Global incremental garbage collection in distributed systems

Soha Safwat Labib
The American University in Cairo AUC

GLOBAL INCREMENTAL GARBAGE COLLECTION IN DISTRIBUTED SYSTEMS

SOHA SAFWAT LABIB

2004

2004/83

١٥

The American University in Cairo
School of Sciences and Engineering

GLOBAL INCREMENTAL GARBAGE COLEECTION IN DISTRIBUTED SYSTEMS

A Master Thesis Submitted by

Soha Safwat Labib

B.Sc in Computer Science

Under the supervision of

Prof. Dr. Amr El Kadi

To

Computer Science Department

In partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

December 2004

2004/83

The American University in Cairo
Computer Science Department
Master Program
December 2004

GLOBAL INCREMENTAL GARBAGE COLLECTION IN DISTRIBUTED SYSTEMS

M.Sc. Thesis
Submitted by
Soha Safwat Labib

Supervised by
Prof. Dr. Amr El Kadi

In partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

has been approved by:

Prof. Dr. Amr El Kadi
Thesis Committee Chair/Adviser --- [redacted]
Affiliation ------------AUC------------

Prof. Dr. Aly Fahmy
Thesis Committee Reader/examiner --- [redacted]
Affiliation ----------------------------

Prof. Dr. Amr Gonied
Thesis Committee Reader/examiner --- [redacted]
Affiliation ------------AUC------------

Dr. Sherif El Kassass
Thesis Committee Reader/examiner [redacted]
Affiliation ------AUC------

[redacted] Jan 10, 04 [redacted] January 11, 2005

Department Chair Date Dean Date

Abstract

Some programming languages present new challenges to garbage-collection design. These languages are based on dynamic object creation and automatic reclamation of storage, a garbage collector can also restore locality to fragmented data by dynamically compacting it, thereby improving the performance of caches and virtual memory systems and reducing memory requirements. The spectrum of garbage-collection schema for linked data structures distributed over a network is reviewed here. Distributed garbage collectors are classified first because they evolved from single-address-space collectors. This taxonomy is used as a framework to explore distribution issues such as: locality of action, communication overhead and deterministic communication latency.

There are many proposed distributed architectures for efficient execution of programs with potential parallelism. These architectures include reduction machines and parallel logic machines. Execution of programs on such architectures usually utilizes a distributed global heap for creation and manipulation of objects, which in turn reside in the local stores of the Processing Elements (PEs).

Traditional garbage collection techniques lead to long and unpredictable delays and are therefore not satisfactory in a number of settings, such as interactive and real-time systems, where non-disruptive behavior is of paramount importance.

This thesis concentrates on garbage collection algorithm suitable for such systems. An idea of a new global incremental garbage collector for distributed environment and its essential parts is surveyed and evaluated according to three different distributed garbage collection algorithms. A proposed algorithm to solve problems found in the global incremental garbage collection algorithm. A total survey is done to compare our algorithm to other algorithms.

TABLE OF CONENTS

Table of Figures

List of Tables

Chapter 1

Introduction

Chapter 1
Introduction

Actually, the interest of people always changes or shifts from a thing to a better one. That is really, what is happening in the computer world, as time goes on the interests of people differ and asks for more technology. First, we were using the basic Von Neumann computer model, and step-by-step we turned to distributed & parallel computer model which to reach, we faced many problems.

One major problem to overcome in the management of resources is the garbage collection of objects in physically separated stores. Since this problem is a part of the function of an object management system (which provides operations to create, access and update the objects, as well as to garbage collect their spaces), a good investigation of this problem should include the entire function of that system. We assume one execution model to demonstrate the work, but many of the results to be presented are applicable to any execution model using a global heap implemented on the local stores of a number of physically separated, but intercommunicating processing elements.

1.1 What is Garbage Collection?

Garbage collection is the process of **determining** which objects are no longer referenced by the program and **make available** the heap space occupied by these objects to the program. Basically, the main memory layout is subdivided into regions as shown in fig 1.1:

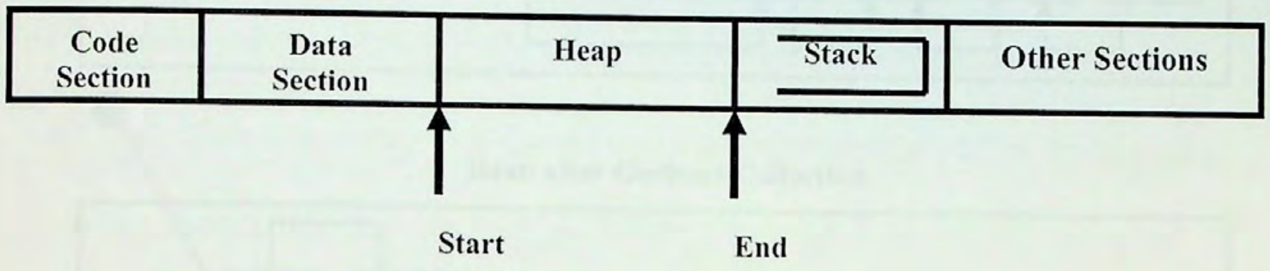

Figure 1.1: Main Memory Layout

The above figure shows the code section where the code needed is found in the main memory, while the data section contains all the declared variables and constants found in any program. The heap part which we are interested in and the stack section which is used by the compiler to parse our code.
We are interested in the heap part only, where inaccessible objects are found or in other words the garbage collection only deals with heap, while the rest of the memory parts are still very important. As we said that the objects are found in the heap but some of them are inaccessible that is we need the

garbage collector to collect these objects. The heap will relocate the accessible objects and remove all inaccessible objects.

Figure 1.2 illustrates the heap layout before and after garbage collection.

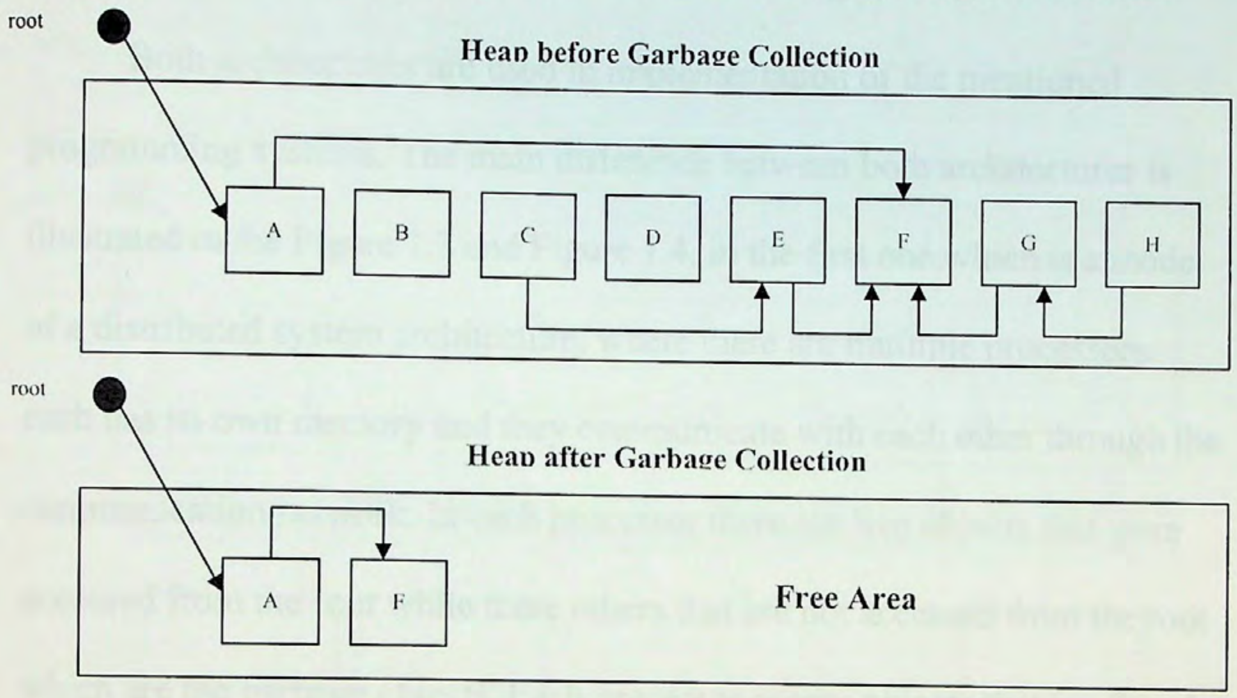

Figure 1.2: Heap Layout

As we had seen in the above example, we always had a root through which the objects in the heap are accessed, these are the accessible objects. All other objects found in the heap are garbage that must be removed and that what the garbage collector had done.

1.2 Why do people study Garbage Collection?

Since there are always high-level programming systems that require garbage collection, for example: Object-Oriented Programming (Java), Functional Programming (LISP, ML…) & Logic Programming (Prolog).

These systems rely upon a part of the system (subsystem) that identifies inaccessible objects and recycle (reuse) their storage space.

1.3 Why Distributed (& Parallel) Garbage Collection?

Both architectures are used in implementation of the mentioned programming systems. The main difference between both architectures is illustrated in the Figure 1.3 and Figure 1.4, in the first one which is a model of a distributed system architecture, where there are multiple processors, each has its own memory and they communicate with each other through the communication network. In each processor there are live objects that were accessed from the root while there others that are not accessed from the root which are the garbage objects. Each processor access objects that are found in other memories of other processors

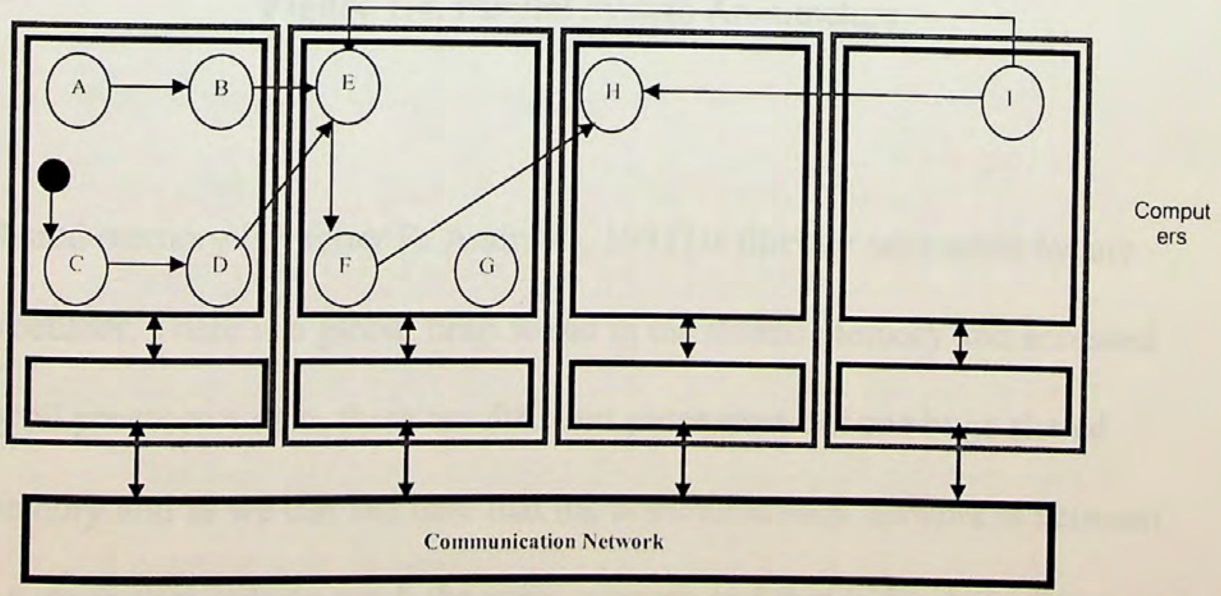

Figure 1.3: Distributed System Architecture

Each processor has its own local memory and accesses only its own memory. It refers to remote objects in other main memory and accesses these objects through message passing along the communication network. There is actually a global heap distributed among all main memories found in the system. This all refers to Figure 1.3.

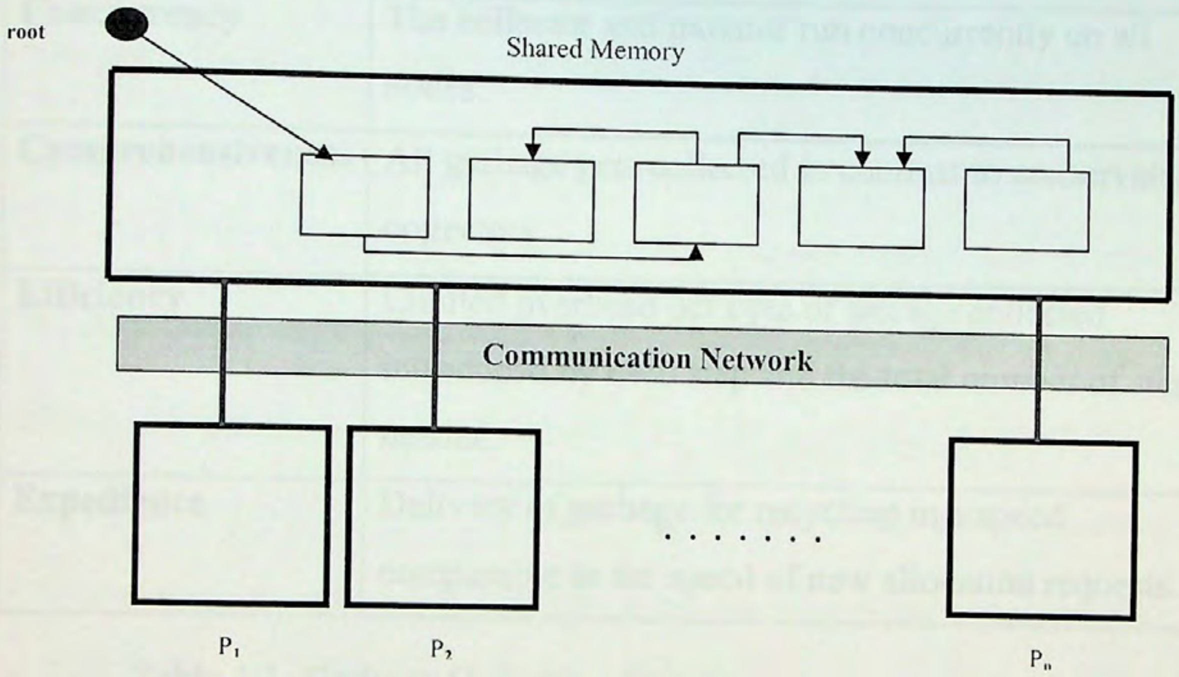

Figure 1.4: Parallel System Architecture

Shared memory [Gregory R. Andrews, 1991] is directly accessible by any processor. There is a global heap found in the shared memory and accessed by all processors also, there are different processors but one huge shared memory and as we can see here that the communication network is between the processors only to reach the main memory and that is the main difference between both the Parallel System and Distributed System, this is shown in figure 1.4.

Garbage collectors for distributed systems are characterized by supporting one or more of the features shown in table 1.1:

Concurrency	The collector and mutator run concurrently on all nodes.
Comprehensiveness	All garbage gets collected in contrast to conservative collectors.
Efficiency	Limited overhead per byte of storage collected introduced by each step and the total number of steps needed.
Expedience	Delivery of garbage for recycling in a speed comparable to the speed of new allocation requests.

Table 1.1: Garbage Collection Criteria

According to the table above we can see why we truly need garbage collection in distributed systems to apply the above criteria.

1.4 Garbage Collection Evaluation Criteria

Usually, any sequential algorithm has two main criteria for evaluation:

- Processing Time: The faster the algorithm in processing data, the better it is to be used.

- Space Overhead: The smaller the space to be used by the algorithm during processing, the better it is.

For Parallel algorithms, there is an additional criterion to the above, which is synchronization overhead, (how to coordinate accessing of shared objects). In other words, how to keep track of which processor is processing now and what objects is this processor accessing. The less synchronized the system is, the better is the algorithm.

For Distributed algorithms, there is also another additional criterion to the above, which is the Communication overhead [L Augusteijn, 1987], [Özalp Babaoğlu , Keith Marzullo, 1993] . That is how much message passing takes place between different computers to perform this algorithm. The minimum the message passing between the computers and each other, the better the algorithm.

