the processing unit on which the object is currently situated and the virtual address of the object which is used to distinguish objects on the same processing unit. A local mapping table translates this virtual address to an actual physical address.

In addition to the address fields, the nametable also has a tag field which has been introduced for memory management purposes. The high order tag bit is the *persistence* bit which is (1) if the object is persistent (outlives the lifetime of the program). A persistence bit value of 0 indicates that the object is *volatile*, i.e., liable to become inaccessible during program execution. The lower order three bits in the tag field $G_0G_1G_2$ comprise the *generation* bits. We accomplish the garbage collection by just flipping the generation bits in the global nametable.

5.3.1 Organization of Global Memory

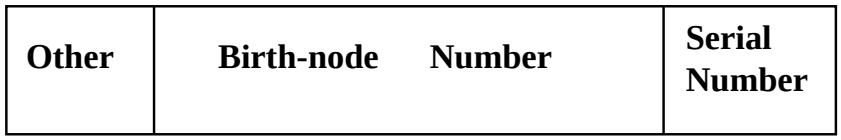
Before describing the meaning of the different bits, we explain the organization of the logical heap (See Figure 5.3):

1. The first section of the heap is used to store objects which have been annotated as *persistent* (objects that outlive a program). This information may be obtained either through annotations specified by the programmer or derived at compile time.

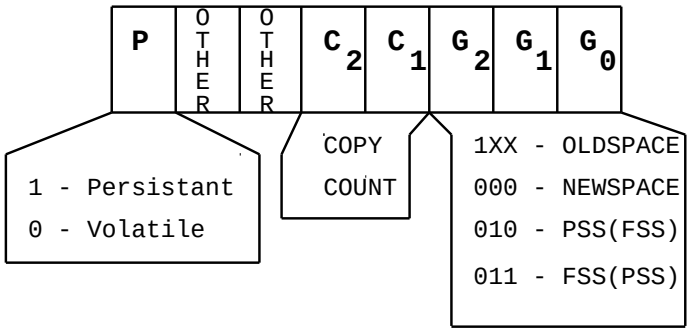
2. The next region of memory is occupied by *OldSpace* identical to the OldSpace in Ungar's Generation Scavenging scheme. Objects that are *tenured*, i.e., in OldSpace may also become inaccessible. These objects are collected by an off-line collection. Objects that have been annotated as *pinned* are a part of OldSpace.

3. The last section of memory in our memory management scheme is *ScavengeSpace* that participates in every global garbage collection. The *ScavengeSpace* region is divided into three regions — NewSpace, PastSurvivorSpace and FutureSurvivorSpace (similar to Ungar's scheme). Figure 5.4 illustrates the organization of scavenge-space. The generation scavenging algorithm proposed by Ungar [Ung84] has been described in the second chapter.

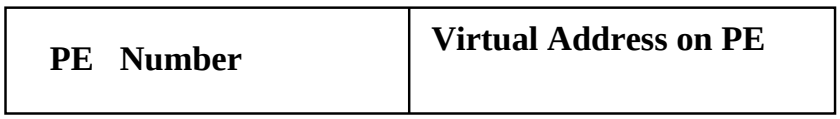
The first two sections of the heap, i.e., the *persistent* region and OldSpace are not examined (traversed) during the process of GGC as they are considered to contain long-lived objects..



FORMAT OF AN OBJECT-NAME



FORMAT OF A TAG



FORMAT OF A VIRTUAL ADDRESS

Figure 5.2: Formats of the Global Nametable Fields

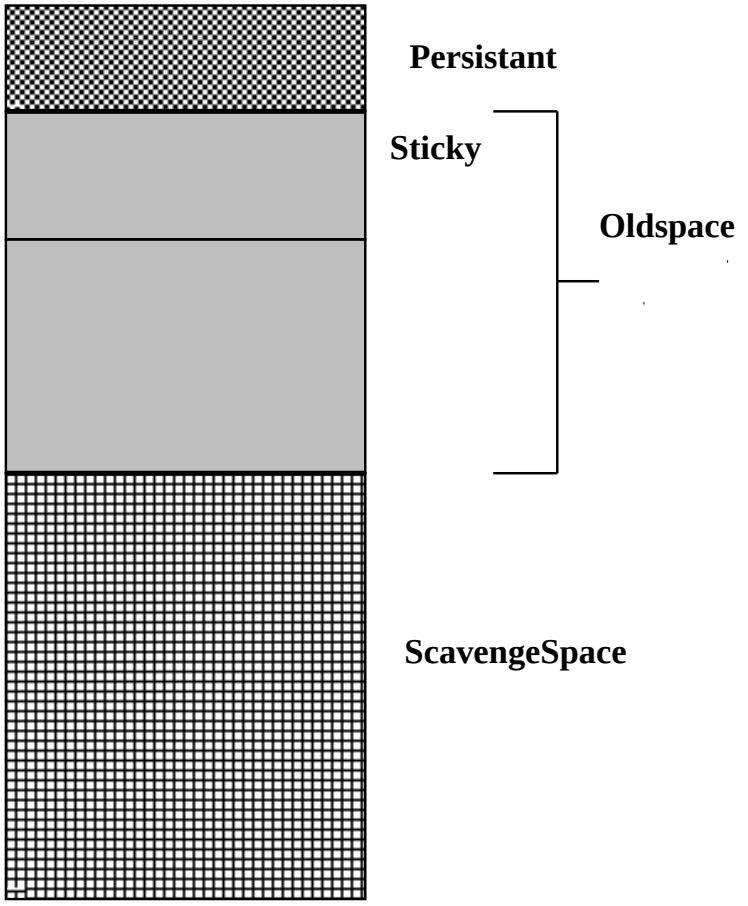


Figure 5.3: Logical Organization of Global Memory

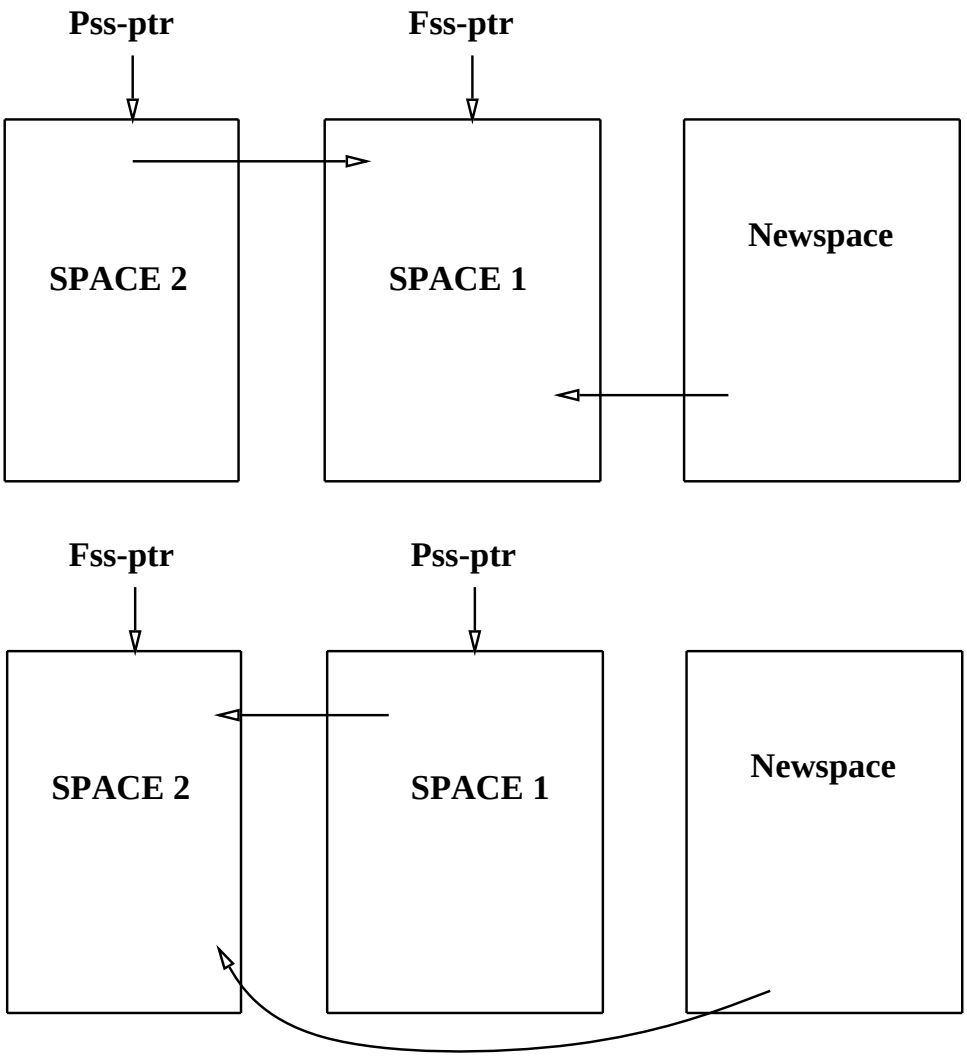


Figure 5.4: Logical Organization of Scavenge Space

Persistent	*Region*	*Tag*
0	**OldSpace**	**1XX**
0	**PastSurvivorSpace**	**001 (010)**
0	**FutureSurvivorSpace**	**010 (001)**
0	**NewSpace**	**000**
1	**In Main Memory**	**000**
1	**In Secondary Memory**	**010**
1	**On Tape**	**100**

Table 5.2: Tag Bits Associated with the Regions in the Workable Heapspace

The bits in the tag field of the global nametable entry of the object indicate the region in memory that the object belong to (See Table 5.2). We would like to emphasize again that the organization described above is logical.

5.4 Persistent Objects

Persistent objects are objects that can be shared among program executions, i.e., the object has the ability to participate in more than one program execution during its lifetime [AFO89]. Many definitions of persistence have been proposed [ABC$^+$84, Low88]; these definitions are in terms of the lifetime of objects.

Commonly, a *persistent object* is an object which outlives the program that creates it. Persistence is a useful property of applications that require sharing of objects, for example, office automation, online transaction processing applications, VLSI design and multiuser systems. The term frequently assumes the ability of an object to survive system shutdowns as well as system, process or media failures. However, Thatte has separated the notion of persistence (ability to outlive the creator) from the notion of *resilience* where data in the object can survive system crashes and failures [Tha86].

To store a persistent object, it must first be mapped into non-volatile or permanent storage like disks. A mechanism must be made available for transferring persistent objects from long term memory to short term memory. If support for persistent objects is not embedded in the language, the programmer explicitly codes the transfer of objects from persistent to short term

memory. The programmer is therefore forced to develop two representations of the same object being modeled, resulting in the loss of abstraction and encapsulation. However, if support for persistent objects is provided for in the programming language, the underlying system handles the translation process. Even in this case, whenever a persistent object needs to be manipulated, it must be brought into short term memory. Storage representations of the object in the two levels of the memory hierarchy could be vastly different and translating between these representations involves substantial time and effort, degrading the performance of the system [ABC+83]. Exclusive persistent memory and persistent object stores have been designed to avoid the translation overheads by using a single representation for both permanent and modifiable versions of an object. [Tha86, CAC+84]

How can one indicate persistence? There have been a number of different approaches to specifying persistence that have had a significant influence on the design of the programming language and object manager. Persistence has been approached from different perspectives and each approach we discuss below presents a unique approach to persistence.

Type Declarations: Persistence can be specified via explicit declarations such as type declarations. Persistence through types may either type-dependent or type-independent. Persitence is *type-dependent* if only objects of certain types can be made persistent.

```
var p: Persistent Integer;
```

Persistence is *type-independent* if all types of objects may be made persistent; this kind of persistence is said to possess *datatype orthogonality* [MA90]. Note that persistent objects may be scoped using this facility.

User or Compiler-inserted Annotations: The user or application programmer may declare persistence by using annotations indicating whether an object is persistent or otherwise. For example,

```
declare human(body:transient, soul:persistent).
```

Explicit creators: This mechanism defines persistence at creation time. We can think of two object creators, one for persistent objects and one for transient objects.

- A transient object creator:

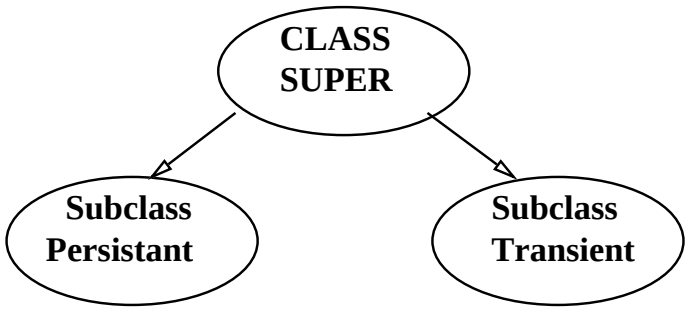


Figure 5.5: Persistence as a Class/Subclass Hierarchy

```
CreateObject (transientObjectName : object-type)
```

- A persistent object creator:

```
CreatePobject(persistentObjectName : object-type)
```

Classification: Alternately, persistence can also be indicated by class specifications mechanisms; creating an object within a specific class or subclass whose behavior implements persistence (See Figure 5.5). If an object/actor is created from within class persistent or if it inherits from class persistent, it is considered to be a persistent object. However, this scheme confines the behavior of the actor to be persistent forever.

Reachability Criteria: A frequently used mechanism to define persistence is to associate objects with a *persistent root*. The *persistent root* is a predefined permanent object that can never die unless explicitly killed. To make an object persistent, it must be made reachable from the persistent root. Using the reachability definition of persistence, what we are doing is functionally identical to annotating persistent objects. In either case, the user is aware of what is persistent and he/she is responsible for specifying that information either via annotations or by making it reachable from the **persistent root** actor.

A tiered naming scheme has other advantages. For instance, in a hierarchically defined object system, say a directory structure, we want entries of tables to be tables themselves. Having a single-level notion of naming i.e. one table with the names of all persistent objects does not facilitate this.

Persistence can be defined at different points in the application development process. It could be defined:

- *Statically at compile time:* This is what happens when a persistent object is created from a particular class or subclass.
- *At object creation time:* using the user/compiler annotation scheme.
- *During execution:* If it is decided during execution that an actor/object is persistent, we can do one of two things.

  1. Make it reachable from a predefined persistent root actor.
  2. Turn on a persistence bit for the object.

  In fact we could adopt a combination of both schemes. The first scheme would be the choice of a language implementor as no additional constructs are introduced into the programming language. Reachability is a concept that is already available in programming languages and using as a specification of persistence simplifies persistent object management.

- *At exit time:* Decide which of the currently existing objects are persistent and give these objects a virtual-name by which they are addressed during subsequent executions.

5.4.1 Implementing Persistence

Bjornerstadt explains three distinct approaches for implementing persistence; the *Workspace* approach, the Persistent Virtual Memory approach, and the *object swapping* approach [Bjo90].

In the Workspace approach, a coarse level of persistence is obtained. Objects are explicitly loaded from files and the systems we are dealing with are not resilient, i.e, they survive controlled shutdowns but not sudden crashes. No runtime distinction can be made between persistent and transient objects in main memory. *Persistent virtual memory* refers to a portion of the address space which is non-volatile and is dedicated to storing persistent objects. But, the number of persistent objects we can define is limited to the virtual memory address space. The Object Swapping approach extends the persistent virtual memory mechanism, where we have a non-volatile secondary storage in addition to *persistent virtual memory*. Objects are swapped

between the virtual address space and secondary storage. As objects are the unit of transfer, there is less thrashing. Furthermore, we can support an arbitrarily large number of persistent objects.

5.4.2 A Generational Approach to Persistence

Our objective is to implement persistent objects using the tiered naming scheme we have developed in this chapter, while allowing the autonomous existence of persistent objects. Our approach is as follows. A portion of the logical address space is dedicated to persistent objects. Transient and persistent objects share the same addressing mechanism without any assumptions about their physical storage. We use a generational scheme for managing persistent objects. Persistent objects can be classified as belonging to a particular generation based on their age. The definition of age with respect to persistent objects differs from that associated with transient objects: the *age of a persistent object* is a measure of when it was last referenced, not when it was created as in conventional objects. In particular, the youngest generation consists of those persistent objects which have been most recently referenced.

Generations of persistent objects are physically located in different storage media. A hierarchy of storage media is specified as follows –

1. Currently referenced persistent objects are are placed in cache memory. However, the size of objects that can fit into a cache is limited. This problem arises because we view an object as an indivisible entity and caching objects in the memory hierarchy is done in terms of whole objects. In languages like CST [HCD89], which implement distributed objects, this may not be a serious problem. Alternately, if an object is implemented as a collection of sub-objects with a common name and an index, we can cache in only the required sub-object.

2. The most recently referenced persistent objects are present in the *persistent* region of main memory.

3. The next generation is located on direct access secondary storage devices like the disk.

4. The final generation, which has not been accessed for a long time is stored away on tape. Magnetic tape storage is the endpoint of the generation hierarchy.

We refer to the transfer of persistent objects from a younger to an older generation as *retiring*. Retiring persistent objects is based on the following tenet:

> *A persistent object which has been recently referenced is more likely to be referenced again than one which has not been recently referenced.*

With the retirement scheme, an object can never be deallocated, it gets archived and the archives are destroyed after a period of time. Therefore, to explicitly delete persistent objects, we use a *delete-persistent* operator. In the object-based memory management system, the *delete-persistent* operator sets the persistent bit of the nametable entry of the object to 0 and the generation-tag to NewSpace. Now the object behaves as a transient object and is collected during the next GC phase if it is inaccessible. Otherwise it is collected when it becomes inaccessible. To summarize, the object-based memory management subsystem provides the following support for persistent objects:

1. A logical region of memory dedicated to persistent objects.
2. A persistent bit in the tag field of the nametable which facilitates explicit deallocation of persistent objects.
3. Generational retirement of persistent objects.

We take no stand on how persistence must be indicated to the system; any one of the techniques described earlier can be used.

Our aim is to integrate the application routines and system management functions within a single computational domain. We find that with suitable abstractions, Actors form a very elegant framework for this. A key feature of system management routines is persistence; these are functions that are applicable over multiple program executions. This has been one of the main motivations in making persistence available as a fundamental feature of our object model.

Chapter 6

Distributed Generation Scavenging

This chapter introduces the distributed generation scavenging algorithm using the hierarchical model of concurrency described in Chapter 3 and the tiered naming scheme that we developed in the previous chapter. In traditional environments, reachability is the only criterion to determine whether an object is accessible. In Section 2, we illustrate complications that can arise while performing GC in actor based environments using traditional schemes. We will then define what is considered garbage in an actor based environment. The distributed generation scavenging algorithm is a two phase algorithm; we describe the algorithm and associated overheads in last section.

6.1 The Problem of Asynchrony in Distributed Garbage Collection

Distributed GC in an asynchronous network introduces new issues which have to be addressed. Traditional garbage collection techniques assume that the topology of reachable objects cannot change dynamically during GC. In the actor model, a mail address may be communicated to an object, thereby increasing the number of acquaintances of an actor. Alternately, an actor could change its behavior and lose an acquaintance.

This dynamicity should be taken into consideration during a GC. Later in this section, we will give examples to show that just accounting for dynamic changes that are taking place

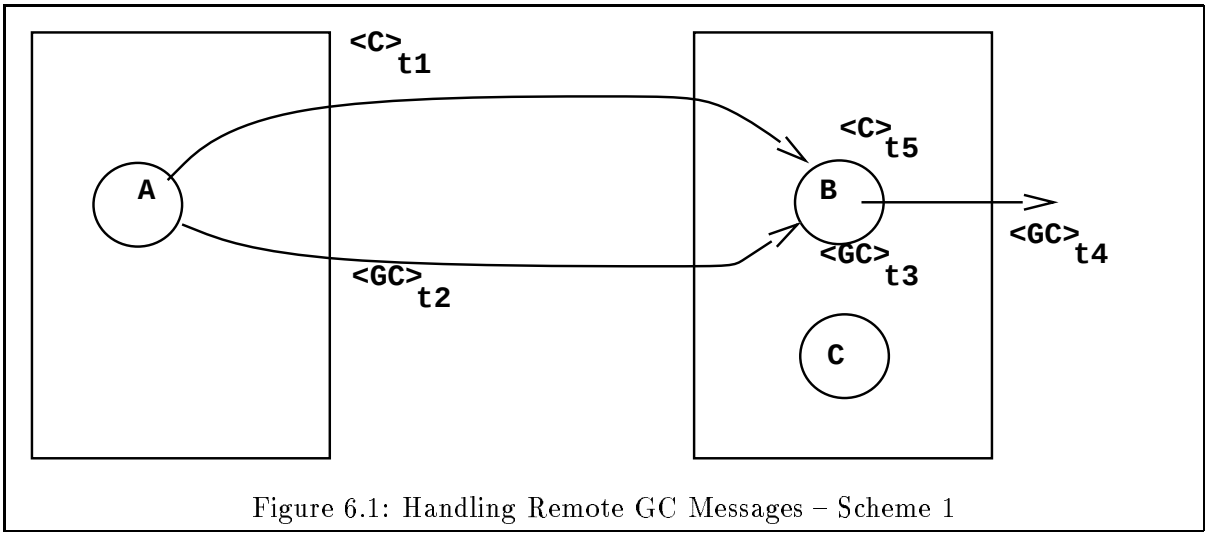


Figure 6.1: Handling Remote GC Messages – Scheme 1

currently is not sufficient to preserve correctness. Changes that can possibly take place should also be taken care of. Otherwise it is possible to lose or gain references.

Furthermore, network communication is asynchronous. Message order is not preserved in communication; a fact which makes it difficult to ensure that there are no messages in the network while an object traverses its references during a GC phase. Therefore, it would be incorrect to make assumptions at the receiving end about the time at which the network has been cleared of all references. Factors such as the non-uniform clock advancement at two nodes in the network and the asynchrony of communication need to be considered when reasoning about the correctness of a GC algorithm for the actor model.

Consider the system of actors depicted in Figure 6.1. Let `A` be an actor and `B` and `C` be its acquaintances. Assume that A first sends a mail message to `B` communicating the mail address of `C` (message $\langle C \rangle_{t1}$). `A` then changes its behavior and loses its reference to object `C`. Now `A` receives a GC message from another object. As `C` is no longer an acquaintance of A, note that `A` will not propagate the GC message to C. Possible methods to handle the GC message at `A` are:

Scheme 1 (Figure 6.1): `A` begins traversing its graph of reachable objects as soon as it receives the GC message. `A` sends a GC message to B (message $\langle GC \rangle_{t2}$). which arrives at `B` before the message containing the mail address of C. `B` continues the traversal phase and propagates the GC message (message $\langle GC \rangle_{t4}$) to its acquaintances, (`C` not included).

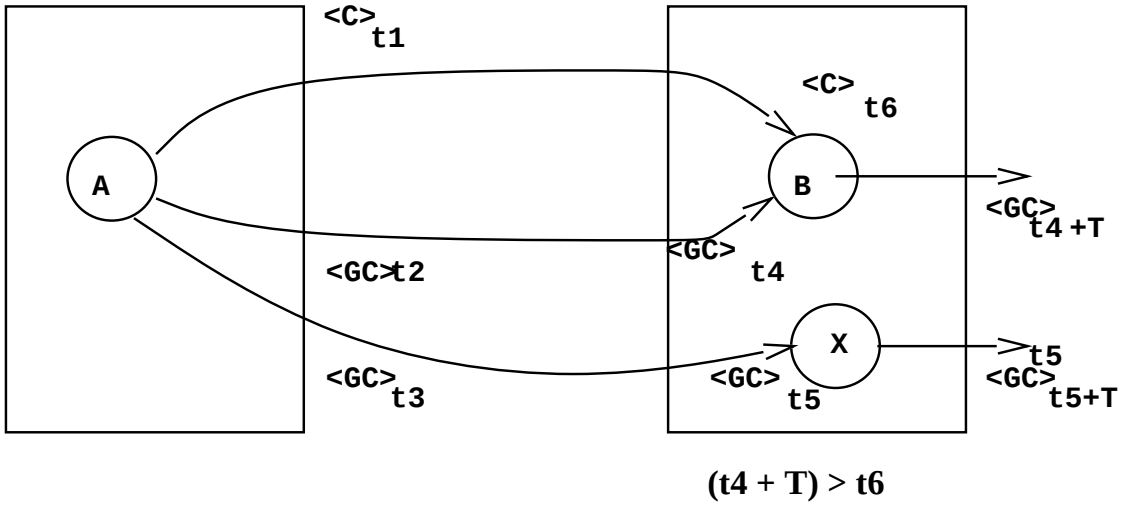


Figure 6.2: Handling Remote GC Messages – Scheme 2a

The message containing the mail address of C arrives later (message $\langle C \rangle_{t5}$). But, at the end of GC, C appears inaccessible and it is garbage collected when it should not be.

Conclusion: Scheme 1 is incorrect.

Scheme 2 (Figure 6.2): The next scheme defines an upper bound on the time taken for a message to reach its destination. We shall assume a maximum message transfer delay T_{max}, i.e., any message injected into the network at time t will reach its destination node by time $t+T_{max}$. Every PU delays the propagation of the GC messages to its acquaintances for a period of time equal to T_{max} to allow any references to the node currently in the network to reach their destinations. A begins propagating GC messages to other reachable nodes in its reachability graph at time T_{max} after it receives the GC message. A sends a GC message (message $\langle GC \rangle_{t2}$) to B which arrives at B (message $\langle GC \rangle_{t4}$) at instant *t4*. Now B delays propagating the GC message to its referents for a period T_{max}. By this time, the reference C would have reached B. This delay is encountered for every GC message. To avoid this, we can selectively delay GC messages only to those objects to which a mail address has been communicated within the last T_{max} clock cycles. In the example in Figure 6.3 we delay the outgoing GC message from A to B for a period T_{max} after the message communicating the mail address of C is sent. In other words, if the message with the address of C was sent at local time t, the GC message to B will be

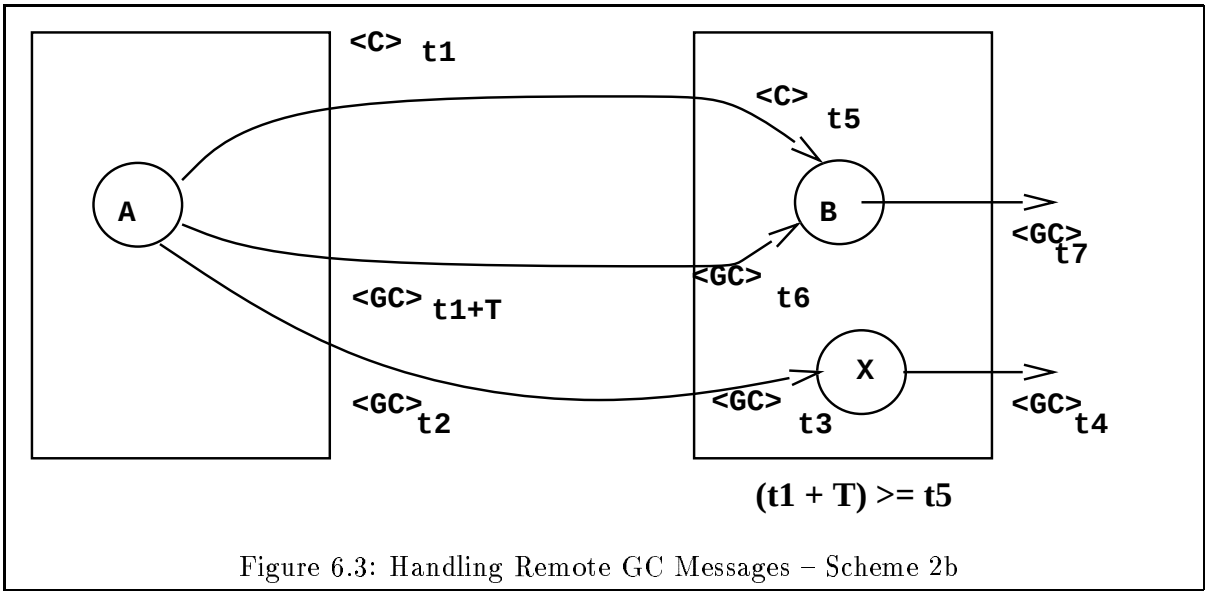


Figure 6.3: Handling Remote GC Messages – Scheme 2b

issued at time t+T_{max}. This would be correct if A was the only inverse acquaintance of B, i.e., A was the only actor that knew about the existence of B. But, if B was known by another actor, say D, the GC message from D may arrive at B prior to the GC message from A, violating the correctness of the situation we have constructed.

Conclusion: Scheme 2 is incorrect and makes an assumption regarding network transfer delay which is unrealistic.

Scheme 3 (Figure 6.4): Both the schemes described above ignore the second and subsequent *GC* messages. Let us refer to such messages as *late* GC messages. Consider the scenario presented in Figure 6.4. As in scheme 3, `A` delays sending the GC message to `B` until the mail address of `C` has reached B. But, now `B` has received a GC message from another inverse acquaintance D. If we ignore the late GC message, `C` will still be considered inaccessible. Therefore, we must provide a mechanism to deal with late GC messages. Here's one solution.

