Comprehensive, Concurrent, and Robust Garbage Collection in the Distributed, Object-Based System Emerald

Niels Christian Juul

Department of Computer Science
University of Copenhagen
Denmark

DIKU-rapport 93/1

February 1993

Comprehensive, Concurrent, and Robust Garbage Collection in the Distributed, Object-Based System Emerald
A Ph.D. Thesis submitted to the University of Copenhagen by:
Niels Christian Juul
Published by DIKU February 1993 as Technical Report (blue series):
DIKU-rapport 93/1.
Typeset using LaTeX and other UNIX tools.
Printed on QMS-PS-810 *turbo* POSTSCRIPT printer.
Bibliography made by BibTeX.
Illustration made by `idraw`.

D.I.K.U. / Dist. Lab.
Laboratory for Distributed Systems and Applications
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK 2100 Copenhagen Ø
Denmark

Phone: +45 35 32 14 16
Fax: +45 35 32 14 01
E-mail: ncjuul@diku.dk

Abstract

Garbage collection schemes are investigated to provide a solution well-suited for the distributed object-based system, Emerald. A distributed garbage collection scheme is implemented. It collects *all* garbage (comprehensive), works while the system is running user applications (concurrent), and copes with partial failures due to the distributed nature of the system (robustness). The collection scheme reaches its goals by using distributed control, a dual-collector scheme, and an object protection mechanism.

Emerald is a distributed, object-based system running on a set of workstations connected by a local-area network. The Emerald garbage collection scheme employs two *mark-and-sweep* collectors on each node in the distributed system. To reduce the latency introduced, both collectors work concurrently with other processes by protecting unmarked objects during the mark-phase, and interleaving the sweep-phase with the allocator.

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 mark-phase and synchronize their initialization and termination of the mark-phase, without using a coordinating node. 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 the mark-phase. 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.

The scheme has been tested in the Emerald system running on four VAXstation 2000 workstations in a local-area network. Measurements show a garbage collection overhead during normal operation on less than 10%. The average pause introduced by garbage collection may be limit to 0.01 seconds.

Contents

List of Figures

List of Tables

Acknowledgment

The use of first person plurals through out the thesis reflect the fact that research is not done in a vacuum by a single person. This work is done by the author, but the influences from and fruitful discussions with other persons have had a large impact on the result.

The work has been carried out during my employment at DIKU, Department of Computer Science, University of Copenhagen. DIKU has supported the work by a 30 month Ph.D. Grant (Kandidatstipendium) and a 12 month extension. Furthermore, this work has been supported by FTU-grant no. 5.17.5.1.35 from the Minister of Research and Education, and two travel-grants J.nr. 90.3.3 and 92.4.16 from the School of Natural Science, University of Copenhagen,

DistLab at DIKU has been established during this period and it has together with its individual members been an encouraging environment for this work. As thesis supervisor, Lektor Eric Jul deserves the right to be credit first. His insight and ability to see through the surface of the ideas presented to him has killed many ill-founded proposals and inspired the ideas worth working on. Birger Andersen and Bjarne Steensgaard have commented on this and previous work and have thus influenced the result.

The numerous services made available to me from the department and its individual members also include computing equipment, software, hardware, hardware repair (this thesis has survived two serious disk crashes), software installation and maintenance from the computing staff members, and secretary service from the department secretaries.

Thanks goes to the world-wide garbage collection community at-large and the participants in the GC-Workshop at ECOOP-OOPSLA'90 and IWMM'92 especially. It has always been an stimulating and challenging background for my work to participate in this community. Special thanks goes to Henry Baker, Elliot Kolodner, and David Detlefs for taking time to comment on my interpretation of their own as well as other algorithms for garbage collection.

Without the support and encouragement from my family, especially my wife Anne, and children Pernille and Mie, this work had not been finished. Hopefully, I will be able to pay back in the future.

Chapter 1

Introduction

The subject of this thesis is garbage collection in distributed, object-based systems. We have designed, implemented, and measured a distributed garbage collector in an existing distributed, object-based system, Emerald. 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. Part of the work contained in this thesis was presented at the First International Workshop on Memory Management [Juul 92].

Many distributed systems are being built these years using the object-based programming paradigm [Bal 89, Chin 91]. In general, the objects in such systems are *dynamically created*, and *persistent*, i.e., conceptually they live forever. An implementation of such systems must cope with a monotonically growing number of objects. Some of these objects may, however, be considered garbage if they are not reachable by references from the active part of the system. Garbage objects can be removed from the system, as they will never be able to influence any future outcome of the system.

The implementation maps objects onto a finite storage. To continuously service the need for more storage space demanded by the creation of new objects, a garbage collector must identify the garbage objects and make their storage space available to the storage allocation routine. Thus the garbage collector has two obligations; (1) it does *garbage detection*, and (2) it does *storage reclamation*, i.e., *recycling* the garbage. This thesis covers garbage detection, whereas storage reclamation is only superficially addressed.

Our solution is based on the well-known mark-and-sweep garbage collection technique [Knuth 68] extended with an object protection and faulting technique to enable concurrency with user processes (called *mutators*). 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]).

Garbage collectors for distributed systems are characterized by supporting one or more of the following features:

Concurrency	The collector and mutators 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.

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 above mentioned features. Such collectors supports one or more of the additional features:

Concurrency & Robustness	By enabling mutators 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.

In general, the non-comprehensive collectors are able to collect more efficiently, while being both expedient and robust to node-failures. The solution described by Shapiro to be implemented in project SOR [Shapiro 90, Shapiro 91] is robust but may fail to collect all garbage. Instead, it has the potential for being expedient. Lang, Queinnec, and Piquer further refine the scheme based on independent node collectors to cooperate for various groups of nodes. The scheme is expected [Lang 92] to be comprehensive eventually, if each distributed (interconnected) graph of garbage objects is contained in a group of nodes, that does not fail during the group collection.

The collector for Galileo [Mancini 91] is implemented as a global stop-and-copy collection of objects on non-volatile storage. If nodes become unavailable a fault-tolerant adaption enables a non-comprehensive collection to complete. The scheme may limit the collection to a subset of nodes and thus only blocking mutators on those nodes. A comprehensive collection is, however, not guaranteed.

A comprehensive collection is often more costly and assumes a simple failure model or no failures at all. The collection scheme for the POOL system [Augusteijn 87] is based on global synchronization of node-local mark-and-sweep collectors, and to be comprehensive it depends on complete node availability. The garbage collector for POOL, as described in [Augusteijn 87], does not cope with failures in the distributed system.

The idea of node-local collectors that cooperate has also been proposed by Liskov and Ladin [Liskov 86]. The cooperation is made possible by introducing a logically centralized, but physically replicated, highly available service for inter-node references. Any such reference is registered at the service, which also does inter-node garbage cycle detection. The service is based on synchronized local clocks and bounded delays of inter-node message exchanges.

Many other schemes for distributed garbage collection with these features has been published. We have, however, not been able to identify an implemented system featuring concurrency, comprehensiveness, and robustness in one system.

This thesis considers the design and implementation of such a garbage detection in a distributed system. Our solution combines the comprehensiveness of mark-and-sweep 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.

The rest of this chapter is organized as follows. First the background on distributed, object-based systems is given and we present our formulation of the garbage collection problem. We then describe the garbage collection problem as a graph problem, and discuss some of the main obstacles faced by an implementation of garbage collection in distributed, object-based systems. A background on the Emerald system is given and then the chapter continues with a description of our goals, design criteria, and the work done. To measure the outcome of the work, a set of evaluation criteria and techniques are presented. We conclude the chapter by summarizing our research contributions.

1.1 Background

Many recent distributed systems have been developed with *objects* as their main structuring concept. *Distributed, object-based systems* take advantage of the object-based programming model to support distribution. The main advantage is that each object may be in a separate protection domain, thus it is self-contained and easily moved.

The common background, on which garbage collection is investigated in this thesis, is distributed, object-based systems. The subject of the thesis lies in the intersection of object-based programming, distributed systems, and recycling of object resources. The characteristics of *object-based* programming and *distributed* computer systems are given here as general background for the rest of thesis.

1.1.1 Object-Based Programming

A major trend in programming during the last ten years has been the use of the object-oriented model. The important and unifying concept in the systems discussed here is that all data is conceptually represented as *objects*. Resources in object-based systems are often visible to the user as objects. In full object-based systems, everything is modeled as objects at the language level independent of the underlying implementation.

Objects encapsulate their data and provide a clean interface. In principle, the internal representation of data may only be accessed directly by operations inside the object. Objects may know about other objects by having *references* to those objects as part of their data. References may be exchanged among objects, i.e., they may be copied and deleted. The *lifetime* of an object is not necessarily linked to the lifetime of the program creating the object and the object store may persist longer than the individual programs. Conceptually, objects live forever in an object-based system. In practice, the space taken by unreachable objects must be reclaimed to make room for new ones.

1.1.2 Distributed Systems

Although computers are getting faster and faster, applications and their needs for computational power are growing even faster both in problem size, computational complexity, and number of problems. The computational power may be achieved by faster computers, using specialized processing units (pipe-lined, floating-point accelerators, etc.), multiprocessors (tightly coupled processors with access to a common bus and memory), or a large number of inter-connected computers (distributed system). In a distributed system computers are connected to cooperate on solving large computational problems. The building blocks are traditional computers, e.g., workstations connected in networks. In our model, the distributed computer system consists of a local-area network with cooperating nodes where each *node* is an autonomous computers equipped with its own storage and programs.

Distributed systems may be characterized by the availability of individual nodes; each node playing a role of its own, and a role as part of the whole system. The most important challenges to garbage collection in these systems are that they feature:

Multiple processors Thus true concurrency is available for both mutators and collector.

Multiple independent storage units Thus storage is addressed as either local (same node) or global (on any other node).

Partial failures Thus from one node, some other nodes may be temporarily unavailable.

1.1.3 Distributed, Object-Based Systems

The object-based model is well suited to structure a distributed system. Objects encapsulate their data and provides a clean interface. Each object may be viewed as a separate protection domain. This makes objects a natural entity for distribution, i.e., objects may be moved around and accessed across node boundaries.

The implementation of a garbage collector in this environment cannot be achieved by traditional garbage collection techniques alone due to the characteristics of these systems. As an example, a garbage collector in the distributed, object-based system Emerald is challenged by the following characteristics of objects in Emerald mainly introduced by distribution:

Migration Objects may move around among the nodes.

Replication Certain objects are replicated at different nodes.

Recovery Objects may be checkpointed to a set of nodes and on non-volatile storage at these nodes. After a node failure such objects may be recovered, thus reinstantiating them with their check-pointed state.

Unavailability References may exist to objects whose representation has been temporarily or permanently lost due to node crashes.

1.2 Garbage Collection Terminology

The terminology used when discussing garbage collection is based on the intuitive model of objects as entities. The entities are related by the references from one object to another. The central vocabulary used through out this thesis is presented in this section. Additional terms will be defined during the thesis, as they are presented.

The objects are interconnected by references going from one object to another. On the user level, they are classified as either *reachable* or *garbage* depending on whether the user can take advantage of them or not. They are always available (all objects are persistent) though only the reachable objects will ever be useful. We define *the root set* of objects as the objects representing user processes and some system-dependent "always useful" objects, e.g., the objects representing i/o channels. The objects in the root set are called root objects.

The transitive closure of the root set using references as the relation between objects, defines the *reachable objects*. All other objects are garbage. These garbage objects may be implemented using zero bit, i.e., the storage that they occupy may be reclaimed. Thus, the name *garbage collection* for the process that collects these objects.

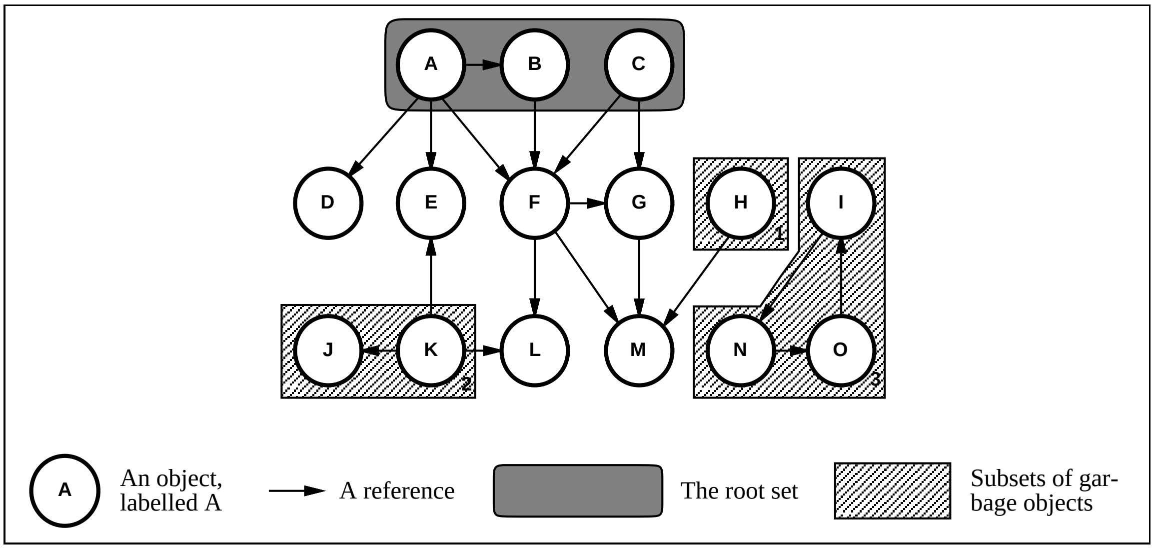


Figure 1.1: A system containing 15 objects and their references

Example: A Graph of Objects.

Figure 1.1 illustrates a graph of objects, which is used as an example through out the thesis. The graph consists of 15 objects, labeled `A`, `B`, ..., `O`, and 18 references between them. In the example, object `A`, `B`, and `C` are the root objects, and three subgraphs (shaded and labeled 1, 2, and 3) contain garbage objects only.

A Garbage collector does two tasks:

Garbage detection Identifies the garbage objects, e.g., the three shaded subsets in Figure 1.1.

Storage reclamation Recycle the storage occupied by the detected garbage objects.

There are two distinct techniques for garbage collection. Algorithms to detect garbage either associate a *reference count* for each object that is updated each time a reference is updated, or traverse the reachable objects to identify the transitive closure of the root set. Traversing algorithms either evacuate the reachable objects during the traversal (*copying*) or mark the live objects so that a subsequent search of all objects can identify and reclaim the dead objects, i.e., those not marked. This exhaustive search is called a *sweep*, and the scheme is called *mark-and-sweep*. The subsequent sweep may alternatively evacuate the marked objects instead of reclaiming the unmarked.

For any given traversing garbage collector, we define a *garbage collection cycle*, as the work done during one full execution of the collector. That is, any initialization necessary, the identification of garbage, and the reclamation of the identified garbage. A system based on a traversing collector must execute the collector repeatedly, i.e., one garbage collection cycle after the other. Whereas a system using reference counting does the work incrementally while the system is running.

The algorithm may work on the total object space or divide the storage in *areas*. If the partitioning is complete, i.e., no objects contains references across area boundaries, each of the areas may employ their own independent garbage detection algorithm. If an area (somehow) knows about incoming references, it may still run its own garbage detection although non-garbage must not be moved unless the objects where the incoming references originate is informed about the move.

We classify algorithms, that reclaim all garbage as *comprehensive*, in contrast to *partial* collectors, that only reclaim some garbage. A *conservative* collector is a collector which may identify some garbage as non-garbage. Because area collectors often use a conservative estimate of the root set, they are usually classified as partial collectors.

1.3 Garbage Collection as a Graph Problem

We now present garbage collection as a graph problem and use this formalism to describe some of the main obstacles encountered when implementing garbage collection.

1.3.1 The Garbage Collection Problem

As illustrated in Figure 1.1, an object-based system may be viewed as a directed graph of objects connected by references. The objects are vertices and each reference from one object to another is an arc in the graph. The purpose of garbage detection is to identify sub-graphs without any incoming arc from the live part of the graph. The live part is the reachable objects recursively defined as those objects that have at least one incoming arc originating in an already live object, starting with the root set as the live objects. By using the arcs as the relation between the objects, the transitive closure of the root objects identifies the reachable objects, and the complement to the transitive closure of the root set identifies the garbage.

An implementation of object-based systems must be able to identify the non-reachable part of the graph to reclaim the resources occupied by those objects for recycling. Our implementation is concerned with efficient garbage collection of these objects by marking the graph of reachable objects (doing the transitive closure of the root set), and then sweeping through all objects to reclaim the unmarked (the complement).

Consider the objects shown in Figure 1.1. The objects `A`, `B`, `C`, `D`, `E`, `F`, `G`, `L`, and `M` are all reachable, thus leaving the rest, i.e., the sub-graphs labeled `1`, `2`, and `3` as garbage.

1.3.2 Complication caused by Distribution

In distributed, object-based systems objects are distributed among the nodes in the system. The graph of objects and references spans several nodes, and sub-graphs of reachable or unreachable objects may also cross node boundaries. Figure 1.2 illustrates our example graph of Figure 1.1 for a distributed system of 6 nodes.

Inter-node references are often represented by two sets (or tables) on each node. *The incoming set* contains incoming references to objects on the node, i.e., objects on this node pointed at by an arc from an object on another node, e.g., `O` on Node `6` (Figure 1.2). *The outgoing set* contains the references from objects on the node to objects on the other nodes, i.e., objects on other nodes pointed at by an arc from an object on this node, e.g., `N` on Node `6` (Figure 1.2). The two sets may be augmented to work as translation tables between a node-local and a system-wide representation of object references in systems with a two

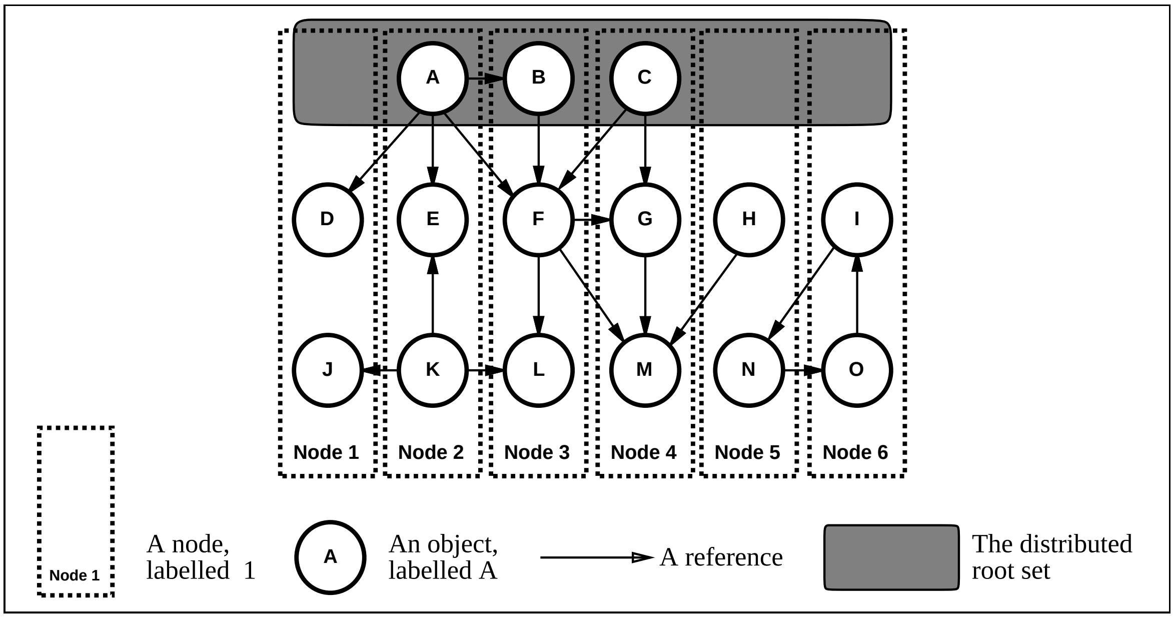


Figure 1.2: An object graph distributed on 6 nodes

level addressing mechanism. We classify the garbage objects on a node as either *distributed garbage* if they are known in the incoming set, or *local garbage* if they are not.

In distributed systems each node is an area candidate, thus a natural distribution of the garbage collection work is a local *collector* per node, i.e., an *area collector*. Thus it is possible to run a local area collector that collects all garbage on the node except distributed garbage. For this purpose, the root set of the local collector must be extended with the objects referenced from other nodes, i.e., the incoming set.

If each of the six nodes on Figure 1.2 employs a local garbage collector, the local root set is extended with the local incoming set. Thus the local root sets of the nodes on Figure 1.2 become: `1(D,J)`, `2(A)`, `3(B,F,L)`, `4(C,G,M)`, `5(N)`, and `6(O)`. Running a local collector in each node will collect object `H` and `K` only.

To use this technique we have to determine how and when inter-node references should be registered and deregistered in one or both of the two sets. The comprehensive collection may be accomplished by a mechanism that adds this service to the local collectors or by a distributed collector, that works on the total distributed object space.

1.3.3 System Requirements

A mark-and-sweep garbage collector needs to know its root set and must be able to mark these objects and the objects identified by following the arcs from the objects already marked. Finally it must be able to sweep the object storage and reclaim the unmarked objects.
This requires that the system using the collector cooperatively:

- identifies the objects in the root set(exactly),
- for each object, identifies the references to other objects(exactly), and
- allows an exhaustive search through the object storage.

In distributed systems this includes the ability to follow the references across node boundaries and to cope with node failures. Furthermore, it would be very inconvenient for the user of a distributed system if the collection may stop the system, i.e., pausing user computation while the collector is running. Thus the collector must be able to traverse the object graph consistently although the graph may be changing concurrently.

Some of the main obstacles due to concurrency and distribution are discussed more detailed in the following sections.

1.3.4 Concurrent Collection

The annoying pauses in user computation, while a garbage collector is running and blocking the user processes, are often raised as a general argument against automatic garbage collection by the system. To limit this latency the collection may be interleaved with user activity. Such collectors are known as *on-the-fly* collectors or *concurrent* collectors. The process of detecting and reclaiming garbage without pausing user activity is important to gain *real-time* behavior and thus meet the critiques. An *incremental* garbage collector does this either by doing a partial collection at a time, e.g., area collection of smaller areas, or by interleaving the garbage detection with the user processes. As an example, interleaving is accomplished on a very fine grain by reference counting, where the garbage detection is done for each reference update.

The obstacle, when introducing concurrency between mutators and collector, is that the collector is faced with a dynamically changing object graph. On the abstract level this obstacle is overcome by specifying a set of invariants that must be respected and preserved by all processes.

1.3.5 Distributed Collection

As mentioned, distribution introduces several challenges to garbage collection:

Cross node reference graph
: The sub-graph spanning all reachable objects may cross node boundaries several times. Thus the algorithm must work in a distributed fashion, i.e., on all nodes in the distributed system.

Partial failures
: Distributed systems may be partitioned by break-downs of nodes or communication links. If each node and link are either working or failed in a fail-stop manner, i.e., no Byzantine failure, the garbage detection should be *robust* to such behavior. To cope with these failures, we demand our algorithm to be able to complete as long as all nodes of the system are at-least pair-wise available (and able to exchange messages) sufficiently often.

Recovered and lost objects
: After a node failure, some objects may be recovered based on a checkpointed (older) version of their state while others may be lost forever.

Older references in recovered objects may suddenly become alive again, thus, a traversing algorithm must also traverse the checkpointed copies of live objects to prevent objects referenced only in those copies from being deleted. The same yields for collection based on reference counting. Checkpointed copies must be managed as real objects in garbage collection contexts.

Lost objects may leave references to them dangling in the object graph, thus the garbage collector must be able to detect that a reference is dangling. For example, if node 1 in Figure 1.2 fails and looses the objects `D` and `J` permanently, the arcs from `A` to `D` and `K` to `J` represent dangling references.

Long-term unavailable nodes

A temporarily failed node may not recover immediately, thus preventing a comprehensive detector from terminating. Meanwhile garbage may fill-up the non-failed part of the system. This makes it important to be able to reclaim some garbage while the comprehensive garbage detection waits for a failed node to be available again. The solution may be to introduce non-comprehensive independent area-collectors that run concurrently to the distributed collector and reclaim the area-local garbage. More than one concurrent collector does, however, introduce the need to synchronize their behavior without long term blocking of any of them.

1.3.6 Towards a Distributed Garbage Collection Scheme

Based on the proposals for garbage collection in [Jul 87, Jul 88a, Jul 88b], we design and implement a garbage collector that does a *comprehensive detection* in a *distributed*, object-based system, *robust to failures* in communication and partial machine availability. Supplementary, we implement a *node-local* area collector that collects local garbage, i.e., objects which have never been known by objects outside the node. This collector is not comprehensive as it may contain global garbage in its root set. Both garbage collectors run concurrently with the weakest possible synchronization. In both cases, concurrency with mutators is obtained by separating the object store in two; one for mutators and one for the collector, and by using a garbage collection fault to trap from a mutator to move the an object from the garbage collector domain into the mutator domain before the mutator access the object.

1.4 The Emerald system

The Emerald system is used as the general testbed for our garbage collection algorithms. Emerald is a distributed object-based system with support of *objective expressiveness*, *concurrent execution*, and *object mobility*. The Emerald system is an implementation of the Emerald programming language [Hutchinson 87b, Raj 91] by means of the Emerald compiler [Hutchinson 87a] and run-time system [Jul 88a].

This section presents a brief overview of the system. A detailed description may be found in [Hutchinson 87a, Jul 88a]. The features interesting from a garbage collection implementation point-of-view is described in Appendix A.

1.4.1 Emerald System Overview

The Emerald system is distributed, i.e., the run-time system supports the execution of Emerald programs across a network of node computers. The system consists of nodes fully interconnected via a network (see Figure 1.3). The run-time system supports inter-node communication by the cooperation of the Emerald kernels running on each node.

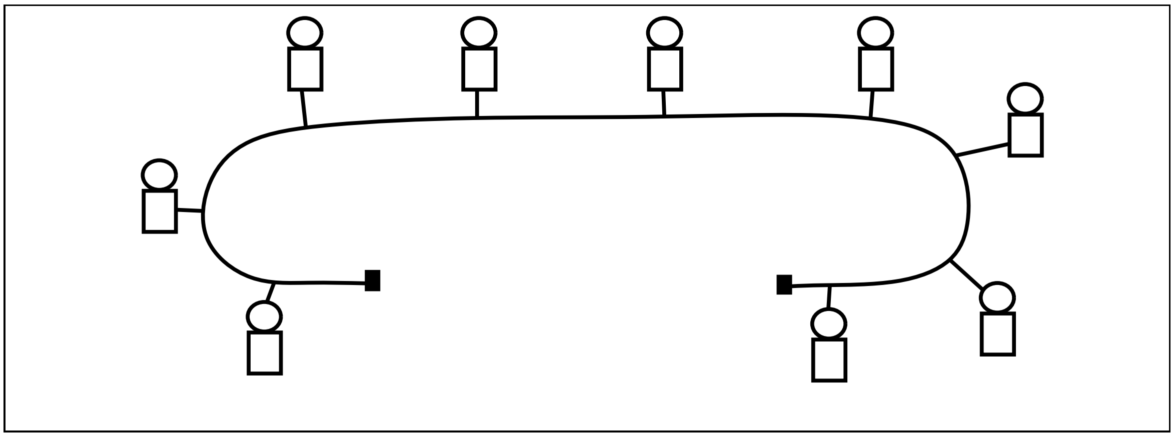


Figure 1.3: The network of Emerald nodes

A distributed Emerald system runs on a set of workstations interconnected by a local-area network of the Ethernet type using Internet protocols for communication. The Emerald tools, the compiler and kernel, are implemented on top of UNIX. Due to the lack of a dedicated programming development environment, Emerald programs are developed using standard UNIX tools. The Emerald compiler delivers native machine code to be executed by the run-time system. When the code of a compiled program is submitted to the run-time system on a node, the Emerald kernel on that node takes over. The kernel runs in a UNIX process and loads the submitted code into the address space of this process. The code may include calls of, and traps to, the run-time services provided by the Emerald kernel, e.g., input/output, cross-node communication, synchronization, and run-time type checking.

Our Emerald prototype runs on a set of VAX-, SPARC-, or Motorola 68000-based computers connected by a local-area network. The implementation runs on top of various versions of the UNIX operating system, all 4.3BSD compatible. The local-area network is a thin-Ethernet on which the nodes communicate via *sockets* (4.2BSD UNIX) using the UDP protocol.

Each computer runs a process, the Emerald kernel, which emulates the concept of a node. A node is either down or up and running. A running node is fully connected with the other running nodes. Thus, the network is never partitioned. Still a node may restart with part of its old state recovered after a failure. Whether to recover an object or not is, however, a global decision as the checkpoint used for recovery of the object may be duplicated on many nodes and/or their stable (backing-) storage.

1.4.2 A Vocabulary of some Emerald Implementation Concepts

The Emerald implementation uses a specific vocabulary for many of the Emerald concepts. Some of these are also used in general through out the thesis. Their definition in the Emerald context is given here.

Node is the Emerald concept of a computer, i.e., a computing entity with its own storage and optional stable storage unit and input/output-system.

Local/global describes the potential distribution of accessibility to an entity in Emerald, i.e., whether the entity may be known node-local only or network-wide.

Resident/non-resident describes whether a given entity in Emerald is locally present or not. A global object in Emerald may—but need not be—resident on the node it is used from.

Immutable denotes objects with a constant state, i.e., an object that never changes its instance variables (they are all constants).

Replication describes the ability to maintain multiple copies of the same object on different nodes. Immutable objects may be replicated without consistency problems in a distributed system as all replica contain the same information all the time.

1.4.3 The Emerald Language

Emerald is an imperative, strongly-typed, object-based programming language [Black 86, Black 87, Jul 88b]. It supports *a unified object model* independent of object size, use, and distribution, with a special emphasis on *distribution* and *concurrency.* Objects are highly mobile and distribution is made transparent to other facilities.

An object in Emerald is defined as a named entity with:

- an encapsulated data structure, i.e., references to objects,
- an exported list of operations,
- the implementation of internal and exported operations,
- an initial section, and
- an optional process.

An object may also contain special sections to handle failure-like exceptions, e.g., if another object, invoked[1] by this object, is *unavailable* or has *failed.*

When an object is created, the initial section is executed before any operation can be invoked and the optional process started. Only exported operation can be invoked from other objects. Data and operations may be protected by a *monitor* providing mutual exclusion on this part of the object.

Objects may be moved between nodes in the network. Explicit move statements and call-by-move for invocation parameters are part of the Emerald programming language. Objects may be classified as *immutable* by the programmer, if they do not change their state over time. Such objects will be copied when asked to move in the current implementation as copying in this case does not introduce any inconsistency. These copies are named replica.

Furthermore, the language contains a rich variety of constructs to ease distributed object-based programming. The main construct is the *object invocation.* An object may invoke an operation on another object, if it is able to address the other object. Invocations may take parameters on both call and return, and looks like conventional procedure calls. Object

[1] Language purists will find our description a bit sloppy. An invocation, e.g., is done by invoking an operation or function in an object which exports that operation or function. Still we will simply write *invoke an object* instead of the longer *invoke an operation in an object.* Object-oriented programmers may read this as *sending a message,* operation, *to an object.* In all cases, the underlying semantics is essentially an ordinary procedure call.

invocations are location transparent. If an object invokes a non-resident object, the call is said to be a *remote invocation* and look like a remote procedure call.

1.4.4 Why Emerald needs Garbage Collection

Conceptually, after it has been created, an Emerald object persists forever. In cases where no other object in the system has a reference to the object, the resources occupied by the implementation of the object, e.g., the storage used to store its state, may be reused for other purposes. Thus unreachable objects must be identified and their resources recycled. As the Emerald language level abstracts away the possibility to do explicit storage management at the user level, the system must do automatic garbage collection. This may be done either by a compiler inserted recycling call, if the compiler knows where the last reference is deleted or by a garbage collector for objects with a more dynamic (not compile-time determinable) lifetime.

The Emerald compiler makes a great effort in reducing the amount of dynamically created objects that must be garbage collected. Still, many objects are created dynamically with unpredictable lifetime. It is the obligation of the Emerald garbage collector to identify and reclaim the unreachable of these objects. To do so the garbage collector must know about the language concepts of an object, a reference, and a process.

The references inside the data area of an Emerald object point at other objects. As mentioned in Section 1.2, all the objects and their references together constitute an object graph. Starting from a root set of objects, the garbage collector must identify the reachable objects and reclaim the rest.

In the Emerald system, the root set is made of some "always useful" objects and the objects containing an active process. The first kind is implementation dependent and given by the run-time system itself. The active processes are those unblocked processes that are ready to run. In Emerald, each process originates in the process section of an object from where it spawn its thread-of-control through other objects following its actual sequence of nested invocations. Thus, each process is represented by a chain of activation records. A blocked process is non-garbage, if an active process knows the object on which the blocked process is blocked, and thus has the potential of unblocking it.

1.5 Goals

The primary goal is to present a working implementation of a comprehensive, concurrent and robust garbage collector for the Emerald system.

The requirements to functionality and performance are presented below. Though separated in presentation these are not separated when designing the solution. Most emphasis is placed on functionality.

1.5.1 Functional Requirements

A Comprehensive Garbage Collector

Reclamation of garbage in the distributed system must be *comprehensive*. By this we demand that all garbage will eventually be collected and nothing else. Our collector will collect everything which were garbage when the collector was started, but it depends on the availability of all nodes in the distributed system.

A Concurrent Garbage Collector

The pauses introduced by the garbage collector must be kept small, so the garbage collection is done *on-the-fly*, i.e., while user activity progresses concurrently.

Robustness

Distributed systems has the ability to fall apart, i.e., partial failures may occur. Our collector scheme must be *robust* to such failures. The collection progresses during partial failures and may finish a comprehensive collection even though the whole system has never been available concurrently. The robust collector will succeed, if the participating nodes pair-wise have been available often enough.

It must also preserve checkpointed copies of live objects and objects reachable from these, to enable smooth recovery of checkpointed objects after failure.

A Node-Local Collection

In general, a simple job should be kept simple, and a local job should be done locally. By defining an area collector for each node in the distributed system, we are able to conservatively collect local garbage, i.e., garbage, that has been known outside the node, is considered alive, as we do not know locally whether a reference exist somewhere outside the node. Thus an area collector should run independently on each node in the distributed system. This further serves the purpose of being able to collect garbage when the comprehensive global collector is blocked, i.e., waiting for a failed node to recover.

1.5.2 Performance Goals

Although performance is a secondary concern in this work, the collector need not be inefficient. At least, the collector must keep pace with the allocating application, i.e., reclaiming objects soon after they become garbage and not later than when the space is needed for new allocations. Another concern is to limit the overhead by garbage collection, i.e., the consumption of time and storage due to the collector, should be kept small both in the short (instantaneously) and long run (overall performance).

The performance goals for our collector are:

- to be *efficient* (low total cost), and
- to work with *low latency* (introducing only very small pauses in other activities).

Emerald is not a real-time system, so the latency should just be small compared to user activity, i.e., we do not have to meet real-time dead-lines. It should also be noticed that expedience is not our goal, as the collector has to wait for failed nodes to be available again. Some expedience may though be achieved by the local area-collector as it is not delayed by inter-node communication nor by unavailable nodes.

1.6 Design Criteria

The design and implementation relies on some underlying assumptions. These are mainly firm requirements to the behavior of our implementation. Some may though be seen as restrictions on the implementation.

Distribution

The underlying system model is that of a distributed system, i.e., a set of computer nodes interconnected by a local-area network. The nodes behaves like normal computers with one processor and own primary memory, that may be visible as a larger virtual memory using secondary storage. The system is characterized by relative slow cross-network communication (10^3) compared to node-local communication.

Our focus is on storage recycling by doing garbage collection at run-time, thus allocation and the many important issues on compile-time garbage collection are not described in this thesis. The same holds storage management in general.

The integration of garbage collection in the run-time system of a programming language raises several semantic issues. These are discussed below.

Transparency

If garbage collection is to be accepted by users in a system already running without it, the semantics of programs must not change. Furthermore, garbage collection must be added without changes in the current definition of the programming language.

Furthermore, we restrict the garbage collector implementation to the run-time system. The compiler is not changed and thus, synchronization between mutators and collector is done solely in the run-time system; mutators are not changed.

Mobility

Objects that moves very fast in distributed systems may move faster than the chasing garbage collector. They must, however, not out-perform the garbage collector. Thus they are handled when they are moved and marked as live, i.e., objects that move around are—by definition—not garbage.

Concurrency

It is an overall requirement that our collector runs without stopping the system at-large. Garbage collection is made concurrent by using the faulting mechanism. The garbage collection fault is used to handle a protected object before a user process gains access to the object.

The latency while initiating a garbage collection or handling a garbage collection fault must be kept short. The distributed collector may run for days until all nodes has been enough available, but it is not allowed to block a node-local garbage collector nor user processes for the whole period.

The Emerald prototype

By using Emerald as a testbed for our collector scheme, we rely on the characteristics of Emerald. Emerald is strongly typed and all storage management in Emerald is an issue for the language implementer. Thus, the Emerald programmer has no explicit control on storage allocation, deallocation etc. and the use of references is under full control by the run-time system with help from the compiler.

1.7 Work Done

The main effort in this work has been on the implementation of our proposed garbage collection scheme. Our proposal has been applied to the Emerald system.

A new Emerald prototype has been built that extends the Emerald kernel with:

- A *global garbage collector* that works distributed by using cooperating sub-collectors on all nodes.
- A *local garbage collector* to reclaim node-local garbage on each node.
- A *garbage collection fault* mechanism that enables both collectors to run concurrently with user processes.
- A robust synchronization scheme to *detect global termination* for the global garbage detection.
- Routines to *identify object references* in both user defined and system objects.
- A new level of parallel processing inside the kernel, the *garbage collector dispatcher*, which enables quasi-concurrency between user processes and garbage collectors on each node.

Further work has been concerned with:

- *Test of the implementation* of our garbage collection scheme on a network of four VAX-station 2000 workstations.
- *Measuring the performance* by executing artificial as well as more realistic Emerald programs using the prototype.
- Cooperation between the *checkpoint system* and the garbage collection is designed.
- Investigation of garbage collection techniques in general.

1.8 Evaluation Methods

Our ideas are evaluated by implementing the proposed garbage collectors and using them by running Emerald programs.

Implementation

We want to show that the comprehensive and faulting garbage collection algorithm works in a distributed, object-oriented system by implementing the collectors and faulting mechanism in the Emerald prototype.

Measurements

By measuring the time used in our Emerald prototype with and without garbage collection we want to evaluate the efficiency of the algorithm and the implementation to determine the overhead introduced. The latency introduced by garbage collection and the overall cost in time and space is measured as well.

Synthetic Garbage Creator

The synthetic garbage creator generates a variety of objects with different size and internal references to each other. Some are generated as local, some as full-fledged global objects. Also the distribution may be varying.

The structure of the object graph, i.e., the way objects and their references are distributed may vary from local sub-graphs to long linked lists crossing node boundaries each time.

Distributed Applications
Real applications gives us a less artificial benchmark. The evaluation is thus extended with applications written in Emerald. We have measured distributed applications like the Emerald Mail System.

1.9 The Thesis

1.9.1 Research Contributions

In summary, the major contributions of this thesis are:

A Comprehensive Garbage Collector in a Distributed System.
The implementation of the comprehensive garbage collector in Emerald works concurrently with user activity. To our knowledge, it is the first working implementation of comprehensive and concurrent garbage collection in a distributed systems.

Robust Garbage Collection in Distributed Systems.
Our implementation works even in the case of partial failures of nodes in the distributed system. We demonstrate that distribution does not need a central control unit nor complete concurrent availability of the whole distributed system. Instead, our robust garbage collector works with distributed control, i.e., no master, and it only needs pair-wise availability of the participating nodes.

Implementation of the Faulting Mechanism.
We have applied the idea of faulting on object invocation to garbage collection by introducing a *garbage collection fault.* The concurrency between our garbage collector and user processes is obtained by protecting objects not traversed from being invoked. User processes resume their activity when the garbage collector has marked and traversed these and protected the new referenced but not traversed objects.

A protected object will get a garbage collection fault, if one of its operations is invoked. The faulting mechanism resumes the invocation after the object has been marked and traversed and its references protected if necessary.

1.9.2 A Guide to the Thesis

The remaining part of the thesis consists of three main parts and a conclusion.

I. Background
Chapter 2 presents an overview of garbage collection techniques from early computer history, list-processing in the 1960's, up to recent 1990's developments. Background information on the Emerald system is presented in Appendix A. Thus, readers familiar with these ideas may skip either or both.

II. The Main Contributions
The design of a comprehensive and concurrent garbage collector for a failure-free, distributed, object-based system is presented in Chapter 3. The issues of garbage collection in distributed systems with temporary and partial failures are discussed in Chapter 4.

III. Implementation and Evaluation
The implementation of our garbage collection proposal in the Emerald prototype is

discussed in more details in Chapter 5. Also, the garbage collection fault mechanism is described here. Measurements and evaluation of the implementation are given in Chapter 6.

After the conclusion follows a bibliography of literature on garbage collection. In Appendix B the algorithms used by the implementation are specified together in more details. Appendix C characterize the systems running the Emerald prototype and presents a short overview of the changes applied to the source code of the Emerald kernel. Finally, a summary of the thesis is given in Danish as Appendix D.

Chapter 2

Garbage Collection Techniques

The need to use the same storage more than once, i.e., *recycling* storage, has existed since the first computer. Rebooting the computer does solve the problem in many situations, whereas others require more advanced methods to let the system run non-stopped or at least nearly uninterrupted. This thesis is concerned with recycling techniques taking effect while the system is running. Both the basic and more advanced techniques for detection and reclamation of no longer used storage, i.e., *garbage collection*, are surveyed in this chapter.

The first section presents a background on storage management, to motivate the use of garbage collection (Section 2.1). The second section identifies the historical roots of garbage collection, that is, the basic techniques that may be identified inside any garbage collector: Reference counting, mark-and-sweep collection, and copying collection (Section 2.2).