Specifically, for Garbage Collection algorithms, in addition to all criteria above, there are two main criteria to evaluate its performance:

- How fast the algorithm detects Garbage Objects (i.e. once there is a garbage object found in the heap, it is must be detected and it can be reused at once.) The faster to detect the garbage, the better is the algorithm.

- Detecting all the garbage found in the heap (i.e. in some cases there are cyclic garbage found in the heap that can't be detected by some algorithms of garbage collection.) The more to detect cyclic graphs and collects its garbage the better is the algorithm.

1.5 Structure of the thesis

In Chapter 2 we review uni-processor garbage collector. Chapter 3 reviews and discusses distributed garbage collectors. In Chapter 4, we present and discuss our idea of a new incremental global garbage collector for distributed environments. Chapter 5 evaluates our algorithm to other distributed algorithms. Chapter 6 concludes our work and gives our plans for future work.

Chapter 2
Uni-Processor Garbage Collection Algorithms

Chapter 2
Uni-processor Garbage Collection Algorithms

2.1 Marking-Sweep GC Algorithm

A marking garbage collector [ALMES, G., BORNING, A., AND MESSINGER, E. 1983] [BADEN, S.B. 1983] [BEKKERS, Y. AND COHEN, J. 1992] usually compromises three phases. The first one is called the marking phase. Its task is to identify the set of objects accessible from a given root by traversing the graph of accessible objects, and marking each object encountered. The second is called the sweeping phase and its task is to sweep the entire storage and to examine each existing object: if the object is not marked, it is reclaimed by returning its space to free storage. The last phase is called the compaction phase, its task is to compact all used storage space towards one end of the storage area. The other end contains a contiguous area of unused space that can be used for creating new objects. Schemes belonging to this category have the following characteristics:

- All garbage is reclaimed.
- Normal program execution is suspended while garbage collection is being performed.

2.2 Copying GC Algorithm

The copying garbage collector [ARNBORG, S. 1974] [BARACH, D. R., TAENZER, D. H., AND WELLS, R. E. 1982] performs all the tasks of marking sweep in one phase, thus it is included in this category. It is based on dividing the available storage space into two areas called semispaces. The role of semispaces is reversed with each garbage collection cycle. At a given time, only one semispace is used for creating new objects. When collection begins, all accessible objects are copied from the current semispace into the other semispace and the role of the semispaces is reversed. When garbage collection has completed, the old semispace contains only garbage, while the new semispace contains all and only all accessible objects residing in one side of that semispace. Now computation can be resumed, and spaces for new objects can be allocated from unused part of the new semi space until it again necessary to garbage collect. This algorithm allows good locality of references since the moved objects are compacted, page faults are likely to be minimized when such an algorithm is used in a virtual memory environment.

2.3 Reference Count GC Algorithm

The reference counting approach [Mordechai Ben-Ari, 1984] [D. R. Brownbridge, 1985] is based on associating a counter with each object to indicate the number of references to this object. When an object is created, a single reference to the object is created, and its associated counter is set to one. Whenever a reference to the object is copied, its counter is incremented by one. Whenever a reference to the object is destroyed, its counter is decremented. An object is identified as inaccessible and can be reclaimed when its associated counter reaches zero. Schemes belonging to this category have the following characteristics:

- They cannot reclaim cyclic structures. Such a structure occurs when an object refers to itself directly or indirectly via other objects. When a cyclic structure becomes inaccessible to the user's program, it will still have nonzero reference counters and therefore cannot be identified as garbage.
- Slower program execution due to the overhead in each object reference operation. On the other hand, normal program execution is never long suspended like in the marking approach.
- Faster detection of garbage, once a counter associated to an object becomes zero, it is identified as a garbage object and its space can be reused.

2.4 Real-Time or Incremental GC Algorithm

Two approaches have been proposed to perform real-time garbage collection [CARLSSON, S., MATTSSON, C., AND BENGTSSON, M. 1990] [C. Chambers, D. Ungar, E. Lee, 1989]. The goal of these approaches is to avoid substantial program interruption due to garbage collection.

The first is based on two necessary processes working in parallel: one is responsible for garbage collection, called collector, and the other for program execution, called mutator. The second approach is proposed by Baker [AMSALEG, L., GRUBER, O., AND FRANKLIN, M.1995]. It is an interesting extension of the copying garbage collector mentioned before. The basic idea is that in each storage operation a small part of the garbage collection work is performed. This implies that the two semispaces are simultaneously active. This approach avoids any complexity, since normal activity and garbage collection represent only one sequential process.

This algorithm [H. D. Backer, 1972] [Henry G. Baker, Jr., 1978] [Henry G. Baker, 1992] works the same as on uni-processor except for garbage collector must keep track of all remotely accessed objects. The heart of the generational real-time garbage collector presented in any research is Baker's incremental garbage collector, in which the collection work is distributed among the mutator's primitives to avoid long pauses.

In Baker's collector, the space devoted to the heap is subdivided into two semi spaces, which change their roles on every GC cycle. During normal program execution, the two semi spaces are in use. Each garbage collection cycle begins with atomic flip, which conceptually invalidates all objects in one semi space, called From space, and copies to the other semi space, called To Space, all objects directly reachable from the root set. Then the program execution (mutator) is allowed to resume. Any object in From space that is accessed by the mutator must first be copied to To space. The background scavenging process is also interleaved with normal execution, to ensure that all reachable data are copied to To space and the collection cycle finishes before memory is exhausted. Objects allocated by the mutator are allocated in To space and they are treated as if they had already been scanned, i.e., they are assumed to be live. Baker allocates new objects at one end of To space and copied objects in the other end of To space. The latter represents the queue of objects to be scanned that begins at S and ends at B. Each time a new object is allocated, an increment of scanning and copying is done.

The following example illustrates more about the algorithm:

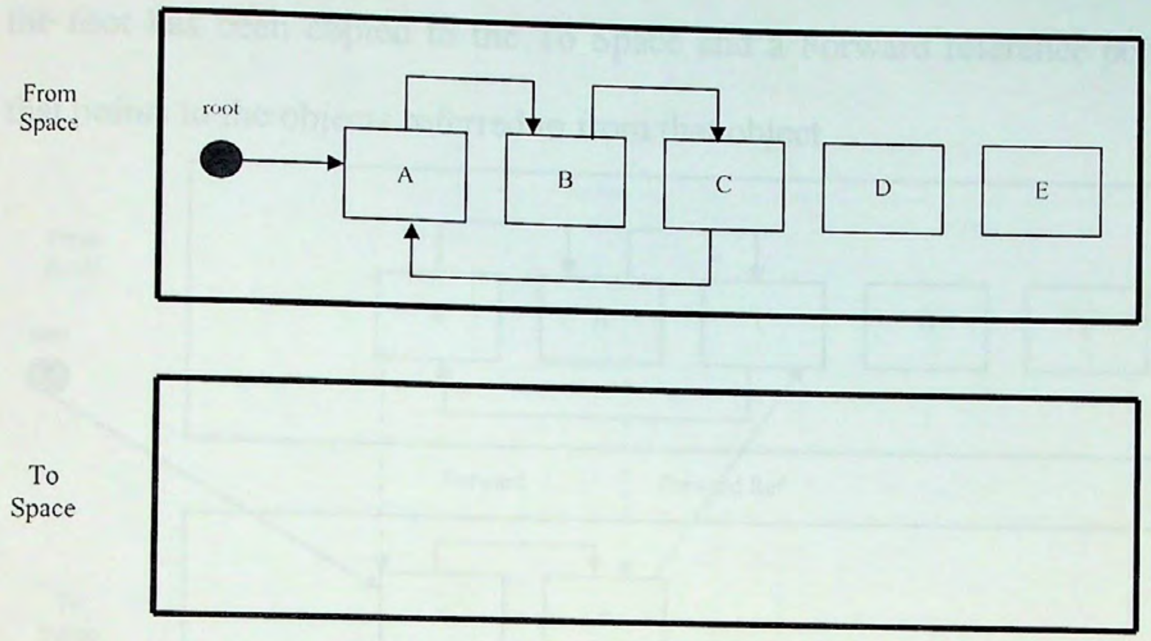

Figure 2.1: Copying Garbage Collector for one Computer

In the above figure, we can see the two semi spaces, the From space has some objects that are accessed from the root while the rest are garbage that must be collected.

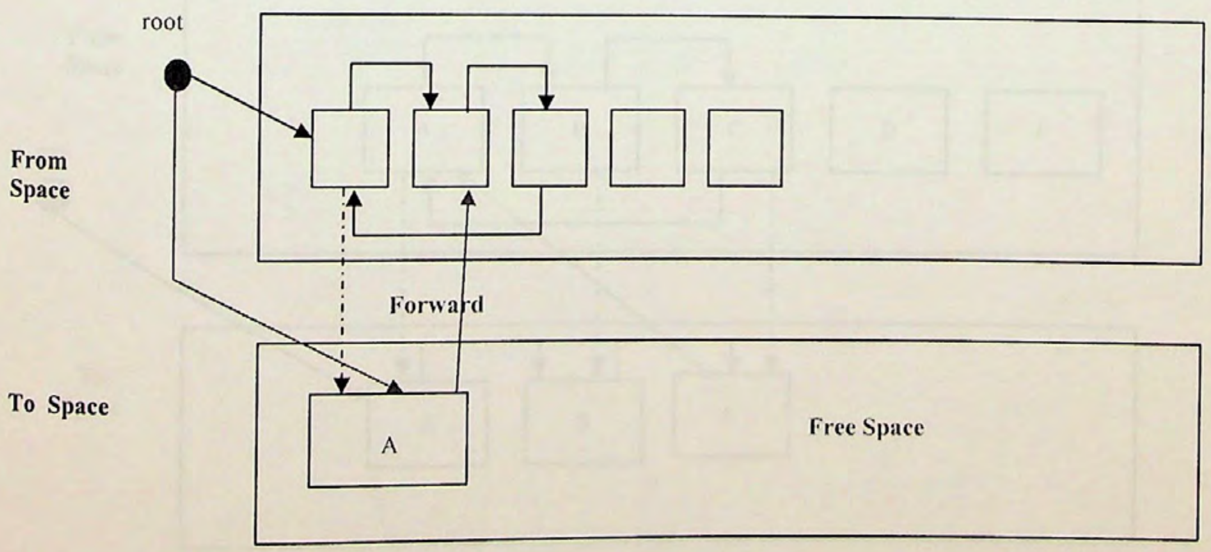

Figure 2.2: After copying the first object

As you can see in the above diagram, which the first object accessed from the root has been copied to the To Space and a Forward reference pointer that points to the objects referred to from that object.

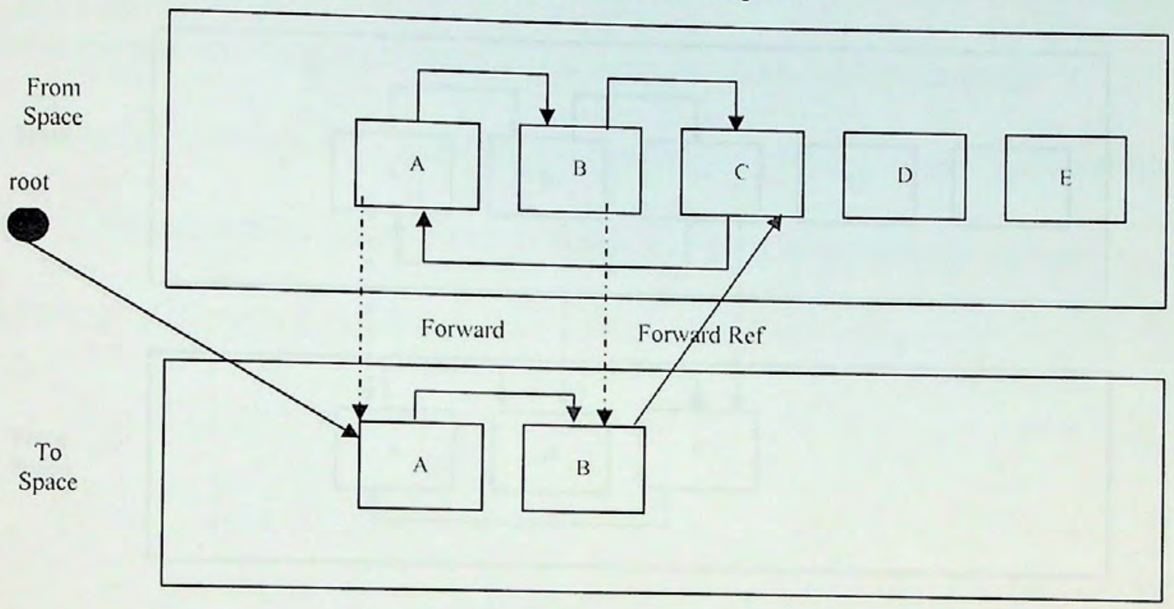

Figure 2.3: After copying the second object

According to the above diagram, we copied the second object and we now have another forward pointer that points to objects in the From space.

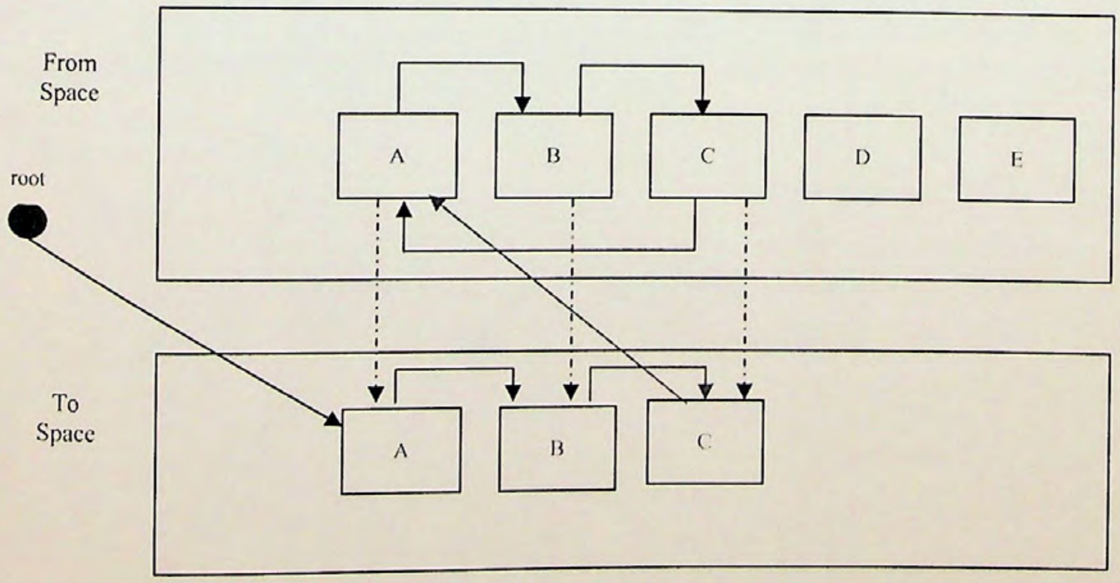

Figure 2.4: After copying the third object

The above diagram shows the copying of the third object which is the last accessible object in the From space. Therefore there is a pointer to show that this is the last one.

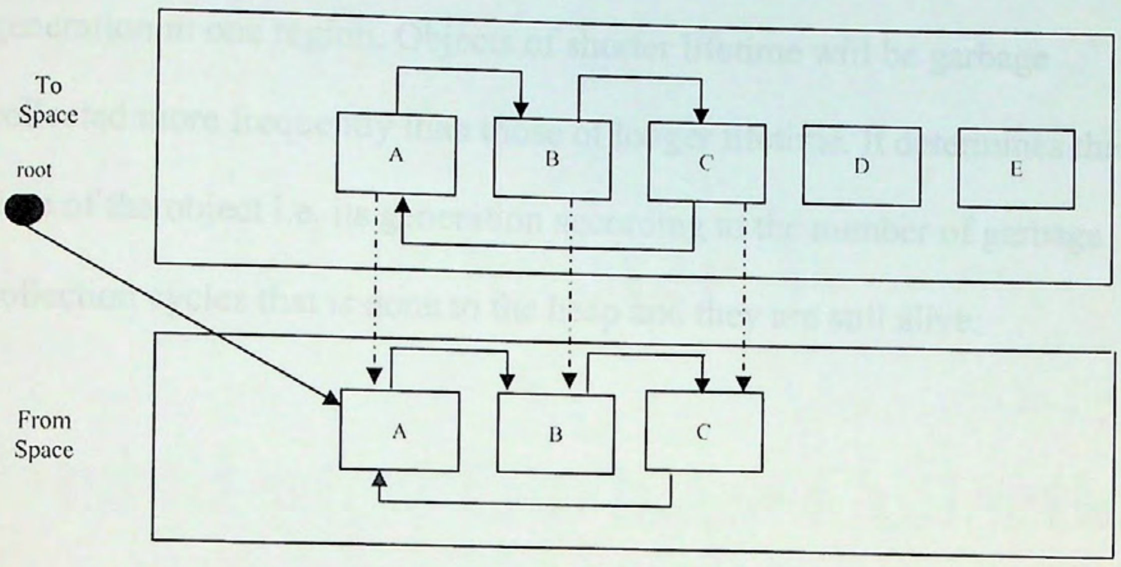

Figure 2.5: The last view of the heap after garbage collection

In this diagram, we had seen that the flipping of the semi spaces has been done. The From space now has to have another cycle of Garbage Collection.