If a PU has already been involved in GC, it has received a GC message and sent out GC messages to its acquaintances. When it receives another GC message, it can ignore it *only* if no new acquaintances have been received. However, if any new acquaintances are received by an object after GC has been initiated on that object, a GC message is issued

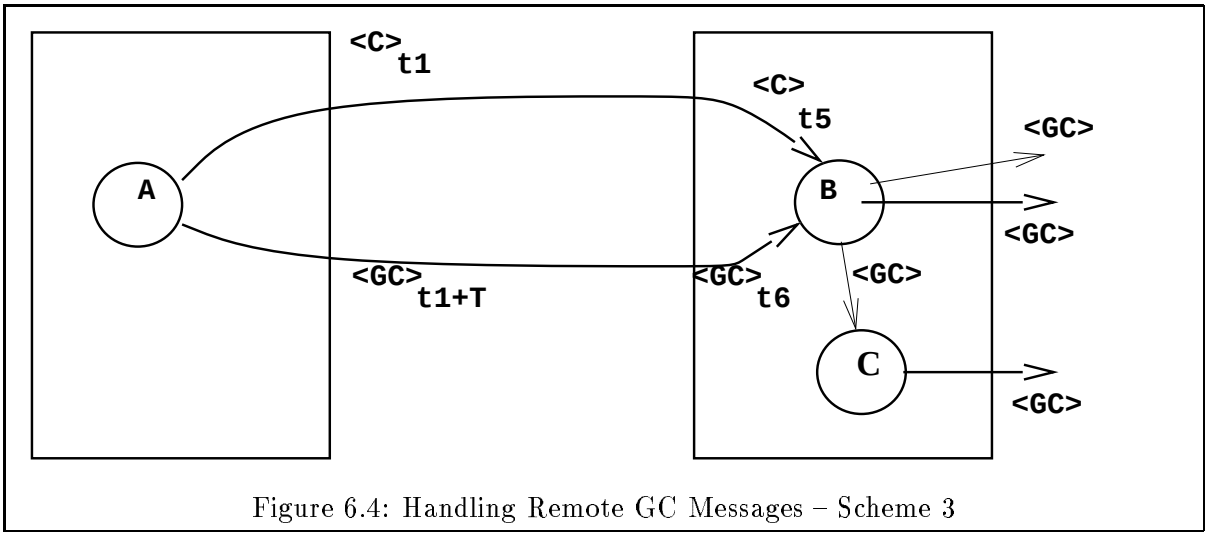


Figure 6.4: Handling Remote GC Messages – Scheme 3

to this new acquaintance. This will cause a traversal of the object graph from the new acquaintance C.

If the object has not participated in GC, we can continue processing as normal. The new acquaintance will be traversed when the object receives a GC message. C has been marked accessible, but we have introduced an additional problem. Now, we cannot determine when the marking phase has completed, i.e., when a node has received all the messages it will need to receive. *There is no way to detect termination of GC.*

Conclusion: Scheme 3 is incorrect.

All the three schemes presented above failed for the same fundamental reason: they assumed that it was possible to record the state of the individual computational agents simultaneously with GC. However, due to the lack of a global time, these systems have no common point of reference based on which all of the Pus can start the GC Phase. The purpose of walking through all these examples was to convince the reader that a consistent global state must be obtained prior to the start of GC. Every processing unit that could get involved in the GC must record a state and use the recorded state (not the current state of the process) during the GC Phase.

6.2 Generational Collection of Actors

The criterion commonly used to detect garbage in conventional garbage collectors is reachability. When dealing with active objects computability must also be considered. We will now define

what are reachable objects and garbage objects in an actor based system. This definition is derived from the work of Kafura et al [KWN90] The *root set* is a set of actors that represent

- the current state of the system (environment variables, control structures like stacks etc.)
- the remembered set (in a generational scheme)
- the persistent set.

A *root actor* is an actor that is a member of the *root set*. An actor which is currently processing messages or has messages to process in its mail queue is an *active actor*. A *inactive actor* is not active currently and has no messages in its mail queue. Also, a inactive actor which has no inverse acquaintances is a *permanently inactive actor*. The set of *reachable actors* is defined recursively as:

- A root actor is a reachable actor.
- Every forward-acquaintance of a reachable actor is reachable.
- Every inverse-acquaintance of a reachable actor that is active or is not permanently inactive is reachable.

An actor which is not reachable is a *garbage actor*.

In other words, an actor which is not currently accessible or cannot potentially become accessible is a garbage actor. We cannot ignore the inverse acquaintances while determining reachability in an actor based model because an inverse acquaintance of a reachable actor may communicate its mail address at any point of time to its reachable acquaintance, thereby making itself reachable. But, if the actor is inactive and can never become active, it cannot communicate its mail address and is therefore inaccessible. Thus, reachability in the Actor model is determined by *liveness*, i.e., by the ability to survive.

Kafura and Nelson proposed an algorithm called the Push-Pull algorithm that is a mark-and-sweep based algorithm augmented with a set of coloring rules [KWN90]. The drawback of this algorithm is that it requires the actor system to be stopped during the entire mark and collection process. We will make use of results from the algorithms proposed in [KWN90] to design a generational garbage collection scheme for actors in a distributed system.

Our objective is the design of a GC algorithm that exhibits the following properties:

- The algorithm is *non-halting*, i.e., the processors in a system need not halt execution of an actor program to start a global garbage collection on the program.
- The algorithm we develop is *conservative*, i.e., it identifies only a subset of inaccessible objects during a GC. Our conjecture is that any non-halting, efficient algorithm for GC will need to be conservative.

Our algorithm is an extension of the *generation scavenging* algorithm originally proposed by Ungar [Ung84]. The algorithm has been adapted to perform efficient storage reclamation on distributed systems with a shared object namespace. It can be extended to any system that has distributed shared memory, i.e., a *shared address space* distributed across the processing units. In the rest of this thesis, we will refer to our algorithm as the *distributed generation scavenging* (DGS) algorithm.

The execution model for the DGS algorithm is based on the universal model described in Chapter 3. As a further optimization, each node of the multicomputer has an actively running local garbage collection process on its local memory. Local garbage collection will be discussed in further detail in the section on optimizations to the DGS algorithm.

The memory model we have adopted is the distributed shared memory model. The organization of the distributed shared memory is as described in Chapter 5, i.e., logical memory space is divided into *persistent*, *OldSpace* and *ScavengeSpace* regions.

The bits in the tag field of the global nametable entry of the object indicate the region in memory that the object belong to (See Table 5.2). We would like to emphasize again that the organization described above is logical. The first two sections of the heap, i.e., the *persistent* region and OldSpace are not examined (traversed) during the process of GC since they are assumed to contain long-lived objects.

6.3 The Distributed Generation Scavenging (DGS) Algorithm

The DGS algorithm is a two phase algorithm. The first phase, also called the *distributed scavenge* phase traverses and processes all reachable objects in the system starting from a predefined *root set*. This is a distributed phase, as the name indicates, and it involves inter-processor communication. The second phase is the *localClear phase* in which every processor clears all inaccessible objects in its local memory. The *localClear phase* does not require any

interprocessor communication. There is a clear line of temporal demarcation between Phase 1 and Phase 2, i.e., Phase 2 on all PU's can begin only after Phase 1 of GC has been completed in the system. The termination of Phase 1 is determined by back-propagation messages to the roots where GC is initiated. The back-propagation messages are transmitted from an actor to its parents in the reachability graph when the subgraph under the node has completed Phase 1 of GC. In other words, when all ScavengeSpace references of a node have been through Phase 1, a back-propagation message is sent to the parent of the node that issued the GC message. The synchronization between the termination of Phase 1 and the initiation of Phase 2 is brought about without halting the configuration of processing units.

Synchronization in the DGS Algorithm

Earlier in this chapter, we illustrated the need for capturing all possible references prior to the start of Phase 1 of the DGS Algorithm. References contained in messages that are in transit must be accounted for either at the source of the message or at the destination. What we require is a consistent snapshot of the system state with respect to which GC can proceed while the user process progresses. As there is no notion of a physically global clock based on which the nodes can obtain a consistent snapshot, the snapshot is distributed, i.e., every processor has a snapshot of all possible acquaintances when the node receives the message communicating that a GC has been requested. We have proposed a set of algorithms that can be used to record the distributed snapshot, called the *network clearance algorithms*, discussed in the following chapter. For the present, we will assume that there exists an underlying scheme that records a uniform state before, during and after a GC.

The pre-GC synchronization is necessary to obtain a uniform system state. Subsequent synchronization is needed to mark end of Phase 1 and trigger the start of Phase 2 on every processing unit in the system. The post-GC synchronization communicates the termination of GC to every node in the system.

When a GC is called, only the generation bits in the tag field of accessible object entries are altered. This is logically equivalent to moving the object from one space to another.

The *copy-count* bits are used to implement a tenuring policy. and are incremented whenever the object survives a GC. When this count reaches a threshold value, the object is tenured from

Region 1	*Copy-Count*	*Region 2*
PastSurvivorSpace	**Greater than Threshold**	**OldSpace**
PastSurvivorSpace	**Less than Threshold**	**FutureSurvivorSpace**
NewSpace	**Don't care**	**FutureSurvivorSpace**

Table 6.1: Moving Objects in Generations

ScavengeSpace to OldSpace and is traversed only during periodic total garbage collections that collect garbage objects in OldSpace and ScavengeSpace.

The process of collection can be summarized as follows: Move all the accessible and active objects from Region1 to Region2 in the order specified - Once DGS-Phase 1 is completed, garbage objects in NewSpace and PastSurvivor Space can be cleared. Such objects can be distinguished by examining their generation tags. The interpretation of the PastSurvivorSpace and FutureSurvivorSpace tags are flipped, which is equivalent to flipping the FSS-ptr and PSS-ptr. Therefore, from a logical standpoint, FutureSurvivorSpace is now empty and ready to begin a new collection. Figure 6.5 illustrates 3 consecutive GCs and the interpretation of the tag bits in each collection.

There are a number of local and global data structures associated with the DGS scheme and we those involved in the DGS algorithm.

- *GC-Phase* – A local variable on every PU

  Every processor maintains a variable called *Global-GC* which is 0 when there is no global GC going on in the processor. The first object on a given processing unit to receive a *scavenge* or GC message sets GC-Phase to 1. When Phase 2 is initiated on the processing unit, a GC thread on the processing unit sets this variable to 2. After Phase 2 is completed, this variable is reset to 0.

- *GC-is-on* – A single system wide variable

  There is a single system-wide *GC-is-on* flag that is set to *True* when there is a global GC. Any processor requesting a GC must check the *GC-is-on* flag before it initiates a global collection. If the flag is already *True*, it indicates that a GC is underway and the requesting processing unit must wait for the GC to complete and release sufficient

Future-Tag = 01

Object Name	Tag	Virtual Address
A	~~00~~ 01	
B	00	
C	~~00~~ 01	
D	00	
E	00	
F	~~00~~ 01	

GGC 1

Tag	Space
00	Newspace
10	PSS

Future-Tag = 10

Object Name	Tag	Virtual Address
A	01	
C	~~01~~ 10	
H	~~00~~ 10	
F	01	
I	~~00~~ 10	
G	01	

GGC 2

Tag	Space
00	Newspace
01	PSS

Future-Tag = 01

Object Name	Tag	Virtual Address
C	10	
J	~~00~~ 01	
H	~~10~~ 01	
K	~~00~~ 01	
I	10	
L	10	

GGC 3

Tag	Space
00	Newspace
10	PSS

Figure 6.5: Global Garbage Collections in Progress

memory. If the flag is not set at the time a processor requests a GC, the processor is responsible for setting this flag to *True* to indicate that a GC is going on. This flag must also be reset to *False* at the end of GC Phase 2.

- **The Distributed Global Nametable:** Used for object-name to virtual address translation.
- **The Local Nametable:** Used for virtual address to physical address translation.
- **The Local Memory:** The local memory in which globally accessible objects are situated.
- **The Remembered Set:** Has ScavengeSpace entries pointed to by OldSpace objects
- **The Forward Acquaintance Set:** Holds the forward acquaintances of the actors on that processing unit.
- **The Inverse Acquaintance Set:** Holds the inverse acquaintances of the actors on that processing unit.

6.3.1 Assumptions of the Algorithm

The following conditions must be guaranteed during a global garbage collection phase.

- Only one global collection can occur in the system at one time. This is ensured by a proper setting of the *GC-is-on* flag.
- Every processing unit knows how to access the root set. The root set for a global garbage collection includes:
  1. The current set of the system including the active control structures.
  2. The Remembered Set
  3. The Persistent Set
- There are no pre-GC messages in the system at the time of GC Phase 1 initiation. The network clearance algorithms described in the next chapter will clear the system of all pre-GC messages prior to GC Phase 1.

- The system is not fault-tolerant i.e. the communication and processing subsystems are not subject to faults. This assumption will later be weakened to allow fail-stop (but not Byzantine) faults.

- Migration of objects during any of the GC Phases is not permitted. Currently, we are not certain if this restriction can be lifted without violating the correctness of the algorithm.

6.3.2 Phase 1: The Distributed Scavenge Phase

We first define some more terms used in Distributed Generation Scavenging. Objects or actors in the system possess an *object-state* which may be *touched*, *untouched* or *suspended*. *Touched* objects are objects which are accessible and whose current acquaintances have been traversed. Untouched objects have not yet received a GC message. GC messages scavenge forward acquaintances and GC^{-1} messages scavenge inverse acquaintances. When an inactive actor receives a GC^{-1} message, it becomes *suspended*, which means that it is non-garbage if it can become active. The process of *touching* the accessible nodes in Scavengespace is carried out in accordance with the *Principle of Monotonicity* which states that once an actor has been marked as *touched* during a GC phase, it cannot subsequently be untouched or suspended during the same GC phase.

We now give a pseudocode of the distributed scavenge phase:

```
(defun DistributedScavengePhase()
  (if (GC-IS-ON)
      (Wait for collection to complete and then reallocate)
    else (traverseRoot(root))))
```

If there is no global garbage collection process ongoing at the time, the function DistributedScavengePhase() is called, a GC process is initiated, and this will set *GC-is-on* to 1.

```
(defun traverseRoot(rootActor)
  (set GC-IS-ON to true)
  (set  object-status of rootActor to touched)
  (for every actor A in the root set
       (set object-status of A to touched)
       (for every forward acquaintance facq of actor A (scavengeForwardAcq (facq, A)))
       (for every inverse acquaintance iacq of actor A (scavengeInverseAcq (iacq, A)))
```

```
      (when (receivedAllAcks (A) (backPropogation(rootactor)))))
  (when (receivedAllAcks (rootActor))
(signal termination(DistributedScavengePhase)))))
```

The initial objects that are traversed are the root-set objects. For every object traversed, the first phase of the DGS algorithm updates the nametable entries of the object and scavenges its references. It also sets the *GC-Phase* field in the object to 1, indicating that the object is involving in Phase 1 of the DGS algorithm. For every actor in the root set, we have to scavenge both its acquaintances (acq) and inverse acquaintances (acq^{-1}) by calling the following functions:

scavengeForwardAcq(actorName)

scavengeInverseAcq(actorName)

```
(defun scavengeForwardAcq(actor, sender)
  (if ((Actor in OldSpace) or (Actor is persistent)) (ignore and exit)
    (if (getGenerationTag(actor) = futureTag) (backPropogation(sender))
      ((set object-status of actor touched)
       (if (copyCount(actor) ≥ Threshold)
   (setgenerationTag (actor, OldSpaceTag))
 (setgenerationTag (actor, futureTag)))
      (for every forward acquaintance facq of actor (scavengeForwardAcq (facq, actor)))
      (for every inverse acquaintance iacq of actor (scavengeInverseAcq (iacq, actor)))
      (when (receivedAllAcks (actor) (backPropogation(sender)))))))
```

If the actor is persistent or is in OldSpace, it is already accessible and can be ignores during the distributedScavenge phase. However, if the actor is in ScavengeSpace and it has (object-tag = future-tag), GC has already been initiated and a back-propagation message is issued to the sender node. However, note that the back-propagation message to the sender node that initiated GC is issued only after the actor has received acknowledgments (back propagation messages) from all its acquaintances and inverse acquaintances. If the actor is in ScavengeSpace and it has (object-tag $\neq$ future-tag), the scavenge message is processed recursively called the SFA and SIA (scavengeInverseAcq) functions. When the object has received back-propagation messages from all its acquaintances, issue a back-propagation message to the sender node that initiated GC.

```
(defun scavengeInverseAcq(actor, sender)
  (if ((Actor in OldSpace) or (Actor is persistent)) (ignore and exit)
    (if (getGenerationTag(actor) = futureTag) (backPropogation(sender))
      (if (actor is ACTIVE)
  ((set object-status of actor touched)
   (if (copyCount(actor) ≥ Threshold)
       (setgenerationTag (actor, OldSpaceTag))
     (setgenerationTag (actor, futureTag)))
   (for every forward acquaintance facq of actor (scavengeForwardAcq (facq, actor)))
   (for every inverse acquaintance iacq of actor (scavengeInverseAcq (iacq, actor)))
   (when (receivedAllAcks (actor) (backPropogation(sender)))))
;;; actor is INACTIVE
((set object-status to suspended)
 (for every inverse acquaintance iacq of actor
      (scavengeInverseAcq (iacq, actor))))))))
```

In this case, if the (object is active) we treat it just as a forward acquaintance. However, if the actor is inactive, the actor is suspended and the SIA algorithm is called recursively on its inverse acquaintances. Because only an inverse acquaintance can cause an inactive actor to become active, we are not interested in the forward acquaintances of a suspended actor. If the inactive actor can become active, it will be touched and then subsequently, its forward acquaintances will be touched as well. The state transitions that take place in the ScavengeSpace during a GC occur as shown in Figure 6.6.

6.3.3 Phase 2: The Local Clear Phase

```
(defun LocalClearPhase()
    (for every processing unit (PU) in the system
        (clearLocalMemory(PU))
        (clearDistributedNametable(PU))
```

The second phase of the DGS algorithm is carried out on every processing unit and it clears all inaccessible objects in that PU's local memory. Global nametable entries on the local processing unit that are inaccessible are also cleared during this phase. A local compaction may be performed in which case the corresponding entries in the local nametable are updated.

```
(defun clearLocalMemory(PU)
```

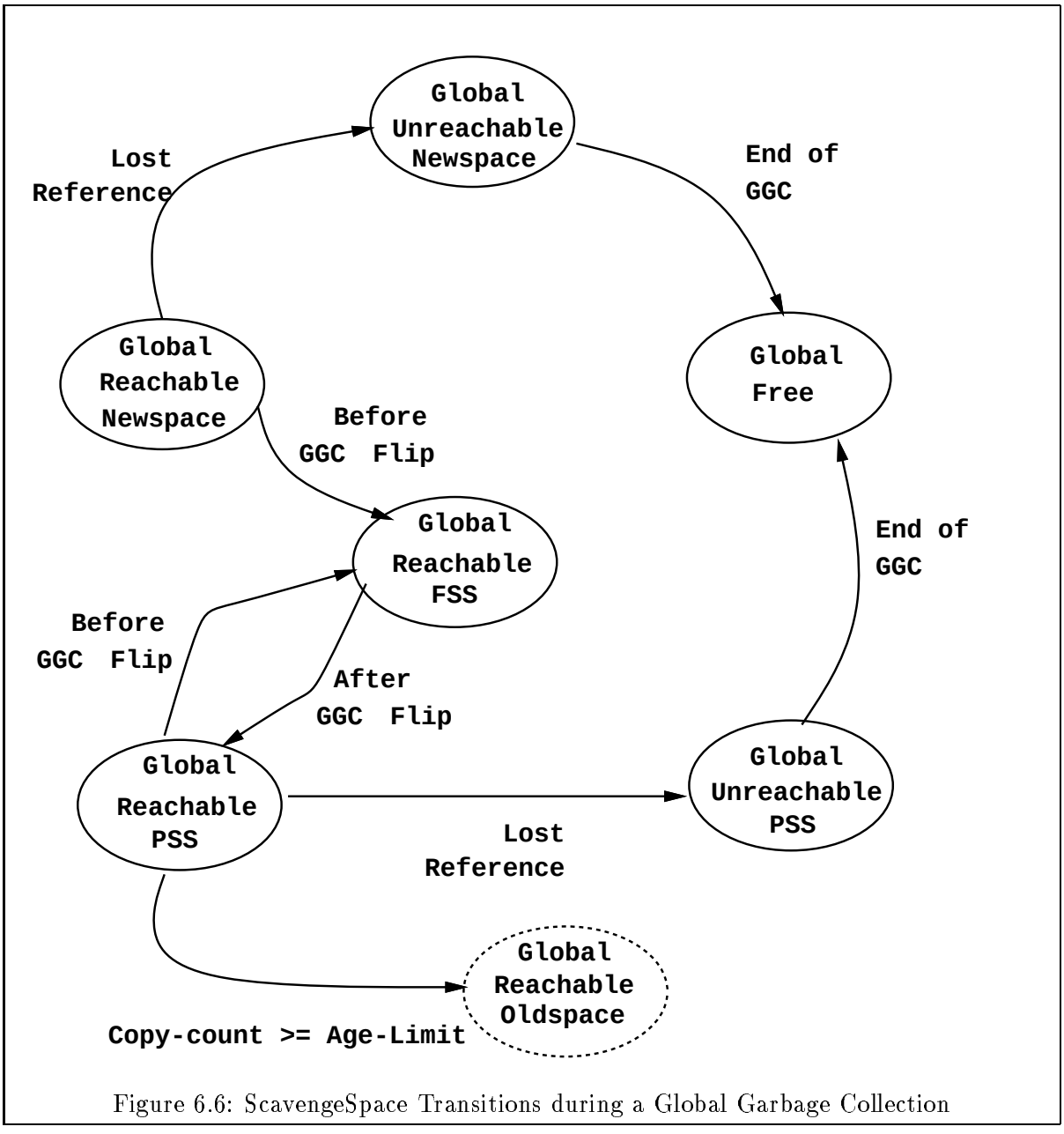


Figure 6.6: ScavengeSpace Transitions during a Global Garbage Collection

```
(for every actor A in the processing unit PU
     (if (((actor is not in OldSpace) or (actor is not persistent))
   and (actor is touched))
  ((move actor to a compacted region of local memory)
   (set object-status of actor to untouched)
   (increment copyCount of the actor by 1))
else (clear memory area occupied by object))))
```

The local nametable has a list of global objects that currently reside in local memory. Using this list, it is possible to get to every object in the memory of the processing unit. Objects that are persistent and objects in OldSpace automatically survive. For objects in ScavengeSpace, the *object-status* field in the object header indicates whether the object is accessible or not. If the object is accessible, the *object-status* field in the header has been set to touched by Phase 1 of the DGS algorithm. Phase 2 of the DGS algorithm resets this field to untouched and increments the copy-count of the object by 1. If the object is not touched, the object is inaccessible. The size (length) of a garbage object is a part of the object header and it is possible to determine how many contiguous words in memory must be cleared. Once the garbage objects have been removed from memory, an optional local compaction will yield a contiguous region of free memory on every processing unit.

```
(defun clearDistributedNametable(PU)
  (for every entry in the distributed nametable on the local PU
       (if ((getGenerationTag(entry)) ≠ futureTag)
   (clear entry corresponding to the object))))
```

In addition to clearing garbage objects in local memory, the second phase must also remove garbage entries in the distributed global nametable. Each processing unit is responsible for removing inaccessible entries in the portion of the global nametable situated on that processor. During Phase 2 of global GC, the processing unit scans the entries in the portion of the global nametable in its local memory and deletes those entries in ScavengeSpace that have a generation-tag equal to PSS-tag or newSpace-tag. Because all accessible objects in ScavengeSpace have a generation-tag equal to future-tag after Phase 1 of the DGS algorithm, only garbage entries can be removed. Note that although the global nametable entries corresponding to garbage objects are cleared at the PU's where the entries are situated, the objects themselves are independently cleared at the PU's where the object currently resides. These objects do not have their touched

bits set and are automatically deleted. There is no message communication between the PU where the nametable entry of the inaccessible object exists and the PU where the object actually resides. Thus, clearing global nametable entries can be carried out independently on every processor without any interprocessor communication. Figure ?? summarizes the actions taken by Phase 2 of the Distributed Generation Scavenging Algorithm.

Housekeeping Work in Phase 2

Although some operations may not be obvious in the description of the algorithm sketched earlier, the local PU has a number of housekeeping duties to perform during a GC. These operations involve a local cost, but they do not contribute to delays in the GC time on a global scale. Some of these operations are:

- Automatic deletion or update of local nametable entries.
- Compaction of the distributed nametable entries.
- Local compaction (optional).
- Clearing words in memory which contain garbage objects.

6.3.4 Post-GC Work

The following globally accessible variables have to be reset at the end of global GC -

1. The generation scavenging algorithm flips FutureSurvivorSpace and PastSurvivorSpace pointers. We achieve the same effect by flipping tag interpretation. Therefore, after Phase 2 of the DGS algorithm has been completed and future-tag = PSS-tag.
2. The flag *GC-is-on* must be reset to 0.

6.4 Migration and Compaction

Two important issues that form a part of object management are migration and compaction. In addition to the normal bookkeeping overhead of updating table entries on migration, we must handle remembered set references appropriately. Anytime an object is migrated from one

processing unit to another, the remembered set entry associated with the object (if one exists) must be noted in the new processing unit. Migration dependencies which were described in Chapter 4 must be resolved.

The drawback of a table-driven approach is that we lose compaction that is characteristic of copying schemes. It is not clear how much and where we want to compact. Local compaction is desirable as long as the corresponding address mapping tables reflect this change. Again, this need not be done at the time of collection. The global nametable needs to reflect this change only if it is maintaining the actual physical offset in addition to the node that has the object. The particular entry being altered must be locked while this change is being made. What we are not achieving is compaction across processing units. Techniques to perform global compaction should be locality driven. The tradeoff between locality and distribution for concurrency, i.e., nodes that refer to each other frequently placed in close proximity but not on the same node, is an interesting topic for further investigation, but beyond the scope of this thesis. This can be done via user-annotations as in Emerald [Hut87]. Techniques to use the virtual memory subsystem to achieve dynamic grouping of objects to promote locality have been proposed in the Mushroom architecture [Wil87].

However, it is important to observe that the price of global compaction is the reduction in parallelism. Compaction may result in loss of concurrency; consider two actors (objects) that have been intentionally allocated on the shared heapspace of different nodes so that messages to these objects can be executed in parallel. If locality considerations may place these objects on the same node in the system, the execution of messages sent to these objects will be sequential.

Implementing a copying algorithm that physically moves objects from one region in a large shared namespace to another would constitute a large communication and processing overhead (assembling and disassembling the message, for instance). It would also give rise to increased NW contentions. The reliability of such a mechanism is highly questionable. With objects being transferred throughout the system, response times of the GC scheme become difficult to predict, ruling out possibilities for real-time extensions. The DGS algorithm achieves the effect of generational GC without physically moving objects across the NW during GC avoiding all the problems mentioned above.

The DGS algorithm assumes that all processing units in the system have a uniform picture of the entire system. In the next chapter we propose techniques to achieve a consistent picture

with respect to which GC is performed. As the DGS algorithm is generation based, it is a low overhead algorithm and can be tuned to the response time requirements of the application. However, we need to assess the costs of obtaining a uniform snapshot of the system and the scalability of this process.

Chapter 7

Network Clearance in an Asynchronous System

This chapter presents an algorithm by which a consistent state of a distributed system can be obtained without halting computation or message transmission in the system. In the next section, we briefly explain methods to achieve a safe garbage collection. In accordance with the goals of memory management established earlier in this thesis, we will develop a scheme which allows the application program to continue simultaneously with garbage collection. Section 3 is a collection of definitions and terminology that we have developed for the network clearance algorithms. In subsequent sections, we will explain two network clearance phases, called *broadcast* and *bulldoze* waves.

7.1 A Consistent State

In a distributed system, the PU's do not have access to a global clock. Hewitt and Baker have argued that the notion of globality in terms of time, state and address space must be replaced by corresponding local parameters [HB77] in a distributed system. Each processor records actions as they occur in terms of a local clock – also called *arrival orders*. The scope of locality is limited to an *event*; global action is replaced by causal relationships between events (*activation order*) and a sequence of global states is replaced by an event diagram which represents the transitive closure of the activation order and all arrival orders. An event represents the transition from one local state to another. Events which do not belong to a partial order can proceed concurrently.

However, such an asynchronous system can be made to exhibit global synchronous behavior by having all synchronizing events cross a global synchronization *barrier*.

As a unique global time cannot be defined in a distributed system, obtaining a consistent state cannot be done at the same instant in real time on all PUs. This problem has been discussed in literature as the *Global State Detection* problem [CL85] and a solution to this problem has solved a number of important issues, for example, deadlock [CML83, GS80] and termination detection [DFVG83, CM86, Fra80].

Network clearance is the process of clearing a computational system of all messages which are currently in transit. It is an abstract term used to include processing, eliminating and handling messages in the system. This process could be utilized to clear the system of *all* messages in transit and obtain a frozen state of the system. Alternately, the network clearance process could be parameterized (with the message type or message color as a parameter) to clear the system of particular kinds of messages.