The following section surveys more advanced techniques, i.e., concurrent collection, real-time collection, incremental collection, area collection based on storage partitioning, traversal of the object graph, how storage reclamation can be done, and the cooperation with a recovery system for non-volatile storage (Section 2.3).

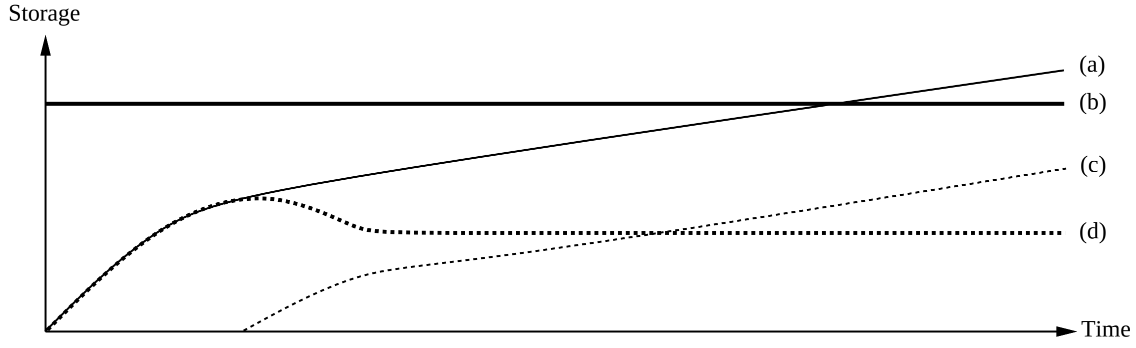
As this thesis is mainly concerned with distributed systems, collectors for such systems are surveyed separately in the last section of this chapter. The distributed collectors are based on various combinations of the non-distributed techniques extended to cope with distribution (Section 2.4).

2.1 Storage must be Recycled

Storage management is concerned with the allocation and deallocation of storage. This section describes the interaction between storage allocation and garbage collection. The main obligation of the garbage collector is derived from this.

In systems with dynamic allocation the *storage manager* delivers *chunks of storage* to the application on request. If the application does not return chunks of storage to the storage manager, the entire storage will sooner or later be allocated and no further allocation request can be met. The growing storage consumption by an idealized application with linear growth is illustrated by the curve (a) in Figure 2.1, where the maximum available storage is shown as line (b). If no deallocation is done explicitly by the application, nor automatically by the storage manager, the intersection of (a) and (b) in Figure 2.1 shows the time after which no more storage can be allocated.

To prolong the life time of the running application its allocated, but unused, i.e., unreach-



(a) The total amount of allocated storage.
(b) The maximum available storage.
(c) The total amount of no longer useful storage (garbage).
(d) The actual claim of storage, if garbage is recycled.

Figure 2.1: Efficient storage management

able storage, should be used again. The no longer used storage is exemplified by curve (c) in Figure 2.1. Thus by recycling this, the actual claim is reduced to the difference between the two curves, (a) and (c) as illustrated by (d). To do this, the application may either implement its own storage manager internally or cooperate with the general storage manager by telling which chunks of storage, it does not use any more. When moving storage management into the application, we have to solve the same set of storage management problems for each application, instead of solving them once in the general case. Thus, we have chosen to discuss the general case only.

The general storage manager must at least include:

- An *internal representation* of storage, where it is possible to identify the chunks that are not allocated, e.g., a *free-list*.
- An *allocation routine* which returns allocated storage to the requesting application.
- An *allocation strategy* which chooses a chunk of storage in response to a given allocation request.
- A *garbage detection strategy* about where, when, and how to identify chunks of storage as garbage.
- A *storage reclamation routine* which moves garbage from the application domain back to the storage manager.

The scope of this thesis is garbage collection in object-based systems, thus the chunks of storage allocated are either user or system objects. By shifting to this more abstract level of description, the obligation of garbage detection may be rephrased in terms of this higher level. The recycling of objects may also be viewed at this level of abstraction. Any kind of resource, that is modeled by an object, may be recycled, if that object is dead. The main resource and our primary concern remains, however, the storage.

2.2 Fundamental Garbage Collection Algorithms

The goal of garbage detection is to identify when an object is no longer reachable from the root set.

As mentioned, essentially two basic approaches to garbage detection exist:

1. *Reference counting* focuses on the individual object and count the number of external references to the object. A zero counter denotes a garbage object.

2. *Graph traversal* focuses on the object graph and traverse the object graph by following references to identify all objects reachable (directly or indirectly) from the root set. The reachable objects are live while the remaining are garbage.

The main difference between these two approaches is due to the perspective, a local versus a global view of the object graph. Graph traversal algorithms have traditionally been based on either a *mark-and-sweep* or a *copying collection*. The difference between them is a matter of how reclamation is done; their garbage detection techniques are essentially the same.

In the next section reference counting, mark-and-sweep collection, and copying collection are described, analyzed, and compared. A more exhaustive survey of these methods (up to 1980) is given in [Cohen 81].

2.2.1 Reference Counting

Reference counting associates a counter with each object to count the number of references to the object. Objects with positive counters are considered alive, while a zero counter identifies a dead object. Reference counting was first described by [Gelernter 60, Collins 60, Weizenbaum 62, Weizenbaum 63].

A system using reference counting must update the reference counter for an object each time a reference to that object is copied or deleted. Objects, that are part of the root set, have their reference counters incremented by one to simulate an invisible external reference to each of them. Objects are otherwise initialized with a zero counter when allocated, but the counter is immediately incremented as the reference to the object is given to the application.

An implementation of reference counting needs at least the subroutines *increment* and *decrement*. The reclamation of garbage, i.e., zero count objects, is done from the *decrement* subroutine when it decrease a counter to zero. When a reclamation takes place, all objects referenced from the dead object must also decrement their counters, as the references in the dead object are now “deleted”.

Example: Object Graph with Reference Counters

Figure 2.2 illustrates our example graph of objects and their reference counters during various updates of references, starting (a) with the graph of Figure 1.1. From (a) to (b) the object `H` is created by `C`, and thus `H`'s reference counter is incremented to 1. Next, a reference to `G` is copied to a variable in `H`, thus the reference counter of `G` is incremented from 2 to 3. Similar the deletion of a reference results in the reference counter of the referenced object being decremented. On Figure 2.2 (c) the reference from object `A` to object `F` is deleted, thus the reference counter of object `F` is decremented from 3 to 2.

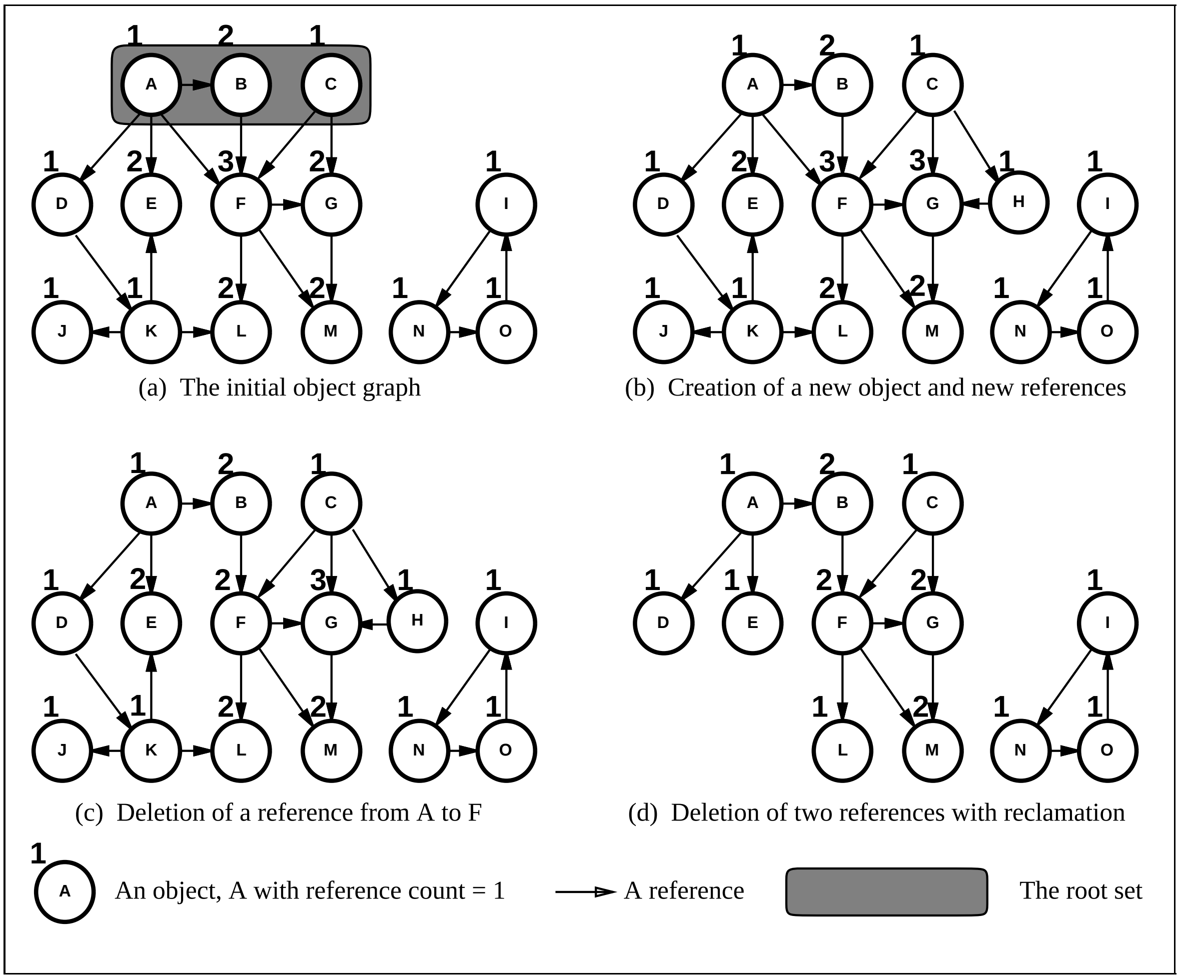


Figure 2.2: Reference counting in a graph of objects

As object `C` in Figure 2.2 (d) deletes its reference to `H` the reference counter of `H` becomes 0 and object `H` is reclaimed. Note that object `G` got its reference counter decremented to 2 as `H` was reclaimed (d).

The deletion of the one and only reference to the head of a long singly linked list results in a recursive reclamation of each of the list elements. Such deletions are also known as *cascade deletions*, as the deletion of the last reference to a group of objects results in a burst of successive storage reclamation. A small example of this is illustrated by (d) in Figure 2.2. When the reference from `D` to `K` is deleted, `K` reaches a zero count. During its reclamation the counter of object `J` becomes zero also. Thus both are reclaimed, whereas the reference counters for `E` and `L` are both decreased from 2 to 1.

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 $\leftarrow$ 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), e.g., the objects I, N, and O in Figure 2.2 are never detected as garbage in pure reference counting systems. 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.

2.2.2 Mark-and-Sweep Collection

The classical mark-and-sweep-algorithm may be used in a stopped system to collect garbage including cycles of garbage. As indicated by the name, the garbage collection is done in two phases. First, all live objects are marked, then the unmarked, garbage objects are reclaimed. Only the mark-phase depends on the temporary stop of the system. The sweep-phase may reclaim the garbage objects in parallel with the running system, as these two activities do not interact. The mark-and-sweep algorithm is usually attributed to independent work by Deutsch and by Schorr and Waite, although [Schorr 67] attributes the ideas to [McCarthy 60].

The algorithm uses a mark-field in each object. All objects are marked either white (potentially garbage), gray (alive with references under consideration), or black (alive with references considered). Furthermore, a root set of objects is given, i.e., the active processes and those "always present" objects.

Initially, all objects are marked white, i.e., they are potentially garbage. If they cannot be reached during the traversal of the object graph starting from the root set, they will stay white and be considered real garbage, when the mark-phase is finished. The mark-phase is initialized with the root set. Each of theses objects is marked black and each of the references in these objects are *shaded*: If the reference is to a non-black object, that object is marked gray. We use the term to *shade a reference at least gray* for this operation. Each gray object is alive but has not yet been traversed to get its references shaded. Thus to ensure that all reachable objects are finally marked black, the mark-phase proceed as long as there are gray objects. Each gray object is marked black and its references are shaded at least gray. When the mark-phase terminates, all objects are either black or white. The live objects have been marked black due to their reachability from the root set and the garbage objects are not touched, i.e., they are still white. The sweep-phase may now traverse the whole object storage to reclaim all white objects.

Example: Mark-and-Sweep in the Object Graph

Figure 2.3 (a) shows the example from Figure 2.2 with an object graph of 15 objects, where object A, B, and C are the root objects. Initially, all objects are marked white (a). After marking the objects in the root set: A, B, and C, the objects D, E, F, and G becomes gray (b).

Next, in the example, object D and E are marked black immediately as they does not contain references, and F and G are marked black while their references to L and M are shaded. Finally, L and M are marked black as they have no references to white objects. The result is

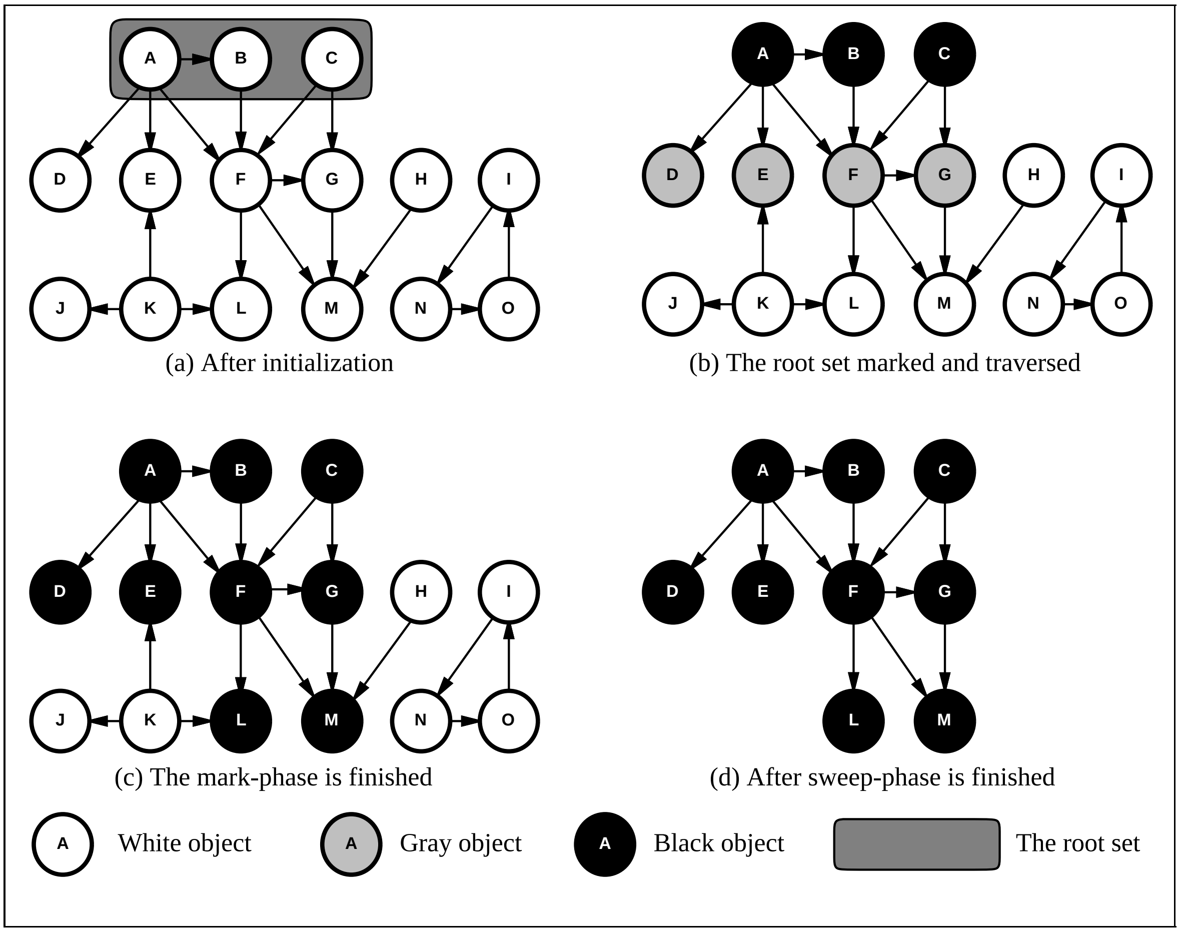


Figure 2.3: Mark-and-sweep collection in a graph of objects

illustrated by (c), where no gray objects are present. This terminates the mark-phase, and the sweep-phase may now reclaim all white objects (d).

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.

2.2.3 Copying Collection

Copying algorithms use more or less the same traversal mechanism as mark-and-sweep to detect the live objects. The difference is mostly how dead objects are reclaimed.

The basic *stop-and-copy* algorithm separates the object store in two halves: *toSpace* and *fromSpace*. All objects are allocated in the *fromSpace* until garbage collection is started and the user processes suspended. Like the mark-and-sweep algorithm the objects in the root set are traversed and their references *shaded*. Instead of marking traversed objects black, they are copied to the *toSpace* and internal references updated accordingly. For translation purpose a *forwarding address* is left behind from the old version in *fromSpace* to the new copy in *toSpace*. When all the live objects have been copied, and their external references updated, the complete *fromSpace* may be reclaimed.

Copying techniques need a closer cooperation with the *object manager* as all objects must be allocated in the *fromSpace*. When the collector is finished, all live objects reside in the *toSpace*, thus at this point the two spaces may be swapped. The swap changes the role of the two spaces, i.e., swap their names.

Stop-and-copy needs no mark-field as this information is implicit available. The *toSpace* contains black objects only. Gray objects are those objects in the *fromSpace* for which a reference in the *toSpace* exist. The backwards references identifies the gray objects during the traversal. Thus the *shading* of references need only register that these objects have to be copied also, and that a reference in *toSpace* has to be updated when they are copied to *toSpace*. Shading a reference to an object already copied results in updating the reference immediately to point at the copy in *toSpace*.

Example: Copying Collection in a small Object Graph

Figure 2.4 (a) illustrates the partitioning of an empty object store in two halves: *toSpace* and *fromSpace*. The situation at the start of a stop-and-copy collection is exemplified by (b). Then objects of the root set are traversed and their references *shaded* (c). When traversed, they are copied to the *toSpace* and internal references updated accordingly. A *forwarding address* is left in *fromSpace* to forward old references to the new copy in *toSpace* (c). During collection, the backwards references (c–d) identifies the gray set. Finally, all the live objects have been copied to the *toSpace* (e), and their external references updated. Then the complete *fromSpace* is reclaimed (f).

Copying collection may also be done as a *mark-and-copy* collection, where the mark-phase from the mark-and-sweep is completed first. Then all objects identified as live (black) are moved to *toSpace*. The first copying implementations were described by Minsky [Minsky 63], and by Fenichel and Yochelson [Fenichel 69].

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 *fromSpace* and a *toSpace*.

+ *Advantages:*
    - Collects all garbage including cycles of garbage.

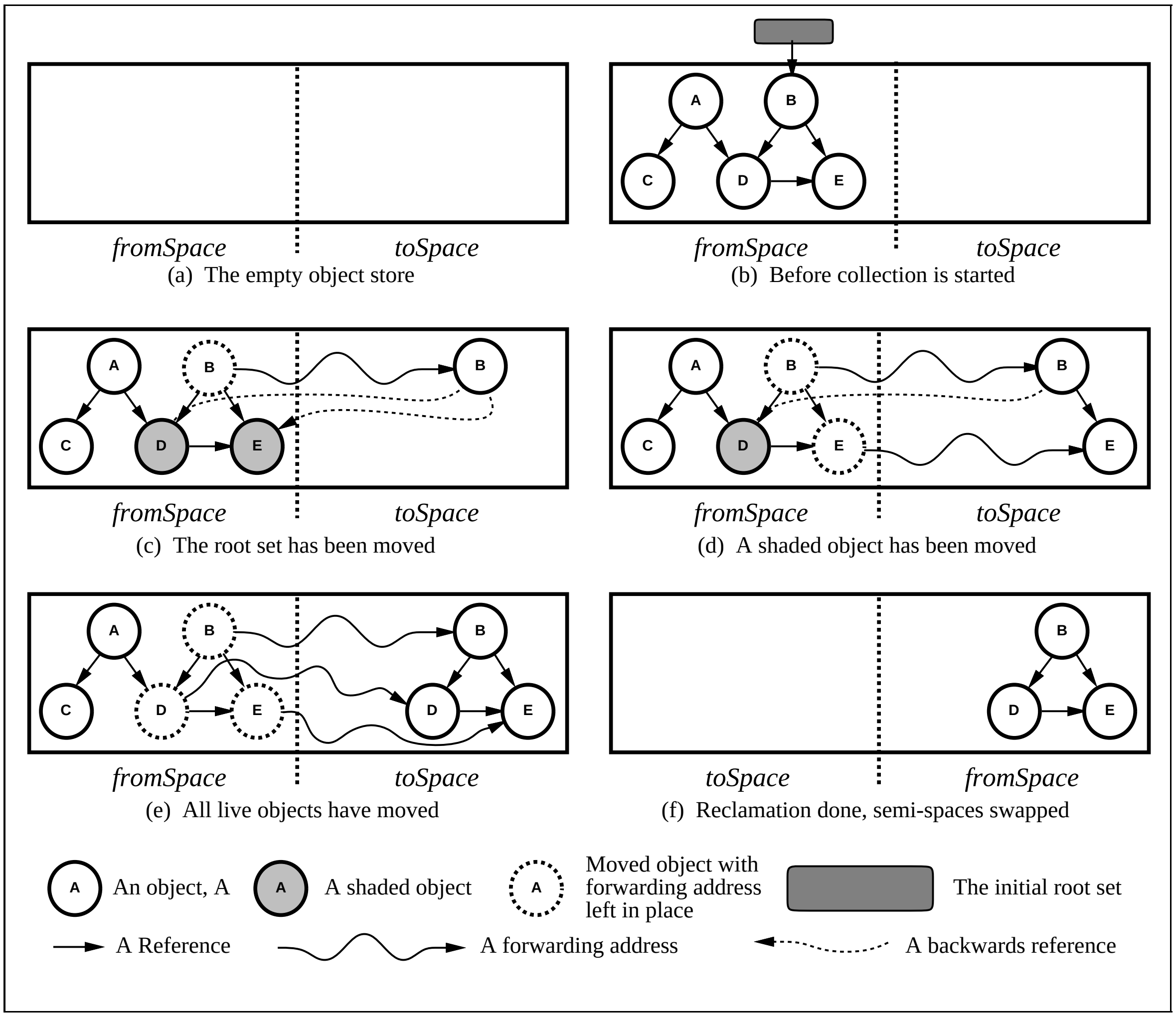


Figure 2.4: Stop-and-copy collection in a simple graph of objects

- 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 *toSpace*.
- All references must be updated and/or indirect/relative references used.
- Stops all activities in the system while running.

2.2.4 Comparison of Basic Collector 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 2.1: 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 $O(n)$.

Time Complexity

The time complexity of the three algorithms are:

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

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

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.

Storage Overhead

The storage overhead due to the three algorithms are:

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].

Copying collection
Half the storage is reserved for garbage collection (*toSpace*). 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 $O(n + l) = O(n)$.

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.

2.3 Advanced Garbage Collection

During the thirty years history of garbage collection, many terms have been used to describe the algorithms. Wilson describes in a lengthy survey [Wilson 92] the different trends in modern garbage collection techniques for non-distributed systems since Cohen's survey [Cohen 81].

Beside the three basic algorithms, focus may be on concurrency to achieve real-time behavior and storage partitioning. Various techniques for concurrent garbage collection is surveyed in Section 2.3.1. The following Section 2.3.2 proceeds by describing how techniques to partition the storage into areas has influenced garbage collection techniques. Our interest in methods coping with a partitioned storage stems from the physical partitioning of storage in a distributed system. Further techniques in advanced garbage collection are also surveyed. Section 2.3.3 describes how the object graph is traversed to identify the garbage objects, and Section 2.3.4 how storage reclamation is done. In Section 2.3.5 garbage collection and the aspects of robustness to failures in connection to transaction oriented systems using non-volatile storage are described by recent examples. Finally, a chronology of distinguished algorithms is given in Section 2.3.6.

Although the described garbage collection algorithms for non-distributed systems are grouped according to their main characteristics: concurrency, storage organization, graph traversal, garbage reclamation, and robustness, respectively, most of the algorithms do focus on more than one of these. Furthermore, other important design decisions exist, e.g., internal concurrency or reclamation of other resources.

[1] The cost of keeping a separate reference to each gray object during the collection adds a marginal $O(l)$ on the needed storage for both traversal algorithms. The mark-field needs only 1 bit in this case.

2.3.1 Concurrency with Mutators

Concurrent collectors need to synchronize with mutators when accessing common data, i.e., the objects they work on. Concurrency between mutator and collector may be achieved by constraining the allowed behavior of mutators during collection. *Incremental* techniques do this by interleaving the garbage collector inside system calls that are used by the mutator, whereas fully *concurrent* techniques need explicit synchronization between mutator and collector processes.

The reference counting scheme usually stops the mutator while accessing the object (increment/decrement the reference counter). By nature, reference counting is incremental as an object is collected immediately when the last reference to the object is deleted. This may still lead to cascades of deleted objects, when a deleted object causes the deletion of the last reference to another object, and thus introduce latency to the mutator.

In [Deutsch 76], Peter Deutsch and Daniel Bobrow present a *deferred, incremental reference counting* scheme supplied with a global copying collector to reclaim cycles of garbage and compact the live objects. Their reference counting uses tables of reference counts and defers the table updates by putting all updates in a transaction file. The file is reduced by removing sequences of reverse updates, e.g., increment follow by decrement of the same object. When the file has been applied to the reference counter tables, zero count objects may be reclaimed if not in the root set. Two ideas are suggested for the transactions due to the reclamation. Either they are applied immediately to the tables risking the mentioned effect of *cascade deletion*, or they are put in the new transaction file, thus deferring the reclamation of the tail of such structures. Most of the work done by deferred, incremental reference counting can be done concurrently with mutators, as their only interactions are:

- mutators put transactions in the tail of the file,
- the collector reads the log sequentially from the head, and
- the collector reads a consistent snapshot of the root set.

The collector work can be done by a separate collector processor or by a process running concurrently to mutators.

The first concurrent traversing algorithm described for a two-processor system was given by [Steele 75]. It follows a mark-copy-translate-reclaim scheme with semaphores for synchronization of access to common data between mutator and collector. The system described is a list-processing system, where the list operations are extended with synchronization operations. This strategy imposes many synchronization constraints on mutators during the various phases. During the copy-phase, the mutator is faced with indirect references until copying is finished.

A more advanced *incremental, copying collector* is given by Henry G. Baker [Baker 78]. It is the first real-time collector that moves a smaller subset of the live objects from *fromSpace* at a time. The garbage collector is build into the primary operation of the system, e.g., the LISP primitives: CONS, CAR, CDR. The storage lay-out is illustrated by Figure 2.5. During a collection the live objects are copied from *fromSpace* to *toSpace* and new objects are allocated in *toSpace*. The *toSpace* is partitioned in four sub-spaces (1–4) by the pointers S (scanned), B (bottom), and T (top) as shown on Figure 2.5. (1) and (2) contain objects copied from *fromSpace*, and (4) contains the newly allocated objects, whereas the subspace (3) is free. Thus the free space is decreased from both sides as objects are either copied or created.

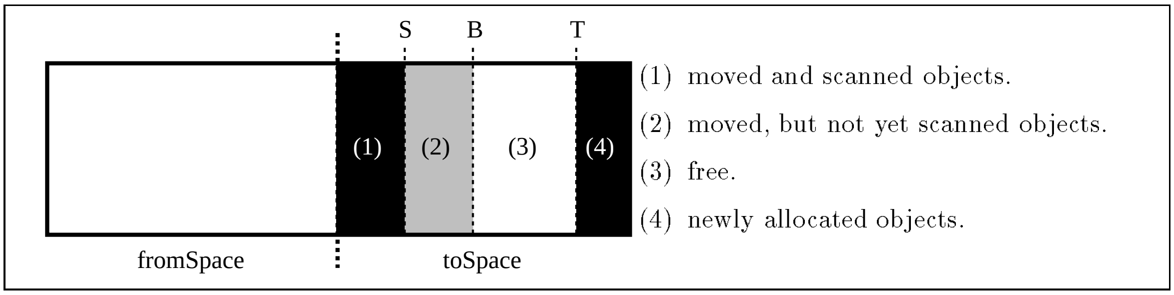


Figure 2.5: Storage lay-out for Baker's incremental copying collector

An object in (1) has been traversed and each of its references are now to a *toSpace* object, i.e., the objects it references, have been copied or are copied immediately as the object in (1) was traversed. New objects in (4) have the same characteristics. Only the objects in (2) have not yet been traversed, thus they may contain references to both *toSpace* and *fromSpace* objects. The correspondence to the three color mark-and-sweep collector is illustrated by the indicated colors on Figure 2.5. In Baker's algorithm objects are not colored, but objects in the sub-spaces hold the same characteristics as the color indicates, i.e., black ≡ marked and traversed, gray ≡ marked but not traversed, and white ≡ potential garbage.

The collector is initiated by moving the root objects to an empty *toSpace* and advancing B accordingly. The S and T are still at the edges of the *toSpace*. The mutators ensures that the rest of the job is done by traversing a number of objects in (2) before each allocation. When all objects in *toSpace* has been traversed, i.e., S = B, all live objects must have been copied and the *fromSpace* can be reclaimed. If the mutator access an object in (2), it may get a reference to *fromSpace*. In such cases the referenced object is immediately copied to *toSpace* and the reference updated before it is given to the mutator.

When T = B, no more free space is available for new objects and the next collection is initiated by swapping *toSpace* and *fromSpace*. Note that it is crucial, that S = B is reached before T = B. This is achieved by adjusting the number of objects traversed during each allocation.

Dijkstra, Lamport, Martin, Scholten, and Steffens describe a concurrent mark-and-sweep collector in [Dijkstra 78] to show how the synchronization can be proven correct. To simplify reasoning about the correctness of the algorithm they regard all non-allocated objects, i.e., the free-list, as part of the live objects. Thus the free-list is put in the root set, the consequence is that non-allocated objects are traversed during the mark-phase. The mark-phase of the algorithm marks live objects black by successive scans of the whole object store looking for gray objects. Each time a gray object is found, it is marked black and its references shaded by marking the referenced objects gray, if they were white. The scanning continues until no gray objects are found during a complete scan. The reclamation is done by yet another scan over the whole object store, where non-marked objects are reclaimed, i.e., doing the traditional sweep-phase. As the algorithm was described to illustrate a technique to prove the correctness of concurrent programs, the inefficiencies were not considered further. Due to its concurrency, the algorithm was named *on-the-fly collection*. The algorithm and its proof has later been refined, e.g., see [Ben-Ari 84, Pixley 88].

Almost simultaneous to [Dijkstra 78], Kung and Song developed another concurrent mark-and-sweep collector with lower overhead [Kung 77]. They does not put the free-list in the root set, and use a dequeue to hold information about gray objects that must be traversed

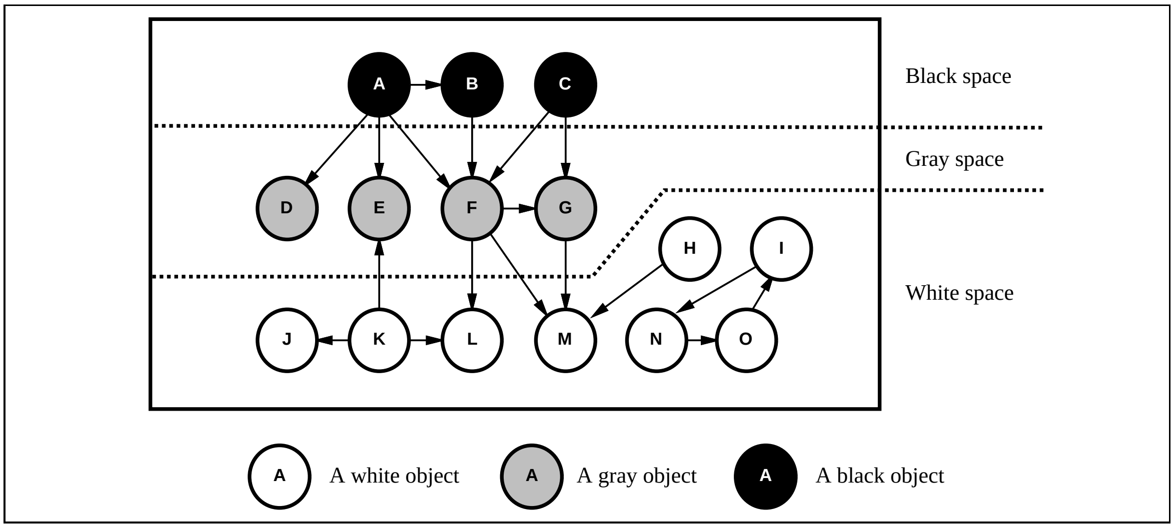


Figure 2.6: Storage partitioning during the mark-phase

during the mark-phase. In this way, much of the overhead paid by Dijkstra's algorithm is removed at the cost of storage overhead to contain the dequeue, which could be problematic as garbage collection is usually done when a shortage of storage has occurred. They use a fourth color, off-white, denoting non-allocated objects to circumvent traversal of the free-list.

With a traversing collector the final result of the collection is the reclamation of garbage objects, while the mutators only access live objects. Their critical interaction happens when the mutators change the object graph while the collector traverses the graph. An important observation, which may be used to limit this synchronization overhead, is the virtually partitioning of storage during the mark-phase by the colors of the objects. Figure 2.5 and 2.6 illustrates this partitioning during copying and mark-and-sweep collection respectively, where three sub-spaces are present. The collector moves white objects to the gray space, and gray objects to the black space. It never change the graph of objects during the mark-phase. Meanwhile mutators may work in the black space as long as only black objects are changed. Thus only black and gray references are copied or deleted. By restricting mutators to update only black objects, the graph traversal by the collector is never disturbed.

A protecting mechanism to prevent mutators from accessing objects and violating the invariants of the collector is useful as illustrated by Baker's collector, where synchronization is implemented by added translation and copying for each reference to *fromSpace*. A general solution includes a way to *protect* the object, *trap* the mutator, and *unprotect* the object again [Juul 91]. This should all be done without changing the code for mutators while still preserving any mutator invariant. The use of a *faulting mechanism* for garbage collection was first proposed by Jul in [Jul 87, Jul 88b] for the Emerald system. Strictly speaking, we need only to prevent mutators from updating gray objects, reading them is okay. Thus the protection scheme is sometimes characterized as *holding the write-barrier* from the black to the gray objects.

Appel, Ellis, and Li present in [Appel 88] a real-time, *concurrent copying garbage collector* implemented on the DEC Firefly shared-memory multiprocessor. The collector uses virtual memory hardware to protect references to *fromSpace* objects. Before such a reference can be used by a mutator the hardware will generate a *page-fault*, thus invoking the collector to copy the live objects on this page to *toSpace*. The needed features of the hardware underneath this and other *faulting collectors* are discussed in [Appel 91].

Such protection schemes could work with a broad range of protection grains. From coarse-grain protection of the whole storage space or semi-spaces, via page protection as above, to fine-grain protection on individual objects or object parts. In any case, *concurrency* is achieved by holding the *write-barrier* between different parts of the storage. Recent works by Sharma and Soffa [Sharma 91] and by Boehm, Demers, and Shenker [Boehm 91] use a variety of page-protection and dirty-bit respectively to achieve concurrency for generational as well as conservative collectors.

2.3.2 Storage Organization

The system imposed organization of storage may be utilized by the garbage collector and the garbage collector may impose further organization constraints on the storage. We have already mentioned copying collection, which impose a partitioning of the available storage in at least a *toSpace* and a *fromSpace*. Further garbage collection techniques are:

Area collection

This strategy takes advantage of a given partitioning in nearly independent semi-spaces or makes such a partitioning to do independent collection in each semi-space. Each semi-space is an area and its root set is extended with the table of *entry* references. Peter Bishop [Bishop 77] describes a partitioning of a large LISP system, where *area collection* has been applied.

Generational collection

An extension to area collection is to move live objects from one area to the next as time passes. Thus young objects are in the *nursery* area, whereas older objects get promoted to an older area. The common name for these areas are generations, as they reflect the lifetime of the objects in them. The first algorithm of this kind was presented by Lieberman and Hewitt in [Lieberman 83]. These algorithms are also called *generational scavenging* and different techniques may be applied to determine when to promote an object [Ungar 84, Ungar 88]. Also the *ephemeral copying collector* by Moon [Moon 84] for garbage collection in a large heap for the Symbolics 3600 LISP implementation uses a storage partitioning based on the lifetime of objects. Both mark-and-sweep and copying algorithms may be extended with generations. A comparative study [Zorn 90] shows that the CPU overhead using mark-and-sweep instead of stop-and-copy is $3-6\%$ whereas the physical storage requirement is $20-40\%$ less while achieving the same page-fault rate (the simulations where done in a paged system).

Group collection

Applied to reference counting, areas may be used to form groups of objects, that are strongly inter-connected, thus making it possible to perform reference counting on groups instead of on the individual objects [Bobrow 80]. This extend reference counting to be able to identify cycles of garbage, if such cycles are the only objects in a group. The algorithm is known as *group reference counting*.

"Don't do it" collection

When the available storage is extremely large, garbage collection could be postponed for years. This does, however, need some kind of computation to ensure locality of the live objects. Jon White [White 80] purpose a copying collector, that takes advantage of a next to infinite tertiary backing storage for seldom used objects.

2.3.3 Graph Traversal

The graph traversing algorithms partition the object storage according to the color as illustrated on Figure 2.6. The sequence in which gray objects are selected has an impact on performance. In large address spaces, supported by virtual memory paging on secondary storage, the different objects have very different access time depending on their current position in- or outside the primary memory.

Boehm and Weiser [Boehm 88] define a collector, which does a *conservative* collection, i.e., not a comprehensive collection. This collector does not depend on identification of references neither in root set nor during the traversal. Every item in storage, which looks like a pointer, is treated as such. While potentially preserving too much, e.g., preserving a garbage object pointed to by an integer, the collector cannot be allowed to move objects between spaces[2]. Thus the conservative collector is based on mark-and-sweep. According to [Wentworth 90], conservative collection may introduce large leakage as they preserve garbage from being collected.

The conservative approach is taken a bit further by Joel Bartlett [Bartlett 88, Bartlett 89], who defines a *mostly copying* collector based upon ambiguous roots. Still the algorithm is not comprehensive. The conservative collection has recently been combined with generational collection by Demers, Weiser, Hayes, Boehm, and Shenker [Demers 90]. This conservative generational mark-and-sweep collector is based on implicit generations due to the lifetime of objects.

The conservative approach may even work in language independent systems where pointers are un-tagged in storage. However, even a conservative collector may fail to preserve a live object if the reference to the object is hidden, e.g., by encoding the pointer as the reverse bit pattern or if offsets and pointer arithmetic is used. Thus conservative collectors may be fooled to collect to much.

2.3.4 Garbage Reclamation

After the garbage has been identified, it must be reclaimed. At least three topics are of interest here, i.e., when, how, and what to reclaim. The scheduling of the method and the method itself are two sides of the same coin. The decision on what to reclaim, e.g., other system resources than storage, may kick-back on the detection process also.

When and how to reclaim garbage

The reclamation of storage may be done incrementally on a per object basis, as in reference counting or for small groups of objects as in area-collectors for small areas. The reclamation process may also be more or less relaxed from the garbage detection. In mark-and-sweep collectors, as well as in multi-generational collectors, the reclamation process can be detached from the garbage detection process as long as the reclamation process gets information about the detected garbage.

[2] Moving an object, requires a translation of the references pointing at the old address. If any of these references are really some valid data, i.e., not a pointer, the translation will change data values, thus invalidating the whole program.

By applying concurrency to the garbage collector itself, a mark-and-sweep collector can be turned into a *mark-during-sweep* as described in [Queinnec 89] which interleaves the mark and the sweep-phases. The scheme takes advantage of the property, that *garbage stays garbage* by doing the sweep-phase of one collection while the next mark-phase proceeds. Instead of using a simple mark-field, the scheme may be extended to use a counter of the last collection, which identified the object as living. After the k'th mark-phase, all objects marked $k - 1$ or less may be reclaimed during a sweep, thus relinquish the reclamation process further from the individual mark-phases.

What shall be reclaimed

Algorithms for collection of unused storage may be equally applicable to collect unused processor power, i.e., executing processes, which will never affect the system by other means than using processor cycles. We call such processes, orphans. The importance of this issue is raised by systems where *call-by-future* evaluation is used extensively. Baker and Hewitt describes an incremental algorithm for the unified collection of storage and process in [Baker 77].

An orphan may be detected by an inverse traversing algorithm using the process as root. Processes, that has no connection to the outside world, i.e., to i/o channels, are traversed and if the traversing reaches a live object this root is a live process, not an orphan. [Kafura 90, Washabaugh 90] describes a combined algorithm, named *push-pull*, where the root set is defined dynamically to be the processes, that reach a live object.

It is worth noting that any system resource, exclusively reserved by a garbage object, should be released as the garbage is collected.

2.3.5 Robustness, Recovery, and Transactions

In object-based systems that work on long-lived data, e.g., object-oriented databases and persistent programming languages, where recovery and robustness to failures are important, the garbage collector must respect and cooperate with the system according to its model of failure and recovery. These systems may use transactions for consistent updates of the non-volatile storage. An *atomic collector* is a collector that preserves correctness of the recovery system, the state of the non-volatile storage, and the collector itself, across failures of these systems.

Elliot Kolodner defines in [Kolodner 89] an *atomic, stop-and-copy collector* for a stable heap (a persistent heap where computation on shared state occurs within atomic transactions). The atomic garbage collector is coordinated with the recovery system (of the transaction system) to retain correctness of both the garbage collector and the recovery system. Due to the pause introduced by "stopping the world", the collector is useful in small sized heaps only. In [Kolodner 90, Kolodner 92a, Kolodner 92b], this inefficiency is removed and an *atomic, incremental garbage collection* scheme for large stable heaps is described. The scheme is based on [Appel 88] to be incremental.