2.5 Generational GC Algorithm

Generational algorithm [BARTLETT, g.F. 1990] classifies objects according to their lifetime (generation) and allocates objects in the same generation in one region. Objects of shorter lifetime will be garbage collected more frequently than those of longer lifetime. It determines the life time of the object i.e. its generation according to the number of garbage collection cycles that is done to the heap and they are still alive.

Chapter 3

Distributed Garbage Collection Algorithm

Chapter 3
Distributed Garbage Collection Algorithm

3.1 Introduction

Any garbage collection scheme for a uni-processor system can be classified as a variant of either the marking approach [ABDULLAHI, S. E. AND EDEMENANG, E. J. A. 1993] or the reference counting approach [Nor79] [Daniel G. Bobrow, 1980] [D. R. Brownbridge, 1985] [CHIKAYAMA, T. AND KIMURA, Y. 1987] . The same is also true for all distributed garbage collection schemes. An adaptation of many ideas that have been developed for uni-processor systems to the distributed environment. Many new problems have to be solved to achieve these problems are due to the unique characteristics of highly parallel distributed systems.

Garbage collection schemes for uni-processor systems have been discussed in Chapter 2. The present chapter discusses different garbage collection schemes that are applicable to loosely coupled multiprocessor systems. It discusses the distributed garbage collection schemes that have been proposed for such distributed environments. It also gives an overview of the main ideas and characteristics of my distributed garbage collection scheme.

3.2 Global Mark-Sweep GC Algorithm

This scheme [ABDULLAHI, S. E. 1992] is similar to the uni-processor marking-sweep, but is on a distributed system (i.e. needs message passing between processors to know which objects are accessed from the root). The following example is illustrating how the Global Mark-Sweep GC algorithm works. We have heap in two different computers, objects are created in both heaps and each computer is accessing objects from the other. Figure 3.1 will show the scenario of how the example works:

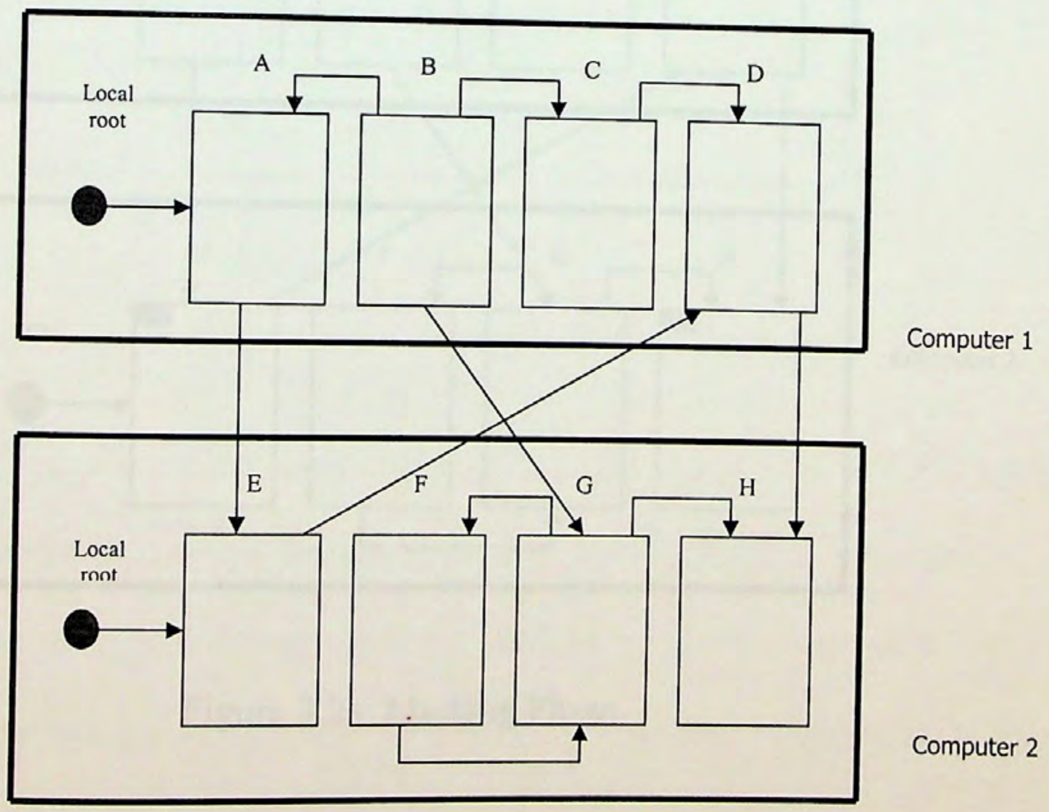

Figure 3.1: The heap memory before Garbage Collection

The above figure shows the global heap on two different computers and how they look before garbage collection. The heap is now full and the processors need garbage collection to take place, the whole system will be suspended until garbage collection takes place.

So, the marking phase will take place, in this phase we mark all accessible objects that are accessed from the root. Figure 3.2a illustrates this phase.

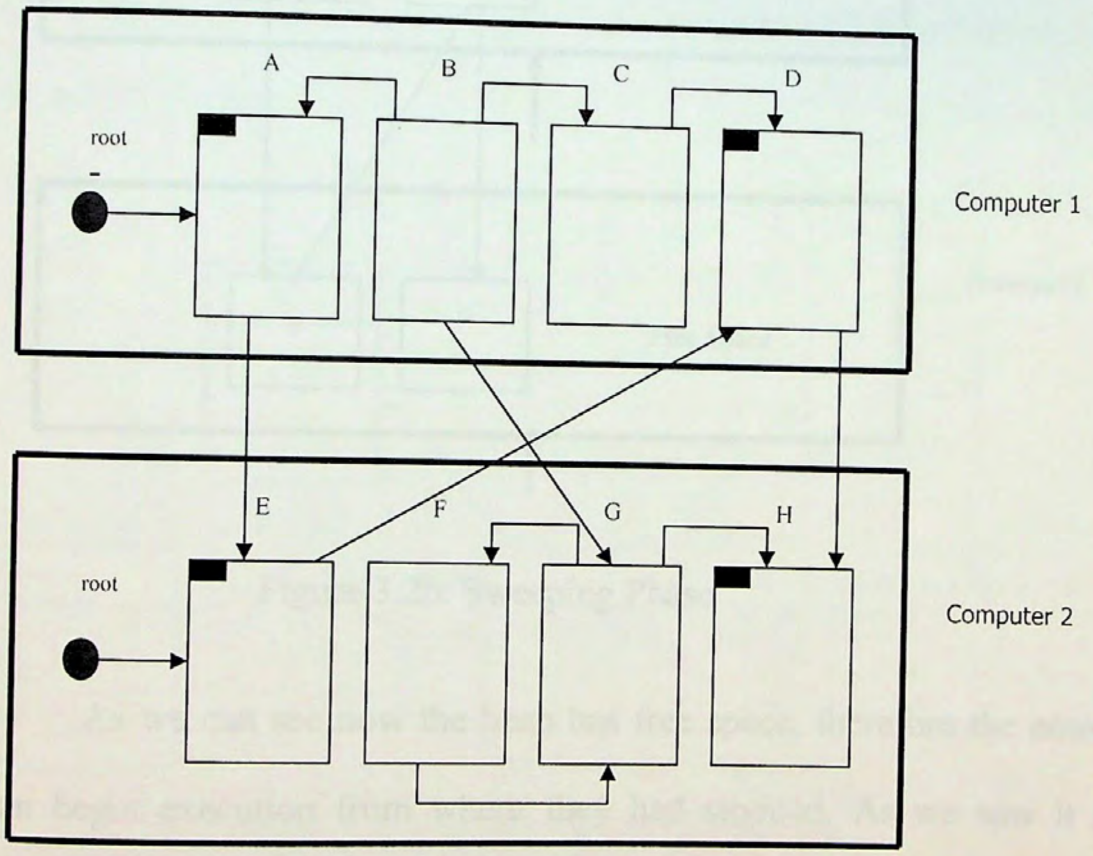

Figure 3.2a: Marking Phase

After marking all the accessible objects from the roots are marked, now we have to sweep the local heap memory of each computer and compact used spaces by marked objects towards one end of the local storage. Figure 3.2b illustrates the sweeping phase.

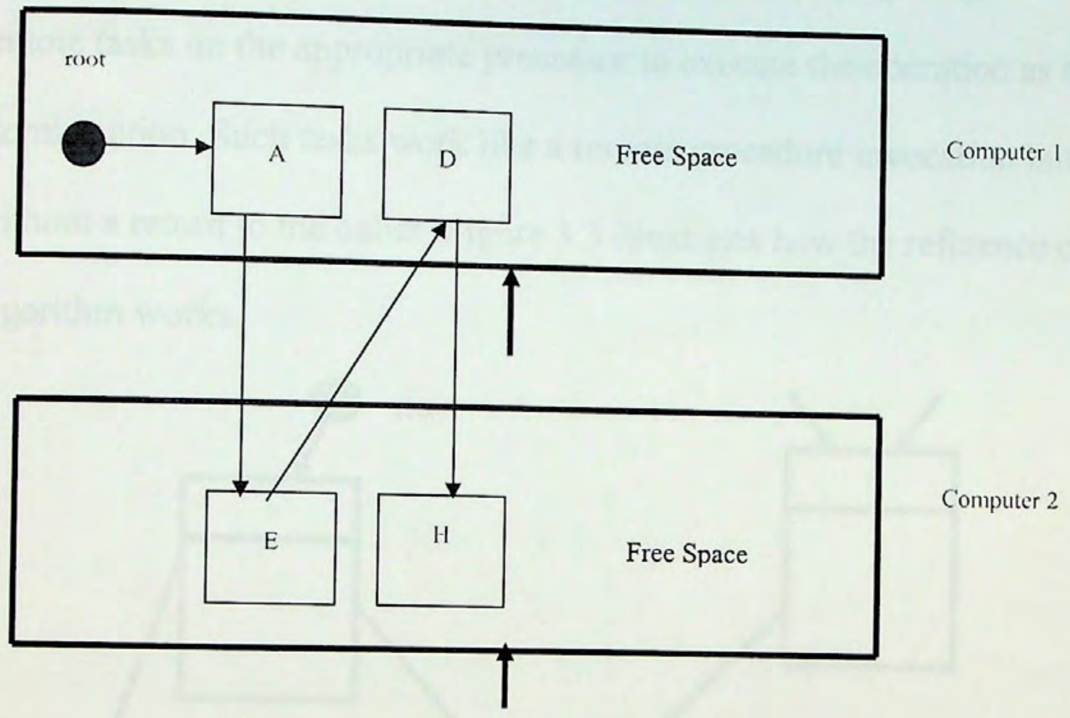

Figure 3.2b: Sweeping Phase

As we can see now the heap has free space, therefore the computers can begin execution from where they had stopped. As we saw it seems simple to do, but actually it is not, and the main overheads in this algorithm are: the number of message passing to mark, sweep, compact and unmark the accessible objects, the second is stopping the whole system to do garbage collection.

3.3 Reference Count GC Algorithm

Nori [Nor79] and Hudak [Hud82] adapted the reference counting approach to the distributed environment. In their schemes, all of the references counting operations are performed by spawning (or generating) remote tasks on the appropriate processor to execute the operation as an atomic action. Such tasks work like a remote procedure invocation but without a return to the caller. Figure 3.3 illustrates how the reference count algorithm works.

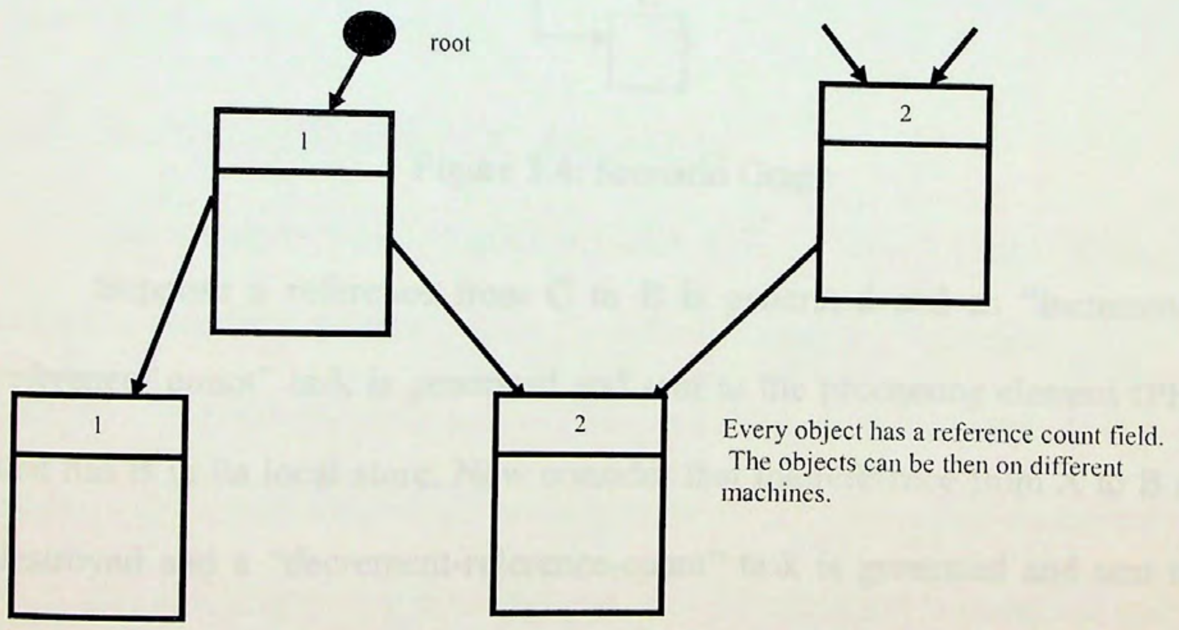
root
1
2
1
2
Every object has a reference count field. The objects can be then on different machines.

Figure 3.3: Reference Count Algorithm

The non-trivial problem in the adaptation of this approach to the distributed environment is to guarantee that reference counting [ABDULLAHI, S. E. 1992] [Nor79] [Daniel G. Bobrow, 1980] [D. R. Brownbridge, 1985]

[CHIKAYAMA, T. AND KIMURA, Y. 1987] operations (increment and decrement of the reference-count) are executed in the same order that they were generated, otherwise a reference count may reach zero prematurely. To demonstrate how the order of executing reference counting tasks may erroneously identify some accessible objects as garbage, consider the following scenario for the graph shown below.

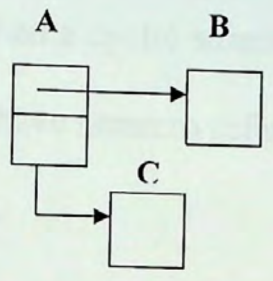

Figure 3.4: Scenario Graph

Suppose a reference from C to B is generated and an "increment-reference- count" task is generated and sent to the processing element (PE) that has B in its local store. Now consider that the reference from A to B is destroyed and a "decrement-reference-count" task is generated and sent to the PE that has B in its local store. The former task was generated before the latter. Suppose that it cannot be guaranteed that the former task is processed first. In this case, if the latter task is processed first, object B's reference count will drop to zero, erroneously identifying it as garbage. Another problem with reference count approach is the availability of space for messages or task for reference count operations. The space requirement of the reference counting tasks is ignored by both Nori [Nor79] and Hudak [Hud82]. The determination of such space is very important to guarantee that this space is always available for reference counting tasks, otherwise a deadlock can occur. A third problem with reference count approach is cyclic structures. Such a structure occurs when an object refers to itself directly or indirectly via other objects. When a cyclic structure becomes inaccessible to the user's program, it will still have nonzero reference counters and therefore cannot be identified as garbage.