For example, consider the distributed GC problem. The marking phase (the process of determining what is accessible) can be performed only after all references in transit have reached their destination. Using network clearance, it is possible to force any messages in transit to reach their destination and continue further processing only after obtaining a consistent picture of the entire system for garbage collection.

In Chapter 6, we proposed the DGS (Distributed Generation Scavenging) algorithm. As a prerequisite for the correct execution of the algorithm, we assumed that the system is free of pre-GC messages when Phase 1 of DGS was initiated. The DGS algorithm also assumed a clear line of temporal demarcation between Phase1 termination and Phase2 initiation. The network clearance scheme can be used to realize both the above assumptions and obtain a uniform state of the system.

A consistent snapshot of the system with respect to which distributed GC can be performed is a critical necessity in a dynamically varying topology such as an actor based system. Acquaintances may be lost and acquired during processing, altering the reachability graph continuously. A currently unreachable actor which may potentially become accessible, must be accounted for. There is, however, one invariant that permits DGC in a system with a dynamic topology.

> *An actor/object which is identified as garbage at some point*
> *in time can never become non-garbage later.*

This implies that there must exist a current or potential reference to a reachable actor at the point at which the snapshot is taken. In other words, addresses of existing actors must be communicated; they cannot be computed or "guessed". A number of subtle errors which can occur due to dynamically changing topology in an asynchronous system have been discussed in Chapter 6. For example, an acquaintance may be lost due to a change of behavior, but the mail address of this acquaintance may currently be in transit in the network. These addresses or *references in transit* must be captured at either the source or the destination. Therefore, in addition to the current and static acquaintances of an actor, we must handle

- **Pending References** – may be present in one of two areas – in transit at some point in the network or in the mail queue of an actor. Pending references are computation messages which may affect the global GC phase in changing the accessibility graph, i.e, a message carrying an object address can introduce a new link in the graph and must be accounted for prior to initiation of GGC. All messages must be processed so that any addresses in transit or in the mail-queue of an actor may be captured at some node.

- **Potential References** – include references or acquaintances that do not currently exist, but could possibly exist.

In order to guarantee safety of the DGS algorithm, we need to guarantee that there are no uncaptured references in the system during garbage collection. There are three ways of ascertaining this condition:

Stop the entire system: and wait for all messages to be cleared out of the network with the assumption of a finite or bounded message delivery latency.

Stop the entire system by issuing *curfew* messages: along the dimensions of the network. The purpose of these messages is to clear every link in the network of messages in transit. After a node issues the *curfew* messages to its neighbors, it does not continue any further processing and waits for the GC process to complete. The *curfew* messages, therefore determine the halting point of the nodes in the system. Once the GC process is completed, the computations suspended on every PU have to be restarted.

Continue processing after obtaining a consistent global state: A uniform global state serves as a checkpoint with reference to which garbage collection can be performed. This

is the mechanism used by the network clearance algorithm to obtain a global snapshot of the system for various phases in the DGS algorithm. Unlike the earlier case, every processor continues with its processing after this checkpoint has been determined.

7.2 Assumptions of the Network Clearance Algorithm

The network clearance algorithm makes the following assumptions about the topology and behavior of the network:

- There is no notion of a global clock in the distributed system.
- The network is a direct network with unbounded message buffers. For the purposes of analysis and illustration, we will assume that the topology of the system is that of a multicomputer mesh interconnection (2-d mesh interconnection). In general, the analysis can be applied to a k-ary n-cube (topologies such as ring, torus and butterfly) The nodes are located at the cross points of a grid and each node is connected to its neighbors by point-to-point interconnections. Each node has a processing unit (PU) with a processor, memory and message handling facilities.
- The *start* and *finish* nodes are defined in such a manner that there exists an Fpath from the *start* node to every other node in the network and a Bpath from the *finish* node to every other node in the network.
- The routing mechanism in the network is a progress-only routing mechanism, i.e., if a message is sent from point A to point B in the network, it moves closer to its destination at every hop. This is less restrictive than *dimension-order* routing where messages are routed in the decreasing order of dimension. Either store-and-forward routing or wormhole routing may be used [Dal86] [1]. The transmission of the flits occurs in a pipelined fashion.
- If there exists only a single link between the source and destination of a message, i.e., if the destination node is a Fneighbor or Bneighbor of the source node, *message-order preservation* is assumed. Along a single link in the network messages are communicated in a FIFO fashion. This is a consequence of progress-only routing.

[1] In wormhole routing is implemented, every message is divided into a number of flits (flow control digits) which is the unit of transmission.

- There is only one actor on a node. This restriction is made in order to simplify the explanation of the algorithm and will later be removed.
- The nodes and links are fail-proof, i.e., no faults or network failures can occur in the system.

The network clearance algorithms have been designed to work in coordination with the DGS algorithm in the hierarchical memory management scheme. Before explaining the *network clearance* algorithms, we define terms used in the distributed garbage collection process.

7.3 Definitions and Terminology

A node $A = (a_1, a_2, ..., a_n)$ where a_i are coordinates, is a *Fneighbor* of a node $B(b_1, b_2, ..., b_n)$ if $\exists i(1 \leq i \leq n)$, $a_i = b_{i+1}$ and $a_j = b_j$ for $j <> i$. An Fpath is a path in the network that progresses only along Fneighbors. Similarly, a node $A(a_1, a_2, ..., a_n)$ is a *Bneighbor* of a node $B(b_1, b_2, ..., b_n)$ if $\exists i(1 \leq i \leq n)$, $a_i = b_{i-1}$ and $a_j = b_j$ for $j <> i$. A path in the network that progresses only along Bneighbors is a *B-path*. The *start node* of the system is the node from which there exists an Fpath to every other node in the system. The *finish Node* is the node from which there exists an Bpath to every other node in the system. As mentioned earlier, the *root node* is always accessible.

A em touched node is an accessible node which has participated in Phase 1 of a GGC. An *untouched node*has not yet participated in GGC and it may be accessible or inaccessible. A suspended node is a blocked node which may become active at any point of time (and thereby become non-garbage).

Broadcast messages are messages that are sent to every node in the network. *Fbulldozer messages* are network clearance messages initiated at the *Start node* propagating along Fpaths from every node to its Fneighbor (complete link cover). *Bbulldozer messages* are network clearance messages initiated at the *Finish node* propagating along Bpaths from every node to its Bneighbor (complete link cover). Note that the number of messages needed to accomplish a broadcast are less than the number of messages needed to accomplish a bulldoze wave. Messages in the system may be *old, new* or *GC* messages. *Old Messages* originate from objects that are untouched and have not received the bulldozer messages. *GC Messages* are garbage collection

messages. They are of two kinds : GC_{acq} messages sent to the acquaintances of a node and GC_{invacq} messages sent to the inverse acquaintances of a node. GC_{acq} messages always make the target node *touched*. When a GC_{invacq} message is sent to a blocked node, the target node is suspended and when it is sent to an active node, the target node becomes *touched*. Once a node receives a bulldozer message, it issues only *new* messages.

7.4 Tagged Messages in the System

In order to detect termination of the global-GC process as well as to ensure progress in the system, we adopt a colored message passing scheme similar in flavor to that used in the MOSAIC system [Ath87]. The *tagged message scheme* is a coloring mechanism associated with the messages traveling in a network. Apart from messages introduced by the network clearance system, messages are given *tags* or *colors* to classify them as pre-GC, GC and post-GC messages. Pre-GC messages are those computation messages that exist in the system prior to the GC snapshot. Scavenge messages or GC messages are of two kinds:

1. GC_{acq} – are garbage collect messages sent from an actor to its forward acquaintances.

2. $GC_{inv-acq}$ – are garbage collect messages sent from an actor to its inverse acquaintances.

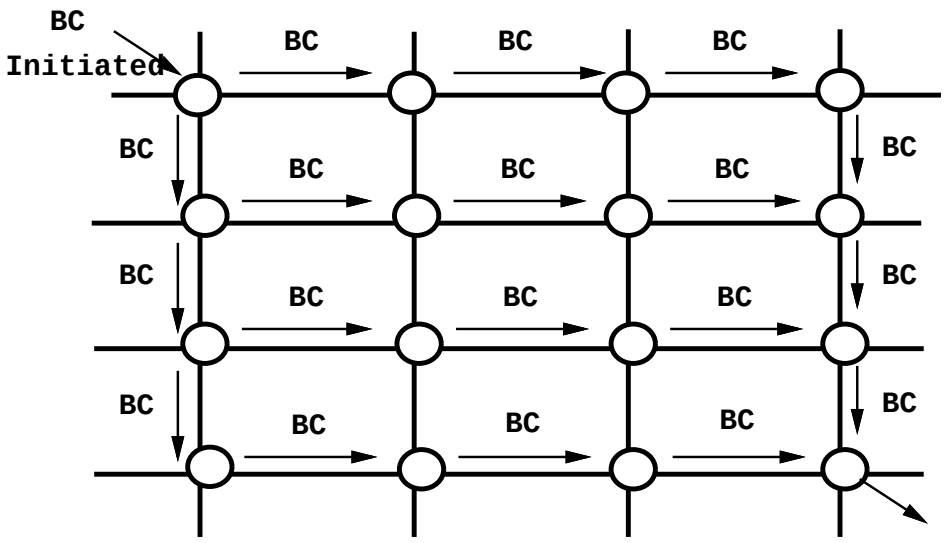


Figure 7.1: The Broadcast Wavefront

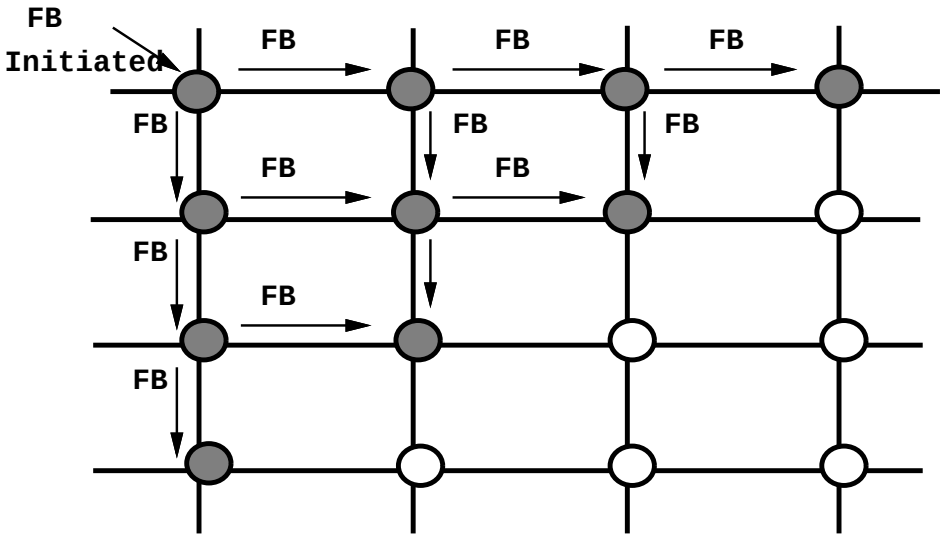


Figure 7.2: The Forward Bulldoze Wavefront

Accessibility of an actor implies accessibility of all its acquaintances but does not necessarily imply the accessibility of its inverse acquaintances. Thus we need to distinguish between GC messages to acquaintances and GC messages to inverse acquaintances. GC messages contain the name of an accessible object, from where further GC messages may be generated. No other information is communicated via a GC message. Messages that are neither *old* messages nor GC messages are *new* messages. Recall that new messages are issued from a node after the GC snapshot has been taken. Thus, prior to the bulldoze waves, only pending/old messages are present in the system.

7.5 Network Clearance Algorithms

We define the *network clearance algorithm* to obtain a consistent snapshot of the system . We will use the algorithm in a distributed GC scheme to clear the network of pending messages without halting the processing of new messages.

There are two main kinds of messages sent out during network clearance; *broadcast messages* and *bulldoze messages*.

Broadcast messages are issued to all the nodes in the system along a fixed path as illustrated in Figure 7.1.

7.5.1 Bulldoze Messages

Bulldoze messages are sent along all dimensions of a network and they traverse every link in the network. Bulldoze messages may be further classified into: Forward Bulldoze (Fbulldoze) messages and Backward Bulldoze (Bbulldoze) messages. They serve as markers on each link. Recall that we assume that a link between any two neighboring nodes is FIFO.

Fbulldoze Phase:

The *start node* of the system issues Fbulldoze messages to the Fneighbors of a node. Other nodes in the network issue a Fbulldoze messages to their Fneighbors only after they receive Fbulldoze message from both their Bneighbors.

When a Fbulldozer message is issued from a node, all pending messages along any Fpath to the node and any pending message in the mail-queues of objects in the current node have been processed and no pending messages will be issued from the node when a PU receives a Fbulldozer message. The processing unit in turn sends the Fbulldozer message to all the objects currently residing on that node. Once the Fbulldoze message has reached the *finish-node* of the network, the forward phase of the *network clearance algorithm* is completed and the backward phase is started.

When a node receives a F-bulldozer message, the following sequence of actions take place:

- Any pending (old) messages in the mail queue of the object are processed.
- The object can no longer issue old messages .
- The object waits for the second F-bulldozer message if this is the first one.
- Once the object has received both F-bulldozer messages, it issues F-bulldozer messages to its F-neighbors.

Figure 7.3: Object Receives the Forward Bulldoze Message

Bbulldoze Phase:

Beginning at the *finish node* of the system, Bbulldozer messages are issued to the Bneighbors of the node. A node in the network can issue a Bbulldozer message to its Bneighbors only after it receives the Bbulldozer message from both its Fneighbors.

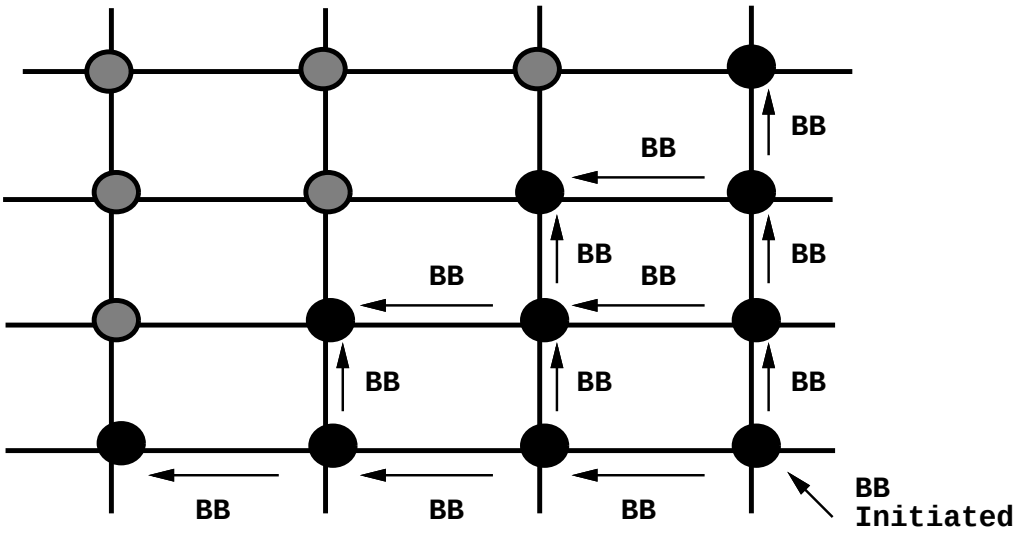


Figure 7.4: The Backward Bulldoze Wavefront

When a Bbulldozer message is issued from a node, all pending messages along any Bpath to the node and any old-message in the mail-queues of objects in the current node have been processed and no pending messages will be issued from the node along a Bpath. The processing unit then sends the Bbulldozer message to every object/actor residing on that node. Once the Bbulldozer message reaches the *start-node* of the network, the backward phase of the *network clearance algorithm* is completed and there are no pending messages left in the network.

The basic network clearance techniques can be applied to many distributed algorithms such as deadlock detection and termination detection. The Broadcast and Bulldoze messages may be inserted at specific points in the execution of the distributed program to achieve consistency. We have adapted the basic network clearance algorithms for use in the hierarchical distributed garbage collection (HDGC) algorithm proposed in the next chapter. The outline of the HDGC scheme is given below. Initially, there are only *old* messages in the system. When GC needs to be initiated, we will inject a broadcast message and a set of bulldoze messages into the system to initialize GC and clear the system of *old* messages. After the bulldoze waves have completed,

When a node receives a B-bulldoze message following sequence of actions take place:

- Any pending (old) messages in the mail queue of the object are processed.
- The object can no longer issue old messages .
- The object waits for the second B-bulldoze message if this is the first one.
- Once the object has received both B-bulldoze messages, it issues B-bulldoze messages to its B-neighbors.

Figure 7.5: Object Receives the Backward Bulldoze Message

only garbage collection and new messages are present in the network. After GC terminates, only *new* messages are present in the system.

The costs of network clearance become prohibitively large as the dimension of the network increases. Therefore, our memory management scheme favors low dimensional networks.

In the next chapter, we will integrate the network clearance mechanisms described in this chapter with the distributed generation scavenging (DGS) algorithm (Chapter 6) to design the Hierarchical Distributed Garbage Collection (HDGC) Algorithm.

Chapter 8

Hierarchical Distributed Garbage Collection

So far, in this thesis, we have developed

- a low overhead GC mechanism (DGS algorithm)
- a technique by which a consistent global state can be captured in a distributed system without halting computation or message transmission in the system (Network Clearance).

In this chapter, we will combine the two schemes to perform garbage collection in a distributed system. The algorithm developed will be called the *Hierarchical Distributed Garbage Collection* algorithm. We will first discuss the algorithm in the context of a two level hierarchy and then show how this can be extended to multiple levels in the hierarchy. Optimizations and extensions for real time or fault tolerant systems will be discussed in the last section.

As we have seen in this thesis, distributed systems may exhibit a hierarchical structure in a number of ways:

- Generalization of parallel computing at the programming language, system design and architectural levels – *A Hierarchical View of Parallel Computation* in Chapter 3.
- Design of a tiered naming scheme and address translation mechanism (see Chapter 5).
- Organization of the processing units for resource management purposes such as scheduling, load balancing and garbage collection.

- Hierarchical sponsorship of speculative work.

We are specifically interested in *hierarchical distributed memory management* . In a hierarchical organization, the distributed (highly parallel) system which consists of many processing units is refined into smaller subunits (also called *regions* or *clusters*) containing fewer processing units. The subunits are in turn partitioned into a smaller subunits and this refinement process is repeated until we have a single processor in each subunit.

The motivation for dividing the computing system into smaller subsystems arises from the need to have an efficient scalable system, the complexity of which can be comprehended. With a hierarchical organization, the system becomes more manageable, because, at every level of abstraction, internal details of the lower levels in the hierarchy can be ignored. Programming abstractions can be developed that make it easier to visualize and manage computations.

From an efficiency viewpoint, we can reduce the communication overhead by scheduling frequently communicating tasks within the same region. By exploiting locality in communication, we can prevent congestion of the network due to excessive traffic and minimize network latency.

The difference between a hierarchical scheme and a single level grouping of the subsystems is that the hierarchical scheme affords an additional level of flexibility. It gives us the ability to tailor the grainsize of the resource management strategies to suit the computational needs and time constraints. It is also possible to choose an optimal resource management strategy for each level in the hierarchy. To illustrate this, let us consider an example in the context of garbage collection. We have claimed earlier that the mark-and-sweep algorithm is wasteful for large memory sizes because the time taken for GC is proportional to the size of the memory being collected. But, mark and sweep is simple and optimal when we have smaller memory sizes, say, cache memory. We would like to choose a garbage collection algorithm depending on the memory size we have to deal with at that point of time. Hierarchical resource management strategies give us the freedom to make that decision at run-time. In a larger system with many processors, we could perform generational garbage collection when we have to deal with large numbers of processors and choose a simpler mechanism for local collection within a processor.

8.0.2 Specification of the Hierarchy

The topmost level: The topmost level of the hierarchy subsumes the entire system; all the processing units in the system constitute a single unit at this level.

The lowermost level: The lowermost level of the hierarchy has a single processing unit per subsystem. This is the point where it is not meaningful to refine the subsystem any further.

The intermediate levels: There may be 0 or more intermediate levels in the hierarchical organization of the distributed system. The intermediate levels in the hierarchy consist of clusters of processing units (called regions or clusters) which are pairwise disjoint and whose union is an element of the next higher level in the hierarchy.

Clusters at the same level in the hierarchy are disjoint. All processors at a lower level in the hierarchy are entirely subsumed in at least one cluster at a higher level in the hierarchy. Every cluster is distinguished by its cluster ID which is structured to reflect the hierarchy.

Given that we have divided the parallel system into suitable hierarchies, we now face the task of managing the hierarchy. This involves partioning, coordination and communication in tasks executing at all levels of the hierarchy. Some of the problems that arise during this process and decisions that must be made are described now.

Distributed vs. Dedicated Cluster Managers: The responsibility of coordinating task execution within a cluster in a hierarchical system is assigned to the *cluster manager* (CM), which could be a single processing unit or a group of processing units. We call such cluster managers *dedicated* and *distributed* cluster managers respectively.

With a *distributed cluster manager*, the task of managing the resources of the cluster are distributed among the among all or some of the processing units within that cluster. Processors responsible for cluster management are called *cluster manager processors* or CMPs. All the PUs in the distributed cluster manager can be equally important, i.e, have the same priority or a *master cluster manager* can delegate cluster management duties to other CMP's. Here is a possible scenario: One CMP is responsible for communication activities and all incoming and outgoing communication to and from the cluster will have to pass through this PU. Therefore, it must be capable of making routing and buffering decisions. Another CMP handles the job of address translation, while other CMPs take care of load balancing, migration and scheduling within the cluster. Note that

Distributed Cluster Manager	*Dedicated Cluster Manager*
Need to assign responsibilities to individual CMPs in the cluster.	Need to choose a single dedicated cluster manager.
Complex because many processing units are involved.	Simple and straightforward because a single PU is involved.
No centralized bottleneck.	Bottleneck for services at the dedicated cluster manager.
Better for fault tolerance because resource management tasks will not be severely disrupted if one processing unit fails.	Not fault tolerant. If the cluster manager fails, activities will be severely disrupted.
Task reallocation overhead on failure of a CMP.	Re-election overhead on CM failure.

Table 8.1: Dedicated vs. Distributed Cluster Manager

the authority of a CMP's activities are limited to that cluster. Every cluster manager reports to the upper level cluster manager which in turn makes more global decisions regarding resource management. This could be information like cluster load level (for load balancing) or paging activity (for locality measures).

With a *dedicated cluster manager*, a single processing unit manages the resources among processes in the cluster. As only one unit has to perform the tasks of intercluster and intracluster communication in addition to scheduling, translation and other resource management duties, these tasks are serialized. Failure of this single unit will disrupt activity in the system. The dedicated cluster manager could be fixed (the same processor is always the cluster manager). Alternately, the dedicated cluster manager could change between tasks in which case a new cluster manager has to be elected for every task. Factors that must be considered when choosing a cluster manager include the average physical proximity to other PUs in the cluster as well as proximity to other cluster managers. We have summarized the differences and overheads that arise in dedicated and distributed cluster managers in Table 8.1.

Introducing additional naming levels: The tiered object naming scheme described in Chapter 5 has three levels of naming for a 2-tiered hierarchy. The *distributed global nametable* and *local nametable* were responsible for address translation between the two levels. In general, when a distributed system has been organized as a *n-level* hierarchy, we need an *n-tiered* naming scheme and $n-1$ nametables to translate between the n levels; a global nametable, $n-3$ cluster nametables, and a local nametable on every processing unit. Each level in the hierarchy can be thought of as an independent namespace with the tables to assist in name translations across virtual address spaces. With too many levels in the hierarchy, namespace management will constitute a large overhead. By sharing address spaces over levels in the hierarchy, we could reduce the number of naming levels and the overhead of name management.

Introducing regional locality: With the introduction of hierarchies or additional virtual address spaces, we have additional levels of locality to deal with. Every processor could receive remote messages of two kinds: Messages from a remote processor within the cluster (intra-cluster messages) at a particular level in the hierarchy and messages from a remote processor outside the cluster (inter-cluster messages). While defining locality, the level in the hierarchy must also be well defined because inter-cluster messages at one level could be intra-cluster messages at a higher level in the hierarchy.

Message transmission overheads: Messages arriving from a PU in a different cluster have to be routed through the cluster managers at both ends. The sending cluster manager maintains a record of messages sent out of the cluster and the receiving cluster manager maintains a record of external messages received by any processing unit in the cluster. We define the *hierarchical distance* between two subregions (or processing units) to be the number of levels that must be ascended in the hierarchy before the two subregions(or processing units) belong to the same region. The message transmission overheads are directly proportional to the *hierarchical distance* of the communicating processing units as routing must occur through all the cluster managers at different levels.

8.1 Hierarchical Distributed Garbage Collection

In Chapter 6, we proposed the DGS (Distributed Generation Scavenging) algorithm. As a prerequisite for the correct execution of the algorithm, we assumed that the system is free of pre-GC messages when the DistributedScavenge phase was initiated. The DGS algorithm also assumed that there is a clear line of temporal demarcation between DistributedScavenge Phase termination and LocalClear Phase initiation. Using network clearance, it will be possible to force any messages in transit to reach their destination and continue further processing after obtaining a consistent picture of the entire system for garbage collection. By combining the two schemes we have developed with the hierarchical resource management scheme discussed in the previous section, we have developed a hierarchical generational distributed garbage collection (HDGC) algorithm. We will first present the HDGC Algorithm in the context of a two level hierarchy, i.e., global and local collections (there are no intermediate levels in the specification of the hierarchy).

8.1.1 Outline of The HDGC Algorithm

The HDGC algorithm has 5 distinct steps:

1. Pre-GC Sweeps
2. DistributedScavenge Phase
3. Interphase Synchronization
4. LocalClear Phase
5. Post-GC Cleanup

We briefly outline the task that each phase is designed to perform and then explain each step in detail.

Prior to GC-initiation the following conditions are satisfied:

- All nodes in the system are untouched.
- All messages in the system are *old* messages, i.e, there are no GC, bulldoze or broadcast messages.

1. **Pre-GC Sweeps:** In this phase, we record a snapshot of the distributed system using the NW clearance algorithms. Two kinds of messages are injected into the system - *broadcast* messages and *bulldoze* messages.

   Broadcast messages are sent out in the system to announce the start of GC. In addition to informing the processing units of a forthcoming GGC, the links of the network must be cleared of all messages in transit. Any pre-GC messages remaining in the system must be scanned for new acquaintances. The bulldoze messages are responsible for all these operations.

   Therefore, after the pre-GC sweeps, but prior to the DistributedScavenge phase the following conditions are satisfied:

   - All nodes in the system are untouched.
   - No messages in the system are *old* messages.

2. **DistributedScavenge Phase:**

   - Run the distributed scavenge phase, making the accessible or potentially accessible objects in ScavengeSpace *touched*.
   - Detect termination of the DistributedScavenge phase by back-propagation messages.

3. **Interphase Synchronization:**

   The purpose of this phase is to inform every PU in the system that the DistributedScavenge phase of GGC has completed. After receiving this message, every PU can start the LocalClear Phase.

4. **LocalClear Phase:** There are two tasks that are performed in this phase:

   - The LocalClear Phase Initiation Broadcast - which is a message to every node to initiate the LocalClear Phase of GGC.
   - LocalClear Phase Completion Backwave that determines the termination point of the LocalClear Phase.

5. **Post GC Broadcast:**

These are messages that tell every node in the system that the LocalClear Phase of has been completed on every other node and that the *Global-GC-is-on* flag can be turned off.

8.1.2 Assumptions of the HDGC Algorithm

The HDGC algorithm makes the following assumptions about the topology of the system and behavior of the network:

- The system is fault-tolerant i.e. the communication and processing subsystems are not subject to faults.
- Migration of objects during any of the GC Phases is not permitted.
- There is no notion of a global clock in the distributed system.
- The network is a direct network with infinite message buffers. For the purposes of analysis and illustration, we will assume that the topology of the system is that of a multiprocessor mesh interconnection (2 dimensional mesh interconnection) as in Figure **??**.
- The routing mechanism in the network is a progress-only routing mechanism with message order preservation across a single link.

8.1.3 The HDGC Algorithm

We now describe the steps involved in a two level HDGC in detail. The terminology presented in the earlier chapters is used in this section.

Step 1: Pre-GC Sweeps

When a GC phase is initiated, all acquaintances that are currently in the network have to be accounted for. The network and the each actor's mailqueue are cleared of *old* messages by means of the forward and backward bulldoze sweeps explained earlier. The objective of the Pre-GC Sweep is to record a consistent snapshot of the global state on every node. In order to do this, we need to guarantee that both *local consistency* and *global consistency* have been achieved. Every node needs a point of reference in time with respect to which it determines the accessibility or inaccessibility of actors in its memory. Once a processing unit has established

this point and recorded the non-garbage actors on its node, we have attained *local consistency*. *Global consistency* is a point in time when all nodes participating in the GC have agreed on a particular state of the distributed system. Global consistency is required at different points in the execution of the HDGC algorithm. In the Pre-Gc sweep phase, it is exhibited when all the PUs are aware of the fact that GC has been initiated. Every node records information used in determining actor accessibility; namely, the acquaintance, inverse acquaintance and lost acquaintance lists which maintain a conservative approximation to the snapshot state. The conservative approximation includes acquaintances 'lost' on a processor P due to processing activity after initiation of a GC and before termination of that GC on P. The following scenario illustrates the need for a *lost acquaintance list*.In the HDGC scheme, processing activity can continue simultaneously with GC in the system. All acquaintance and inverse acq. lists are traversed at the time of GC.