A similar collector was also presented by David Detlefs, [Detlefs 90]. His scheme is a conservative collector for a transaction system integrated with C++. Detlefs presents a concurrent mostly-copying collector for C++, which employs the read barrier of Appel [Appel 88], and then shows how to make it atomic. Detlefs's atomic collector is likely to be less efficient than Kolodner's because it is not as well integrated with the recovery system.

2.3.6 Chronology of non-distributed Collectors

The Tables 2.2 and 2.3 give an overview of the surveyed sequential garbage collection techniques. The overview is sorted by publication dates, to serve as a chronology of garbage collectors.

Year	Name	Characteristics	References
1974	Deferred incremental reference counting	Reference counting using a transaction file and separate tables. [2.3.1]	Peter Deutsch and Daniel Bobrow [Deutsch 76]
	Multiprocessing compactifying collection	A parallel, mark-and-copy collector. [2.3.1]	Guy L. Steele [Steele 75]
1975	On-the-fly collection	Mark-and-sweep garbage collector, that works concurrently with mutators. [2.3.1]	Dijkstra, Lamport, Martin, Scholten, Steffens [Dijkstra 76, Dijkstra 78]
1976	Incremental garbage collection of processes.	Collection of both storage and irrelevant processes. [2.3.4]	Henry G. Baker, Carl Hewitt [Baker 77]
	Incremental, real-time garbage collector	A concurrent copying collection, using semi-spaces and a protection of the fromSpace [2.3.1]	Henry G. Baker [Baker 78]
1977	Area copying collection	Large address space partitioned in areas with separate copying collections. [2.3.1]	Peter B. Bishop [Bishop 77]
	On-the-fly collection	Mark-and-sweep garbage collector which work concurrent with mutators.[2.3.1]	H. T. Kung, S. W. Song [Kung 77]
1979	Group reference counting	Reference counting applied between groups of objects instead of individual objects for reclamation of cycles of garbage. [2.3.2]	Daniel Bobrow [Bobrow 80]
1980	"Don't do it" collection	In gigantic LISP systems the adding of a third level backing storage may be used for old unused objects instead of doing garbage collector. Bakers copying collection is used to move the old objects out of primary and into tertiary storage.[2.3.2]	Jon L. White [White 80]
	Real-time collection based on object lifetime	Generation-based semi-spaces [2.3.2]	Henry Lieberman, Carl Hewitt [Lieberman 83]
1984	Generational scavenging	Copying between multiple semi-spaces, defining objects to belong to semi-space according to their current lifetime. [2.3.2]	David Ungar, Frank Jackson [Ungar 84, Ungar 88]
	Ephemeral copying collection	Large heap collection for Symbolics 3600 LISP. [2.3.2]	David Moon [Moon 84]

Table 2.2: Garbage collection schemes from 1970 – 1984 (part 1)

Year	Name	Characteristics	References
1987	Faulting collection	Mark-and-sweep collector for Emerald using object protection to achieve concurrency. [2.3.1]	Eric Jul, Norm Hutchinson, Andrew Black, and Hank Levy [Jul 87, Jul 88a, Jul 88b]
1988	Conservative collection	Conservative mark-and-sweep collection. [2.3.3]	Hans Boehm and Mark Weiser [Boehm 88]
	Mostly copying collection	Copying collection with ambiguous roots. [2.3.3]	Joel Bartlett [Bartlett 88, Bartlett 89]
	Real-time concurrent collection	A concurrent two-space copying collector using page-fault for synchronization with mutators. [2.3.1]	Andrew Appel, John Ellis, and Kai Li [Appel 88]
1989	Mark-during-sweep	Mark-and-sweep collector with parallel execution of the mark-phase while the sweep-phases of the previous cycle is done. [2.3.4]	Christian Queinnec, Barbara Beaudoing, and Jean-Pierre Queille [Queinnec 89]
1990	Atomic incremental collection	Atomic, robust, and concurrent copying collector cooperating with the recovery system for transactions in a persistent heap. [2.3.5]	Elliot Kolodner [Kolodner 90, Kolodner 92a, Kolodner 92b]
	Concurrent atomic collection	Atomic, concurrent, mostly-copying collector for C++ including cooperation with the recovery system for transactions in a persistent heap. [2.3.5]	David Detlefs [Detlefs 90]
	Conservative generational collection	Conservative mark-and-sweep collector based on implicit generations due to the lifetime of objects. [2.3.3]	Demers, Weiser, Hayes, Boehm, Bobrow, and Shenker [Demers 90]
	Push-pull collection	A push-pull mark-and-sweep collector with dynamic defined root set based on the processes' ability to access their environment. [2.3.4]	Dennis Kafura, Doug Washabaugh, and Jeff Nelson [Kafura 90]
1991	Parallel generational collection	Generational copying with collection in multiple generations concurrently. [2.3.2]	Sharma and Soffa [Sharma 91]
	Mostly parallel collection	Using dirty-bits to do concurrent collection with either conservative mark-and-sweep or a generational copying collector. [2.3.2]	Boehm, Demers and Shenker [Boehm 91]

Table 2.3: Garbage collection schemes from 1986 — 1992 (part 2)

2.4 Distributed Collection

In distributed systems inter-node references complicates garbage collection, especially when some nodes are temporarily unreachable. However, many references tend to be short lived and local [Lieberman 83, Schelvis 88, Jul 88b, Rudalics 86]. Thus partitioning the storage in node-local areas and applying garbage collection locally to each, should collect a large part of the garbage.

Most distributed collectors, whether based on local area collectors or not, are satisfied with the collection of most of the garbage [Bevan 89, Goldberg 89, Lester 89, Shapiro 90, Mancini 91]. For many practical purposes this is enough, though memory leakage is a long term threat to such solutions.

To make a comprehensive collection, the system must extend the local area collectors with a global collector. Such a collector can be made by letting a collector on each node cooperate [Augusteijn 87, Schelvis 89]. Or, as Lieberman and Hewitt suggest, by moving objects to a node that needs them or the creator node, when no local reference exist on the current node [Lieberman 83].

The distributed collectors may be classified by their basic algorithm. This does not say, that they may not use a mixture of algorithms. In general, there is no constraint between the internal algorithm used locally on a node, and the global algorithm used to make the local collectors cooperate as one global collector.

Abdullahi, Miranda, and Ringwood survey distributed garbage collection techniques at large [Abdullahi 92]. In the following, we present some of the main techniques used in distributed collection. A variety of local/global cooperation schemes are described mostly independent of their basic algorithm, followed by how distributed systems have achieved garbage collection using *reference counting* and *traversing collection* respectively. Finally, a chronology of distributed collectors is given.

2.4.1 Partitioning Collectors

In Bennett's version of Distributed Smalltalk [Bennett 87] an existing local collector on each node perform its own collection. To prevent globally known objects from being locally collected, each node has a table of remotely accessed objects, which are part of the root set. We call this the *InTable*. The global collector removes globally detected garbage from these tables, thus leaving the reclamation to the local collectors. In general, this may be done by removing the global garbage from the local root set.

Another version of Distributed Smalltalk by Marcel Schelvis [Schelvis 88, Schelvis 89] is based on local collectors which cooperate by exchanging messages with timestamps. Thus the basic algorithm may be either traversing or counting.

Barbara Liskov and Rivka Ladin describes in [Liskov 86] a collection scheme, where independent local collectors work independently according to their own algorithm. Their cooperation on collecting, what is only globally detectable as garbage, is achieved by utilizing a general mechanism of the system, i.e., they use a logical centralized, but physically distributed, service to reveal cross-node references. The service makes the global collection robust to failures and limited availability of the individual nodes of the distributed system.

Mohamed Ali presents a number of distributed collection schemes [Mohamed Ali 84]. The simplest schemes do stop-mark-and-sweep on the entire distributed heap, managed by a coordinator node. The *distributed local scheme* consists of independent local collectors with incoming references in their root set. The result of each local collection is local reclamation and an update of the incoming references on the other nodes. The last scheme does *distributed, real-time, copying collection* by combining the area-collector idea [Bishop 77] and Baker's real-time copying collection [Baker 78]. Still, neither schemes collect distributed cycles of garbage.

2.4.2 Distributed Reference Counting

In true parallel systems, e.g., distributed systems, locking or synchronization is necessary to ensure the correct serialization of reference counter updates. When extending reference counting to cope with distribution, two problems must be solved. First, cyclic garbage needs to be detected by other means. This is a general defect of reference counting, although it is even harder to solve in the distributed case. Second, non-garbage may be reclaimed when a decrement message arrives before an earlier increment message. This may happen when the counter update messages arrive out of order when transmitted across node boundaries, i.e., a counter may be decremented to zero, and the object reclaimed, though a reference exist, but the increment message indicating this has been delayed.

A variety of solutions to overcome the later problem has been published [Watson 87, Bevan 87, Bevan 89, Piquer 91]. The key feature to prevent decrement messages from overrunning increment messages is to send one type of messages only. This is achieved in *weighted reference counting* by associating a counter with both the object and the references to the object. Initially, during, and after collection, the sum of counters on references to an object is equal to the counter with the object. When a reference is copied, its counter value is divided between the new and the old reference so that the sum of the two is equal to the counter of the original reference, e.g., by associating half of the original value with both references. Such a division of a counter in two halves can be done locally, where the reference is copied, thus no update messages need to be transmitted across the network. A deleted reference results in a decrement message being sent to the object. Thus objects are born with a large count value that becomes zero, when the last reference is deleted. To overcome problems dividing counters with small values, e.g., 1, indirection objects are introduced. An indirection object replaces the reference with the small value, still associated with that reference. The indirection object contains a large value itself which is divided among the references to the object. Piquer shows how the indirection object may be avoided at the cost of an additional counter field in the objects [Piquer 91].

Goldberg has proposed a similar scheme called *generational reference counting* in [Goldberg 89, Goldberg 91], which classifies the references according to the number of times they have been copied. Eckart and LeBlanc describes in [Eckart 87] a proposal for a distributed *reference marking* algorithm. References are classified as either *defining* or *borrowed* as proposed in [Spector 82]. Instead of reference counters in objects references are supplied with information of their origin and use. This resembles a reference counters with values 0 or 1 on the reference and ordering of references to an object in a tree. To collect cycles of garbage restrictions are imposed on how applications may update references.

The *deferred reference counting* scheme proposed for sequential systems may also be applied to distributed reference counting to limit the communication costs.

2.4.3 Distributed Traversing Collectors

A global traversing garbage collector must handle all the potentially cross-node references and detect garbage not referenced from any node, even if the nodes do not run simultaneously.

The *marking-tree collector* [Hudak 82] does a distributed mark-and-sweep collection with real-time performance and reclamation in parallel with mutators. It prevents to use centralized data and control, other than a logical rendezvous between the phases of the collector. The scheme is based on forking off a marker task incrementally for each referenced object, which has not been marked yet. Each marker task keeps a counter on the marker tasks it has initiated, and reports back to its initiator, when it has received reports from all its own marker tasks. The marker tasks represent a distributed marking-tree. The mark-phase is finished when the tree is empty, i.e., when the root of the marking tree receives a report from the last of its marker tasks. Thus, the overhead due to communication is fairly large. The collector does orphan collection also.

POOL [Bronnenberg 89] is a family of parallel object-oriented programming languages to be executed on a distributed system architecture, DOOM. The object model, the language, and the underlying architecture are all very similar to the Emerald approach [Beemster 90, Wester 90, Spek 90]. The garbage collector for POOL [Augusteijn 87] consists of cooperating local mark-and-sweep collectors. Its concurrency with mutators is achieved by traversing and marking incrementally at invocation time. The end of the mark-phase is detected by messages exchanged between the nodes and a synchronizer on one of the nodes.

Especially distribution introduces the need to survive failures, as failures may affect only part of the system. The following two schemes let cooperating local collectors adapt to the current availability and complete in the available part of the system, using conservative estimates for the non-available parts.

Marc Shapiro, David Plainfossé, and Oliver Gruber propose in project SOR [Shapiro 90, Shapiro 91] a low level, language independent system collector. The conservative local tracing collectors are extended with tables for incoming and outgoing references. A protocol for table updates based on previous local collection results and another that detects distributed cycles of garbage make the scheme robust to node failures and recovery, but it is not comprehensive. The proposed scheme was tested in a distributed LISP system (Transpive) [Plainfossé 92].

Luigi Mancini, Vittoria Rotella, Simonetta Venosa describe in [Mancini 91] a similar collector for Galileo to collect objects on non-volatile storage. The collector is a stop-and-copy type, which may not be the best solution in a distributed system, if it stops the mutators for too long a period. The authors propose a fault-tolerant adaption when nodes become unavailable. The scheme enables a non-comprehensive collection, based on conservative estimates of the root set to complete while part of the system is unavailable.

Also Lang, Queinnec, and Piquer describes distributed collection schemes [Lang 92], which eventually reclaims all inaccessible objects. They further refine a scheme based on independent node collectors, like that used in the project SOR to cooperate for various groups of nodes. The scheme is expected [Lang 92] to be comprehensive eventually, if each distributed (interconnected) graph of garbage objects is contained in a group of nodes, which do not fail during the group collection.

2.4.4 Current Issues in Research on Garbage Collection

The recent development of robust collectors for distributed systems are even richer than described here. The problems in distributed garbage collection are currently an ongoing

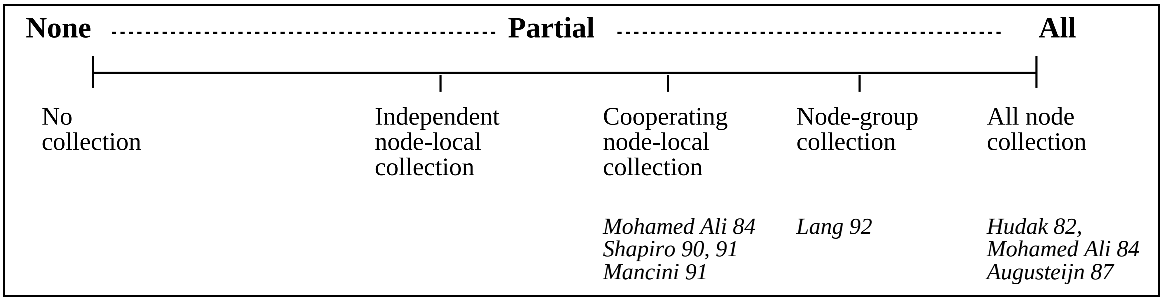


Figure 2.7: The degree of partial garbage collection

research field. Further proposals may be found in [Hughes 85, Lermen 86, Vestal 87, Tel 87, Tel 91] and in the survey of Abdullahi, Miranda, and Ringwood [Abdullahi 92] as well as other papers in [Bekkers 92].

One of the challenging problems is to combine partial collectors and comprehensive collectors to achieve both robustness to failures and expedient collection of some garbage. As an algorithm approaches comprehensiveness it becomes more and more dependent on the availability of the entire distributed system, thus robustness is traded-off for expedience. Furthermore, a comprehensive collection depends on all nodes in the distributed system, thus, the mere presence of communication delays in distributed garbage collection often rules out the possibility of fulfilling the goals of comprehensiveness and expedience by a single collector. Thus, a trade-off between comprehensiveness and expediency is necessary.

By trading off comprehensiveness, we achieve a partial collection, i.e., only part of the garbage is collected. Figure 2.7 describes the various degrees of partial collection, from collection of nothing, to collection of all garbage, with a broad variety of partial collectors, which are able to collect only part of the garbage, in between. From the most partial distributed collectors like independent node collectors with conservative root estimates to comprehensive, distributed collectors like the ones implemented for POOL [Augusteijn 87] and described by [Hudak 82, Mohamed Ali 84]. In between, we find collectors which collects more or less distributed garbage, by cooperation of local node collectors. The degree of cooperation may range from a few nodes, over a group of nodes, to nearly all nodes. The Galileo and SOR projects are examples of such cooperating node collectors [Mancini 91, Shapiro 90]. [Lang 92] describes a distributed collection scheme, which reclaims all inaccessible objects in a group of nodes, and adapts the definition of the group of nodes according to current availability of nodes.

General Implications

Last but not least, we also note that distributed collectors face the same problems as does any distributed concurrent application. Thus the study of distributed garbage collection has a merit both as an important case in garbage collection and as an interesting distributed application in its own right.

The systems described in this section, their garbage collection schemes, and references to further descriptions are shown chronologically in Table 2.4.

Year	Name	Characteristics	References
1982	Marking-tree collector	A parallel, distributed marker task per reachable object is created, representing the tree of reachable objects. [2.4.3]	Paul Hudak and Robert M. Keller [Hudak 82]
1984	Distributed real-time copying collection	Independent node-local Baker collectors with InTables in roots and removal of garbage entries from remote InTables. [2.4.1]	K. A.-H. Mohamed Ali [Mohamed Ali 84]
	Replicated garbage collection service	Independent local collectors and a global logical centralized (but distributed) service for cross-node references. Robust to failures and limited availability. [2.4.1]	Barbara Liskov and Rivka Ladin [Liskov 86]
1987	Local/global collection	Dual collector scheme with InTables for Bennetts version of Distributed Smalltalk. [2.4.1]	John Bennett [Bennett 87]
	Local collection with cooperation	Local collectors cooperating by exchanging messages with timestamps for another Distributed Smalltalk. [2.4.1]	Marcel Schelvis [Schelvis 89]
	Cooperating local collectors	Local incremental mark-and-sweep and cross-node marking at invocation time. The "no gray object" state is the global termination point detected by message exchange between the nodes and a synchronizer on one of the node. [2.4.3]	Lex Augusteijn, Ben Hulshof, Marcel Beemster [Augusteijn 87, Beemster 90]
	Weighted reference counting	Decrement from a positive count associated with both the object and its references as references are deleted. [2.4.2]	Watson and Watson, Bevan, Piquer [Watson 87, Bevan 87, Bevan 89, Piquer 91]
1989	Generational reference counting	References classified by the number of times they have been copied. [2.4.2]	Benjamin Goldberg[Goldberg 89, Goldberg 91]
1990	Adaptive local tracing	Low level, language independent local tracing collectors with tables for incoming and outgoing references. [2.4.3]	Marc Shapiro, David Plainfossé, and Oliver Gruber [Shapiro 90, Shapiro 91]
1991	Non-volatile storage collector	Global stop-and-copy collection of objects on non-volatile storage with adaption to the current set of available nodes. [2.4.3]	Luigi Mancini, Vittoria Rotella, and Simonetta Venosa [Mancini 91]
1992	Adaptive group tracing	Local tracing collectors cooperating for dynamically identified groups of nodes with tables for incoming and outgoing references for each group. [2.4.3]	Bernhard Lang, Christian Queinnec, and José Piquer [Lang 92]

Table 2.4: Distributed garbage collection schemes

2.5 Summary of Survey

We have surveyed traditional garbage collection techniques and described the basic schemes:

- reference counting,
- mark-and-sweep collection, and
- copying collection.

Furthermore, the techniques to make traversing collections concurrent is described. An overview of further refinements to the basic collection schemes is given, including techniques for:

- real-time collection,
- incremental collection,
- area collection, and
- conservative collection.

Garbage reclamation and recovery in transaction system is mentioned with current solutions like:

- mark-during-sweep, and
- atomic collection.

The non-distributed schemes are used as building blocks in distributed garbage collection proposals. Recent schemes and current issues in the field of distributed garbage collection have been presented to further motivate our goals towards an implementation of a comprehensive, concurrent, and robust garbage collection scheme for distributed systems.

Chapter 3

The Design of a Comprehensive Distributed Garbage Collector

This chapter discusses problems in comprehensive and concurrent garbage collection for failure-free distributed systems. The basic assumptions and our requirements to a garbage collector, i.e., *comprehensiveness*, *concurrency*, *distribution*, and *efficiency*, are rephrased (Section 3.1). Algorithms and invariants concerning efficient garbage collection in a distributed environment are developed by step-wise refinement (Sections 3.2-3.8).

The first algorithm is a distributed, but centralized, sequential, and blocking mark-and-sweep algorithm (Section 3.3). This generic algorithm forms the basis for the later refinements. It is first constrained with the requirements (Section 3.4) and each of the deficiencies found is then analyzed and circumvented (Sections 3.5-3.7). The chapter concludes that a comprehensive, concurrent, distributed, and efficient garbage collection algorithm has been achieved in a distributed system where all nodes are fully available (Section 3.8).

In this chapter, we have assumed a failure-free distributed system. Chapter 4 discusses garbage collection and robustness with respect to temporary and partial failures in the distributed system. Chapter 5 discusses the actual implementation of our garbage collector in the Emerald prototype.

3.1 Garbage Collector Goals Revisited

Before developing our collector algorithm, we will rephrase our requirements to a collector in the current environment, i.e., a failure-free, distributed system. The requirements are based on the goals set forth in Section 1.5.

Example: A Distributed Graph of Objects

In the following, we use the graph described in Section 1.2 and first illustrated in Figure 1.1 as an example. The distributed graph of 15 objects on 6 nodes is shown as Figure 3.1.

Under the assumption that the distributed system works without failures, the goals may be rephrased: *The collector must be comprehensive, concurrent, and distributed without introducing unnecessary inefficiencies.* Of course, the collector must never collect a live object, i.e., it must be correct. A closer look at the revised goals and their consequences reveals:

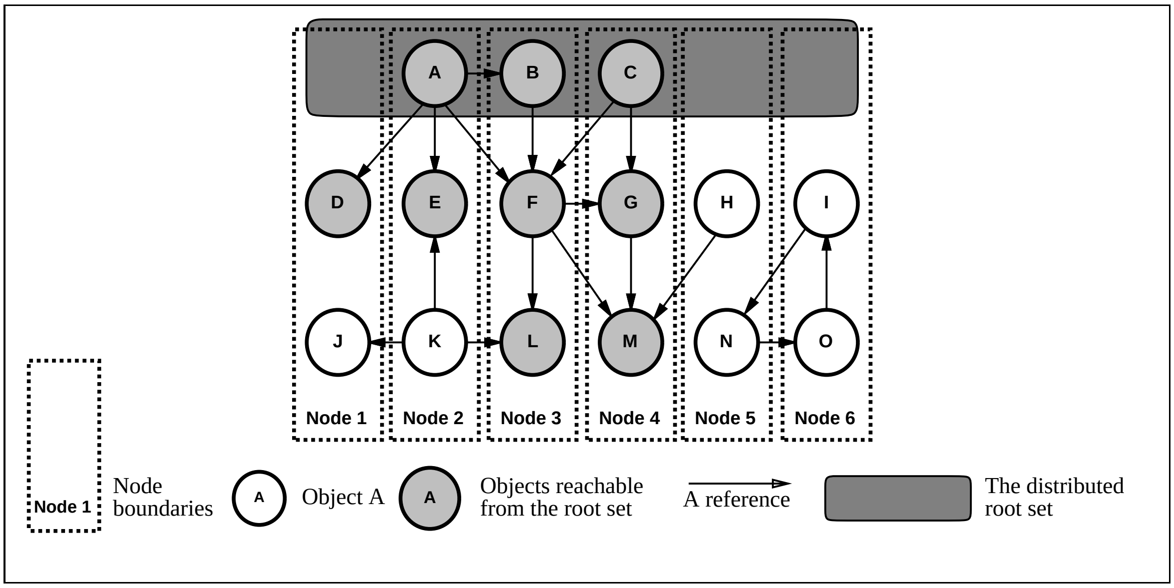


Figure 3.1: A distributed graph of objects

Comprehensiveness

As all objects, that were garbage before the garbage collector is started, must be collected, the collector cannot be based on conservative techniques like [Boehm 88] or partial collections like [Shapiro 90]. Both the root set and the live references must be determined exactly. Comprehensiveness also demand collection of cycles of garbage, even cycles spanning several nodes.

The Example: The objects `I`, `N`, and `O` on Figure 3.1 constitute such a distributed cycle of garbage, which must be collected by our collector.

Moreover, comprehensiveness restricts the solution from using traditional reference counting techniques, i.e., a comprehensive collector must traverse the graph of references.

Concurrency

Our collector must work "on-the-fly", i.e., concurrently with user processes (mutators). The blocking period should be kept small enough to be almost insignificant to the mutators, e.g., on the order of the time slice used by the CPU scheduling algorithm. The maximum time should be bounded, i.e., not dependent on the number of objects. Thus blocking collectors like *stop-and-copy* cannot be used. As the collector may run concurrently with mutators, there is a need to synchronize their access to the common object store.

The Example: If the root set of Figure 3.1 consist of three mutators originating in `A`, `B`, and `C`, these are able to execute in object `A`, `B`, `C`, `D`, `E`, `F`, `G`, `L`, and `M` while the collector is running concurrently.

To further minimize the blocking period, the collector may be designed as a set of concurrently running, cooperating processes (coroutines). The concurrency inside the collector introduces further, yet more fine-grained, synchronization constraints.

Distribution

In a distributed system the graph of references may cross node boundaries. Also interconnected garbage may be distributed. Thus, *comprehensiveness* and *distribution* together demands that up-to-date global information on the graph of references between objects is available.

The Example: As an example, `Node 1` on Figure 3.1 cannot determine whether objects `D` and `J` are alive or not without information from the other nodes.

Efficiency

Efficiency is one of the design constraints, but expedience is not always achievable. To be comprehensive, we tolerate that the garbage collector may run for a very long time before all parts of the distributed system have been checked. Thus, the collection must be done concurrently with other activities and each pause introduced by a garbage collection-step must be kept short. Furthermore, the total number of steps should summarize to the smallest possible grand total. To take advantage of the parallelism available in the distributed system, our collector should be partitioned into node collectors, one on each node. Also, concurrency inside each node collector introduces means of exploiting parallelism and keeping each step small, and thus limiting the period the mutators are blocked.

To summarize, efficiency can be achieved by letting the garbage collector exploit natural parallelism, cut down pauses, and limit the total overhead.

3.2 Plan for the Development of a Distributed Collector

A garbage collector, that meets the revised goals, is developed during the rest of this chapter. First a generic algorithm, which does not meet all requirements, is presented in Section 3.3. This sequential mark-and-sweep garbage collector is distributed, but centralized, sequential, and blocking. This generic algorithm is held up in Section 3.4 against the revised requirements to analyze and describe each of its deficiencies.

The following three sections contains a step-wise refinement where each of the deficiencies found is circumvented and more advanced algorithms developed through the Sections 3.5 to 3.7. First, the collector and mutators are allowed to run concurrently by means of the garbage collection fault mechanism (Section 3.5). Second, the collector is distributed, i.e., partitioned in node-local collectors that cooperate (Section 3.6). Third, the algorithm is made more streamlined by detailing how the invariants and color coding may be implemented including asynchronous inter-node communication and interleaving of the mark and sweep phases on each node (Section 3.7). Finally, the algorithm developed in this chapter is evaluated (Section 3.8). It is comprehensive, concurrent, distributed, and efficient, but depends on full node availability.

Throughout the presentation, invariants that maintain a consistent system during garbage collection are also developed. An assumption, underlying all these collectors and invariants, is that *garbage stays garbage.* This is, however, a natural assumption in object-based systems where unreachable objects never become reachable again.

3.3 The Generic Garbage Collector

A generic algorithm is described as Algorithm 1. It is based on the traditional mark-and-sweep collector, but differs from the classic version described in Section 2.2.2 in that it works in a distributed system where all mutators are stopped during the whole collection. The collector is centralized, i.e., it runs on one node in the distributed system and reaches all the other nodes by synchronous communication.

As usual when doing a mark-and-sweep garbage collection, we use a mark-field in each object. The field may take one of the three colors white, gray, or black, as value. The meaning of the colors are:

white The object may be garbage. When garbage collection starts every object is regarded potentially garbage (white) until the collector has reached it by a reference from a live object. When a garbage detection terminates all white objects are really garbage.

gray The object is reachable, but the marking of objects, referenced by this one, has not been finished yet, i.e., the traversal of the object has not finished yet.

black The object is reachable, and all objects referenced directly from this object are gray or black.

The color of an object is changed from the initial white to the intermediate gray—and from gray to the final black—by means of two functions. The functions which may be applied to the object during the mark-phase are:

Shade marks a white object gray.

Mark-and-traverse marks a gray object black.

Moreover, Mark-and-traverse takes the object it marks black and *Shade* all the objects it references. Thus Mark-and-traverse is applied to an object, whereas *Shade* may be view as being applied to a reference to an object.

The algorithm depends on the following invariants and rules about coloring.

Invariant 1: Monotone shading

During one garbage collection cycle (except during the initialization of the mark-fields) objects never get a brighter color, i.e., the coloring is a monotone function that shades the live objects ($white \mapsto gray \mapsto black$). $\boxed{\textbf{I 1}}$

Invariant 2: No black to white references

A black object never contains a reference to a white object, i.e., when an object is marked black, then every other object it references is marked at least gray, i.e., gray or black. $\boxed{\textbf{I 2}}$

The algorithm can be described as eight steps grouped in three phases as indicated in Algorithm 1. These phases are found in all version of the algorithm. The tasks of the individual steps must also be present, but their ordering and/or grouping in phases may be different. Thus, one *garbage collection cycle* is equivalent to one execution of these eight steps.

Algorithm 1 (Generic)
The centralized garbage collector:

Initialize	1.1	For all nodes, all mutators are suspended.
	1.2	For all nodes, all objects are marked *white*.
	1.3	The root set of all root objects from all nodes is established, thus defining the *global root set*.
Mark-Phase	1.4	The *global root set* is marked *black*, i.e.: For each object in the *global root set*, *Mark-and-traverse*[1] the object on the node hosting the object.
	1.5	The *gray set* is marked *black*, i.e.: While there are more *gray* objects, choose one and *Mark-and-traverse*[1] that object on the node hosting the object.
Sweep-Phase	1.6	For all nodes, the run-time system is updated to reflect the fact that *white* objects are now considered garbage.
	1.7	For all nodes, the storage occupied by *white* objects is reclaimed.
	1.8	For all nodes, the suspended mutators are resumed.

Subroutine Mark-and-traverse:

Traverse the object to *Shade*[1] all its references to other objects and mark the object *black*.

Subroutine Shade:

Shade an reference at least *gray*, i.e., the referenced object becomes *gray*, unless it was *black* (or *gray*) already. Thus the net effect is, that a *white* object is marked *gray* on the node hosting the object.

$\boxed{\textbf{A 1}}$

3.4 Deficiencies in the Generic Algorithm

The generic algorithm, Algorithm 1, does not meet all the requirements of Section 3.1. Its main deficiencies are discussed here. First, it suspends mutators during a whole collection cycle, i.e., no concurrency with mutators and thus longer pauses. Second, it does not employ any parallelism internally. Moreover, due to the distribution of objects, a large overhead is introduced by the synchronous communication with other nodes in all steps. Thus many delays are introduced by time consuming communication between the nodes. The algorithm needs to be optimized to diminish the latency and provide a more efficient collection. Algorithm 1, though simple and inefficient, does, however, collect all garbage correctly, i.e., it is comprehensive.

[1]The exact definition of the functions *Mark-and-traverse* and *Shade* are given at the bottom of the algorithm.

Now, consider each of these deficiencies in turn:

The garbage collector is not interleaved with the mutators

The mutators are suspended during the whole collection. This should be changed without changing the mutators, as correct user programs working without garbage collection must work with garbage collection without change. Section 3.5 presents a solution to this.

No distributed concurrency achieved

The centralized garbage collector does its actions on the other nodes in a remote procedure call fashion. Thus only one node is executing at a time. The available processing power should be utilized by distributing the execution of the garbage collector to concurrent processes, at least one on each node. Moreover, mutators and collectors should also be allowed to run concurrently on different nodes. Section 3.6 and 3.7 present solutions that cope with these problems.

Further inefficiencies

The above mentioned deficiencies are our primary concern. Further optimizations may be obtained by also interleaving different parts of the garbage collection cycle. Both the mark-phase and the sweep-phase may be implemented as separate processes. Thus step 1.5 of the current garbage collection cycle and step 1.7 of the previous cycle may both progress concurrently with mutators, i.e., *mark-during-sweep* is achievable in parallel with running mutators. Also the encoding of colors by mark-fields and/or sets of references can be optimized. A solution to these problems is discussed in Section 3.7.

The first steps: Some quick fixes

Besides the optimization obtainable by taking the above mentioned deficiencies into account the two traversals of the whole object storage (step 1.7 and 1.2) may be interleaved. This optimization are applied immediately by removing step 1.2 from Algorithm 1. Instead the obligations of step 1.7 is changed to do the tasks of both steps during the traversal of the object storage. For each object, i.e., each allocated object, we look at the color of the mark. A white object is recycled, i.e., returned to the *object manager*. The mark-field of a black object is changed to white, i.e., made ready for the next garbage collection cycle.

New objects allocated before the first garbage collection cycle and between the revised version of step 1.7 and step 1.3 of the next garbage collection cycle must be initialized with white in their mark-field also. Thus we constraint the object manager to allocate all objects white. Such an initialization may come for free, if made in cooperation with the default object initialization.

3.5 Concurrency with Mutators

The next step is to limit the length of the period where mutators are blocked by the collector. A much shorter latency than available in Algorithm 1 may be achieved by not suspending the mutators during the whole garbage collection cycle, i.e., narrowing the gap between the task of the first and last step (1.1 and 1.8). Mutators and collector do, however, interact as they use the same storage. Thus restrictions in their concurrent execution are implied. To identify the restrictions needed, we describe the characteristics of the two types of processes.

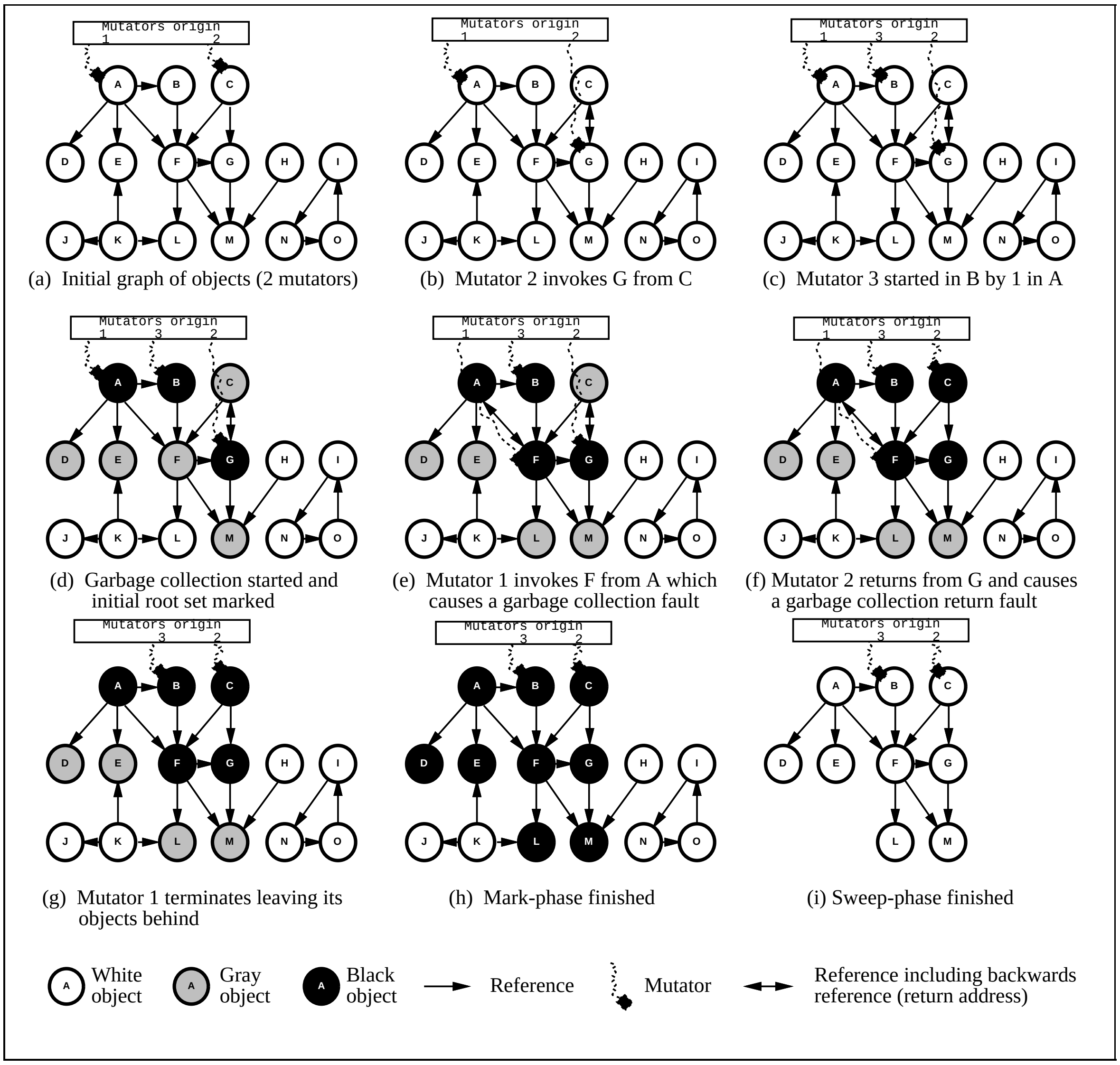


Figure 3.2: Mutators in the graph of objects before, during, and after a garbage collection

On this basis, our concurrent algorithm is presented. It takes advantages of the atomicity of certain garbage collection operations, as viewed from the mutators. Furthermore, the mutators are allowed to execute in only a small part of the object graph. When a mutator extends its execution outside this small part, a run-time mechanism ensures that the graph is kept consistent as viewed from the garbage collector.

Example: A Graph of Objects with Mutators

The interference between mutators and the collector is illustrated by the various stages of the object graph in Figure 3.2. The initial state of the graph of objects (a) follows our general example from Figure 3.1 with the addition of two mutators `1` and `2`, executing in object `A` and `C` respectively. *(Example to be continued later)*

3.5.1 Mutation of the Object Graph

A mutator is characterized by its position in its own life cycle: *birth, life,* or *dead.* A mutator is created by another mutator and a mutator may (but need not) cease to exist, e.g., if it terminates. While *alive* a mutator may be characterized as either *active* or *passive*, i.e., either running, or blocked by another mutator. An active mutator may cause the shift from passive to active of another mutator and from active to passive of it self, e.g., by waiting for a synchronization signal from another mutator. Furthermore, an active mutator may mutate the object graph by:

- **create** a new object or mutator,
- **copy** a *known*[2] reference,
- **delete** a *known*[2] reference, and
- **delete** or otherwise invalidate an object *known*[2] by a reference.

Implicit copying of a reference occurs when a mutator invokes an operation in another object. The mutator continues in the invoked object, which gets an implicit reference to the previous object, i.e., the return address.

Continued Example: Mutators working in a Graph of Objects

In the example of Figure 3.2, when mutator `2` does an invocation from `C` to `G` (b) an implicit reference `G` $\rightarrow$ `C` is added. However, when mutator `1` forks another mutator `3` to start running in object `B` (c) by using the reference `A` $\rightarrow$ `B`, *no* implicit reference is added from `B` $\rightarrow$ `A` as processes do not return. *(Example to be continued later)*

3.5.2 Where Collectors Affect Mutators

As seen from the mutators, the collector is characterized by its execution of the tasks in Algorithm 1. The interference from the collector is best described by the three types of interference: directly, by the use of the object graph, and by the use of data structures in the run-time system.

1. **Mutators** are affected directly by the *suspension* of active mutators, and the *resumption* of the suspended mutators.
   The goal is to limit the interval between these two tasks.
2. **The object graph** is read to identify references between objects and the mark-field is read and updated by the collector while:
   - All root objects are marked black and their references to white objects marked gray.
   - Each gray object is marked black, and its references to white objects marked gray.
   - White objects are recycled and the mark on all other objects is reset to white.

   The creation of new objects falls into this category also. The object graph must retain a consistent coloring of the individual objects and protect the invariants.

[2]A *known* reference is a reference available—explicit or implicit—in the current object.

3. **The run-time system** data structures are read while the root set is gathered and updated when tables containing "weak-pointers" are adjusted, e.g., references to white objects are deleted during the sweep-phase.
   The main purpose here is to keep the run-time system consistent and to deliver a consistent snapshot of the system to any mutator.

We now have to decide when to suspend the mutators explicitly, or implicitly by providing synchronized access to the common data (*critical regions*).

3.5.3 Mutators Enabled during most of a Garbage Collection Cycle

As noted previously, *garbage stays garbage*. Thus, once it is identified, the mutators are unable to interfere on this part of the object graph, i.e., the white objects. Thus the sweep-phase may be done independent of the resumption of the mutators, which may be moved from the bottom to the top of the sweep-phase. Synchronized access to tables in the run-time system by step 1.6 and the creation or deletion of objects by mutators is still needed. We demand these operations to be atomic.

During the initialize-phase the collector gets informed about the "system state" at-large. Thus a mutator that changes its own state (active/passive), is born, dies or creates new mutators, must be suspended at least during creation of the root set (step 1.3) and during its own *Mark-and-traverse* (step 1.4). This enables mutators to be resumed individually after their *Mark-and-traverse* has completed.

The remaining interaction is that mutators may now change the graph of references while the rest of the root set and later the gray set is *Mark-and-traverse*. Our solution to this problem is inspired by the logical partitioning of the object graph by the color of the objects as illustrated on Figure 2.6 and by Figure 3.2(d-h). The mark-phase of the garbage collector moves an object from:

- white to gray, when reachability is detected, and
- gray to black, when all its references are at least gray.

The garbage collector does no further change to black objects during the mark-phase; only it might need to read the mark-field when trying to shade the object from another object. Thus, mutators may execute in black objects without problem and that is where they are resumed. Invariant 2 (page 46) ensures that mutators executing in black objects has references to black or gray objects, but never to white ones.