3.4 Reference Weight GC Algorithm

The reference weight [ABDULLAHI, S. E. 1992] is a variant of the reference counting approach. With the reference counting scheme an 'increment-reference-count' task is generated and sent to the appropriate processing element for execution when a new object reference is generated. Similarly, a 'decrement-reference-count' task (or message) is generated and sent to the appropriate processing element when an object reference is destroyed. The idea of reference weight is as follows: when an object A creates a new object B, a weight W is associated with B and this value becomes a part of the created object reference to B. There is always weight that is divided on all the objects referencing that object.

Figure 3.5 shows the above algorithm how it works:

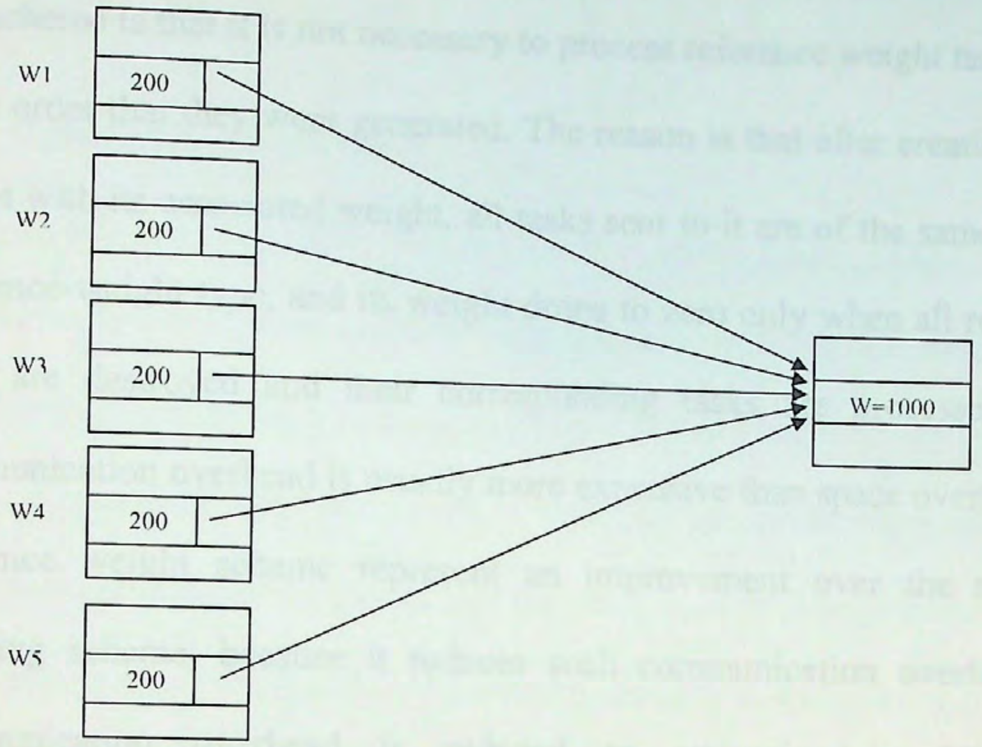

Figure 3.5: Reference Weight Algorithm

Making a copy of an object reference does not require communicating with the processing element that has the object in its local store. On the other hand, destroying an object reference requires generating a "reduce-reference-weight" task and sending it to the processing element that the object in its local store to decrease the current object's weight by the weight associated with that object reference. An object will be identified as garbage only when its weight reaches zero (exactly as in the fundamental reference counting approach).

Comparing with the reference counting scheme, the first advantage of this scheme is that it is not necessary to process reference weight tasks in the same order that they were generated. The reason is that after creating a new object with its associated weight, all tasks sent to it are of the same reduce-reference-weight type, and its weight drops to zero only when all references to it are destroyed and their corresponding tasks are processed. Since communication overhead is usually more expensive than space overhead, the reference weight scheme represent an improvement over the reference counting scheme, because it reduces such communication overhead, the communication overhead is reduced to approximately 50%. The disadvantage is that it requires more space in each object reference for saving the weight part. It is done as follows: when an object A creates a new object B, a weight W is associated with B and this value becomes a part of the created object reference to B.

In this scheme, when the weight part of an object reference reaches 1 and that reference needs to be copied, the object's weight has to be increased before making a new copy of that object reference. An "add-reference-weight" task has to be generated and sent to that object to increase its weight by some value. The added weight value plus 1 split to form the weight parts of the two object references as before. Associating high weight with each newly created object minimizes such add-reference-weight tasks but requires more space for the weight field in each object and each object reference.

3.5 Local GC Algorithm

In this scheme [1995] [Barbara Liskov , Rivka Ladin, 1986] , we avoid invoking the global garbage collection each time a processing element needs to perform garbage collection. In systems with high locality of reference and with rapid generation of local garbage, performing local garbage collection before global, one may free enough storage space to make a processing element continue its normal computation without engaging the entire system as is required during global garbage collection. Using this scheme in such systems, the number of global garbage collection invocations can be decreased and thereby the total garbage collection overhead will be decreased and the overall system performance will be increased. This is because, the space, time and communication overhead of the global garbage collection is much higher than a local sequential uni-processor garbage collection.

The following example illustrates how this algorithm operates:

Suppose we have two computers P and Q are connected together and their heap is full, so they need garbage collection so it will goes in the following scenario.

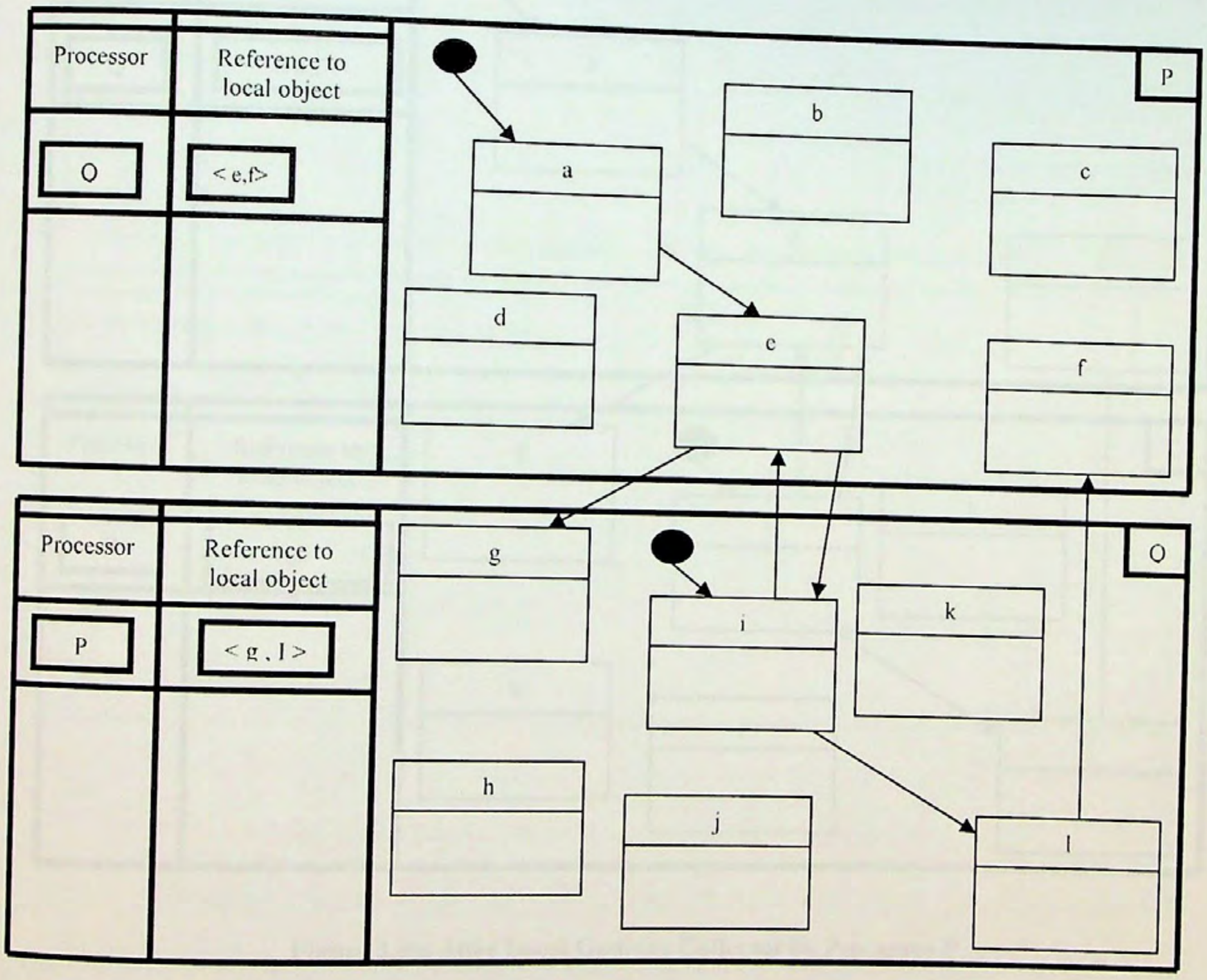

Figure 3.6a: The heap of two computers P & Q before GC

The above diagram shows how the heap looks before garbage collection in both computers and how objects are referenced remotely from the root. While the following figure shows processor P after doing garbage collection locally and how did it keep tack of the remotely accessed objects.

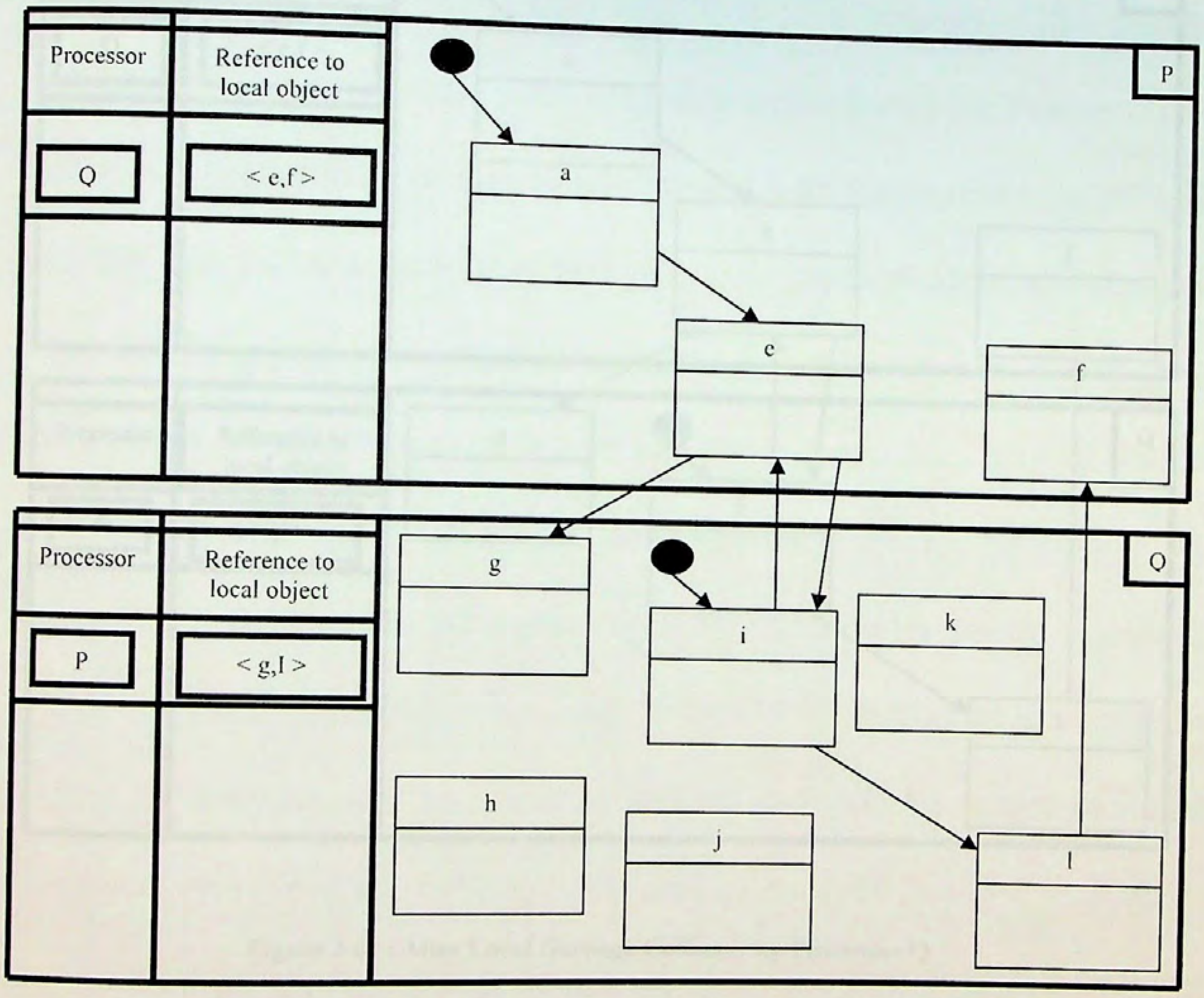

Figure 3.6b: After Local Garbage Collector by Processor P

Then processor Q did its garbage collection cycle, and also emptied its heap locally to have free space. The following figure shows this:

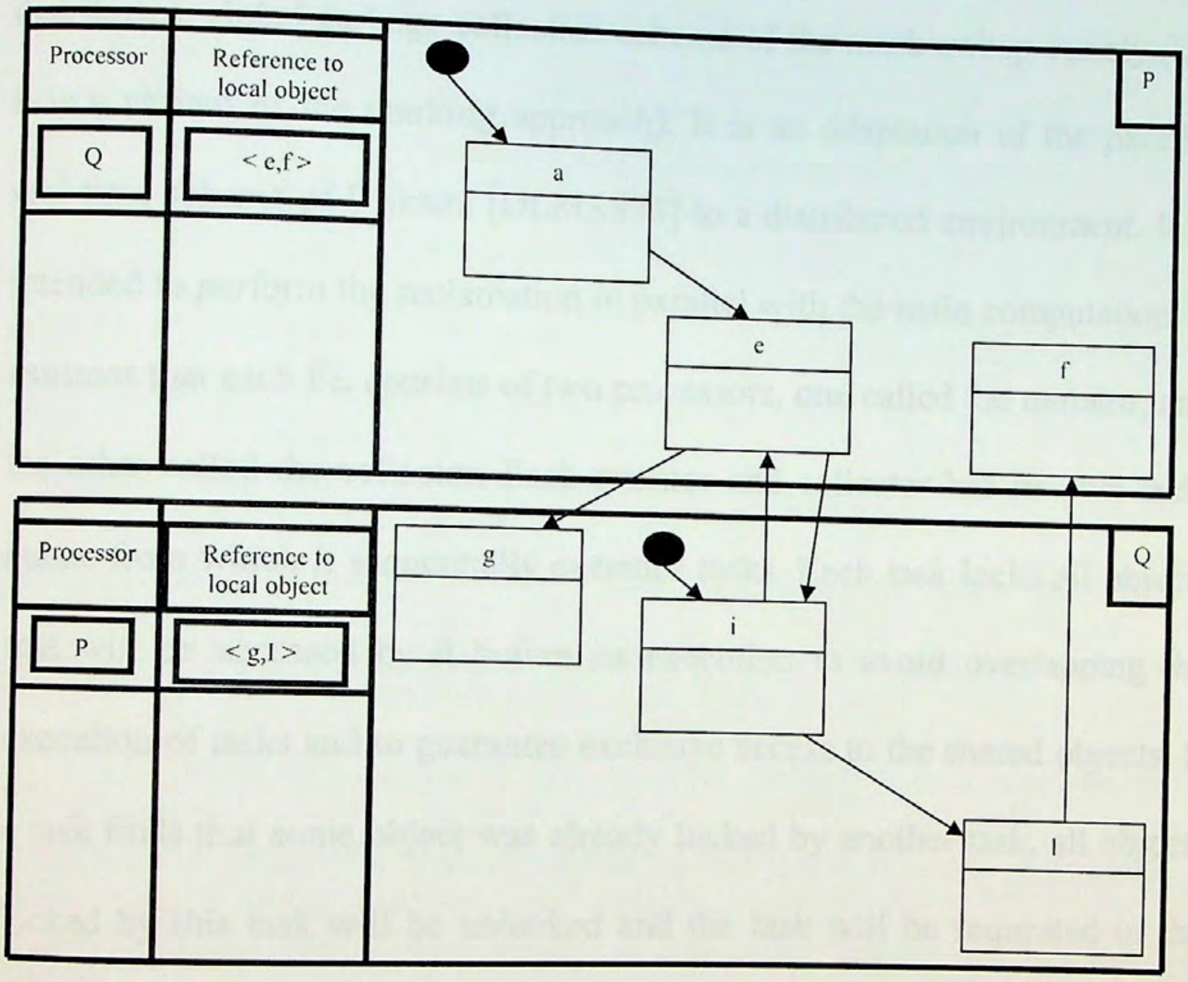

Figure 3.6c : After Local Garbage Collector by Processor Q

As we had seen in the above example how the local GC algorithm works on multiple processors in a distributed system environment. Each processor in the distributed environment goes in each garbage collection cycle.