Consider an acq. A in the acq. list of an actor B. Let C be an entry in the acq. list of A. By virtue of its being an acq., A could have communicated either its mail address or the mail address of C to another actor. After obtaining the consistent snapshot, we rely on the original acq. A to continue the scavenging process to reach C. However, due to changes in processing activity, the entry A may be removed from the acq. list of B prior to receiving the GC_{acq} message. Now if the reference A is destroyed, the reference C has not been captured anywhere else in the system and is wrongly collected as a garbage actor. The purpose of the lost acquaintance list is to maintain the reference to A and use it during the scavenge process .

The lost acquaintance list at a node can be destroyed after the completion of the ongoing GC. If the node is still accessible during the next GC, the reference will be captured by one of its acquaintances. However, if this acquaintance becomes a garbage actor, it will be collected in the next GGC (or the next collection at the corresponding level in the hierarchy).

Three functions in the NCA algorithm that are called during the Pre-GC sweep phase are Node-Broadcast Function, the Forward Bulldoze Wave, and the Backward Bulldoze Wave, in that order.

- Pre-GC Broadcast – This is a message broadcast to every PU in the system (hierarchy). It is an indication that a GC is about to begin and that the network will be cleared of all *old* messages. Once a PU has received the pre-GC sweep message, it traps all the acquaintances that it currently has on its acquaintance list, all mail addresses that are

currently in its mail queue as well as addresses communicated via *old* messages. Any acquaintance that is received as a result of mutation is placed in the acquaintance list and this is used to determine reachability when the corresponding actor is touched in the garbage collection process. During this entire phase, all messages released into the network are still tagged *old*. Thus, it is possible that an acquaintance is saved in more than one node, but this does not violate correctness. In fact, it may not be possible to avoid it in order to guarantee termination.

- Forward Bulldoze Wave – Clears the system of all *old* messages by ensuring that nodes will no longer transmit any more *old* messages. The forward bulldoze phase begins at the *start* node of the system and travels only along *F-paths*. When a processing unit receives a F-bulldoze message, it saves all the acquaintances and inverse acquaintances (that have not yet been saved) including any new acquaintances on its mail queue. Any messages communicated subsequently from this PU along either an F-path or B-path will be tagged *new*. The *new* tag on a message informs the recipient of the message that any addresses or acquaintances that are communicated in the message have already been accounted for. The recipient does not examine the message for communicated mail addresses.

  After an object, say A, has crossed the F-bulldoze phase it can send only one kind of message, i.e. *new* messages. But, it may receive *old* messages from an actor H that had/has not yet received the F-bulldoze message (see Figure 8.1). Any references communicated from H to A have not been accounted for in H. Therefore, when A receives the *old* message, it will save any new addresses within the message and process the message. Outgoing messages due to processing will now be tagged *new*. On the other hand, if A receives a *new* message, it can continue processing the message as usual and acquaintances do not have to be accounted for prior to GC, i.e, somebody else is responsible for holding the reference. When a processing unit finishes processing the F-bulldoze messages, it sends F-bulldoze messages to both its F-neighbors. The wavefront propagates as illustrated in Figure 7.2.

- Backward Bulldoze Wave – Clears any remaining unprocessed old messages left in the system.

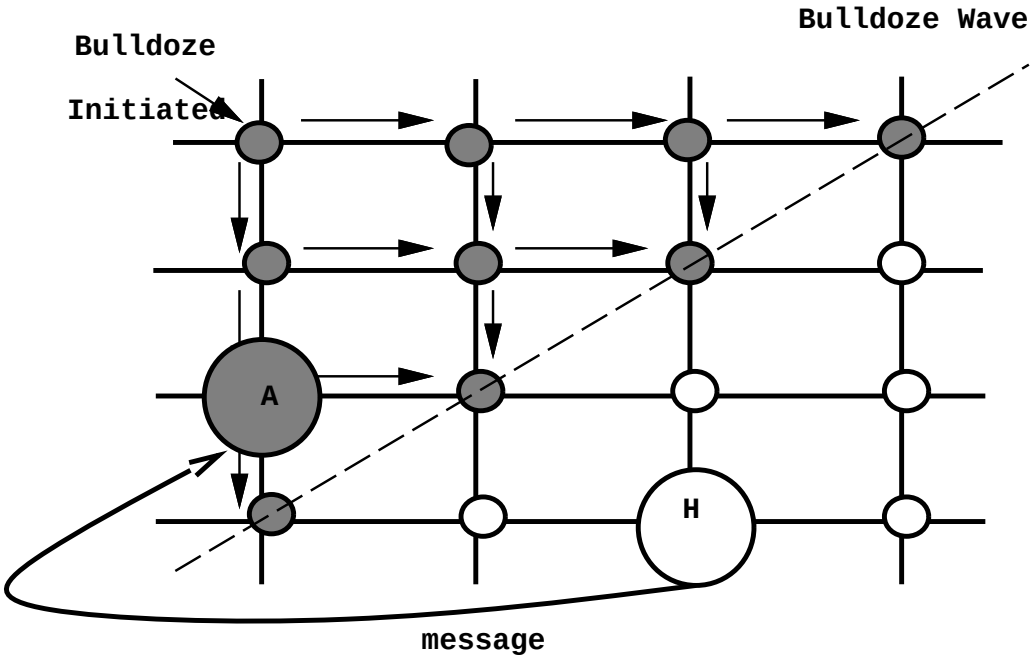


Figure 8.1: The Bulldoze Wavefront in Progress

When a processing unit finishes processing the B-bulldoze messages, it sends B-bulldoze messages to both its B-neighbors. The wavefront propagates as illustrated in Figure 7.4.

The following paragraphs explain actions taken by touched and untouched objects in the DGS algorithm when they receive a particular message. (also see Table 8.2).

Untouched Objects

Processes an *old* message: If the pre-GC phase has completed, the object has received the bulldozer messages, it saves any mail addresses communicated in the message and continues processing issuing *new* messages. However, if a GC phase is not currently underway, the object continues with the processing of the *old* message, issuing more *old* messages.

Receives a GC_{acq} message: it recognizes that a GGC phase is going on. It makes a request to run the GGC thread on its local processor that performs the following sequence of steps:

- Sets the *object-status* field of the object to 10 or touched.
- Sends out GC_{acq} messages to actors in its acquaintance list and GC_{invacq} messages to actors in its inverse acquaintance list.

Message Type	*Untouched Object*	*Suspended Object*	*Touched Object*
Old Message	Remains Untouched, Saves Acquaintances	Not Applicable	Not Applicable
GC_{acq} **Message**	Becomes touched, Issues further GC_{acq} and GC_{invacq} messages	Becomes touched, Issues further GC_{acq} and GC_{invacq} messages	Remains Touched
GC_{invacq} **Message**	If active, becomes touched and issues more GC messages. Else, becomes suspended.	Remains suspended	Remains touched
New Message	No acquaintances are saved. Sends out more new messages.	Becomes active and touched. More GC messages.	Remains touched

Table 8.2: Actions Taken by Objects Receiving Tagged Messages

- Continues further processing sending out only *new* messages.

Receives a GC_{invacq} message: If the object is currently blocked, the *object-status* field of the object is set to suspended (01). However, if the object is currently active, the *object-status* field is set to touched (10) and the object issues GC_{acq} messages to its acquaintances and GC_{invacq} messages to its inverse acquaintances.

Receives a *new* message: Continues processing and issues more *new* messages.

Touched Objects

Receives an *old* message: This is not a possible situation, because if the object has been touched, it has participated in the DistributedScavenge phase and a precondition of the DistributedScavenge Phase is the absence of any *old* messages in the system.

Receives a GC_{acq} message: The *object-status* field of the object continues to remain touched.

Receives a GC_{invacq} message the object continues to remain touched.

Receives a *new* message: Continues processing and issues more *new* messages.

We have explained the process of touching objects and

Interphase Synchronization

The purpose of this phase is to inform every PU in the system (hierarchy) that the DistributedScavenge phase has completed. It is accomplished by means of a broadcast message that is initiated at the start node. The contents of this message indicate the completion of the DistributedScavenge phase. After receiving this message, every PU can start the LocalClear Phase which is a local phase.

LocalClear Phase

LocalClear Phase is a local phase on every PU. Once LocalClear Phase has been initiated on a PU, it perform the following steps:

- Clears garbage objects in its local memory.
- Clear invalid global nametable entries.

A LocalClear Phase Completion Backwave is initiated which determines the termination point of the LocalClear Phase. LocalClear Phase Completion Backwave is a bulldoze message that spans the entire system, to detect termination of LocalClear Phase on every processing unit.

Post GC Broadcasts

Post GC Broadcast is a network broadcast wave that tells every node that the LocalClear Phase of has been completed on every node. Every node can now turn off its local *GC-is-on* flag. Also the interpretation of the *future-tag* bits can be flipped at this point, i.e, *future-tag* = PSS-tag

8.2 Extensions and Optimizations to the HDGC Algorithm

8.2.1 Multiple Levels in the Hierarchy

We will now describe the process of hierarchical distributed garbage collection. First, we state some terminology and rules that have to be followed while performing distributed GC using the HDGC algorithm with a multi-level hierarchy.

Every HDGC is assigned a *HDGC number* which is equal to the level in the hierarchy at which HDGC is being initiated. Before a HDGC is begun, a message is broadcast to all the

nodes included in the subregion of the hierarchy. This sets a flag on every PU in the region which indicates the HDGC number.

Rules for hierarchical DGC:

- When a HDGC is active at a higher level in the hierarchy, no HDGC can go on concurrently at the lower levels in the hierarchy.

  Lemma:

  When a HDGC is active at a higher level in the hierarchy, no HDGC can go on concurrently at the lower levels in the hierarchy.

  Proof: Every HDGC is assigned a HDGC number which is equal to the level in the hierarchy at which HDGC is being initiated.

  Let's assume that a HDGC is attempted on a node when another HDGC is already active on the node. Let us designate the collections as HDGC1 and HDGC2 respectively; HDGC1 represents the collection that is already active and HDGC2 is the requested HDGC. Let *HDGC1-number* and *HDGC2-number* be the HDGC numbers of the two collections respectively. We could have one of two cases:

  Case 1: *HDGC1-number* > *HDGC2-number* – HDGC1 has priority; note that this HDGC message is blocked as soon as it is started, because th start-node of HDGC2 is a part of HDGC1. Therefore, HDGC2 is aborted and the broadcast for HDGC 2 is not initiated.

  Case 2: *HDGC1-number* < *HDGC2-number* – HDGC2 is completed before the HDGC1 message is broadcast to the next node in the region. If a higher level broadcast arrives at a node when a lower level HDGC is going on, the broadcast message is not transmitted to the subsequent cluster or region in the hierarchy. If a HDGC is going in a region B when a HDGC message from region A arrives, the HDGC broadcast from A is not transmitted to C until the HDGC on region A completes.

The promotion policies applied to different levels in the hierarchy are independent of each other.

8.2.2 Fault Tolerance in the HDGC Algorithm

In order to make a system fault tolerant, we must be able to rollback to a previous consistent state whenever a failure occurs. Fault Tolerance assumes the existence of a checkpointing facility in the system. Network Clearance techniques can be used to obtain the checkpoint. Whenever a GGC (HDGC) is called, the snapshot of the previous checkpoint is used to determine accessibility. Assumptions for HDGC are different from the DGC scheme in that all failures except Byzantine failures are permissible.

8.2.3 Abstracting Local Garbage Collection

It is impractical to call a GGC whenever any allocation request is not satisfied. Intermediates, temporaries and locally allocated structures may occupy an a large percentage of local memory for some applications. Local collection allows us to reclaim only objects that have an existence only within the local PU like temporaries and intermediates. In order to support local GC, every processing unit has a receptionist table that holds all the local actors that are referenced from outside. An actor name is added to the receptionist table whenever the mail address of the actor is passed out of the processor. The entries in the receptionist tables are a part of the root set during a local GC. As a further optimization to the HGC algorithm, local objects (entries not in the receptionist table) can be ignored in a non-local hierarchical collection. Local objects can be reclaimed during local collection; this overhead need not be encountered during a non-local collection. An efficient version of the HGC algorithm would therefore need to use only the receptionist tables during a collection.

8.3 Discussion

In this chapter, we have approached the problem of distributed resource management from a pragmatic angle. We have looked at mechanisms to efficiently manage a large system of active computational entities. As a highly parallel system scales up in size, we would like the resource management schemes to scale up as well. Hierarchical RM allows us to deal with the complexity of systems as they scale up in size. The HDGC algorithm proposed in this chapter is a conservative, generation based distributed storage reclamation mechanism that seems well suited to scalable systems. There are a number of issues that have to be tackled in extending

the HDGC algorithm to exhibit realtime response and fault-tolerant behavior. We have yet to analytically model the tradeoffs that arise with an increase in the hierarchical depth of the system.

In Part 3 of the thesis, we get into more formal issues in modeling the hierarchical actor system and proving the correctness of the HDGC algorithm.

Part 3: A Formal Model

In this part of the thesis, we develop an Abstract Actor Machine (AAM), which is a formal representation of an actor based multicomputer system. We intend the AAM to serve as a general framework that can be suitably improvised to model any aspect of the actor based distributed system. We have shown how the AAM can be used to model the memory management subsystem. Furthermore, we found that with this model we can effectively portray the globality of the resource management problem in a high performance system. We use the AAM to determine and prove the correctness of the HDGC algorithm.

Chapter 9

A Theoretical Framework for Memory Management

In any formal model, the elements of the system being modeled are represented as abstract entities with operations and relations defined on them. Examples of such entities in a distributed system are: (1) processes, (2) processing units, (3) the communication network, (4) message routing, injection, reception and processing assumptions. Algorithms executed by the system are mapped on these entities. This allows us to test the completeness and correctness of the algorithmic specification.

In this chapter, we will design an abstract actor machine ($\mathcal{AAM}$) on which we will impose a transition semantics to formalize the GC process. In the abstract actor system, we classify actors into

1. base-actors (application) actors
2. meta-actors (system) actors

Meta-actors can access information from base-actors that other base-actors cannot access. They can modify internal information like the state and tag bits in the header of an actor and serve as managers of base-actors. At present, meta-actors are no more than a resource management facility in actors. With both base-actors and meta-actors, message passing is a fundamental mechanism required for any kind of information transfer, for example, information access or spawning of new tasks. We propose to have one kind of meta-actor for each resource

management duty; i.e. meta-actors for GC, meta-actors for prioritization and so on. Not only does this support encapsulation of the resource management facilities, it gives us a framework to deal with system management as well as application management within a single unified model — Actors. In this chapter we will deal with only one kind of meta-actor, the GC meta-actor. Therefore, we will use the terms meta-actor and GC meta-actor interchangeably.

9.1 Model of a Multicomputer System

An informal description of a distributed model has a collection of autonomous processing entities that do not physically share a common address space, have no notion of a global time and that communicate via message passing. A finite set of such processes is mapped on the set of processing units. We describe the basic model of a multicomputer system from various angles.

9.1.1 The Computation Model

The computation model of the system consists of actors that can send and receive messages: The following actions may take place during computation:

- Change of process state
- Inter-actor communication of messages.
- Creation of new objects

9.1.2 The Memory Model

The memory model of the system consists of a uniform actor space distributed across the processing entities. Every entity in the system can access every other entity it knows the address of, i.e., every actor on its acquaintance list.

9.1.3 The Communication Model

The network is a fully connected graph i.e. all the PUs in the system are reachable through the network. The vertices of the graph represent processing units and the edges of the graph represent communication channels. Properties of the communication channel:

1. The channel has infinite buffers.
2. Message transfer delay is arbitrary but finite.
3. Asynchronous communication.
4. Routing mechanism is progress-only (adaptive).

9.1.4 Timing Considerations

There is no notion of a global clock in a truly distributed system. However, we can assume that a meaningful global state of the system can be recorded by having each processing unit (PU) present a consistent picture to the entire system.

9.1.5 System Considerations

A multicomputer system configuration has two attributes:

1. **The processing state** which is the set of states of all the processes in the system.
2. **The communication state** which is the sum total of all the message occurrences in the system.

Execution of tasks in the abstract machine is represented by transitions in the system. An event in the above model of a multicomputer system is associated with the following:

- the state of each process prior to the execution of that event
- the state of each process after the execution of that event
- the state of each channel along which messages travel.
- the contents of the message sent or received along a channel
- the new processes created by the execution of the event

9.2 Terminology and Notation

We now define terminology that is used in the Abstract Actor Machine ($\mathcal{AAM}$) as intuitive notions in the multicomputer system. Corresponding mathematical definitions will appear as we develop the $\mathcal{AAM}$.

Multicomputer System: The infrastructure on which the application program executes. This includes the processing as well as communication capabilities of the system.

Base-Actors: Actors that are created as a result of the execution of the application program.

Meta-Actors: Actors that exist in the system as part of the resource management system. These actors manage the base-actors and they can access the protected information within a base-actor that other base-actors cannot access.

Channel: Unidirectional links of the network that communicate information. They connect one node in the multicomputer system to another node.

Multicomputer Configuration: A snapshot of the sum total of the entire state of the system that encompasses both the computation and communication activities in the system.

Meta-Actor Configuration: A snapshot of the sum total of all meta-actors currently under consideration, which presents the computational aspect of a multicomputer system as a whole.

Base-Actor Configuration: A snapshot of the sum total of all base-actors currently under consideration, which presents the programmatic aspect of a multicomputer system as a whole.

Channel Configuration: A snapshot of the sum total of all messages currently in transit, which presents the commmunication aspect of a multicomputer system as a whole.

Multicomputer Initial Configuration: The configuration of the multicomputer system prior to any GC or in between two consecutive GCs in the system.

Meta-Actor Initial Configuration: The configuration of meta-actors in the system prior to any GC or in between two consecutive GCs in the system.

Base-Actor Initial Configuration: The configuration of base-actors in the system prior to any GC or in between two consecutive GCs in the system.

Channel Initial Configuration: The configuration of the channels (messages in the channels) prior to any GC or in between two consecutive GCs in the system.

Multicomputer Transition: A transition function that operates on a multicomputer configuration and returns a new multicomputer configuration.

Preconditions: Conditions that the system must satisfy prior to the defined point in time, or prior to the execution of a specified event.

Postconditions: Conditions that the system must satisfy subsequent to the defined point in time, or subsequent to the execution of a specified event.

We begin by giving a formal definition of the abstract actor machine, the actor model implemented on a multicomputer system. Subsequent to this, an initial configuration definition of the abstract actor machine prior to GC is presented. Using the notation and terminology described earlier, we then define a transition semantics for the process of hierarchical garbage collection.

9.3 Formal Configuration Definition of the $\mathcal{AAM}$

9.3.1 Multicomputer System

The physical multicomputer system has a set of nodes ($\mathcal{N}$) representing the processing units and a set of physical channels that connect the nodes. The abstract entities mapped on the hardware are actors: both base-actors and meta-actors. Every node has a single GC meta-actor that resides on it. As we are concerned with only GC at this point, the name of the GC meta-actor is equivalent to the node name (n). A multicomputer configuration consists of the configuration of meta-actors in the multicomputer system, the configuration of the base-actors in the system and the channel configuration of the channels in the system. Let $\mathcal{M}$ be the set of all possible multicomputer configurations.

- Multicomputer Configuration:

  The set of all possible multicomputer configurations is a 3-tuple of the set of all possible meta-actor configurations ($\mathcal{A}_m$), the set of all possible base-actor configurations ($\mathcal{A}_b$) and the set of all possible channel configurations($\mathcal{C}$).

$$\mathcal{M} = \langle\ \mathcal{A}_m, \mathcal{A}_b, \mathcal{C}\rangle$$

- Multicomputer Transition:

  A multicomputer transition or $\mathcal{T}_\mathcal{M}$ in the system operates on a $\mathcal{M}$-configuration ($\mathcal{M}$) and returns a set of new possible $\mathcal{M}$-configurations.

$$\mathcal{T}_\mathcal{M} = \mathcal{M} \longrightarrow \mathcal{P}\ (\mathcal{M})$$

9.3.2 Meta-Actor System

As described earlier, the processing state of the system is represented by the actors. Actors are of 2 kinds: Base-actors and meta-actors. A meta-actor is referred to by a name. Note that this is equivalent to the name of the node (n) on which the meta-actor resides. Each meta-actor has an internal configuration. $\mathcal{A}_m$, the set of all possible meta-actor configurations ($mAic$) is a set of maps from the set of meta-actor names into the set of corresponding meta-actor internal configurations.

$$\mathcal{A}_m = \mathcal{N} \longrightarrow mAic$$

- Meta-Actor Internal Configuration:

  A meta-actor internal configuration ($\alpha_m \in mAic$) embodies the internal state of the actor ($\mathcal{S}_{m\alpha}$), the set of acquaintances and inverse acquaintances of the meta-actor, the current mail queue of the meta-actor (χ) and the behavior (β) of the meta-actor in the system. Every meta-actor knows every other meta-actor. Therefore, the acquaintance list and inverse acquaintance list of a meta-actor are trivially the set of nodenames ($\mathcal{N}$). The behavior (β) of a GC meta-actor implements the GC algorithm and this is not altered during the course of execution.

$$\alpha_m = \langle\ \mathcal{S}_{m\alpha}, \mathcal{N}, \mathcal{N}, \chi, \beta\rangle$$

As the only parameters of variation in a meta-actor configuration are the meta-actor state and the mail queue of the meta-actor, we will reduce the representation of the meta-actor configuration (α_m) to

$$\alpha_m = \langle\ \mathcal{S}_{m\alpha},\ \chi \rangle$$

The other parameters are implicit.

- Meta-Actor State:

  The meta-actor internal state, $\mathcal{S}_{m\alpha}$, contains the following information:

  $$\mathcal{S}_{m\alpha} = \langle\ \rho,\ \pi,\ Pos,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} \rangle$$

  - ρ: indicates whether the meta-actor is the root actor of the ongoing hierarchical GC.
  - π: indicates whether the meta-actor is involved in a GC and if so, in which phase.
  - Pos: marks the position of the node on which the meta-actor resides in the network.
  - $\sharp\mathcal{GC}$: Number of GC messages sent.
  - $\sharp\mathcal{GC}_{ack}$: Number of GC_{ack} messages received.
  - $\sharp\mathcal{GC}^{-1}$: Number of GC^{-1} messages sent.
  - $\sharp\mathcal{GC}^{-1}_{ack}$: Number of GC^{-1}_{ack} messages received.
  - $\mathcal{NC}$: Network clearance messages being processed

  We now explain the values each field can take.

  $\rho = 1$: indicates that the meta-actor is the root object for the ongoing collection.

  $\rho = 0$: indicates that the meta-actor is not the root object for the ongoing collection.

  The GC phase field indicates whether the actor is involved in a GC and if so, in which phase (π):

  $\pi = 0$: The actor is in phase0 of GC, no GC is going on.

  $\pi = 1$: The actor is in phase1 of GC.

  $\pi = 2$: The actor is in phase2 of GC.

The position (Pos) field is used to distinguish the start and finish nodes in the system from the intermediate nodes. As there is only one meta-actor on a node, we encode this information into the meta-actor state. If the node on which the meta-actor exists is the start-node, then $Pos = S$. If the node on which the meta-actor exists is the finish-node, then $Pos = F$. Otherwise, $Pos = M$. The fields $\sharp\mathcal{GC}$, $\sharp\mathcal{GC}_{ack}$, $\sharp\mathcal{GC}^{-1}$, $\sharp\mathcal{GC}^{-1}_{ack}$are used to keep track of the number of GC messages sent and acknowledgments received during the GC process. The network clearance messages, $\mathcal{NC}$, are the broadcast, forward bulldoze and backward bulldoze messages. These messages are initiated at and communicated between meta-actors residing on the nodes of the system. It is therefore appropriate to include these messages as part of the meta-actor system. The values that the field $\mathcal{NC}$can take are

- $\mathcal{NC} = \phi$: the node is not processing any $\mathcal{NC}$messages.
- $\mathcal{NC} = br$: the node is processing a broadcast message.
- $\mathcal{NC} = fb$: the node is processing a forward bulldoze message.
- $\mathcal{NC} = bb$: the node is processing a backward bulldoze message.

9.3.3 Base-Actor System

An actor is referred to by an actor name. Let $b\alpha$ be the actor name associated with a base-actor. Each actor has an internal configuration α_b. $\mathcal{A}_b$ or the set of all possible actor configurations is the set of finite maps from actor names ($bAct$) into the set of corresponding actor internal configurations ($bAic$).

$$\mathcal{A}_b = \text{Fmap } (bAct, bAic)$$

- Base-Actor Internal Configuration:

  A base-actor internal configuration ($\alpha_b \in bAic$) includes the internal state of the actor, the set of acquaintances and inverse acquaintances of the actor, the current mail queue of the actor and the behavior of the actor in the system and the channel configuration of the channels in the system.

  We use $\overline{x}$ to refer to finite sets of elements from a category X, i.e, if x ranges over X, then $\overline{x}$ ranges over finite sets from X.

$$\alpha_b = \langle\ \mathcal{S}_{b\alpha},\ \overline{Acq},\ \overline{Acq^{-1}},\ \chi,\ \beta \rangle$$

- Base-Actor State:

  The actor state $\mathcal{S}_{b\alpha}$ contains the following information:

  $$\mathcal{S}_{b\alpha} = \langle\ \mathcal{OS},\ \delta,\ \lambda,\ \mathcal{CC} \rangle$$

  - $\mathcal{OS}$: the object-status.
  - δ: the dynamicity of the actor.
  - λ: the logical memory area in which the object resides.
  - $\mathcal{CC}$: the copy count.

  The object-status field ($\mathcal{OS}$) indicates the status of the object: 1.e., untouched objects ($\mathcal{OS} = u$), touched objects ($\mathcal{OS} = t$) and suspended objects ($\mathcal{OS} = s$).

  The dynamicity (δ) of the object indicates whether the object is currently active or blocked.

  $\delta = a$: the object is currently active
  $\delta = b$: the object is currently blocked

  The object could reside in different areas of logical memory (λ):

  - $\lambda = o$: the object is in Oldspace
  - $\lambda = p$: the object is Persistent
  - $\lambda = pss$: the object is in Past Survivor Space
  - $\lambda = fss$: the object is in Future Survivor Space
  - $\lambda = ns$: the object is in New Space

  The copy-count field ($\mathcal{CC}$) is the number of scavenges an object has to survive before it is promoted into Oldspace.

9.3.4 Channel System

- Channel Internal Configuration:

  Let γ_n be the name associated with a channel. $\mathcal{C}$ or the set of all possible channel configurations is the set of maps from channel names (C_n) into the set of all channel internal configurations (Cic).

  $$\mathcal{C} = C_n \longrightarrow Cic$$

  The channel internal configuration of a channel (γ) includes the set of messages on the channel, the source and destination of the channel.

  $$\gamma = \langle \overline{\mu}, Source, Destination \rangle$$

- Messages:

  Messages themselves are tuples that include a message state ($\mathcal{S}_\mu$), the message information (ι), the source (σ) and destination (θ) of the message.

  $$Message = \langle\ \mathcal{S}_\mu, \iota, \sigma, \theta \rangle$$

  The message state ($\mathcal{S}_\mu$) can be further refined as

  $$\mathcal{S}_\mu = \langle\ \tau, \overline{b\alpha} \rangle$$

  The message tag field (τ) indicates whether the message is an old message ($\tau = o$) or a new ($\tau = n$) message. The second field ($b\alpha$) contains any mail addresses communicated in the message. If the message is *old*, the mail addresses communicated ($b\alpha$) may not have been captured in the system. So, if an old message is received during the preGC phase, all mail addresses communicated in messages have to be saved. If the message is tagged *new*, however, the mail addresses communicated need not be saved during a GC.