To prevent mutators from executing in objects other than black ones, the gray ones are protected. When the mutator tries to access a protected object, a *garbage collection fault* is caused. The scheme is similar to a *page-fault* caused by a processor issuing a memory request, if the memory page is not available. Like a page-fault handler ensures that the needed page is made available, the garbage collection fault handler ensures that the object is executable by doing a *Mark-and-traverse* on that object, removing the protection and resuming the mutator. Thus, the invoked object is moved from the gray to the black set of the virtually partitioned storage.

Continued Example: Mutators and Collector working Concurrently

In the example of Figure 3.2, a garbage collection is started after (c). The initial root set, i.e., the objects where mutators are running currently, are marked black and referenced objects at least gray (d). While the collector runs, mutator `1` invokes a gray object, `F` from `A` which causes a garbage collection fault. The invocation results in a temporary reference from `F` to `A`, and the garbage collection fault ensures that object `F` is traversed and marked black (e). When mutator `2` returns from `G` to `C` (f), it returns to a gray object. Here a garbage collection fault on return ensures that `C` is Mark-and-traversed before execution of mutator `2` is resumed in `C`. *(Example to be continued later)*

3.5.4 The Faulting Collector

The concurrent collector is presented as Algorithm 2. It is based on the two previous mentioned (page 46) invariants and the following two, i.e., Invariant 3 and 4.

Invariant 3: Gray objects are protected

A gray object is protected against any use, i.e., invocation of its operations. **I 3**

Invariant 2 on page 46 and Invariant 3 above ensure, that when a mutator wants to do something with a non-black object, that object is *gray and protected.*

The faulting garbage collector is outlined as Algorithm 2. The mutators are constrained in this scheme without applying changes in the mutator program. Instead, the run-time system ensures that the invariants are maintained.

Algorithm 2 (Faulting)
The Garbage Collector:

Initialize	2.1	For all nodes, each mutator is suspended including mutators arriving from other nodes.
	2.2	For all nodes, the root set of all root objects is established, thus defining the *global root set.*
Mark-Phase	2.3	The *global root set* is marked *black* and mutators resumed, i.e.: For each object in the *global root set*, *Mark-and-traverse* the object on the node hosting the object. If the object was a suspended process, i.e., one of the suspended mutators of step 2.1, the process is resumed.
	2.4	The *gray set* is marked *black*, i.e.: While there are more *gray* objects, choose one and *Mark-and-traverse* that object on the node hosting the object.
Sweep-Phase	2.5	For all nodes, the run-time system is updated to reflect the fact that *white* objects are now considered garbage.
	2.6	For all nodes, the storage is traversed sequentially: Each *white* object is reclaimed and each *black* object is marked *white.*

The Object Manager (allocation):

New objects are allocated with the same color in their mark-field as the object (mutator) creating them, i.e., *black* or *white.*

(Algorithm continued on next page)

The Garbage Collection Fault Handler:

Suspend mutator while the object causing the fault is marked (*black*) and traversed to *Shade* its references at least *gray*, i.e., *Mark-and-traverse*. Finally, the object is unprotected and the mutator resumed.

Subroutine Mark-and-traverse:

Traverse the object to *Shade* all its references to other objects and mark the object *black*.

Subroutine Shade:

Shade a reference at least *gray*, i.e., the referenced object becomes *gray*, unless it was *black* (or *gray*) already. Thus the net effect is, that a *white* object is marked *gray* on the node hosting the object.

A 2

Invariant 1, 2, and 3 must be maintained by the garbage collector. Our faulting mechanism supports this maintenance and all together this ensures correct garbage collection if mutators are executing in black objects only during the mark-phase.

Invariant 4: Mutators only work inside black objects

A running process is always executing in a black object during the mark-phase. **I 4**

This invariant is initially maintained when mutators are resumed in step 2.3. The *Mark-and-traverse* of a mutator includes its *current stack frame* and the object in which it will be resumed. The other invariants ensure that Invariant 4 is maintained.

During garbage collection mutators are only able to execute in black objects, and they have no access to white objects. When they progress, they may either:

1. *Continue* execution in the current object.
2. *Invoke* an operation in a referenced object.
3. *Return* from current object to the object, from where the current invocation was made.

The first does no harm to the invariants. The second and third moves the thread-of-control forwards (Figure 3.2 (b and e)) and backwards (Figure 3.2 (f)), respectively, to a gray or black object. In the case of a gray object (Figure 3.2 (e-f)), the invoke or return causes a *garbage collection fault* to ensure that the object is black. For reasons of efficiency, the Emerald garbage collector ensures that the whole call stack of a mutator is made black in step 2.3. Thus mutators never return to a non-black object[3].

A new object may only be created by a mutator. Mutators may, however, execute during most of the garbage collection cycle. To preserve our invariants, especially Invariant 2, new objects must be born with the same color as their mutator. Thus a new object becomes either white or black depending on the mutator that creates it and the status of the concurrent

[3]However, due to distribution, we later relinquish this constraint to only ensure, that the node-local part of the call stack is black. Then the *Mark-and-traverse* must be applied to the local part of the stack, when a mutator returns from another node. In general, the restriction may be completely removed when shading the invoking object, and thus using *faulting on return* to a gray object.

garbage collector. The object becomes:

white if the mutator has survived the previous sweep-phase (step 2.6) and not yet reached its suspension in the next garbage collection cycle (step 2.1), or

black if the mutator is resumed, i.e., between the start of the mark-phase (step 2.1) and the survival test of the sweep-phase (step 2.6).

Continued Example: Finishing the Collection

Figure 3.2 illustrates how Algorithm 2 works concurrently with mutators. The progression of marking is ensured both by mutators through the garbage collection fault mechanism (e and f), and by the collector itself (d, g, and h). When no more gray objects exist (h), the mark-phase is finished and the sweep-phase executed resulting in (i).

Algorithm 2 fulfills our requirements to enable mutators and collector concurrently and thus, limiting the latency introduced on the mutators by garbage collection. The mechanism used to achieve this is the garbage collection fault mechanism on both invocation and return from invocation across node boundaries.

3.6 The Distributed Garbage Collector

The next step is to let the garbage collector utilize the true concurrency available in a distributed system by distributing parts of the garbage collector. Though working concurrently with mutators the previous Algorithm 2 does not take advantage of the natural parallelism in a distributed system. Instead, its centralized collector waits for each remote shading to complete, where the number of shadings is on the order of the number of cross-node references. The next refinement distributes the whole garbage collector by running parallel garbage collectors, one on each node. Algorithm 3 (page 55) shows how the collector on each node is defined. These *node collectors* cooperate, e.g., synchronize, as discussed here.

The individual steps of the algorithm has been refined accordingly. Note how global synchronization is applied to ensure that the individual node collectors run in-step at 3.1, 3.5, and 3.6. The synchronization in step 3.5 is hidden inside the wording *until the global gray set is empty.* Step 3.6 must be done immediately after the termination of step 3.5. It is done as an atomic update on each node, thus it works like a global atomic update also. The rest of the sweep-phase may run independently on all nodes. However, the next garbage collection cycle cannot be started by step 3.1 until all nodes have completed their previous sweep-phase.

Algorithm 3 present a garbage collection scheme with one local collector on each node and a coordinator process on one node. The coordinator ensures a simple way to achieve global synchronization, at the start and end of a new collection (step 3.1 and 3.7) and between the mark-phase and the sweep-phase (the end of step 3.5). Furthermore, the distributed concurrency in the garbage collector of Algorithm 3 is achieved by synchronous request messages send to shade objects from the node referencing the object to the node hosting the object. The handling of the remote request to shade a resident object is done while the collector is progressing with its resident gray set.

Algorithm 3 (Distributed)
The Garbage Collector on each node:

Initialize	3.1	Synchronize nodes and approve a coordinator node. This ensures that all collectors are started in their initialize-phase, no matter which node took the initiative.
	3.2	All mutators are suspended including arriving mutators from other nodes.
	3.3	The *resident global root set* is established.
Mark-Phase	3.4	The *resident root set* is marked *black* and mutators resumed, i.e.: For each object in the *resident root set*, *Mark-and-traverse* the object. If the object has a suspended process, i.e., one of the suspended mutators of step 3.2, the process is resumed.
	3.5	Until the *global gray set* is empty: While there are more gray objects on this node, choose one and *Mark-and-traverse* that object.
Sweep-Phase	3.6	For all nodes, the run-time system is updated to reflect the fact that *white* objects are now considered garbage.
	3.7	For all nodes, the storage is traversed sequentially: Each *white* object is reclaimed and each *black* object is marked *white*.

The Object Manager on each node:

New objects are allocated with the same color in their mark-field as the object (mutator) creating them, i.e., black or white.

The Garbage Collection Fault Handler on each node:

Suspend mutator while the object causing the fault is marked (*black*) and traversed to shade its references at least gray, i.e., *Mark-and-traverse*. Finally, the object is unprotected and the mutator resumed.

Subroutine Mark-and-traverse:

Traverse the object to *Shade* all its references to other objects and mark the object *black*.

Subroutine Shade:

If the reference is to a resident, *white* object, put it in the *resident gray set* and protect the object. If the reference is to a *white*, non-resident object, a remote shade call is issued on the node hosting the object.

Remote Shade Handler on each node:

Mark-and-traverse the object and return an acknowledgment to the caller.

A 3

3.7 Refinements of the Distributed Garbage Collector

The final refinements of the garbage collection algorithms of this chapter are done now. The garbage collection presented by Algorithm 3 did not explain the role of the coordinator node. Nor has the implementation of the colors been discussed. Solutions to these problems and further refinements to do more efficient, distributed, concurrent garbage collection are discussed here.

3.7.1 Asynchronous Shading of Non-Resident Objects

As objects are recognized as living by *Mark-and-traverse*, they get *shaded* by the task *Shade* described in Algorithm 3. Due to the constraints implied by the Invariants 2 and 3, the shading was done immediately, even in the case of a non-resident object, i.e., shading a reference included access to the object referenced. Thus shading was done as a remote procedure call to the node hosting the object, which introduced pauses on the calling node. We want to investigate ways to prevent this blocking of the caller node, and to batch such asynchronous *remote shading requests*, to enable the shading of a reference without accessing the object referenced while still working correctly. As the invariants may be violated temporarily, such situations must be limited and made invisible to those depending on the invariants.

Asynchronous exchange of remote shading request and reply messages between collectors on different nodes removes the blocking period from the requesting collector. However, until the reply is received, either of the invariants 2, 3, and 4 may be violated by a mutator accessing the remote object before the shading has taken effect.

The violation may only be possible from a mutator on the node that issued the request, as it is here we pretend that the shading has taken place. We may, thus, either withhold mutators from accessing objects currently being remotely shaded:

- on the requesting node (until a reply is received), or
- on the remote node (until the shading has taken effect).

Moreover, a mutator may cause objects and references to be moved or copied from one node to another. Thus the inconsistency introduced by delayed remote shading must be restricted to time intervals in which mutators do not communicate. This way consistency is achieved again before any mutator is able to recognize, that it has been violated.

An *object moving* between nodes must either be the top of an executing thread-of-control, i.e., a mutator, or an object known by a mutator. In the first case, the object is black and its references at least gray on the node it comes from. In the second case, the moving object was at least gray, but its references could be of any color. A *remote invocation* and a return from a remote invocation may be viewed as a moving mutator, with the parameters as part of its references. The remote object invoked or returned to is expected to be at least gray.

From the perspective of the receiving node, an arriving object must be alive, as it is actually moving. The object invoked or returned to from remote is also alive. Thus the receiving node need not receive an explicit shade request for these objects. During the mark-phase a *Mark-and-traverse* may be done for each arriving object and for each non-black object, invoked or returned to from another node.

All references moved across node-boundaries are at least gray, as they reside inside a moving object that is black. Due to the delayed shading of non-resident objects, an arriving reference may denote a resident object still not shaded here. Thus an explicit *Shade* of received references to resident objects must be done during the mark-phase to ensure that the object is at least gray.

Both the *Mark-and-traverse* and *Shade* must be done immediately when the object or reference is received, and before the requested action takes place on the receiving node. To facilitate efficient shading the *Mark-and-traverse* is also applied to the non-black moving objects on the sending side. Thus immediately shading the resident objects known by the moving object on the sending side also.

If the communication of objects and references between nodes during the mark-phase causes *Mark-and-traverse* or *Shade* automatically on both nodes, no mutator is able to see any inconsistency. The remaining shaded references to non-resident objects may be batched, as they do not violate the invariants, i.e., the mutator does not see them. To complete the mark-phase they must be shaded on the node hosting them. For this purpose, a register of references to these non-resident gray objects must be kept on each node. It needs to be adjusted when the non-resident objects become black. For each object, which is still registered, an explicit shade request must be sent to the node hosting the object. This node will reply if/when shading has taken place, thus provoking the reference to be de-registered. The non-resident object is then considered black by the requesting node.

Implementation of the Gray Color

The local collectors mark a reachable resident object gray and protect it when the first reference to the object is found. Non-resident reachable objects cannot be marked locally but a *non-resident gray set* of references to reachable, non-resident objects may hold this information until the asynchronous remote shade request/reply has succeeded.

Beside protection of a resident gray object, we need not mark the object gray, as the gray color is used only to identify objects that still need to be traversed. Therefore, a *resident gray set* may be used instead to keep the references to the resident gray objects.

During the mark-phase, each node runs a collector and each collector traverse the reachable objects one after the other. Each reference to a white object is followed to mark the object black and put the reference in the gray set as the object still needs to be traversed. The references to resident objects are put in the *resident gray set*, the others in the *non-resident gray set*. Furthermore, the objects of the resident set are protected.

3.7.2 No Gray Objects: Termination of the Mark-Phase

The mark-phase is finished when all nodes have emptied both their gray sets. Until the *all gray sets are empty* state has been reached, both the gray sets on each node may become non-empty due to incoming remote shade requests. Thus, termination of the mark-phase is not detectable locally, it requires global information. The status of the global gray set, i.e., the information about the cardinality of the union of all local resident and non-resident gray sets, may be implemented centralized or distributed.

A centralized representation is achieved by collecting information at one node, e.g., the coordinator node, about references that have been shaded. When their objects are Mark-end-traversed, the references are removed again from the central gray set. A distributed representation is achieved by collecting information about the gray objects on the nodes, where they are shaded as illustrated by the use of two gray sets on each node. Still, the information must be forwarded to the node hosting the object and back again, when the object is *Mark-and-traverse* in both cases as described in Section 3.7.1.

The centralized gray set may work as a clearing center to detect when the global gray set is empty. To do this, it must be informed about objects changing color. In principle, all changes of colors must be send to the clearing center. The previous mentioned set of non-resident shaded objects on each node could be moved to this clearing center also. Thus the explicit shade requests of non-resident objects would then be sent to the node hosting the objects via the clearing center.

The centralized service has the advantage that *global gray set empty* is easily detected when the last object is removed from the gray set. In case of a distributed gray set, a distributed termination detection protocol must be added. [Liskov 86] describes that this may be compromised by a logical centralized, but physical distributed, service. In a failure-free system (the premise of this chapter), we may rely on a centralized mechanism. Thus we use the approved *Coordinator node* and start a *Coordinator process* there.

A revised version of step 3.5 and the subroutine *Shade* of Algorithm 3 would then look like:

3.5 Until *Coordinator* is finished, do:

3.5.1 While there are more gray objects on this node, choose one and *Mark-and-traverse* that object.

3.5.2 Move the current non-resident gray set to the *Coordinator*.

Subroutine Shade:

If the reference is to a resident *white* object, mark the object *black*, put it in the *resident gray set*, and protect the object. If the reference is to a non-resident object, put it in the *non-resident gray set*, if it has not been put there already.

The role of the *Coordinator process* on the coordinator node could be expressed as:

Accumulate a global gray set of remotely referenced objects and forward *remote shade requests* from nodes sending their *non-resident gray set* to nodes hosting the referenced objects. Keep a list of nodes that have sent an *empty* non-resident gray set and have *not* been given any *shade requests* since then. When all nodes are in the list, the *Coordinator process* finishes by requesting all node-collectors to terminate their mark-phase.

3.7.3 Mark-During-Sweep by Smart Encoding of Black and White

The encoding of white and black has been located within the object, i.e., using a mark-field. Thus *Mark-and-traverse* is able to update the color (to black), traverse the object, and remove the protection by accessing nearby storage locations only.

The sweep-phase traverse the storage (presumably sequentially) to recycle the white objects and initialize the black objects for the next garbage collection cycle. During the sweep a white object may be garbage or alive, depending on whether the sweep has passed it. We may, however, choose not to reinitialize the color of the live (black) objects during the sweep. Instead, the encoding of black and white may be swapped. Thus during the sweep white objects are recycled, black objects are left unchanged and new objects are born black. When sweep is finished we simply swap the encoding of black and white. Thus all objects are now considered white and new objects are born white, until the next garbage collection cycle is

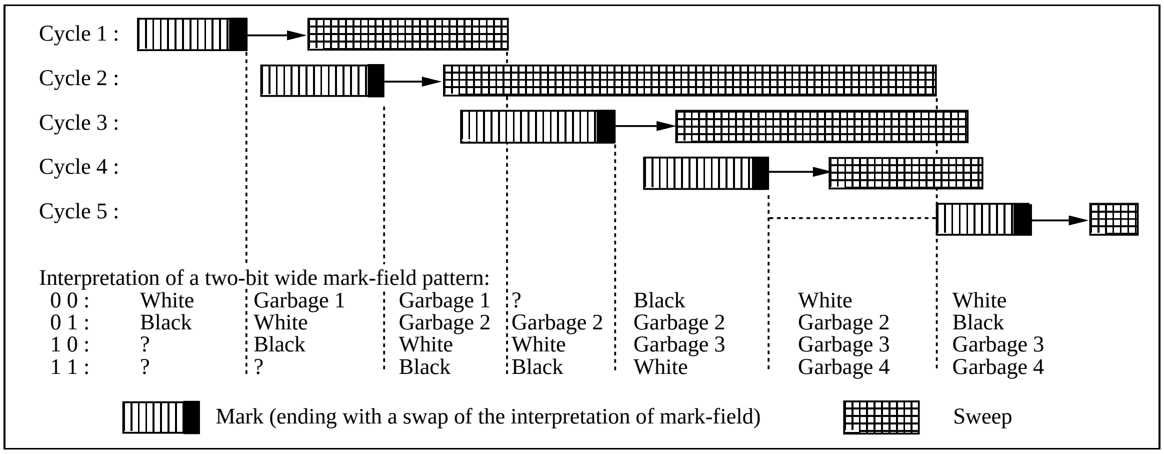


Figure 3.3: *Mark-during-sweep* scheme using a two-bit mark-field

instantiated. This scheme has the advantage that objects are born with a mark-field, which is only changed one time for each garbage collection cycle the object survives.

Queinnec, Beaudoing, and Queille describes [Queinnec 89] how the sweep-phase of a garbage collection cycle can be interleaved with the mark-phase of the following garbage collection cycle. Such a *mark-during-sweep* scheme may be accomplished by a simple coding scheme of the mark-field. Figure 3.3 shows an example of how the mark- and sweep-phase are interleaved by using a two-bit wide mark-field. The algorithm never changes the mark-field of garbage objects, thus the sweep process may work, even if it is delayed, and the next garbage collection cycle started. As the interpretation of the mark-field is swapped after each mark-phase, we need at least three values available, that may be interpreted as *white, black,* and *garbage* (that is previous white), respectively.

When the mark-phase is finished a concurrent *Sweep process* may run in parallel with the two next garbage collection cycles. As the mark-field is limit, a pattern interpreted as old garbage by a still running concurrent sweep process cannot be used as black in the following mark-phase. Thus a new garbage collection cycle must wait until all nodes has a free bit-pattern in their mark-field, that may be used for black. This is illustrated by the postponing of the fifth cycle in Figure 3.3.

We may take this idea a bit further. Instead of marking an object black, we can mark an object with the current garbage collection cycle number. Thus new objects and surviving objects are all marked with the current cycle, whereas objects marked with a lower cycle are garbage. Still the capacity of the mark-field limits the number of cycles the sweep-phase is behind the mark-phase. If the mark-field is two-bit, the scheme looks like that illustrated in Figure 3.3. It should be noted, however, that the swap of mark-field interpretation is partitioned. A new black-value is defined when a new garbage collection cycle is initiated and the previous black value is now white. The white value is changed to mean garbage when the mark-phase is finished. New objects are always allocated black.

Furthermore, instead of starting a new *Sweep process* each time a mark-phase is finished, we change step 3.7 of Algorithm 3 to update the interpretation of the mark-field and inform the sweep process accordingly. The sweep process traverse the object storage repeatedly and collects all garbage objects, i.e., objects marked with a cycle less that the cycle number of the last completed mark-phase.

Due to the limited width of the mark-field, the sweep process tells the collector when all garbage marked with the oldest cycle number has been collected. This value of the mark-field may then be reused. In the two-bit wide mark-field case, the collector must wait until *less than* three cycles remain to be taken care off by the sweep process.

Because *garbage stays garbage*, we may drop the sweep for the oldest garbage and reuse this identification as the new black mark. This enables the garbage detection to run whenever it wants to. The drawback is limited; the oldest garbage may not be reclaimed during the next two mark-phases, where the mark is reused; but after the completion of the second mark-phase the sweep process will again reclaim objects marked like the oldest garbage.

On the other hand, it seems inefficient to run a mark-phase without reclaiming the garbage afterwards. Instead of doing two mark-phases without an intermediate sweep of any garbage, the execution of the latest of the two mark-phases would give the same result, i.e., identifies exactly the same amount of garbage. Thus the mark-phase shall only be started, when the sweep process is close to its completion or later.

3.7.4 The Refined Algorithm

The refined version of the comprehensive, concurrent, and distributed garbage collector is shown as Algorithm 4. For completeness the processes, routines and handlers used by the algorithm is described shortly here again:

The Garbage Collection Fault Handler on each node:
: Suspend mutator while the object causing the fault is *Mark-and-traversed*. Finally the object is unprotected and the mutator resumed.

The Remote Shade Handler on each node:
: *Mark-and-traverse* the object and return an acknowledgment to the caller, which then removes the reference from its *non-resident gray set*.

The Sweep Process on each node
: Traverse the object storage (sequentially) and reclaim objects marked as garbage, i.e., mark with a cycle value less than the current value, *cycle*.

Coordinator Process on Coordinator node
: Accumulates a global gray set of remotely referenced objects and forwards *remote shade requests* from nodes sending their *non-resident gray set* to nodes hosting the referenced objects. Keeps a list of nodes, which have sent an *empty* non-resident gray set and have *not* been given any *shade requests* since then. When all nodes are in the list, the *Coordinator* finishes by requesting all node-collectors to terminate their mark-phase.

Mark-and-traverse an object
: Traverse the object to *Shade* all its references and if it is referenced from the gray set, remove that reference from the gray set.

Shade a reference
: If the reference is to a resident, *white* object, mark the object *black*, put it in the *resident gray set*, and protect the object. If the reference is to a non-resident object, the reference is put in the *non-resident gray set*, if it has not been put there previously.

On the mutators the algorithm imposes one restriction:

> Whenever a reference or object is crossing a node-boundary during the mark-phase, it must be shaded.

Algorithm 4 (Comprehensive)
The Garbage Collector on each node:

Initialize	4.1	Synchronize nodes and approve a coordinator node, which initiate a *Coordinator Process.*
	4.2	Wait until *Sweep Process* of $(cycle - 2)$ is finished, then synchronize with *Coordinator Process.* The current garbage collection cycle, *cycle*, is incremented and the new *black* defined for marked as well as new objects.
	4.3	Suspend all mutators on this node and prevent arriving mutators from other nodes from being started.
	4.4	Enable *Garbage Collection Fault Handler* for resident and arriving mutators, and *Remote Shade Handler* for incoming requests.
Mark-Phase	4.5	While there are more suspended mutators in the *resident root set*, choose one, *Mark-and-traverse* it, and resume it.
	4.6	Establishing a set of all supplementary roots of the object graph for this node and *Shade* these.
	4.7	Until the *Coordinator Process* is finished do:
	4.7.1	While there are more resident *gray* objects, choose one and *Mark-and-traverse* that object.
	4.7.2	Move the current *non-resident gray set* to the *Coordinator Process.*
Sweep-Phase	4.8	Run-time system tables are adjusted to reflect that *white* objects are now considered dead.
	4.9	Change the interpretation of the mark-field: $white \mapsto$ garbage of $(cycle)$ Inform the *Sweep Process*, that the marking of cycle, *cycle*, is finished.

A 4

3.8 Comprehensive Distributed Collection Summary

We have developed a distributed garbage collection algorithm by step-wise refinement of a simple *stop-mark-and-sweep* garbage collector. The generic version (Algorithm 1) works in a distributed system. It is comprehensive but neither efficient nor expedient. By using a *faulting mechanism*, concurrency is introduced (Algorithm 2). Thus, pauses in user processes due to garbage collection are limited. The overhead in communication by using a sequential garbage collector (even for the faulting version) is further limited by employing a sequential collector on each node and letting the collectors cooperate via a *Coordinator* (Algorithm 3). Finally, further optimizations have been thrown in to limit the storage traversal and doing garbage detection and garbage reclamation in parallel. The last version (Algorithm 4) also optimizes the organization of coloring information. Still, it depends on a node appointed as *Coordinator* that takes care of all live inter-node references.

Thus the final Algorithm 4 is:

Comprehensive by using mark-and-sweep to do a traversal of the entire graph of objects.

Concurrent by using a faulting mechanism, a sweep process, and cooperating node collectors.

Distributed by using cooperating node collectors and a centralized coordinator process.

The main deficiencies are the dependency on a *Coordinator* node, and that all nodes must be available all the time. The algorithm needs further refinements to overcome these. A solution which overcomes these and works in a system where partial failures may occur is discussed in the next chapter. More details on implementation in the Emerald prototype is presented in Chapter 5.

Chapter 4

The Design of a Robust Garbage Collector

This chapter contains the discussion on robustness of our distributed garbage collection scheme. Without giving up our goals in a failure-free distributed system (as described in Chapter 3), we now drop the assumption about the absence of failures and accept that individual nodes may be temporarily unavailable.

Our goal is to provide a robust collector that does not assume that all nodes are up simultaneously, and progress despite notes repeatedly crashing and restarting. Our collector is even capable of completing its job *even if it is never the case that all nodes are up simultaneous*, only pair-wise simultaneous up-time of the nodes is assumed.

First we describe in Section 4.1 the general failure model to which our garbage collector is robust and the simplified failure model adopted by our solution. The general goals of the garbage collection scheme are rephrased in Section 4.2, this time with emphasize on robustness. The garbage collection scheme that fulfills these goals are described and discussed in Section 4.3 and 4.4. The major design decision to make a robust, *system-wide*, distributed garbage collector for comprehensive collection work is to use distributed control where applicable (Section 4.3). The detailed discussion on each of the sub-problems in this solution is given in Section 4.4. A supplementary collector for more expedient collection, the *local* collector for each non-failed part of the system, is presented in Section 4.5. Finally, the full garbage collection scheme as proposed for the Emerald system is evaluated in Section 4.6.

As the previous chapter, solutions presented here are mostly applicable to any distributed system. However, the discussion is restricted to our specific distributed system, the Emerald system, where a concrete system is needed, e.g., where precise characteristics and requirements to the garbage collector from the surrounding system are needed.

4.1 A General Failure Model for Distributed Systems

Distributed systems may fail, like any system; but in contrast to ordinary sequential systems, distributed systems may fail partially. In a network of computers, communication between computers may introduce errors or failures, and computation on a single computer in the network may produce false results or stop completely for shorter, longer, or infinite periods.

The distributed system model, the failure model, and how robustness can be expressed in this context is described next. The presentation is restricted to or exemplified by our

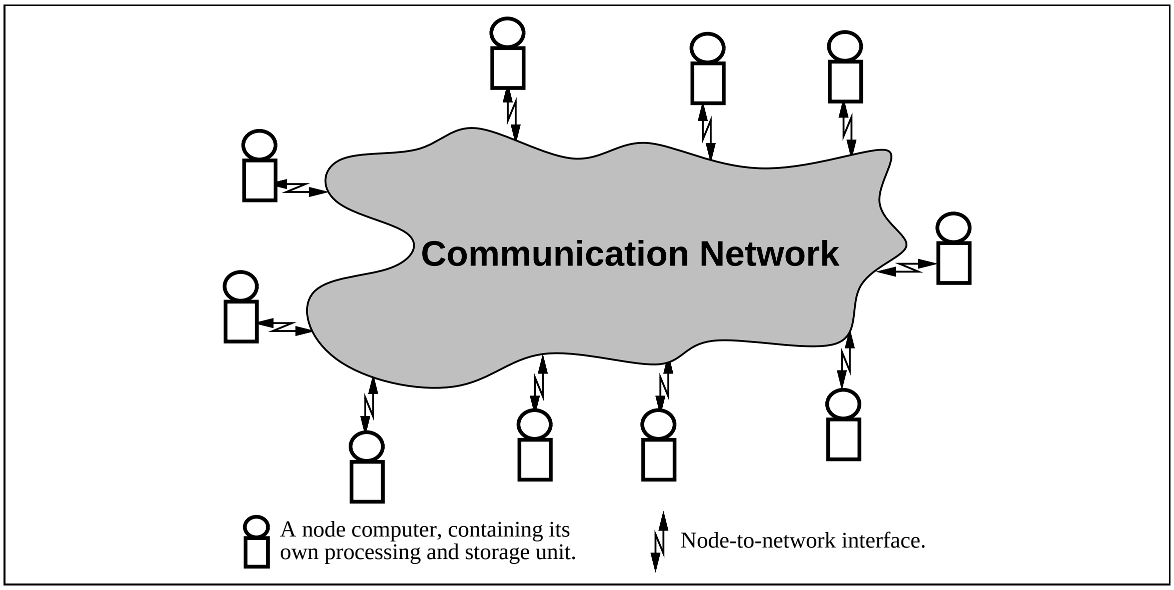


Figure 4.1: The distributed system model

Emerald prototype where applicable.

4.1.1 The Distributed System Model

Our model of a distributed system is shown at Figure 4.1. A distributed system is a network of n nodes with self-contained computers as nodes. Each node is connected to the other nodes by means of the network. The nodes have, however, no direct access to any part of the other nodes. Communication between nodes is done by sending messages from one node to another or by unreliable broadcasts from one node to all nodes in the network. An example of this distributed system model is workstations connected by a local-area network of the Ethernet-type on which they communicate by means of Internet-protocols.

Each node is a self-contained unit with at least:

- a *processing unit* (CPU),
- a *virtual memory*, based on primary memory and secondary disk storage, and
- a *network interface*, through which messages to/from all nodes may be communicated.

In addition, a node may have:

- a *stable storage* unit (disk), and
- a *user interface* (e.g. keyboard and display) or other means of interfaces to the external world of the distributed system.

The network connects the nodes in a way that supports full connectivity between the nodes in the absence of failures. The only assumption about the network is that connectivity is an *equivalence relation* even in case of node failures. From the nodes, the network is visible as a delay on node-to-node communication only. Thus, data may be in transit between two nodes without being available on either nodes.

4.1.2 The Failure Model

Our failure model is based on *fail-stop* semantics thus excluding more nasty failures like *Byzantine failures*. Each functional unit will either deliver the correct result or no result at all. Moreover, a failure may be permanent or temporary, thus a unit may be reset and start functioning again after a failure.

The units that may fail in a distributed system are either the node computers or the network connections between individual nodes. The network failures are modeled at each node as the ability of the node to communicate with each of the other nodes. Thus, it has a failure state for each of the $n-1$ other nodes independent of their state. This independence allows us to model that two nodes may both be running while being unable to communicate with each other due to a failure partitioning the network.

For node i the following units are either running or failed:

- The processor and its volatile storage viewed as one unit.
- $\forall j \neq i$: The communication link between node i and j.

The stable storage never fails, thus it may be used as a back-up media for the volatile storage between failures.

The failure state of the whole distributed system can be described by the following state variables:

$$\begin{array}{llll} \forall i \in \{1,2,...,n\}: & NodeState[i] & = \{ \; running \; | \; failed \; \} \\ & \forall j \in \{1,2,...,i\}: LinkState[j,i] & = \{ \; available \; | \; unavailable \; \} \end{array}$$

Less than half of the matrix *LinkState* is needed because of symmetry, $LinkState[j,i] = LinkState[i,j]$. The diagonal $LinkState[i,i]$ is equivalent to $NodeState[i]$. Thus at any time, t, the whole failure model can be modeled by a two-dimensional stochastic variable:

$$\forall i \in \{1,2,...,n\} \forall j \in \{1,2,...,i\}: \quad State_t[j,i] = \begin{cases} false & \text{i.e., service unavailable.} \\ true & \text{i.e., service available.} \end{cases}$$

Two nodes i and j $(i < j)$ may communicate at time t if neither node is failed nor the communication link between them, i.e.:

$$State_t[i,i] \wedge State_t[j,j] \wedge State_t[i,j]$$

The Emerald system is based upon a further simplified failure model. It assumes a reliable network, thus only nodes may fail. This assumption is enforced by the implementation of inter node communication by a reliable sliding window protocol. At any time, t, the simplified failure model is a stochastic variable per node:

$$\forall i \in \{1,2,...,n\}: \quad State_t[i] \quad = \begin{cases} false & \text{i.e., node } i \text{ has failed.} \\ true & \text{i.e., node } i \text{ is running.} \end{cases}$$

4.1.3 Robustness of a Distributed System

When failures occur, it must be decide what services remain available. On top of the distributed system model described in Section 4.1.1, the actual distributed system may want to cope with certain failures. This yields for the distributed system as such, but also for any supplementary application or extension like a garbage collector for the system.

A robust distributed system with our failure model should at least enable processing to continue in non-failed parts of the system as long as relevant data is available. The availability of data may be extended by having multiple copies of the same data on different nodes. This is not always possible, if data must be consistent. Another problem is failed nodes which may restart using data from other nodes or from their own stable storage.

The Emerald system enables robustness according to the simple failure model by:

replica, i.e., objects with an unchangeable state (called *immutable*) are copied on demand to all nodes that need the object.

checkpoint images, i.e., a snapshot of an object state is copied to one or more nodes and, optional, to their stable storage. These checkpoint images are passive copies, one[1] of which may take over the responsibilities of the no longer accessible original.

To ensure progress of the garbage collection process in a distributed system, the collector must be able to cope with node failures. Furthermore, it must respect the facilities that makes the system itself robust. It is important to note that we do not use robustness as an absolute term. A system robust to certain failures may not survive, if important data is lost. Thus the goal is to be robust to certain failures.

The Emerald system itself survives failures according to our simplified failure model, specifically, it is robust to nodes that crash and restart later. The checkpoint system enables Emerald applications to survive failures by using the special recovery and failure sections available at the programming language level. The simplified failure model ease the recovery of inaccessible Emerald objects because the model ensures that an inaccessible object will not become available by other means than recovery. In case of an unreliable network, during a network partitioning, an inaccessible object could have survived at an isolated node while the rest of the system decided to recover from an old checkpoint image of the object leading to inconsistency problems when the network connection became available later. Such problems must be solved when extending Emerald to run on wide-area networks, where our general failure model applies.

4.2 Goals for a Robust Garbage Collection

The goal is to design a *comprehensive* and *concurrent* garbage collection scheme for a potential *robust distributed* system, where garbage collection *preserves this robustness* while being absolute *robust itself*. The collector must not be *inefficient* and it must at least return garbage to the allocator with the same speed as the allocator creates new objects (*expedience*). Our goal with respect to robustness is to provide a garbage collector which is:

- robust to node failures and node recovery,
- assuming only pair-wise availability of nodes, and
- limited in assuming stable storage available.

[1] Recovery from a checkpoint image includes a protocol to ensure that only one copy of the most recent checkpoint image is reinstantiated.

4.2.1 Robustness Goals

To achieve robustness, conflicting goals may be compromised. In our case, it is possible to achieve all functional goals (see Section 1.5), including robustness, if performance, especially expedience, is sacrificed. Our primary goal in this chapter is such a collector. Secondary, if we compromise on comprehensiveness when faced with failures, we may achieve some expedience. Our secondary goal is to always collect some garbage (expedient) without abstaining from the primary goal, thus introducing a secondary, non-comprehensive—but more expedient—collector.

1. **Comprehensiveness and Robustness**
   Eventually all garbage must be collected. Thus information about the object graph on an unavailable node must be made available before the collector may determine which objects to collect. The garbage collector should, however, be able to progress as long as the nodes become available even though all nodes are never available concurrently. The garbage collector must be able to complete, if the nodes have been at least pair-wise available often enough.

2. **Expedience and Robustness**
   As comprehensiveness and temporary unavailability may delay the garbage detection for long periods, supplementary collections not depending on the unavailable nodes must be done to at least achieve some expedience in the delivery of reusable storage. This supplementary garbage collection scheme must work independently of node failures, and the two collection schemes must not block each other nor the running system. The supplementary collection cannot be comprehensive as it is based on partial information about the distributed object graph, thus it is supplementary to the comprehensive, but potentially slower, garbage collector.

4.2.2 Robustness and Design Restrictions

While the garbage collection scheme is robust and applies to the functional requirements, robustness must also be view with respect to the implementation of the collectors. Such design restrictions to ensure robustness are concerned with:

Concurrency and Robustness
The garbage collection must not block the running distributed system while it waits for access to a temporarily unavailable node. Thus it must preserve the concurrency achieved by the distributed garbage collector of Chapter 3 although some nodes are currently unavailable.

Respecting Facilities for Robustness
The facilities to enable robustness of the distributed system must be respected. Thus facilities, like checkpoint, must have the same functionality with garbage collection as they had without. The impact on garbage collection is that objects and references in checkpoint copies remains alive as long as the checkpoint copy may be used to reinstantiate an object. Thus, an object may be kept alive due to a reference originating in a checkpoint copy of a live object, although no live objects reference the object. By adding references from live objects to their checkpoint copies and the checkpoint copies into the object graph, the garbage collector will be able to respect these facilities for robustness.

Robustness to Dangling References

During a traversal of the object graph, the garbage collector may be faced with references to objects that are no longer available. If they are permanently lost—due to some previous failure—the references are no longer valid. Any user process trying to access the object will fail, e.g., in Emerald, such an invocation will return to the unavailable section of the caller. The garbage collector must also handle this situation or the emptying of the gray set will eventually be dead-locked by these references to unavailable objects.

Robust Implementation

While the garbage collector is running on a node, the node may fail and information about the state of the collection is lost. Thus, a robust collector must itself be distributed and avoiding centralized control. Furthermore, distributed control decrease the risks of creating a central bottle-neck that may sequentialize the whole garbage collector, thus loosing the potential concurrency available in a distributed system and gaining lower performance.

The goals about robust, comprehensive, concurrent, and distributed garbage collection have been achieved by extensions to the distributed garbage collector (Algorithm 4 in Section 3.7.4). The extensions are described in the next section. The expedience is due to our dual collector approach which is presented afterwards in Section 4.5.

4.3 A Garbage Collector without Centralized Control

A first step to make our non-robust distributed garbage collector (Algorithm 4, page 61) robust to node failures is the removal of the centralized control by the *Coordinator Process* on a *Coordinator Node*. This is a "heel of Achilles" as all inter-node communication is passed through the *Coordinator Process*. By applying distributed control we may, however, completely circumvent such a centralized design. This section discusses the problems and solutions when distributed control is used as the major design criteria to relinquish inter-node dependencies as much as possible.

4.3.1 Using Distributed Control

The problems in using distributed control are related to the inter-node synchronization points of the algorithm. According to the phases of the general mark-and-sweep algorithm these occur in:

Initialize How to decide that a new garbage collection cycle is needed and achieve consensus between all nodes when the decision has been taken.

Mark-Phase How to ensure that objects found via inter-node references are shaded consistently within the entire system.

Sweep-Phase How to detect that the mark-phase is finished globally and ensure consensus between all nodes.

The initial coordination may be done at each node when the node is ready, i.e., when the node has less than three outstanding cycles for sweeping by the *Sweep Processes*[2]. At this point, the collector informs all nodes about its ability to start a collection by broadcasting[3] a *Ready* message. These messages are used, together with local information from the storage allocator, to decide when a new garbage collection cycle is profitable.

When a ready node decides to initiate a new garbage collection cycle, it just starts the collection independently of the other nodes. A hint, about the decision, is, however, broadcasted to inform the other nodes, who might be ready to start immediately. A node that begins a new cycle on its own or remote hints, starts the collection by blocking all activities on the node. Thus, all resident mutators are suspended and all communication from other nodes are blocked. Blocking communication prevents unsuspended mutators at other nodes from changing the object graph on the blocking node and thereby invalidating the invariants. The establishing of an initial root set, the synchronization with the local storage allocator, and the enabling of the garbage collection fault handler are all done as in Algorithm 4. The whole initialize-phase is done atomically, i.e., as one indivisible operation on the node. At the end of the initialize-phase, the mark-phase is started and soon hereafter concurrency is available again.

The mark-phase starts by doing a *Mark-and-traverse* of the suspended mutators and resume each of them, as they have been traversed. Any message sent to another node is marked with the current garbage collection cycle number, which will ensure that the garbage collector is initiated on the other node before the message is handled. See Section 4.4.1 for more details on how the incoming messages are screened to ensure initialization of the garbage collector.

The blocked communication can be resumed with nodes in the same garbage collection cycle. We simply pretend that all other nodes are in the same garbage collection cycle and resume the blocked communication after having done a *Mark-and-traverse* of the objects in transfer. The tagging of all inter-node communication with the *cycle* of the sending node will ensure that the other nodes become aware of our pretension before the contents of the communication is processed. Thus, a node receiving a message tagged with a greater *cycle* than its own is forced to start its next garbage collection cycle to keep in pace. The cycle count must not be incremented faster than the collectors complete their former garbage collection cycles. The common synchronization point between the mark- and the sweep-phase ensures that one node can only be one cycle ahead of the others.

The major work is done during the remaining mark-phase, where gray objects are marked and traversed. As in Algorithm 4 the resident objects are marked and traversed locally as long as there are more resident objects in the gray set. Each non-resident gray object, that needs to be *shaded*, is processed by sending a *Shade request* to the node hosting it. Meanwhile incoming *Shade requests* for resident objects are processed as if those objects were in the resident gray set. In Section 4.4.2 the need to acknowledge each received *Shade request* is discussed further. References to objects which are not resident on any node must be identified and deleted from the non-resident set.