3.6 Marking-tree Collector

The marking-tree collector [Huke82] [Özalp Babaoğlu, Keith Marzullo, 1993] [Henri E. Bal, 1990] [John K. Bennett, 1987] is a real-time, distributed, global garbage collection scheme of the mark-sweep variety (i.e. it is a variant of the marking approach). It is an adaptation of the parallel real-time scheme of Dijkstra [DLMSS78] to a distributed environment. It is intended to perform the reclamation in parallel with the main computation. It assumes that each PE consists of two processors, one called the mutator, and the other called the collector. Each mutator and collector has its own task-queue from which it sequentially executes tasks. Each task locks all objects that will be accessed by it before its execution to avoid overlapping the execution of tasks and to guarantee exclusive access to the shared objects. If a task finds that some object was already locked by another task, all objects locked by this task will be unlocked and the task will be requested in the same queue to prevent deadlock. Since objects involved in a task may reside on different processors, this locking mechanism introduces high processing time and communication overhead when the collector and the mutator have high degree of contention to shared objects. There is only one root of the graph of objects distributed over the entire system. The collector performs one garbage collection after the other beginning with the root. It can reclaim all garbage, even unreachable cyclic structures. It is not able to perform storage compaction, but a modified version of this marking-tree collector combined with the copying approach, in which compaction of memory is accomplished on each PE, is presented by Hudak[Hud82]. But this copying version is more complex, and the inter-processor communication overhead of the marking-tree collector for the purpose of updating remote pointers.

Unfortunately, the space requirement of the collector queues of the marking-tree collector and its copying version is very high and very difficult to determine due to the dynamic nature of the graph and the parallel breadth-first strategy of tracing the graph. Such schemes can work only if the collector does not run out of space, otherwise, deadlock will result.

The space requirement of the collector queues is very high, because the parallel breadth-first strategy of tracing the graph is used. In general, it allows large number of simultaneous tracing tasks. The dominating part of the collector queue's space requirement is due to the room required for such mark tasks. The maximum number of such tasks that could exist at any one instant of time should be determined and their required space should be reserved for the collector to avoid it running out of space. Unfortunately, in the worst case the number of mark tasks is approximately equal to the number of arcs in the computation graph due to the parallel breadth-first strategy of tracing the graph.

3.7 Distributed Global Scheme

3.7.1 Global Mark Phase

One task of the global mark phase [Chambers et.al 1984] is to guarantee that all objects referred from messages in transit are investigated before completing this phase. Guaranteeing arrival and investigation of such messages is crucial, because it may happen that objects in the system are reachable only from such massages, and if these massages are not investigated and reachable objects from them are not marked during the global mark phase, such objects will erroneously be garbage collected during the sweep-compaction phase.

3.7.2 Computation Messages in Transit during the Global Mark phase

3.7.2.1 Storage Problem of Marking Messages

In any uni-processor garbage collector a work space is needed and has to be available before invoking the collector. Shortage of the required work space may stop the task of the collector. In distributed garbage collectors there are two sources of storage space requirement, the first one corresponds to what is needed in a uni-processor collector, while the second is needed for garbage collection messages. There are two categories of garbage collection messages: 1) synchronization messages and 2) marking messages. All types of messages found in any distributed environment belong to the first category. Their number in the worst case can be easily determined. Also, their storage requirement can be determined and guaranteed to be available before performing the garbage collection.

Shortage of storage space may stop the task of the collector as mentioned before and possibly lead to deadlock. Therefore, we want to design a marking scheme with a limited and predetermined storage space requirement. The scheme should also allow as much concurrency in the system as possible.

3.8 Global - Local Schemes

3.8.1 Performance

In this section we discuss the important performance aspects of the global schemes and local-global scheme [Jacques Cohen , Alexandru Nicolau, 1983] [George F. Coulouris , Jean Dollimore, 1988] [Robert Courts, 1988]. We discuss and determine the space and communication overhead for each scheme. We also discuss the relative performance of our schemes presented in this chapter with respect to the marking-tree collector [Huke82].

The two primary sources of space overhead in our schemes are the extra space embedded within the graphs of objects and the room for the garbage collection message queues.

3.8.2 Space storage problem

There is an important aspect concerning the memory space required for computation message queues and the flow control mechanisms for the computation messages in system. The most important problem is the space requirements of the garbage collection queues and the respective flow control mechanisms.

3.8.2.1 Object Overhead

Each object requires extra space which is needed for garbage collection purposes. This extra space is the same for the first two schemes and is estimated as follows. A single bit is required for each In-Use and mark-Bit fields. Also some bits are required for the child-Count field. The size of this field depends on the maximum number of children of an object. A few bits are required for the last-Child field. The size of this field depends on the maximum size of an object. One object reference cell is required for each object for the parent-Ref field. Only a single bit more is required for the Ext field in the third scheme. This means that the extra required space for each object is one memory cell plus a few bits per each object.

3.8.2.2 Garbage collection Queue Overhead

The second form of space overhead is the room required for the garbage collection queue of each PROCESSING ELEMENT. We will agree that the garbage collection messages consist of synchronization and marking messages. So for each scheme, we first discuss and determine the total number of the synchronization messages and the space requirement for these messages. Third, we discuss and determine the required space for the garbage collection message queue in each PROCESSING ELEMENT.

3.8.3 Global Schemes where there is a certain master

3.8.3.1 Synchronization Messages

Synchronization Messages = s start – garbage-collection

+ (N – 1) start-mark-phase

+ (N – 1) end-mark-phase

+ (N – 2) 2 block

+ (N – 1) sweep-and-compact (*Eq.*3.1)

Where N is the total number of Processing Elements in the system, S is the number of normal processing elements that simultaneously invoke the global garbage collection. The range of s is from 1 to N – 1.

3.8.3.1.1 Space Requirements for Synchronization Messages

The space requirements for synchronization messages are determined as follows.

The master: In the worst case, N – 1 start – garbage – collection and N – 1 end – mark – phase messages may simultaneously be received by the master [Ali84]. All synchronization messages are of the same size. Therefore, the master requires storage space as large as 2 *(N – 1) * message size. The size of each message is equal to the size needed for coding the message type (i.e. a few bits) plus the size needed for a link field. The size of the link field can be less than one full memory cell.

A normal processing element: **In the worst case, N – 2 block messages and one start – mark – phase message may simultaneously be received by any normal processing element. Therefore, any normal processing element requires storage space as large as (N – 1) * message size [Ali84].**

3.8.3.2 Marking Messages

3.8.3.2.1 Number of marking messages

The number of mark-object messages is equal to the number of reply-mark – object messages which is equal to the number of reachable arcs in the graphs.

3.8.3.2.2 Space requirements for marking messages:

The space required by each processing element for marking messages is determined by the size of the maximum number of these messages that may simultaneously be received by any processing element in the system.

Total Space Requirements for Garbage collection Messages

Since both synchronization and marking messages may simultaneously be received by any processing element during the mark phase, then the total space required for each processing element is the total space requirements for the synchronization messages plus the marking messages.

That is, the master requires extra space for garbage collection queue equal to.

2 * (N – 1) * (size of a synchronization message).

+ M * (size of the reply – mark – object message). (*Eq*.3.2)

And any normal processing element requires extra space for garbage collection queue equal to

3.8.3.3 Performance

3.8.3.3.1 Space Overhead

The space overhead of the scheme consists of three parts; space overhead per object, space overhead of maintaining references and space overhead of garbage collection queues.

3.8.3.3.2 Space Overhead per Object

The space overhead per object is only one bit for the garbage collector field. Since information stored in the garbage collector field can be determined from the status of Loc field, the space overhead per object can be avoided.

3.8.3.3.3 Space Overhead of Maintaining References

This storage consists of two parts; the static (constant) part the dynamic part. The constant space overhead for each processing element is the space of the tables. The space of the head vectors of the In Table and the out table is from two to four memory cells times the number of processing elements in the system. The space of the head of each list is equal to one memory cell. (This constant space is relatively small.) The dynamic space overhead is the out Elements, because the heap space of garbage objects is used for constructing the out table, i.e. for creating out Elements. The number of the used tem-elements in the system is approximately equal to the number of activities in the SQs, i.e., the number of activities that are simultaneously accessing non-local objects. The dynamic space overhead for maintaining references depends directly on the locality of references; it decreases with increasing locality of reference and vice versa.

It is not space efficient to represent local references and remote references in equally size cells. Further work can be done to allow local references and different types of objects to be represented more efficiently.

(N – 1) * (size of a synchronization message)

+ M * (size of a reply – mark – object message). (*Eq*.3.3)

The total space requirements can be decreased if we restrict the parallelism in the scheme by dividing the global mark phase into two sequential parts. In the first part we empty the communication subsystem from the in transit computation messages. In the second part we mark objects. In this case, the synchronization and marking messages cannot simultaneously exist in any processing element, and only the largest one of these two spaces is required. The disadvantage of that solution is that the processing element will be idle for a longer time, which in turn increases the time of garbage collection.

3.8.4 Global scheme where any processing element can be a master

3.8.4.1 Synchronization Messages

Total number of synchronization messages:

$$
\begin{aligned}
&= s\,(N-1)\ \text{start} - \text{mark} - \text{phase}\\
&+ (N-s-1)\ 2\ \text{block}\\
&+ (N-1)\ \text{end} - \text{mark} - \text{phase}\\
&+ (N-1)\ \text{sweep} - \text{and} - \text{compact} \quad (\textit{Eq}.3.4)
\end{aligned}
$$

Where s is the number of normal processing element that simultaneously request the global garbage collection invocation. The range of s is from 1 to N – 1. The same comment about the number of block messages as in the first scheme.

3.8.4.1.1 Space requirements for synchronization messages:

The space required for these messages in each processing element is determined as follows. In the worst case, N – 1 of start – mark – phase or block messages, and N – 1 end – mark – phase message may simultaneously be received by any processing element. In this scheme, the size of the start-mark-phase is larger than the size of the block phase by the size needed for coding the sender. Then each processing element requires storage space equal to the size of (N-1) start-mark-phase messages plus the size of (N – 1) end-mark-phase messages. The size of the start-mark-phase message is equal to the size needed for coding the message type plus the size needed for coding the names of processing element plus the size needed for the link field. The size of the end-mark-phase message is equal to the size of any of the synchronization messages of the first scheme.

3.8.4.2 Marking Messages

3.8.4.2.1 Number of marking messages

The total number of mark-object and reply-mark-object messages is less than the respective number in the first scheme. In this scheme there is no communication overhead for already marked local objects. The amount of this communication overhead that can be reduced depends on the locality of reference. With increasing reference locality the marking communication overhead will decrease.

3.8.4.2.2 Space requirements for marking messages

Total Space Requirements for Garbage Collection Messages

The total space required for each processing element is equal to the respective space requirements for the master in the first scheme plus (N – 1) times the space needed for coding the name of a processing element. This is because a start-mark-phase message in the second scheme contains one more field than in the first scheme, which identifies the sender processing element.

3.9 Local – Global Scheme

It works exactly as the Global-Local schemes, with all its characteristics, advantages and disadvantages. But still it had certain criteria for calculating its speed and performance so that we could compare with other algorithms.

3.9.1 Speed

In the first scheme, for each accessible arc one mark-object message and one reply-mark-object message is sent, i.e. each arc is visited twice during the mark phase. In the second scheme, each arc to an already locally marked object is visited once while each other arc is visited twice.

In the local-global scheme, each processing element first performs a fast local garbage collection. Global garbage collection is invoked only when the local garbage collection can not free enough space. The local garbage

```
XL error
   Subsystem:   KERNEL
   Error:       IllegalTag
   Operator:    0x1b
   Position:    16966
```

Chapter 4

Global Incremental Garbage Collection Algorithm

Chapter 4

Global Incremental Garbage Collection Algorithm

Chapter 4
Global Incremental Garbage Collection Algorithm

4.1 The idea of the Algorithm

As in Baker's incremental GC algorithm, each memory operation performs some GC work and the heap space is divided into two semispaces (To & From), and new objects are created in the To semispace and accessible objects in the From semispace will be moved incrementally to To semispace during computation. Once all accessible objects are in To semispace, the system flips the roles of the two semispaces and moves first objects accessible from the roots of the computation into the new To semispace. Again, new objects will be created into the new To semispace and all objects accessible from the objects in the new To semispace will be moved into the new To semispace as parts of memory operations.

Our global GC algorithm is a generalization of Baker's scheme to distributed systems environment, where the global heap of a distributed system is spanned over local memories of the processing elements in the system and the global heap is divided also into two semispaces as shown in figure 4.1.

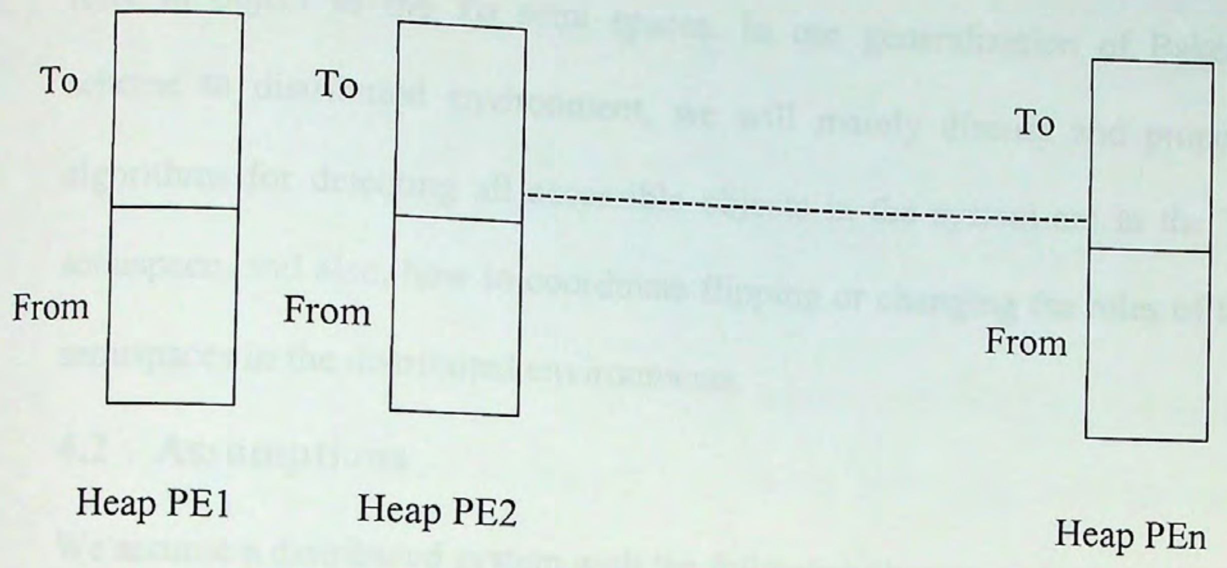

Figure 4.1: The Global Heap

Here also each memory operation performs some GC work and new objects in the system are created in the To semispaces and all objects accessible from the system roots are moved into To semispaces and when it guarantees that all accessible objects are in the To semispaces, the roles of the To and From semispaces can be changed.