9.4 Initial Configuration of the $\mathcal{AAM}$

9.4.1 Multicomputer System

The set of all possible initial multicomputer configurations ($\mathcal{M}^i$) with respect to garbage collection is a triple of the set of initial meta-actor configurations (α_m^i), the set of initial base-actor configurations (α_b^i) and the set of initial channel configurations (γ^i).

$$\mathcal{M}^i = \langle\ \alpha_m^i,\ \alpha_b^i,\ \gamma^i \rangle$$

9.4.2 Meta-Actor System

The initial meta-actor configuration states

- Number of GC messages and GC_{ack} messages are 0.
- Number of GC^{-1} messages and GC_{ack}^{-1} messages are 0.
- All actors are in GC-Phase0.
- There are no network clearance messages in the system.

$$\alpha_m^i = \langle\ \mathcal{S}_m^i,\ \chi \rangle$$

where

$$\mathcal{S}_m^i = \langle\ \rho,\ \pi, Pos,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}_{ack}^{-1},\ \mathcal{NC} \rangle\ ,$$

$$\pi = 0,$$

$$\sharp\mathcal{GC} = \sharp\mathcal{GC}_{ack} = \sharp\mathcal{GC}^{-1} = \sharp\mathcal{GC}_{ack}^{-1} = 0,$$

and

$$\mathcal{NC} = \phi, \text{ i.e., empty set.}$$

9.4.3 Base-Actor System

The initial base-actor configuration states

- Every actor is untouched.
- None of the actors are in FutureSurvivor Space, i.e., actors may reside in persistent space, Oldspace, PastSurvivor Space.

$$\alpha_b^i = \langle\ \mathcal{S}_b^i,\ \overline{Acq},\ \overline{Acq^{-1}},\ \chi,\ \beta \rangle \text{ where}$$
$$\mathcal{S}_b^i = \langle\ \mathcal{OS} = u,\ \delta,\ \lambda,\ \mathcal{CC} \rangle \text{ and } (\ \lambda = p \text{ or } \lambda = o \text{ or } \lambda = pss \text{ or } \lambda = ns).$$

9.4.4 Channel System

The initial channel configuration states

- All messages in the system are tagged *old*.

$$\gamma^i = \langle\ \mathcal{S}_\gamma^i,\ \overline{\mu},\ Source,\ Destination\ \rangle \text{ where}$$
$$\overline{\mu} \text{ is such that for every } \mu \in \overline{\mu},\ \mu = \langle \mathcal{S}_\mu^i,\ \iota,\ \sigma,\ \theta \rangle, \text{ and}$$
$$\mathcal{S}_\mu^i = (\tau = o,\ \alpha)$$

The $\mathcal{AAM}$ described above has been specialized to represent the memory management subsystem of an actor configuration. It would be interesting to extend the notion of meta-actors to classical actor facilities like encapsulation, inheritance and reflection. For instance, encapsulation properties become capability based at this point; a meta-actor can request for "classified information". At the meta-level, a system still obeys a "pure" actor semantics because base level actors are really passive data structures. In the following chapter, we are going to use the $\mathcal{AAM}$ to determine and prove the correctness of the HDGC algorithm.

Chapter 10

Correctness Proof of the Hierarchical Distributed Garbage Collection Scheme

Proof techniques for distributed algorithms ensure certain properties of the execution of the algorithm. In order to prove the correctness of a distributed algorithm, the temporal properties we must consider can be divided into two main categories, i.e., *safety* and *liveness* properties. In this chapter, we would like to show that the Hierarchical Distributed Garbage Collection Scheme proposed earlier in this thesis is correct. We will use the notation and the abstract machine developed in the previous chapter extensively in the correctness proof.

10.1 Correctness Proofs of Distributed Algorithms

To prove the correctness of the hierarchical distributed garbage collection scheme, we must first enumerate the safety and liveness properties that we must satisfy.

10.1.1 Safety or Invariance Properties

Safety properties of a distributed system are those properties that must always be true. In temporal logic, these properties have the following syntax: For a program P and property ψ, either

- $\Box\psi$ which says that every computation of P continuously satisfies ψ or
- $\varphi \supset \Box\psi$ which says that whenever φ is true, ψ is immediately true and will hold continuously for the rest of the computation.

Safety properties of a distributed system are those properties that must always be true. Safety is a finitely refutable property which means that if there exists an unsafe situation, it is possible to detect it in a finite number of steps and the point at which the safety property is violated can also be detected. Examples of safety properties are partial correctness, error-free execution, freedom from deadlock and mutual exclusion. Safety properties for the HDGC algorithm include the following:

A non-garbage actor will not be collected by the distributed garbage collection algorithm.

The user program progresses as normal without any semantic interference with the distributed garbage collection algorithm.

10.1.2 Liveness Properties

Liveness properties of a distributed system are those properties that must eventually be true. Temporally, these properties are expressed using the syntax shown below:

For a program P and property ψ, either

- $\Diamond\psi$ which says that the occurrence of event ψ is guaranteed unconditionally or
- $\varphi \supset \Diamond\psi$ which says that the occurrence of event ψ is guaranteed conditionally on the occurrence of φ

Total correctness, termination and fairness are liveness properties of a distributed system. To prove an algorithm for distributed garbage collection, we must consider the following liveness properties:

Termination: which states that the distributed garbage collection scheme terminates for every execution

Eventuality: which states that every garbage object will eventually be collected.

10.2 Summary of the Generation Based Distributed Garbage Collection Scheme

Before getting into an involved proof of the distributed garbage collection algorithm proposed in the earlier chapters, we will recapitulate the definition of a garbage actor as well as the steps involved in the HDGC scheme. A *reachable* actor is defined recursively as follows:

- A root actor is a reachable actor.
- Every forward-acquaintance of a reachable actor is reachable.
- Every inverse-acquaintance of a reachable actor that is active or can become active is reachable.

This definition automatically handles an actor whose mail address is currently in the network where the sender has forgotten about the actor and the receiver has not yet received it. A garbage actor is an actor which is not *reachable*. In other words, an actor which is not currently accessible or cannot potentially become accessible is a garbage actor. We cannot ignore the inverse acquaintances while determining reachability in an actor based model because an inverse acquaintance of a reachable actor can communicate its mail address at any point of time to its reachable acquaintance, thereby making itself reachable. But, if the actor is blocked and can never become active, it cannot communicate its mail address and is therefore inaccessible. Thus, reachability in the Actor model is determined by *liveness*, i.e., ability to survive.

The HDGC Algorithm consists of five steps.

1. Pre-GC Sweeps
2. DistributedScavenge Phase
3. Interphase Synchronization
4. LocalClear Phase
5. Post-GC Cleanup

The proof that we are going to develop in this chapter for the HDGC algorithm uses a two level hierarchy for the sake of simplicity. Therefore, the only form of cooperative garbage collection occurs at the global level also called Global Garbage Collection (GGC).

Prior to GC-initiation the following conditions are satisfied:

- All nodes in the system are untouched.
- All messages in the system are *old* messages, i.e, there are no *new* messages, GC or network-clearance messages in the system.

1. **Step 1 – Pre-GC Sweeps:** These include a set of messages sent out in the system to announce the start of GGC. In addition to informing the processing units of a forthcoming GGC, the links of the network must be cleared of all messages in transit. All pre-GC messages remaining in the system must be processed (at least tested for any possible acquaintances). The pre-GC broadcast and bulldoze messages are responsible for all the three operations mentioned above.

   To initiate the DistributedScavenge the following conditions must be satisfied:

   - All nodes in the system remain untouched as before, but
   - No messages in the system are *old* messages.

2. **Step 2 – DistributedScavenge Phase:**

   - Run the DistributedScavenge phase, making the reachable objects in Scavengespace *touched*.
   - Detect termination of the DistributedScavenge phase by back-propagation messages.

3. **Step 3 – Interphase Synchronization:** Each node in the system (hierarchy) is informed that the DistributedScavenge Phase has completed. This is accomplished by means of a broadcast message initiated at the start node. After receiving this message, each node can start the LocalClear phase.

4. **Step 4 – LocalClear Phase:**

   - LocalClear Initiation Broadcast, a message to every node to initiate the LocalClear phase.
   - LocalClear Completion Backwave that determines the termination point of the LocalClear phase.

5. **Step 5 – Post GC Broadcast:** This set of messages tells every node in the system that the LocalClear phase has been completed on every other node and that the *Global-GC-is-on* flag can be turned off.

10.3 Postconditions to be Verified

For each step of the HDGC algorithm, the preconditions are identical to the postconditions of the preceding step. Using the $\mathcal{AAM}$, we now state the postconditions for each GC step.

10.3.1 GC Step 1: Pre-GC Sweeps

- Postconditions: Characteristic features:
  - GC State should be 1
  - All actors are untouched
  - All references have been captured
  - No old messages in the system
  - No broadcast or bulldoze messages
  - No GC messages and GC acknowledgments in the system.

1. Meta-Actor Configuration:

$$\alpha_m = \langle\, \mathcal{S}_{m1},\, \chi \rangle$$

where finally

$$\mathcal{S}_{m1} = \langle\, \rho,\, \pi = 1,\, Pos,\, 0,\, 0,\, 0,\, 0,\, \mathcal{NC} = \phi \rangle$$

However the meta-actor configuration passes through a number of stages in the Pre-GC Sweep Phase; a broadcast phase and two sets of bulldoze phases. The transitions can be represented as follows:

$$\mathcal{S}_{m\alpha} = \langle\, \rho,\, \pi = 0,\, Pos,\, \sharp\mathcal{GC},\, \sharp\mathcal{GC}_{ack},\, \sharp\mathcal{GC}^{-1},\, \sharp\mathcal{GC}^{-1}_{ack},\, \mathcal{NC} = \phi \rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\, \rho,\, \pi = 0,\, Pos,\, \sharp\mathcal{GC},\, \sharp\mathcal{GC}_{ack},\, \sharp\mathcal{GC}^{-1},\, \sharp\mathcal{GC}^{-1}_{ack},\, \mathcal{NC} = br \rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ \pi = 1,\ Pos,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = fb\rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ \pi = 1,\ Pos,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = bb\rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ \pi = 1,\ Pos,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = fb\rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ \pi = 1,\ Pos,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = bb\rangle$$

$$\downarrow$$

$$\mathcal{S}_{m1} = \langle\ \rho,\ \pi = 1,\ Pos,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = \phi\rangle$$

Note that in all the above cases,

$\sharp\mathcal{GC} = \sharp\mathcal{GC}_{ack} = \sharp\mathcal{GC}^{-1} = \sharp\mathcal{GC}^{-1}_{ack} = 0$

2. Base-Actor Configuration:

$$\alpha_b = \langle\ \mathcal{S}_{b1},\ \overline{Acq},\ \overline{Acq^{-1}},\ \chi,\ \beta\rangle$$

where

$$\mathcal{S}_{b1} = \langle\ \mathcal{OS} = u,\ \delta,\ \lambda,\ \mathcal{CC}\rangle \text{ and } \lambda = p \text{ or } \lambda = o \text{ or } \lambda = pss \text{ or } \lambda = ns$$

3. Channel Configuration:

$$\mathcal{C}_1 \in \langle\ \mathcal{S}_\gamma, \overline{\mu},\ Source,\ Destination\ \rangle \text{ where}$$

$$\overline{\mu} \text{ is such that for every } \mu \in \overline{\mu},\ \mu = \langle\ \mathcal{S}_{\mu 1},\ \mu,\ \sigma,\ \theta\rangle, \text{ and}$$

$$\mathcal{S}_{\mu 1} = (\tau = n,\ \overline{b\alpha})$$

10.3.2 GC Step 2: DistributedScavenge Phase

- GC State should be 1
- All Reachable actors in scavengespace have been touched.
- No old, GC or invGC messages in the system at the end
- No broadcast or bulldoze messages

- No. of GC = No. of Gcacks, No. of invGc = no. of invgcacks
- Postconditions:
  1. Meta-Actor Configuration:

$$\alpha_m = \langle\ \mathcal{S}_{m2},\ \chi \rangle$$

where finally

$$\mathcal{S}_{m2} = \langle\ \rho,\ \pi,\ Pos,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} \rangle$$

$$\pi = 1,$$

$$\sharp\mathcal{GC} = \sharp\mathcal{GC}_{ack} \text{and}$$

$$\sharp\mathcal{GC}^{-1} = \sharp\mathcal{GC}^{-1}_{ack},$$

and

$$\mathcal{NC} = \phi, \text{ i.e., empty set.}$$

  2. Base-Actor Configuration:

$$\alpha_b = \langle\ \mathcal{S}_{b2},\ \overline{Acq}, \overline{Acq},\ \chi,\ \beta \rangle$$

where

$$\mathcal{S}_{b2} = \langle\ \mathcal{OS},\ \delta,\ \lambda,\ \mathcal{CC} \rangle$$

and we must further state that every reachable object in Scavengespace is touched and is logically in FutureSurvivor Space. An object is reachable if it is the acquaintance of a reachable object. Also, if the object is the inverse acquaintance of another reachable object and it is active, it is reachable. If the object is the inverse acquaintance of another reachable object and it is blocked, it is suspended. Let us first formulate a recursive definition of reachability. Note that the definition of reachability is developed in a fixed multicomputer configuration, i.e., w.r.t. a snapshot of the multicomputer system.

We begin with a root set, $\mathcal{R}$, that contains a predefined set of actornames and a root element ρ. Every element in $\mathcal{R}$ is reachable from ρ. Let $b\alpha_1$ and $b\alpha_2$ be base-actors. The relation *Reach* is defined as the least set of base-actors that satisfy the following conditions: The set of suspended actors (*Susp*) is the set of blocked base-actors which would be reachable if they can become active.

- if $b\alpha_1 \in \mathcal{R}$, then $b\alpha_1 \in Reach$.
- if $b\alpha_1 \in Reach$, then $b\alpha_2 \in Reach$ if $b\alpha_2 \in$ Forward-acq $(b\alpha_1)$.
- if $b\alpha_1 \in Reach$, then $b\alpha_2 \in Reach$ if $(b\alpha_2 \in$ Inverse-acq $(b\alpha_1))$ and $(b\alpha_2$ is active).
- if $b\alpha_1 \in Susp$, then $b\alpha_2 \in Reach$ if $(b\alpha_2 \in$ Inverse-acq $(b\alpha_1))$ and $(b\alpha_2$ is active).

On applying this definition, all reachable actors are contained in the set $Reach$.

Thus the postcondition for Step 2 of GC translates as follows:

Consider a base-actor $b\alpha$ with state $(\mathcal{OS}, \delta, \lambda, \mathcal{CC})$.

$$\text{if } (b\alpha \in Reach \text{ and } \mathcal{OS} = t) \text{ then } \lambda = fss \text{and}$$
$$\text{if } (b\alpha \in Reach \text{ and } \lambda = fss) \text{ then } \mathcal{OS} = t$$

3. Channel Configuration:

$$\mathcal{C}_2 = \langle\ \mathcal{S}_\gamma, \overline{\mu},\ Source,\ Destination\ \rangle \text{ where}$$
$$\overline{\mu} \text{ is such that for every } \mu \in \overline{\mu},\ \mu = \langle\ \mathcal{S}_{\mu 2},\ \iota,\ \sigma,\ \theta \rangle, \text{ and}$$
$$\mathcal{S}_{\mu 2} = (\tau = n,\ \overline{b\alpha})$$

10.3.3 GC Step 3: Interphase Synchronization

- GC Phase should be 2
- Postconditions:

1. Meta-Actor Configuration:

$$\alpha_m = \langle\ \mathcal{S}_{m3},\ \chi \rangle$$
$$\text{where finally}$$
$$\mathcal{S}_{m3} = \langle\ \rho,\ \pi,\ Pos,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} \rangle$$
$$\pi = 2,$$
$$\sharp\mathcal{GC} = \sharp\mathcal{GC}_{ack} \text{and}$$
$$\sharp\mathcal{GC}^{-1} = \sharp\mathcal{GC}^{-1}_{ack},$$
$$\text{and}$$
$$\mathcal{NC} = \phi, \text{ i.e., empty set.}$$

However the meta-actor configuration passes through a broadcast stage in this step;the transitions can be represented as follows:

$$\mathcal{S}_{m2} = \langle\ \rho,\ \pi = 1,\ Pos,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = br\rangle$$

$$\downarrow$$

$$\mathcal{S}_{m3} = \langle\ \rho,\ \pi = 2,\ Pos,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = \phi\rangle$$

2. Base-Actor Configuration: The base-actor configuration is not altered during the interphase synchronization. However, it preserves the basesystem state specified in the postconditions of the DistributedScavenge Phase.

$$\alpha_b = \langle\ \mathcal{S}_{b3},\ \overline{Acq},\ \overline{Acq^{-1}},\ \chi,\ \beta\rangle$$

where

$$\mathcal{S}_{b3} = \langle\ \mathcal{OS},\ \delta,\ \lambda,\ \mathcal{CC}\rangle \text{ or}$$

3. Channel Configuration:

$$\mathcal{C}_3 = \langle\ \mathcal{S}_\gamma, \overline{\mu},\ Source,\ Destination\ \rangle \text{ where}$$

$$\overline{\mu} \text{ is such that for every } \mu \in \overline{\mu},\ \mu = \langle\ \mathcal{S}_{\mu 3},\ \iota,\ \sigma,\ \theta\rangle, \text{ and}$$

$$\mathcal{S}_{\mu 3} = (\tau = n,\ \overline{b\alpha})$$

10.3.4 GC Step 4: LocalClear Phase

- No scavengespace untouched objects exist
- Postconditions:

1. Meta-Actor Configuration:
2. Base-Actor Configuration:

$$\alpha_b = \langle\ \mathcal{S}_{b4},\ \overline{Acq},\ \overline{Acq^{-1}},\ \chi,\ \beta\rangle$$

where

$$\mathcal{S}_{b4} = \langle\ \mathcal{OS},\ \delta,\ \lambda,\ \mathcal{CC}\rangle \text{ and}$$

$$\text{if } \lambda = fss,\ \mathcal{OS} = t \text{ and}$$

$$\text{if } \mathcal{OS} = t,\ \lambda = fss \text{ and}$$

$$\{\mathcal{S}_{b4} = \langle\ \mathcal{OS},\ \delta,\ \lambda,\ \mathcal{CC}\rangle \mid \mathcal{OS} = t \text{ implies that } \lambda = pss \text{ or } \lambda = ns\ \} = \phi, \text{ i.e., there are no untouched objects in Scavengespace.}$$

3. Channel Configuration:

$$\mathcal{C}_4 = \langle\ \mathcal{S}_\gamma, \overline{\mu},\ \mathit{Source},\ \mathit{Destination}\ \rangle \text{ where}$$

$$\overline{\mu} \text{ is such that for every } \mu \in \overline{\mu},\ \mu = \langle\ \mathcal{S}_{\mu 4},\ \iota,\ \sigma,\ \theta\rangle, \text{ and}$$

$$\mathcal{S}_{\mu 4} = (\tau = o,\ \overline{b\alpha})$$

10.3.5 GC Step 5: Post GC Sweeps

- Postconditions: Same as initial conditions.

10.3.6 Proof Strategy

The HDGC scheme is a program consisting of a set of n subprocesses $SP_1, SP_2, ..., SP_n$ corresponding to the different steps in the algorithm. We will begin proving the correctness of the HDGC scheme from the point at which a request for GGC has been issued by a node in the system. We postulate invariant conditions, attach them to the different stages of the algorithm and then show that executions of the subprocesses preserve the truth of the invariant conditions. We state the conditions used to show the soundness, completeness and termination of the individual steps in the distributed garbage collection scheme as Lemmas. Using these lemmas, we will prove that the integrated scheme satisfies the temporal properties enumerated earlier, namely, liveness and safety. The proofs of the individual lemmas will follow the proof of the HDGC algorithm.

To formalize the proof of a distributed garbage collection algorithm, we need to formally express the GC process as a transition relation and show that the possible computations of the system satisfy the postconditions and that these in turn imply the desired correctness properties.

An automatically verifiable proof of the distributed garbage collection scheme will require a number of messy details. The purpose of this proof is to detect and clarify the correctness of

the error-prone regions of the HDGC scheme. We therefore refrain from getting into the messy details needed for a completely formal proof. A rigorous formalization of the correctness proof will appear in a forthcoming paper.

Claim: The Hierarchical Distributed Garbage Collection Scheme is correct.

1. Safety Theorems
   - **Safety Theorem 1 (ST1):** *A non-garbage actor will not be collected by the distributed garbage collection algorithm.*
   - **Safety Theorem 2 (ST2):** *The user program progresses as normal without any semantic interference with the distributed garbage collection algorithm.*
2. Liveness Theorems
   - **Liveness Theorem 1 (LT1):** *The generation based distributed garbage collection scheme terminates for every execution.*
   - **Liveness Theorem 2 (LT2):** *Every garbage object will eventually be collected.*

10.4 Lemmas used in the Correctness Proof

Lemma 1.1 : *After the pre-GC phase, every processing unit has been informed that a global GC is about to take place.*

Lemma 1.2 : *The pre-GC sweeps clear the network links of any* old *messages.*

Lemma 1.3 : *At the end of the Pre-GC Sweep Phase, a consistent distributed snapshot of the system for beginning GC has been obtained. All references in the system have been captured.*

Lemma 2.1 : *The DistributedScavenge phase marks all reachable objects in scavengespace as a* **touched** *object.*

Lemma 2.2 : *The DistributedScavenge phase will not mark an unreachable object corresponding to the snapshot as a* **touched** *object.*

Lemma 2.3 *After the DistributedScavenge phase, all non-garbage actors in scavengespace have a generation-tag equal to future-tag.*

Lemma 2.4 *The DistributedScavenge phase terminates.*

Lemma 3.1 : *The LocalClear phase is initiated on every node in the system.*

Lemma 4.1 : *The LocalClear phase collects all and only garbage (untouched) scavengespace objects on a node.*

Lemma 4.2 : *The LocalClear phase terminates.*

Lemma 5.1 : *The termination of a GC has been correctly detected and all the nodes in the system have been informed of the same.*

10.5 Correctness Proof of the Hierarchical Distributed GC Scheme

10.5.1 Invariant Properties of the Transition System

Consider a multicomputer configuration $\mathcal{M}$. Let $\mathcal{M}'$ be the final multicomputer configuration after the HDGC process. The HDGC process is represented by the transition system $\mathcal{T}_{\mathcal{M}}$.

$$\mathcal{T}_{\mathcal{M}}\colon \mathcal{M} \longrightarrow \mathcal{M}'$$

$\mathcal{T}_{\mathcal{M}}$ preserves the following properties:

1. There are a finite number of actors in the system. finite.
2. Only a finite amount of processing is done anytime a system receives a *scavenge* message.
3. Presently, we will assume that there is only one meta actor on a node. However, there may be more than one base actors on a node.

4. If a message is sent from point A to point B in the network, it moves closer to its destination at every hop.

5. Along a single link in the network messages are communicated in a FIFO fashion.

6. The mutator cooperates with the collector. Any new actors created during GC are created as touched actors with a generation tag of FutureSurvivorspace (future-tag). Also, once a garbage collection process has been initiated, the mutator does not delete any acquaintances and inverse acquaintances.

7. A garbage actor can never become non-garbage.

8. Only one GC can be active in the system at a point in time.

9. The touching process is *monotonic*, i.e., once an actor has been marked as *touched* during a GC phase, it cannot subsequently be untouched or suspended during the same GC phase.

10. The header or tag bits in every base actor are not accessible to other base actors, they are accessible to other meta-actors; i.e., the tag bits are transparent to the user process.

Temporal properties that have to be taken into consideration to prove the correctness of the integrated GC scheme using the DGS and NCA algorithms are safety and liveness properties.

10.5.2 Safety Properties

We will now prove the safety theorems of the HDGC scheme.

Safety Theorem 1 (ST1): *A non-garbage actor will not be collected by the distributed garbage collection algorithm.*

Proof by Contradiction:

Say a reachable base actor $b\alpha$ is collected by the HDGC algorithm. Therefore,

$$b\alpha \in Reach$$

If $b\alpha$ is not in Scavengespace, it is not considered for collection during the LocalClear Phase. If $b\alpha$ is in Scavengespace, by Lemma 2.1, $b\alpha$ is touched and by Lemma 2.2, $b\alpha$ has a generation-tag equal to future-tag. The base-actor, therefore, has the following configuration at the end of the DistributedScavenge Phase.

$$\alpha_b = \langle\ \mathcal{S}_{b\alpha},\ \overline{Acq},\ \overline{Acq^{-1}},\ \chi,\ \beta \rangle$$

where

$$\mathcal{S}_{b\alpha} = \langle\ \mathcal{OS} = t,\ \delta,\ \lambda = fss,\ \mathcal{CC} \rangle$$

As we have explained earlier, the main process of clearing garbage objects is done by the LocalClear Phase. When the LocalClear Phase begins execution, it walks down the objects in local memory and clears those objects that are in *Scavengespace* and have their object-status field set to touched ($\mathcal{OS} = t$). As actor $b\alpha$ is *touched*, it cannot be cleared by the LocalClear Phase of the algorithm and the conjecture that actor $b\alpha$ was collected by the DGS algorithm is false. Therefore, our initial assumption that accessible actors are collected by the LocalClear Phase of the algorithm is false.

Safety Theorem 2 (ST2): *The user program progresses as normal without any semantic interference with the distributed garbage collection algorithm and vice versa.*

Proof:

The user application program is executed by the base actor system and it does not require any information from the meta-actor system. The HDGC algorithm, on the other hand, is executed by the meta-actor system and it uses information contained in the base-actor system like the object-tag as well as acquaintance and inverse acquaintance information.

In order to show semantic independence of the two processes, we need to show that the information modified by the two processes (user application and GC meta-actor) do not interfere in any manner.

Let $\mathcal{ALT}_b$ be the set of data structures altered by the base actor system. This set includes;

- the mail queue of the base actor (χ)
- the behavior of the base actor (β)
- the acquaintance and inverse acquaintance lists of the base actor $\overline{Acq}$, $\overline{Acq^{-1}}$,
- the dynamicity of the actor (δ)

$$\mathcal{ALT}_b = \{\ \chi,\ \beta,\ \overline{Acq},\ \overline{Acq^{-1}},\ \delta\}$$

Note that the base-actor system does not alter the following information:

- information in its tag ,the object-status ($\mathcal{OS}$), the gcphase (π), the logical area in memory where the object resides (λ) and the copy-count ($\mathcal{CC}$).
- any information in the meta-actor system.

Let $\mathcal{ALT}_m$ be the set of data structures altered by the meta actor system. This set includes:

- Fields in the meta-actor State; the current GC phase (π), the number of GC messages sent and number of acknowledgments received ($\sharp\mathcal{GC}$, $\sharp\mathcal{GC}_{ack}$), the number of GC inverse messages sent and number of acknowledgments received ($\sharp\mathcal{GC}^{-1}$, $\sharp\mathcal{GC}^{-1}_{ack}$)
- Some fields in the base-actor State; the object-status ($\mathcal{OS}$), the logical area in memory where the object resides (λ) and the copy-count ($\mathcal{CC}$).

$$\mathcal{ALT}_m = \{\ \pi,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{OS},\ \lambda,\ \mathcal{CC}\}$$

The transitions induced by the meta-actor system do not alter

- the mail queue of the base actor
- the behavior of the base actor
- the acquaintance and inverse acquaintance lists of the base actor
- the dynamicity of the base actor

As the set of altered fields, $\mathcal{ALT}_b$ and $\mathcal{ALT}_m$ do not intersect with each at any point in execution, the two processes are totally independent threads and they cannot conflict with each other.

$$\mathcal{ALT}_b \cap \mathcal{ALT}_m = \phi$$

Furthermore, we must also show that in addition to altering information, information read will not semantically interfere with the execution of the application. Note that although the meta-actor system reads information altered by the base actor system (the acquaintance and inverse acquaintance lists, dynamicity), the base-actor system does not read any information altered by the meta-actor system. Therefore, the execution of the HDGC algorithm by the base actor system does not semantically interfere with the execution of the application program.

10.5.3 Liveness Properties

Here, we outline a proof of the liveness theorems that we have stated earlier.

Proof of Termination

Liveness Theorem 1 (LT1): *The generation based distributed garbage collection scheme terminates for every execution.*

Proof: The proof of termination relies on the *Principle of Monotonicity* which states that during a GC, a node that is touched cannot become untouched or suspended during that GC phase. Also, a suspended node cannot become untouched.

We need to show that the different phases of garbage collection start and terminate correctly. As we have formulated in the earlier chapter, the termination state of one step in the HDGC algorithm is equivalent to the initiation state of the subsequent step. Thus proper termination of one phase implies proper initiation of the next phase.

Proper Initiation and Termination of the Pre-GC Phase: The pre-GC phase is initiated when a node requests a global GC and a GC request message is sent to the root actor (ρ). The root actor initiates the pre-GC sweeps at the start node. The correct functioning of the pre-GC sweep phase trivially follows from Lemmas 1.1, 1.2 and 1.3. At the end, the GC field in the meta-actor state is set to one ($\pi = 1$), all objects are untouched ($\mathcal{OS} = u$) and all messages in the system are tagged new ($\tau = n$). When the final B-bulldoze message reaches the start node of the system, a message is sent back to the root node informing the completion of the Pre-GC sweeps. This automatically initiates the DistributedScavenge Phase.

Proper Initiation and Termination of the DistributedScavenge Phase: The DistributedScavenge Phase is initiated at the root node of the system at the end of the Pre-GC sweeps.

We have assumed that the number of actors that exist in the distributed system is finite and that a finite amount of work is done on each scavenge message. Proper execution of the DistributedScavenge phase relies on the following finiteness criteria: finite number of actors, finite amount of work on each scavenge message and finite number of scavenge

messages. The first two criteria follow from the assumptions we have made. We now show that the third finiteness condition is satisfied. Once an actor receives a scavenge message and processes it, subsequent scavenge messages do not initiate another GC process at this actor. A back-propagation message is sent to the sender of the GC message. This assures us that an actor cannot repeatedly send GC messages to its acquaintances. Thus, there a finite number of scavenge messages propagated by the system. The proper detection of the termination of the DistributedScavenge Phase follows from Lemma 2.3.