The mark-phase of the garbage collector is finished, when all nodes in the network have an empty gray set. The nodes need not be reachable simultaneous, as long as every node has been able to exchange its gray set of non-resident objects with the nodes hosting these objects. Thus the termination of the mark-phase is a distributed termination detection problem, i.e.,

[2] The maximum number of outstanding Sweep Processes acceptable depends on the width of the mark-field. Two is the maximum according to the chosen encoding in Algorithm 4.

[3] The semantics of a broadcast is an un-confirmed delivery of the message to all nodes on the network. Nodes are allowed to drop broadcasts, thus the sender knows nothing about what nodes received the message.

we need to detect, that the distributed gray set is empty or contains lost objects only. Such a detection can be done without centralizing the control. We present a variant of a distributed termination protocol that uses two-phase commitment in Section 4.4.3 to solve this problem.

The sweep-phase of Algorithm 4 already has complete distributed control, thus its global synchronization cannot be relaxed further. The garbage collection scheme is shown as Algorithm 5 in the next section.

4.3.2 The Collector with Distributed Control

The garbage collection scheme presented in Chapter 3 concluded with a comprehensive, concurrent and distributed algorithm (Algorithm 4). This algorithm is now refined to work in a distributed system with node failures and unavailable objects. The centralized control is distributed and synchronization points relaxed. The suspension of mutators is made independent of the availability of other nodes, thus the period a garbage collector is blocking other tasks is limited and independent of failures. We now present this garbage collector algorithm, Algorithm 5, as outlined in the previous section. Some of the details are discussed in the next section (Section 4.4), specifically the distributed termination algorithm is presented in Section 4.4.3.
For completeness, the processes, routines and handlers used by Algorithm 5 are described shortly here also.

Screening incoming messages on the i^{th} node:
: Each received message, *mess*, is screened to ensure that $cycle_{mess} = cycle_i$. If $cycle_{mess} > cycle_i$, the garbage collector is asked to start and the message is queued until the collector resumes communication (step 5.5).

The Garbage Collection Fault Handler on the i^{th} node:
: Suspend the mutator while calling *Mark-and-traverse* for the object causing the fault. Finally, the object is unprotected and the mutator resumed.

The Remote Shade Request Handler on the i^{th} node:
: Mark-and-traverse the objects in the request and return an acknowledgment to the collector on the calling node.

The Remote Shade Reply Handler on the i^{th} node:
: Remove the references from the *non-resident gray set*.

The Sweep Process on the i^{th} node:
: Traverse the object storage (sequentially) and reclaim objects marked as garbage, i.e., marked with a *cycle* less than the current *white*, if the collector is in its mark-phase, or less that the current *black*, i.e., include the newly identified garbage, otherwise.

Mark-and-traverse an object:
: Traverse an object to *Shade* all its references, and remove the references from the *resident gray set*, if it is in it.

Shade a reference:
: If the reference is to a resident, *white* object, mark the object *black*[4], put it in the *resident gray set*, and protect the object. If the reference is to a non-resident object, the reference is put in the *non-resident gray set*, if it has not been put there previously.

Algorithm 5 (Distributed Control)
The Garbage Collector on each node, i:

Initialize	5.1	Wait until the *Sweep Process* of $cycle-2$ has completed, and someone, including the *storage allocator* on this node, expresses its desire to start a new garbage collection cycle. Then announce that a new garbage collection cycle, $cycle_i + 1$, is started on node i ($cycle_i$ is not incremented yet).
	5.2	Suspend all mutators on this node and block incoming messages.
	5.3	Enable *Garbage Collection Fault Handler* for resident and arriving mutators and *Remove Shade Handlers* for incoming requests/replies.
	5.4	Increment $cycle_i$ and define the new *black* color to be used to color marked and new objects.
Mark-Phase	5.5	While there are more suspended mutators, choose one, *Mark-and-traverse* it, and resume it. For each of the blocked messages, *Shade* its references, *Mark-and-traverse* objects in the message, and unblock the message.
	5.6	Establishing a set of all additional roots of the object graph for this node and *Shade* these.
	5.7	Until *termination* is detected do:
	5.7.1	While there are more *resident gray* objects on this node, choose one and *Mark-and-traverse* that object.
	5.7.2	While there are more references to *non-resident gray* objects registered here, send a *Shade request* to the node hosting the objects.
Sweep-Phase	5.8	Run-time system tables are adjusted to reflect that *white* objects are now considered dead.
	5.9	Change the interpretation of the mark-field, as *white* objects are garbage now. Inform the *Sweep Process*, that mark of $cycle_i$ is finished. **A 5**

4.4 Details in the Robust Garbage Collector Design

Algorithm 5 presented a robust garbage collection scheme. Some of its mechanisms have not been discussed yet. This is done here together with an extension of the scheme to cope with facilities for robustness inside the system, i.e., how the validity of checkpoint images, replica, etc., is preserved.

The discussion is organized around the following issues:

1. Screening of incoming messages
2. Shading of remote requests
3. Distributed termination detection

[4] Gray objects may be marked black immediately as long their status of being gray is recorded by putting them in one of the gray sets (performance optimization).

4. Considerations for robustness facilities of the system
5. The use of stable storage

4.4.1 Screening All Incoming Messages

Incoming messages, containing moving objects or references, are all tagged with the *garbage collection cycle number* of the sending node ($cycle_{message} = cycle_{sender}$). All messages between nodes are tagged independent of their level of abstraction (Emerald objects, system, or garbage collector messages) and independent of the message being a request or a reply. The received $cycle_{message}$ is compared to the garbage collection cycle number of the node receiving the message ($cycle_{receiver}$) and used according to the current status of its garbage collector to ensure that the two nodes share a common view on the garbage collection, i.e., running the same *cycle*. At least three different situations may be identified:

1. **The receiver and sender are running the same garbage collection cycle**
   ($cycle_{message} = cycle_{receiver}$)
   This should be the case most of the time, both during and between garbage collections. No special action is needed.

2. **The receiver is running a newer garbage collection cycle than the sender**
   ($cycle_{message} < cycle_{receiver}$)
   This situation arises the first time another node sends a message to a node that has initiated a new garbage collection cycle if the other node has not heard that the new collection was initiated. The only action needed is to force the sender to run in step, which will eventually happens by the first message send in the opposite direction. Thus, the message is simply handled as if the sender was already running the same garbage collection cycle.

3. **The receiver has not started this garbage collection cycle yet**
   ($cycle_{message} > cycle_{receiver}$)
   The receiver may be waiting to start this (or a previous) garbage collection cycle, or it may be running in a previous cycle. This is the situation that eventually brings the receiving node to run in step with the sending. The action taken is simply to initialize the garbage collection cycle number, $cycle_{message}$ on the receiving node.

It is important, though, that the initialization of a new garbage collection cycle in situation 3 is possible and done instantaneously, i.e., both fast and blocking, as it is done at the arrival of a message which may need to be handled before the sender gets the impression that the message has been lost. The initialization is blocking from step 5.2 to 5.5; but all actions taken during this are fast. The remaining problem is to ensure that a new garbage collection cycle may be initiated in the cases where it is not ready to start in step 5.1. This would be the case if the Sweeper was more than two cycles behind or the collector was already running but in a lower *cycle*. Sweeping of old garbage may be skipped as this garbage will be found again later, but what if the collector is running an old mark-phase? Such a situation should not appear as the mark-phase is terminated at a global synchronization point, and thus all nodes have agreed that it was finished. In any case, the old mark-phase may be skipped together with the associated sweep-phase. The only problem would be if too many sweep-phases are outstanding. Then the oldest sweep may be discarded as the mark-field of the objects has a limit field width and thus must be reused. This is, however, more a symptom of a too slow implementation of the Sweep process than a problem of the algorithm in general. Under such circumstances more storage is immediately reclaimable by the Sweep process, thus there is no need to instantiate the next mark-phase from a node-local point of view. Still the next mark-phase may be needed desperately by another node, in which case the oldest sweep can be safely discarded to enable the other nodes to progress and terminate.

4.4.2 Remote Shading Requests

To limit the communication overhead each node accumulates *Shade requests* for non-resident objects until all resident gray objects have been marked and traversed. Then the other nodes are informed about the need to *Shade* and eventually *Mark-and-traverse* these references.

When all references to non-resident gray objects have been accumulated and the resident gray set is empty, the *remote shading protocol* is initiated. The protocol is based on broadcasting. Each node receiving the broadcasted gray set checks, if any of the references are to a resident object. An immediately response, a *Shade reply*, is sent back to the broadcasting node confirming the shading of the resident objects. Furthermore, these objects are *Shaded* to make sure that they become black eventually, and at least before they are used.

The broadcast protocol is made reliable by using the *Shade replies* as acknowledgments and repeating the broadcasting of the non-acknowledged references, with increasing delays between the repeats. This ensures that nodes not receiving the broadcast while they are temporarily unavailable will get the message sooner or later.

The repeated broadcasting is continued until the last *Shade reply* has arrived. In this way *Shade requests* that are not acknowledged at all, are broadcasted again and again until a responsible node wants to take over the request. No request is lost, in the sense that the requester holds the reference until another node takes over.

References to permanently unavailable objects may invalidate this protocol, if they slip into the non-resident gray set, thus preventing the protocol from terminating. As references may become dangling after their entrance into the non-resident set it is not enough to prevent dangling references from entering the set. Thus either these references must be removed from the non-resident set by special consideration, e.g., by checking their availability from time to time, or the repetition of broadcast must be terminated by other means. By use of a standard *location protocol*, references that are not removed from the non-resident set by *Shade replies*, may be identified as references to either lost or living objects. If lost, they are removed from the non-resident gray set, otherwise we have to wait for a *Shade reply*, thus continuing the broadcasts.

In most cases the *Shade request* is acknowledged immediately by the node hosting the object. This node *Shade* the resident object after having send the *Shade reply*. The scheme works independent of the actual color of the object, even for an already black object, the node hosting the object will answer with a *Shade reply*. Moreover, *Shade requests* may also be acknowledged with a *Shade reply* by a node that knows the object referenced as black although the object resides somewhere else.

4.4.3 Termination of a Distributed Mark-Phase

Each node has two sets of references to gray objects, the *resident gray set* and the *non-resident gray set*. Thus we have a distributed gray set represented by two sets on each node.

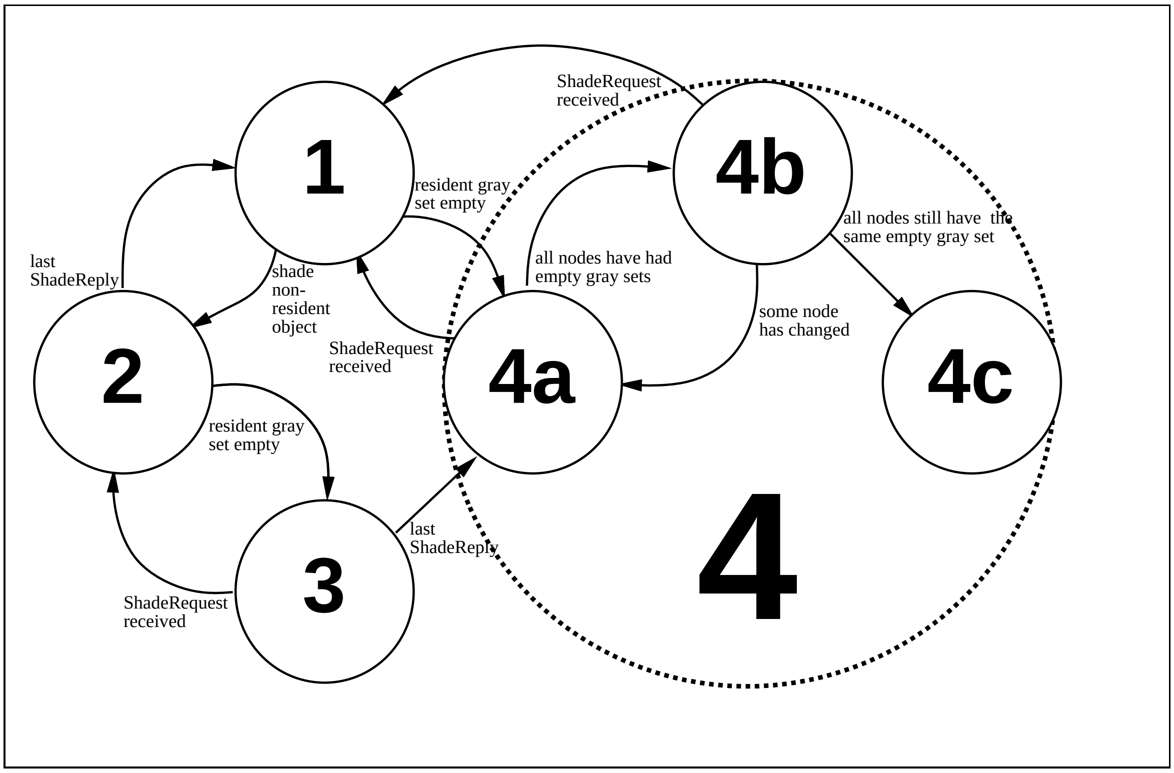


Figure 4.2: The states and transitions of Algorithm 5 running on each node

During the mark-phase each node may be in one of four states depending on its two gray sets:

State	The resident gray set	The non-resident gray set
1	non-empty	empty
2	non-empty	non-empty
3	empty	non-empty
4	empty	empty or dangling references only

During state 1 and 2, the local *Mark-and-traverse* continues (step 5.7.1 in Algorithm 5). During state 3 remote *Shade requests* are send out (step 5.7.2 in Algorithm 5). When state 4 is reached the mark-phase *may* be finished. However, the state is not stable, as another node may send this node a *Shade request*, thus changing the state of this node. Still, global termination is detectable, as the global state, *all gray sets empty*, is stable. The mark-phase is finished when all nodes have exhausted both their gray sets (state 4). A distributed termination protocol is needed to detect this global state. The above states are illustrated on Figure 4.2 with state 4 sub-divided into its three sub-states of the distributed termination protocol. Each event which triggers a transition from one state to another is also illustrated.

The obligation of the distributed termination protocol is to detect the stable state, *all nodes have finished*, i.e., emptied both their gray sets. The task is complicated by the fact that any node may be unavailable for any number of times in any length of time. Also, the dangling references to permanently unavailable objects must be identified (and removed from the gray sets). Furthermore, the global state is built by combining the local states from each node, while these local states are unstable.

When looking for efficient and robust solutions, the following observations are important to stress:

1. The nodes, constituting the system, must be known to determine, what *all nodes* means.
2. Information from *all nodes* is necessary to detect the global state.
3. By using *distributed control*, we have precluded the use of a specific *coordinator node*.
4. Due to failures, the collection of information may be suspended anywhere in the process, any number of times.
5. References to non-resident objects may become dangling at any time.
6. The object graph may include objects in a singly linked list crossing from node to node any number of times up to the order of the number of objects. Thus the local state, *both gray sets empty*, may be reached many times on each node (O(number of objects) in worst case).

On this background we have decided to let the nodes exchange information in an anarchist style. No leader is approved, nor elected. Due to the risk of loosing information, the communication is based on retransmissions with increasing delays. The delays are prolonged or diminished according to the local available information on each node.

Due to observation 5. above, we cannot just wait for the non-resident sets on each node to become empty. Hence, the dangling references must be identified and removed from the non-resident set during state 3 or 4. This may be done when no more *Shade replies* is received, or all nodes must agree on these references to be dangling. Whether to remove them from the non-resident set or globally agree on their existence is a matter of implementation. In the following, we assume that they are identified as references to unavailable objects and hence removed from the non-resident set. Furthermore, observation 6. above suggests that it is not enough to collect the states of emptiness from all nodes once. A non-empty node may become empty after it has "passed the buck" to a previous empty node, which now becomes non-empty again. The communication scheme described in Section 4.4.2 ensures, however, that the two nodes are never empty concurrently, while passing the buck (*Shade request/reply*) between them.

This has lead us to a *two-phase commit protocol.* The protocol is illustrated by the division of state 4 in three internal states: 4a, 4b, and 4c on Figure 4.2. It may be started by any node, that has entered state 4 (4a), i.e., any node with both its gray sets empty. The protocol takes advantages of a timestamp on each node. The timestamp counts the number of times state 4 has been reached, i.e., it is a counter which is incremented each time *both gray sets are empty* is reached. During the first phase (while in state 4a on Figure 4.2), we try to reach a situation, where we believe that the global state has been reached, i.e., all nodes have been empty and nothing seems to indicate that this is no longer true. The first phase must collect information about the nodes including the timestamp from each node, to be used in the second phase. The second phase checks (in state 4b on Figure 4.2), that all nodes have stayed empty since the end of phase 1, i.e., for each node both gray sets are still empty and their timestamps are unchanged. If this is true, the global state (state 4c on Figure 4.2) is reached, otherwise a new phase 1, followed by 2, must be initiated.

It is important that the timestamps used for verification in phase 2 are original, i.e., sent from their node or origin. However, during phase 1, available information about other nodes may be send together with the timestamps originating from the sending node. This helps new timestamps to be spread to the other nodes very fast. Thus the available space in the message should be loaded with the most recently received updates of timestamps on this node. During verification in phase 2, the added timestamps from other nodes may invalidate the verification (if they have been incremented); but they cannot be used as the acknowledgment from their node (if they are unchanged). The acknowledgment must be sent from the node of origin.

4.4.4 Checkpoint Images Extend the Object Graph

From a garbage collection point of view a checkpointed object may exist in any number of potentially different passive copies of the same object, *the checkpoint images*. Thus a reference to a checkpointed object is a reference to both the original object and the passive copies. The copies are all alike, but they may be slightly out of date compared to the original object.

The garbage collector needs to accept both the original and the copies as reachable when meeting a live reference to such an object. Thus the references to the copies are all followed, i.e., mark-and-traverse to ensure shading of all objects reachable from the checkpoint image. Furthermore, unmarked checkpoint images are garbage, just like ordinary objects are garbage, if they do not get marked.

No special action is needed to handle checkpointed objects, other than following these extra references. The traversal of the checkpoint images needs special attention though, as these objects are often represented in a linear and compact form suited for communication among nodes.

The recovery of a failed object by using an old copy, i.e., a checkpoint image, must take care of wrong marks in the mark-field of the copy. Thus the recovered object should be marked like a new object when recovered.

4.4.5 A Failure-Robust Collector

An important property of our algorithm is that all nodes need not be available at the same time as long as they are pair-wise available and able to communicate with each other when needed. If one of the nodes fails and restarts during garbage collection, the global termination detection ensures that we will wait for this node, but the node itself should also be aware of its own status with respect to garbage collection when it is restarted. To ensure correct restart, the garbage collector must store its own state information on stable storage and a restarted node must read this information and recover the garbage collector before it interacts with the other nodes.

The primary state information is the garbage collection cycle number and the position inside that cycle. If the collector was started but had not reached the global termination point, i.e., started the sweep-phase, it may simply be started from step 5.1 again. We only need to guarantee that no object is reclaimed during this cycle and that all reachable, non-resident objects are being shaded. At restart, all objects are black by default, thus the re-instantiation of the garbage collector is just done to gather the non-resident gray set and ensure that the other nodes do not collect these objects.

If the garbage collector was in the sweep-phase, when the node failed, the information about which objects to reclaim is lost and the recovered collector terminates its current cycle immediately. The failure has, however, made an almost complete storage reclamation as only recoverable objects are present after the failure. Thus no storage is lost from a garbage collector point of view.

4.5 Supplementary Collectors

The temporary unavailability of some nodes may prolong the time where the global garbage collection scheme must wait before it is able to reclaim garbage. To be able to collect garbage on a node, when some of the other nodes are not available, we use a *supplementary, local collector*.

The primary goal of the supplementary collector is relative expedience, i.e., it must be able to reclaim garbage independent of long-term unavailable nodes. Such a collector could run like the global collector of Section 4.4 but on a subset of available nodes only. References to objects in the subset from other nodes must go into the root set, to prevent objects, kept alive by such references only, from being garbage collected. These external references must be available independent of the current availability of the nodes hosting them, and all nodes in the subset of nodes must be immediately available.

The partitioning of the distributed system in such "independent" subsets must ensure that each subset is inter-connected long enough to do a full garbage collection cycle and reclaim garbage in a tempo comparable to the tempo of the allocator. As many references tend to be short lived and local [Lieberman 83, Schelvis 88, Jul 88b], a good candidate to such a partitioning is *one node per subset.* Thus each node should employ an independent local garbage collector. In fact, the delay imposed by any inter-node communication suggests the need for a single node supplementary collector on each node.

4.5.1 The Local Collector

The local collector is based on Algorithm 4 modified to work independent on each node with node-local information only. To prevent it from collecting objects only referenced outside the current node, we extend the root set with objects, for which a reference has been given to another node in the past. This information is held as a *ReferenceGivenOut*-bit in the descriptor of those objects. The bit is set the first time a reference to the object is made available outside the current node. There may not be any outside references to the object any longer, i.e., the bit is never cleared as no global reference count is held for the object. Thus it is not locally deterministic whether the object is still referenced from another node or not. For the local collector, we use the conservative approach and regard potentially referenced objects as reachable.

As the local collector works node-local only, references to non-resident objects have no meaning and they are simply skipped when they should be shaded according to Algorithm 4. The objects referenced by these references will not be reclaimed due to this, as they are already in the root set of the nodes hosting them. Moreover, no inter-node communication is needed. The algorithm for the local collector is shown as Algorithm 6.

Besides a new, local variant of the Shade routine, the *Sweep Process*, routine *Mark-and-traverse*, and the handler *Garbage Collection Fault Handler* of Algorithm 5 are reused in the local collector also.

Shade a reference

> If the reference is to a resident, *white* object, mark the object *black*, put it in the *gray set*, and protect the object.

Algorithm 6 (Local)
The Local Garbage Collector on each node:

Initialize	6.1	Wait until *Sweep Process* of ($cycle - 2$) has finished and the global collector has an empty gray set on this node.
	6.2	The current garbage collection cycle, *cycle*, is incremented and the new *black* defined for marked as well as new objects.
	6.3	Suspend all mutators on this node and prevent arriving mutators from other nodes from being started.
	6.4	Enable *Garbage Collection Fault Handler* for resident and arriving mutators.
Mark-Phase	6.5	While there are more suspended mutators, choose one, *Mark-and-traverse* it, and resume it.
	6.6	Establishing a set of all supplementary roots of the object graph for this node including objects potentially known outside this node (*ReferenceGivenOut*-bit) and *Shade* these.
	6.7	While there are more *gray* objects, choose one and *Mark-and-traverse* that object.
Sweep-Phase	6.8	Run-time system tables are adjusted to reflect that *white* objects are now considered dead.
	6.9	Change the interpretation of the mark-field: *white* $\mapsto$ garbage of (*cycle*) Inform the *Sweep Process*, that the marking of cycle, *cycle*, has finished.

A 6

The local garbage collector will collect the "local garbage", i.e., objects, that has never been referenced outside the node. We expect most garbage to be "local garbage", thus the scheme supports an expedient collection of most garbage.

The local collector has no need to cope with failures in the distributed system. It is local to each node, as each node runs this supplementary local collector. The main problem is how the global and the local collector on a node can cooperate. We want neither to depend on the other, specifically it is crucial that the local collector can run undisturbed while the global is blocked, e.g., by failures.

4.5.2 Synchronization between Global and Local Collector

To achieve independence from the global collector, the local collector works with its own set of mark-fields for each object, but the *garbage collection fault mechanism* is used by both, only the reason to protect is different which is captured by separate protection-fields as well.

As seen from one collector, the other collector is just another mutator. The collectors are, however, not respecting the invariants, when viewed as mutators. They change the object graph when they reclaim garbage. Thus we want to prevent one collector from reclaiming garbage, which—due to latency—is known by the other collector as non-garbage.

The two collectors on the same node must synchronize their actions

A simple synchronization between the global and local garbage collectors on each node may be achieved by not reclaiming garbage in the sweep-phase of the global collector. Instead, the global collector clears the *ReferenceGivenOut*-bit for the garbage objects [Bennett 87]. The local collector will then be able to collect the garbage next time it runs. This is, however, not enough as a delayed global collector may later try to mark-and-traverse an object already reclaimed by the local collector. Thus a more secure scheme is needed.

In general, the problem is that the two collectors may both get a reference to an object, regarded as garbage by the other. Both collectors consider objects reachable from their root set as non-garbage. The local collector has a local root set only and runs locally only, thus it has a comparably shorter run-time and it will not block due to failures or other external events. With this in mind, we may allow the global collector to be blocked by the local, while the local does its mark-phase.

During the mark-phase the global collector *pulsates*, i.e., its gray set of resident object shifts between empty and non-empty. We name the period from the set becomes non-empty until it is emptied again a *pulse*. The first pulse includes the gathering of all the local root objects, whereas the rest of the pulses are triggered by an incoming *Shade request*. The local collector is viewed as one such pulse also. Its work is equivalent to the first pulse of the global collector.

To ensure that the two collectors do not get objects that are considered garbage by the other in their gray set, only one pulse must proceed at a time. This means that while the local collector is running its pulse, no new pulse of the global is allowed. Conversely a local collector is not started while the global is working in one of its pulses. The underlying constraint here is that objects already identified as garbage by either collectors will not become member of the root set of a future pulse. As *garbage stay garbage*, this constraint is fulfilled by only initiating a root set of a pulse while the pulse of the other is done.

The delays introduced inside this scheme is limited, as both collectors depends on node local information only during each pulse.

By only initiating the local gray set when the global is empty (from a node local point of view) those other objects later traversed by the global collector must be reachable from an object known externally. Thus not collected by the local collector, as these are already in the extended local root set.

The local garbage collector will only collect garbage objects that has never been known outside the node. The garbage collector uses the faulting scheme, so the system has only to be stopped, during the initialize-phase. The marking-phase proceed in parallel with other activities.

The reclamation must be synchronized also, but this is easily achievable by employing a common *Sweep Process*. The Sweep Process is given information each time a mark-phase is terminated about the bit-pattern of the mark-field that means garbage now. This works for both the local and the global mark-fields. Alternatively, we could revert to Bennett's scheme, by making objects identified as global garbage, local, thus only reclaiming in the local collector.

4.6 Robust Collection Summary

We have presented a robust solution to the distributed garbage collection problem. Our solution does not sacrifices the other goals of being comprehensive and concurrent. Moreover, it is able to collect garbage in a partially failed system and even complete a comprehensive collection without the entire system being available simultaneously.

As described the Emerald garbage collection scheme consists of a global mark-and-sweep collector on each node doing a comprehensive garbage detection for the entire system. Algorithm 5 describes how this is achieved in a concurrent fashion, robust to temporary unavailable nodes. The garbage collection scheme further employs a local mark-and-sweep garbage detection as described by Algorithm 6 on each node for expedient detection of local garbage. Both detectors on each node use the same *Sweep process* to reclaim the detected garbage.

The global collectors constitute a distributed collector which is:

Comprehensive by using mark-and-sweep to do a traversal of the entire graph of objects.

Concurrent by using a faulting mechanism and a sweep process.

Robust by using distributed control for the cooperating node collectors and a distributed termination detection algorithm to end the global mark-phase.

The local collectors are independent node-local collectors which collect garbage *expedient* and *concurrent*, thus making the entire scheme more robust. Likewise, does the robustness of the communication method of the global collection make the global collection extremely tolerant to nodes crashing and restarting during the collection.

Chapter 5

The Implementation of Garbage Collection in Emerald

The distributed, robust collector and its derived local version have been implemented, tested, and measured. The main implementation problems and their Emerald solutions are discussed in this chapter. The design of the Emerald garbage collection scheme was described in the previous chapters. The implementation is based on the algorithms presented in chapter 4, i.e., the global garbage collector (Algorithm 5) and the node-local garbage collectors (Algorithm 6). An overview of the implementation and the garbage collection scheme is given in Section 5.1, whereas the full specification of the implemented garbage collection scheme is given in Appendix B.

Based on a description of the main problems encountered (Section 5.2), we discuss the solutions applied to get a working implementation in the following sections. These are divided into three groups:

Major policies implemented in the Emerald garbage collectors (Section 5.3).

Primary mechanisms implemented to do garbage collection in Emerald (Section 5.4).

Cooperation with the existing Emerald kernel, especially the storage allocator (Section 5.5).

5.1 The Modules of the Implementation

The full garbage collection scheme in Emerald consists of two sets of collectors. One set of collectors cooperates on a global collection while the other set of collectors does an independent node-local collection on each node. The global collection is comprehensive but potentially slow, whereas the node-local is more expedient but conservative. The two collectors on each node are called the *global* and the *local garbage collector*, respectively. Their basic algorithm is the same; but the root set is larger in the local garbage collector, whereas the global must facilitate inter-node cooperation and distributed termination detection.

Both collectors are divided into a *marker* and a *sweeper* process, plus additional processes acting as *event handlers* driven by external events. Moreover the local and global sweeper processes are put together as one process interleaved with the storage allocator on each node. Thus on each node we have the same set of processes and event handlers constituting the whole garbage collection scheme.

The modules on one node are illustrated on Figure 5.1 with arrows indicating their interaction. Their algorithms have already been presented in the two previous chapters. To

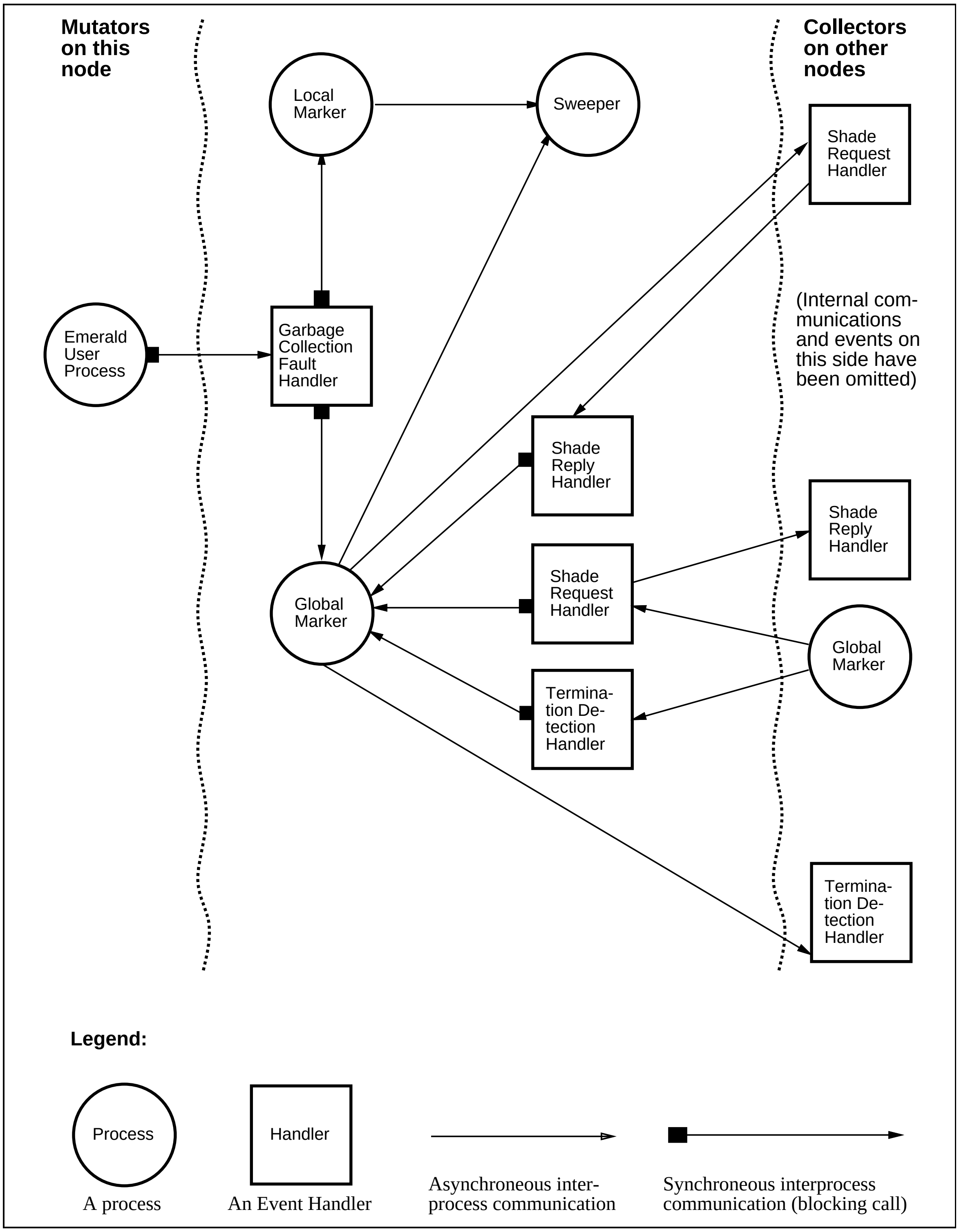


Figure 5.1: The garbage collection processes and handlers as seen from one node

summarize, on each node the complete Emerald garbage collection scheme is based on these modules:

Processes:

The Local Marker Process, as described by Algorithm 6 on page 78.

The Global Marker Process, as described by Algorithm 5 on page 71.

The Sweeper Process, as described by Algorithm 5 on page 70.

Event Handlers:

The Garbage Collection Fault Handler, as described by Algorithm 5 on page 70.

The Shade Request Handler, as described by Algorithm 5 on page 70.

The Shade Reply Handler, as described by Algorithm 5 on page 70.

The Termination Detection Handler, as described in Section 4.4.3 on page 73.

5.2 Implementation Problems

The implementation of the garbage collection scheme in Emerald has much in common with implementation of distributed applications in general. The garbage collector needs concurrent processing capabilities and associated mechanisms for synchronization and communication. Also, inter-node communication with fault-tolerance and distributed termination detection are needed to implement distributed garbage collection. Inter-node communication is needed for the start and termination of the global collector and to shade remote objects during the collection. The end of the mark-phase is found by a distributed termination detection protocol using two-phase commitment.

Furthermore, a garbage collector must fit into the memory management scheme of the system it serves. Objects on the basic allocation level (chunks of storage) must be identifiable by the sweep process in its search for the reclaimable ones. Low level deallocation must also be available to return the reclaimed objects to the allocator.

Cooperation on a higher level is also needed, as the graph of references between objects must be exactly identified. In Emerald, all allocated objects are tagged with information about their implementation type. The storage layout of user defined parts of objects and their activation records is also available in the form of templates associated with the objects (see [Jul 88a]). We also need to define the root set of objects and references among objects exactly, and to determine how the traversal of the Emerald object graph shall be organized.

The implementation of garbage collection in Emerald is faced with three kinds of problems:

Implementation of policies, that is, adapting the policies inherit in the algorithms to the run-time system of Emerald. This includes the inter-node communication protocols and, especially, the distributed termination detection protocol used at the end of the mark-phase. The exact identification of references between objects, the order of traversal through the graph of objects, and the amount of kernel data in the root sets are also determined by implemented policies.

Implementation of mechanisms, that is, adding or modifying internal functions inside the kernel to support the garbage collectors. This includes augmenting the existing faulting mechanism to take care of garbage collection faults, facilities for parallel processing of garbage collection processes, and inter-node communication facilities.

Cooperation between kernel and collector, especially the cooperation with the storage allocator, is needed to decide when to start and not to start a local or global collector. The same holds when and what to sweep and the internal adjustment of tables in the run-time system according to the result of the mark-phase. To start and stop a global collector, knowledge about *all* nodes is needed. Thus the garbage collector must cooperate with the existing host management system (HOTS) of Emerald to achieve an up-to-date picture of the global state.

We discuss these problems in the following three sections.

5.3 Implementing Garbage Collection Policies in Emerald

The major policies implemented to do garbage collection in Emerald are concerned with either global communication or the object graph traversal. We have implemented a protocol for communication and synchronization between the global collector on each node. Communication is needed when the collection is started, to follow inter-node references, and to detect when the mark-phase is finished. The order chosen to traverse the object graph must take the taxonomy of the implementation of the object graph into account also. The traversal takes its offset in the root set, containing the running Emerald processes and other *always present* objects. To implement the collector in Emerald, we must identify these objects among all those references available in the kernel, to find a small, yet large enough to be complete, set of root objects.

5.3.1 Inter-node Communication Protocols

Inter-node communication is needed between the global collectors to:

- ensure synchronization between the collectors, as these must run in step, and
- exchange information about remote shading.

Furthermore, each collector may need to communicate with all other nodes to determine that a reference is dangling.

Synchronization between the global collectors is not an absolute requirement; but they have to run in step relative to a virtual global clock, i.e., two global collectors need only agree when their nodes exchange information. Explicit synchronization takes place when:

- a node starts a collection (broadcasting its intention),
- shading information is exchanged,
- termination of the mark-phase is deleted.

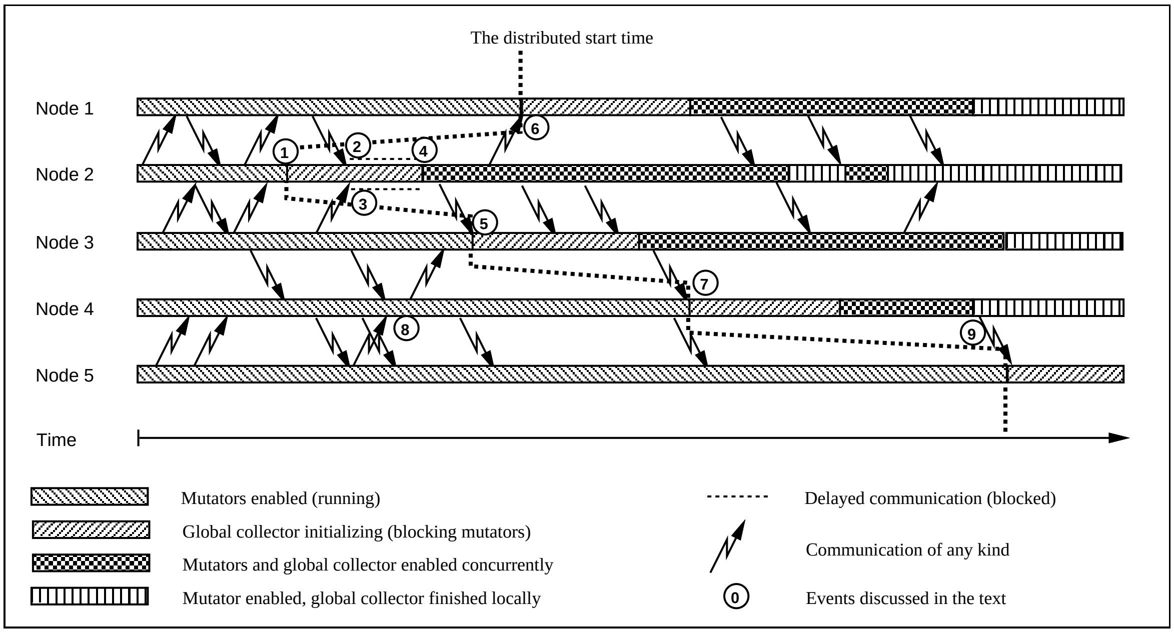


Figure 5.2: Synchronizing the start of the global collector

5.3.2 Synchronizing the Start of a Global Collection

The start of a global collection must *not* be a hard synchronization point, as a robust collection shall be able to start even though some nodes are temporarily inaccessible. Thus each node must be able to start its global collector independently, as long as those nodes started do not communicate with those nodes not started yet. We achieve this by the tagging of all communication with the value of the current *garbage collection cycle* of the sending node, combined with screening of all received messages. By implementing this tagging and screening of messages in the low level part of the message handling on all nodes, it is ensured that any message from a started node to a non-started will be deferred long enough to let the receiving node start its global collector before the message is handled.

Example: Communicating Nodes Forced to Synchronize

Figure 5.2 shows how the start of a new garbage collection cycle spreads slowly to all nodes in the system in a case where the initially broadcasted information about the start of a new collection never reaches the other nodes.

Node 2 initiates the cycle (1) and defers its processing of messages received from node 1 (2) and node 3 (3) at least until the mark-phase has started (4), i.e., along the thin-dotted line from 2 to 4, and from 3 to 4. In Emerald, the incoming messages may be processed invisibly to the sender, i.e., the acknowledgment is not sent back until the node allows mutators to run in parallel with the marker (4). From a node that has started a cycle, any outgoing message triggers the other still not initiated nodes (5, 6, and 7). At any moment, nodes that have not started on this collection may communicate as if the global collection was still not started, e.g., node 4 and 5 (8). All nodes must also synchronize at the end of the mark-phase to detect termination, thus, if a node has not communicated with those started ones before this point (9) it will be forced to start now, i.e., before global termination is detected.

The exchange of shading messages between the nodes may—like any other messages—ensure that a garbage collector is running on the receiving node in the same cycle as the sending node. This loose node synchronization ensures that all nodes will eventually participate in the same garbage collection cycle even in the case of temporary node failures. However, the initial broadcasting of the intention from a node starting a new cycle will in most cases—where the message is not lost—ensure full synchronization efficiently and immediately.

The protocol impose an overhead to all communication by adding the cycle value in all send operations and testing the value of the cycle in all receive operations. The overhead is, however, insignificant compared to the general cost of communicating across a local-area network in the Emerald system.

5.3.3 Remote Shading Policies

The remote shading is needed during the mark-phase to ensure that references from an object on one node to an object on another node is followed and the destination object marked and traversed. Thus we must locate the destination of the object, e.g., by using the Emerald location protocol, and send a remote shade request to the destination node.

For efficiency reason, we chose to batch the remote shade requests on each node in a *non-resident gray set* until the *resident gray set* is empty. Then all references in the *non-resident gray set* are send to the relevant nodes. The used shading protocol bypasses the location protocol by using one common broadcast from each node to all other nodes of all non-resident object references as OIDs. Each node hosting any of these objects replies with the OIDs of their resident part of the broadcasted OIDs. This way the non-resident gray set is diminished as replies arrives. New references found after the exchange, e.g., by the remote shading, may introduce new references to non-resident objects, thus requiring the method to be repeated.