In single processor systems detecting the situation where all accessible objects are in the To semispace is very straight forward (when S = B in Baker's algorithm). Flipping or changing the roles of the two semispaces is a very simple atomic operation. In distributed environments, we require new mechanisms for detecting the situation in which all accessible objects in the system are in the To semispaces and all object references in the network refer to object in the To semi spaces. In our generalization of Baker's scheme to distributed environment, we will mainly discuss and propose algorithms for detecting all accessible objects in the system are in the To semispace, and also, how to coordinate flipping or changing the roles of the semispaces in the distributed environments.

4.2 Assumptions

We assume a distributed system with the following characteristics:

- We have a number of PEs say n; the heap of the system of n PEs is spanned over local memories of n PEs.
- Each PE heap consists of two parts (To & From).
- Each PE has its own local root for the heap objects.
- One of the PEs is assigned as a master for coordinating, starting and termination of each new GC cycle.
- No assumption about the order of message, i.e. it is a free distributed environment.
- We will assume that there is enough memory space in each processing element for completing one global GC cycle.
- During computation each memory operation performs some GC work as in Baker's scheme.

4.3 General Algorithm

We look to the whole system as in one of the two states: Computation state where all processors perform GC & computation, and Synchronization state where all processors perform switching the roles of the semispaces.

In the computation state when the master moves all its accessible objects in the system To spaces. The master coordinate with the other processors to make sure that all objects accessible in the system are in the To semispace. It does that by invoking an algorithm called termination algorithm. When the master detects that all accessible objects in the system are in the To space, it starts another phase called switching phase to flip the roles of the two semispaces in the system.

When the master detects successful flipping for the whole system, it starts a new cycle of computation and garbage collection by informing all processors to do that. The essential parts of the algorithm are the termination algorithm and switching algorithm. These two algorithms are presented in the remaining part of this chapter.

4.4 System state transition diagram:

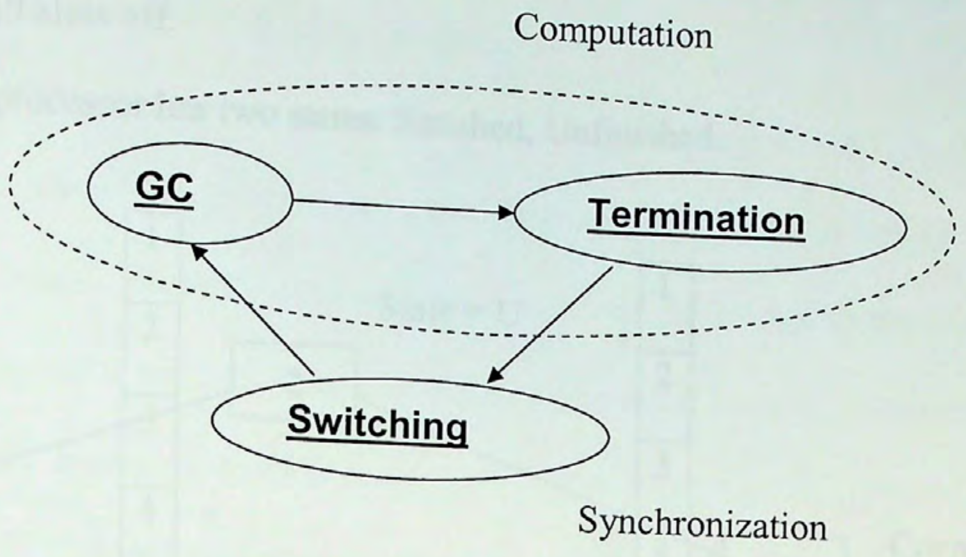

Figure 4.2: System State Transition Diagram

The above diagram shows how my system actually works, we have three main states which compose the three main algorithms that we had implemented. The GC state in which we apply Baker copying algorithm, the other states are the switching algorithm where the flipping of the two semispaces and the termination algorithm which is the topic of this thesis, which detects the termination of the garbage collection cycle.

4.5 The termination algorithm:

- We assume the system logically forms a ring.
- There is one master, has a counter and its initial value is set to zero.

- There is a vector circulating in the system,one slot for each processor, initially all slots off.
- Each processor has two states: Finished, Unfinished.

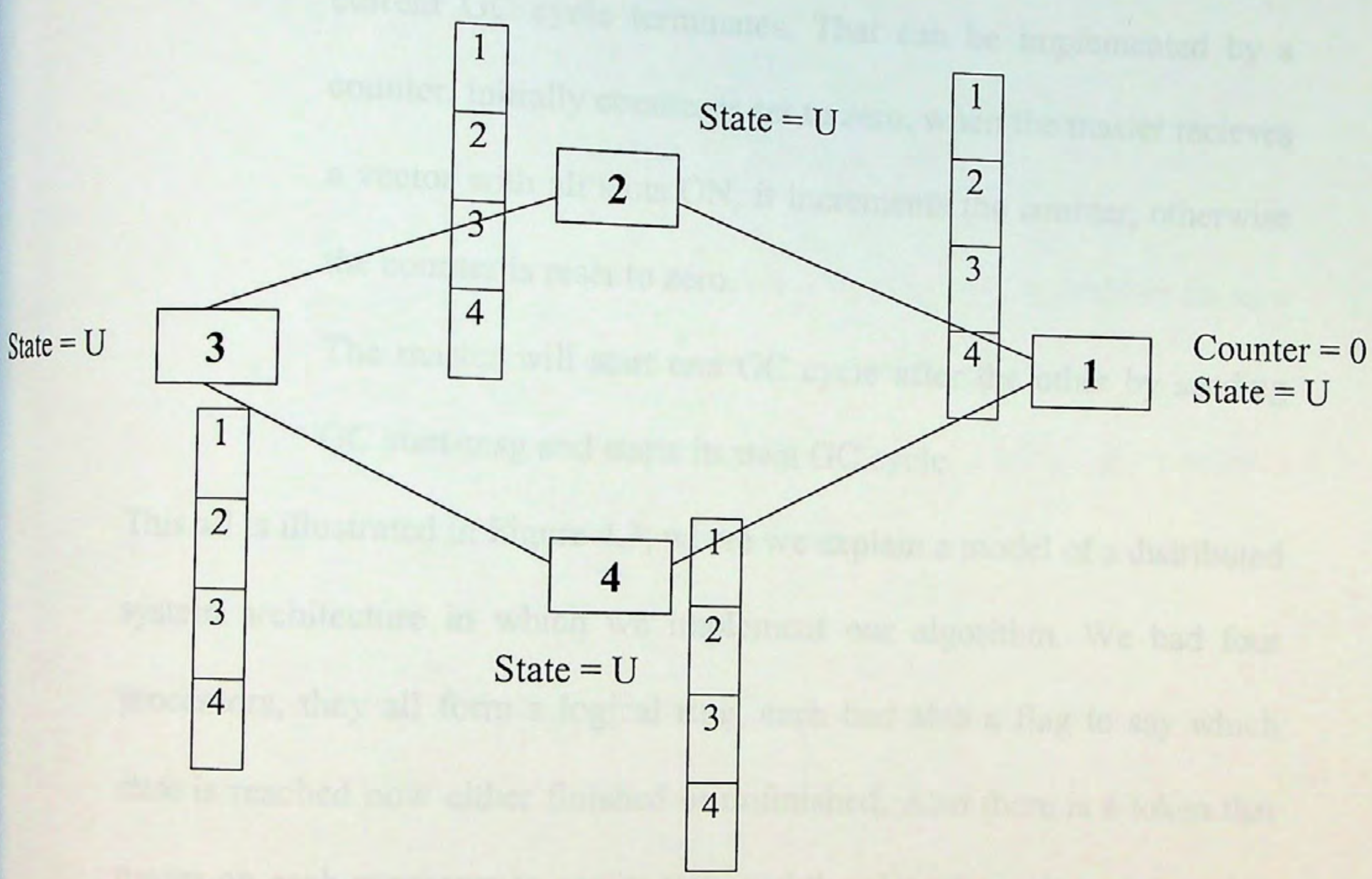

Figure 4.3: An Example of Distributed System Model

- When a processor finishes its GC, it changes its state to F.

When the vector reaches a processor and its current state F, it sets its corresponding slot to ON, otherwise to OFF.

- When a processor in state F and recieves a GC message, it sets its state to U and replies to the sender.
- When the master recieves two consequtive vectors all ON, the current GC cycle terminates. That can be implemented by a counter, initially counter is set to zero, when the master recieves a vector with all slots ON, it increments the counter, otherwise the counter is reset to zero.
- The master will start one GC cycle after the other by sending GC start-msg and starts its own GC cycle.

This all is illustrated in Figure 4.3, where we explain a model of a distributed system architecture in which we implement our algorithm. We had four processors, they all form a logical ring, each had also a flag to say which state is reached now either finished or unfinished. Also there is a token that passes on each processor to see its state and the algorithm takes place as we described above.

4.6 Switching Algorithm

When the master detects the termination of GC cycle, the master intiates the switching algorithm. When the master recieves acknowledgement from all PEs responding to the switching msg, it will start a new computaion and GC cycle by sending to all PEs a GC-message.

4.6.1 Global Cycle

Once a global GC cycle is started, it will go as follows :

- The master informs all processors to switch its semispaces by sending message say (switch_msg).
- When a processor recieves a switch_msg, it switches its state and replies to the master with switch_reply_msg.
- When the master recieves all replies, it sends a start_computation and GC_msg to start the computation & GC.

To implement this start phase of a new GC cycle, we have a counter, intially is set to zero. The counter is incremented by sending a msg and decremented by receiving a reply msg, when it recieves all replies the counter will be zero.

4.7 Master & Slave State Transition Diagrams

4.7.1 Master States

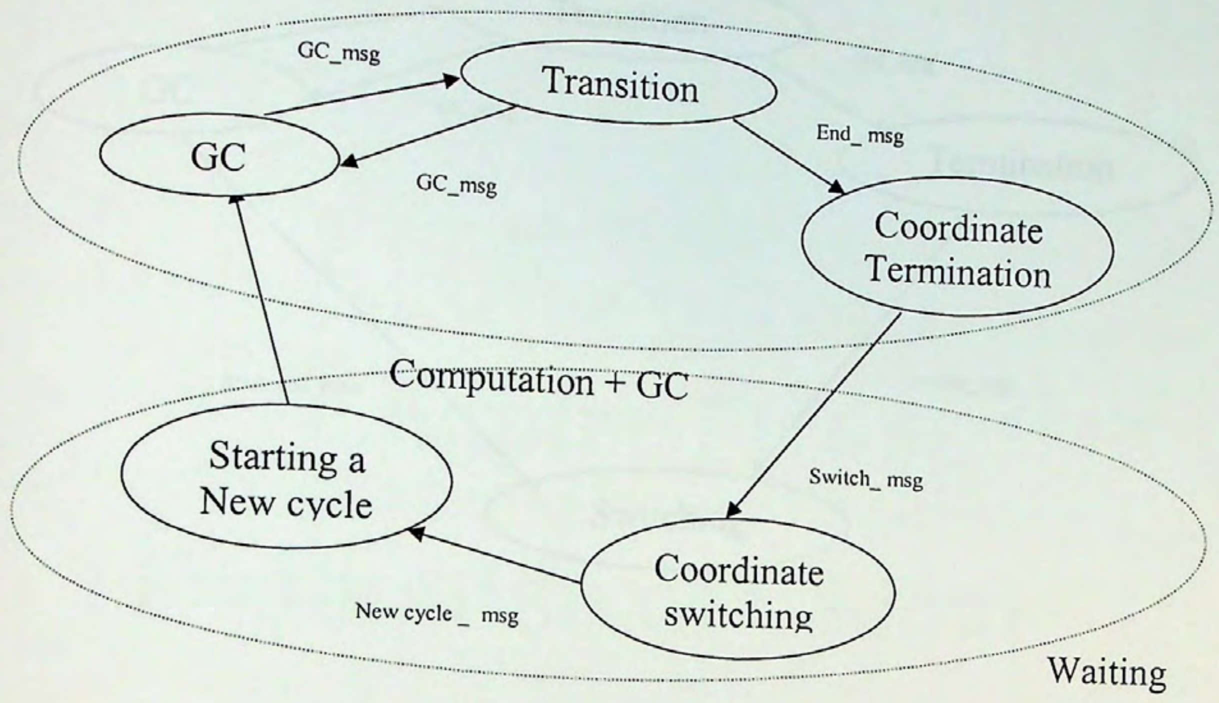

Figure 4.4: Master State Transition Diagram

The master states is subdivided into two sections, in one of them the garbage collection works along with the computation is taking place, while the other section the system stops completely to do the flipping and starts a new cycle.

4.7.2 Slaves States

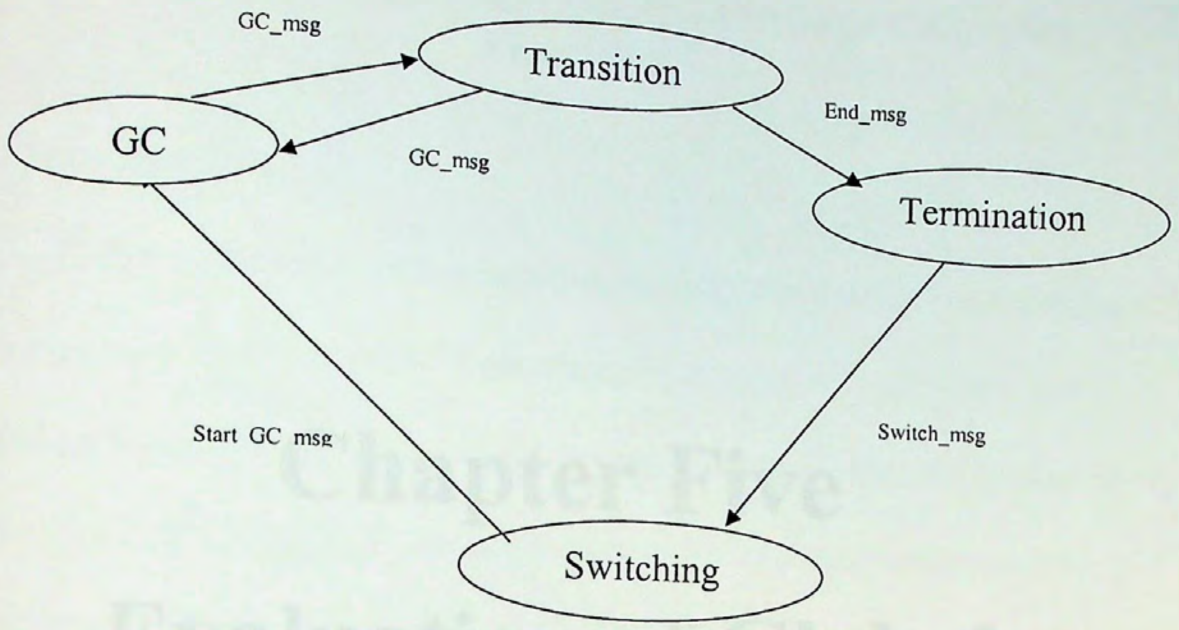

Figure 4.5: Slave State Transition Diagram

The slave states are four states; the first state is the garbage collection (GC), while there is a transition state in which a processor may finish its garbage collection while according to other processors it may have some more garbage so it must do some more garbage collection work. The other states are the same states as for the master states.

Chapter Five
Evaluation of Global Incremental Garbage Collection Algorithm

Chapter 5
Evaluation of Global Incremental Garbage Collection Algorithm

5.1 Introduction

Global garbage collection scheme is a distributed object-based system running on a set of workstations connected by a local-area network. The global garbage collection scheme employs garbage collectors on each node in the distributed system. To reduce the latency introduced, collectors work concurrently with other processes by copying all accessible objects to the To space, while other processes are executing without holding them back. A comprehensive collection of all garbage in the entire distributed system is achieved by the cooperation of one collector from each node. These global collectors exchange information about reachable objects during their copying and synchronize their initialization and termination of the GC cycle. The scheme is robust to temporary node failures and depends only on nodes being pair-wise able to exchange messages. As the comprehensive collection depends on information about the entire distributed graph of objects and references between them, any node failure may postpone the termination of its cycle. To achieve more expedient collection the other set of collectors' does node-local collection on each node using a root set extended with the objects potentially known from other nodes.