Proper Initiation and Termination of Interphase Synchronization: Termination of DistributedScavenge Phase occurs at the root node (ρ). At this instant, the root node initiates the interphase synchronization at the start node. This step involves a broadcast wave through the system, informing every node that the DistributedScavenge Phase has completed and that the LocalClear Phase can begin. When this broadcast message reaches the finish node, interphase synchronization has completed. The finish node sends a message to the root node. The correct execution of the Interphase Synchronization follows from Lemma 3.1.

Proper Initiation and Termination of the LocalClear Phase: Processing involved in the LocalClear Phase of the algorithm, the local clear phase, is initiated at the point the processor receives the interphase synchronization broadcast. When the interphase synchronization has completed, the root node starts a FBulldoze message at the start node. The meta-actor at each node propagates this message to its Fneighbors when it has completed its local LocalClear Phase. The proper detection of the termination of the LocalClear Phase follows from Lemma 4.2.

Proper Initiation and Termination of the Post-GC Phase: The bulldoze wave in the LocalClear Phase terminates at the Finish node. A message is sent to the root node . The interpretations of the following tags are interchanged (these are in effect for the next GC).

- the object-status bits corresponding to touched objects and untouched objects.
- Logical Memory area tags - FSS-Tag and PSS-Tag.
- Old Message Tag and New Message Tag.

The Post-GC Phase is a broadcast wave that informs all processing units of the flipped interpretations. This follows from Lemma 5.1. Once the broadcast wave has completed, the global GC variable *GC-is-on* is set to 0 at the root node.

Proof of Eventuality

Liveness Theorem 2 (LT2): *Every garbage object will eventually be collected with a finite number of GC's.*

Proof:

The proof relies on the following two assumptions. Firstly, every actor in scavengespace which is reachable has an existing or potential path from a root node leading to it. Secondly, once an actor has been determined to be a garbage actor, it will remain garbage. All reachable objects are eventually touched by the DistributedScavenge phase. In addition, from Lemma 2.2, we also know that garbage objects are not touched by the GC process. Garbage objects can exist either in Oldspace or in Scavengespace.

1. **Case 1: Garbage Objects in Scavengespace**

   There are two situations that must be considered here

   - The object in Scavengespace is garbage at the beginning of a GC Phase. The object is left *untouched* by the distributed scavenge phase and all untouched scavengespace (garbage) objects are collected during the LocalClear Phase by Lemma 4.1.
   - The object in Scavengespace becomes inaccessible during a GC Phase. The object was accessible at the start of the GC and will be *touched* by the DistributedScavenge Phase. The object will remain uncollected by this garbage collection phase. However, it is garbage at the beginning of the next GC Phase. This is a consequence of the invariant rule which states that garbage actors can never become non-garbage. The actor is therefore collected by the next GC Phase.

   By the argument given above, garbage objects in Scavengespace are collected either in an ongoing GC or by the next GC. Therefore, garbage objects in Scavengespace are eventually collected.

2. **Case 2: Garbage Objects in Oldspace**

The DGS algorithm collects only garbage objects in Scavengespace. If an object is in Oldspace, it automatically survives a GC. Periodically, a total garbage collection (TGC) is executed. A TGC is a garbage collection involving both Scavengespace and Oldspace objects. The DistributedScavenge Phase of the TGC algorithm touches all reachable objects in the system (much like the marking phase of a mark-and-sweep algorithm). During the LocalClear Phase of the TGC, objects survive only if their *touched* bits are set irrespective of whether they are in Scavengespace or Oldspace.

- The object in Oldspace is garbage at the beginning of a TGC Phase. The object is left *untouched* by the distributed scavenge phase of the TGC and all untouched (garbage) objects either in Scavengespace or Oldspace are collected during TGC LocalClear Phase by Corollary 4.1.1.
- The object in Oldspace becomes inaccessible during a TGC Phase. The object was accessible at the start of the TGC and will be *touched* by the DistributedScavenge Phase of the TGC Algorithm. The object will remain uncollected by this total garbage collection phase. However, it is garbage at the beginning of the next TGC Phase. This is a consequence of the invariant rule which states that garbage actors can never become non-garbage. The actor is therefore collected by the next TGC Phase.

By the argument given above, garbage objects in Oldspace are collected either in an ongoing TGC or by the next TGC. Therefore, garbage objects in Oldspace are eventually collected.

As all garbage objects either in Oldspace or Scavengespace are eventually collected, all the garbage actors in the system are eventually collected.

All temporal properties (safety and liveness) are satisfied by the HDGC scheme. Therefore, the HDGC scheme executes in a bounded amount of time, correctly identifies only garbage actors, terminates safely and guarantees correct progress of the distributed system. Therefore, the generation based distributed garbage collection scheme is correct.

10.6 Correctness Proof of the Individual Lemmas in the Hierarchical Distributed Garbage Collection Scheme

10.6.1 Initial Conditions

- *All actors in the system are untouched*
- *Messages in the system are of one kind – Pre-GC or Old messages*

Using the formalization described in the earlier chapter, we can state this as a property of the base actor configuration and channel configuration.

$$\mathcal{A}_b^i = \langle\ \mathcal{S}_b^i,\ \overline{Acq},\ \overline{Acq^{-1}},\ \chi,\ \beta\rangle \text{ where}$$
$$\mathcal{S}_b^i = \langle\ \mathcal{OS} = u,\ \delta,\ \lambda,\ \mathcal{CC}\rangle \text{ and } (\ \lambda = p \text{ or } \lambda = o \text{ or } \lambda = pss \text{ or } \lambda = ns).$$

$$\gamma^i = \langle\ \mathcal{S}_\gamma^i,\ \overline{\mu},\ \textit{Source},\ \textit{Destination}\ \rangle \text{ where}$$
$$\overline{\mu} \text{ is such that for every } \mu \in \overline{\mu},\ \mu = \langle \mathcal{S}_\mu^i,\ \iota,\ \sigma,\ \theta\rangle, \text{ and}$$
$$\mathcal{S}_\mu^i = (\tau = o,\ \alpha)$$

10.6.2 GC Step 1: Pre-GC Sweeps:

This step is responsible for informing all the processing units that a GGC is about to start. After this, it records a consistent snapshot of the system with respect to which GGC is performed.

The pre-GC sweeps consists of the following steps:

1. Pre-GC Broadcast Wave
2. Pre-Gc Bulldoze Waves: These include the forward bulldoze and backward bulldoze waves.

In an actor based scheme, the set of bulldoze waves is performed twice, the first time to handle acquaintances and the second time to handle the corresponding inverse acquaintances.

Lemma 1.1 : *After the pre-GC phase, every processing unit has been informed that a global GC is about to take place.*

Proof: The pre-GC broadcast informs every processing unit in the system of a forthcoming GGC. During this phase, changes are made to the meta-actor configuration of a system. The *broadcast wave* described in Chapter 7 is initiated at the start node of the system and a message

is broadcast to every node in the system along a predefined path. The message that is broadcast communicates the start of a GC. As the number of nodes are finite and as all broadcast messages reach their destination, the broadcast wave terminates and all the processing units are informed of a GC.

The transitions corresponding to the broadcast wave are summed up in the meta-actor configuration as:

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ Pos,\ \pi = 0,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = \phi \rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ Pos = S,\ \pi = 0,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = br \rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ Pos = M,\ \pi = 0,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = br \rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ Pos = F,\ \pi = 0,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = br \rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ Pos,\ \pi = 1,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = \phi \rangle$$

Note that in all the above cases, $\sharp\mathcal{GC} = \sharp\mathcal{GC}_{ack} = \sharp\mathcal{GC}^{-1} = \sharp\mathcal{GC}^{-1}_{ack} = 0$.

When a node receives the broadcast message, it begins to record all its acquaintances, inverse acquaintances and any other dynamic acquaintances (mail addresses that are communicated to the node) of base-actors on that node. Also, subsequent to the broadcast, the *GC-is-on* flag is turned on to indicate that a GC is already underway. Following this, the links of the network must be cleared of all messages in transit and all pre-GC messages remaining in the system must be treated (tested for any possible acquaintances). The pre-GC bulldoze messages are responsible for all these operations. The first set of bulldoze messages (forward and backward) guarantee the communication of all forward acquaintances to their destinations. The second set of bulldoze messages ensure that corresponding inverse acquaintance messages have been communicated.

Lemma 1.2 : *The pre-GC sweeps clear the network links of any* old *messages.*

Proof: Beginning at the start node, the pre-GC bulldoze messages travel down every link and node in the system. When a node/object receives a bulldoze message, it processes any *old* messages in its mail queue and captures all references in the acquaintance and inverse

acquaintance lists. Messages to other destinations that are being routed along the node are rerouted along appropriate directions in accordance with the progress-only rule. Also, no *old* messages are issued from the node beyond this point. When a bulldoze message is issued from a node to its Fneighbor or its Bneighbor, progress-only routing guarantees that it travels along the single link between the nodes. All messages along that link reach the subsequent node before the bulldoze message gets there, as a consequence of message order preservation along a link. This means that references traveling along the link will be handled at the subsequent node. Furthermore, this happens along every link and thus no references will be lost in transit. When the F-bulldoze messages reach the finish node, all current messages in the nodes as well as messages in the forward direction have been handled. But, so far, we have not yet handled acquaintances that are traveling in the backward direction, implying that *old* messages may still exist in the network. The backward bulldoze message carries out the same series of actions that the forward bulldoze message did in the reverse direction. These transitions can be expressed in terms of the actor abstract machine ($\mathcal{AAM}$) as follows:

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ \pi = 1,\ Pos,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = \phi\rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ \pi = 1,\ Pos = S,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = fb\rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ \pi = 1,\ Pos = M,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = fb\rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ \pi = 1,\ Pos = F,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = fb\rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ \pi = 1,\ Pos = F,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = bb\rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ \pi = 1,\ Pos = M,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = bb\rangle$$

$$\downarrow$$

$$\mathcal{S}_{m\alpha} = \langle\ \rho,\ \pi = 1,\ Pos = S,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = bb\rangle$$

$$\downarrow$$

$$\mathcal{S}_{m1} = \langle\ \rho,\ \pi = 1,\ Pos,\ \sharp\mathcal{GC},\ \sharp\mathcal{GC}_{ack},\ \sharp\mathcal{GC}^{-1},\ \sharp\mathcal{GC}^{-1}_{ack},\ \mathcal{NC} = \phi\rangle$$

Again, $\sharp\mathcal{GC} = \sharp\mathcal{GC}_{ack} = \sharp\mathcal{GC}^{-1} = \sharp\mathcal{GC}^{-1}_{ack} = 0$.

We have explained the necessity of maintaining inverse acquaintances in an actor based system. Now, as a result of the forward and backward bulldoze phases, new acquaintances may be communicated. When a node A receives an acquaintance B during the pre-GC phase, we must place the mail address of node A on the inverse acquaintance list of B before initiating the DistributedScavenge Phase. During the GC phase, messages communicating inverse acquaintances are sent to corresponding nodes. As new acquaintances may be acquired till the first backward bulldoze phase completes, we need to execute an additional set of bulldoze waves to clear any inverse acquaintance messages in the links of the network.

The effect of the above actions on the channel configuration of the system can be summarized as:

$$\gamma^i = \langle\ \mathcal{S}_\gamma, \overline{\mu},\ \mathit{Source},\ \mathit{Destination}\ \rangle \text{ where}$$
$$\overline{\mu} \text{ is such that for every message}$$
$$\mathcal{S}_{\mu 1} = (\tau = n,\ b\alpha)$$

Lemma 1.3 : *At the end of the Pre-GC Sweep Phase, a consistent distributed snapshot of the system for beginning GC has been obtained. All references in the system have been captured.*

Proof:

During the pre-GC broadcast, every PU is informed of a GC. We begin recording acquaintances and inverse acquaintances at this point.

In order to show that a consistent distributed snapshot has been obtained we must show that

1. All references in the system have been captured.
2. None of the messages in the system are *old*.

Claim 1 : All references in the system have been captured.

Proof:

References in the system can exist at two points, i.e., in the nodes (acquaintance lists of actors) or in messages (this includes messages in the mail queue of an actor).

During the pre-GC broadcast, all nodes are informed of a GC. All acquaintances in nodes are recorded once the base actor receives the pre-GC broadcast. Subsequently the actor

cannot lose these acquaintances till the collection currently in progress has been completed.

All acquaintances in messages that have not already been recorded are captured during the forward and backward bulldoze phases. Messages may be in transit in the network or they may reside in the mail queue of an actor. The bulldoze messages clear the links along which they travel and thus push messages toward their destinations. When a node receives a bulldozer, it records all references in the mail queues of actors on that node. During the Fbulldoze phases, all addresses communicated in forward messages [1] are captured at their destination node in the system. Similarly, during the Bbulldoze phases, all addresses communicated in backward messages [2] are captured at their destination node in the system. Thus, references in the mailqueue and references in transit are captured by the Fbulldoze and Bbulldoze messages.

Claim 2: None of the messages in the system are *old*.

Proof:

At the start of DGC all the messages in the system are tagged as *old*. The transition from *old* to *new* occurs when the message is issued from a node that has already received the pre-GC broadcast. Therefore, every destination node has received the pre-GC broadcast, no further messages issued from these nodes will be *old*. Also, during the bulldoze phase, all *old* messages have eventually reached their destination nodes. This is a consequence of Lemma 1.2. As all old messages will eventually reach their destination after the bulldoze phases and no *old* messages are generated after the GC-broadcast, there will be no old messages that have unrecorded references at the end of the bulldoze phase.

Corollary 1.3.1 : *At the end of the Pre-GC Sweep phase, every accessible or potentially accessible node has been recorded as an acquaintance or inverse acquaintance at some node in the network. In other words, everything that is potentially accessible has been accounted for before the start of the DistributedScavenge Phase.*

[1] Messages traveling along Fpaths are forward messages
[2] Messages traveling along Bpaths are backward messages

10.6.3 GC Step 2: DistributedScavenge Phase

Prior to the DistributedScavenge Phase, all nodes in the system are untouched. All messages in the system are *new* messages.

The DistributedScavenge Phase of the algorithm has two functions to perform.

- Run the distributed scavenge phase, making the accessible objects in Scavengespace *touched*.
- Detect termination of the DistributedScavenge Phase by back-propagation messages.

During the DistributedScavenge phase, we initiate GC at a predefined set of root nodes in the system. Nodes in the root set are marked *touched* and GC messages are issued to the acquaintances and inverse acquaintances of the touched actors according the SFA (Scavenge-forward-acquaintance) and SIA (scavenge-inverse-acquaintance) algorithms in Chapter 6. The transitions that can take place are enumerated below. We restate the recursive definition of reachability given in the earlier chapter.

We begin with a root set, $\mathcal{R}$, that contains a predefined set of elements and a root element ρ. Every element in $\mathcal{R}$ is reachable from ρ. Let $b\alpha_1$ and $b\alpha_2$ be base actors. The relation *Reach* is defined on the set base-actors

- if $b\alpha_1 \in \mathcal{R}$, then $b\alpha_1 \in Reach$.
- if $b\alpha_1 \in Reach$, then $b\alpha_2 \in Reach$ if $b\alpha_2 \in$ Forward-acq $(b\alpha_1)$.
- if $b\alpha_1 \in Reach$, then $b\alpha_2 \in Reach$ if $(b\alpha_2 \in$ Inverse-acq $(b\alpha_1))$ and $(b\alpha_2$ is active).
- if $b\alpha_1 \in Susp$, then $b\alpha_2 \in Reach$ if $(b\alpha_2 \in$ Inverse-acq $(b\alpha_1))$ and $(b\alpha_2$ is active).

On applying this definition, all reachable actors are contained in the set *Reach*.

The postcondition for Step 2 of GC translates as follows:

Consider a base actor $b\alpha\langle\ \mathcal{OS}, \delta, \lambda, \mathcal{CC}\rangle$.

$$\text{if } (b\alpha \in Reach \text{ and } \mathcal{OS} = t) \text{ then } \lambda = fss \text{ and}$$
$$\text{if } (b\alpha \in Reach \text{ and } \lambda = fss) \text{ then } \mathcal{OS} = t$$

Thus, every reachable object in Scavengespace is touched and is logically in FutureSurvivor Space.

The system does not halt or freeze during a garbage collection and new actors may be created while the existing actors are being marked. Newly created actors are always reachable. They are created as touched objects with a generation-tag equal to future-tag. Actors in Oldspace and actors that are persistent are assumed to be reachable and do not participate in the GC process. The LocalClear Phase of the algorithm ignores Oldspace and persistent objects. The condition that must be satisfied by an object in order to survive the LocalClear Phase is:

```
( (Actor is persistent) or
  (Actor is in Oldspace) or
  ((Actor is in Scavengespace) and (Actor is touched)))
```

Using the definition of reachability, we can formalize the set of transitions that can take place. Let $b\alpha_1$ and $b\alpha_2$ be base actors.

Case 1: Every forward acquaintance of a reachable actor is reachable.

For every $b\alpha_1$, if

$$\mathcal{S}_{b1} = \langle\ \mathcal{OS} = u,\ \delta,\ \lambda,\ \mathcal{CC} \rangle \text{ and}$$
$$\lambda = pss \text{ or } \lambda = ns, \text{ and}$$
$$b\alpha_1 \in \mathcal{GC}_{acq}\ (b\alpha_2),$$
$$b\alpha_2 \in Reach,$$
$$\text{then the state of } b\alpha_1 \text{ is changed to}$$
$$\mathcal{S}_{b2} = \langle\ \mathcal{OS} = t,\ \delta,\ \lambda = fss,\ \mathcal{CC} + 1\ \rangle$$

Case 2: Every active inverse acquaintance of a reachable actor is reachable.

For every $b\alpha_1$ that is an active actor, if

$$\mathcal{S}_{b1} = \langle\ \mathcal{OS} = u,\ \delta = a,\ \lambda,\ \mathcal{CC} \rangle \text{ and}$$
$$\lambda = pss \text{ or } \lambda = ns, \text{ and}$$
$$b\alpha_1 \in \mathcal{GC}^{-1}_{acq}\ (b\alpha_2),$$
$$b\alpha_2 \in Reach,$$
$$\text{then the state of } b\alpha_1 \text{ is changed to}$$
$$\mathcal{S}_{b2} = \langle\ \mathcal{OS} = t,\ \delta,\ \lambda = fss,\ \mathcal{CC} + 1\ \rangle$$

Case 3: Every blocked inverse acquaintance of a reachable actor is suspended.

For every $b\alpha_1$ that is a blocked actor, if

$$\mathcal{S}_{b1} = \langle\ \mathcal{OS} = u,\ \delta = b,\ \lambda, \mathcal{CC}\rangle \text{ and}$$
$$\lambda = pss \text{ or } \lambda = ns, \text{ and}$$
$$b\alpha_1 \in \mathcal{GC}^{-1}_{acq}\ (b\alpha_2),$$
$$b\alpha_2 \in Reach,$$

then the state of $b\alpha_1$ is changed to

$$\mathcal{S}_{b2} = \langle\ \mathcal{OS} = s,\ \delta,\ \lambda = fss, \mathcal{CC}\rangle$$

Case 4: An active inverse acquaintance of a suspended actor is reachable.

For every $b\alpha_1$ that is a suspended actor, if

$$\mathcal{S}_{b1} = \langle\ \mathcal{OS} = s,\ \delta = b,\ \lambda, \mathcal{CC}\rangle \text{ and}$$
$$\lambda = pss \text{ or } \lambda = ns, \text{ and}$$
$$b\alpha_2 \in \mathcal{GC}^{-1}_{acq}\ (b\alpha_1)$$

and $b\alpha_2$ is an active actor, $b\alpha_2$ becomes touched and the state of $b\alpha_1$ is changed to

$$\mathcal{S}_{b2} = \langle\ \delta = a,\ \delta,\ \lambda = fss, \mathcal{CC}{+}1\ \rangle$$

Case 5: A blocked inverse acquaintance of a suspended actor is suspended.

For every $b\alpha_1$ that is a suspended actor, if

$$\mathcal{S}_{b1} = \langle\ \mathcal{OS} = s,\ \delta = b,\ \lambda, \mathcal{CC}\rangle \text{ and}$$
$$\lambda = pss \text{ or } \lambda = ns, \text{ and}$$
$$b\alpha_2 \in \mathcal{GC}^{-1}_{acq}\ (b\alpha_1)$$

and $b\alpha_2$ is a blocked actor, $b\alpha_2$ becomes suspended and the state of $b\alpha_1$ is not changed.

Lemma 2.1 : *The DistributedScavenge Phase marks all reachable objects in scavengespace as a* **touched** *object.*

Proof:

Follows from the transitions described above.

Lemma 2.2 : *The DistributedScavenge Phase will not mark an unreachable actor as a* **touched** *actor.*

Proof: By contradiction

Say an unreachable base actor $b\alpha$ is touched by the HDGC algorithm. An unreachable actor is an actor that cannot be reached from any root actor. Initially, all actors, both reachable and unreachable, are untouched. The process of touching begins at the root set actors and propagates to all actors reachable from the root set. As

$$b\alpha \notin Reach,$$

it has following configuration at the end of the DistributedScavenge Phase.

$$\alpha_b = \langle\ \mathcal{S}_{b\alpha},\ \overline{Acq},\ \overline{Acq^{-1}},\ \chi,\ \beta \rangle$$

where

$$\mathcal{S}_{b\alpha} = \langle\ \mathcal{OS} = u,\ \delta,\ \lambda,\ \mathcal{CC} \rangle$$
$$\mathcal{S}_{b\alpha} = \langle\ \mathcal{OS} = s,\ \delta,\ \lambda,\ \mathcal{CC} \rangle$$

We also know that once an object has been determined to be unreachable, and will not be touched by the garbage collection system. The actor preserves the unreachability for all future executions. This is a feature of the mutator system and can be stated formally as follows. The world of actors is partitioned into reachable actors ($Reach$) and ($Unreach$) actors. Consider a baseactor $b\alpha$ that is unreachable in state S1. Say a sequence of transitions causes the base actor to belong to state S2. The following is a property of the transition system:

S1: ($b\alpha \in Unreach$) and ($b\alpha$ is not touched in S1) $\longrightarrow$ S2, then
S2: ($b\alpha \in Unreach$) and ($b\alpha$ is not touched in S2).

Therefore, our initial assumption that unreachable actors are touched by the Distributed-Scavenge phase is false.

Lemma 2.3 *After the DistributedScavenge phase, all reachable actors in scavengespace have a generation-tag equal to future-tag.*

Proof: Follows from the transitions stated above.

Lemma 2.4 *The DistributedScavenge phase terminates.*

Proof:

In order for the DistributedScavenge Phase to terminate, we use the following finiteness criteria:

- Finite number of actors in the system: This implies that every actor can have only a finite number of acquaintances and inverse acquaintances.
- Finite amount of work on each scavenge message: When an actor receives a scavenge message, it processes the scavenge message according to the SFA and SIA algorithms. This involves sending scavenge messages to all the acquaintances and inverse acquaintances of the actor. But as the number of acquaintances and inverse acquaintances are finite, the time taken to process the scavenge message is finite.
- During a GC, am actor sends out scavenge messages to its acquaintances and inverse acquaintances just once.

GC messages start out at the root node and are sent to all actors reachable from the root set. Every meta-actor keeps track of the number of GC messages sent to acquaintances and inverse acquaintances. When a node has completed GC, the node has the following configuration,

$\sharp\mathcal{GC} = \sharp\mathcal{GC}_{ack}$ and $\sharp\mathcal{GC}^{-1} = \sharp\mathcal{GC}^{-1}_{ack}$

.

The node then issues a back-propagation (acknowledgment) to its parent in the reachability tree from which it received the first GC message. These messages propagate up the reachability tree. The termination condition is detected when the root GC meta-actor has the following configuration.

$\sharp\mathcal{GC} = \sharp\mathcal{GC}_{ack}$ and $\sharp\mathcal{GC}^{-1} = \sharp\mathcal{GC}^{-1}_{ack}$

.

i.e., all acknowledgments have been received at the root node.

10.6.4 GC Step 3: Interphase Synchronization

The LocalClear Phase of the algorithm eliminates all the garbage actors in the system and is initiated when all non-garbage objects have been identified. Termination of the Distributed-Scavenge Phase has been shown correct in Lemma 2.3. All the PU's in the system must be informed of this condition so that the local sweep phase can be started on every node. The *interphase synchronization* stage consists of a broadcast wave (explained in Chapter **??**).

When a processing unit receives a broadcast wave, it sets the local variable *GC-Phase* to be equal to 2 (an indication that the LocalClear phase of the algorithm has begun on that node).

Lemma 3.1 : *The LocalClear Phase is initiated on every node in the system.*

Proof:

This follows from the nature of the broadcast message that reaches every node in the system and initiates the LocalClear phase on that node. This is formally represented by a change in the meta-actor configuration.

10.6.5 GC Step 4: LocalClear Phase

$$\alpha_b = \langle\ \mathcal{S}_{b4},\ \overline{Acq},\ \overline{Acq^{-1}},\ \chi,\ \beta \rangle$$

where

$$\mathcal{S}_{b4} = \langle\ \mathcal{OS},\ \delta,\ \lambda,\ \mathcal{CC} \rangle \text{ and}$$

$$\text{if } \lambda = fss,\ \mathcal{OS} = t \text{ and}$$

$$\text{if } \mathcal{OS} = t,\ \lambda = fss \text{ and}$$

$\{\mathcal{S}_{b4} = \langle\ \mathcal{OS},\ \delta,\ \lambda,\ \mathcal{CC} \rangle \text{ — } \lambda = pss \text{or } \lambda = ns\} = \phi$, i.e, there are no untouched objects in Scavengespace.

Channel Configuration:

$$\gamma^i = \langle\ \mathcal{S}_\gamma, \overline{\mu},\ Source,\ Destination\ \rangle \text{ where}$$

$$\overline{\mu} \text{ is such that for every message} \in \mu$$

$$\mathcal{S}_{\mu 1} = (\tau = o,\ b\alpha)$$

Lemma 4.1 : *The LocalClear Phase of the algorithm collects all and only garbage (untouched) scavengespace objects on a node.*

Proof:

The meta-actor on every node has a list of base actors residing on that node as well as the physical offsets in local memory to access the base actors. During the execution of the LocalClear Phase, all the actors residing on that node are scanned.

If the node is in Scavengespace and it has its object-status field set to *touched* ($\mathcal{OS} = t$), the object is not collected during Phase. By Lemmas 2.1 and 2.3, all reachable objects in Scavengespace will be touched ($\mathcal{OS} = t$) and have their generation tag set to FutureTag ($\lambda = fss$). This proves that only the garbage objects on a node will be collected by the LocalClear Phase.

Furthermore, if the object is in Scavengespace and is not reachable, it is a garbage node and by Lemma 2.2, it will be left untouched by the DistributedScavenge Phase. The LocalClear Phase collects all Scavengespace objects that are untouched. Therefore, all garbage objects on a node will be collected by the LocalClear Phase.

Corollary 4.1.1 : *The LocalClear Phase of the TGC algorithm collects all and only garbage (untouched) objects in Scavengespace and Oldspace on a node.*

Lemma 4.2 : *The LocalClear Phase terminates.*

Proof:

There are a fixed number of PU's and entries in the global nametable. Once the LocalClear Phase has been initiated on every node, each node cleans up garbage objects in its local memory according to Lemma 4.1. In addition, using the same argument as Lemma 4.1, all garbage entries in the nametable are removed. As the number of actors on a node and the number of nametable entries are finite, the clearing process takes a finite amount of time. Therefore, the local actions in the LocalClear Phase terminate on every node.

We need to prove that the LocalClear Phase terminates in the entire system and this can be detected in finite time. A bulldoze wave collecting the completion information is initiated at the start node of the system. Propagation of the bulldoze message from a node occurs when the node has completed its LocalClear Phase. As the local phase takes finite time, every node propagates the bulldoze wave and it eventually reaches the finish node. LocalClear Phase termination occurs when

- The finish node has completed its LocalClear Phase.
- The finish node has received the LocalClear Phase completion bulldoze message from all its Bneighbors

This process can be represented in the meta-actor configuration of the system.

10.6.6 GC Step 5: Post GC Sweeps

This set of messages tells every node in the system that the LocalClear Phase of has been completed on every other node and that the *Global-GC-is-on* flag can be turned off.

Lemma 5.1 : *The termination of a GC has been correctly detected and all the nodes in the system have been informed of the same.*

Proof:

It is possible to determine the point at which the LocalClear Phase has been completed (Lemma 4.2). The first step in the Post-GC Sweep is a broadcast that informs every processing unit that GC has completed. On receiving this message, every processing unit resets the local variable *GC-Phase* to 0. Each node also flips the interpretations of the following fields:

- the object-status bits corresponding to touched objects and untouched objects.
- Logical Memory area tags - FSS-Tag and PSS-Tag.
- Old Message Tag and New Message Tag.