Failing to get replies for non-resident gray objects after doing multiple broadcasts indicates that the object may not be available any more. Thereafter, the location protocol is used to determine whether each of these objects are available or not. References to unavailable objects can be safely removed from the non-resident gray set, whereas the available objects are send a shade request to the node where they are located, using a reliable send.

5.3.4 Distributed Termination Detection Protocol

The end of the mark-phase is detected locally on each node as the two gray sets become empty. The end of the global mark-phase is a distributed termination detection problem. As discussed in section 4.4.3, this is solved by a two-phase commit protocol.
The termination criteria may be phrased as either of the following equivalent statements:

the *global mark-phase has finished*
$\iff$ *all nodes have finished the mark-phase* of their global garbage collector
$\iff$ *all nodes have* a global collector with *an empty* resident and non-resident *gray set.*

The protocol is entered by any node that believes that the goal has been reached (that it is reached locally is a good hint). During the first phase (Phase 4a on Figure 4.2, page 74) information is collected until it is true that all nodes have been locally finished and are still believed to be finished. During the second phase (Phase 4b on Figure 4.2, page 74) each node verifies that it is still finished and has not been non-finished since the end of the first phase. Thus the second phase verifies that all nodes have been finished simultaneous, which confirms that the global mark-phase is terminated as no work is in transit from a finished node.

During the termination protocol, each node associates a timestamp (a counter) with each state, i.e., each time a mark-phase is finished (again) locally the timestamp is incremented. Receiving the same timestamp from a node during the second phase, as was the result of the first phase, verifies that the node has been finished during the whole period. Any node may detect this by running the protocol, i.e., collecting the timestamps from the other nodes.

As it turns out, it is sufficient to use a simple timestamp namely a count of the number of times this node has reached the empty state during the current garbage collection cycle.

During the first phase the protocol is realized by the exchange of *status* messages of the form *Node 1 has reached its empty state 5 times*. We use tuples $< 1, 5 >$ to describe such messages. These messages must be combined, thus each node must send the message to at least one other node. For reasons of efficiency, we want to limit the number of messages send out, which impose a kind of structure on the inter-node communication. A fixed communication structure should, however, be prevented to circumvent failures most gracefully.

We search for a solution between:

- a *fixed* communication structure, e.g., the organization of all nodes in a circular list or a hierarchy, and
- a *dynamic* all-to-all communication pattern, e.g., everybody informs everybody on every change of state.

A solution, which impose a dynamic structure on the inter-node communication pattern based on current availability of nodes, would be a compromise, if the adaption of structure is achieved without a large overhead.

In real systems, the many artificial situations, which we try to foresee, are rare. Thus the simple solution will solve the whole problem most of the times. Such a solution can be further backed up by more advanced methods in the rare circumstances, where it fails, if its lack of success is detectable.

Our simple solution to distributed termination detection is based on *repeated broadcasts*. By repeating the broadcasts, we turn the unreliable communication into a reliable one. Due to the rather high cost imposed on all systems on the local-area network by broadcasts, the frequency of repeats are halved each time, i.e., the delays between successive repeats are doubled until the repeater is finally canceled (See Section 5.4.1). Note, however, that this solution is inappropriate in case of larger distributed systems on wide-area networks, where broadcasts would be an intolerable burden.

Each node, i, holds a list of tuples $< j, count_j >$, where j is a node and $count_j$ is the number of times node j has reached its local termination state (both gray sets empty). A handler on each node reads broadcasted tuples and adjust its own list of tuples to keep the youngest tuple from each node. The handler is enabled during the whole mark-phase to build the list of nodes which have finished before the current node and, thus accumulating full information fast. Each node, i, starts the termination detection protocol when the local termination is detected. It increments its own timestamp, $count_i$ and schedules the first of a series broadcasts of tuple $< i, count_i >$. If the node leaves its own termination state

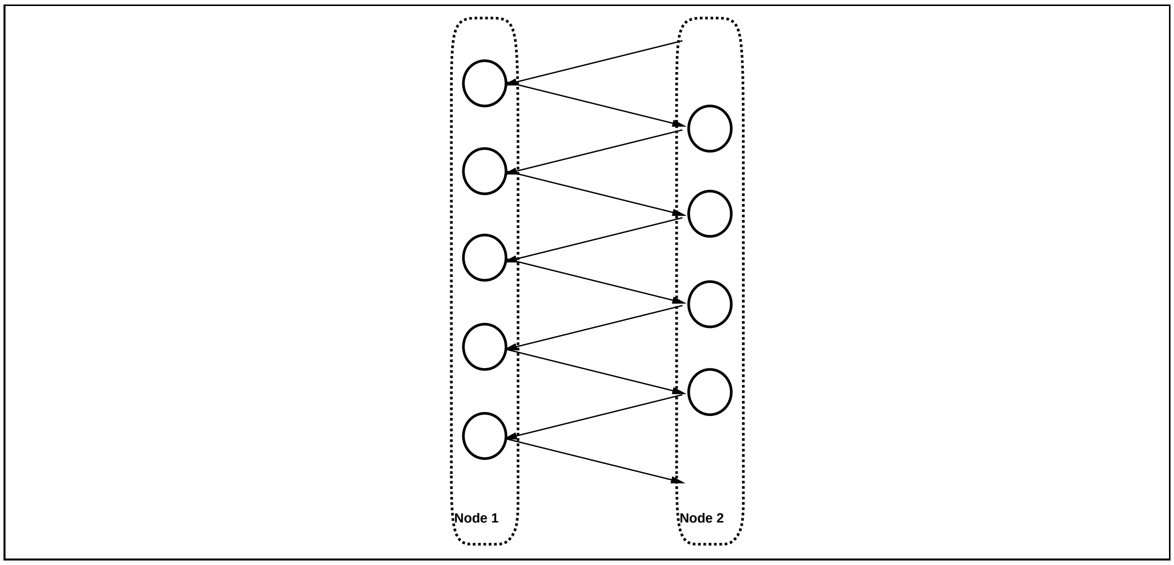


Figure 5.3: A graph of objects crossing node boundaries by each reference

due to incoming messages (*Shade request*), the termination detection protocol is left and the scheduled broadcast is canceled. The delay until the first message is broadcasted is chosen long enough to capture new *Shade request* in the cases where a few nodes alternates between finishing, e.g., a singly linked list of objects crossing node boundaries between each object (Figure 5.3).

When all tuples have been received at least once including a valid version of the tuple of the current node, it is time to verify that the contents of the list of tuples is up to date. The tuples received in the following period is used to verify, that each of the other nodes has been stable since the period started. When the last tuple is verified, the global termination point has been reached. If a tuple with a newer timestamp is received during this period, the tuple list is updated accordingly and the verification period is started again. Thus we need at least one message more from each node, to be convinced that nothing has changed since the new period was started.

The second phase is thus entered and re-entered each time a node has got a new complete list of tuples. As all nodes repeatedly broadcast their tuple, no polling from the node entering the second phase is necessary to make the other nodes verify their tuples. Visually, we may picture the implicit entrance of the second phase by setting a mark on a time line. After the mark has been set, the node continues to send out broadcasts and collects the information from other broadcast. Any new tuple value simply causes this mark to be moved forward, i.e., reset to now.

Example: Global Termination Detection

Figure 5.4 shows an example on how the two-phase commit protocol is realized. The protocol is initiated when the local termination is detected, e.g., on node 1 at the point marked 4, on node 2 at the points marked 1 and 3, on node 3 at the point marked 6, on node 4 at the points marked 5 and 9, and on node 5 at the point marked 8. The termination detection protocol may be left again due to incoming messages (*Shade request*) as illustrated at point 2 for node 2, and at point 7 for node 4.

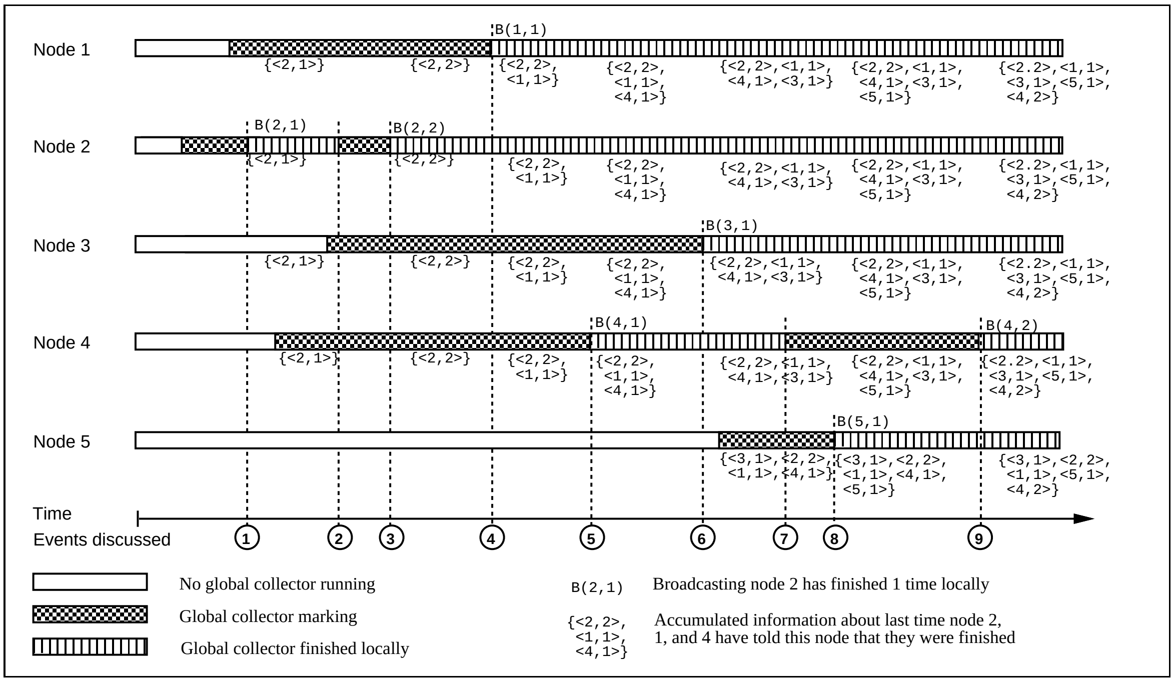


Figure 5.4: Detection global termination by repeated broadcasts

During the first phase each received status information for other nodes is accumulated in the list of tuples associated with the nodes and illustrated on Figure 5.4 below the node status lines. The second phase is entered when all tuples have been received at least once including a valid version of the tuple of the current node, e.g., after the point 8 on node 5.

During the second phase the existing list of tuples is verified. If a tuple with a newer timestamp is received during this period as done by node 4 at point 9, the tuple list is updated accordingly and the verification period (the second phase) is started again. The second phase is thus entered at point 8 and re-entered at point 9.

5.3.5 Definition of the Object Graph

The traversal of the object graph during the mark-phase is done by following references to other objects in the objects. The traversal algorithm makes use of a set of references to objects, which shall be traversed, but has not been yet, i.e., the *gray set*.

The implementation of the traversal in Emerald must identify exactly which pointers in the Emerald kernel that make up *the root set*. Furthermore, the traversal needs for each type of object a precise definition of which references to follow. Those back-pointers that are included to optimize the execution of Emerald are not followed.

In short, the root set includes the processes ready-to-run, where each process is implemented as a chain of stack segments (SS). The user objects are implemented as local (LODATA) or global system objects (GODATA), or the special-purpose condition objects (COND). Furthermore, the processes and user objects may have code associated with them.

The main work done by the mark-phase of the garbage collector is to follow the SSs in the ready-to-run queue and mark these as well as all other objects directly or indirectly reachable from these.

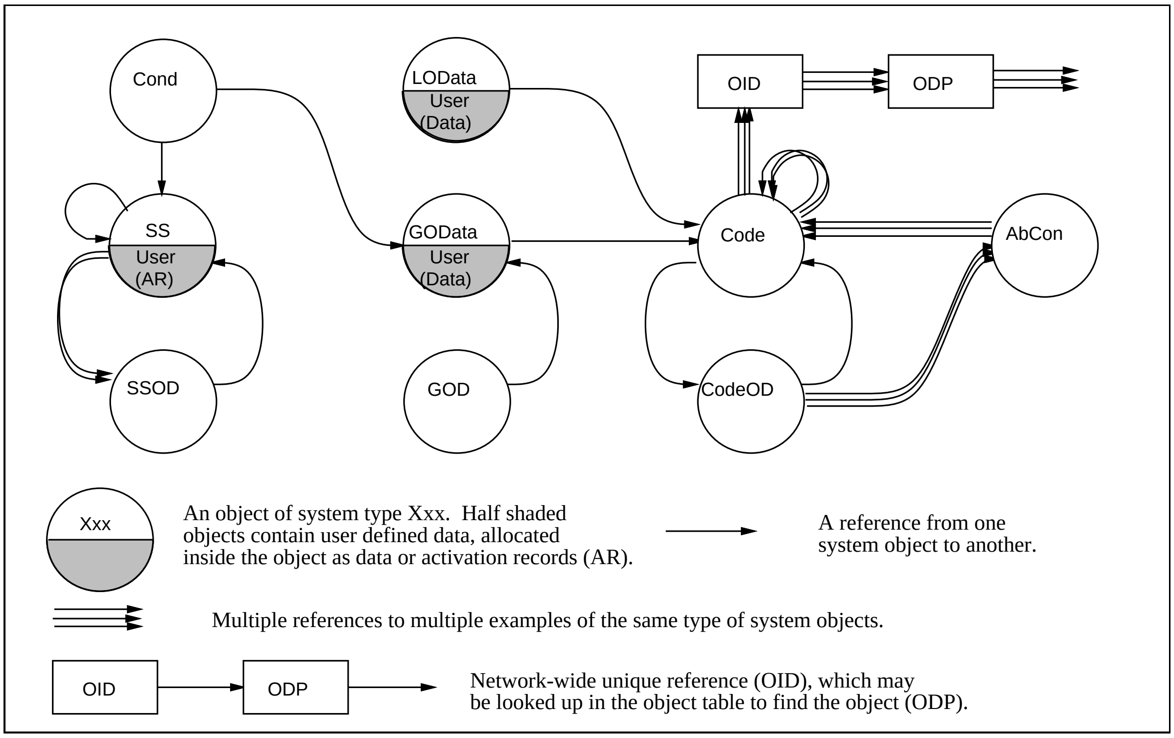


Figure 5.5: System and user object references to be followed by the garbage collector

5.3.6 Traversal of the Object Graph

The objects in the object graph are all allocated in the distributed Emerald heap. The objects are of either of the implementation types listed in Section A.4 and shown on Figure 5.5. References are either absolute memory addresses or symbolic references (OIDs). Symbolic references are used when crossing node boundaries, but may also be used internally on a node. Each node maintains an *object table*, where OIDs may be looked up to find the absolute memory address of the OD for the object, and thus the object itself if it is resident.

Figure 5.5 shows the system references which must be followed between system objects. These system references ensure that the garbage collector may treat system and user objects equally. The traversal of the user objects includes, however, a traversal of the associated templates describing the layout of variables according to the internal types described in Section A.4 also. The USER(DATA) and USER(AR) contains the data and activation records respectively. In the data areas, references to be followed in the traversal are internally typed as either: ODP, VARIABLE, or MONITOR.

- The ODP references another system object.
- The VARIABLE contains one or two references to system objects of the types (LODATA or GODATA, and ABCON).
- The MONITOR contains a list of SS.

In the activation record, references to be followed in the traversal have the internal types ODP or VARIABLE with the same contents as above. The templates describing the layout of both the data areas and activation records are found in the code associated the data area or invocation.

The distributed traversal has already been partitioned in a node-local traversal and the exchange of batched shade requests. Node-locally, the traversal must protect objects put in the gray set for later traversal. As only some system objects have the protection mechanism available, it is necessary to traverse the non-protected objects immediately, i.e., without putting them in the gray set. The amount of immediately traversal must still be kept small to limit the blocking period of mutators on the same node.

Between a depth-first and a breath-first traversal, we have chosen the intermediate solution of a complete traversal of all non-protected objects, leaving the protected for later traversal. If the local Emerald heap is residing in a paged virtual memory, the traversal has to cooperate with the paging system to limit the number of page-faults[1]. The collector may try to traverse all gray objects on an available page immediately to limit thrashing. The garbage collection fault mechanism ensures that the traversal of protected objects is done on pages made available in primary memory for execution already. Thus the access pattern of pages by the collector follows the access pattern of the executing user processes. This suggests that a tighter cooperation with the paging system is of lesser importance to the mark-phase.

5.3.7 The Identification of the Root Set

The root set is distributed. Each global collector identifies its own part of the global root set, thus the union of these root sets constitutes the distributed global root set. As with any mark-and-sweep collector, the processes that are ready to run, must go into the root set. Moreover, objects inherently available to the user of the Emerald system must also be part of the root set. These objects are so persistent that they may not be reclaimed.

In the Emerald system, some objects are created at compile-time, i.e., the compiler cheats the type system by pretending that these objects already exist. The Emerald kernel ensures that they are created the first time they are needed. These objects should persist as long as an Emerald program using them may be started in the future and therefore they are maintained by the kernel as *cheating created* and put in the root set.

At any moment, some objects may be only partly instantiated. This yields objects moved from another node or submitted from the compiler, if they are not yet fully translated. The garbage collector may not traverse them, as their references are not correctly initiated yet. Still their references must be shaded. The Emerald kernel maintains special lists of objects *in transit*, i.e., objects awaiting completion of their translation and the correct values of their eventual references. Objects in transit must a priori be alive, thus the mark-phase may simply add these references to its root set. Moreover, it must do so repeatedly as further references may arrive during the mark-phase. Thus, the mark-phase awaits termination until all references in these lists have been followed also.

Not all the user processes are part of the root set. Those running or ready-to-run are of course, but those waiting in other queues are alive only, if "someone" is able to change their status from waiting to running. Thus, reachability defines their liveness, i.e., a waiting process is alive (reachable), if and only if, the queue where it waits is reachable. This determines the liveness of processes waiting for access to monitors or in condition queues. The traversal of

[1]A similar argument goes for the cache of a caching system, or any other level of a multi-level storage organization.

the running processes will ensure that these waiting processes are marked and traversed also if their queues are reachable from the running processes. If they are not reachable, no one is able to take them out of the queue, thus they should be and will be identified as garbage. This way *orphan processes*, i.e., processes waiting forever, will be collected after the last process that was able to resume them has disappeared. Processes waiting in system queues, where the kernel is able to resume them, must of course be treated as part of the root set. This includes the processes waiting on an i/o-operation to complete or a timer to expire.

The establishing of the root set needs not be done completely at the start of the mark-phase. Only the user processes are suspended and processed at start. The additional root objects need not be protected, but may simply be added to the gray set during the mark-phase.

5.3.8 Traversal of Mutators

The user processes identified as non-garbage at start or during the mark-phase are suspended until their stacks have been traversed and all contained references marked at least gray. A process may, however, have a stack which is partitioned in several stack segments (SS's) that might reside on different nodes. Thus, the traversal of a process could be a large and potentially distributed job faced with unavailable SS's.

To preserve robustness and correctness, and to limit the pauses introduced, only part of a mutator is traversed at first. The representation of a mutator is a chain of SS's of which the activation record at the top of the topmost SS is identified as living when the process is suspended. The traversal may be limit to:

1. the topmost activation record,
2. the topmost stack segment (SS),
3. the chain of SS's from the top until the chain crosses a node boundary,
4. the chain of SS's from the top, across node boundaries, until a SS on an unavailable node is needed, or
5. the entire chain of SS's,

after which references to the rest is protected and gray.

The candidates fitting best with the implementation model of system objects are 2 and 3, whereas no protection between activation records in the same SS is available, and potential failures and inter-node communication delays rule out 4 and 5. The boundary between system objects, i.e., SS's, delivers the most general solution, thus 2 is selected.

This requires that the reference from the bottom of a SS to the underlying SS is protective and able to use the garbage collection fault mechanism, independent on whether the reference are intra- or inter-node. The reference is protected and marked gray when the topmost SS is marked and traversed. Also processes waiting in queues reachable from either user objects or the kernel must be marked gray and protected when found reachable. The faulting on return across SS's is further discussed in Section 5.4.2.

5.4 Support for Garbage Collection in Emerald

Distributed garbage collection needs support for inter-node communication between the global collectors using the protocols described in Section 5.3.1. The implementation of this communication facility is described in Section 5.4.1. The requested concurrency between collectors and mutators demands a parallel processing capability (Section 5.4.2) and the garbage collection fault mechanism (Section 5.4.3) for protection of objects to ensure the garbage collection invariants.

5.4.1 Inter-node Communication Facility

The Emerald kernel has builtin facilities for communication between the nodes. These are build on top of UNIX sockets using the UDP protocol. The available communication services include reliable message delivery to the kernel at any destination node as well as out-of-band broadcast messages delivered to all the kernels currently alive on the network of nodes. The service defines a set of message types and associate handlers, i.e., routines called on arrival of the corresponding message type.

During one garbage collection cycle a node needs the following communication with other nodes:

1. Broadcasting once and unreliable a hint about the start of the current cycle at this node.

2. Reliable multicasting of the batch of shade requests to all relevant nodes in the distributed system. Only the relevant nodes know, which are relevant, thus they must reliably reply to the sender to notify the sender about which shade request that have been serviced.

3. Reliable multicasting of termination detection messages at the end of the nodes local mark-phase to all nodes in the distributed system combined with an acknowledgment to all nodes when global termination is detected on a node.

4. Unreliable broadcasting of the readiness of a node to start a new garbage collection cycle, i.e., a hint about the willingness of the node to participate in the next garbage collection cycle due to the termination of the sweep-phase of a previous cycle.

The *unreliable broadcasting* of hints (1 and 4) is implemented by sending the message once by the broadcasting mechanism available. The messages might be lost, i.e., unavailable nodes do not get them. Both 2 and 3 above needs a *reliable multicast* to all nodes. This may be accomplished by reliable point-to-point delivery of the message from one node to each of the others or by turning the unreliable broadcasting service into a reliable service by using repeated broadcasts until sufficient nodes have acknowledged. The use of the reliable multicasting in 2 and 3 favors, however, the repeated broadcasting with exponential back-off until sufficient acknowledgments are received by reliable point-to-point reply messages.

The contents of the message in 2, is a batch of shade requests, where each request must be acknowledged by at least one node in the system, e.g., the node hosting the object to be shaded. Thus one broadcast would be enough if all nodes are listening. To achieve robustness to failed nodes, the receiving nodes send back acknowledgments for those shade requests they are able to handle. This way, the broadcasting node may send out a batch of shade requests for the still not acknowledged requests a little later. If the designated receiver of the shade request is a failed node, the broadcast will be repeated until the node recovers, which may saturate the network by useless broadcasts. Therefore, the repeated broadcasting of the same message is delayed by doubling the time between each broadcast (exponential back-off). Eventually, the designated node will recover and the broadcasting node stops broadcasting as all its shade requests have been replied.

We start with a minimum delay of n times the cost of handling a broadcast, where n is the number of participating nodes. The delay may be advanced to a system constant, e.g. 30 minutes, at most. When the longest delay is reached the location protocol is invoked to remove shade requests due to dangling references and the frequency of broadcasts are kept constantly low until the needed nodes become available again.

The termination detection protocol (3) takes advantage of the same repeated broadcasting method. Acknowledgment in this situation is only needed when a node detects the global termination. The global state is stable, thus by broadcasting the termination message once, the available nodes reach the same conclusion faster than else. It is, however, an optimization not a necessity, as any message exchange hereafter implicitly contains the termination information.

The distributed termination detection may be further optimized to limit the number of broadcasts and the amount of communications, without compromising on robustness to node failures. As mentioned in Section 5.3.1, the inter-node communication between any node and all other nodes is needed at least twice. The second time, to confirm that nodes still have the same timestamp. Whereas the first time is needed to convince any node, that it is worth trying to verify, that we are finish. We want to limit the number of messages, which do not contribute to the convergence towards global termination detection. Still enough messages must be send to ensure liveness and achieve robustness to temporary node failures.

The following limitations on repeated broadcasts are thus applied:

- When a node is finished, it schedules a message of its own new tuple and recently received tuples for later broadcasting.
- As long as other nodes are broadcasting *Shade requests* or *Shade replies* this node suspends its scheduled broadcast, i.e., it reset its delay, while the other nodes are busy.
- When the other nodes calm down, the scheduled broadcast will be send for real, and a new broadcast with a longer delay scheduled.
- When a broadcast, containing the tuple of the receiving node, is received, the broadcasting of the node is given a longer delay, as this indicates that our message has reached at least one other node.

This way the broadcasts will be calmed done quickly. The resting problem is that we would like the frequency to be enlarged when we think, we are done and need verifications. Such a situation may, however, be hard to separate from the situation illustrated by Figure 5.3. The only true indication that we are all finished is when no one broadcasts anything, but this may lead to a deadlock situation. Thus the simple scheme to schedule and reschedule the current message is still the best to do, as this makes us complete robust to unavailability of nodes. The cost is that we are broadcasting while there is no need for it, as someone do not listen, i.e., an unavailable node.

5.4.2 Parallel Processing in the Emerald Kernel

Concurrent execution of collector processes requires a new level of process scheduling in the Emerald kernel between the kernel and the user level. The levels are described in Section A.3 and summarized in Figure A.1. Figure 5.6 gives almost the same picture including the new level of priorities described here as the Emerald Garbage Collector Dispatcher. The added level allows garbage collector processes to be executed in parallel with other processes, but on a separate priority level. The primitive dispatch system allows the garbage collector processes to schedule them selves, and requires that they are self-preemptive and able to be rescheduled by them selves, thus supporting a *continuation passing style* implementation of the processes.

As described in Section A.3, the Emerald kernel supports three levels of processes:

interrupts,	i.e., UNIX signals, which are handled immediately on arrival by scheduling an appropriate kernel task.
kernel tasks,	which are short, indivisible executable jobs, e.g., handlers associated with external or internal events.
user processes,	i.e., Emerald processes, with compiler inserted self-preemptive behavior (*flick*).

The user processes are scheduled round-robin in the *ready-to-run queue* if their time slice is exceeded when they *flick*, or resumed running if they are otherwise able to continue. During each *flick*, the *kernel queue* of kernel tasks is emptied completely. Thus kernel tasks are handled with the highest priority—besides interrupt routines, which we do not consider further.

One each node, the garbage collectors are largely implemented as:

- two mark processes (the *local marker* and the *global marker*),
- a common sweep process (the *sweeper*), and
- event handlers activated by incoming messages via the communication sub-system or the garbage collection fault mechanism.

Both the markers and the sweeper must be able to run while user processes are running also on the same node. This is achievable by quasi-concurrency only, which demands us to multiplex the various processes. The event handlers are invoked by the system according to the incoming events, and thus, handled like any other kernel task via the *kernel queue*. When enabled at the end of the mark-phase, the sweeper may work rather independent of the garbage collector, i.e., the markers. On the other hand, it is tightly connected with storage allocation and thus hardwired into the allocation routines (see Section 5.5.2).

The markers may, however, run more independently. Their burden is to heavy to schedule them as kernel tasks, and they are not user processes either. The Emerald kernel has thus been extended with a new internal dispatcher. The *Emerald Garbage Collector Dispatcher* borrows the CPU when the *kernel queue* has been emptied, but not each time it is emptied. At this, rather well defined place in the kernel, the dispatcher will enable one of its queued processes to run non-preemptive. The dispatcher may start at most one of its queued processes each time the kernel has emptied its *kernel queue*. The amount of "stolen" CPU cycles used by the dispatcher is tunable, as a counter enables the dispatcher to start a process each n^{th} time

Priority	Level	Queue	Processes on this level
1.	Interrupt level	—	UNIX signals results in an appropriate kernel task (the corresponding event handler) being inserted in taskQ.
2.	Kernel level	taskQ	Actions to be done by the kernel due to in- and external events including garbage collector events.
3.	Dispatcher level	dispatchQ	Marker processes scheduled for execution by the dispatch system.
4.	User level	current	The user process running or doing *flick* right now.
		readyQ	The *ready-to-run* user processes.

Figure 5.6: The hierarchy of processes in the Emerald kernel

the *kernel queue* is emptied. When no user process nor kernel task is runable, i.e., both the *ready-to-run queue* and the *kernel queue* are empty, and one or more collection processes are pending, the dispatcher ensures that the collection processes are executed one at a time. The new hierarchy of processes in the Emerald kernel is shown in Figure 5.6.

The dispatcher system lets garbage collector processes queue-in for quasi-concurrent execution. It accepts a routine to be scheduled for later execution. The routine must do a limit amount of work only, before it preempts itself by either rescheduling itself, scheduling another routine, or simply removing itself from the dispatch system. Thus, the collection processes are trusted by the dispatch system to perform cooperatively and in a friendly manner. Scheduled collection processes are placed in a queue in the dispatch system (dispatchQ). The dispatch system also allows removal of any process scheduled for execution from the queue again. For simplicity and efficiency the dispatch system does not include a private stack and process description saved between successive executions of the collection processes. Instead, a *continuation passing style* implementation of the collection processes is enforced. It is backed up by global variables for simplicity and efficiency.

The two markers are implemented as a set of routines executed sequentially for each marker, by scheduling either the same routine again or the following, at the end of the running routine. As the routines are not given their own private stack saved across successive calls, the few necessary data structures are saved as global data instead.

5.4.3 The Garbage Collection Fault Mechanism

The synchronization between collectors and mutators is achieved by using a garbage collection fault mechanism similar to the general Emerald fault mechanism used for remote invocation (Section A.5). The synchronization is needed during the mark-phase to ensure that the user processes access black object only even though they might reference gray objects also.

Invocation Faulting

As described in Section 3.5, the gray objects are protected (FROZEN is the term used inside the kernel) similar to remote objects. When accessed, i.e., when a user process invokes an operation in a protected object, the control is given to a general fault handler inside the kernel. When appropriate actions have been taken, control is passed back to the user process, i.e., the invocation of the operations in the (former) protected object is resumed or finalized.

The appropriate actions done during the fault are selected according to the status bits in the descriptor of the object (the OD). The two marker processes have their own status bits (GLOBALGCFROZEN and LOCALGCFROZEN). The FROZEN-bit of the OD is a logical-or of these two bits and the other status bits, e.g., ISRESIDENT and SETUPDONE. As more than one reason may have caused the fault, each is inspected and the associated handler called before the status bits including the FROZEN-bit are reset. Thus, if both the local and global marker have protected the object, the object will be marked and traversed according to both the local and the global mark-bits.

Faulting on Return

The mutators are not only faced with protected object at invocation time. Also returning from an invocation may resume action in a non-black object, if the entire stack of the process has not been traversed and marked yet. Mutators are expected to be black, thus all their references are at least gray. This requires that the entire stack of each mutator is traversed and marked. The stack is, however, partitioned in stack segments (SS), each of which may reside on another node. This was fully discussed in Section 5.3.5.

As the liveness of a mutator is detected at the topmost SS, the immediately underlying SS is marked gray and protected, while the top SS is traversed and marked black. When a mutator returns across the bottom of a SS, the underlying, protected SS forces the scheduler of user processes to call the garbage collection fault handler first. Thus faulting on return across stack segments is implemented, independent of whether a node boundary is crossed during the return or not.

5.5 Cooperation Between Kernel and Collector

The garbage collector need to cooperate with the kernel, especially the storage management routines. This is used to ensure that the relevant root set of objects are delivered to the collector when started (Section 5.5.1) and to incorporate the reclamation, i.e., the sweep process, in the allocator (Section 5.5.2). The kernel also delivers hints on when to start a new collection as described in Section 5.5.3. The knowledge about all nodes of the entire system is needed by the mark-phase to determine when a global collection is completed. Section 5.5.4 describes how the kernel determine the global set of nodes.

5.5.1 Delivering an Accurate Root Set

To deliver an accurate root set at the start of a garbage collection as described in Section 5.3.5 on page 91, the kernel keeps information on certain objects.

First—and foremost—the queue of Emerald processes *ready to run*, i.e., the mutator list is available in the kernel. This list is extended with Emerald processes waiting for external events like i/o and timers, but not those processes waiting in a queue that is only known by other Emerald processes. The liveness of the latter processes are solely defined by their reachability—via the queue they are waiting in—from the processes in the root set.

Second, references to the objects generated at system start up are recorded inside the kernel and put in the gray set at step 6.6 of the local and step 5.6 of the global marker process. At this point, objects made by *cheating creates*, objects under translation, and other objects promised to be "always available" are also added. These have all been registered inside the kernel when they were created. The objects potentially known from outside the node in focus, i.e., those resident objects with *ReferenceGivenOut*-bit set, are also dynamically registered inside the kernel. These are, however, only added to the local marker (in step 6.6).

When objects are found to be garbage, they are reclaimed, thus any reference to them from inside any kernel table, set, list, queue, or other kernel data structure, must also be removed. This is done in step 5.8 of the global, and in step 6.8 of the local marker process for the above mentioned references inside the kernel and for the *Object Table* on the node as well.

5.5.2 Cooperation with the Storage Allocator

The sweeper process is enabled on a node by either of the completed marker processes. According to the coding of the mark-field of the completed marker process, the sweeper traverse the storage to reclaim any object with a bit-pattern in its mark-field matching one of the current garbage patterns.

The sweeper is parameterized to reclaim at least the given amount of storage, if possible at all. Instead of forcing the sweeper to run as a separate process, it is called from the run-time storage allocator. When enabled, the sweeper is asked by the allocator to sweep cyclically through the entire heap. It is called to reclaim at least the same amount of storage as was requested from the allocator.

For each possible garbage pattern to be reclaimed, the sweeper does a cyclically traversal of the entire heap from its current position.

5.5.3 When to Stop and Start

The Emerald kernel has been instrumented to collect statistics, and to take action in certain situations. This is needed to ensure that a new garbage collection cycle of either the local or global kind is started at the right point.

A new garbage collection should be started before the allocator runs out of available storage. Thus the local marker is enabled to run one more garbage collection cycle when a certain threshold is meet. This way local garbage, which is expected to be most of the garbage, is marked between the threshold and the maximum of available storage, where after the sweeper is enabled to sweep the storage and reclaim the garbage. The distance between the threshold and the maximum must be tuned to match the run-time used by the marker. Should the tuning fail and the allocator effectively run out of available storage, its caller must be delayed, and the marker made a high priority process, i.e., the marker must identify the garbage in a hurry to enable the sweeper and the application calling the allocator again.

The global marker is triggered like the local but on other statistical values. One of the indicators, that global garbage might be available, is that the number of objects on a node potentially known from other nodes has grown large. Thus when the number of objects with *ReferenceGivenOut*-bit set reaches a certain threshold on any node, the global marker is enabled.

5.5.4 A Global view of the World

The global garbage collector stops the mark-phase when all objects are black or white, i.e., all gray sets are empty. For this purpose it needs to know about gray sets on all nodes that have been available. An example is a previous active node which has checkpointed some objects who refer to white object on the running nodes. When a checkpointed object is restarted in the future, we shall guarantee all its references valid.

To do a global garbage collection we need to know about all nodes of the actual Emerald system. Not only the nodes which are currently running. We need to know about all nodes, who may have a reference to an object on any of the other nodes. For this and other purposes we maintain a list of other nodes (HOTS) on each node. The list is updated as new nodes are recognized by the node holding the list. We do only insertion of new recognized nodes (never deletions). Thus as objects move around the list on each node gets a fairly good picture of the world. Furthermore, each new node broadcast a message about itself at startup.

In a distributed system, like Emerald, it is often difficult to maintain the same and realistic picture of the state of the total system. Without centralized control the system state is only retainable as the state of the parts, it consists of.

But before issuing a global garbage collection, we force the nodes to synchronize their lists, i.e., exchange information until everybody have the same picture. This gives a global picture when garbage collection starts. The mark-phase is the defined to be finished when all "start nodes" have empty gray sets. New nodes are of no concern as new objects and every reference passing node boundaries hereafter will be defined living by the garbage collector as they move between the nodes.

5.5.5 Additional Benefits

Emerald has a general mechanism to locate objects. The mechanism takes advantage of hints, i.e., each node that has had a reference to an object maintains a hint, the most recently known location of the object known here. Such hints may be old and useless if the object we want to locate has been moving around a lot without passing this node and updating the hint. By following the hints from node to node the object may at last be located, but this forwarding chain may potentially be as long as there are nodes in the system.

As the mark-phase of the global garbage collector has to locate non-resident, living objects during remote shading, it may as well update these hints as it becomes aware of more recent information about the current location of an object. In this way, the length of the forwarding chain is reduced, and better performance of this type of application expected.

5.6 Implementation Problems Surmounted

We have presented some of the more important problems surmounted in the implementation of the Emerald garbage collection scheme. The scheme has been implemented in the Emerald prototype, although it has not been integrated with the checkpoint/recovery system of Emerald.

The inter-node communication scheme uses point-to-point messages and repeated broadcasts with exponential back-off. These are used for the exchange of gray sets and by the distributed termination protocol which detects the end of the distributed mark-phase. A simple synchronization scheme by tagging all inter-node messages ensures that the garbage collectors on all nodes will always run in step.

The system objects and basic data structures of the run-time system have been analyzed to define which references to follow and to identify the root set for each garbage collection.

The necessary mechanisms inside the Emerald kernel have been added including the communication mentioned above, the intermediate processing queue, and augmentation of the faulting mechanism for garbage collection faulting on both invoke and return. Further cooperation between the existing kernel and the garbage collectors is supported by hooks inside the kernel to deliver the correct root set, interleave the sweeper with the storage allocator, ensure that the collector is called before the allocator is unable to service its clients, and tells which node the system is build of.

Chapter 6

Evaluation

The comprehensive, concurrent, and robust distributed garbage collection scheme described in the previous chapters has been incorporated in the Emerald prototype. The scheme do collect all garbage and works during long sessions without storage leakage. In contrast to most distributed collectors, this collector do 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 in Emerald. 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 1.8. These are now used to see whether the goals presented in Section 1.5 have been met. 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 an Emerald program that creates artificial garbage and by ordinary Emerald applications.

This chapter 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 6.1). In Section 6.2 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 6.3 we show how the traditional problem due to cyclic garbage is solved without any special effort, and in Section 6.4, 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 6.5. The pauses introduced into normal execution by the garbage collector have also been investigated and the measurements showing this is presented together with estimates of the pauses in general (Section 6.6). Our scheme gives us orphan detection at no additional cost (Section 6.7). We end this chapter with a description of the implementation of the garbage collection scheme in general, quantitative figures (Section 6.8) and summarize the evaluation at last (Section 6.9).

The measurements have been done at the VAX implementation on top of UNIX, version 4.3BSD, on a network of four VAXstation 2000 workstations at DIKU, Department of Computer Science, University of Copenhagen. Unless otherwise stated, all figures given in the following are based on this implementation. Further statistical information about the implementation is given in Appendix C.

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 6.1: The number of objects in the object store

6.1 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 of the mark-phase and the start of the sweep-phase, and
4. after a full collection is done.

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 6.1. 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 either 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 persists 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 Emerald compiler and kernel. These are a priori known outside the kernel, and thus, their references goes 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 Emerald compiler may compile new user programs with references to these objects included. An independent Emerald 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 6.1 seems to verify this.

To detect leakage and dangling references we have also implemented the mark-phase of a simple, conservative collector. The collector inspect all cells and detect 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. Among the later type, it was able to identify leakage in a standard C-library routine supplied with the ULTRIX, version 3.1 system used under debugging.

6.2 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. Emerald is running on a four node network.
2. A global collection proceeds on all four nodes until the mark-phase 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 Emerald prototype has been disabled we have not been able to restart crashed nodes with checkpointed objects. Instead the node crash and later recovery is simulated by stopping the UNIX 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 restart from that state when the node recovers. During any other point of the garbage collection, the recovered node simply restart 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.

6.3 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 mark-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 that 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 is potentially known from other nodes. The global collector does, however, remove such dead cycles without even knowing about cycle detection.

6.4 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 mark-phase 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.

One of the distributed structures that complicate distributed collection is the *zipper* illustrated on Figure 5.3 in Section 5.3.1. During the mark-phase each reference in this structure has to be done as a separate remote shade. Each time a shade request is confirmed with a reply the distributed termination detection protocol is entered but left again immediately afterward as the other node sends a shade request in the opposite direction. 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 delaying the re-entering of the termination detection protocol a short while. The delay is kept small but long enough to capture simple forms of *the zipper problem*.

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.

6.5 Performance Degradation Due to Garbage Collection

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

1. Micro-timings by logging time-stamps on entry and exit of various garbage collector segments of the Emerald prototype.

2. Counting instructions added for garbage collection purpose by code inspection of the C source implementing the Emerald prototype.

3. Macro-timings by wall-clock and user/system time, comparing a run with and a run without a garbage collector.

The applications run are either artificial Emerald programs managed by the *Synthetic Garbage Creator* or real user programs like the Emerald mail test program.

The main result is that typical applications are slowed down by 10% while a collection is progressing simultaneous.

The time is mostly used to traverse the objects and identify the references to other objects inside objects. Another significant factor is the internal management of gray objects during the mark-phase. These figures are, however, bound to the number of live objects in the system. The sweep-phase is bound to traverse the entire heap. Although the cost per object (both the live objects and the garbage objects) during sweep is small, the total work of the sweeper introduces a significant overhead.

Part of the overhead due to both sweeper and markers are in the internal management of these as parallel processes, running concurrently with user processes. This overhead pays off by limiting the length of the pauses introduced into user computation by the garbage collection.

On the bottom-line, it should be noted that no matter how costly garbage collection is, it enables programs to run although their accumulated consumption of dynamically allocated storage is larger than the available storage. Programs that do not need the garbage collection, e.g., because the accumulated amount of their dynamic storage allocation is limited, may run full speed simply by disabling the garbage collection.

6.6 Limited Latency

As the garbage collection is implemented to work while user processes are running, it may introduce pauses into user computation while the two are running concurrently on the same CPU. Instead of introducing one unacceptable long pause in user computation while the entire garbage collection is done, our collector does a smaller step at a time thus, introducing smaller pauses only. The collector work is already partitioned in phases, and the mark-phase is further partitioned into doing one object traversal at a time. As some of the simpler implemented objects cannot be protected separately, these are each traversed together with the traversal of the first object that references them.