One of the most important characteristics of our algorithm is its ability to collect all garbage that existed at the start of the algorithm. We say that the garbage collection is comprehensive. Our algorithm relies on a faulting mechanism to enable user processes to continue during garbage collection. Moreover, our distributed garbage collection scheme is robust to node failures, i.e., it is able to survive temporary partial failures and to progress in non-failed part of the distributed system.

Our solution is based on the Baker's garbage collection technique [Baker 82] extended with an object protection and faulting technique to enable concurrency with user processes (called mutator). In this way, garbage collection is done "on-the-fly" without introducing complicated synchronization schemes to preserve consistency. To do collection in a distributed environment, the collector is also distributed. The distributed collector takes advantage of distributed control to facilitate robustness to failures in parts of the distributed system. In general, distribution challenges garbage collection in several ways, e.g., true concurrency among user and collector processes on different nodes, objects moving around, inter-node references, and partial and temporary failures of the system.

The advantages of garbage collection have, however, made proposals for, and implementations of, garbage collectors in distributed systems popular. Due to the many challenges most of these proposals remain unimplemented (80 % of the surveyed distributed garbage collectors in [Abdullahi 92]).

5.2 Why Our Algorithm?

Actually according to many algorithms, we can perform garbage collection globally and without hanging ups the system, but there was always one major problem which is how to detect the termination of each GC cycle. Our algorithm solved this problem effectively without interrupting the processing and also without losing any data in any of the semi spaces. Also, failures in distributed systems need not stop the entire system, thus robustness, i.e., survival of failures in part of the system, is also a goal in many of these systems. In our failure model, failures become visible to the garbage collector as previously, currently, or permanently unavailable objects or nodes. The various distributed garbage collectors may combine robustness with each of the previously mentioned features in chapter 1.

Such collectors support one or more of the features shown in table 5.1:

Concurrency & Robustness	By enabling mutator to run although a collector may be blocked by a failure.
Comprehensiveness& Robustness	By continuing a comprehensive collection on the currently available part of the system, resuming collection on the other parts as they become available again after a failure, and detecting permanent unavailability.
Efficiency & Robustness	By limiting the overhead due to robustness to be paid mostly when actually facing failures.
Expedience & Robustness	By allowing garbage collection in available parts of the system to complete a partial collection independent of the unavailable parts.

Table 5.1: Global GC evaluation criteria

This thesis considers the design and implementation of such garbage detection in a distributed system. Our solution combines the comprehensiveness of copying & incremental collection with distribution and concurrency. Our garbage detection is:

- comprehensive,
- works concurrently with user processes by using a faulting mechanism, and

- detects garbage although part of the distributed system is temporarily unavailable, i.e., it is robust to failures.

It features independent collection of local garbage on each node and efficient (but potentially slow) collection of all garbage in the distributed system while only depending on pair-wise availability of nodes.

5.3 Comparison of Basic Complexity

The difference between the mark-and-sweep and the copying collection is mainly in the reclamation part. The difference between any graph traversal algorithm and reference counting is their perspective. Graph traversal uses a global view, traversing from the root set to all reachable objects. Whereas reference counting uses the local view, the object is reclaimed when the counter is decremented to zero, as the last reference to the object is deleted.

The overhead introduced by these algorithms can be roughly estimated. We use n as the total number of objects, and l denotes the number of live objects. Thus the goal is to reclaim.

Algorithm	Time complexity	Storage need
Mark-and-sweep collection	$O(l+n)$	$O(n)$
Copying collection	$O(l^2)$	$O(n)$
Reference counting	$O((n-l)+a)$	$O(n)$

Table 5.2: The basic cost in time and space of traditional garbage collectors

n	total number of objects.
l	number of live objects.
a	number of assignments.

$n-l$ garbage objects. As the number of references in each object influences the complexity added when traversing the individual objects, this number could also be given as could the almost proportional number that indicates the size of each object.

To estimate the complexity of the three basic algorithms and do comparison we must, however, look at the main factors of a large system, i.e., the number of garbage and non-garbage objects: *(n _ l)* and *l*. Thus the estimates are given in *O (n)*.

5.3.1 Time Complexity

The time complexities of the three algorithms are:

5.3.1.1 Mark-and-sweep collection

A traversal that identifies the live objects by the repeatedly scanning method may cost $O(n^2)$ in the worst cases. By adding a simple structure that contains references to gray objects during the mark-phase, the traversal is limited to $O(l)$. In all cases, a storage sweep needs to look at all objects, i.e., $O(n)$. Thus mark-and-sweep collection has a run-time complexity of

$O(l + n) \leq O(2n) = O(n)$.

5.3.1.2 Copying collection

The copying collectors need only traverse the live objects, a work proportional to the references in the live objects, l, plus copy the live objects, a work proportional to l, plus update the references between the copied objects, a work proportional to l^2. In total a copying collection has a run-time complexity

$O(l + l + l^2) = O(l^2) - O(n^2)$

5.3.1.3 Reference counting

Reference counting slows down all computation that updates references by at least two memory references to update a counter. When a garbage object is identified, additional costs are paid to update the counters of its references. Reference counting has a run-time complexity proportional to $(n - l)$ for reclamation, plus a cost proportional to the number of reference updates (assignments), a. In total reference counting costs $O((n - l) + a)$. To do comparison with the other algorithms, the number of assignments is said to be proportional to the number of objects, thus reference counting has a run-time complexity $O(n)$.

These costs are summarized in Table 2.1 together with an estimate of storage overhead.

5.3.2 Storage Overhead

The storage overhead due to the three algorithms are:

5.3.2.1 Mark-and-sweep collection

A mark-field per object, which is able to contain three values, i.e., 2 bits per object. Thus the needed storage is $O(n)^{1}$.

5.3.2.2 Copying collection

Half the storage is reserved for garbage collection (To Space). Thus a maximum of half the storage is available for the application, i.e., a cost of $O(n)$. Furthermore, translation tables may take up $O(l)$ space during copying, thus the needed storage is $O(n + l) = O(n)$.

5.3.2.3 Reference counting

A counter per object, which is able to accumulate values between *0* and *n*, i.e., log n bits per object. The needed storage is thus *O (n)*. These basic algorithms are extensively described in the literature before 1980, an overview in terms of list processing systems is given by [Knuth 68] and more thoroughly by [Cohen 81]. Since then, more advanced methods have been developed, but essentially they are all based on these three basic techniques.

5.4 Evaluation of Distributed Garbage Collection Algorithm

According to all what we surveyed in the previous chapters, we had developed the evaluation set for our algorithm to the other distributed algorithm. This evaluation will be discussed in the next section and it is summarized in the following table:

Algorithm	Concurrency	Comprehensiveness	Efficiency	Expedience	Termination
Mark-and-sweep collection	Concurrent	Collects all garbage	Not Efficient	Moderate	N/A
Copying collection	Concurrent	Collects all garbage	Efficient	Fast	N/A
Reference counting	Concurrent	Does not collect all garbage	Not Efficient	Fast	N/A
Global Incremental GC	Concurrent	Collects all garbage	Efficient	Fast	detects Termination

Table 5.3: Comparison of different algorithms

5.4.1 Reference Count Evaluation

Reference counting is inherently incremental, but the overhead introduced on each reference assignment makes it inefficient [Ungar 83, Baden 83]. Each implicit or explicit assignment of references in the application results in one or two reference counters being updated. Given two variables X and Y, then the assignment X← Y results in an incremented counter for Y's object and a decremented counter for the object originally referenced by X. Furthermore, reference counting does not collect cycles of garbage (self-referential groups of garbage). As reference counting may fail to detect all garbage, the undetected garbage may also prolong the life of other objects, i.e., objects only reachable from undetected garbage are also kept alive.

+ Advantages:

- Concurrency with mutator, i.e., very fine-grained interleaving.

_ Disadvantages:

- Non-comprehensive because it does not collect cycles of garbage.
- Inefficient because of the large overhead on each assignment.

5.4.1 Mark and Sweep Evaluation

Mark-and-sweep collection has to traverse all live objects once and to sweep the entire object storage twice. The first sweep clears all mark-fields to white, and the second reclaims the white objects in the sweep-phase. Further complexity is added to identify all gray objects for traversal. The basic algorithm repeatedly scans the object store from top to bottom for gray objects. During each scan, objects located between the top and the current position may become gray, thus a new scan is needed.

+ Advantages:

- Collects all garbage including cycles of garbage.
- The main work is proportional to the live objects.

_ Disadvantages:

- Needs to traverse entire object storage for reclamation of garbage.
- May lead to storage fragmentation.
- Stops all other activities in the system while running.

5.4.3 Copying Evaluation

Copying algorithms have the advantage over reference counting and mark-and-sweep, that they also compact the storage, thus preventing fragmentation, at the cost of updating references and the loss of approximately half the storage due to the partitioning in a From Space and a To Space.

+ Advantages:

- Collects all garbage including cycles of garbage.

- Work in time proportional to the number of live objects.
- Compacts storage, thus preventing fragmentation, and improving locality of reference.

_ Disadvantages:

- Wastes 50 % of the storage reserved for the new To Space.
- All references must be updated and/or indirect/relative references used.
- Stops all activities in the system while running.

5.5 Evaluation of Global Incremental GC algorithm

The comprehensive, concurrent, and robust distributed garbage collection scheme described in the previous chapter has been incorporated in our algorithm. The scheme does collect all garbage and works during long sessions without storage leakage. In contrast to most distributed collectors, this collector does collect distributed cycles without any special effort. The distributed collection will always complete even when node failures occur frequently. The general overhead has been limited to less than 10%, and the pauses incurred on user computation may be limited to the same order of magnitude as the current time slice used for round-robin scheduling of user processes. Furthermore, orphan processes are collected and the scheme may be used to recycle any kind of resource represented as an object. A set of evaluation methods was presented in Section 1.4. Measurements as well as estimates of the collectors' run-time behavior have been done to identify both the short and long term overhead in space and time introduced. The measurements have been backed up by a program that creates artificial garbage and by ordinary garbage collection applications. This section evaluates the implementation by investigating how the goals have been meet. The first section shows that comprehensiveness has been achieved without introducing storage leakages (Section 5.5.1). In Section 5.5.2 we show how the scheme survives node failures and are able to progress in a system where node failures are the norm. Some structures of objects may harass a garbage collector more that others. In Section 5.5.3 we show how the traditional problem due to cyclic garbage is solved without any special effort, and in Section 5.5.4, how one of the most challenging distributed structures of live objects, the zipper problem is overcome. The overhead introduced by garbage collection has been measured and we show some of the basic performance figures in Section 5.5.5. The pauses introduced into normal execution by the garbage collector have also been investigated and the measurements showing this are presented together with estimates of the pauses in general (Section 5.5.6).

5.5.1 Comprehensive Collection

A key feature of the implemented collector is its ability to collect all garbage. To test that the collector indeed does collect all garbage and nothing else, a mixture of user programs has been run while garbage collecting. This shows that all the objects that were garbage at the start of the collection are eventually reclaimed by the succeeding global collection. We have conducted this test by inspecting the object store at four central points:

1. before the programs were executed,

2. between the termination of the programs and the start of the collection,

3. between the end and the start of the copying phase, and

4. after a full collection is done.

Object type	Data	Code	Object	Process	Total
basis contents of a new kernel	28	79	104	2	213
after program load, before it starts	68	111	174	3	356
after a full collection at this point	49	107	168	3	327
after program termination	4317	121	4213	8	8559
after a final full collection	56	116	190	4	366

Table 5.4: The number of objects in the object store

After the user programs have filled up the object store, the garbage collector must identify and remove all these objects again. The test programs generate 4000 objects explicit known to be garbage after termination, and provoke the system to generate further objects, that may survive the test programs, as discussed later in this section.

A summary of this test is presented in Table 5.1. The columns show the current number of objects in the object store. The total number of objects has been broken down into the number of objects representing any of the four categories:

Data	i.e., mostly objects containing user data
Code	i.e., the executables related to either system objects or the user programs.
Object Management	i.e., objects generated by the run-time system for the management of all objects.
Process Management	i.e., objects representing user processes and user defined process queues.

The test shows that all 4000 user objects, as indicated by the reduction of 4317 to 56 in the data column are reclaimed again by the collector. Some of the system generated data objects persist longer. Note also that each of the user objects has been registered by the objects management system as indicated by the object column, and that these are also collected by the collector. As the system generated representation of processes (Stack Segments) is usually recycled internally, the reduction in the process column is due to the garbage collection of four user defined Condition objects.

Due to this test and several other test sessions including long-running tests with several garbage collections, we conclude that all garbage is collected properly. The conclusion is based on careful inspection of the objects remaining in the object store after a collection to ensure that it is exactly the right ones that survive. To broaden the picture, it must be noted that some objects cannot be collected due to the cooperation between the compiler and kernel. These are a priori known outside the kernel, and thus, their references go into the root set of any collection. The preservation of such objects may lead to a growing number of objects. This is, however, not a memory leak as the objects are promised to stay alive because the compiler may compile new user programs with references to these objects included. An independent file system garbage collector determines which of these references the compiler will ever use again by cleaning up the file system of compiled code and saved references. By interfacing this collector to our collector at run-time we expect to be able to diminish this problem even further.

Until now, it has not been a problem in practice. The extended storage picture given in Table 5.1 seems to verify this. To detect leakage and dangling references we have also implemented the copying conservative collector. The collector inspects all cells and detects all objects in the heap which either have no reference to them while alive, or have at least one reference to them while already reclaimed. This Memory Analyzer was used during the debugging to find leakage and dangling references in the entire heap which contains objects as well as kernel allocated data structures.

5.5.2 The Distributed Collection Overcomes Repeated Node Failures

The distributed collector works also when faced with node failures. It will always complete its collection although all the nodes are never available simultaneously. It is able to proceed on the available nodes and even the distributed termination detection will eventually succeed with few nodes available at a time. We have conducted several convincing experiments to be sure that both termination detection and remote shading is achieved as described. On this background, we conclude that the system works despite one or more node failures anytime during the garbage collection cycle. In fact, experiments show that the system is extremely robust to node failures. To show how robust our implementation is to node failures, a scenario where only three out of four nodes are available at any time during the distributed termination detection protocol has been constructed. The test example behaves as follows:

1. Our algorithm is running on a four node network.

2. A global collection proceeds on all four nodes until the copying has finished on all nodes; but the global state all nodes locally finished is not detected by any node yet.

3. Furthermore, node 1 has reached a state where it has told all the other that it has finished.

4. Then node 1 crashes, while the other three nodes continue collecting information about the global state until all three is ready to terminate if they get commitment from node 1 (which they do not get, as node 1 is down).

5. Then node 2 crashes and later node 1 recovers and re-enters the termination protocol.

6. Node 1 is able to confirm the last two node, that global termination is detected, but it cannot be confirmed itself as it need confirmation directly from node 2 which is currently down.

7. Node 3 and 4 terminates the collection and tells all other reachable nodes to do the same. Thus node 1 terminates the collection also.

8. When node 2 recovers, it will -by its first communication with any other node-be informed that the collection has terminated already.

This example has been simulated on the implementation by simulating the node crashes. As failure recovery in the current implementation of the Global garbage collection prototype has been disabled we have not been able to restart crashed nodes with check pointed objects. Instead the node crash and later recovery is simulated by stopping the process that execute the kernel and later resume it as a recovered node. To succeed in the real world, the termination detection mechanism must checkpoint its current state and restarts from that state when the node recovers.

During any other point of the garbage collection, the recovered node simply restarts the current collection. Other test examples are concerned with the distributed shading protocol. As each node continues its remote shading until all its needs have been serviced, it may tolerate that any of the other nodes are unavailable frequently often. As long as each pair of nodes that needs to exchange a remote shade request and reply has been available simultaneously to service the communication, the shading will progress and eventually terminate when the last request has been confirmed with a reply. Robustness to node failures is achieved not only by the global collection scheme, but also by the local collectors. The local collector survives failures of other nodes as it does not bother about them. If a node crashes while a local collector is running, the local collector may just be restarted after the recovery of the node itself.