When the broadcast is completed and all the nodes have reset their *GC-Phase* flags to 0, the global flag *GC-is-on* must be reset to False.

In terms of the abstract model, we will represent the broadcast wave as a transition in the meta-actor configuration.

Chapter 11

Conclusions and Future Research Directions

11.1 Thesis Conclusions

In this thesis, we have brought out some of the critical issues in the design of scalable, parallel systems. We highlight the need for a universal model of parallel computation using which we can express different forms of parallelism without any concern for the architecture of the resource management system or the hardware. Our current focus in the area of universal concurrency is the development of powerful meta-abstractions and meta-level programming constructs. Also, in order to achieve high performance computing, new techniques must be developed for managing the computational resources of large numbers of concurrent processors. We have proposed a novel resource management technique for scalable, parallel systems called *hierarchical distributed resource management*. In a hierarchical scheme, the parallel system is divided into clusters which in turn are divided into smaller subclusters. The cluster manager at a higher level in the hierarchy delegates the responsibility (relinquishes authority) of resource management to cluster managers lower in the hierarchy. In this thesis, we specifically deal with efficient memory management. The underlying premise of our MM mechanism is that, for a parallel system to be architecturally scalable, memory must be physically distributed. We have dealt with object management in a distributed actor based environment. The memory

management scheme designed has a tiered naming strategy that reflects the hierarchical nature of the system.

We have identified and defined the problem of object reclamation in a distributed actor based environment. Noteworthy properties of the HDGC algo include:

- Non-stop, concurrent GC in a distributed environment
- Scalable GC techniques.
- Generational GC (which is conservative)
- Logical transfer of objects during a copying GC
- Possible extensibility to fault tolerant GC
- Possible extensibility to real time GC

Any mechanism for efficient GC in a large system must be conservative. Generational storage management techniques are conservative and they exploit characteristic reference patterns observed in many applications [Ung86]; we therefore believe that they are well-suited to machines with large numbers of processing elements. As we avoid physically moving objects across generations, this scheme also turns out to be less error prone because interprocessor management of forwarding pointers can get very complex and frustrating. With well engineered abstractions, the complexity of implementing the HDGC algorithm can be reduced.

In actor based systems, we must note that GC of actors involves more than data deallocation. An actor is the basic entity within which behavior (code), communication information (mail addresses in the acquaintance lists, and mail addresses in the mail queue) and task processing information is embedded. Thus, deletion (GC) of an actor involves:

- Deletion of a process or a set of processes
- Deletion of any communication associated with that process
- Deletion of memory objects.

When an actor is deleted, all resource management responsibilities associated with an actor disappear. Memory Management in Actors is more than a data management facility, it is a process management facility as well.

11.2 Optimizations and Further Work

What we have avoided in this thesis is a detailed discussion of optimizations to the HDGC scheme. These optimizations come in many flavors; hardware, operating system, compiler and programming language optimizations and must be considered for an efficient, realistic implementation. Some designs, for eg., Cantor [Ath87] assign optimization responsibilities to a specific level in the system: to the programmer, compiler or run-time system. We are reluctant at this point to commit optimization tasks to *one* level in the system; this process cannot be isolated and solved at one level. A consideration of various deficiencies of the HDMM scheme has revealed some optimizations to overcome the time and space overheads encountered in translation and bookkeeping. They are:

Avoid multiple tier addressing scheme: Ideally, the impact of address translation on object access should be minimal. But, in practice, object migration and system recofigurability make this hard to achieve. By using an address caching mechanism or by clever compilation, we can avoid the multi-level access that may be necessary on every non-local reference.

Decrease bookkeeping information: The HDMM scheme requires a significant amount of bookkeeping to implement the naming hierarchy as well as maintain the translation tables on object migration and compaction. It may be possible to derive information by a static analysis of the program text that would help in reducing the amount of bookkeeping information that needs to be maintained.

Architecture specific optimizations: We have earlier discussed the implications of architectural parameters like the locality, load distribution on the performance of a scalable system. These parameters can be manipulated to suit the specific requirements of an architecture.

In addition to possible optimizations, this study has also brought to the surface many interesting issues:

Compaction of memory to obtain locality: It is not clear to what extent we would like to exploit global locality in a scalable machine. Particularly, in the case of a system

designed for fine-grained parallelism, global compaction reduces parallelism. The hierarchical scheme with region specifications can be used to exploit locality at different levels. Regions of processors in the architectural configuration are grouped together to define a *locality scope*. Processes that communicate frequently can be placed on surrounding processors in the same scope without sacrificing concurrency.

Static Analysis: We can statically extract information used in optimal actor allocation and placement, lifetime analysis techniques and so on.

Garbage collection of persistent objects: Would it be possible to extend the same technique for secondary storage garbage collection? What are the advantages of having a uniform memory management scheme over an integrated scheme where different reclamation techniques are applied to the different components of the memory hierarchy.

Integrated garbage collection schemes: Here, different levels of the memory hierarchy, i.e, cache, main memory and secondary memory. have their own GC schemes, possibly different. Decision criteria for appropriate algorithms are the amount of memory to be collected in the memory level as well as time constraints that must be met in maintain reasonable response times. A uniform representation of a memory object for all levels in the hierarchy is required.

Fault Tolerant Memory Management: As the size of computational systems increase, so does the possibility of failure. It is no longer pragmatic to base the correctness and functionality of a computational model on the assumption that processing units and communication links will never fail. The resilience of a system or its ability to tolerate failures plays a critical role in actual implementations. A number of failure models have been proposed in literature: Failstop, crash, omission and Byzantine models are a few. Fault tolerant GC in distributed systems using the actor model requires the conception and evolution of an actor-based fault model to reason about failures. Such a model should essentially encompass faults of the following kinds:

- Omission failures
- Timing Failures
- Authentication Detectable and unauthenticated Byzantine failures

Fault tolerance in the HDRM scheme was discussed in Chapter 8.

Hierarchical Distributed Resource Management: Algorithms for other resource management purposes like scheduling and load balancing on the hierarchical system must be developed. Hierarchical sponsorship of speculative work is one of the areas we are currently investigating.

Real Time GC in a Distributed System: A total realtime collection should collect the garbage objects generated during a collection phase. The advantage of having such a facility in a massively parallel system is questionable; it appears that the time and space overhead encountered in a total realtime collection is both unnecessary and inefficient. Realtime behavior is desirable to the extent that we be able to predict the response times within some margin of accuracy; we have described how the HDRM schemes can be tailored to display pseudo-realtime behavior.

Comprehensive Analytical Models and Performance Analysis: Once an optimized implementation is available, we are faced with the task of performance evaluations and comparisons of the HDGC scheme with other existing methods. Any reasonable analysis will require a comprehensive, accurate modeling of the overheads and costs encountered during execution.

Harnessing the knowledge and experience from multiprocessing research over the past two decades, we have realized the importance of organization and management in parallelism. We believe that the ability to design efficient, scalable, concurrent systems does not lie in esoteric programming paradigms and architectures that are difficult to comprehend. It lies in representing applications as well classified, intuitive specifications and organizing hardware resources to render flexible and manageable concurrency using natural strategies such as *hierarchical resource management.*

BIBLIOGRAPHY

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chishlom, P.W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26:360–365, april 1983.

[ABC+84] M.P. Atkinson, P.J. Bailey, K.J. Chishlom, W.P. Cockshott, and R. Morrison. Databases - role and structure: An advanced course. In *Progress with Persistent Programming*. Cambridge University Press, 1984.

[AC86] Arvind and David E. Culler. Dataflow architectures. In *Annual Review of Computer Science*, pages 225–253. Annual Reviews Inc., 1986.

[AEL88] Andrew Appel, John Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *SIGPLAN '88 Conference on Programming Language Desig!e n and Implementation*, pages 11–20, Atlanta, GA, June 1988. SIGPLAN, ACM Press.

[AFO89] Ole Agesen, Svend Frolund, and Michael Hoffmann Olsen. Persistent and shared objects in beta. Master's thesis, Aarhus University, Computer Science Department, DK 8000 Aarhus C-Denmark, April 1989.

[Agh85] G. Agha. A message-passing paradigm for object management. *IEEE Database Engineering Bulletin*, Dec. 1985.

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[Agh90] G. Agha. Concurrent object oriented programming. *Comm. of the ACM*, 33(9):125–141, 1990.

[AH87] G. Agha and C. Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In B. Shriver and P. Wegner, editors, *Research Directions in Object Oriented Programming*, Cambridge, Mass, 1987. MIT Press.

[AL91] Andrew Appel and Kai Li. Virtual memory primitives for user programs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, California, April 1991. ACM.

[Ali84] Khayri Mohamed Ali. *Object Oriented Storage Management and Garbage Collection in Distributed Processing Systems.* PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, December 19984.

[ANP87] Arvind, Rishiyur Nikhil, and Keshav Pingali. I-structures: Data structures for parallel computing. Technical Report Computation Structures Group Memo 269, Massachusetts Institute of Technology, Cambridge, MA, February 1987.

[App89] Andrew Appel. Simple generational garbage collection and fast allocation. *Software - Practice and Experience*, 19(2):171–183, February 1989.

[Ath87] W. Athas. *Fine Grain Concurrent Computations.* PhD thesis, Computer Science Department, California Institute of Technology, 1987. Also published as technical report 5242:TR:87.

[AWY89] G. Agha, Peter Wegner, and Aki Yonezawa, editors. *Proceedings of the NSF Workshop on Object-Based Concurrent Programming.* ACM, April 1989. Special Issue of SIGPLAN Notices.

[BA84] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, 1984. correctness-proof of a mark-and-sweep based algorithm using temporal logic.

[Ben90] Mats Bengtsson. *Real Time Compacting Garbage Collection Algorithms.* PhD thesis, Department of Computer Science, Lund University, Lund, Sweden, 1990.

[Bev87] D. I. Bevan. Distributed garbage collection using reference counting. In *Parallel Architectures and Languages Europe, LNCS 259*, pages 273–288. Springer-Verlag, June 1987.

[BH77] Henry Baker and Carl Hewitt. The incremental garbage collection of processes. Technical Report Memo AI-454, Massachusetts Institute of Technology, M.I.T Artificial Intelligence Laboratory, December 1977.

[Bjo90] Anders Bjornerstadt. *Secondary Storage Garbage Collection for Decentralized Object Based Systems.* PhD thesis, Department of Computer and System Sciences, The Royal Institute of Technology, Stockholm University, Sweden, June 1990.

[BVN91] Francoise Baude and Guy Vidal-Naquet. Actors as a parallel programming model. In *Proceedings of the 8th Symposiumon Theoretical Aspects of Computer Science*, pages 184–195. Lecture Notes in Computer Science, No. 480, Springer Verlag, February 1991.

[BW88] Hans-Juergan Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software- Practice and Experience*, 1988.

[CAC+84] W.P. Cockshott, M.P. Atkinson, K.J. Chishlom, P.J. Bailey, , and R. Morrison. Persistent object management system. *Software - Practice and Experience*, 14, January 1984.

[Che70] C. J. Cheney. A nonrecursivelist compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[Che88] David Cheriton. The v distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[Chi90] Andrew Chien. *Concurrent Aggregates (CA): An Object-Oriented Language for Fine-Grained Message Passing Machines.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, July 1990.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[Cli81] W. D. Clinger. Foundations of actor semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory, May 1981.

[CM86] K.M. Chandy and J. Mishra. An example of stepwise refinement of distributed programs: Quiescence detection. *ACM Transactions on Programming Languages*, 8(3):326–343, July 1986.

[CML83] K.M. Chandy, J. Mishra, and L.Haas. Distributed deadlock detection. *ACM Transactions on Computing Systems*, 1(2):144–156, May 1983.

[Cou88] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[Dal86] W. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Press, 1986.

[Dal90] W. Dally. The j-machine: System support for actors. In Hewitt and Agha, editors, *Towards Open Information Systems.* MIT Press, 1990. To be published.

[DB76] Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[DCC+87] W. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty, and Scott Wills. Architecture of a message driven processor. In *14th Annual Symposium of Computer Architecture*, 1987.

[DFVG83] E. W. Dijkstra, W. H. J. Feijen, and A.J.M. Van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16:217–219, 1983.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A.J. Martin, C.S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[DW89] W. Dally and Scott Wills. Universal mechanisms for concurrency. In *Parallel Architecture and Languages Europe 1989, LNCS 365*, pages 19–33, Eindhoven, The Netherlands, June 1989. Springer-Verlag. Volume 1, Parallel Architectures.

[DWH+90] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 261–269, San Francisco, California, January 1990. SIGACT, SIGPLAN, ACM Press.

[Fra80] N. Francez. Distributed termination. *ACM Transactions on Programming Languages*, 2(1):42–55, 1980.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage collector for virtual memory systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[Gab85] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press Series in Computer Science. M.I.T. Press, Cambridge, Massachusetts, 1985.

[GMN+88] A. Goto, M.Sato, K. Nakajima, K. Taki, and A. Matsumoto. Overview of the parallel inference machine architecture. In *Fifth Generation Computing Systems*, pages 208–229, 1988.

[Gol89] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 313–320, Portland, OR, June 1989.

[Gri77] David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, pages 921–930, December 1977.

[GS80] V.D. Gligor and S. H. Shattuck. Deadlock detection in distributed systems. *IEEE Transactions on Software Engineering*, 6(5):435–440, September 1980.

[H+87] Norman Hutchinson et al. The emerald programming language report. Technical Report 87-10-07, University of Washington, Seattle, Washington, October 1987.

[Hal78] Robert H. Jr. Halstead. Multiple processor implementations of message passing systems. Technical Report 198, Massachusetts Institute of Technology, April 1978.

[Hal79] Robert H. Jr. Halstead. Reference tree networks: Virtual machine and implementation. Technical report, Massachusetts Institute of Technology, July 1979.

[Hal85] Robert Jr. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HB77] C. Hewitt and H. Baker. Laws for communicating parallel processes. In *1977 IFIP Congress Proceedings*, pages 987–992. IFIP, August 1977.

[HCD89] Waldemar Horwat, Andrew A. Chien, and William J. Dally. Experience with cst: Programming and implementation. In *Proceedings of Sigplan '89 Conference on Programming Language Desgin and Implementation*. ACM, 1989.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8-3:323–364, June 1977.

[Hew80] Carl Hewitt. The apiary network architecture for distributed systems. In *Conference Record of the 1980 LISP Conference*, August 1980.

[HK82] Paul Hudak and Robert M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *ACM Symposium on LISP and Functional Programming*, pages 168–178, 1982.

[HM90] Maurice Herlihy and Eliot B. Moss. Non-blocking garbage collection for multiprocessors. Technical Report CRL 90/9, Digital Equipment Corporation, Cambridge Research Laboratory, November 1990.

[HTD90] Waldemar Horwat, Brian Totty, and William J. Dally. Cosmos: An operating system for a fine-grain concurrent computer. In Hewitt and Agha, editors, *Towards Open Information Systems*. MIT Press, 1990. To be published.

[Hut87] Norman Hutchinson. *Emerald: An Object Based Language for Distributed Programming*. PhD thesis, University of Washington, Seattle, Washington, 1987. TR-87-01-01.

[IY90] Yuugi Ichisugi and Akinori Yonezawa. Distributed garbage collection using group reference counting. Technical report, University of Tokyo, Dept. of Information Science, 1990.

[Jul89] Eric Jul. *Object Mobility in a Distributed Object-Oriented System*. PhD thesis, University of Washington, 1989.

[KKR+86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudakand James Philbin, and Norman Adams. Orbit: An optimizing compiler for scheme. In *Sigplan Symposium on Compiler Construction*. ACM, 1986.

[KR87] Vipin Kumar and V. Nageshwara Rao. Parallel depth first search, part ii: Analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[KR89] Leslie J. Keedy and John Rosenberg. Support for objects in the monads architecture. In *Proceedings of the Third Workshop on Persistent Object Systems*, pages 202–213, Newcastle, Australia, January 1989. Springer-Verlag.

[KS77] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. Technical report, Carnegie-Mellon University, Pittsburgh, September 1977.

[KT90] Hanpei Koike and Hidehiko Tanaka. Generation scavenging gc on distributed-memory parallel computers. In *High Peformance and Parallel Computing in Lisp*, U.K., 1990.

[KWN90] Dennis Kafura, Doug Washabaugh, and Jeff Nelson. Garbage collection of actorrs. In Norman Meyrowitz, editor, *1990 ECOOP/OOPSLA Proceedings*, pages 126–134, Ottawa, Canada, October 1990. ACM Press.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[LL86] Barbara Liskov and R. Ladin. Highly available distributed services and fault-tolerant garbage collection. In *Fifth ACM Symposium on Distributed Computing*, pages 29–39, Calgary, Canada, August 1986. ACM. Fault-tolerant GC.

[LLG+89] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennesey, M. Horowitz, and M. Lam. Design of the stanford dash multiprocessor. Technical Report CSL-TR-89-403, Stanford University, Palo Alto, California, December 1989.

[Low88] C. Low. A shared, persistent object store. In *ECOOP '88 European Conference on Object-Oriented Programming*. Springer-Verlag, 1988.

[MA90] R. Morrison and M.P. Atkinson. Persistent languages and architectures. In *Persistence and Security, Bremen*. Springer-Verlag, 1990.

[Moo84] David A. Moon. Garbage collection in a large lisp system. In *Conference Record of the 1984 Symposium on Lisp and Functional Programming*, pages 235–26, Austin, Texas, August 1984.

[Moo85] David A. Moon. Architecture of the symbolics 3600. In *Proceedings of the Twelfth Symposium on Computer Architecture*, Boston, Massachusetts, June 1985.

[Nel89] Jeff Nelson. Automatic, incremental, on-the-fly garbage collection of actors. Master's thesis, Virginia Tech, Dept. of Computer Science, Blacksburg, VA, February 1989.

[PS89] C.J. Peng and Gurinder Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, University of Wisconsin-Madison, Dept. of Computer Science, July 1989.

[PU88] Joseph Pallas and David Ungar. Multiprocessor smalltalk: A case study of a multiprocessor-based programming environment. In *SIGPLAN Conferencce on Programming Language Design and Implementation*, pages 268–277. ACM, 1988.

[Sar86] John Sargeant. Load balancing, locality and parallelism control in fine-grained parallel machines. Technical Report UMCS-86-11-5, Dept. Of Computer Science, University of Manchester, 1986.

[SB88] Marcel Schelvis and Eddy Bledoeg. The implementation of distributed smalltalk. In *Proceedings of ECOOP 88*, 1988.

[SG90] M. Sarkar and Edward F. Gehringer. Dimensions of addressing schemes. In Rosenberg and Keedy, editors, *Security and Persistence*, 1990. Bremen.

[SGP90] Marc Shapiro, Olivier Gruber, and David Plainfosse. A garbage detection protocol for a realistic distributed object-support system. Technical Report 1320, INRIA, November 1990.

[SKAT91] Vineet Singh, Vipin Kumar, Gul Agha, and Chris Tomlinson. Scalability of parallel sorting on mesh multicomputers. In *Proceedings of the International Parallel Processing Symposium*, 1991.

[SS91] Ravi Sharma and Mary Lou Soffa. Parallel generational garbage collection. In *OOPSLA '91 Conference Proceedings*, pages 16–32. ACM Press, 1991.

[Ste75] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[TBLN85] Guy T.Almes, Andrew P. Black, Edward Lazowska, and Jerry Noe. The eden system: A technical review. *IEEE Transactions on Software Engineering*, 11(1):43–59, 1985.

[Tha86] S. Thatte. Persistent memory: Merging ai-knowledge and databases. *Texas Instruments Engineering Journal*, pages 151–159, January-February 1986.

[TLP86] David Thomas, Wilf R. LaLonde, and John R. Pugh. Actra - a mutitasking multiprocessing smalltalk. Technical Report SCS-TR-92, Carleton University, May 1986.

[Tot87] Brian Totty. An operating system for the jelly bean machine. Technical report, Massachusetts Institute of Technology, 1987.

[TS87] Charles P. Thacker and Lawrence C. Stewart. Firefly: A multiprocessor workstation. In *Proceedings of the Second International Conference on Architectural Support for Programmingy Languages and Systems*, pages 164–172. ACM, October 1987.

[Ung84] David M. Ungar. Generation scavenging - a non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments*, pages 157–167. Pittsburgh, PA, April 1984.

[Ung86] David M. Ungar. *The Design and Performance of a High Performance Smalltalk System.* PhD thesis, University of California, Berkeley, March 1986.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[Ves87] Stephan C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault-tolerant Programming.* PhD thesis, University of Washington, Seattle, January 1987.

[VMA91] Nalini Venkatasubramanian, Shakuntala Miriyala, and Gul Agha. Scalable concurrent computing. In *Jl. of the Indian Academy of Sciences*, 1991.

[Was90] Doug Washabaugh. Real-time garbage collection of actors in a distributed system. Master's thesis, Virginia Tech, Dept. of Computer Science, Blacksburg, VA, February 1990.

[Wil87] Ifor Williams. The mushroom machine - an architecture for symbolic processing. Technical report, Manchester University, 1987.

[Wil90] Donald Scott Wills. *Pi: A Parallel Architecture Interface for Multi-Model Execution.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, July 1990.

[WK90] Doug Washabaugh and Dennis Kafura. Incremental garbage collection of real concurrent objects for real-time applications. In *11th Real-Time Systems Symposium*, pages 21–30, Lake Buena Vista, Florida, December 1990. IEEE, IEEE Computer Society Press.

[WW87] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In *Parallel Architectures and Languages Europe, LNCS 259*, pages 432–443. Springer Verlag, 1987.

[WWH87] Ifor Williams, Mario Wolczko, and Trevor Hopkins. Dynamic grouping in an object oriented virtual memory hierarchy. In *1987 European Conference on Object-Oriented Programming*, pages 87–96, Paris, June 1987.

[Z+87] Benjamin Zorn et al. Spur lisp: Design and implementation. Technical Report UCB/CSD 87/373, University of California, Berkeley, 1987.

[Zor89] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California, Berkeley, December 1989.

APPENDIX A

Glossary of Terms and Definitions

Acquaintance List: List of mail addresses in every actor that represents the set of actors in the universe known by the actor. Note that every actor is its own acquaintance.

Actor: Individual agents of computation in a system distributed in time and space. Each actor is associated with a unique address called the mail address and a behavior.

Actor Model: This is a model of concurrent object oriented programming, where each of the individual entities are called actors that communicate via asynchronous message passing and is used to capture the essence of concurrent computation at an abstract level.

Active actor: An actor which is currently processing messages or has messages to process in its mail queue [KWN90].

Annotations: Hints provided to the runtime system that may be either user or compiler inserted, usually to increase efficiency..

Asynchronous Communication: This is a mechanism for interprocess communication where the sender and receiver can operate independently, the sender can send information without waiting for the receiver to be ready to accept the information.

Base-Actors: Actors that are created as a result of the execution of the application program.

Base-Actor Configuration: A snapshot of the sum total of all base-actors currently under consideration, which presents the programmatic aspect of a multicomputer system as a whole.

Base-Actor Initial Configuration: The configuration of base-actors in the system prior to any GC or in between two consecutive GCs in the system.

Bbulldozer messages: Network Clearance Messages initiated at the *Finish node* propagating along B-paths from every node to its B-neighbor.

Behavior: Set of procedures associated with an actor that determines processing to be done in an actor when it receives a message.

Birth Node: The birth node of an object is the node where the object is first created.

Inactive actor: An actor which is not active currently and has no messages in its mail queue [KWN90].

Bneighbor: A node $A(a_1, a_2, ..., a_n)$ in a multicomputer system is a bneighbor of a node $B(b_1, b_2, ..., b_n)$ if for any i, a_i is equal to $b_i - 1$ and $a_j = b_j for j <> i$.

Bpath: A path in the network that progresses only along Bneighbors.

Broadcast messages: Messages that are sent to every node in the network unlike the bulldoze messages that traverse every link in the network.

Buffered Asynchronous Communication: In asynchronous communication, the presence of buffers at the receiver allows us to store the messages arriving from different senders or the same sender. This kind of communication is called " buffered asynchronous communication".

Bulk Synchronization: A synchronization mechanism used in the BSP model of generalized parallel computation where a periodic synchronization signal is sent out to every processing unit in the system along a spanning tree. If synchronization is achieved, we can proceed with the next phase of computation. Otherwise, a new timeslot is allocated and the process is repeated till synchronization is achieved.

Bulk Synchronous Parallel Model: The Bulk Synchronous Parallel (BSP) Model was proposed as a suitable bridging gap between parallel languages and architectures [Val90] The BSP model consists of three units:

- Components – which are computation or memory units
- Router – responsible for point to point message passing
- Periodic Synchronization – that performs synchronization of all or a subset of the processors with a periodicity L.

Bulldozer messages: Messages that traverse every link in the network in a wavelike fashion beginning at one end of the system and terminating at the other end.

Channel: Unidirectional links of the network that communicate information. They connect one node in the multicomputer system to another node.

Channel Configuration: A snapshot of the sum total of all messages currently in transit, which presents the commmunication aspect of a multicomputer system as a whole.

Channel Initial Configuration: The configuration of the channels (messages in the channels) prior to any GC or in between two consecutive GCs in the system.

Cluster: In a hierarchical organization, the distributed (highly parallel) system which consists of many processing units is refined into smaller subunits (also called *regions* or *clusters*) containing fewer processing units.

Cluster Manager: The responsibility of coordinating task execution within a cluster in a hierarchical system is assigned to the *cluster manager* (CM), which could be a single processing unit or a group of processing units.

Collector: The collector is a system task that is responsible for keeping track of accessible locations in memory and reclaiming unused storage.

Conservative GC Algorithm: - only a subset of inaccessible objects are identified during a GGC. This processis also sometimes referred to as a partial GC.

Concurrent Garbage Collection: This is the term used to refer to the process where the garbage collection tasks proceed concurrently with the mutator tasks. There may be a single mutator or multiple mutators.

Copy-count: This is a proposed solution to the Tenuring problem and it involves the use of a counter where objects are promoted to the next generation after they have survived predetermined number of scavenges in one generation. It is also referred to as the *threshold age*.

Dimension-order Routing: A scheme of routing where messages are routed in a specified order of dimension (usually increasing or decreasing).

Distributed Garbage Collection: The term used to refer to the process of garbage collection in a distributed system.

Distributed Generation Scavenging (DGS): An extension of the *generation scavenging* algorithm adapted to perform efficient storage reclamation on distributed systems with a shared object namespace.

Distributed Memory Machines: Computer systems that consist of multiple processors with their own memory connected by a message-passing network.

Distributed Nametable: Object name to virtual address translation is accomplished via a translation table called the Nametable. If the nametable is distributed across all the processing units in the system, we have a *distributed nametable*.

Distributed Scavenge Phase: The first phase of the global garbage collection algorithm that traverses and processes all reachable objects in the system starting from a predefined *root set*.

Distributed Shared Memory: A memory model where a shared address space is physically distributed across the local memories of distinct processing units.

Distributed System: An informal description of a distributed system has a collection of autonomous processing entities that do not physically share a common address space, have no notion of a global time and that communicate via message passing. A finite set of such processes is mapped on the set of processing units.

Dynamic Resource Management: A technique where the management of the computational resources of a system is performed at runtime.

Ephemeral Garbage Collection: A variation of the generational garbage collection mechanism adopted in the Symbolics Lisp machines. This strategy sorts objects on the basis of their expected lifetimes and focuses garbage collection efforts on objects that have a high probability of being garbage (ephemeral objects).

Event Diagrams: A graphical representation of actor execution that illustrates arrival orders of events within a thread of execution and causal dependencies between different threads.

Fbulldozer messages: Network Clearance Messages initiated at the *Start node* propagating along F-paths from every node to its F-neighbor.

Fine Grained Computation: A computation scheme where concurrency is exploited by executing a large number of small (fine-grained) tasks in parallel.

Finish Node: The node from which there exists an B-path to every other node in the system.

Fneighbor: A node $A(a_1, a_2, ..., a_n)$ is a Fneighbor of a node $B(b_1, b_2, ..., b_n)$ if for any i, a_i is equal to $b_i + 1$ and $a_j = b_j for j <> i$.

Fpath: A path in the network that progresses only along Fneighbors.

Fromspace: In the stop-and-copy GC algorithm, fromspace contains both accessible and inaccessible objects. All accessible objects are moved from Fromspace to Tospace.

FutureSurvivorSpace: In Ungar's Generation Scavenging algorithm, new objects can be in one of the three areas that are actively involved in the garbage collection process: Newspace, PastSurvivorSpace and FutureSurvivorSpace. This region is empty during program execution and is used during scavenging to copy live objects.

FutureTag: Generation tag associated with an object in FutureSurvivorSpace.

Garbage: Memory which is not accessible is referred to as *garbage*

Garbage Actor: An actor which is not reachable is a garbage actor.

Garbage Collection: The reclamation of inaccessible memory (garbage) is called *garbage collection.*

GC Messages: These are garbage collection messages. They are of two kind : GC_{acq} messages sent to the acquaintances of a node and GC_{invacq} messages sent to the inverse acquaintances of a node. $GC - acq$ messages always make the target node *touched*. When a GC_{invacq} message is sent to an inactive node, the target node is suspended and when it is sent to an active node, the target node becomes *touched*.