Although it is possible to construct esoteric examples of a large group of objects that need to be traversed together, the observed behavior is to traverse between 3 and 10 objects at a time.

The length of the pauses introduced depends on the number of objects and the number of references inside the objects. The average traversal time has been limited to $100\mu sec$; although for some of the more complicated objects the traversals may costs more. The pauses are limited, mainly due to the limited amount of objects traversed at a time. If large monolithic objects were constructed in a non-object-oriented sense, longer pauses introduced from garbage collection would of course be the result.

The conducted experiments show that the latency introduced by garbage collection is no worse than the latency already present due to timeslicing of the user processes. Thus we conclude that garbage collection does not introduce new annoying pauses of the user processes.

	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	-40.0%
Storage usage	916 Kbytes	1,083 Kbytes	167 Kbytes	18.2%

Table 6.2: Storage usage for the kernel at boot time

6.7 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.

6.8 Kernel Boot Time Statistics

When the new—with garbage collector—extended kernel is running, the extensions for debugging and monitoring is usually not executed while Emerald programs are executed. Thus, these additions mostly influence the kernel by its additional consumption of storage and the time it takes to compile and load the kernel. As an example, the storage overhead at boot-time due to the extensions in the new kernel has been measured as shown in Table 6.2.

Measured at boot-time of the Emerald kernel, i.e., before any Emerald user programs were loaded, the extended Emerald kernel initially uses 1,083 Kbytes of storage, a growth of 18% storage usage.

The initial heap size is the pre-allocated heap-space, i.e., inclusive the heap blocks available on the free-lists inside the kernel. At boot-time a lot of useful objects is loaded into the kernel to be immediately available for the user processes. Thus the initially heap contains these objects as well as dynamically allocated kernel data structures such as dynamically expandable kernel tables. In general, the heap contains kernel date structures as well as data and code of Emerald objects. The adding of garbage collection does not change the initial contents of the heap with respect to the Emerald objects, only the dynamically expandable kernel structures are extended as shown in Table 6.3. In the new kernel with GC, 95% of the heap-space is in-use at boot-time (299 out of 313 Kbytes), thus there is almost no pre-

	Kernel without GC	Kernel with GC	GC overhead Kbytes	Percentage
Initially heap size	267 Kbytes	313 Kbytes	46 Kbytes	17.2%
Kernel data	143 Kbytes	183 Kbytes	40 Kbytes	28.0%
Emerald objects	108 Kbytes	108 Kbytes	0 Kbytes	0.0%
Available	10 Kbytes	14 Kbytes	-4 Kbytes	-40.0%
Heap usage	96.4%	95.4%		
Used heap	257 Kbytes	299 Kbytes	42 Kbytes	16.3%

Table 6.3: Heap usage at boot time

	Kernel without GC	Kernel with GC	GC overhead Kbytes	Percentage
User time	11.5 sec	14.0 sec	2.5 sec	21.7%
System time	10.1 sec	10.1 sec	0 sec	0%
Boot time	21.6 sec	24.1 sec	2.5 sec	11.6%
CPU-usage	60.4%	64.1%	3.7%	6.1%
Wall-clock time	35.6 sec	37.5 sec	1.9 sec	5.3%

Table 6.4: Boot time

allocated space immediately available for the first application requesting space. The total size of the allocated heap, including available storage blocks on the free lists growth from 267 Kbytes to 313 Kbytes.

The booting of an Emerald kernel costs a little extra when garbage collection is included. However, most of the time spend during boot of the kernel is due to pre-loading and initialization of the standard build-in objects, i.e., waiting for the code to arrive from the disk and translating, i.e., dynamically linking, it. Table 6.4 shows that the overhead—due to garbage collection—is approximately 2.5 seconds additional user-time.

All costs due to the added garbage collector should also be viewed as a trade-off between being or not being able to run large "applications which are otherwise unable to run if they cannot take advantage of the garbage collector. Thus the prize is paid for something—it is not just an additional overhead—it makes it possible to run programs with higher storage allocation and deallocation rates than else possible.

6.9 Evaluation Summary

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

1. a *comprehensive* collection of garbage,
2. complete a global collection while nodes are unavailable,
3. collection of *distributed cycles* of garbage,
4. progress while nodes are unavailable,
5. introduce a limited overhead on user computation,

6. work concurrently with user computation with small latency,

7. 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 local mark-and-sweep to be generational and use a copying collector for the young generation. The memory allocator currently in use runs a *quick-fit* strategy. This allocation strategy—together with the copying techniques—has to be tuned to operate in the virtual memory system of the actual UNIX implementation used.

Chapter 7

Conclusion

We have developed and implemented a distributed garbage collection scheme in the distributed, object-based system, Emerald. The scheme works in systems where failures are the norm. It is robust to individual node failures while still collecting all garbage. The collection scheme consists of a node local collector on each node and a global collector based on another set of collectors that cooperate. The global collector may not be expedient as it depends on pair-wise availability of all nodes, whereas the local collectors will always be able to identify local garbage expediently. Moreover, both collectors may run concurrently with each other and user processes while only introducing smaller pauses. The main contribution is concerned with distributed garbage detection, whereas the reclamation of identified garbage is solved ad hoc. A beneficial side-effect of the global garbage collection is that forwarding address chains are collapsed.

7.1 Goals Revisited

The goal of this thesis has been to prove that garbage collection can be comprehensive in a distributed system without loss of responsiveness. More precisely we stated our goals in two parts:

- Functional requirements:
  - *Comprehensive* and *concurrent* garbage detection in a distributed, object-based system.
  - *Robustness* to failures and temporary unavailability of nodes in the distributed system.
  - Ability to collect *local garbage* fast, also while the global collector is temporarily unable to complere.
- Performance goals:
  - Every part of the garbage collection process should be done *efficient* and the total scheme should introduce as little overhead as possible.
  - The *latency* introduced by the individual steps of the garbage collectors must be low to preserve responsiveness.
  - The node-local collection also must be *expedient* (in contrast to the global collector that depends on nodes which may be long-term unavailable).

7.2 The Solution

The requirements to functionality and performance have been met by our implementation in the Emerald system. Though performance is not our primary concern the garbage collector is not in-efficient. The functionality is achieved by our dual garbage collector scheme:

1. **The Global Collector** works across the distributed system and identifies all objects, that were garbage when the collector was initiated. It depends on pairwise availability of nodes, thus it is robust to partial and temporary failures of nodes.
   It is comprehensive, concurrent and efficient though it may not guarantee expedience.

2. **The Local Collectors** work independently on each node, where they collect local garbage immediately.

Both collectors may run concurrently with user programs on all nodes. The largest delay introduced by a garbage collector is when it is initiated because all processes must be marked and protected to ensure that they are traversed before they begin to execute. Synchronization between the local and global collector on each node is kept to a minimum. The only situations where the local collector is not allowed to run is when the global collector is doing some local traversing. These situations takes a limit amount of time (always finite!) after which the global will allow the local to run. The parallellization of the mark-phase with one or more sweep-phases on each node makes it possible to reclaim garbage incrementally.

The concurrency between collector and mutators has been achieved by the garbage collection fault mechanism. This mechanism is also used for mobility purpose and remote invocations in Emerald. The overhead in added code and data structures for garbage collection purposes is paid by a static overhead due to the potential of having a garbage collector and a small overhead per object. The later overhead is paid also when running without an initiated garbage collection, but it amounts to a few bits per object descriptor and a common data structure on each node only. When a garbage collection is initiated the work is interleaved with mutators during the whole collection.

The *garbage collector dispatcher*, which schedules the concurrent mark-process, and sweep-processes on each node, introduces a slow down in the user processes by stealing processor cycles up to 20%. We have tuned the dispatcher to steal cycles mainly when the system is otherwise idle. Thus the bare cost of a dispatcher call is only visible when garbage collecting in a highly saturated system.

The global collector is further characterized by its *distributed control* mechanism. There is no master-node in the system, and no master is elected during the collection. Any node is able to determine when a synchronization point is met. Thus failures that could be hard to handle in a system with centralized control, are easily handled here. Of course this robustness has its drawback, some extra amount of work is done, though that would also be the case in a centralized system, that took care of failures.

The recycling of Emerald system objects, e.g., stack segments, is automatically achieved, as system objects are handled like any other object by the garbage detection mechanism. Not only system objects, but also internal run-time resources, which are dynamically allocated, may be recycled by our scheme. In Emerald, all kind of limited resources in the run-time system are recycled this way. When garbage objects are detected all resources, not only storage, may be recycled. In Emerald, garbage collection of objects representing system resources, e.g., internal buffers and slots for file- and i/o-descriptors, allows simultaneous recycling of the system resources.

In Emerald, processes may run forever without ever doing anything useful. Though they are not able to interact with any live object in the whole distributed system, they possess the ability to suddenly open an i/o-stream and begin using it. Thus we are not able to collect such processes although they really do nothing useful.

Orphans, i.e., processes waiting in a queue forever, are, however, reclaimed. The *waiting forever* state is detected when no live part of the system has a reference to the queue during garbage collection. Thus our garbage detection identifies orphans at no extra cost. The recycling of such orphans is a bit more costly as they may themselves possess other system resources that has to be recycled properly also.

The implemented garbage collection scheme is also *transparent* to the Emerald programmer. It is purely an implementation optimization of the run-time system, no changes have been applied to the Emerald programming language. This is very much in contrast to the problems when adding garbage collection to languages with low-level pointer manipulation capabilities like C and C++.

7.3 Work Done

We have pursued our goals by investigating garbage collection, distributed systems and object-oriented programming. The different flavors of garbage collection techniques from the early history of LISP and up to recent proposals for realistic and efficient garbage collection in both sequential and distributed systems are surveyed in Chapter 2.

Our proposals has been applied to the Emerald system. The Emerald prototype has been extended with routines to identify object references in both user defined and system objects. The faulting mechanism in the Emerald kernel has been generalized to take garbage collection faults into account also. The local and global garbage collector has been implemented in the Emerald prototype. Further changes has been done to implement a new level of parallel processing inside the kernel, *the garbage collector dispatcher*, thus enabling quasi-concurrency between user processes and garbage collectors on each node.

The Emerald prototype has been tested on a network of four VAXstation 2000 workstations. Artificial as well as more realistic Emerald programs have been written and run on the prototype. Parallel work has made the Emerald system available on SUN-3, HP and SPARC workstations, but the garbage collector has only been tested on VAX and SPARC architectures.

7.4 Limitations in the Current Implementation

Though we have a working implementation some short cuts have been made to make the Emerald prototype running, i.e.:

- The garbage collection scheme should be integrated with the file system collector, that manage the libraries of compiled code available for the Emerald kernel. This will enable objects, previously know by the compiler system, to be removed from the root set, when the compiler and all compiled programs have dropped them.

- The combined effect of node failures and checkpointed objects being recoverable has not been tested, as the checkpoint system was not available in the used implementation of Emerald.

- The reclamation part has left room for further improvements. The tuning of the sweeper to cooperate with the virtual memory system is a matter of further research as is the tuning of when to initiate new collections.

7.5 Contributions

Our work shows that it is both possible to identify a large group of storage as garbage and to do a comprehensive recycling, while the system is running. Our solution tolerates temporary and partial failures, a general challenge when using distributed systems instead of single computer systems. The solution has been implemented in the distributed, object-based system, Emerald. In summary we have achieved:

A Comprehensive and Concurrent Garbage Collector in a Distributed System.
The implementation of the comprehensive garbage collector in Emerald works concurrently with user activity. To our knowledge it's the first working implementation of comprehensive and concurrent garbage detection in distributed systems.

Robust Garbage Collection in Distributed Systems.
Our implementation works even in the case of partial failures of nodes in the distributed system. The global collector simply waits for unavailable nodes to become available, while the local collectors are able to collect local garbage independently.

The implementation is extremely robust to node failures at any time, and as long as nodes becomes pair-wise available the collection will finally complete its job even if some nodes are not available at the end.

Implementation of the Faulting Mechanism.
We have applied the idea of faulting on object invocation used by remote invocations to garbage collection by introducing a garbage collection fault. Thus the *garbage collector invariant*: *Mutators only access traversed and marked objects* is retained by protecting known, but not traversed, objects.

7.6 Future Directions

7.6.1 Orphans

Sometimes it is possible to determine that processes will never affect the system by other means than using resources. Such orphans may be detected by the scheme proposed by [Kafura 90]. Our detection mechanism may be extended to reclaim this kind of orphans also by changing the definition of the root set. Following Kafura's scheme the liveness of each process is dynamically determined by following references from the process to see whether it has access to a live part of the system, i.e., an object reachable from a live process, a persistent object or a communication path to the environment, e.g., an i/o-stream.

7.6.2 Off-line Garbage Detection

It is a general observation that: *garbage stay garbage*. Thus the garbage property is stable. This enables us to use a picture of the whole system instead of the system itself for garbage detection. The garbage detection may use the copy and run off-line to the rest of the system.

When the off-line garbage detection is finished, a list of garbage objects is send to the running system, as these may be reclaimed.

In Emerald, system objects containing executable code may be identified as garbage when the object using the code cease to exist. Later, a new object may be instantiated, that needs the same code. If the code has been collected in between, the code is re-loaded from the stable storage like a first-timer. If the code is still loaded, the new object will reuse the existing code. Thus for efficiency purpose, the garbage property: *garbage stay garbage* is violated in the Emerald prototype. The implemented garbage collector does, however, take this violation into account to preserve correctness.

7.6.3 Object-Oriented Kernel Design

The Emerald kernel has grown out of old kernel modules from the Eden kernel. Further development has modified the kernel to its current size of over 100 Kbytes (more than 50,000 lines of C programs). It is modular in design, but based on traditional C programming practice, employing header files, etc.

Our experience while maintaining the kernel have been that many modules would have benefited from a more object-oriented design. Instead, we have added new features by either copying or rewriting existing code. Large part of the code has been copied and a few changes applied to make the copy behave like the new feature. Other parts has been rewritten as classes, that are instantiated for the different purposes. Examples are:

Location
: The garbage collector needs to find objects in the same way that objects are located for remote invocations, thus the implementation could be reused if it had been build as a generic service.

Mobility
: Checkpoint moves objects to stable store nearly as objects are moved between nodes. Thus a generic mobility service could be instantiated for these slightly different purposes.

Translation
: The mechanism to serialize an object for mobility purpose could also be re-used when applying a copying garbage collector. The marchalling and un-marchalling could also be used to move objects for storage compaction purposes on a single node.

7.6.4 A New Object Store

Hierarchical memories may serve our purpose better than running on top of a virtual memory system, which we do not control. Instead, we could build a reactive object store on each node. Stored objects are semi-persistent, i.e. if they are not known from a running process or a persistent directory service, they get automatically garbage collected.

By not using the virtual memory system, we get liberated from viewing the Emerald file system of code files etc. and the swap space as different. Thus disk space may be preserved. Furthermore by allocating immutable objects separately, we prevent the overhead of write-back, when swapping out immutable object pages.

7.6.5 Room for improvements

The study of memory access in a hierarchical memory design reveals a need to make the main application and the memory system cooperate to ensure that the most often used access patterns are supported by the memory system. The memory system must deliver the needed memory on the highest possible level. In a cache, we want to improve the cache hit ratio, in a paging system we want to avoid thrashing, etc.

The Emerald system has no cooperation with these kind of basic memory systems. The allocator does not look at the memory as a virtual memory with pages, nor does it take advantage of any ways to improve caching. The current implementation of the sweeper, called from the allocator, incurs a second access pattern due to its sequential search over the entire virtual memory. The sweep is expected to be largely improved by determine the amount to sweep by the available pages. Moreover, the sweeper should sweep a complete page from its beginning to its end before it returns to the allocator. The sweeper should issues requests for further pages to be swapped in while user programs are allowed to proceed.

7.6.6 General applications of our Scheme

Our solution works for a one language system, i.e. with full knowledge about identification of references. It could however be applied to any system where references are identifiable. The philosophy behind garbage collection may be generalized as *resource management behind the user*. Our garbage detection scheme is not limited to storage reclamation. The garbage detection delivers a first hand description of other wasted resources also.

In fact the management of any resource in a system may be extended with automatic detection and reclamation of unused resources. As long as reachability is the underlying definition, that separates non-garbage from garbage, our garbage detection algorithm may be applied.

This is further simplified in object-based systems where all resources are implicit modeled by objects. This enables our garbage detection algorithm to identify all kind of garbage. Thus the reclamation process should not only recycle the storage but also other kinds of resources associated with the garbage object.

7.7 Final Conclusion

We have achieved our goals of *Comprehensive, Concurrent, and Robust Garbage Collection in Distributed, Object-Based System* by developing and implementing a new dual garbage collection scheme in the Emerald prototype. The scheme is robust to node failures, works concurrently, and collects all garbage.

Chapter 8

References

[Abdullahi 92] Saleh E. Abdullahi, Eliot E. Miranda, and Graem A. Ringwood. Collection schemes for distributed garbage. In Yves Bekkers and Jacques Cohen, editors, Memory Management, International Workshop IWMM 92, Proceedings published in: *Lecture Notes in Computer Science* 637, pages 43–81, INRIA, IRISA, and ACM Sigplan, Springer Verlag, St. Malo, France, September 1992.

[Almes 85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[Appel 88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, Proceedings in: *SIGPLAN Notices* 23(7), pages 11–20, ACM, SIGPLAN, Association for Computing Machinery, Georgia, USA, July 1988.

[Appel 91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-IV Proceedings in: *SIGPLAN Notices* 26(4), pages 96–107, ACM SIGARCH/SIGOPS/SIGPLAN and IEEE Computer Society, TC MM / TC VLSI / TC OS, ACM Press, Santa Clara, California, USA, April 1991. Simultaneous published as SIGARCH *Computer Architecture News* 19(2) and SIGOPS *Operating Systems Review* 25, special issue.

[Augusteijn 87] Lex Augusteijn. Garbage collection in a distributed environment. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, PARLE'87, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Proceedings published in: *Lecture Notes in Computer Science* 259, pages 75–93, ESPRIT, Eindhoven, The Netherlands, Springer-Verlag, June 1987.

[Baden 83] Scott B. Baden. Low-overhead storage reclamation in the Smalltalk-80 virtual machine. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 19, pages 331–342, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Baker 77] Henry G. Baker, Jr. and Carl Hewitt. *The Incremental Garbage Collection of Processes.* AI Memo 454, AI Lab/MIT, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, December 1977.

[Baker 78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bal 89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[Bartlett 88] Joel F. Bartlett. *Compacting Garbage Collection with Ambiguous Roots.* WRL Research Report 88/2, Digital, Western Research Laboratory, Palo Alto, CA, USA, February 1988.

[Bartlett 89] Joel F. Bartlett. *Mostly-Copying Garbage Collection Picks Up Generations and C++.* WRL Technical Note TN- 12, Digital, Western Research Laboratory, Palo Alto, CA, USA, October 1989.

[Beemster 90] Marcel Beemster. Back-end aspects of a portable POOL-X implementation. In Pierre America, editor, Parallel Database Systems (PRISMA Workshop) Proceedings published in: *Lecture Notes in Computer Science* 503, pages 193–228, PRISMA project, supported by the Dutch *Stimuleringsprojectteam Informaticaonderzoek (SPIN)*, Springer-Verlag, Noordwijk, The Netherlands, September 1990.

[Bekkers 92] Yves Bekkers and Jacques Cohen, editors. Memory Management, International Workshop IWMM 92, Proceedings published in: *Lecture Notes in Computer Science* 637, INRIA, IRISA, and ACM Sigplan, Springer Verlag, St. Malo, France, September 1992.

[Ben-Ari 84] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.

[Bennett 87] John K. Bennett. The design and implementation of Distributed Smalltalk. In OOPSLA'87, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Proceedings published in: *SIGPLAN Notices* 22(12), pages 318–330, ACM SIGPLAN, Association for Computing Machinery, Orlando, Florida, USA, October 1987.

[Bevan 87] David I. Bevan. Distributed garbage collection using reference counting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, PARLE'87, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Proceedings published in: *Lecture Notes in Computer Science* 259, pages 176–187, ESPRIT, Springer-Verlag, Eindhoven, The Netherlands, June 1987.

[Bevan 89] David I. Bevan. An efficient reference counting solution to the distributed garbage collection problem. *Parallel Computing*, 9(2):179–192, 1988/89.

[Bishop 77] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection.* PhD thesis, LCS/MIT, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, May 1977.

[Black 85] Andrew P. Black. Supporting distributed applications: experience with Eden. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 181–193, Association for Computing Machinery, ACM Press, December 1985.

[Black 86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In OOPSLA'86, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Proceedings published in: *SIGPLAN Notices* 21(11), pages 78–86, October 1986.

[Black 87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[Bobrow 80] Daniel G. Bobrow. Managing reentrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[Boehm 88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice & Experience*, 18(9):807–820, September 1988.

[Boehm 91] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, Proceedings in: *SIGPLAN Notices* 26(6), pages 157–164, ACM SIGPLAN, ACM Press, Toronto, Ontario, Canada, June 1991.

[Bronnenberg 89] Wim Bronnenberg. POOL and DOOM — A survey of Esprit 415 subproject A, Philips Research Laboratories. In E. Odijk, M. Rem, and J.-C. Syre, editors, PARLE'89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures, Proceedings published in: *Lecture Notes in Computer Science* 365, pages 356–373, ESPRIT, Eindhoven, The Netherlands, Springer-Verlag, June 1989.

[Chin 91] Roger S. Chin and Samuel T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[Cohen 81] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[Collins 60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[Demers 90] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: framework and implementations. In *ACM Symposium on Principles of Programming Languages, 17. annual Symposium, Conference Record*, pages 261–269, ACM, SIGPLAN, Association for Computing Machinery, San Francisco, CA, USA, January 1990.

[Detlefs 90] David L. Detlefs. *Concurrent, Atomic Garbage Collection*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, USA, October 1990. Technical Report CMU-CS-90-177.

[Deutsch 76] L.Peter Deutsch and Daniel G. Bobrow. An efficient, incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[Dijkstra 76] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In F. L. Bauer and K. Samelson, editors, Language Hierarchies and Interfaces, *Lecture Notes in Computer Science* 46, pages 43–56, NATO Scientific Affairs Division, European Research Office, National Science Foundation, Springer-Verlag, 1976. International Summer School 1975 in Marktoberdorf.

[Dijkstra 78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[Eckart 87] J. Dana Eckart and Richard J. LeBlanc. Distributed garbage collection. In Thomas Turba, editor, ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, Proceedings in: *SIGPLAN Notices* 22(7), pages 264–273, ACM SIGPLAN and IEEE Computer Science, TC Computer Languages, Association for Computing Machinery, St.Paul, Minnesota, USA, June 1987.

[Fenichel 69] R. Fenichel and J. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[Gelernter 60] H. Gelernter, J. R. Hansen, and C. L. Gerberich. A FORTRAN-compiled list-processing language. *Journal of the ACM*, 7:87–101, April 1960.

[Goldberg 89] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. In ACM SIGPLAN'89 Conference on Programming Language Design and Implementation, Proceedings in: *SIGPLAN Notices* 24(7), pages 313–321, ACM SIGPLAN, Association for Computing Machinery, Portland, Oregon, USA, June 1989.

[Goldberg 91] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, Proceedings in: *SIGPLAN Notices* 26(6), pages 165–176, ACM SIGPLAN, ACM Press, Toronto, Ontario, Canada, June 1991.

[Hudak 82] Paul Hudak and Robert M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In Daniel P. Friedman and David S. Wise, editors, *1982 ACM Symposium on LISP and Functional Programming, Conference Record*, pages 168–178, ACM SIGPLAN/SIGACT/SIGART, Association for Computing Machinery, Pittsburgh, Pennsylvania, USA, August 1982.

[Hughes 85] John Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud, editor, Functional Programming Languages and Computer Architecture, Proceedings published in: *Lecture Notes in Computer Science* 201, pages 256–272, Springer-Verlag, Nancy, France, September 1985.

[Hutchinson 87a] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, January 1987. Technical Report 87-01-01.

[Hutchinson 87b] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. *The Emerald Programming Language Report.* Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, Washington, October 1987. Also available as DIKU Report (Blue series) no. 87/22, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark and as TR no. 87-29, Department of Computer Science, University of Arizona, Tucson, Arizona.

[Jul 87] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 105–106, Association for Computing Machinery, December 1987. Extended abstract only; full paper published as [Jul 88b].

[Jul 88a] Eric Jul. *Object Mobility in a Distributed Object-Oriented System.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, December 1988. Technical Report no. 88-12-6. Also available as DIKU Report (Blue series) no. 89/1 from Department of Computer Science, University of Copenhagen, Denmark.

[Jul 88b] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobilty in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Juul 91] Niels Christian Juul. Workshop: Garbage Collection in Object-Oriented Systems. In Jerry L. Archibald and K. C. Burgess Yakemovic, editors, OOPSLA/ECOOP'90, Conference on Object-Oriented Programming: Systems, Languages, and Applications. European Conference on Object-Oriented Programming, Addendum to the Proceedings, published as *SIGPLAN Notices* Special Issue, ACM SIGPLAN, ACM Press, Ottawa, Canada, August 1991.

[Juul 92] Niels Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In Yves Bekkers and Jacques Cohen, editors, Memory

Management, International Workshop IWMM 92, Proceedings published in: *Lecture Notes in Computer Science* 637, pages 103–115, INRIA, IRISA, and ACM Sigplan, Springer Verlag, St. Malo, France, September 1992.

[Kafura 90] Dennis Kafura, Douglas Washabaugh, and Jeff Nelson. Garbage collection of actors. In Norman Meyrowitz, editor, OOPSLA/ECOOP'90, Conference on Object-Oriented Programming: Systems, Languages, and Applications. European Conference on Object-Oriented Programming, Proceedings published in: *SIGPLAN Notices* 25(10), pages 126–134, ACM SIGPLAN, Association for Computing Machinery, Ottawa, Canada, October 1990.

[Knuth 68] Donald E. Knuth. *Fundamental Algorithms.* Volume 1 of *The Art of Computer Programming*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1968.

[Kolodner 89] Elliot Kolodner, Barbara Liskov, and William Weihl. Atomic garbage collection: managing a stable heap. In James Clifford, Bruce Lindsay, and David Maier, editors, Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data in *SIGMOD RECORD* 18(2), pages 15–25, ACM SIGMOD, Association for Computing Machinery, Portland, Oregon, USA, June 1989.

[Kolodner 90] Elliot K. Kolodner. Atomic incremental garbage collection and recovery for a large stable heap. In A. Dearle, G. Shaw, and S. Zdonik, editors, *Implementing Persistent Object Bases: Principles and Practice*, pages 193–206, Morgan-Kaufmann Publishers, San Mateo, CA, USA, September 1990. Fourth International Workshop on Persistent Object Systems at Martha's Vineyard, MA, USA, September 1990.

[Kolodner 92a] Elliot K. Kolodner. *Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap.* PhD thesis, LCS/MIT, Laboratory for Computer Science, Massachusetts Institute of Technology, February 1992. Technical report MIT/LCS/TR-534.

[Kolodner 92b] Elliot K. Kolodner and William E. Weihl. Atomic incremental garbage collection. In Yves Bekkers and Jacques Cohen, editors, Memory Management, International Workshop IWMM 92, Proceedings published in: *Lecture Notes in Computer Science* 637, pages 365–387, INRIA, IRISA, and ACM Sigplan, Springer Verlag, St. Malo, France, September 1992.

[Kung 77] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science*, pages 120–131, IEEE, Providence, Rhode Island, USA, IEEE, New York, USA, October 1977.

[Lang 92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92)*, ACM SIGPLAN and ACM SIGACT, Association for Computing Machinery, Albuquerque, New Mexico, USA, January 1992.

[Lazowska 81] Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Fischer, Robert J. Fowler, and Stephen C. Vestal. The architecture of the Eden system. In *Proceedings of the Eigth ACM Symposium on Operating Systems Principles*, pages 148–159, Association for Computing Machinery, December 1981.

[Lermen 86] Claus-Werner Lermen and Dieter Maurer. A protocol for distributed reference counting. In William L. Scherlis and John H. Williams, editors, *1986 ACM Symposium on LISP and Functional Programming, Proceedings of,* pages 343–350, ACM SIGPLAN/SIGACT/SIGART, Association for Computing Machinery, Cambridge, Massachusetts, USA, August 1986.

[Lester 89] David R. Lester. An efficient distributed garbage collection algorithm. In E. Odijk, M. Rem, and J.-C. Syre, editors, PARLE'89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures, Proceedings published in: *Lecture Notes in Computer Science* 365, pages 207–223, ESPRIT, Springer-Verlag, Eindhoven, The Netherlands, June 1989.

[Lieberman 83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[Liskov 86] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th annual ACM Symposium on Principles of Distributed Computing (PODC'5)*, pages 29–39, Association for Computing Machinery, Vancouver (Canada), August 1986.

[Mancini 91] Luigi V. Mancini, Vittoria Rotella, and Simonetta Venosa. Copying garbage collection for distributed object stores. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, IEEE Computer Society, TC Distributed Processing, Pisa, Italy, September 1991.

[McCarthy 60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[Minsky 63] M. L. Minsky. *A LISP Garbage Collector Using Serial Secondary Storage*. AI Memo 58, AI Lab/MIT, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, October 1963.

[Mohamed Ali 84] Khayri Abdel-Hamid Mohamed Ali. *Object-Oriented Storage Management and Garbage Collection in Distributed Processing Systems*. PhD thesis, The Royal Institute of Technology, S-100 44 Stockholm, Sweden, December 1984. Technical Report TRITA-CA-8406.

[Moon 84] David A. Moon. Garbage collection in a large LISP system. In Guy L. Steele Jr., editor, *1984 ACM Symposium on LISP and Functional Programming, Conference Record*, pages 235–246, Association for Computing Machinery, Austin, Texas, USA, August 1984.

[Piquer 91] José M. Piquer. Indirect reference counting: a distributed garbage collection algorithm. In Emile H. L. Aarts, Jan van Leeuwen, and Martin Rem, editors, PARLE'91, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms, Proceedings published in: *Lecture Notes in Computer Science* 505, pages 150–165, ESPRIT, Eindhoven, The Netherlands, Springer-Verlag, June 1991.

[Pixley 88] Carl Pixley. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing*, 3(1):41–50, 1988.

[Plainfossé 92] David Plainfossé and Marc Shapiro. Experience with a fault-tolerant garbage collector in a distributed lisp system. In Yves Bekkers and Jacques Cohen, editors, Memory Management, International Workshop IWMM 92, Proceedings published in: *Lecture Notes in Computer Science* 637, pages 116–133, INRIA, IRISA, and ACM Sigplan, Springer Verlag, St. Malo, France, September 1992.

[Queinnec 89] Christian Queinnec, Barbara Beaudoing, and Jean-Pierre Queille. Mark DURING sweep rather than mark THEN sweep. In E. Odijk, M. Rem, and J.-C. Syre, editors, PARLE'89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures, Proceedings published in: *Lecture Notes in Computer Science* 365, pages 224–237, ESPRIT, Springer-Verlag, Eindhoven, The Netherlands, June 1989.

[Raj 91] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software – Practice & Experience*, 21(1):91–118, January 1991.

[Rudalics 86] Martin Rudalics. Distributed copying garbage collection. In William L. Schelis and John H. Williams, editors, *1986 ACM Symposium on LISP and Functional Programming, Proceedings of*, pages 364–372, ACM SIGPLAN / SIGACT / SIGART, Association for Computing Machinery, Cambridge, Massachusetts, USA, August 1986.

[Schelvis 88] Marcel Schelvis and Eddy Bledoeg. The implementation of Distributed Smalltalk. In S. Gjessing and K. Nygaard, editors, ECOOP'88, European Conference on Object-Oriented Programming, Proceedings published in: *Lecture Notes in Computer Science* 322, pages 212–232, Springer-Verlag, Oslo, Norway, August 1988.

[Schelvis 89] Marcel Schelvis. Incremental distribution of timestamp packets: A new approach to distributed garbage collection. In OOPSLA'89, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Proceedings published in: *SIGPLAN Notices* 24(10), pages 37–48, ACM SIGPLAN, Association for Computing Machinery, New Orleans, USA, 1989.

[Schorr 67] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[Shapiro 90] Marc Shapiro, David Plainfossé, and Olivier Gruber. *A garbage detection protocol for a realistic distributed object-support system.* Rapport de Recherche INRIA 1320, INRIA-Rocquencourt, Paris, France, November 1990.

[Shapiro 91] Marc Shapiro. A fault-tolerant, scalable, low-overhead distributed garbage detection protocol. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, IEEE Computer Society, TC Distributed Processing, Pisa, Italy, September 1991.

[Sharma 91] Ravi Sharma and Mary Lou Soffa. Parallel generational garbage collection. In Andreas Paepcke, editor, OOPSLA'91, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Proceedings published in: *SIGPLAN Notices* 26(11), pages 16–32, ACM SIGPLAN, ACM Press, Phoenix, Arizona, USA, October 1991.

[Spector 82] David Spector. Minimal overhead garbage collection of complex list structure. *SIGPLAN Notices*, 17(3):80–82, March 1982.

[Spek 90] Juul van der Spek. POOL-X and its implementation. In Pierre America, editor, Parallel Database Systems (PRISMA Workshop) Proceedings published in: *Lecture Notes in Computer Science* 503, pages 309–344, PRISMA project, supported by the Dutch *Stimuleringsprojectteam Informaticaonderzoek (SPIN)*, Springer-Verlag, Noordwijk, The Netherlands, September 1990.

[Steele 75] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975. 1975 ACM Student Award Paper: First Place.

[Tel 87] Gerard Tel, Richard B. Tan, and Jan van Leeuwen. The derivation of on-the-fly garbage collection algorithms from distributed termination detection protocols. In F.J.Brandenburg, G. Vidal-Nagnet, and M. Wirsing, editors, STACS'87 4th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings in *Lecture Notes in Computer Science* 247, pages 445–455, Springer-Verlag, Passau, Germany, February 1987.

[Tel 91] Gerard Tel and Friedemann Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. In Emile H. L. Aarts, Jan van Leeuwen, and Martin Rem, editors, PARLE'91, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms, Proceedings published in: *Lecture Notes in Computer Science* 505, pages 137–149, ESPRIT, Eindhoven, The Netherlands, Springer-Verlag, June 1991.

[Ungar 83] David M. Ungar and David A. Patterson. Berkeley Smalltalk: who knows where the time goes. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 19, pages 189–206, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Ungar 84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In Peter Henderson, editor, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *SIGPLAN Notices* 19(5), pages 157–67, Association for Computing Machinery, May 1984.

[Ungar 88] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In OOPSLA'88, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Proceedings published in: *SIGPLAN Notices* 23(11), pages 1–17, ACM SIGPLAN, Association for Computing Machinery, San Diego, California, USA, September 1988.

[Vestal 87] Stephen C. Vestal. *Garbage Collection: An Exercise in Distributed Fault-tolerant Programming.* Technical Report 87–01–03, Department of Computer Science, University of Washington, Seattle, Washington, USA, January 1987. Adaption of PhD Thesis.

[Washabaugh 90] Douglas M. Washabaugh. *Real-Time Garbage Collection of Actors in a Distributed System.* Master's thesis, Virginia Polytechnical Institute and State University, Blacksburg, Virginia, USA, February 1990.

[Watson 87] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, PARLE'87, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Proceedings published in: *Lecture Notes in Computer Science* 259, pages 432–443, ESPRIT, Eindhoven, The Netherlands, Springer-Verlag, June 1987.

[Weizenbaum 62] J. Weizenbaum. Knotted list structures. *Communications of the ACM*, 5(3):161–165, March 1962.

[Weizenbaum 63] J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(10):524–544, September 1963.

[Wentworth 90] E. P. Wentworth. Pitfalls of conservative garbage collection. *Software – Practice & Experience*, 20(7):719–727, July 1990.

[Wester 90] R. H. H. Wester and B. J. A. Hulshof. The POOMA operating system. In Pierre America, editor, Parallel Database Systems (PRISMA Workshop) Proceedings published in: *Lecture Notes in Computer Science* 503, pages 396–423, PRISMA project, supported by the Dutch *Stimuleringsprojectteam Informaticaonderzoek (SPIN)*, Springer-Verlag, Noordwijk, The Netherlands, September 1990.

[White 80] Jon L. White. Address/memory management for a gigantic LISP environment or, GC considered harmful. In *1980 LISP Conference, Conference Record*, pages 119–127, The LISP Conference, P.O.Box 487, Redwood Estate, CA 95044, USA, Stanford University, Stanford, CA, USA, August 1980.

[Wilson 92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, Memory Management, International Workshop IWMM 92, Proceedings published in: *Lecture Notes in Computer Science* 637, pages 1–42, INRIA, IRISA, and ACM Sigplan, Springer Verlag, St. Malo, France, September 1992.

[Zorn 90] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *1990 ACM Conference on LISP and Functional Programming, Proceedings*, pages 87–98, ACM, SIGPLAN/SIGART/SIGSAM, Nice, France, June 1990.

Appendix A

The Emerald System

Emerald is a distributed object-based system with support of *objective expressiveness*, *concurrent execution*, and *object mobility*. The Emerald system is an implementation of the Emerald programming language [Hutchinson 87b, Raj 91] by means of the Emerald compiler [Hutchinson 87a] and run-time system [Jul 88a].

The purpose of this chapter is to provide sufficient information about both the Emerald language and its implementation to understand the Emerald specific details of the thesis. The following sections present a short discussion on the language implementation (Section A.1), and a presentation of the current run-time system (Section A.2). The implementation of processes (Section A.3) and objects (Section A.4) is given next, followed by a description of the object protection and faulting mechanism (Section A.5). The location protocol (Section A.6) is summarized.

This appendix is, however, not a general introduction to Emerald, neither the programming language nor its compiler and run-time system. The overall perspective of this presentation is to present what is needed to understand the problems and solutions from a garbage collection implementors point-of-view.

A.1 Objects in the Emerald Language

Emerald is an imperative, strongly-typed, object-based programming language [Black 86, Black 87, Jul 88b]. It supports *a unified object model* independent of object size, use, and distribution, with special emphasize on *distribution* and *concurrency*. Objects are highly mobile and distribution is made transparent to other facilities.

In general objects reside on exactly one node, though immutable objects may be replicated when asked to move to another node. For efficiency reasons objects are implemented differently depending on their usage. The compiler classifies the objects as either *global*, *local*, or *direct* to enable the most efficient representation in the run-time system.

Direct objects are objects implemented inside other objects for efficiency. They are almost invisible to the run-time system as they are optimized away by the compiler.

Local objects need no mechanism for mobility and global visibility, and hence they are only invoked locally using the traditional procedure call mechanism.

Global objects may be known outside the node where they reside, and include full mechanisms for mobility and *remote invocation*, the Emerald implementation of remote procedure calls used when invoking a non-resident object.

Note, however, that this classification of objects is purely a matter of implementation by the compiler/run-time system. To the Emerald programmer, all objects look alike. This was one of the goals in the design of Emerald, a unified object model with multiple implementations for efficiency; a lesson learned by using the Eden system [Lazowska 81, Almes 85], where programmers were led to use two distinct object models for efficiency [Black 85].

As distribution is transparent in Emerald, the programmer may move objects between the nodes by requests to the run-time system. Moving an object includes the movement of any processes executing inside the object, thus Emerald supports process mobility as well. The processes originate in the optional *process section* of objects. Each process may be described as a thread-of-control from the process section, through each operation invoked in nested calls; the call chain or a stack of activation records when seen from the run-time system. The thread-of-control may be distributed among the nodes as the activation records resides on the same node as the object invocation they represent.

Conceptually, after it has been created, an object persists forever. However, failures may result in the disappearance of objects, and in cases where no other object in the system has a reference to the object, the object may be garbage collected to reuse its resources.

The transparency of distribution also leaves the responsibility to cope with failures, availability, checkpoints and recovery from failures on the programmer. The Emerald language has explicit tools for this purpose. Objects may be *checkpointed* to stable storage on their own and possibly other nodes. When a node restarts after a crash the checkpointed objects are recovered from the stable store. The recovery of each object includes the execution of the optional *recovery section* of the object.

Not only the state of the object is saved at checkpoint time, also the state of attached objects as well as all relevant system objects like those containing the associated code for the objects. The checkpoints are done explicit by executing checkpoint statements in the Emerald program.

The copies on stable storage will be outdated as the original object may continue to mutate. Thus after a node failure the checkpointed objects may pertain a different view of the global system than is actually the reality. It is the obligation of the programmer, who knows the object, to specify in the recovery section of the object, what is needed to adjust the object to the current state of the global system, otherwise inconsistent state of objects may invalidate global or local invariants of the Emerald program.

Checkpointed objects may also be recovered by the system, if the original has disappeared. This is done when the object is needed by other objects and a checkpoint copy is available and may be recovered safely. Safeness is concerned with quorum, to prevent multiple inconsistent copies from being available concurrently.

A.2 The Emerald Run-time System

The Emerald kernels, running on different nodes, constitute the run-time system for Emerald programs. The compiler submits a program for execution by submitting a unique *object identifier (OID)* representing the program to one of the kernels. The kernel loads the code associated with the received OID, creates objects and translate the references necessary to execute the program. Further code may be loaded and translated during this process. The code is native machine code augmented with calls to kernel operations.

While the kernel is running, various user and system objects will be created dynamically and the Emerald processes scheduled for execution. The handling of processes in the kernels are detailed in section A.3. In the following section (A.4) the implementation of user and system objects is described. Global objects facilitate mobility and remote accessibility, but the actual access may be node-local, if the invoker and invoked reside on the same node. To achieve the high local performance in the case where the two objects are on the same node, remote access is handled as an exception. The *Emerald faulting mechanism* is used to turn the invocation of an object, that is not resident, into a remote invocation without adding overhead to the case where the object is resident (Section A.5). The location protocol is used to determine the location of another object, e.g., before doing a remote invocation. The protocol is described in section A.6 where object availability and unavailability is also discussed.