5.5.3 Distributed Cycles of Garbage are collected

Many collectors, e.g., those based on reference counting; fail to collect cycles of garbage. There are "workarounds" to chase and identify if such cycles are garbage or not. In distributed system, the "workarounds" are further complicated, and many distributed collectors based on cooperating local collectors fail to collect distributed cycles of garbage.

Our implementation of the global collection scheme does, although it is based on a collector on each node, collect distributed cycles of garbage without any special effort. The implementation does essentially a global termination phase, and can thus guarantee that any cycle of garbage is collected. We have conducted several tests and observed how such distributed cycles are reclaimed by the first global collector running after they became garbage.

As one of these tests, we have constructed a set of objects which constitute a cycle of objects, where each references its two neighbors in the cycle (a cyclic double linked list). The cycle spans over more those three nodes. As long as at least one live reference exist from a live object to any of the objects in the cycle the entire cycle survives garbage collection. When the last reference is removed, the cycle still survives local garbage collection, as the elements in the cycle are potentially known from other nodes. The global collector does, however, remove such dead cycles without even knowing about cycle detection.

5.5.4 Termination Detection is always achieved

The protocol for remote shading is not only robust to nodes temporarily unavailable; it also ensures that remote shade and reply messages lost during communication would be recovered. Moreover, distributed termination detection of the global garbage collection cycle is always deferred as long as one single node has an outstanding shade request not acknowledged yet. We have observed the actual behavior of the running system and concluded that the implemented protocols work.

Thus the distributed termination detection is harassed and all other nodes in the system will try to reach a new agreement on global termination each time. The problem is partly circumvented in the current implementation by doubling the check for each cycle as described in the previous section. The problem could of course be further complicated by introducing further nodes and a spiral of references among the objects from node to node resulting in a linear list of objects crossing node boundaries each time. It degrades performance of the global collector in the sense that it takes longer to terminate it. Eventually, it terminates, and the outstanding work does not add substantial overhead nor does it prevent a local collection from being made on each of the nodes in between. Even a structure with 50 objects on each side of a node boundary, and thus a requirement of 100 additional shade request, does not add any visible delay to the termination protocol.

5.5.5 Performance Degradation Due to Garbage Collection

The overhead due to the added garbage collection has been measured by running various programs on our algorithm prototype instrumented with our garbage collectors and various counters holding statistics for each garbage collection cycle. The measurements include:

1. Micro-timings by logging time-stamps on entry and exit of various garbage collector segments of the algorithm prototype.
2. Counting instructions added for garbage collection purpose by code inspection of the C source implementing the algorithm prototype.
3. Macro-timings by wall-clock and user/system time, comparing a run with and a run without a garbage collector.

	Kernel without GC	Kernel with GC	GC overhead	
			Kbytes	Percentage
Kernel code	273 Kbytes	351 Kbytes	78 Kbytes	28.6%
Static allocated kernel data	386 Kbytes	433 Kbytes	47 Kbytes	12.2%
Initially heap size	267 Kbytes	313 Kbytes	46 Kbytes	17.2%
Total storage reservation	926 Kbytes	1,098 Kbytes	172 Kbytes	18.6%
Unused heap space	10 Kbytes	14 Kbytes	-4 Kbytes	-40%
Storage usage	916 Kbytes	1,083 Kbytes	167 Kbytes	18.2%

Table 5.5: Storage usage for the kernel at boot time

5.5.6 Garbage Collection of Processes

Besides the fact that some processes are also part of the root set, processes are handled like other objects. The processes are traversed as objects and marked as reachable as references to them are found. This result in orphan detection as a side-effect of garbage collection. Orphan processes are waiting forever in a queue that no live process is able to reach because the object which contains the queue is unreachable. Thus, neither the object nor the process will be marked as living, and both will be garbage collected as unreachable objects. We have tested orphan detection in a distributed system where the Condition queue could be known from another node. When all processes were queued, a local collection on each node removed most of the garbage, but the processes were still waiting on their queues, as these were in the local root set. Then a global collection was issued and it detected that the Condition and, thus the queue were garbage, and thus also the processes, that were only referenced from these queues.

5.6 Evaluation Summary

The implementation has been tested and it has been shown that our collection scheme does:

1. detects the termination of each global garbage collection cycle,

2. a comprehensive collection of garbage,

3. complete a global collection while nodes are unavailable,
4. collection of distributed cycles of garbage,
5. progress while nodes are unavailable,
6. introduce a limited overhead on user computation,
7. work concurrently with user computation with small latency,
8. collect orphan processes

As this work is mostly concerned with garbage detection, the reclamation part has only been mentioned briefly. This is an area where there has been left plenty of room for future optimizations. The fragmentation of the object storage indicates that more cooperation between allocator and collector is needed. One way to achieve this is to move all live objects to one area of the storage. In other words, a copying collector ensures a better compaction of live objects at the cost of more translation. It may be adequate to change the Global Incremental GC to be generational and use a copying collector for the young generation. The memory allocator currently in use runs a quick-fit strategy.

Chapter 6
Conclusion & Further Work

Chapter 6
Conclusion & Further Work

6.1 Conclusion

This work is a survey of garbage collection schemes for distributed systems and proposes an idea of a new global incremental garbage collector for distributed environments. We started by defining the problem of garbage collection and surveyed garbage collection schemes for uniprocessor systems. We presented a general criterion for evaluating garbage collection algorithms for uniprocessor, parallel systems and distributed systems. We surveyed different classes of garbage collection schemes for distributed systems and discussed their characteristics and associated overhead. We presented a generalization of Baker incremental GC scheme for distributed environments. We presented algorithms for the essential parts of the algorithm.

The thesis has presented a real-time scheme based on Baker's real-time scheme. The intention was to keep the advantage of Baker's scheme of not stopping normal program execution to perform garbage collection by solving the problem of detecting the termination of each garbage collection cycle. The performance of the presented scheme has been empirically analyzed and compared with the performance of a number of related schemes. The results indicate that our scheme would show interesting performance improvements over Baker's scheme for a class of programs that have considerable amount of long-lived data and execute much fewer general memory access operations than they do optimized ones. Functional programs in a language such as ML, Pure Lisp, Haskell, etc. are typical examples of such programs.

Also in this thesis, a complete survey is done about the different classes of distributed garbage collection algorithms, and how they differ from one another. Also, the advantages and disadvantages of each class and the different criteria that we measure everything on it. Through this survey we had reached my algorithm, as we had taken the weak points in Baker's and tried to solve some of them to improve the algorithm. Though we had implemented this algorithm, there is still so many works to be done further.

6.2 Further Work

According to our algorithm, we had reached some new aspects to survey in the world of garbage collection in distributed systems. This is an endless world of research, which we must search more and more. As for my algorithm it has one draw back that we must take care of, is that we didn't put any assumption for the order of messages.

Although this algorithm collects all garbage even cyclic ones but still we had to prove its correctness and completeness. This will be the target then, is that we are going to prove this algorithm and show its complexity. Also, we can do some more surveying and try to update many other algorithms my improving their weak points either in collecting all garbage or the speed of communication. We also plan to complete details of our proposed algorithm and validate its functionality and performance by simulation.

References

Abdullahi, S. E. (1992). Managing computer memory: Dynamic allocation and de-allocation strategies. In Proceedings of the 2nd Conference on Information Technology and its Applications. (Leicester, UK, Dec. 19-20), 25-40.

Leicester (1994). Recycling garbage. In Proceedings of the 3rd Conference on Information Technology and its Applications (Leicester, UK, April 2-3), 192-197.

Abdullahi, S. E (1995). Empirical studies of distributed garbage collection. Ph.D. thesis, Dec. 1995. Univ. of London.

Abdullahi, S. E. and Edemenang, E. J. A. (1993). A comparative study of dynamic memory management techniques. Advances in Model Analysis 15, 2, 17-31.

Abdullahi, S. E. and Ringwood, G. A. (1996). Empirical studies of distributed garbage collection, parts I, II, and III. TR, Dept. of Computer Science, QMW College, Univ. of London.

Abdullahi, S. E., Miranda, E. E., and Ringwood, G.A. (1992). Collection schemes for distributed garbage. In Proceedings of the International Workshop on Memory Management (St. Malo, France). LNCS 637, Springer-Verlag, 43-81.

Bengtsson, M. and Magnusson, B. (1990). Real-time compacting garbage collection. Position paper. In Proceedings of the ECOOP / OOPSLA '90 Workshop on Garbage Collection.

K.A.M. Ali. (1985). Garbage Collection Schemes for Distributed Storage Systems. Proceedings of Workshop on Implementation of Functional Languages, Aspenas, Sweden, February 1985, 422-428.

Saleh E. Abdullahi , Graem A. Ringwood. (1998). Garbage collecting the Internet: a survey of distributed garbage collection, ACM Computing Surveys (CSUR), 30, 3, 330-373.

Dijkstra, E.W. et al. (1978). On-the-fly Garbage Collection: An Exercise in Cooperation. Commun. ACM 21, 11, 966-975.

Hudak P. (1982). Object and Task Reclamation in Distributed Applicative Processing Systems. Ph. D thesis. Department of Computer Science, University of Utah.

P. Hudak and Keller, R.M. (1982). Garbage Collection and Task Deletion in Distributed Applicative Processing Systems," Proceedings of the 1982 ACM Symposium on LISP and Functional Programming, Carnegie-Mellon University, Pittsburgh, 168-178.

A. K. Nori, " A Storage Reclamation Scheme for Applicative Multiprocessor system," Master Th., Department of Computer Science, University of Utah, Dec.1979.

Almes, G., Borning, A., and Messinger, E. (1983). Implementing a Smalltalk-80 system on the Intel 432: A feasibility study. In Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, 175-187.

Amsaleg, L., Gruber, O., and Franklin, M. (1995). Efficient incremental garbage collection for workstation-server database systems. In Proceedings of the 21st International Conference on Very Large Data Bases (Zurich, Switzerland).

Gregory R., Andrews (1991). Concurrent programming: principles and practice, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA: Benjamin Cummings.

Appleby, K. et al. (1988). Garbarge collection for Prolog based on WAM, Communications of the ACM, 31 6, 719-741.

Arnborg, S. (1974). Optimal memory management in a system with garbage collection. BIT 14, 375-381.

Arvind , Robert A. Lannucci (1988) Two fundamental issues in multiprocessing, 4th International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering, 61-88, April 1988, Bonn, Germany .

Atkinson, M. P. (et al) (1983). An approach to persistent programming. Computer J. 26, 4, 360-365.

Augusteijn, L. (1987) Garbage collection in a distributed environment, Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe, 75-93, Eindhoven, The Netherlands.

Babaoğlu , Özalp and Marzullo, Keith (1993). Consistent global states of distributed systems: fundamental concepts and mechanisms, Distributed systems (2nd Ed.). New York, NY: Addison-Wesley.

Baden, S.B. (1983). Low-overhead storage reclamation in the Smalltalk-80 virtual machine. In Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, 331-342.

Baecker, H.D. (1972) Garbage collection for virtual memory computer systems, Communications of the ACM, 15, 11, 981-986.

Baker Jr., Henry J. (1978). List processing in real time on a serial computer, Communications of the ACM, 21, 4, 280-294.

Baker, Henry G. (1992).The treadmill: real-time garbage collection without motion sickness, ACM SIGPLAN Notices, 27, 3, 66-70.

Bal, Henri E. (1990) Programming distributed systems, Silicon Press, Summit, NJ, 1990

Ballard, S and Shirron, S. (1983). The design and implementation of VAX/Smalltalk-80. In Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, 127-150.

Barach, D. R., Taenzer, D. H., and Wells, R. E. (1982). A technique for finding storage allocation errors in C-language programs. ACM SIGPLAN Not. 17, 5, 16-23.

Liskov, Barbara and Ladin, Rivka (1986). Highly available distributed services and fault-tolerant distributed garbage collection, Proceedings of the fifth annual ACM symposium on Principles of distributed computing, 29-39, August 11-13, 1986, Calgary, Alberta, Canada

Bartlett, G.F. (1990). A generational, compacting garbage collector for C + +. Position paper, ECOOP/OOPSLA '90 Workshop on Garbage Collection.

Raymond L. Bates , Dyer, David and Koomen, A. G. M. (1982) Implementation of Interlisp on the VAX, Proceedings of the 1982 ACM symposium on LISP and functional programming,.81-87, August 15-18, 1982, Pittsburgh, Pennsylvania, United States

Bekkers, Y. and Cohen, J. (1992). General discussions. In Proceedings of the International Workshop on Memory Management (St. Malo, France). LNCS 637, Springer-Verlag.

Bekkers, Y., Ridoux, O., and Ungaro, L. (1992). Dynamic memory management for sequential logic programming languages. In Proceedings of the International Workshop on Memory Management (St. Malo, France). LNCS 637, Springer-Verlag, 82-102.

Ben-Ari, Mordachai (1984). Algorithms for on-the-fly garbage collection, ACM Transactions on Programming Languages and Systems (TOPLAS), 6, 3, 333-344.

Bennett, John K. (1987). The design and implementation of distributed Smalltalk, ACM SIGPLAN Notices,.22, 12, 318-330.

Bengtsson, M. and Magnusson, B. (1990). Realtime compacting garbage collection. Position paper. In Proceedings of the ECOOP / OOPSLA '90 Workshop on Garbage Collection.

Bevan, D. I. (1987) Distributed garbage collection using reference counting, Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe, 176-187. Eindhoven, The Netherlands.

Black, Andrew et al. (1987) Distributed and abstract types in Emerald, IEEE Transactions on Software Engineering, 13, 1, 65-76.

Birrell, A. et al. (1993). Distributed garbage collection for network objects. TR 116, Digital Equipment Corp. Research Center.

BISHOP, B. 1977. Computer systems with very large address space and garbage collection, Ph.D. thesis, MIT, Cambridge, MA.

Daniel G. Bobrow (1980). Managing Reentrant Structures Using Reference Counts, ACM Transactions on Programming Languages and Systems (TOPLAS), 2, 3, 269-273.

Boehm, Hans-Juergen and Weiser, Mark (1988). Garbage collection in an uncooperative environment, Software—Practice & Experience, 18, 9, 807-820.

Brooks, Rodney A., Gabriel, Richard and Steele Jr., Guy L. (1982) S-1 Common Lisp implementation, Proceedings of the 1982 ACM symposium on LISP and functional programming, p.108-113, August 15-18, 1982, Pittsburgh, Pennsylvania, United States

Brownbridge, D. R. (1985) Cyclic reference counting for combinatory machines, Proc. of a conference on Functional programming languages and computer architecture, p.273-288, January 1985, Nancy, France.

Carlsson, S., Mattisson, C., and Bengtsson M. (1990). A fast expected-time compacting garbage collection algorithm. Position paper. In Proceedings of the ECOOP / OOPSLA '90 Workshop on Garbage Collection.

Chambers, C., Ungar, D. and Lee E. (1989). An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes, ACM SIGPLAN Notices, 24, 10, 49-70.

Chambers, Fred B., Duce, David A. and Jones, Gillian P. (1984). Orlando, Fl.: Academic Press, Inc.

Chenney, C. J. (1970) A non-recursive list compacting algorithm, Communications of the ACM, 13, 11, 677-678.

Chikayama, T. and Kimura Y. (1987) Multiple reference management. In Flat GHC, ICLP, MIT Press, 276-293.

Clark, Douglas W., and Green, C. Cordell. (1977) An empirical study of list structure in Lisp, Communications of the ACM, 20, 2, 78-87.

Clark, D. W. and Green, C. C. (1977) A note on shared list structure in Lisp. Inf. Process. Lett. 7, 6, 312-314.

Cohen, Jacques. (1981) Garbage Collection of Linked Data Structures, ACM Computing Surveys (CSUR), 13, 3, 341-367.

Nicolau, and Alexandru (1983). Comparison of Compacting Algorithms for Garbage Collection, ACM Transactions on Programming Languages and Systems (TOPLAS), 5, 4,.532-553.

Baker and Trilling (1967). Remarks on garbage collection using a two level storage. BIT 7, 1, 22-30.

Collins, George E. (1960). A method for overlapping and erasure of lists, Communications of the ACM, 3, 12, 655-657.

Coulouris, George F. and Dollimore, Jean (1988). Distributed systems: concepts and design. Boston, Ma.: Addison-Wesley Longman.

Courts, Robert (1988). Improving locality of reference in a garbage-collecting memory management system, Communications of the ACM, 31,9, 1128-1138.