Generational Garbage Collection: In this garbage collection mechanism, records with a similar age are grouped together in a contiguous area of memory. Once we have segregated objects on the basis of their age, reclamation efforts can be concentrated on the youngest generation.

Global Compaction: Compaction of the address space across processing units as opposed to local compaction where we compact a contiguous local memory area.

Global Garbage Collection (GGC): A systemwide garbage collection involving all processors and processes in the system.

Global Time: A uniform notion of time across multiple processing units. With global time, it is possible to define an order on events occuring on all processing units.

Guarantee of Delivery: A property by virtue of which messages in the system are guaranteed to eventually reach their destination. This implies that intransit mail cannot be indefinitely buffered.

Hierarchical Distance: The *hierarchical distance* between two subregions (or processing units) is the number of levels that must be ascended in the hierarchy before the two subregions(or processing units) belong to the same region.

Hierarchical Distributed Garbage Collection: The Hierarchical Distributed Garbage Collection scheme is an extension of Ungar's Generation Scavenging algorithm to multicomputers. The distributed system is organized into hierarchies of clusters andh the crux of the algorithm lies in obtaining a consistent state of the machine without disrupting ongoing computation.

Hierarchical Naming Schemes: Typically, a hierarchical name consists of many fields concatenated together. Information is already present in the structure of the name. In order

to access an object referred to by a hierarchical name, the fields of the name must be translated to obtain a physical address.

Inverse Acquaintance List: An actor B is an inverse acquaintance of an actor A if A is on the acquaintance list of B.

Isoefficiency Metric: The isoefficiency metric [KR87] evaluates a parallel algorithm on the basis of the speedup obtained. This metric is based on the assumption that the entire machine is dedicated to executing this single job and the time taken for completion is a measure of performance.

J-Machine: The Jelly Bean Machine [Dal90] is a scalable, highly parallel fine-grained MIMD machine consisting of special purpose processors that are loosely coupled by means of a high-speed, low-latency network. The J-Machine is designed to provide low-overhead system services to support the actor programming paradigm [Agh86].

Join Continuations: Join Continuations are a synchronization mechanism used in Actor based languages. The join continuation function allows progress of two or more threads of control after the synchronization point.

Load Balancing: Load balancing is the task of keeping the processors of a parallel machine uniformly busy.

Local Clear Phase: The second phase of the global garbage collection algorithm in which every processor clears all inaccessible objects in its local memory.

Local Compaction: Compaction of a contiguous address space located within a single processing unit.

Local Nametable: The translation of a virtual address of an object on a processor to an actual physical address is accomplished via a local translation table on a processor, called the *local nametable*.

Mail Address: Conceptual location associated with an actor.

Mail Queue: Queue associated with every actor that holds messages that need to be processed by the actor.

Mark and Sweep Collectors: The *Mark and Sweep* algorithm detects garbage by halting the mutator (application program) and then performing two (optionally three) phases. A predetermined root set is used to *mark* all the reachable objects in the first phase, the second phase reclaims dead objects one at a time by walking down the entire memory space and a third, optional phase compacts memory to avoid fragmentation.

Memory Locality: Memory locality is a property of the pattern of a program's references to memory.

Message Dispatch and Reception Structures: The data structures responsible for message handling.

Message Passing: A mechanism of communication between individual entities in a distributed system where point-to-point messages are sent from source to destination.

Meta-Actors: Actors that exist in the system as part of the resource management system. These actors manage the base-actors and they can access the protected information within a base-actor that other base-actors cannot access.

Meta-Actor Configuration: A snapshot of the sum total of all meta-actors currently under consideration, which presents the computational aspect of a multicomputer system as a whole.

Meta-Actor Initial Configuration: The configuration of meta-actors in the system prior to any GC or in between two consecutive GCs in the system.

Migration: *Migration* refers to the movement of heavyweight processes and their associated address spaces from one node to another. Migration is usually initiated for the purpose of load balancing either by the system or by the user. The move of the object or process is transparent to other processes in the system.

Multicomputer: Multicomputers are distributed memory multiprocessors that use a large number of small programmable computers (processors with their own memory) which are connected by a message-passing network.

Multicomputer Configuration: A snapshot of the sum total of the entire state of the system that encompasses both the computation and communication activities in the system.

Multicomputer Initial Configuration: The configuration of the multicomputer system prior to any GC or in between two consecutive GCs in the system.

Multicomputer Transition: A transition function that operates on a multicomputer configuration and returns a new multicomputer configuration.

Mutator: The user task that corresponds to the application program that reference and updates memory objects.

New Messages: New Messages from objects that have been touched during GGC (???? Untouched nodes may also send black messages if they receive a black message.

Network Clearance Algorithm: The network clearance algorithm makes the following assumptions about

Newspace: In Ungar's Generation Scavenging algorithm, new objects can be in one of the three areas that are actively involved in the garbage collection process: Newspace, PastSurvivorSpace and FutureSurvivorSpace. All new objects are created here. This is a comparatively large area.

New-Tag: Generation tag associated with an object in Newspace.

Non-halting Algorithm: - An algorithm where the processors in the system need not halt computation and synchronize to start a global garbage collection.

Non-stop Algorithm: - An algorithm where a processing unit continues with normal processing (computation) both before and after it receives and processes a global GC message.

Object: A computational entity that encapsulates both code and data.

Object Mobility: The property by virtue of which an object is able to migrate from one processing unit to another in the parallel system. This implies that any reference to the migrated object are automatically rerouted.

Object Name: When we have a uniform address space across all nodes in a network, all objects (entities) in the system are referred to by a name that is uniform over the entire system. This name is called the object name. The object name of an object remains unchanged giving us a uniform namespace over the entire system.

Old Messages: Old Messages are messages from a node prior to its participation in a garbage collection.

Oldspace: In a generational algorithm, objects are categorized as being either old or new. Old objects reside in an *old* or tenured space and do not participate in every marking and scavenging process.

Oldspace-Tag: Generation tag associated with an object in Oldspace.

Open Systems: Open systems are information systems which permit continuous influx of input from new and different sources. An open system has the following attributes:

1. Decentralization
2. Inherent concurrency
3. Organizational cooperation

Parallel Machine Interface: A generalized machine model that has been proposed to support various parallel models of computation [DW89]. This is an architecture oriented machine model that deals with efficient hardware implementations of mechanisms basic to concurrency like communication, synchronization and naming.

PastSurvivorSpace: In Ungar's Generation Scavenging algorithm, new objects can be in one of the three areas that are actively involved in the garbage collection process: Newspace, PastSurvivorSpace and FutureSurvivorSpace. The PastSurvivorSpace has objects that have survived previous scavenges.

PSS-Tag: Generation tag associated with an object in Past Survivor Space.

Pending messages: are messages in the network that may be computation messages that will affect the global GC phase or scavenge messages that contain the name of an accessible object, from where further scavenge messages may be generated.

Pending References: A pending message carrying an object address can introduce a new link in the graph and must be accounted for prior to initiation of GGC.

Permanently Inactive Actor: An inactive actor which has no inverse acquaintances is a *permanently inactive actor.*

Persistent Objects: *Persistent* objects are objects that can be shared among program executions, i.e., the object has the ability to participate in more than one program execution during its lifetime [AFO89]. Many definitions of persistence have been proposed [ABC+84, Low88] that look at persistence of an object as a measure of its lifetime. Commonly, a *persistent object* is an object that outlives the program that creates it.

Physical Address: The physical address is the actual memory offset on a particular processor where the object resides.

Potential References: These include references or acquaintances that do not currently exist, but could possibly exist.

Postconditions: Conditions that the system must satisfy subsequent to the defined point in time, or subsequent to the execution of a specified event.

Promotion Policies: Policies that are used to promote objects from a lower to a higher generation. The policies may be fixed or may dynamically vary.

Preconditions: Conditions that the system must satisfy prior to the defined point in time, or prior to the execution of a specified event.

Progress-only Routing: In this scheme of message routing, whenevera message is sent from point A to point B in the network, it moves closer to its destination at every hop.

Reachable Actor: The set of reachable actors is defined recursively as:

- A root actor is a reachable actor.
- Every forward-acquaintance of a reachable actor is reachable.
- Every inverse-acquaintance of a reachable actor that is active or is not permanently inactive is reachable.

Real-time Memory Management: A memory management mechanism is said to be real-time if the required to perform all operations in this scheme like object creation, naming and garbage collection is bounded by a constant.

Receptionists: Actors or agents that can receive communication. In a hierarchical configuration of the processing units in the system, every level in the hierarchy is allocated to a

receptionist that is responsible for handling messages sent to and received from members outside its hierarchy. This plays an important role in overriding regional acquaintances during collections that occur at a higher level in the hierarchy.

Reference Counting: This is a garbage collection mechanism where every cell (object) in memory is associated with a field called the *reference count* which is a count of the number of references to the cell. The reference count of a cell is continuously updated as pointers to the cell are created or destroyed. When the reference count of a cell becomes 0, the object is no longer referenced and can be collected.

Remembered Set: The Remembered Set has scavengespace entries pointed to by Oldspace objects

Remote Acquaintance Set: Holds the remote acquantainces of objects on that PU;

Replacement Behavior: When an actor processes a message, its behavior can be altered. This new behavior, called the *replacement behavior* decides how the $n+1^th$ communication will be processed.

Resource Consumption Metric: A metric for resource utilization in a parallel system, used to determine which resources are frequently used in a time-shared mode and those which remain idle.

Root Set: The root set is a set of actors that represent

- the current state of the system (environment variables, control structures like stacks etc.)
- the remembered set (in a generational scheme)
- the persistent set.

Root actor: A root actor is an actor that is a member of the *root set*.

Semispace Garbage Collector: The Semispace garbage collector divides the available memory in a system into two regions – the *fromspace* and the *tospace*. Only one of these regions, i.e., the fromspace, is available to the mutator at any time . The copying algorithm [Che70, FY69] traverses all the reachable records in the from-space starting at a root

pointer set and copies these records into a separate area of memory, the tospace, that is currently unused by the mutator. Forwarding pointers are placed in the from-space to redirect any references to the swapped record. At the end of collection, the only records in to-space are the reachable ones. The from and to-space pointers are swapped and the mutator now works on the new from-space.

Synchronous Communication: This is a mechanism for interprocessor communication where both the sender and receiver must be available before communication begins. A real life example of synchronous communication is a telephone that waits for users at both ends to be available at the same time before conversation can occur.

Start Node: The node from which there exists an F-path to every other node in the system.

Suspended Node: An inactive node that is non-garbage if it can potentially become active.

Shared Memory Machines: Shared memory computers have multiple processors and share a global memory. For efficiency reasons, each processor also has a local cache.

Spatial Locality: References which are tightly grouped together in terms of addresses are said to have spatial locality.

Scalability: Scalability, in the context of parallel computation, implies that given a program with sufficient parallelism, it will be possible to increase the performance of the system by adding physical resources.

Sticky Object: When allocating an object that is long-lived, we annotate it to be a sticky object. As this information is available statically, we allocate the initial portion of the heap to the persistent and sticky objects and divide the remaining heap space among other generations as required.

Scavengespace: The portion of memory that is actively considered during garbage collection is called Scavengespace. It is further divided into Newspace, PastSurvivorspace and FutureSurvivorSpace.

Temporal Locality: References close together in time are said to have temporal locality.

Tenuring: In generational garbage collection schemes, once an object has survived for a sufficiently long period in one generation, it is promoted to the next higher generation. This process is called Tenuring.

Tenuring Problem: The generational garbage collection scheme face s a drawback known in garbage collection literature as the *Tenuring Problem.* It is possible for an object to die as soon as it has been promoted into the higher generations. Thus, old objects that die may take a long time to be reclaimed. Even worse, they may never be reclaimed until the occurrence of an offline collection.

Threshold age: See Copy-count.

Touched Node: An accessible node which has participated in Phase 1 of a GGC.

Tospace: In a semispace collector, the region of memory into which accessible objects are moved is called *tospace.*

Transition Semantics: This is a formal specification of how one system configuration is replaced by another when a particular task is processed in the system.

Universal Concurrency Models: Models which view parallel computation as a set of abstractions that capture the semantics and functionality of concurrent program execution. This problem has been approached from different perspectives and various models of concurrency have been proposed by language theorists, complexity analysts and architects of parallel machines.

Untouched Node: A node which has not yet participated in GGC. This node may be accessible or inaccessible.

Virtual Address: Every object in the system has a virtual address that is comprised of the processor number on which the object currently resides as well as a relative address on that processor.

Volatile Objects: Objects whose lifetime does not exceed the lifetime of the processes that created them.

Wormmhole Routing: Every message is divided into a number of flits (flow control digits) which is the unit of transmission. The entire message need not be available to begin transmission. The node begins transmitting the message as soon as the header flit is received. The transmission of the flits occurs in a pipelined fashion.

APPENDIX B

Pseudocode for the HDMM Scheme

This appendix contains the pseudocode and auxiliary functions needed to implement the hierarchical distributed memory management scheme. The system hierarchy is divided into two levels for simplicity: local and global. Furthermore, we assume that the processing units in the system are connected by a two dimensional direct network.

B.1 General Object Functions

B.1.1 Variables and Data Structures

Promotion Threshold

```
This variable holds an integer value that indicates the number of
collections that an object has to survive in scavengespace before
it can be promoted to  Oldspace.  It is constant over an
execution of the garbage collection algorithm, but can vary
over GC's during the execution of a single program.
```

GC-is-on

```
A global flag in the system, owned by the  root actor.
When there is no system-wide GC going on, this flag is set to 0.
It is set to one by the root actor when a GC is started.
```

```
[A base actor]
   baseactor =  (baseState, acquaintance list,
```

```
    inverse acquaintance list,
          mailqueue, behavior)
    baseState = (objectStatus, dynamicity, logicalArea, copyCount)
[A meta actor]
    metaactor = (metaState, acquiantance list,
    inverse acquaintance list,
    mailqueue, behavior)
    metaStates = (root, gcPhases, position, numgc, numgcack,
    numinvgc, numinvgcack, ncMessage )
[A Node]
    node = (X-dimension, Y-dimension)
[A Channel]
    channel = (channelStates, Messages, Source, Destination)
[A Message]
    Message = (messageStates, messageData, source, destination)
    messageStates = (messageTag, mailAddress)
```

In order to extract components of these list data structures, we use the functions first(actorName), second(actorName), third(actorName) etc. that return the corresponding elements of the data structure.

B.1.2 Actor Manipulation Functions

```
;;;;;;;; Object Creation Function
```

(defun createActor (Persistant?, Touched?)

```
 Returns the actor name of the created actor.
(create an actor name)
(Create an entry in the distributed global nametable)
(set the persistant bit if it is a persistant actor)
(set the processor number of the PU on which the actor
is being created)
(create a virtual address)
```

```
(allocate space for the actor in physical memory)
(if (Touched?) set the actor-status field = 10
else   set the actor-status field = 00)
(make a local nametable entry with the virtual address
    and the physical offset of the actor in local memory)
(set the virtual address field in the global nametable)
(return the name of the actor))
;;;;;;;;; Actor Deletion Function
```

(defun deleteActor(actorName)

```
;; Deletes the actor-name corresponding to actor
(access the actor from the actor name)
(deallocate physical memory occupied by the actor)
(delete the  local nametable entry correponding to the actor)
(return the virtual address to a free VA pool)
(Delete the entry for the actor in the distributed global nametable)
(return the actor name to a free name pool))
;;;;;;;;; Actor Query Functions
```

(defun Touched? (*actorName*)

```
      ;; Returns true if the actor-status field in the actor is touched
      (if ((getObjectStatus (actorName)) eq touched)
  true
false))
```

(defun Untouched?(*actorName*)

```
      ;; Returns true if the actor-status field in the actor is untouched
      (if ((getObjectStatus (actorName)) eq untouched)
  true
false))
```

(defun Suspended?(*actorName*)

```
      ;; Returns true if an actor is suspended
      (if ((getObjectStatus (actorName)) eq suspended)
  true
```

```
false))
```

(**defun Scavengespace?(** *actorName*)

```
      ;; Returns true if an actor is in scavengespace
      (if (((getGenerationtag (actorName)) eq NewSpaceTag) or
   ((getGenerationtag (actorName)) eq PSSTag) or
   ((getGenerationtag (actorName)) eq FSSTag))
  true
false))
```

(**defun Newspace?(** *actorName*)

```
      ;; Returns true if an actor is newspace
      (if ((getGenerationtag (actorName)) eq NewSpaceTag)
  true
false))
```

(**defun Oldspace? (** *actorName*)

```
      ;; Returns true if an actor is oldspace
      (if ((getGenerationtag (actorName)) eq OldSpaceTag)
  true
false))
```

(**defun PSS?(** *actorName*)

```
      ;; Returns true if an actor is Past survivor space
      (if ((getGenerationtag (actorName)) eq PSSTag)
  true
false))
```

(**defun FSS?(** *actorName*)

```
      ;; Returns true if an actor is Future Survivor Space
      (if ((getGenerationtag (actorName)) eq FutureTag)
  true
false))
```

(**defun Persistant?(** *actorName*)

```
      ;; Returns true if an actor is persistant
      (if (getPersistent(actorName))
  true
false))

;;;;;;;;; Actor Settor Functions
```

(**defun setTouched(** *actorName*)

```
;; Sets the actor-status field of the actor corresponding to
;; actorName = touchedtag
(setObjectStatus (actorName, touchedTag))
```

(**defun setUntouched(** *actorName*)

```
;; Sets the actor-status field of the actor corresponding to
;; actorName = untouchedTag
(setObjectStatus (actorName, untouchedTag))
```

(**defun setSuspended(** *actorName*)

```
;; Sets the actor-status field of the actor corresponding to
;; actorName = suspendedTag
(setObjectStatus (actorName, suspendedTag))
```

(**defun Persistant(actorName)**

```
;;  The actor to be allocated is a persistant actor
;;  Allocate it with a P-tag Persistant and gen-tag of newspace
(setPersistent (actorName))
```

B.2 Base-Actor Specific Functions

(**defun getBaseState(actorName)**

```
;; returns the base state of the actor
;; Get the mail address corresponding to the actorName and
;; extract the first element from that list.
```

```
(first(getActorAddress(actorName))))
```

(defun getAcquaintanceList(actorName)

```
;; returns the acquaintance list of the actor
;; Get the mail address corresponding to the actorName and
;; extract the second element from that list.
(second(getActorAddress(actorName))))
```

(defun getInvAcquaintanceList(actorName)

```
;; returns the inverse acquaintance list of the actor
;; Get the mail address corresponding to the actorName and
;; extract the third element from that list.
(third(getActorAddress(actorName))))
```

(defun addAcquaintance (actorName, acq)

```
;; Adds the acquaintance acq to the acquaintance list
;; of the actor actorName.
;; Get the mail address corresponding to the actorName and
;; cons acq to the second element of the data structure.
(set(getAcquaintance(actorName)),
    cons(acq, (getAcquaintance(actorName)))))
```

(defun addInvAcquaintance (actorName, invacq)

```
;; Adds the inverse acquaintance acq to the inverse
;; acquaintance list of the actor actorName.
;; Adds the acquaintance acq to the acquaintance list
;; of the actor actorName.
;; Get the mail address corresponding to the actorName and
;; cons acq to the third element of the data structure.
(set(getInvAcquaintance(actorName)),
    cons(invacq, (getInvAcquaintance(actorName)))))
```

(defun getMailQueue(actorName)

```
;; Returns the mail queue corresponding to actorName.
;; Get the mail address corresponding to the actorName and
```

```
;; extract the fourth element from that list.
(fourth(getActorAddress(actorName))))
```

(defun addMailQueue(actorName, Message)

```
;;Adds  Message to the mailqueue  corresponding to  actorName.
;; Get the mail address corresponding to the actorName and
;; cons acq to the fourth of the data structure.
(set(getMailQueue(actorName)),
    cons(acq, (getMailQueue(actorName)))))
```

(defun getBehavior(actorName)

```
;; Returns the behavior corresponding to  actorName.
;; Get the mail address corresponding to the actorName and
;; extract the fifth element from that list.
(fifth(getActorAddress(actorName))))
```

B.3 Distributed Generation Scavenging

B.3.1 The DistributedScavenge Phase

(defun DistributedScavengePhase()

```
    (if (GC-IS-ON)
        (Wait for collection to complete and then reallocate)
      else (traverseRoot(root))))
```

(defun traverseRoot(rootActor)

```
  (set GC-IS-ON to true)
  (set  object-status of rootActor to touched)
  (for every actor A in the root set
       (set object-status of A to touched)
       (for every forward acquaintance facq of actor A
    (scavengeForwardAcq ( facq, A)))
       (for every inverse acquaintance  iacq of actor A
    (scavengeInverseAcq ( iacq, A)))
       (when (receivedAllAcks (A) (backPropogation(rootactor)))))
  (when (receivedAllAcks (rootActor))
```

```
(signal termination(distributedScavengePhase)))))
```

```
(defun scavengeForwardAcq(actor, sender)
  (if ((Actor in OldSpace) or (Actor is persistent)) (ignore and exit)
    (if (getGenerationTag(actor) = futureTag) (backPropogation(sender))
      ((set object-status of actor touched)
       (if (copyCount(actor) ≥ Threshold)
   (setgenerationTag (actor, OldSpaceTag))
 (setgenerationTag (actor, futureTag)))
       (for every forward acquaintance facq of actor
    (scavengeForwardAcq ( facq, actor)))
       (for every inverse acquaintance iacq of actor
    (scavengeInverseAcq ( iacq, actor)))
       (when (receivedAllAcks (actor) (backPropogation(sender))))))))
```

```
(defun scavengeInverseAcq(actor, sender)
  (if ((Actor in OldSpace) or (Actor is persistent)) (ignore and exit)
    (if (getGenerationTag(actor) = futureTag) (backPropogation(sender))
      (if (actor is ACTIVE)
  ((set object-status of actor touched)
   (if (copyCount(actor) ≥ Threshold)
       (setgenerationTag (actor, OldSpaceTag))
     (setgenerationTag (actor, futureTag)))
   (for every forward acquaintance facq of actor
(scavengeForwardAcq ( facq, actor)))
   (for every inverse acquaintance iacq of actor
(scavengeInverseAcq ( iacq, actor)))
   (when (receivedAllAcks (actor) (backPropogation(sender)))))
;;; actor is BLOCKED
((set object-status to suspended)
 (for every inverse acquaintance iacq of actor
      (scavengeInverseAcq ( iacq, actor)))))))
```

B.3.2 The LocalClear Phase

```
(defun LocalClearPhase()
     (for every processing unit (PU) in the system
```

```
        (clearLocalMemory(PU))
        (clearDistributedNametable(PU))
```

(defun clearLocalMemory(PU)

```
;; clearing the garbage actors in global memory
  (for every actor A in the processing unit PU
        (if (((actor is not in OldSpace) or (actor is not persistent))
    and (actor is touched))
    ((move actor to a compacted region of local memory)
     (set object-status of actor to untouched)
     (increment copyCount of the actor by 1))
 else (clear memory area occupied by object))))
```

(defun clearDistributedNametable(PU)

```
  ;; clearing the garbage entries in the distributed global nametable
  (for every entry in the distributed nametable on the local PU
        (if ((getGenerationTag(entry)) ≠ futureTag)
   (clear entry corresponding to the object))))
```

B.4 Network Clearance Algorithms

B.4.1 Node Functions

- [Start Node:]

```
;; The start node of the system
startNode = (0,0)
```

- [Finish Node:]

```
;; The finish node of the system
finishNode = (n,n)
```

(defun X-dim(node)

```
    ;; returns the X-dimension of the node
    (first(node)))
```

(defun Y-dim(node)

```
    ;; returns the Y-dimension of the node
```

```
(second(node)))
```

(defun whichNode(objectName)

```
;; returns the node on which the object currently
;; resides.
())
```

(defun Get-Start-Node()

```
;; returns the  start node of the system

(startNode))
```

(defun Get-Finish-Node()

```
;; returns the  finish node of the system

(finishNode))
```

(defun startNode? (node)

```
;; returns true if node is the start node of the
;; configuration, false otherwise
  (if ((X-dim(node) eq 0) and (Y-dim(node) eq 0))
      true
    false))
```

(defun finishNode? (node)

```
;; returns true if node is the finish node of the
;; configuration, false otherwise
  (if ((X-dim(node) eq n) and (Y-dim(node) eq n))
      true
    false))
```

(defun cornerNode? (node)

```
;; returns true if node is a corner node in the
;; configuration, false otherwise.
;; A corner node is defined as a node whose
```

```
;; X dimension is 0.
  (if ((X-dim(node) eq 0))
      true
    false))
```

(defun Get-F-neighbors(node)

```
;; returns the forward neighbors of the node as a pair

((X-dim(node)+1, Y-dim(node)),
 (X-dim(node), Y-dim(node)+1)))
```

(defun Get-B-neighbors(node)

```
;; returns the backward neighbors of the node as a pair

((X-dim(node)-1, Y-dim(node)),
 (X-dim(node), Y-dim(node)-1)))
```

(defun F-path?(node1, node2)

```
;;  indicates if there exists an F-path from node 1 to node 2

;;  if node2 is a F-neighbor of node1, return true
;;  otherwise, call f-path? recursively with the F-neighbors
;;  of node1.
(if (finishNode?(node1))
false
((if (member (node2 (Get-F-neighbors(node1))))
     true
   ((F-path? ((first(Get-F-neighbors(node1))), node2))
    (F-path? ((second(Get-F-neighbors(node1))), node2)))))))
```

(defun Bpath?(node1, node2)

```
;;  indicates if there exists an Bpath from node 1 to node 2
;;  if node2 is a Bneighbor of node1, return true
;;  otherwise, call bpath? recursively with the Bneighbors
;;  of node1.
```

```
(if (startNode?(node1))
       false
       ((if (member (node2 (Get-Bneighbors(node1))))
    true
  ((Bpath? ((first(Get-Bneighbors(node1))), node2))
   (Bpath? ((second(Get-Bneighbors(node1))), node2)))))))
```

B.4.2 Message Functions

(defun tag(*message*)

```
;;  returns the tag of the message
())
```

(defun Old?(*message*)

```
;;  returns true if the message is an old message
      (if (tag (message)) false true))
```

(defun New?(*message*)

```
;;  returns true if the message is a new message

      (if (tag (message)) true false))
```

B.4.3 The Broadcast Algorithm

```
The broadcast algorithm initiates the
;; broadcast of a message at the start node
;;
```

(defun Broadcast(message)

```
  (BroadcastNodes(message,startNode)))
```

(defun BroadcastNodes(message,node)

```
  (cond
   ((finishNode? (node)) exit)
   ((cornerNode? (node))
    ((BroadcastNodes(message,right(node)))
```

```
        (BroadcastNodes(message,down(node)))))
      ((BroadcastNodes(message,down(node))))))
```

B.4.4 The Forward Bulldoze Algorithm

```
The Fbulldoze algorithm initiates the
    ;; Fbulldoze wave  at the start node
    ;; of the configuration.
    ( defun Fbulldoze()
      (FbulldozeNodes(startNode)))

    ( defun FbulldozeNodes(node)
      (cond
       ((finishNode? (node))
        (if (first-FBulldoze-received?)
    (exit)
          ( wait for second FBulldoze)))
       ((process messages in the mailqueue)
        (FbulldozeNodes(message,first(Fneighbors(node))))
        (FbulldozeNodes(message,second(Fneighbors(node)))))))
```

B.4.5 The Backward Bulldoze Algorithm

```
The Bbulldoze algorithm initiates the
    ;; Bbulldoze wave  at the start node
    ;; of the configuration.
    ( defun Bbulldoze()
      (BbulldozeNodes(startNode)))

    ( defun  BbulldozeNodes(node)
      (cond
       ((startNode? (node))
        (if (first-Bbulldoze-received?)
    (exit)
          (wait for second Bbulldoze)))
       ((process messages in the mailqueue)
```

```
    (BbulldozeNodes(message,first(Bneighbors(node))))
    (BbulldozeNodes(message,second(Bneighbors(node)))))))
The bulldoze wave consists of an Fbulldoze
;; followed by a Bbulldoze.
( defun BulldozeWave()
      (Fbulldoze())
      (Bbulldoze()))
```

B.5 Hierarchical Distributed Garbage Collection

```
(defun HDGC()
     (if (GC is not active in the system)
         ((PreGC-operations())
          (Broadcast(GC-IS-ON))
          (repeat the following step twice
          (BulldozeWave()))
          (DistributedScavengePhase())
          (Broadcast(DGS-completed))
          (LocalClearPhase())
          (Broadcast(GC-completed))
          (PostGC-operations()))))
```

B.5.1 Cluster Management Functions

```
defun getClusterID(actorName)
   ;; Return the clusteID of the lowermost cluster that
   ;;em actorName belongs to.

    defun getClusterManager(actorName)
   ;; Returns the cluster manager of the actor corresponding
   ;; to actorName

    defun electClusterManager(clusterID)
   ;; Returns the elected cluster manager of the cluster
   ;; with the ID clusterID
```