A.3 Process Implementation in Emerald

Each process is represented in the Emerald run-time system by a chain of stack segments. Each stack segment is an Emerald object. When a process grows out of the top of the stack in the current stack segment, another stack segment is allocated, and the two segments are chained. Each time a process does a remote invocation the stack is continued on another stack segment on the remote node. The chain of stack segments may represent any kind of processes. If the process is local, all stack segments resides on the same node, whereas the stack segments of a distributed process are spread among the nodes.

The process may move from one node to another leaving a node gap between two stack segments. Old parts of its stack segments may, however, also be moved before the chain of nested invocations returns. The later happens when an object that moves has a chain of calls passing through itself. Then the activation record representing the call through the moving object must follow the object, and thus, the stack segment it is in must be divided in three parts of which the middle moves with the object. Still all the stack segments are chained, so that they all together represent the current chain of nested calls done by the process.

The Emerald kernel executes user processes on a round-robin basis. The Emerald compiler and kernel ensures that each process will preempt itself very often, i.e., at each invocation call, system call, and end of current statement block, the process lets the kernel decide, whether the process shall continue or be rescheduled to the back of the *ready-to-run queue* (readyQ). These breakpoints of self-preemption (*flick*) are inserted by the system; the Emerald programmer is not aware of how quasi-parallelism is achieved.

Furthermore, the Emerald kernel has a set of internal queues of actions to be done, and mechanisms to ensure execution and synchronization among the different types of processes. The *kernel queue* (taskQ) is the primary list of actions to be done between user processes. Each time a *flick* is done by a user process, the outstanding queue of actions in *kernel queue* is done before a user process may continue. All events external to the kernel, e.g., interrupts from other kernels, i/o-interrupts, etc., are put in the *kernel queue* without being serviced further. Thus it is important to clean up the *kernel queue* very often, and each action must be of limited time, to ensure fast enough service of the external events. Interrupts from external events are received by the Emerald kernel as UNIX signals.

Level	Queues in the kernel	Processes on this level
User level	current	The user process running or doing *flick* right now.
	readyQ	The *ready-to-run* user processes.
	—	Other user processes may be waiting inside monitor and condition queues, defined by the Emerald programmer.
Kernel level	taskQ	Actions to be done by the kernel due to in- and external events.
Interrupt level	—	UNIX signals results in an appropriate kernel task (the corresponding event handler) being inserted in taskQ.

Figure A.1: The levels of execution inside the Emerald kernel

Both user processes and internal tasks may be scheduled for execution at a certain future clock value. When the timer they are waiting for expires, the tasks are scheduled on the *kernel queue*. This task will rescheduled a user process on the *ready-to-run queue*, if it was waiting for the timer.

The various levels of execution inside the Emerald kernel may be summarized as shown in Figure A.1.

A.4 Objects in the Run-time System

Each Emerald kernel has a heap in which the dynamically allocated objects are stored. The heap includes both dynamically allocated structures owned by the kernel (various tables and internal text strings etc.) and objects, which are managed as Emerald objects. The storage allocated and owned by the kernel are not discussed further here. Our focus is on the object graph containing Emerald objects, which may be system or user objects.

On the implementation level, all Emerald objects are tagged with an implementation type and the distinction local/global, as well as resident/non-resident, and replicated applies to all of them. This information is kept as an object descriptor (OD) for each object. The descriptor also contains space for two sets of garbage collector-bits, one for local and one for global garbage collection. The object descriptor is maintained as either a header of an object or as a separate descriptor object. Separate descriptor objects are for objects that move between the nodes, they leave a copy of their descriptor behind them, as the descriptor is augmented with hints to the new location of the object (How to locate an object is described in section A.6). Where needed the OD contains information about the system wide name of the object (the OID).

The system objects implement the basis for running Emerald programs on top of the kernel. To implement a program, the compiler constructs a code object. By using the templates inside the code object the kernel is able to create the objects representing the data. Special objects are created for language concepts like processes (stack segments) and condition queues (condition). In total Emerald supports objects tagged as either of:

Local Data (LODATA) includes a header as OD, and user data of a local or a global, replicated object. The OD contains references to the code describing the layout of the user data (templates).

Global Data (GODATA) contains user data of a global object and a reference to its OD and to the code describing the layout of the user data (templates).

Global Data Descriptor (GOD) is the OD of a global data object. The OD contains information about the status of the data object.

Stack Segment (SS) represent a process. Besides information about the process and its code, this is a stack of activation records. The bottom contains a reference to the previous stack segment, whether resident or not, or identifies the bottom of the process stack.

Stack Segment Descriptor (SSOD) is the OD of a Stack Segment. The OD contains information about the status of the data object, but not the process represented by the chain of Stack Segments.

Code[1] (CODE) is a container of both the executable code for an object and templates describing how internal references in the code are organized, as well as the organization of the data (in a local or global data object) and on the stack when running. References for run-time type checking to abstract type objects is also included. Code objects are immutable, i.e., they may be treated like immutable data objects, which are replicated when needed on multiple nodes.

Code Descriptor (CODEOD) is the OD of the code object. Containing references to code and references for run-time type checking to abstract type objects.

Condition (COND) includes a header as OD and represent a queue, used by processes made waiting inside a *monitor*. The object is distinct from, but tightly connected to the monitor inside the data object using it.

Abstract/Concrete Type (ABCON) is used by the type system of Emerald to do run-time type checking. These objects represent connections between abstract and concrete code objects. No OD is associated with these.

The code objects contain type information to outline the data objects defined by the user. The kernel sees these objects as either local or global objects. The templates in the associated code objects describes how the data shall be interpreted, and thus defines which bytes contain references to other objects. The kernel applies internal type specification to ensure correct interpretation of the bytes of user data. These internal types are also used when the data is allocated on the stack, i.e., in an activation record inside a stack segment. The internal types are:

Data A sequence of bytes containing data not interpretable as references.

ODP A reference to an object descriptor (OD) or object of one of the aforementioned types.

[1]Code is inline in DOTO object, each behaving like a complete unlinked output file from the assembler (UNIX assembler names their output file with suffix ".o", thus the name Doto).

OID A system-wide unique object identification, which may be looked up locally in the local *object table* or system-wide via the *location protocol.*

Variable In general a dual reference to an object containing both the ODP for the object and a reference to an ABCON determine the type of the object.

Address A reference to an entity inside the object.

Vector A counter of elements and a list of references to these element objects.

Monitor A queue of processes, i.e., stack segments, waiting to enter the monitor.

Invoke queue A supplementary queue element header, used to link stack segments, SS, in a double linked queue.

Replicated objects are implemented as resident and local, with special care taken to make a copy instead of a move, when asked to move between nodes.

In principle, each object has an object descriptor (OD) and a unique identification (OID). On each node an object table of all globally known objects (OID, OD) at the node exists. As mentioned the OD may be allocated as a header in the real object. An object, which makes no use of the system-wide unique OID, may for efficiency purpose skip the allocation of OID. OID's are requested from a name server, to achieve uniqueness. Each node runs its own name server, thus the node allocating the OID is part of the identification.

The object graph, which is the target of the mark-phase of the garbage collector, is distributed among the nodes. The object graph consists of both user and system objects. In garbage collection context, Emerald consists of a set of well defined objects, where all references are easily identified. The system objects has kernel support to identify their references and user references are identified by templates in the associated code object. The garbage collection perspective is that all Emerald objects may be managed by the same garbage collector leaving only explicit kernel allocated structures on the heap to be treated separately. These are recycled by explicit deallocation inside the kernel (`free()`).

A.5 The Faulting Mechanism

The *Emerald faulting mechanism* is used where an object needs access to an object not immediately available. The missing availability may be due to the non-residence of the object on the current node, or the object being broken (invalidated by some failed process). During its initialization, the object will also be unavailable to ensure that only fully initialized objects are accessed. A non-resident object may be resident on another node or in the stable store of the node.

The faulting mechanism ensures that appropriate actions are taken by the kernel in these cases to bring the object into an available position while delaying the invoking process temporarily. If the object is remotely available, a *remote invocation* is established, and if the object is not available at all, the *unavailable section* of the invoking object is called. The kernel may enable the faulting mechanism thereby protecting the object by setting certain status-bits in the OD of the object. The mechanism is available for global objects only, i.e., Emerald objects implemented as GODATA, SS, and CODE. All these has separate ODs. The OD contains both a general FROZEN-bit to trigger a failure and several bits, each indicating a different reason to protect the object.

The main purpose of the faulting mechanism is to limit the overhead when accessing a global object locally, i.e., to turn a local invocation of an object into a remote invocation when the object is non-resident only. Global objects facilitate mobility and remote accessibility, but the actual access may be node-local, if the invoker and invoked reside on the same node. To achieve the high local performance in the case where the two objects are on the same node, remote access is handled as an exception. The *Emerald faulting mechanism* is used to turn the invocation of an object, that is not resident, into a remote invocation without adding more overhead than a single test instruction to the case where the object is resident.

Even a remote invocation will eventually be executed as a local invocation. The fault handling mechanism of Emerald implements a remote invocation as three successive steps:

1. Move the current stack top (the process invoking) to the node hosting the invoked object.
2. Perform the invocation locally on the node hosting the object.
3. Return the top of the stack back to the invoking node.

Thus 2. is performed on the destination node, whereas 1 and 3 both include packing and sending, and receiving and unpacking on the two nodes.

A.6 The Location Protocol

The location protocol is used to determine the location of another object, e.g., before doing a remote invocation. This section presents the protocol and how the availability and unavailability of objects is achieved also.

The Emerald Location Protocol is based on Fowlers work. It is detailed in [Jul 88a] and summarized here. It starts by using the most simple and common step, and fails through a number of more general steps if the former steps fail to succeed.

1. If the object is available on the same node as the requester a simple lookup in the object table is enough.
2. Failing 1. results in a system-wide identification (OID) being returned and a location hint (last known residence as seen from this node) which may be out of date. The hint is followed by asking the hinted node if the object is there. A new hint is piggy-backed if the object were not there either.
3. If 2 fails also, we update the hint and resort to the global *locate request*, a reliable broadcast to all nodes. The node hosting the object will answer the request.
4. Also failing 3 implies that the object is not on any available nodes. A request to recover the object from a checkpoint copy is the tried.
5. Should recovery be impossible also, the object requesting the unavailable object is interrupted with an unavailable exception. Objects may define their own unavailable handlers in Emerald to take care of such situations — the failure is propagated down

through the call stack until an object with an unavailable handler or the bottom of the stack is met.

Due to individual node-failures, objects may be lost permanently. An object trying to access such an unavailable object will be given an unavailable exception from the run-time system.

Appendix B

The Specification of the Emerald Garbage Collection Scheme

This specification of the full Emerald garbage collection scheme is based on the design discussed in Chapter 3 and 4 of the thesis. The most important issues on implementing the scheme in the Emerald prototype are discussed in Chapter 5. The full garbage collection scheme in Emerald consists of two sets of collectors. One set of collectors cooperates on a global collection while the other set of collectors does an independent node-local collection on each node. The global collection is comprehensive but potentially slow, whereas the node-local is more expedient but conservative. The two collectors on each node are called the *global* and the *local garbage collector*, respectively. Their basic algorithm is the same; but the root set is larger in the local garbage collector, whereas the global must facilitate inter-node cooperation and distributed termination detection.

The chapter contains a specification of the full Emerald garbage collection scheme based on the rationale given in the previous chapters. The first section gives a brief outline of the modules that constitute the entire garbage collection scheme and the primary data structures (Section B.1). The following sections describe the two garbage detectors, their interaction, and their common storage reclamation, i.e., the local marker process (Section B.2), the global marker process (Section B.3), the synchronization of the local and global marker processes (Section B.4), and the sweeper process (Section B.5). Finally, the distributed communication scheme is described (Section B.6).

B.1 Overview of the Specification

Both collectors are divided into a *marker* and a *sweeper* process, plus additional processes acting as *event handlers* driven by external events. Moreover the local and global sweeper processes are put together as one process interleaved with the storage allocator on each node. Thus we have two marker processes, a *local marker* (Section B.2) and a *global marker* (Section B.3). A picture of all processes and event handlers for each node was given in Section 5.1 as Figure 5.1, page 82.

On each node the two marker processes work on their own set of data structures and utilize their own set of mark-bits associated with each object. Thus each object has room for both local and global mark-bits in its object descriptor (OD). The descriptor contains two local and two global garbage collector-bits for coloring and bits to indicate that the object is protected (FROZEN) and why (LOCALGCFROZEN, GLOBALGCFROZEN, etc.). The marking scheme is implemented by using the mark-field of the objects to mark the object with the current garbage collection cycle number (= *black*). Lower *cycle*s in the mark-field are interpreted as *white*. The width of the mark-field (2 bits) restrict the stored value to be *cycle* modulo 4. The same mark is thus used again later, and hence the sweeper process must know about the current interpretation and stop reclaiming objects marked similar to current live objects.

The color *gray* is represented by:

- putting a reference to the object in a gray set,
- protecting the object, and
- marking the object with the current *cycle*.

The gray set is divided in two sets for the local marker process and three sets for the global. The local marker process uses:

A temporary gray set for references to non-protectable objects, that are *Shade*. The marker process ensures that these objects are definitely traversed and their references *Shade* before it finish its current step. Thus the *temporary gray set* is a stack representing those objects, which must be traversed together with the current object traversal before any mutator is allowed to run.

A local gray set for references to resident and protectable objects, that are *Shade*. Also the additional root objects goes into this set.

The global marker process uses two similar sets, and:

A non-resident gray set for references to non-resident objects, that are *Shade*. The set is resident, but it references non-resident objects.

Both marker processes take advantage of the common *sweeper process* (Section B.5) and the *garbage collection fault handler* (Section B.2). The sweeper process reclaims objects marked with a *cycle* less than the *cycle* of the latest completed mark-phase. The global collector further utilize the handlers for message exchange between nodes (Section B.6). The message exchange ensures that the global collectors will cooperate during the mark-phase by exchanging references from their *non-resident gray sets* and detects when the mark-phase is finished by a distributed termination detection algorithm using two-phase commitment.

B.2 The Local Marker Process

The *local marker process* is implemented by Algorithm 7. As indicated, the marker runs mostly in parallel with other activities in the system. Phase L1 and L4 are however done non-interruptible (atomically), to ensure consistency during the start and the end of the mark-phase. The Garbage Collection Fault Handler, described by Algorithm 8 ensures that mutators only access *black* objects. Concurrency is achieved by self-preemptive scheduling, i.e., the marker process *flicks* in Phase L2 – L3 each time the *temporary gray set* is empty. *Flick* is a voluntary suspension of the process. The marker process is resumed by the run-time system in between other system activities and user processes. The *Kernel Dispatcher System* takes care of this by managing the self-preemptive garbage collector processes on a priority level, between system interrupts (signals) and Emerald processes (user processes). At most one garbage collector process is resumed once between the progressing Emerald processes.

Algorithm 7 (Local Marker Process)
A new local collection is requested on node i:

Phase L0	7.1	Wait until the previous local cycle, $cycle_i$ is finished. Force the *Sweeper* to finished its work with the oldest local garbage collection cycle number, $cycle_i - 2$[1] if not done yet. Wait until no *Global Marker* is working with a non-empty *gray set* of objects resident on this node.
Phase L1[2]	7.2	Increment $cycle_i$, thus defining the new *black* color.
	7.3	Suspend all mutators on this node.
	7.4	Enable the Garbage Collection Fault Handler for resident and arriving mutators.
Phase L2	7.5	For each suspended mutator do:
	7.5.1	Traverse the mutator, and *Shade* all the references found.
	7.5.2	While the *temporary gray set* is non-empty extract one object at a time and traverse the object to *Shade* its references.
	7.5.3	Resume the mutator.
	7.5.4	*Flick*.
	7.6	Establish a set of references to any additional roots of the object graph for this node and *Shade* these. These are standard objects kept available for efficiency, as well as objects granted available to the the run-time system, and objects potentially known from another node (REFERENCEGIVENOUT).
Phase L3	7.7	While the *local gray set* is non-empty:
	7.7.1	Move one object from the *local gray set* to the *temporary gray set*.
	7.7.2	While the *temporary gray set* is non-empty extract one object at a time and traverse the object to *Shade* its references.
	7.7.3	*Flick*.
Phase L4[2]	7.8	Run-time system tables are adjusted to reflect that *white* objects are now considered garbage.
	7.9	Change the interpretation of the mark-field, as *white* objects (marked with $cycle_i - 1$) are garbage now. Inform the *Sweeper*, that the *Local Marker* of $cycle_i$ is now finished, thus objects marked with a cycle lower than $cycle_i$ are to be reclaimed.
	7.10	Enable pending collectors to try running again.

Shade a reference:

If the reference is to a resident, *white* object, mark the object *black*, put it in the *gray set*, and protect the object. If the object is non-protectable, it is put in the *temporary gray set* instead of the *local gray set*. A reference to a non-resident object is skipped.

A 7

[1] The mark-field is two-bit wide, thus only the latest 4 cycle numbers are recognizable. Current cycle modulo 4 is stored as *black*, and thus the sweeper process must not be more that 2 cycles behind a running marker process.

[2] Phase L1, step 7.2 – 7.4 is done atomically, i.e., as one indivisible operation when viewed from the mutators or any other process on the same node. The same yields the steps from the end of step 7.7 to the end of Phase L4, i.e., step 7.8 – 7.10.

The *garbage collection fault handler* may be enabled for either local, global, or both collectors, depending on whether a local, a global, or both markers have not terminated yet.

When the handler is invoked due to a LocalGCFrozen protection[4], and the handler is already enabled for local garbage collection faults, it does the actions described in Algorithm 8 for the local collector using the mark-fields and data structures associated with local collection.

Algorithm 8 (Garbage Collection Fault Handler)
The garbage collection fault handler on node i is invoked due to a fault when a mutator wants to access a protected object (protected due to LocalGCFrozen respectively GlobalGCFrozen*):*

8.1	Suspend the mutator.
8.2	Move the reference to the object from the *local gray set*, respectively the *global gray set*, to the *temporary gray set*.
8.3	While the *temporary gray set* is non-empty extract one object at a time and traverse the object to *Shade* its references.
8.4	Unprotected the object and remove the protection reason handled here, i.e., LocalGCFrozen, respectively GlobalGCFrozen.
8.5	Resumed the mutator.

A 8

As shown, similar actions are taken in both the case of a LocalGCFrozen, and a GlobalGCFrozen fault using the local and global mark-fields and data structures respectively. If both collectors have enabled the fault handler and protected the object, Algorithm 8 is done twice, i.e., both for the local and global case.

B.3 The Global Marker Process

The *global marker process* is implemented by Algorithm 9. The marker processes on the different nodes communicate be sending messages. *Send(m, n)*, i.e., send message m to node n, shall be interpreted as a reliable communication of the message m from the current node to node n. Actually it is implemented in the Emerald prototype as a *repeated broadcast* until a node accept the message and returns an acknowledgment. The repeated broadcast is repeated by doubling the interval between successive broadcasts, to limit the noise interfered on other nodes.

Algorithm 9 (Global Marker Process)
A global collection (number $cycle_i$) is requested on node i:

Phase G0	9.1	Drop any non-terminated *Global Marker* with a smaller *cycle*, and any non-terminated *Local Marker* on this node. Force the *Sweeper* to finished its work with the old global garbage collection cycle number, $cycle_i - 2$ if not done yet.

[4] The object may be protected according to local collection, global collection, or other purposes like non-availability and non-residence.

Phase G1[5]	9.2	Increment $cycle_i$ thus defining the new *black* color.
	9.3	Suspend all mutators on this node.
	9.4	Enable the Garbage Collection Fault Handler for resident and arriving mutators, and the *Remote Shade Handlers* for incoming request/reply. Broadcast a *GGC-Started* message to tell the other nodes that global garbage collection number $cycle_i$ has started on node i.
Phase G2	9.5	For each suspended mutator do:
	9.5.1	Traverse the mutator, and *Shade* all the references found.
	9.5.2	While the *temporary gray set* is non-empty extract one object at a time and traverse the object to *Shade* its references.
	9.5.3	Resume the mutator.
	9.5.4	*Flick.*
	9.6	Establish a set of references known at node i to any additional roots of the distributed object graph and *Shade* these. These are standard objects kept available for efficiency, as well as objects granted available to the the run-time system.
Phase G3–6	9.7	Until *global termination*[6] is detected do:
Phase G3	9.7.1	While the *global gray set* is non-empty:
	9.7.1.1	Move one object from the *global gray set* to the *temporary gray set.*
	9.7.1.2	While the *temporary gray set* is non-empty extract one object at a time and traverse the object to *Shade* its references.
	9.7.1.3	*Flick.*
Phase G4	9.7.2	While the *non-resident gray set* is non-empty:
	9.7.2.1	Send a ShadeRequest to the nodes hosting the objects referenced from this set.
	9.7.2.2	*Flick.*
Phase G5–6	9.7.3	While both sets are empty (no *gray* objects), start a distributed termination detection by sending a Status to the other nodes and *flick.*
Phase G7[5]	9.8	Run-time system tables are adjusted to reflect that *white* objects are now considered garbage.
	9.9	Change the interpretation of the mark-field, as *white* objects (marked with $cycle_i - 1$) are garbage now. Inform the *Sweeper*, that the *Global Marker* of $cycle_i$ is now finished, thus objects marked with a cycle lower than $cycle_i$ are to be reclaimed.
	9.10	Enable pending collectors to try running again.

A 9

[5] Phase G0–G1, step 9.1 – 9.4 is done atomically, i.e., as one indivisible operation, when viewed from the mutators or any other process on the same node. The same yields the steps from the end of Phase G6 through the entire Phase G7 (step 9.8 – 9.10).

[6] Distributed termination detection is described in details in section B.6.

Shade a reference:

If the reference is to a resident, *white* object, put it in the *gray set* and protect the object. If the reference is to a *white*, non-resident object, the reference is put in the *non-resident gray set.*

Screening incoming messages on node, i:

Each received message, *mess*, is screened to ensure that $cycle_{mess} \leq cycle_i$. If not, the *Global Marker* is called to start its $cycle_{mess}$.

The exchange of SHADEREQUEST, SHADEREPLY, and STATUS messages are described further in Section B.6.

B.4 Synchronization of the Local and Global Marker Processes

As indicated by the *flick*-statement in both marker processes (Algorithm 7 and 9) these processes may run in parallel. Their quasi-concurrent execution is, however, constrained. According to section 4.5.2, the local and global marker processes are blocked by each other as they are not allowed both to have a non-empty gray set concurrently. Figure B.1 illustrates an execution of the two processes with *flick*-points marked. Those point where one of the processes may allow the other to run, i.e., they *flick* and have an empty gray set, are annotated as enable points.

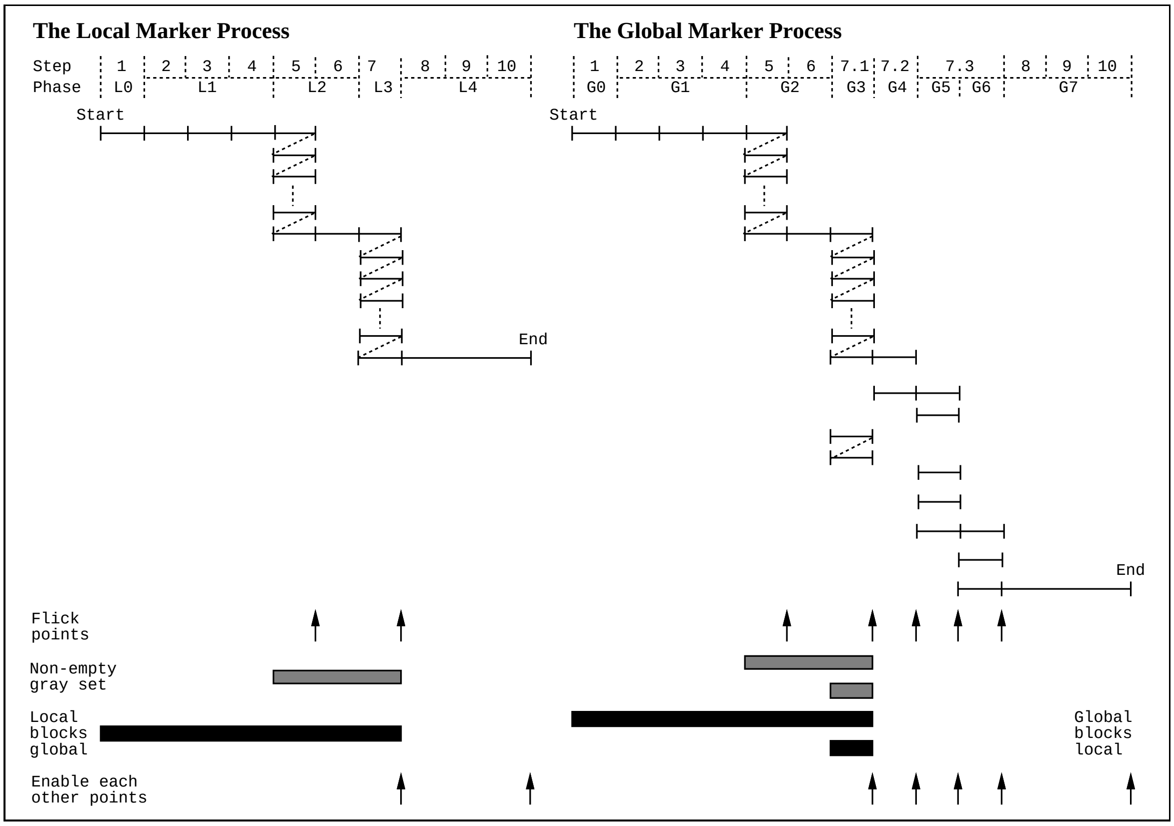


Figure B.1: The points of synchronization between the global and local marker processes

B.5 The Sweeper Process

The common *sweeper process* is implemented by Algorithm 10. When enabled, it traverse the object storage (sequentially) and reclaim objects marked as garbage in their local or global mark-field according to the following scheme. The scheme depends on the status of the corresponding collector, i.e., whether a local and/or global marker process is running concurrently.

When the marker process of the $cycle^{th}$ garbage collection is finished, the sweeper process is enabled to reclaim objects marked with $cycle - 1$. When the next marker process (of $cycle + 1$) is started the sweeper process is disallowed to reclaim anymore objects marked with $cycle - 3$ as $(cycle - 3)$ modulo $4 = (cycle + 1)$ modulo 4. If the sweeper has not finished sweeping for this old cycle, it is forced to do so in Phase 1 of both the global and local marker process.

When the sweeper process is enabled by one or more local or global marker processes, the storage allocator will invoke it, each time more storage is requested. This ensures that the garbage will be reclaimed in a speed comparable to the current allocation speed. Each time, it is invoked, it will try to reclaim the same amount of storage, as requested from the allocator.

The sweeper process makes an infinite sweep of the object store by viewing the heap cyclic, i.e., when it reaches the top of the heap, it continues from the bottom again. During each invoke it will at most traverse the entire heap once. It also stops, when enough has been reclaimed or no more mark-values is enabled.

Algorithm 10 (Sweeper Process)
Sweeping of storage on node i, resumed when enabled and more storage is requested:

10.1 While enabled and until enough has been reclaimed, get the next object from the heap:

10.1.1 If the object is marked with a mark enabled as garbage, the object is returned to the allocator, i.e., to the appropriate free-list.

10.1.2 The requested amount of storage is reduced with the amount just reclaimed.

A 10

B.6 The Distributed Communication Scheme

The global marker process in Algorithm 9 takes advantage of the garbage collection fault handler also used by the local marker process and described by Algorithm 8. Furthermore, the global marker process utilizes four more handlers for arriving messages, i.e., ShadeRequest, ShadeReply, Status, and Terminate.

The actions taken by these handlers depend on the actual state of the running collector. The global marker process on each node was structured as 8 phases from G0 to G7 (See Algorithm 9). Table B.1 describes the arriving events and actions taken according to the current phase, plus the new phase. The actions of this state/transition table is labeled A–G and outlined by Algorithm 11, 12, 13, and 14.

Furthermore, the system ensures that any two communicating nodes are running in the same *cycle* of the global marker process. The screening of incoming messages in Algorithm 9 ensures this.

Current Phase / Events	G1	G2	G3	G4	G5	G6	G7	G0
ShadeRequest	—	A/2	A/3	A/3	A/3	A/3	—	—
ShadeReply	—	—	B/3	C/4-5	—	—	—	—
Status	skip	D/2	D/3	D/4	E/5-6	F/5-6-7	skip	skip
Terminate	—	—	—	—	G/7	G/7	skip	skip

Table B.1: The phase/event-action/transitions of the global collector on each node The actions named A–G are detailed in Algorithm 11–14. The fields marked "—" represent illegal combination of event and phase. The fields marked skip are situations which may happen, but they are skipped as the event has no relevant meaning during that phase. The phases correspond to the phases listed in Algorithm 9.

Algorithm 11 (Shade Request Handler)
On arrival of a ShadeRequest *containing a set of object references, SR, on node i while the Marker Process is in phase G2–G6:*

A 11.1 Each of the received references in SR to resident objects is *Shade*.

11.2 One ShadeReply is send explicit to the requesting node containing references to all the resident (and new *Shade*) objects as a common acknowledgment.

A 11

Algorithm 12 (Shade Reply Handler)
On arrival of a ShadeReply *containing a set of object references, SR, on node i while the Marker Process is in phase G3:*

B 12.1 Delete the acknowledged references given in SR from the *non-resident gray set.*

On arrival of a ShadeReply *containing a set of object references, SR, on node i while the Marker Process is in phase G4:*

C 12.1 Delete the acknowledged references given in SR from the *non-resident gray set.*

12.2 If both the *non-resident gray set* and the *global gray set* have become empty this node is finished for now:

12.2.1 Increment $timestamp_i$.

12.2.2 Send Status $< i, timestamp_i >$.

12.2.3 Set next phase to G5.

A 12

Algorithm 13 (Status Handler)
The Status *signal from node j (with $timestamp_j$, and a set of status for other nodes $\{< n, timestamp_n >\}_n$) received on node i, during phase G2–G4:*

D 13.1 Record status information received: $< j, timestamp_j >$ and any more recent information from the set $\{< n, timestamp_n >\}_n$.

The STATUS *signal from node j (with $timestamp_j$, and a set of status for other nodes $\{< n, timestamp_n >\}_n$) received on node i, during phase G5:*

E 13.1 Record status information received: $< j, timestamp_j >$ and any more recent information from the set $\{< n, timestamp_n >\}_n$.

13.2 If status has been received from all nodes, it is time to get commitment from all other nodes, and the next phase is set to G6.

The STATUS *signal from node j (with $timestamp_j$, and a set of status for other nodes $\{< n, timestamp_n >\}_n$) received on node i, during phase G6:*

F 13.1a Record status information about the sending node, j: $< j, timestamp_j >$ only.

13.2a If the status for node j is unchanged, mark it as a commitment, else the commitment must be restarted and next phase is set back to G5 again.

13.3 If all nodes has committed, send a TERMINATE, and set the next phase to G7.

A 13

Algorithm 14 (Terminate Handler)
The TERMINATE *signal from node j received on node i, during phase G5 or G6:*

G 14.1 Next phase is set to G7, as another node has detected that this mark-phase is globally completed.

A 14

Appendix C

Implementation Statistics

The garbage collection scheme has been implemented in the Emerald prototype by additions to the Emerald run-time system, i.e., the Emerald kernel only. The current prototype may run on either of the architectures: VAX, SPARC, and Motorola 68000 under various versions of the UNIX operating system. The VAX implementation has been tested with ULTRIX 4.2 and Mt.XINU 4.3BSD UNIX. Under SUNOS 4.1.1. (also 4.3BSD compatible) it may run at SUN-3 workstations, SUN-4 servers, and SPARC-stations. Furthermore, additions to let it run under HP-UX (Version 8) at M68000-based HP workstations are also available.

The current version does not facilitate heterogeneous[1] execution of Emerald programs, i.e., each Emerald system consists of a set of homogeneous computers connected by a thin and/or thick Ethernet segmented by bridges.

[1] Another version of the Emerald system has been developed for heterogeneous computing; but the garbage collector has not been tested on this version yet.

Kernel size (in lines of source code) Kernel Modules	Kernel without GC	GC changes and additions	Kernel inclusive GC
Message Module (MM, LM, and GCM)	5,904	1,265	7,146
Dynamic Code Load	3,748	413	4,044
Language Interface (stubs)	994	0	994
Process Management (in- and external)	4,789	2,041	6,607
Support for Mobile Object and Remote Access	9,420	1,762	9,953
Storage Management (inclusive GC)	1,392	6,323	7,248
Host Tables (HOTS)	1,582	237	1,804
Signals and User i/o	1,584	7	1,586
Checkpoint and Recovery Module	1,216	55	1,234
Various	8,544	1,348	9,755
Kernel Measurement and Debugging (KMD)	3,407	296	3,589
Total	42,580	13,747	53,960

Table C.1: Code Statistics for the added garbage collection in the Emerald kernel

The extension to the Emerald kernel are quantified in Table C.1 as lines of source code mostly written in C. As shown, the source code has grown from 42,000 lines to more than 53,000 lines by the addition and change of 14,000 lines. 25% of the new kernel is either new code or modifications of the old code.

The density of "real code" in the implementation is fairly low as more debugging and monitoring code has been included following the existing style of the Emerald kernel. Such additional code has been added all over the kernel during debugging, and the KMD system itself has also been extended. These debugging aids remain in the final version of the kernel source files, but the influence on run-time measurements has been limited by special compilation and compaction.

The primary additions are the garbage collector modules inserted in the modules for Storage Management and Message handling. Process Management now includes the management of the internal kernel processes for garbage collection processes, also. The changes in the Mobile Object Support modules are concerned with registering inter-node references, location of objects, remote invocations, and the garbage collection fault handler.

Appendix D

Danish Summary

Spildopsamling i distribuerede systemer

Et dansk resume af licentiatafhandling indleveret til forsvar ved Københavns Universitet af Niels Christian Juul

Licentiatarbejdet taget sit udgangspunkt i behovet for automatisk at kunne genanvende dynamisk allokeret lager i distribuerede systemer. Automatisk genanvendelse er muligt ved at spildopsamle ikke-frigivne lagerstykker, når disse kan identificeres som spild. En spildopsamler har således til opgave at identificere hvilke lagerstykker, der er spild, og indsamle dem. Dette arbejde fokuserer på identifikation af spild, hvor spild—kort fortalt—defineres som de allokerede lagerstykker ingen kan anvende, fordi ingen kender dem. I distribuerede systemer kompliceres identifikationsprocesses af at relationen "kender til" kan krydse frem og tilbage over grænserne mellem de knuder (datamater), som udgør systemet.

Den udviklede spildopsamlingsmetode for distribuerede systemer er karakteriseret ved:

Fuldstændighed idet alt spild opsamles.

Parallellitet idet spildopsamlingen udføres parallelt med afviklingen af andre processer.

Robusthed idet spildopsamlingen foretages selvom alle knuder i det distribuerede system ikke er tilgængelige samtidigt.

Spildopsamlingen er implementeret i køretidssystemet for det distribuerede, objekt-baserede programmeringssprog, Emerald.

Baggrund

Behovet for spildopsamling af arbejdslager i datamatsystemer er en konsekvens af behovet for dynamisk lagerallokering (hob-allokering), hvor levetiden af det allokerede ikke er bundet til levetiden for initiativtageren til allokeringen. Problemstillingen er f.eks. aktuel i objekt-orienterede systemer med persistente objekter. Emerald er et sådant distribueret, objekt-orienteret system, bestående af en oversætter og et køretidssystem, som afvikles på arbejdsstationer (knuder) forbundet i et lokalnet.

Spildopsamling efter metoderne *reference counting*, *mark-and-sweep* og *stop-and-copy* har været kendt siden begyndelsen af 1960'erne, hvor de blev introduceret i forbindelse med behandlingen af dynamiske datastrukturer som lister. Med udbredelsen af blok-orienterede programmeringssprog (og stak-allokering) gennem 1970- og 1980'erne svandt behovet for hob-allokering og spildopsamling. Den objekt-orienterede bølge har dog igen sat fokus på spildopsamling, hvor objekternes levetid er uafhængig af levetiden for den som opretter dem. På grund af de omkostninger, som anvendelsen af de første spildopsamlere introducerede, har spildopsamling haft et dårligt omdømme. Der findes imidlertid idag metoder, som nedbringer den generende ventetid, hvor spildopsamleren låser systemet, til et minimum.

Den udviklede spildopsamlingsmetode

Den udviklede spildopsamlingsmetoed er baseret på to *mark-and-sweep* spildopsamlere på hver knude i det distribuerede system. De enkelte spildopsamlere afvikles parallelt med brugerprocesser ved hjælp af en objektbeskyttelsesmekanisme med en effekt svarende til *page-fault* i side-opdelte virtuelle lagersystemer. Objektbeskyttelsen anvendes på objekter, som endnu ikke er mærket under mærkningsfasen. Selve indsamlingsfasen er gjort uafhængig af både brugerprocesser og mærkningsfasen ved at være indlejret i lagerallokeringsoperationerne.

For at kunne identificere alt spild samarbejder den ene spildopsamler fra hver knude i en global spildopsamling. Ved hjælp af decentral styring og en distribueret termineringsdetektion opnår den globale spildopsamler at blive robust overfor temporære fejl, som knuder der går ned og kommer op igen. Knuderne behøver kun være parvis tilgængelige for udveksling af informationer, omend metoden risikerer at være meget længe om at afslutte identifikationen af spild. Spild må imidlertid identificeres og returneres til lageradministratoren i et tempo, der svarer til allokering af lager til applikationerne. Derfor udfører det andet sæt af spildopsamlere hver for sig en uafhængig indsamling af lokalt spild på deres respektive knuder.

Spildopsamlingen er implementeret i Emerald systemet som en lokal og en global mærkningsprocess (Marker), samt en fælles opsamlingsprocess (Sweeper) suppleret med rutiner til håndtering af kommunikation og synkronisering mellem disse og resten af køretidssystemet. Hver mærkningsprocess starter med sine rødder (referencer til lagerstykker, som ihvertfald ikke er spild) og gennemløber alle lagerstykker, den kan nå direkte og indirekte via referencer, hvorunder disse mærkes. Indsamlingsprocessen gennemløber hele lageret og noterer sig de ikke mærkede lagerstykker som spild, der registreres til senere genbrug.

Opnåede resultater

Ved at implementere den designede spildopsamlingsmetode er det lykkedes samtidigt at opnåen *fuldstændig, parallel og robust spildopsamling i det distribuerede, objekt-baserede system, Emerald.* Alt spild i det distribuerede system, inklusiv cirkulære, selvrefererende strukturer af spild opsamles parallelt med at systemet fortsat afvikler brugerprogrammer. De pauser, som spildopsamlingens enkelte deltrin introducere i den normale programafvikling, kan normalt begrænses til samme størrelsesorden som den tid hver Emerald process tildeles som maximal kørselstid af gangen. Pauser på grund af spildopsamling er således ikke mere synlige for den enkelte process end de pauser systemet i forvejen introducerer for at flere processer kan afvikles "samtidigt".

Konklusion

Det er således eftervist, at det i praksis er muligt at foretage spildopsamling i distribuerede systemer mens de kører. Dette er opnået ved:

En fuldstændig og parallel spildopsamler i et distribueret system
Implementationen af den fuldstændige spildopsamler i Emerald fungerer parallelt med afviklingen af brugerprogrammer. Såvidt vides, er det den første fungerende implementation af en fuldstændig og parallel spilddetektor i distribuerede systemer.

En robust spildopsamler i distribuerede systemer
Implementationen fungerer selvom dele af det distribuerede system fejler. Den globale spildopsamler afventer simpelthen at utilgængelige knuder bliver tilgængelige igen. Hvorimod den lokale spildopsamler er istand til uafhængigt heraf at opsamle lokalt spild.

Faktisk er den anvendte protokol for kommunikation og synkronisering mellem knuderne i systemet så robust at den sikre såvel fremdrift (mærkningsfasen konvergerer mod sin afslutning) som detektion af at afslutningen er nået, selvom knuderne hver især fejler vilkårligt ofte. Parvis tilgængelighed af alle knuder mindst en gang under afslutningen er det hårdeste krav den globale spildopsamler stiller til det distribuerede system.

Udnyttelse af objektbeskyttelsesmekanismen
Ved at anvende ideen om objektbeskyttelse, som den i forvejen er brugt til at detektere hvornår et procedurekald ikke er lokalt, men skal omformes til et fjernprocedurekald, under mærkningsfasen opnås en klar skillelinie mellem de objekter brugerprocesserne anvender og de som endnu ikke er mærket af spildopsamleren. Herved beskyttes kendte, men endnu ikke gennemsete, objekter fra at blive forandret bag ryggen af spildopsamleren.

Vitae

Niels Christian Juul was born in København (Copenhagen, Denmark) on the 3rd of June, 1955. After 9 years of primary school at Østersøgades Gymnasium, Kildegaard Gymnasium, and N.Zahles Seminarieskole, and three years at high school he graduated from high school in 1974 with a Danish *studentereksamen, matematisk-fysisk gren* from Gladsaxe Gymnasium. He was enrolled as student (stud.scient.) at the University of Copenhagen from 1974 to 1988. During which he earned a M.Sc. in Computer Science. Since graduation in 1988 he has been employed at DIKU, Department of Computer Science, University of Copenhagen on various grants. He started his Ph.D. Program in 1989 and is now submitting the Thesis to earn the Ph.D. degree. Since January 1993 he holds a PostDoc at University of California, Riverside.

Niels Christian Juul has been employed in work related to computer science since 1979. He has been at the Regnecentralen A/S as programmer in the production of communication software and as analyst in the development department working on a test operating system for hardware test and debugging in high level systems programming language. He has also been with the Danish Road Data Lab. working as programmer on graphical packages. He left the industry in 1981 and worked as student instructor for five years on first and second year undergraduate courses. In 1986 he switched to be coordinating teacher of the second year undergraduate courses.

Niels Christian Juul is married to Anne Nordly, with whom he has two girls, Pernille (14th of February, 1982) and Mie (31st of September, 1984).